# PARALLELISED AND VECTORISED ANT COLONY OPTIMIZATION

J PEAKE

PhD 2021

# PARALLELISED AND VECTORISED ANT COLONY OPTIMIZATION

JOSHUA PEAKE

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS OF
MANCHESTER METROPOLITAN UNIVERSITY
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Department of Computing and Mathematics
Manchester Metropolitan University

2021

# Contents

# List of Tables

# List of Figures

# Abstract

Ant Colony Optimisation (ACO) is a versatile population-based optimisation meta-heuristic based on the foraging behaviour of certain species of ant, and is part of the *Evolutionary Computation* family of algorithms. While ACO generally provides good quality solutions to the problems it is applied to, two key limitations prevent it from being truly viable on large-scale problems: A high memory requirement that grows quadratically with instance size, and high execution time. This thesis presents a parallelised and vectorised implementation of ACO using *OpenMP* and *AVX* SIMD instructions; while this alone is enough to improve upon the execution time of the algorithm, this implementation also features an alternative memory structure and a novel *candidate set* approach, the use of which significantly reduces the memory requirement of ACO. This parallelism is enabled through the use of *Max-Min Ant System*, an ACO variant that only utilises local memory during the solution process and therefore risks no synchronisation issues, and an adaptation of *vRoulette*, a vector-compatible variant of the common *roulette wheel* selection method. Through the use of these techniques ACO is also able to find good quality solutions for the very large *Art TSPs*, a problem set that has traditionally been unfeasible to solve with ACO due to high memory requirements and execution time. These techniques can also benefit ACO when it comes to solving other problems. In this case the *Virtual Machine Placement* problem, in which Virtual Machines have to be efficiently allocated to Physical Machines in a cloud environment, is used as a benchmark, with significant improvements to execution time.

# Declaration

No part of this project has been submitted in support of an application for any other degree or qualification at this or any other institute of learning. Apart from those parts of the project containing citations to the work of others, this project is my own unaided work. This work has been carried out in accordance with the Manchester Metropolitan University research ethics procedures, and has received ethical approval number *SE171844C* .

Signed:

Date:

# Acknowledgements

Firstly, I would like to sincerely thank my Principal Supervisor Dr. Huw Lloyd for the extensive amount of assistance provided throughout the entire PhD, and for always being there to answer any questions I might have.

I would also like to thank Prof. Martyn Amos, who inspired my initial interest and passion for research and without whom I probably would not even be doing this PhD, as well as Dr. Paraskevas Yiapanis, Prof. Rene Doursat, Dr. Nicholas Costen and Dr. Giovanni Masala for the extremely valuable assistance and feedback provided during this process.

My thanks also go to Antoine Gaget and Jamie Webster, who went through their PhD journeys alongside me and were always there for mutual discussion, feedback and support.

I would also like to thank my partner, Jade, who has done a fairly decent job of keeping me sane throughout this process, and will hopefully continue to do so in the future.

My final and probably most important thanks go to Juno, my cat, who has stepped on my keyboard frequently throughout the writing of this thesis and therefore probably deserves a co-author credit.

# Chapter 1

# Introduction

## 1.1 Background and Motivation

Nature has always been an inspiration for humankind, not just in art and literature, but also in engineering (Helms, Vattam, and Goel 2009). Artificial Intelligence is one such field that takes inspiration from nature, not just by trying to replicating human behaviour as Turing initially focused on when he introduced the Turing test (Turing 1950), but by replicating the behaviour seen by plants and animals. Thousands and even millions of years of evolution have contributed to the behaviour of the species in the world today, with some of them exhibiting behaviour that is optimised for a specific activity. This optimisation is especially clear in swarming creatures, who are able to thoroughly forage for food in an efficient and organised manner despite the limited intelligence of each individual swarm member. One such example of this is bee hives (Tereshko and Loengarov 2005), in which individual bees are are "foragers", either employed (currently exploiting a found food source) or unemployed (either searching for a new source to exploit, or waiting at the hive for information from employed foragers). In order to communicate information to other bees, a "waggle dance" is performed by employed foragers, informing unemployed foragers of the food source. Another example is that of the ant, which has developed the ability to communicate through the use of pheromones (Wilson 1962). Through pheromones, ants are able to leave trails for other members of the ant colony to allow them to quickly and efficiently travel between the nest and a food source. This is achieved through an initial exploratory search in which the ant colony distributes itself throughout a wide area, with ants that find food returning to the nest while leaving behind a pheromone trail that will lead other ants to that same food source. This exploratory behaviour in which ants initially search for an optimum route inspired an optimisation algorithm, Ant Colony Optimisation (ACO) (Dorigo, Maniezzo, and Colorni 1991).

Naturally, ACO was initially applied to the Traveling Salesman Problem (TSP), a route optimisation problem focusing on finding the minimal possible distance to tour every city in a given set of cities. While optimising the TSP resembles the route-finding behaviour that ACO was initially based on, the algorithm was soon proven to be applicable to a much wider range of problems. Over the years ACO became a staple technique used for solving NP-hard problems, ranging from other route optimisation techniques such as the Vehicle Routing Problem (Bell and McMullen 2004) to feature selection (Ghosh et al. 2019) and cost estimation (Zhang et al. 2020).

While ACO is able to find good quality to solutions to a vast array of problems, the length of time it takes to reach a good quality solution can be a prohibitive factor, especially for large problems. This thesis focuses on utilising parallel hardware and readily-available vectorisation instructions to optimise the execution time of the ACO algorithm, attempting to improve execution time while retaining the technique's ability to find good quality solutions to problems.

The motivation to optimise the execution time of the algorithm is not merely a case of doing it to prove that a faster execution is possible. When it comes to real-world problems, low execution time can be vital for several reasons: Reactivity is vital in certain problem domains, such as route planning, where a change in conditions can lead to a route needing to be calculated as quickly as possible; Lowering runtime also increases energy efficiency for demanding computational tasks, and in the case of multithreading it is more efficient to have higher utilisation on multiple threads for a shorter period of time than lower utilisation on a single thread for a longer period of time (Li, Brooks, et al. 2004).

## 1.2 Aims and Objectives

The aim of this thesis is to increase the efficiency of Ant Colony Optimisation, primarily through parallelisation and vectorisation, but also through the use of other techniques. Parallelisation allows for the simultaneous processing of data, or in this case, "ants", which can lead to substantial savings in execution time providing that synchronisation issues are avoided. Vectorisation allows for the use of Single Instruction Multiple Data (SIMD) instructions to perform operations on multiple items of data simultaneously, which again can lead to substantial savings in execution time. The traveling salesman problem is traditionally used for investigating new developments to the core ACO algorithm, and so that will be the initial benchmark problem. However, in order to demonstrate the general applicability of the techniques developed using TSP, another benchmark problem will be chosen, a problem with a real-world focus. Comparisons will be made to other algorithms, both alternative ACO implementations and other optimisation algorithms. As other techniques exist to parallelise and vectorise ACO, the first step will be to try and improve upon these implementations.

- Aim 1: Develop an implementation of ACO that utilises the AVX-512 instruction set and the hardware of the Intel Xeon Phi. This instruction set has never been applied to ACO before, so using it in this implementation will be a proof of the viability of the instruction set.

- Aim 2: Develop a novel nearest neighbour list structure to take advantage of the available vectorisation potential. While candidate sets have seen extensive use in previous ACO iterations, none have developed an implementation that makes full use of vector instructions.

- Aim 3: Develop a technique to reduce the memory complexity of ACO. One of the main obstacles when it comes to solving large problems with ACO is the significant memory requirements of the pheromone matrix structure, so overcoming this obstacle would allow ACO to scale to much larger problems than were previously possible.

- Aim 4: Apply a parallelised and vectorised ACO implementation to a real-world focused problem in which solution time is crucial. Previous work on vectorised ACO has largely been restricted to TSP, so implementing a vector-enabled ACO algorithm on a real-world problem would demonstrate the general applicability of the vectorised approach.

Each of these aims contributes to the overall objective of this thesis, which is the development of an ACO implementation that makes full use of parallelisation and vectorisation, as well as other techniques that take advantage of the presence of these technologies, and demonstrate the benefit of these technologies on multiple problem domains.

## 1.3 Contributions

- Contribution 1: The development of an ACO implementation that makes use of the AVX-512 instruction set in order to make use of 16-wide vectors of single-precision floating point values.
- Contribution 2: A novel candidate set structure allowing for a more efficient selection phase, designed for effective vectorisation.
- Contribution 3: A novel memory structure allowing ACO to solve very large Traveling Salesman Problem instances effectively, with the core ACO algorithm remaining intact.
- Contribution 4: Demonstrated the general applicability of the vectorisation techniques developed in Contribution 1 by porting them to be AVX2-compatible, a more widely-available vector instruction set.
- Contribution 5: The development of the first parallelised and vectorised implementation of ACO for the Virtual Machine Placement problem, in order to demonstrate that the benefits of this approach can be applied to problem domains other than the Traveling Salesman Problem.

The contributions above are described in the following publications:

- Peake, J., Amos, M., Yiapanis, P. and Lloyd, H., 2018, July. Vectorized Candidate Set Selection for Parallel Ant Colony Optimization. In Proceedings of the Genetic and Evolutionary Computation Conference Companion (pp. 1300-1306).
- Peake, J., Amos, M., Yiapanis, P. and Lloyd, H., 2019, July. Scaling Techniques for Parallel Ant Colony Optimization on Large Problem Instances. In Proceedings of the Genetic and Evolutionary Computation Conference (pp. 47-54).
- Peake J., Amos M., Costen N., Masala G., Lloyd H., 2021. PACO-VMP: Parallel Ant Colony Optimization for Virtual Machine Placement. In Submission.

I contributed the majority of the work for each of these papers, with the other authors acting in a supervisory capacity.

## 1.4 Thesis Structure

Chapter 2 provides a comprehensive literature review of the technologies utilised in order to fulfil the aims of this thesis, focusing on Ant Colony Optimisation and looking at the origins of the metaheuristic, different variants of the technique, the history of parallelising ACO and different problem domains that the technique has been applied to.

Chapter 3 discusses Ant Colony Optimisation in-depth, describing each phase of the algorithm as well as discussing the different approaches taken by each ACO variant.

Chapter 4 introduces an AVX-512 enabled ACO implementation, as well as describing the novel Vectorised Candidate Set Selection technique which makes use of a novel Candidate Set structure and a vectorised selection method to make best use of the new structure. This is compared to the previous best-performing vectorised ACO implementation, as well as benchmarked against a standard sequential implementation.

Chapter 5 discusses the novel Restricted Pheromone Matrix data structure, a novel implementation of ACO's core data structure, the pheromone matrix. An ACO implementation utilising this data structure is benchmarked on a well-known set of very large Traveling Salesman Problem instances, the Art TSPs. Compared to the implementation described in Chapter 4, local search is also utilised in order to improve solution quality. This is compared with the only previous ACO implementations that have been able to solve these very large instances.

Chapter 6 describes a parallelised and vectorised ACO implementation designed to solve the Virtual Machine Placement (VMP) problem, a problem with real-world applications. This is compared with the previous best performing ACO solver for VMP, as well as a recently developed Genetic Algorithm implementation.

Chapter 7 concludes the thesis with a general discussion of the results obtained in the previous chapters, as well as discussing ideas for further work and specifically discussing work that has since been published that iterates upon the ideas presented in Chapter 5.

# Chapter 2

# Literature Review

## 2.1 Evolutionary Computation & Swarm Intelligence

Evolutionary Computation is a wide-ranging and diverse family of optimisation algorithms largely consisting of nature-inspired trial-and-error based approaches. A general structure for Evolutionary Computation algorithms consists of a population of problem solving *agents*, which attempt to solve a given problem over a number of *generations*, with each generation consisting of a number of attempts to solve that problem. Once a generation has finished solving the problem, the population of solutions will go through a *selection* process and/or *mutation*, depending on the given algorithm. Over time, this leads to an *evolution* of the population, which will gradually increase the *fitness* of the solutions found, with the definition of fitness varying depending on the given problem. Over time, the solutions will improve and the end solution of the algorithm will ideally be superior to the solutions found in the earlier generations of the algorithm.

Evolutionary Computation was initially proposed in the late 1950s. Friedberg (1958) proposes a program called "Herman" containing 64 instructions, each of which is a 14 bit number. These instructions all modify an array of 64 bits, with the state of the array at the end of the program's execution being the end result of the program. An outside agent, the "Teacher", inspects the output data and determines whether the program has been a success or a failure based on the criteria of a given "problem". The first problem used in this experiment, simply known as Problem 1, had a single bit of "input data", the first bit of the 64 bit array, and a single bit of "output data" which was the final bit of the array. If the input and output bits match at the end of the program, the solution is defined as a success, otherwise it is defined as a failure. The teacher randomly generates the input bit, either 0 or 1. For each of the 64 instructions, two instructions actually exist, and at any one time one will be active and one will be inactive. If an instruction is active during a successful run, this is recorded, so each instruction has an attributed number of successes. A second outside agent, the "learner", regularly switches between the active and inactive instructions, as well as occasionally randomly replacing the instruction entirely with a new instruction. These switches are governed by the success number of each instruction. Over time, the instructions with a high number of successes will be switched out less often, until a variation of the program is present that successfuly solves the problem in a high percentage of attempts. In practice, every 10,000 runs of the Herman program are considered as a "block", and by the 16th block of runs Problem 1 is successfully solved 100% of the time. Problems with more difficult success criteria were also tested, with a perfect program being found for 2 of them, while 3 were unable to be perfectly solved. While the results of the experiments were mixed, enough were successful to prove that evolutionary computation is able to solve some problems by gradually evolving an algorithm.

The foundation of modern Evolutionary Computation was laid by three separate techniques: Evolutionary Programming (Fogel, Owens, and Walsh 1966), Genetic Algorithms (Holland 1975), and Evolution Strategies (Bremermann et al. 1962). Initially developed independently, these techniques were later unified as seperate branches of Evolutionary Computation. Further Evolutionary Computation techniques have been developed over time, such as Artificial Immune System (Farmer, Packard, and Perelson 1986), Memetic Algorithms (Moscato et al. 1989) and Cultural Algorithms (Reynolds 1994) among many others.

While differing in the details, most Evolutionary Computation techniques retain the same base structure. The techniques utilise a population of individuals, which either represent a solution or a problem solving agent. These individuals are then subjected to probabilistic operations such as selection, mutation and/or recombination (crossover). These operations are applied in a loop, with each iteration of the loop being known as a generation. A fitness evaluation process will determine the quality of each solution based on one or several parameters that the algorithm aims to optimise. The decisions made based on this fitness value vary depending on the approach used in a given algorithm: An elitist approach may only retain or make use of the solution(s) with the highest fitness value, while a fitness proportionate approach will still have the ability to retain solutions with lower fitness, albeit with a lower chance of retention, usually for the purposes of maintaining population diversity and preventing premature convergence through the perturbation of existing solutions (an example of this is an algorithm that uses mutation and recombination operations, such as a Genetic Algorithm).

The differences between the various Evolutionary Computation techniques are largely down to differences in the operations. For example, Genetic Algorithms are far more focused on the crossover phase, in which the "genes" of two "parent" solutions are exchanged up to a certain crossover point, with the offspring of this crossover being added to the population of solutions and some of these offspring being subject to random gene mutations. Evolutionary Programming, on the other hand, does not utilise a crossover operation, instead focusing on mutation. The encoding of genes as bits and chromosomes as sequences of bits is also unique to Genetic Algorithms. Evolutionary Programming and Evolution Techniques are more similar, both utilising self-adaptation and real-valued representation of search points. However, Evolutionary Programming uses a probabilistic selection mechanism while Evolution Strategies uses deterministic selection.

These techniques have all been applied to combinatorial optimisation problems, namely the Traveling Salesman Problem (TSP). Genetic Algorithms initially performed poorly on TSP, though performed well on other combinatorial problems such as Assembly Line Balancing (Anderson and Ferris 1994). Over time, Genetic Algorithms have been adapted to perform well on TSP problems through the use of local search techniques (Braun 1990), adaptations to the crossover methodology such as Sequential Constructive Crossover (Ahmed 2010) and Edge-Assembly Crossover (Honda, Nagata, and Ono 2013), and changes to the quantity of offspring (Wang, Ersoy, et al. 2016), with the Edge-Assembly Crossover approach providing the best known solutions to two of the very large Art TSP instances.

One particularly significant family of Evolutionary Computation sub-techniques is *Swarm Intelligence*. While Evolutionary Computation takes inspiration directly from evolution, Swarm Intelligence instead focuses on the problem-solving and collective intelligence exhibited by swarms or colonies of social organisms. Ant Colony Optimisation (ACO), the focus of this thesis, was one of the first iterations of Swarm Intelligence, and will be covered in future sections of the Literature Review. Another significant Swarm Intelligence algorithm is Particle Swarm Optimisation (PSO) (Kennedy and Eberhart 1995). PSO takes inspiration from bird flocks, and how they are able to identify the locations of food in a wide area despite none of the flock having any prior knowledge of the location of food sources. In PSO, the individual solutions are referred to as “particles”, with the population of solutions referred to as the “swarm”. The system is initialised with a population of random solutions, much like Genetic Algorithms and Evolutionary Programming. Each particle is also assigned a “velocity”, which determines the speed at which the particle navigates the search space. The behaviour of these particles is influenced by two values, *pbest* - the highest fitness value achieved by the individual particle - and *gbest* - the highest fitness value achieved by any particle in the swarm. The particle also records the search space co-ordinates associated with high fitness values. For each step of the PSO algorithm, the velocity of a particle is altered to accelerate it towards the *gbest* and *pbest* values. Both of these values have a separate acceleration value, which is weighted with a random number. The velocity of a particle in both directions is clamped to a maximum value, *Vmax*. The value of *Vmax* is crucial to PSO as an unsuitable *Vmax* value could lead to particles moving too quickly past good solutions, or moving too slowly to fully explore beyond locally good regions, trapping them in local optima. The addition of the inertia weight value in later iterations of PSO (Shi and Eberhart 1998) eliminated the problem of finding a good *Vmax* value, as it allowed *Vmax* to be set at the dynamic range for each variable and reliably perform well, with the inertia weight value *w* becoming the value that needs to be optimised (though it was quickly determined that relatively high *w* values perform well).

While this section has described the original PSO algorithm and early adjustments, PSO is now a hugely diverse subject area with many variants such as Binary PSO (Kennedy and Eberhart 1997), Dynamic PSO (Eberhart and Shi 2001), and a large number of hybrid variants (Robinson, Sinton, and Rahmat-Samii 2002; Hendtlass 2001; Zhao et al. 2005; Shi, Li, et al. 2010). PSO has been used to solve a large number of optimisation problems, a selection of which are Antenna Design (Donelli et al. 2006; Boeringer and Werner 2004; Khodier and Christodoulou 2005), Biomedical(Selvan et al. 2006), Solar Energy (Elsheikh and Abd Elaziz 2019), Geotechnical Engineering (Hajihassani, Armaghani, and Kalatehjari 2018) and Traveling Salesman Problem (Wang, Huang, et al. 2003).

Another significant Swarm Intelligence technique is Artificial Bee Colony (ABC) (Karaboga 2005). As the name suggests, the algorithm is inspired by the previously discussed behaviour of honey bee swarms, specifically the food foraging process. The swarm consists of three essential components: Food sources, employed foragers and unemployed foragers. Employed foragers are associated with a particular food source they are “employed” at, while unemployed foragers are either scouting for new food sources, or waiting in the nest for information from employed foragers. In order to communicate, bees perform a “waggle dance” which communicates information about food sources to other onlooking bees. The waggle dance can cause unemployed foragers to be “recruited” to a certain food source, becoming “employed” by that food source. The employed bee will then take nectar from the food source, return to the hive and unload the nectar to a food store. The bee will then either return to unemployment, perform the waggle dance to recruit other bees to the food source, or return to forage without recruiting other bees.

In the ABC algorithm, the artificial bee colony consists of three groups: Employed bees, onlookers and scouts. Each “food source”, representing a possible solution, employs a single bee. The amount of “nectar” in each food source corresponds to the solution quality of the solution represented by that food source. The employed bee moves to other food sources in the neighbourhood of its current food source, and determines the nectar level. If the nectar level is higher than their previous food source, they forget their previous source and instead become employed by the new, better food source. A limit parameter is used to determine how many times a bee can fail to find a better food source before their current source becomes abandoned. Once this happens, employed bees become scouts. The onlooker bees are placed on food sources in the vicinity of the food source they are being recruited to using roulette-wheel selection (Goldberg 1989). The probability is influenced by the amount of nectar in the food source of the bee that is recruiting them. The scout bees explore the rest of the search space looking for food sources, generally finding low-quality solutions but occasionally discovering a new high quality solution. Throughout this process, the global best solution is recorded and updated when a new global best is found. Once the algorithm reaches its termination state, this global best solution is returned.

As with PSO, ABC now has many variants and hybrids to solve a vast array of problems. Applications of ABC include training neural networks (Karaboga and Akay 2007; Karaboga, Akay, and Ozturk 2007; Karaboga and Ozturk 2009), nuclear power plant accident diagnosis (Oliveira, Schirru, and Medeiros 2009), subway route optimisation (Yao et al. 2010), traveling salesman problem (Karaboga and Gorkemli 2011; Li, Li, et al. 2011) and vehicle routing (Szeto, Wu, and Ho 2011). Notably, many of these problems are combinatorial rather than the numerical problems that ABC was initially designed to solve.

## 2.2 Ant System

Rather than being influenced by every aspect of a species' behaviour, nature-inspired algorithms generally take inspiration from one or a small number of specific behaviours exhibited by that species. For ants, this is the act of pheromone distribution, which was first observed in the late 1940s and 1950s. Wilson (1962) observed the foraging behaviour of Fire Ants, in which solitary workers move away from the ant nest in irregular, looping paths. Once an ant finds a food source that it is unable to move by itself, it inspects the source and then returns to the nest, leaving a chemical trail as it travels. Once the ant reaches the perimeter of the nest and makes contact with other ants, it turns around and follows the trail back to the food source, doubling the strength of the trail. Most other worker ants that encounter the trail are drawn to follow it in an outward direction, towards the food source. This is an example of *stigmergy*, indirect co-ordination between agents through the environment (Grassé 1986).

This was proven with the use of the double bridge experiment, as seen in Figure 2.1, which presented ants with two paths of differing lengths. Over time, the ants stopped using the longer path until they were solely using the short path. The shorter path allows for a shorter journey, and an ant returning to the hive more quickly means a stronger pheromone trail, as the ant reinforces the trail while returning to the hive. This stronger pheromone trail attracts more ants, until the trail is so strong that ants no longer consider the trail on the longer path.

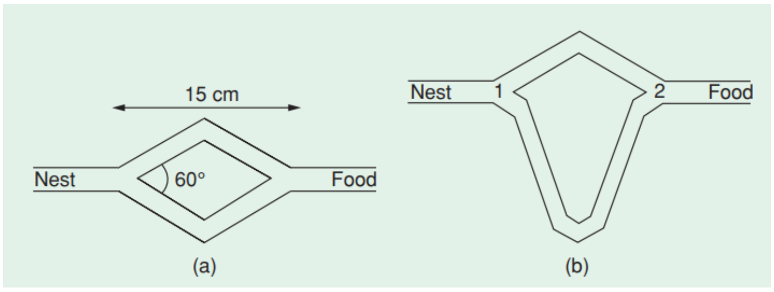


Figure 2.1: The double bridge experiment, designed to determine which path an ant would take given the choice of two paths with differing lengths (Dorigo and Birattari 2011)

An optimisation technique based on this behaviour was first proposed as *Ant Algorithms* (Colorni, Dorigo, Maniezzo, et al. 1991), and later expanded upon as *Ant System* (Dorigo and Gambardella 1997a). The technique makes use of software agents (or "Ants") that each attempt to find a solution to a given problem. The problem used to demonstrate these algorithms is the Traveling Salesman Problem (TSP). The TSP (Lawler 1985) is an NP-hard problem presented in the form of a set of cities, with the end goal being to create the shortest Hamiltonian cycle that visits every city. While many variants exist, the two most prominently featured TSP variants in ACO research are *symmetric* TSP, the default form of TSP in which the distance between two cities is the same in both directions, and *asymmetric* TSP (ATSP) in which the distance between two cities is different in both directions. The basic structure of the *Ant System* is as follows:

- An ant's starting city is selected at random
- The ant selects which city to move to next with a probability based on distance between the target city and the current city, the pheromone value for the edge between the two cities (which will initially be the same for every path, and will change over time), and a randomly generated number.
- Ants are unable to visit the same city twice through use of a tabu list
- Once an ant has completed its tour, it updates the pheromone value for the edges traversed in its tour, with the size of the pheromone value increase being indirectly proportional to the quality of the solution (length of the tour, with shorter tours being considered better solutions).

The AS algorithm is divided into iterations, with each iteration containing a certain amount of ants. Once every ant in an iteration completes its tour, *pheromone evaporation* takes place which reduces pheromone globally by a pre-defined amount. As more ants perform tours, the pheromone values for paths that lead to high-quality solutions increase, making it more likely that ants will travel on these edges and produce their own high-quality solutions. AS algorithms can continue until either convergence is reached, or a pre-defined number of iterations have been executed. The shortest tour at this point is considered the best solution. This basic structure persists through all future iterations and variations of the *Ant System* algorithm. (Colorni, Dorigo, Maniezzo, et al. 1991; Dorigo, Maniezzo, and Colorni 1996; Colorni, Dorigo, Maniezzo, and Trubian 1994; Gambardella and Dorigo 1995; Maniezzo, Muzio, et al. 1994)

## 2.3 The Ant Colony Optimisation metaheuristic

The development of multiple variants of the original *Ant System* algorithm prompted the definition of the *Ant Colony Optimisation* meta-heuristic in (Dorigo and Di Caro 1999), which establishes a common framework shared by these techniques and allows for a unified view of the ACO research area. This framework contains characteristics shared by each different ACO technique including ant behaviour, suitable problem characteristics and the three phases of ACO algorithms: tour construction, pheromone distribution and daemon actions. Variants of ACO used with TSP and not previously mentioned are: *Elitist Ant System* (Dorigo, Maniezzo, and Colorni 1991), a precursor to $\mathcal{MM}$AS which only distributes pheromone for the best-so-far tour; ANTQ (Gambardella and Dorigo 1995), a version of *Ant System* that takes inspiration from *Q-Learning* (Watkins and Dayan 1992); *Rank-based AS* (Bullnheimer, Hartl, and Strauss 1997), which ranks ants based on tour lengths and only distributes pheromone to the top *n* ants; and *Best-Worst Ant System* (Cordón García, Fernández de Viana, and Herrera Triguero 2002), which implements a pheromone mutation phase in the AS algorithm. Additionally, while TSP was the problem used for initial benchmarking, ACO has been successfully applied to many other problems domains including *Vehicle Routing* (Gambardella, Taillard, and Agazzi 1999), *Quadratic Assignment* (Stützle and Hoos 2000), *Multiple Knapsack* (Leguizamon and Michalewicz 1999) and *Protein Folding* (Shmygelska and Hoos 2005).

The next variant of the AS algorithm is *Ant Colony System*(ACS) (Dorigo and Gambardella 1997a), which has three key differences compared to the original AS. Firstly, while *Ant System* updates the pheromone trails of every ant in the global *pheromone update* process, ACS only updates the path of the ant that found the best tour in the current and every previous iteration (the *global-best* ant). Alongside this change, a local *pheromone update* process is introduced which is carried out by ants, updating pheromone levels as they move around the TSP. The third and final main distinction between ACS and AS is a change to the state transition rule (the rule that determines which city an ant will visit next). The change introduces the idea of exploration vs exploitation, with exploration focusing on visiting cities not currently in the best tour and exploitation making use of the current best tour to select the next city. This is chosen randomly each time an ant is selecting the next city to visit, weighted by the value of the parameter *q0* which is a number between 0 and 1 which indicates the probability of exploitation. Additionally, ACS is the first variant of ACO to make use of *local search*, using the *3-opt* technique to attempt to improve already completed tours.

A further improvement to *Ant System* was then proposed in (Stützle and Hoos 2000). $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS) introduces several new techniques into the *Ant System* algorithm. Like ACS, only the best ant distributes pheromone after tour completion. However, in order to counteract the potential for stagnation that comes with this approach, pheromone levels are bound between a maximum and minimum value. The implementation of the minimum bound prevents paths from having no pheromone, meaning that no path will ever be fully excluded from the tour. The maximum value prevents a particular edge of the graph from amassing so much pheromone that it will be traveled on whenever possible by an ant, as while this may lead to a good tour, better tours may require the edge to be ignored. The results of experimentation with $\mathcal{MM}$AS determined that it performed at least as well as ACS, and regularly outperformed the algorithm. A key difference between $\mathcal{MM}$AS and ACS is the local pheromone update, which does not take place in $\mathcal{MM}$AS. This allows $\mathcal{MM}$AS to be easily parallelised, which will be discussed further in later sections.

## 2.4 Parallel Ant Colony Optimisation

### 2.4.1 Distributed Systems

While ACS and $\mathcal{MM}$AS were being developed, the possibility of parallel ACO was also being looked into for the first time. This would allow the computationally expensive ACO algorithm to divide its workload, leading to reduced execution time which comes with the added effect of bringing the execution time of larger problems down to a feasible level. Two separate approaches to parallelised ACO were investigated (Figure 2.2). The first, which was described as a *fork-join* algorithm (Bullnheimer, Kotsis, and Strauß 1998a), involved having individual ants solving the TSP instance in parallel and then updating pheromone levels once every ant had completed its tour. This is possible due to the lack of communication between ants when they're carrying out a tour. Limited hardware availability at the time meant that in most cases there were more ants required than available workers, which required each worker to handle more than one ant, increasing the granularity of the application. This approach also still requires some communication between ants (sending tours back to a master process and receiving updated pheromone levels from said process), which decreases efficiency. To attempt to prevent this, a partially *asynchronous* (Bullnheimer, Kotsis, and Strauß 1998a) approach was also investigated. This approach introduces the idea of local updates, where processes (ants) within each worker update pheromone trails locally within the worker rather than globally, with global updates between all workers happening far less frequently. A drawback of this approach is that ants in different workers will only be made aware of a good quality tour in another worker once a global update happens, delaying the impact of the good tour. A fine balance between local and global updates is required for this technique to be effective. While the experiments using these approaches were simulated rather than tested on a problem, the results indicated that the *asynchronous* technique performed better than the *synchronous* technique, as it benefitted from reduced idle time due to the lower communication overhead. However, the benefits of the *asynchronous* approach are less prominent on larger problem sizes.

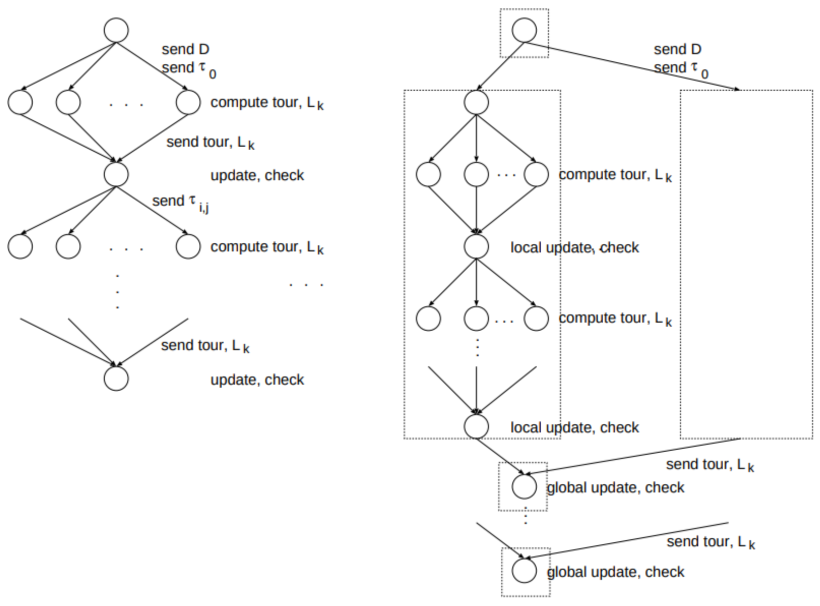


Figure 2.2: Diagram of the two parallelisation approaches used in (Bullnheimer, Kotsis, and Strauß 1998a), *synchronous* (or *fork-join*, L) and partially *asynchronous* (R). The arrows indicate the sending of tour information and/or pheromone information between the master processor and worker processers. The worker processors each compute a tour and send it back to the master processor, which updates the pheromone matrix and checks for the best tour found so far.

The approach of dividing ants between workers is also discussed in (Stützle 1998), which parallelises the $\mathcal{MM}$AS algorithm. This approach differs from Bullnheimer's approach in that there is no communication between workers; each worker executes ACO sequentially and the only interaction with other workers occurs when the best tour is being determined at the end of execution. Along with this, an entirely new approach is presented, focusing on speeding up individual runs using parallelisation rather than having many ants running in parallel, which allowed the contemporary hardware to more efficiently make use of parallelisation. This master-worker approach (Figure 2.3) involves using one processor (the master processor) to update the data structures, create solutions that will be improved by local search, and send the solutions on to other processors (worker processors) which then perform local search on the solutions. Once the solutions have been improved by the workers, they are returned to the master who updates the pheromone matrix appropriately. The structure of the master-worker system can vary depending on the problem instance it's being applied to. In situations where local search only takes 75% - 80% of processing time, some worker processors may instead be given the task of constructing new tours, which are then sent to the local search workers, which then send their improved tours back to the master processor. The more coarse-grained asynchronous approach performed better than the synchronous approach due to reduced idle time.

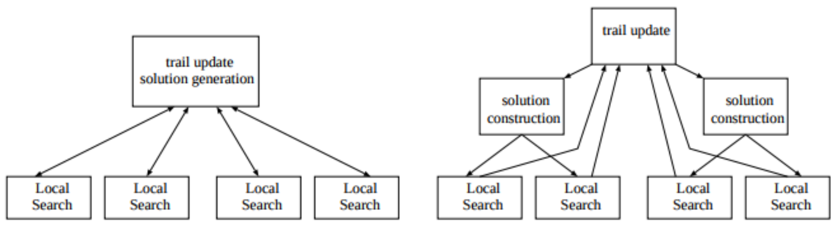


Figure 2.3: Diagram of the master-worker parallelisation approach (Stützle 1998)

Research on parallel ACO continued to focus on coarse-grained methods, which were performing better at the time than fine-grained techniques, and while fine-grained techniques were still being developed (Randall and Lewis 2002) communication overhead was still proving to be an issue. This was due to techniques largely being developed on distributed systems as this was the most viable method of parallelisation at the time. This would continue to be the case until performance gains would minimise the impact of this overhead, as well as developments in hardware allowing parallelisation to be carried out by a single machine. Coarse-grained algorithms continued to follow the structure of the master-worker approach, with new techniques focusing on optimising communication between independent worker ant colonies. Rather than communicating the pheromone matrix between colonies, these solutions opt to communicate single solutions between colonies as this significantly reduces communication time. In (Middendorf, Reischle, and Schmeck 2002), different methods of communication are tested to determine the most efficient way to communicate best tours between ant colonies. The best performing technique has two communication policies: *circular exchange* of local best solutions, where ant colonies are organised in a directed ring structure with each ant colony periodically sending its best tour to the next colony in the ring; and *migration*, which uses the same ring structure but compares a pre-defined number of best ants between a colony and the next colony in the ring, with the best ants in the comparison being used to update the pheromone matrix. This technique performs well on the TSP, as do several other implementations of multi-colony parallel ACO (Doerner et al. 2006) (Chen and Zhang 2005) (Twomey et al. 2010). While these techniques significantly reduce the communication overhead of parallel ACO, further developments were being made in the previously established technique of independent Ant Colonies (Stützle 1998), which has no communication at all and therefore no overhead. This technique is compared against several communicating multi-colony techniques in (Manfrin et al. 2006), and performs considerably better than all other techniques in terms of solution quality despite having no communication between ant colonies, although the author notes that this may not be the case with some kind of anti-stagnation mechanic implemented. However, this does demonstrate that communication between multiple colonies isn't necessary for an effective parallel implementation of ACO, though running individual ants in parallel was still a long way from being a viable technique at this point.

### 2.4.2 GPGPU

In 2007, NVidia released CUDA, a parallel computing architecture allowing for general purpose use of GPUs (known as GPGPU). This allowed developers and researchers to make use of the processing power provided by GPUs for a wide range of applications. Due to the much higher level of parallelism available through the use of GPUs rather than CPUs, as well as the ability to research parallel computing on a single machine rather than a distributed system, many researchers began to take advantage of this new architecture. A GPU-based variant of the $\mathcal{MM}$AS algorithm (known as G$\mathcal{MM}$AS ) was developed in (Bai et al. 2009), making use of the CUDA framework and a GeForce 8800 GTX GPU. This G$\mathcal{MM}$AS solution divides the ants into multiple colonies, with each colony using slightly different parameters and each colony corresponding to a thread block. Each colony is also parallelised, with ants divided onto threads in that colony's block. This implementation shows speedups of just over 2x versus the sequential implementation, performing better than (Jiening, Jiankang, and Chunfeng 2009), which does not make use of CUDA. One issue with this approach is the use of a consumer-level GeForce GPU which is significantly less powerful than Nvidia's range of Tesla GPUs which are specifically designed for GPGPU. While the 8800 GTX features 128 CUDA cores, the Tesla C1060 used in (You 2009) has 240 CUDA cores available, as well as being generally more powerful. This allows the Tesla implementation to reach speedups of around 12x on the same 400 city instances used in (Bai et al. 2009), and 21x on 800 city instances.

While previous techniques focused on parallelising the already existing techniques developed for ACO, (Cecilia, García, Ujaldón, et al. 2011) is the first of many papers to begin to address the need for new, parallel-specific techniques that would further improve ACO on parallel architectures. The new technique forgoes the traditional "*roulette wheel*" approach employed by sequential ACO algorithms and instead utilises a *data-parallel* approach. A thread block is associated with each ant, with each thread representing a city that an ant is able to visit. These threads load the heuristic value associated with the city it represents, generate a random value between 0 and 1, and check the *tabu* list to see whether the city has already been visited or not (a city will have a value of 0 if it has been visited, and 1 if it has not). These values are multiplied together and stored in a shared memory array, which is then reduced to determine the next city that an ant will visit. This technique would later be called *I-Roulette* (Figure 2.4) (Cecilia, Nisbet, et al. 2013), and shows speedups of up to 21.71x against the sequential CPU implementation on the *pr2392* TSP instance.

Another technique developed for parallel ACO is *Double Spin* Roulette (DS-Roulette) (Dawson and Stewart 2013b), which like *I-Roulette* aims to replace roulette-wheel selection with a technique more suited for parallel architecture. DS-Roulette aims to improve on *I-Roulette* in several ways, including removing the need for costly random number generation and reducing dependence on shared memory. *DS-Roulette* is divided into three stages: in the first stage, each thread checks whether the city it represents has been visited in the current tour, similar to the *tabu* list in *I-Roulette*. If valid cities remain within a thread sub-block, the heuristic information is retrieved from shared memory and multiplied by the *tabu* value. A block-wide reduction is then performed on these values, with the results from each sub-block being stored in shared memory. At this point there is a list of sub-block probabilities. A selection is then made from these sub-block probabilities using *roulette wheel* selection. Once a

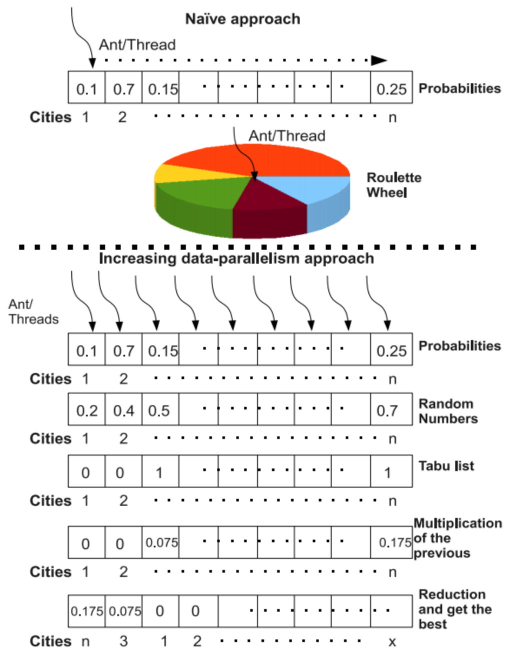


Figure 2.4: Diagram of I-Roulette technique (Cecilia, Nisbet, et al. 2013)

sub-block has been selected, another *roulette wheel* selection is performed, this time selecting a thread within the sub-block. The selected thread represents the city that will be traveled to next. DS-Roulette shows speedups of up to 8.5x against *I-Roulette* on the *pcb442* TSP, although speedup drops to 4.06x for *pr2392*. When compared to a sequential CPU implementation, DS-Roulette is up to 82x faster.

While developments in parallel ACO largely focused on adapting the $\mathcal{MM}$AS technique, the ACS technique was also parallelised (Skinderowicz 2016). As with other parallel ACO implementations, each ant corresponds to a single thread block, and a candidate set of equivalent size to the GPU's warp (groups of stream processors) size, which in this case is 32. Due to the size restriction of potential destination cities being limited to 32 due to the *candidate set*, *I-Roulette* and other methods are ignored in favour of a more straightforward *roulette wheel* selection. The global pheromone update is performed by a separate kernel using 128 threads, with the local pheromone update being performed as part of the tour construction process. Two versions of this code have been designed, one (ACS-GPU) which executes in a similar fashion to the sequential code, with an ant taking a step in its tour, then performing local pheromone update. This is a single step of the process, which corresponds to one GPU kernel execution. The other (ACS-GPU-Alt) implementation aims to maximise speedup by constructing a whole tour and performing local distribution in a single kernel execution. One of the drawbacks of this method is potential loss of solution quality due to simultaneous memory reads and writes which may cause updated pheromone values to be lost. These methods have also been tested with the use of the previously mentioned selective pheromone memory technique (ACS-GPU-SPM) (Skinderowicz 2012). These techniques were compared with the base ACOTSP code, with ACS-GPU showing speedups of between 2x and 6x, and ACS-GPU-Alt showing speedups of between 13x and 17x. However, as expected, the average solution quality of the ACS-GPU-Alt technique is poorer than ACS-GPU, though it is better than the base ACOTSP code. ACS-GPU-SPM outperforms ACS-GPU in terms of speedup, but has a lower speedup than ACS-GPU-Alt, though the SPM technique produces better quality solutions.

Díaz et al. (2020) proposed a parallelisation techique, referred to as the “Multiverse method”, which could be widely applied to a range of optimisation techniques, including ACO. Rather than parellelising a single instance of an algorithm (such as dividing ants in an ACO algorithm between threads), Multiverse instead runs multiple instances of an algorithm in parallel. One of these instances has special status, and is referred to as “The Collector”. The Collector receives updates of the best solutions from every other instance, and can apply this knowledge to its own instance. As this is just a one way injection of information from the other instances to The Collector, overhead is minimal. The results of the Multiverse method were compared with the Multistart method, which operates similarly to Multiverse but without the Collector, meaning there is no communication between instances. Both methods were tested using the asymmetric TSPs available in the TSPLIB package. For both the ACO and GA variants of the experiments, Multiverse provides identical or superior solutions compared to Multistart. In terms of solution time, Multiverse is generally slightly faster than Multistart, though not for every experiment. This demonstrates that the overhead time of Multiverse is negligible. Overall, the Multiverse technique outperforms Multistart.

While CUDA simplified research into parallelisation, GPGPU applications required a significant amount of CUDA-specific code which severely limited the portability of these techniques. With the release of their first manycore co-processor, Intel provided a parallel computing platform that could run native x86 code with minimal changes.

## 2.5 Single Instruction Multiple Data

Single Instruction Multiple Data (SIMD) instructions were initially restricted to supercomputers in their earliest history, first used in 1966, and this continued to be the case through the 1970s and most of the 1980s, until Multiple Instruction Multiple Data (MIMD) instructions became more accessible in the late 1980s. With this development, the future of SIMD lay in the desktop market, with early desktop SIMD instruction sets such as Hewlett-Packard's MAX instructions and Sun Microsystems' VIS instructions being developed to satisfy the increasing demand for video codec processing and real-time gaming. Intel soon followed suit, introducing their MMX instruction set in 1997. Intel would continue to supplement the MMX instruction set, with several releases of their Streaming SIMD Extensions (SSE) adding to the SIMD capabilities of their processors. SSE allows for instructions to be carried out simultaneously on a four-wide vector of single-precision floating point numbers, with available instructions including loading from memory to a vector (and vice-versa), arithmetic instructions, and comparisons between two vectors, among many others. The SSE2 instruction set, released in 2000, adding integer, double-precision float and character compatibility to SSE, as well as adding instructions that were compatible with these data types. SSE3, released in 2004, added the ability to perform instructions within a vector. Two further expansions of the instruction set, SSSE3 and SSE4 were released in 2006, shifting focus away from multimedia applications.

In 2012 Intel released the first of their new range of manycore processors, the Intel Xeon Phi (Figure 2.5) (Chrysos 2014). The Xeon Phi began as a derivative of Intel's Project Larabee, which was intended to be a GPGPU micro-architecture. While Project Larabee was cancelled in 2010, development of the Xeon Phi continued. The first publicly available generation of Xeon Phi processors, *Knight's Corner*, contained between 57 and 61 cores each, with each core having 4 processing threads available. The cores themselves, being modified versions of the original Pentium processor, have low clock speeds, ranging from 1GHz to 1.2GHz. A key feature of the Xeon Phi is the *Vector Processing Unit* (VPU), which features a novel 512-bit *Single Instruction Multiple Data* (SIMD) instruction set known as *Intel Many Core Instructions* (IMCI) (Lomont 2011). The notable addition in this instruction set was the ability to perform fused multiply-add instructions, which executes 32 single-precision float or 16 double-precision float operations in a single cycle. The Extended Math Unit featured in the Knights Corner's VPU allows for vectorised square root and log operations. This instruction set only ever featured on the Knights Corner architecture, and was not carried forward to the next generation of Xeon Phi, Knights Landing. Instead, Knights Landing featured the AVX-512 instruction set, a 512-bit extension of the AVX2 instruction set. Unlike previous instruction sets, which required every instruction from the set to be implemented, AVX-512 consists of many extension sets which may be optionally implemented, with only the core extension AVX-512F being mandatory. This leads to the AVX512 instruction set available on Knights Landing being more limited than the IMCI instructions available on the previous generation. However, AVX-512 will be available on future generations of desktop processors such as AMD's Zen 4 architecture, while IMCI is limited to the expensive and fairly uncommon Knights Corner architecture. This VPU allows for the execution of 16 single-precision floating point operations per processor cycle. Prior to Xeon Phi, a solely CPU-based implementation of ACO had no chance of performing as well as a GPU implementation, but the high number of available threads provided by Xeon Phi greatly increase the potential of CPU-based parallel ACO techniques.

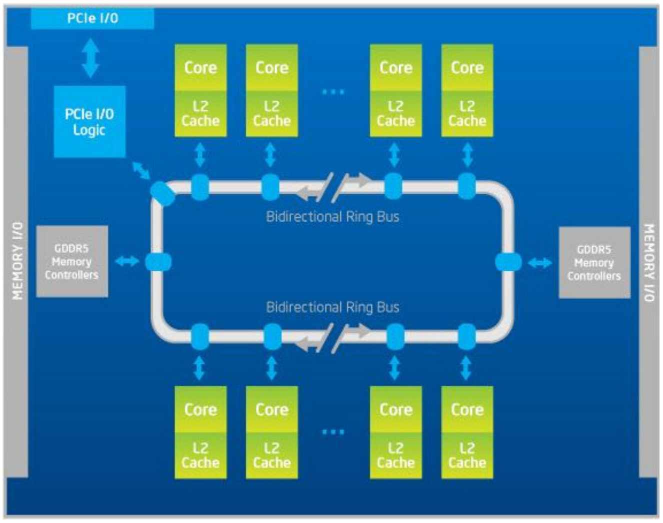


Figure 2.5: A diagram of the Xeon Phi

Initial experimentation using ACO with the Xeon Phi gave poor results, although this was for the quadratic assignment problem rather than TSP (Sato et al. 2014). This solution also fails to make proper use of the Xeon Phi's SIMD capability, which is crucial for good performance. The first research on solving the TSP using ACO was in (Tirado, Urrutia, and Barrientos 2015), which includes useful experiments in determining the best way to make use of the Xeon Phi's hardware. As with previous work, this research focuses on the tour construction phase of the ACO algorithm as this is the main bottleneck. Firstly, the *choice_info* matrix is calculated, which is size $n^2$. Two methods have been created to calculate the matrix, with the only difference being the use of the C++ *pow()* function. While v1 of the method uses *pow()*, v2 instead uses several consecutive multiplications using IMCI.

The second development in this technique is the calculation of the probability matrix, which makes use of the previously calculated *choice_info* matrix. The rows and columns of this matrix represent ants and cities respectively, with the value of a cell indicating the chance of a specific ant visiting a specific city. As with the *choice_info* calculation, two separate methods were developed for this task. Method one divides the calculation task between available cores rather than available threads, meaning four threads work simultaneously to determine the probability for one ant. Method two divides the task between threads rather than cores, which was implemented to make use of the Xeon Phi's VPU.

The experimental results of these techniques give a useful insight into how to get the most efficiency from the Xeon Phi. Firstly, *choice_info* v2 outperforms c*hoice_info* v1, demonstrating that making use of several multiplication IMCI instructions is more efficient than using the C++ pow() method, although the advantage is a very slim one. A more promising discovery is the huge advantage that the second, thread-based probability method has over the first, core-based method, with speed ups that are around 5x greater than the speedups demonstrated by the first method. The fact that the second method is designed to make the most efficient use of the Xeon Phi's VPU demonstrates how powerful it can be when fully utilised. Overall, the two techniques combined provide a speedup of up to 41x vs the sequential counterpart. Contrary to the initial work on ACO with Xeon Phi (Sato et al. 2014), this work indicates that the Xeon Phi is viable platform for further ACO research.

In (Lloyd and Amos 2016), the aim is to further increase the efficiency of ACO on Xeon Phi and achieve greater speedups. As with the previous paper, the tour construction phase is the main focus. However, this time the aim is to increase the efficiency of the edge selection process. Two techniques have been developed, both based on the two best performing techniques for GPU: *vRoulette*-1, based on *I-Roulette*, and *vRoulette*-2, based on *DS-Roulette*. Unlike *I-Roulette*'s original implementation, *vRoulette*-1 is a *task-based* approach. Rather than thread blocks representing ants and threads representing cities, ants are each represented by a single thread. When selecting the next city to visit, each ant loops through potential cities 16 at a time by making use of 16-wide vectors and IMCI instructions. Three 16-wide vectors are created for each loop, one containing the probability of an ant visiting the city, one containing a random number and one containing the *tabu* value. Due to the IMCI instruction set this process can be done in two process cycles, with a masked load of 16 cities from the weight matrix being one process, and multiplying 16 cities by 16 random numbers being another. This would be done until every city that can be traveled to has been processed, with a reduction being performed on the resulting edge weights. The edge with the largest weight is then traveled to, and the process repeats until a complete tour is formed. The IMCI instruction set and number of threads available to the Xeon Phi (240 in this paper) make this approach preferable to the *data-based* approach of the original *I-Roulette*.

Unlike *vRoulette*-1, *vRoulette*-2 selects cities with probabilities that are proportionate to the weights (cities with higher weights are more likely to be selected in *vRoulette*-1 but it isn't exactly proportionate). In *vRoulette*-2 cities compete against each other in trials in which their weight is multiplied by a random number and their tabu value, with the winning city in a trial accumulating the weight of the losing city. These trials are carried out 16 at a time using the IMCI instructions, with weights and indices being stored in 16-wide vectors. The winning weights from each vector are then compared against each other sequentially, with the winning city at the end of this process being traveled to next.

The other significant parallelisation done in this code is in the pheromone update stage. As the base ACO technique used here is $\mathcal{MM}$AS only one ant distributes pheromone at the end of an iteration, meaning that there isn't any scope for parallelisation with this process. Pheromone evaporation and clamping between the maximum and minimum values, however, takes place in a nested *for* loop, with the outer loop being distributed amongst available threads using *OpenMP* and the inner loop being vectorised using IMCI.

The results of the experiments performed on these techniques show that *vRoulette*-1 very slightly outperforms *vRoulette*-2 on all tested instance sizes in terms of execution time, though the time difference on the largest tested instance, pr2392, was less than a hundredth of a second per iteration. Where *vRoulette*-1 really has an advantage over *vRoulette*-2 is in solution quality, which ideally would be as close to optimal as possible despite not being the focus of the paper. Both techniques are an order of magnitude faster than the technique presented in (Tirado, Urrutia, and Barrientos 2015), although Tirado's technique was based on Ant System without a nearest neighbour list which had a significant impact on execution time compared to the reference ACOTSP code (Stützle 2004), which uses $\mathcal{MM}$AS and a nearest neighbour list.

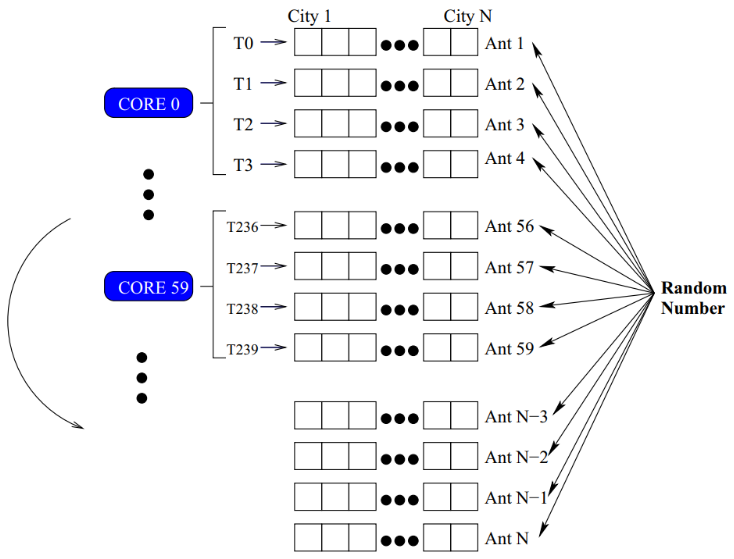


Figure 2.6: Diagram of the *UVRoulette* method

A similar vectorised implementation of *I-Roulette* is demonstrated in (Tirado, Barrientos, et al. 2017). This method, known as *UVRoulette* (Figure 2.6), makes use of the same 16-wide vectors as *vRoulette*-1, with one each for probability, random number and tabu value, with these values being multiplied together. When combined with the techniques developed in (Tirado, Urrutia, and Barrientos 2015), this method produces speedups of up to 42x vs the sequential version of the code, which is slightly higher than the results measured without using UVRoulette. While this doesn't produce a notable new technique due to UVRoulette's similarity with *vRoulette*-1, it does include a useful comparison with results measured by GPU techniques. While it's difficult to directly compare the Xeon Phi's hardware with the GTX580 and Tesla C2050 used in the GPU experiments, a speedup of 1.53x and 2.43x respectively is predicted on the Xeon Phi based on hardware alone. Against *I-Roulette*, these techniques begin to outperform this expected speedup on instances of size 783 and up, whereas against *DS-Roulette* the outperformance begins at *pr1002*. This proves that Xeon Phi is able to outperform GPU implementations of ACO even when hardware differences are taken into account.

## 2.6 The Scalability of Ant Colony Optimisation

While parallelisation significantly reduces the execution time of ACO, one of the main limitations of using ACO with TSP, especially on large instances, is the dependence on the pheromone matrix. While the size is manageable on smaller instances (the pheromone matrix for the *pr442* TSP, a TSP instance from the widely-used TSPLIB library, contains 195,364 32-bit floats, meaning that overall the pheromone matrix requires 0.7MB), on larger instances the memory requirements become a significant obstacle. The Mona Lisa TSP, which contains 100,000 cities, has a pheromone matrix memory requirement of around 40GB, far more memory than most computers are able to allocate. In order to enable ACO to effectively solve very large problem instances an alternative to the pheromone matrix needs to be used. Several steps have been made to remove the need for a pheromone matrix.

The first development in removing the need for the pheromone matrix is *Population-based ACO* (P-ACO) (Guntsch and Middendorf 2002). This technique instead adds the best ant in each given generation to a solution population. The ants consult this population when performing edge selection, using paths from their current city that exist in tours in the solution population to decide the city that will be traveled to next. Once a pre-defined number of solutions is reached, the weakest tour in the population is removed after every new generation as well as the best new solutions being added to the population, maintaining the number of solutions in the population. When a solution is removed, the pheromone of that solution is no longer used by ants when performing edge selection. This technique was developed for use with dynamic TSPs which require a more reactive pheromone matrix in order to quickly adapt to changes in the problem. While by itself this technique isn't too useful in solving larger instances, it forms the basis of further work that seeks to eliminate the pheromone matrix altogether.

*PartialACO* (Figure 2.7) (Chitty 2017) is inspired by P-ACO, but makes several modifications. Firstly, the solution population is replaced by each ant having a local memory of the best tour that the ant has found. This is used during the edge selection phase. The pheromone provided by these *local best* tours is equal to the solution quality of the *global best* tour divided by the solution quality of the *local best* tour. Another change is the use of the *I-Roulette* technique (Cecilia, Nisbet, et al. 2013) to parallelise the algorithm. As with most new ACO techniques, particular attention is paid to the tour construction phase, as it is the most computationally expensive phase with a computation time that increases as the number of cities increases.

The main difference between *PartialACO* and previous techniques is that while previous techniques have each ant create a full tour every iteration, *PartialACO* instead has ants take a previous good quality tour and change a small part of it. At each iteration an ant randomly selects a starting city, and also randomly selects section of the local best tour to remain unchanged, with the remaining part of the tour constructed as normal. This greatly reduces the amount of steps an ant needs to take while constructing a tour, reducing computation time of each ant.

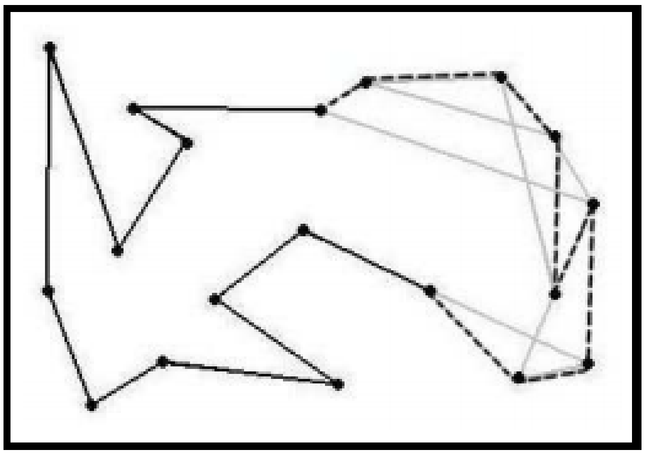


Figure 2.7: Illustration of *PartialACO* demonstrating the area of the tour which is able to be changed (dashed section) and area of tour that will be unchanged (solid section) (Chitty 2017)

*PartialACO* was evaluated against the P-ACO technique that it was based on, and then evaluated on very large TSP instances. In the first stage of testing, *PartialACO* is proven to be faster than P-ACO with speedups of up to 2.8x as well as providing better quality solutions. Further evaluations were carried out that restrict the amount of the local best tour that an ant is able to to modify, with speedup increasing as less of the tour was able to be modified. However this has the opposite effect on solution quality, with solution quality decreasing as the restriction percentage increased. In order to counteract this effect, *2-opt local search* is used which improves solution quality while slightly reducing speedup amount. For the tests on the very large TSP instances (ranging from 100,000 to 200,000 cities), each ant can change only 1% of the local best tour, significantly reducing the problem size faced by each individual ant, which in turn reduces computation time. Each ant also has a 0.001% chance of performing a *2-opt local search*. The execution time for this technique ranges from an average of 1.07 hours for the 100,000 city TSP to 5.06 hours for the 200,000 city TSP. This is a speedup of up to 1199x against P-ACO on the same problems, while also greatly increasing solution quality against P-ACO. This approach uses just one quad-core i7 CPU, so utilising this technique on more powerful hardware could lead to even greater speedups. Despite these promising results, it should be noted that the quality of solutions found by the P-ACO technique is still much poorer on these very large TSP instances than the current state documented state of the art (*TSP Art Instances* n.d.), which makes use of genetic algorithms (Honda, Nagata, and Ono 2013).

While the memory complexity of large TSP instances is the focus of techniques that replace the pheromone matrix, time complexity is still a key focus, as even with a reduced matrix, the instances are still very large. An important development in this regard is the use of *candidate sets* (Figure 2.8). While candidate sets were used by both ACS and $\mathcal{MM}$AS in the context of *local search*, only in later versions of ACS and $\mathcal{MM}$AS are candidate sets used by ants in the tour construction phase in order to reduce the complexity of larger TSP instances (Stützle and Dorigo 1999). In this case a static approach to candidate set is employed, in which the nearest neighbor list for each vertex is determined before the ants begin their tours and remains constant throughout execution. A dynamic approach to candidate sets has also been investigated (Randall and Montgomery 2002), and while this was shown to perform better on complex problems like quadratic assignment, more basic problems like TSP are simple enough that the extra processing time required for a dynamic approach actually increases processing time as the benefits of a dynamic approach are outweighed by the higher amount of computation required.

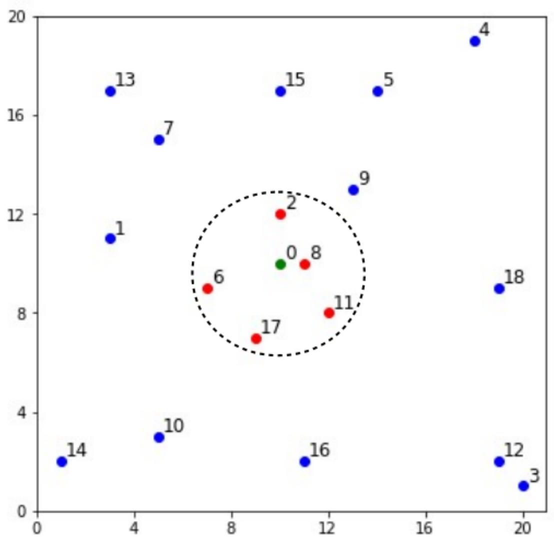


Figure 2.8: Illustration of a *candidate set* within a TSP - red cities are the 5 closest and are in the nearest neighbour list of the green city

The first variant of ACO to make use of *candidate sets* is ACS, for experiments with large TSP instances. This initial implementation constructs a list for each city of the fifteen closest cities. When an ant is attempting to determine which city to move to next, it can only select from cities in the *candidate set*. If there are no available cities in the *candidate set*, the ant is able to move to a city outside of the *candidate set* instead. *Candidate sets* prove very useful for larger TSP instances as they restrict the number of possible decisions an ant has to take, causing only slightly more than linear growth in solution time rather than the quadratic growth that occurs without the use of *candidate sets*. For *symmetric* TSPs, ACS performs comparably to other contemporary solutions that were considered "very good", such as the large step Markov chain algorithm, while being outperformed by genetic algorithms (Dorigo and Gambardella 1997b). More recently, the use of *nearest neighbour* candidate lists has shown significant promise in solving large instances of the TSP using ACO (Cecilia, García, Ujaldón, et al. 2011; Dawson and Stewart 2013a). This refinement is based on the assumption that good solutions to the TSP avoid travelling between distant vertices, an assumption reinforced by results (Dorigo and Gambardella 1997b), and that they can generally be found by making only relatively *local* transitions from vertex to vertex. This assumption can be reinforced by examining methods of solving TSPs which lead to good solutions: The Lin, Kerninghan and Helgsaun (LKH) (Helsgaun 2000) technique focuses on k-opt moves, replacing $k$ edges with $k$ other edges - replacing long edges with short edges always leads to an improvement in solution quality. While this assumption makes sense on most TSPs, several - such as pr2392, a TSPLIB problem used in future sections - feature clusters of cities with long distances between clusters. In these instances, the assumption may not be valid, and when the assumption is not valid it is safe to assume that the use of candidate sets can overly restrict the search space, leading to reduced solution quality. Candidate sets still have a place in these clustered algorithms through the substitution of the standard euclidian distance metric with other metrics. An example of this is the POPMUSIC technique (Taillard and Helsgaun 2019), a metaheuristic designed to solve clustered TSP problems, uses tour merging to generate candidate edges. Although candidate lists are now a standard component of parallel ACO-based algorithms (Cecilia, García, Ujaldón, et al. 2011; Dawson and Stewart 2013a), previous implementations of this feature have failed to take advantage of the vector processing capabilities of processors such as the Xeon Phi. A modified representation of the nearest neighbour list can fully utilise vector processing, yielding significant performance improvements. Moreover, the speedup obtained increases as the problem size grows, suggesting that this method will be a required component of future ACO-based algorithms for large-scale instances of similar problems.

## 2.7 ACO on Real World Problems and Virtual Machine Placement

While ACO is most commonly benchmarked using the Traveling Salesman Problem, it has been applied to a wide range of optimisation problems. From traditional optimisation problems such as the Quadratic Assignment Problem (Maniezzo and Colorni 1999) and Vehicle Routing Problem (Bell and McMullen 2004), to pencil puzzles such as Sudoku (Lloyd and Amos 2020) and Nurikabe (Amos, Crossley, and Lloyd 2019), ACO is proven to be a dynamic and versatile algorithm that has produced good results on a wide range of problems.

ACO is also being used to solve modern real-world problems, one of which is Finanicial Crisis Prediction (FCP) (Uthayakumar et al. 2020), a process undertaken by financial firms. FCP aims to calculate risk and allow firms to avoid new credit proposals when risk is higher than a pre-defined acceptance level. This process involves analysing high-dimensional financial data, which can lead to extremely high computational complexity. This issue can be solved through the use of feature selection, identifying a subset of the features of the data which are key to the FCP and removing noisy features. The ACO-FCP model consists of two stages: Feature selection and data classification. For feature selection, a feature's heuristic is based on the number of times the feature appears in a pre-existing set of best solutions. In the data classification phase, the aim is to determine the best value for each of the attributes that are included in the feature subset, with "best" in this context meaning the value which is most likely to lead to a more accurate classification. The heuristic value for this stage is the entropy associated with a given term (with a term being a part of the classification rule). For the feature selection stage, ACO-FS outperformed Genetic Algorithm, Particle Swarm Optimisation and Grey Wolf Optimisation (Mirjalili, Mirjalili, and Lewis 2014) based on the computational cost of the selected subsets. The ACO-based classification method is competitive with other significant classification methods.

Another modern real-world problem that has been solved with ACO is the Electric Capacitated Vehicle Routing Problem (E-CVRP) (Mavrovouniotis, Menelaou, et al. 2020), a variant of the well known Vehicle Routing Problem that deals with electric vehicles. E-CVRP includes additional constraints compared to VRP that are specific to electric vehicles. The most significant constraint is the need for vehicles to regularly visit charging stations. The problem involves a fleet of electric vehicles each with a limited battery charge level and limited cargo capacity. These vehicles must be used to satisfy the delivery demands of a set of customers as efficiently as possible. The current cargo load of a vehicle also affects the speed at which the battery drains. Every customer must be visited a single time by a single vehicle, and each vehicle must start and end at a depot. Vehicles start fully charged and fully loaded, and visits to charging stations fully charge the vehicle. $\mathcal{MM}$AS is utilised to solve this problem. Ants begin at the depot, and each complete a full E-CVRP solution. They are unable to travel to nodes that would cause negative cargo, negative power, or lead to the ant being too far from a charging station. As with TSP, the heuristic value represents distance. ACO was compared with a mixed-integer linear program (MILP) approach, and while the ACO solutions qualities were generally worse for smaller problems, the MILP approach was unable to solve larger problems in the allocated time, meaning ACO provided the sole results for large problem instances.

An increasingly relevant real-world problem is Virtual Machine Placement (VMP), in which the aim is to maximise the efficiency of virtualisation. Hardware virtualisation in cloud computing allows for many separate machine instances to be created that are distinct from the host machine they are utilising. These instances, known as Virtual Machines (VMs), essentially act as a completely separate computer, distinct from the other VMs on their host. These VMs are managed by a hypervisor (also known as a Virtual Machine Monitor, VMM) which creates VMs, optimises their performance and monitors them. Each host machine has its own hypervisor.

VMP was selected as a benchmark problem for the techniques developed in this thesis for several reasons: firstly, it is a highly relevant problem, with cloud computing becoming increasingly prominent especially in the wake of the COVID-19 pandemic which has caused many businesses to depend on remote working (Alashhab et al. 2020); secondly, it is a highly time-sensitive problem, with VMs needing to be allocated efficiently and quickly in order to minimise the time spent in an inefficient configuration, needlessly wasting energy and money for the host; finally, it is a problem that has not yet been solved using a parallelised ACO implementation, and so the results of these new techniques can provide a benchmark for any potential future experimentation on VMP.

Many companies such as Amazon (AWS) and Microsoft (Azure) provide access to Virtual Machines hosted on their own servers. Due to the sheer size of these operations a significant amount of hardware is required to offer these services to customers. In order to actively run as little physical hardware as possible, a process called *Virtual Machine Migration* is often used to move Virtual Machines from one Physical Machine to another without any disruption for the user (Clark et al. 2005). This ability to seamlessly migrate VMs allows the mass migration of many VMs to more efficiently allocate VMs to PMs. For scenarios where a small number of servers are available, determining the most efficient layout of VMs can be trivial. However, services such as AWS have millions of users and hundreds of thousands of servers which significantly complicates this process.

VMP is an NP-hard problem (Stillwell et al. 2010) in which the aim is to allocate Virtual Machines (VMs) to Physical Machines (PMs) as efficiently as possible (the definition of "efficient" may be expressed in terms of energy usage, or some other metric). Unlike other problems, VMP currently does not have a widely accepted problem definition as small variations exist between implementations. The problem defined in (Liu et al. 2016) is a straightforward version of the VMP problem, and is variant used in this thesis. While VM requirements differ between implementations, the two most typical requirements are RAM and CPU. RAM requirements are generally measured in Gigabytes (GB), while CPU requirements are generally measured in either processor cores or MIPS (million instructions per second). Every VM will have its own specific requirements, which means it occupies its own resource "footprint". The aim of the problem is to allocate every VM to a PM in such a way that the number of PMs required is minimised (this may be thought of as a bin packing problem). Here, the static variant of the VMP problem is focused on, where a fixed set of VMs needs to be allocated to PMs. A more detailed definition of the VMP problem can be found in Chapter 6.

Feller et al. (Feller, Rilling, and Morin 2011) developed an early implementation of ACO for VMP. This treated VMP as a multi-dimensional bin-packing problem (MDBP), with the Physical Machines representing the bins and the VMs representing the item to be packed. As reducing the number of used PMs is the most effective way of reducing energy usage, the objective of the algorithm is to minimise the number of bins used. The algorithm fills PMs one-at-a-time, with each bin being closed when no remaining VMs can fit inside. Pheromone is deposited on Item-Bin pairs, with VMs being linked to the specific PM they were allocated to. The heuristic is based on the total resource utilisation of the PM if the current VM were to be assigned to it, and the pheromone is based on the average utilisation of all utilised PMs. The Feller ACO technique outperforms FFD in terms of energy usage, saving 4.1% on average. However, execution time for the algorithm is significant, ranging from 37.46 seconds for 100 VMs to 2.01 hours for 600 VMs.

A more recent ACO-based VMP solving algorithm is OEMACS (Liu et al. 2016), named for the combination of the Ordering Exchange and Migration (OEM) local search it utilises, and the Ant Colony System (ACS) algorithm used for solution construction. The most significant changes between OEMACS and FellerACO are the switch from an $\mathcal{MM}$AS based implementation to an ACS-based implementation, and the addition of two Local Search procedures: an exchange procedure similar to a local search procedure used for Bin Packing Problems (Alvim et al. 1999) which swaps VMs between PMs in an attempt to find a more efficient configuration, and an insertion/migration procedure which attempts to remove a VM from one PM and insert it into another. OEMACS outperforms FellerACO in both solution quality and execution time.

Aside from ACO, other techniques have been applied to the VMP problem. As well as heuristic-based approaches such as Next Fit, First Fit, First Fit Decreasing (FFD) and Best Fit (BF) (Jr, Garey, and Johnson 1996), more advanced optimisation techniques have been successfully applied to the VMP problem; these include Genetic Algorithms (GA) (Gao et al. 2013; Mi et al. 2010; Tao et al. 2015), Particle Swarm Optimisation (Wang, Liu, et al. 2013), and Q-Learning (Masoumzadeh and Hlavacs 2013).

A recently-published method for VMP is the Improved Genetic Algorithm for Permutation-based Optimisation Problems (IGA-POP) (Abohamama and Hamouda 2020). IGA-POP frames the VMP as a Variable-Sized Bin Packing Problem (VSBPP), a variant of the Bin Packing Problem (Friesen and Langston 1986). VSBPP differs from the classic Bin Packing Problem by allowing containers to have differing capacities, rather than having a set container type with identical capacities across the board. As the IGA-POP algorithm deals with three resources, RAM capacity, CPU capacity and Bandwidth, it can also be considered a Multi-dimensional Bin Packing Problem. In IGA-POP, a solution (contained within a *chromosome*) encodes an ordering of VM assignments to PMs.

A population of random chromosomes is generated and then each chromosome sequence is assigned to PMs. This assignment is done using the Best Fit (BF) greedy search algorithm. Once each of the initial chromosomes have undergone the BF assignment process, the $M$ best are determined and added to a separate set, with $M$

being equal to the number of PMs. Crossover or Mutation then takes place, with each of these operations having an equal probability of being selected. Crossover combines the current chromosome with a randomly selected chromosome from the $M$ best set, and the best of these 3 chromosomes (original, $M$ best and child) is retained and added to the solution population. If crossover is not selected, a mutation operation is then performed, of with there are three types. These are swap (two VMs in the chromosome are randomly selected and swapped), reversion (a subset of the chromosome is randomly selected and then reversed) and displacement (a subset of the chromosome is randomly selected and then moved to another position within the chromosome), again chosen with equal probability. This process is then repeated until the algorithm reaches its end condition, after which the best chromosome is selected as the final results.

The fitness function for this algorithm prioritises low power usage, and it performs competitively in terms of solution quality against the BF and First-Fit (FF) greedy algorithms, the Sine-Cosine Optimisation Algorithm (SCA) (Mirjalili 2016) and a generic GA.

This chapter discussed the ACO algorithm as well as the inspiration, origins and variants of the algorithm. Also discussed were a wide array of problem domains that the ACO algorithm has been successfully applied to, as well as other ideas, such as parallelisation, that have been successfully applied to ACO. This chapter demonstrates the large impact that the ACO algorithm has had and continues to have within the Evolutionary Computation research area, emphasising the continuing relevance of the technique and the general applicability that the technique has. The next chapter will discuss the technical details of the ACO algorithm, as well as describing the different approaches taken by the main ACO variants, ACS and $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System.

Table 2.1: Summarising table of all discussed ACO parallelisation techniques

| Title | Description | Reference |
|---|---|---|
| Fork-join | Distributed system featuring ants solving problems independently, updating global pheromone once all ants had completed solution. | (Bullnheimer, Kotsis, and Strauß 1998b) |
| Asynchronous | Evolution of Fork-join approach, with ants assigned to "worker" machines that update pheromone locally between ants on that "worker", with infrequent global updates. | (Bullnheimer, Kotsis, and Strauß 1998b) |
| Independent Colonies | Independently run ant colonies on seperate machines, with no communication at all | (Stützle 1998) |
| Master-worker | Similar to asynchronous approach but with no global updates - only global communication is determination of best solution at end of algorithm. | (Stützle 1998) |
| Parallel Ants | Variant of the master-worker approach with each ant (or cluster of ants) assigned to a seperate processor and global pheromone updates | (Randall and Lewis 2002) |
| Multiple Colony | Several colonies are run in parallel, with differing methods of information exchange between colonies tested - circular exchange between colonies performed best | (Middendorf, Reischle, and Schmeck 2002) |
| G$\mathcal{MM}$AS | A GPU-based variant of $\mathcal{MM}$AS utilising the CUDA framework, dividing ant colonies amongst GPU thread blocks | (Bai et al. 2009) |
| I-Roulette | An ACO implementation replacing the roulette wheel selection mechanism with a data-parallel selection technique, using CUDA | (Cecilia, García, Ujaldón, et al. 2011) |
| DS-Roulette | Another implementation demonstrating a new selection technique, Double Spin Roulette, which performs a roulette wheel to select a GPU thread block, and then a thread within the block | (Dawson and Stewart 2013b) |
| ACO for Xeon Phi | The first Xeon Phi implementation of ACO - fails to make proper use of SIMD | (Sato et al. 2014) |
| ACO for Xeon Phi (TSP) | The first Xeon Phi implementation of ACO for TSP, focusing on vectorising the tour construction process | (Tirado, Urrutia, and Barrientos 2015) |
| ACS-GPU | A parallelised implementation of the ACS technique, with each ant corresponding to a single thread block, and a candidate set with a size equal to the GPU's warp size | (Skinderowicz 2016) |
| ACS-GPU-Alt | A variant of ACS-GPU that features ants that construct a whole tour in a single kernel execution | (Skinderowicz 2016) |
| vRoulette-1 | A SIMD implementation of I-Roulette, task-based rather than data-based. Each thread represents a single ant. | (Lloyd and Amos 2016) |
| vRoulette-2 | A SIMD implementation of DS-Roulette, which makes use of a tournament selection, with vectors of ants competing against each other. | (Lloyd and Amos 2016) |
| UVRoulette | A SIMD implementation of I-Roulette, similar to vRoulette-1 | (Tirado, Barrientos, et al. 2017) |
| Multiverse Method | A variant of independent colonies, featuring a "collector" colony which receives pheromone information from the other colonies. | (Díaz et al. 2020) |

# Chapter 3

# Ant Colony Optimization

In this chapter, the framework of ACO is discussed, with more detail provided on elements of the ACO algorithm that are referred to in the previous chapter, as well as subsequent chapters. This will be broken down into the phases of the ACO algorithm, and includes some of the differences between the established variants of ACO, as well as differences between approaches to various problems. The core ACO algorithm is displayed in 3.1.

**Algorithm 1:** Pseudo-code for Vectorized Candidate Set Selection.

```
Input : Edge Weight array $\mathbf{W}_{0...N-1}$, Tabu Mask array $\mathbf{T}_{0...n-1}$, Maximum
        number of nearest neighbours $N_p$, nearest neighbour list $L_{0...N_p-1}$
Output: Selected Edge
// Variables in bold are p-vectors, superscripts indicate vector lanes
$\mathbf{W}_{max} = (0...0)$;
$\mathbf{I}_{max} = (0...0)$;
for i = 0 to $N_p - 1$ do
    if $L[i]$.ivec $\neq -1$ then
        $\mathbf{R} = Random()$;
        $\mathbf{V} = L[i]$.mask;
        $\mathbf{I} = (pL[i].\mathtt{ivec}...pL[i].\mathtt{ivec} + p - 1)$;
        $\mathbf{w} = ApplyMask(V_i, \mathbf{W}_i \times \mathbf{R}, (-1... - 1))$;
        $\mathbf{w} = ApplyMask(T_i, (-1...1), \mathbf{w})$;
        $max\_mask = \mathbf{w} > \mathbf{W}_{max}$;
        $\mathbf{I}_{max} = ApplyMask(max\_mask, \mathbf{w}, \mathbf{W}_{max})$;
    end
end
//Reduction
j = argmax($\mathbf{W}_{max}$);
return $\mathbf{I}^j_{max}$;
```

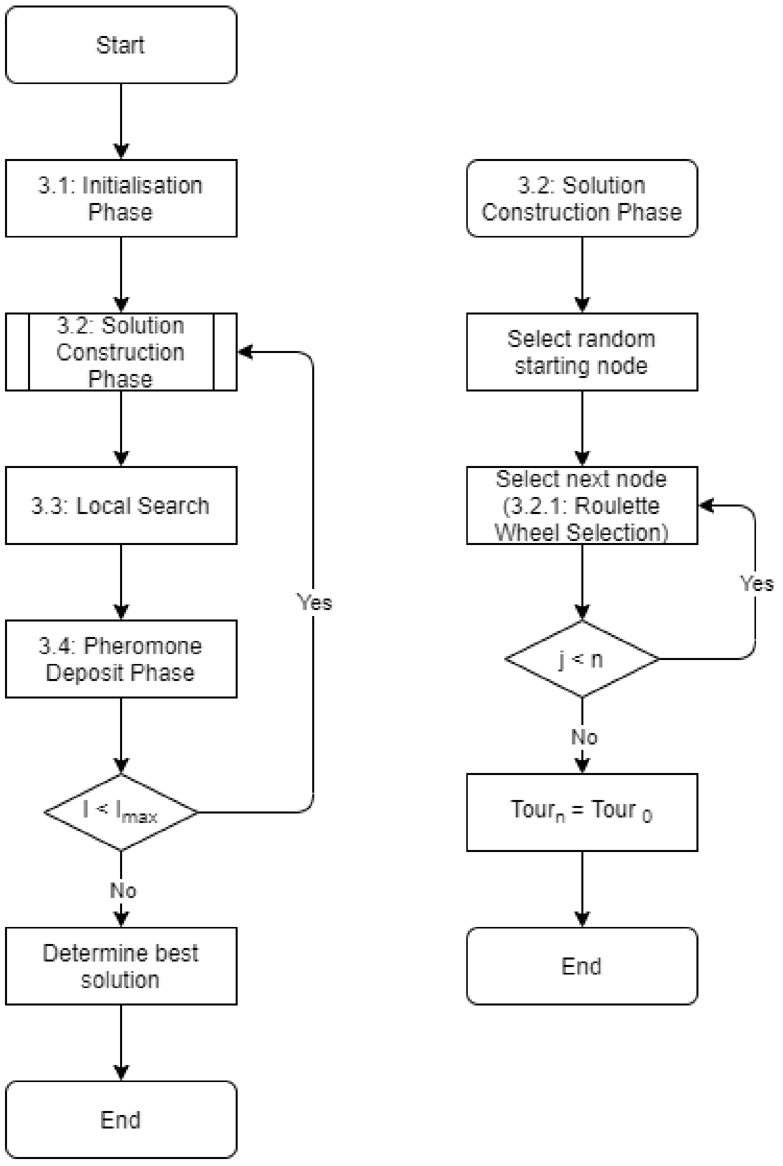

Start
3.1: Initialisation Phase
3.2: Solution Construction Phase
3.3: Local Search
3.4: Pheromone Deposit Phase
I < I_max
Yes
No
Determine best solution
End
3.2: Solution Construction Phase
Select random starting node
Select next node (3.2.1: Roulette Wheel Selection)
j < n
Yes
No
Tour_n = Tour_0
End

Figure 3.1: A flowchart demonstrating the core ACO algorithm. It should be noted that Local Search is not always performed after every pheromone deposit phase; it may be performed every *n* iterations, or never performed at all

## 3.1 Initialisation Phase

The initialisation phase has two major roles: the first being to initialise data structures such as the pheromone matrix and the ants themselves, and the second is to load the given problem into memory in a format that allows the ACO algorithm to solve it. In the case of TSP and some other problems, this can lead to the creation of a second heuristic matrix, usually identical in size to the pheromone matrix and containing heuristic information used by ants to make decisions.

This phase is generally straightforward, as the only thing that needs to be decided is the size of the pheromone matrix, which can vary depending on the problem. The pheromone matrix is generally a float matrix, with the size defined by the number of solution components in a given problem: For a TSP problem, the matrix size is $n^2$, where $n$ is the number of cities. A key decision that has to be made when applying ACO to a problem is how pheromone values are associated with the components of a solution. For example, when solving the Traveling Salesman Problem, pheromone values are associated with the edges between nodes in the graph, representing the distance between cities. The Virtual Machine Placement problem has multiple definitions, depending on the chosen approach, and pheromone can either be associated with the link between a Physical Machine and a Virtual Machine, or a link between two Virtual Machines. In order for ACO to function, the solution components that are selected from in the solution construction phase must also be the solution components on which pheromone is distributed. For the Traveling Salesman Problem, the size of the pheromone matrix is set to the number of cities in the problem squared, as this allows for pheromone data to exist between every city in the problem. A heuristic matrix also usually exists which for static TSP stores the distances between cities, and is the same size of the pheromone matrix. Like the pheromone matrix, the heuristic matrix is a float matrix. As the cities in the static TSP are constant, there will never be any change in the distances between them, meaning all distance calculation can be done on load and no further calculation is needed, saving time during the solution construction phase. A third matrix, the weight matrix, also exists for TSP, which stores the probability (weight) of moving between cities. As with the previous matrices, it stores float values and has a size of $n^2$. In the case of VMP, there are two options for the matrix size, as two different approaches for laying pheromone exist for the Virtual Machine Placement problem: the first lays pheromone between VMs that are allocated to the same PM, and the second lays pheromone between VMs and the PMs that they're allocated to. In the first case, the matrix size is the number of virtual machines squared, and in the second case the number of virtual machines is multiplied by the number of physical machines.

A key consideration for the construction of the pheromone matrix is the initial pheromone value. This can differ by technique, as ACS uses a low initial pheromone value while $\mathcal{MM}$AS has a high initial value. A smaller initial pheromone value leads to higher exploitation of the initial best solutions, whereas a higher initial value leads to a more explorative approach. The original Ant System does not specify an initial value.

As discussed in the previous section, these matrix structures can be prohibitive in regards to the scalability of ACO. Techniques such as P-ACO and PartialACO construct no matrices at all, so the initialisation phase for those techniques will only involve the initialisation of the ants and potentially other small data structures.

## 3.2 Solution Construction Phase

This phase has each ant agent independently attempt to construct a valid solution to the given problem. While this phase can differ between different problems, the majority of ACO algorithms represent their problems as a graph structure, meaning that most ACO algorithms will have a similar solution construction phase. An ant starts with an empty solution stored in local memory, and chooses a new solution component to add to that solution in every "step" of the solution construction phase

The first step of this phase is the selection of a *random starting node*, a simple process in which a starting node is selected randomly with no pheromone or heuristic input - each node will be traversed, meaning starting node selection will have no impact in overall solution quality.

For each subsequent node selection step, the ant makes a probabilistic choice, considering either the full set of solution components or a subset if candidate sets are being utilised. The selection process has three key influencing factors: heuristic ($\eta$), pheromone ($\tau$) and a random number (as well as an influence from the current partial solution, prohibiting ants from selecting invalid solution components, i.e. a city in a TSP problem that has already been visited). The initial probabilistic rule developed for the original Ant System algorithm is still widely in use. The probability of ant $k$, currently at city $i$ visting city $j$ is given as:

$$p_{i,j}^{k} = \begin{cases} \frac{\tau_{i,j}^{\alpha} \cdot \eta_{i,j}^{\beta}}{\sum_{i \in N_i^k} \tau_{i,j}^{\alpha} \cdot \eta_{i,j}^{\beta}} & i \in N_i^k \\ 0 & \text{otherwise.} \end{cases} \tag{3.1}$$

Where $N_i^k$ is the *feasible* region for city $i$, the collection of cities that the ant is able to visit from $i$, $\alpha$ and $\beta$ are *a priori* values that dictate the relative influence of pheromone and heuristic respectively, and $\tau_{i,j}$ and $\eta_{i,j}$ are the pheromone and heuristic value respectively of the edge between cities $i$ and $j$. Roulette Wheel selection (Goldberg 1989) is then used, with each possible city assigned an area of the roulette wheel proportionate to their probability as per equation 3.1.

The Ant Colony System variant of ACO uses a selection technique based on the concept of exploration vs exploitation. It makes use of a random number $q$ in the range [0,1] and a parameter $q_0$ ($0 \leq q_0 \leq 1$). If $q > q_0$, the selection is conducted using a roulette wheel and equation 3.1, but if $q \leq q_0$, the solution component with the highest probability is selected. Therefore, a higher $q_0$ value leads to a greedier solution with more exploration.

In the scenario where a candidate set is in use, the choice will be restricted further to solution components within that candidate set. By restricting the number of available cities, the time complexity of the solution construction phase can be reduced; rather than $O(n^2)$, the time complexity is $n \cdot m$, where $m$ is the size of the candidate set. The size of this candidate set varies by problem; the first ACO implementation to make use of a candidate set was ACS, which used a candidate set size of 20.

### 3.2.1 Roulette Wheel Selection

The selection method almost universally used with ACO is Roulette Wheel selection, a fitness proportionate selection technique initially developed for use with Genetic Algorithms (Goldberg 1989). The motivating factor behind the development of Roulette Wheel selection was to create a selection technique in which the chance of an outcome being selected is directly proportional to the fitness of that outcome, while preserving the stochastic nature of the selection, as a less fit outcome is still able to be selected. The selection method is illustrated in Figure 3.2.

The roulette wheel selection used in the initial version of Ant System (Dorigo, Maniezzo, and Colorni 1996) operates as follows: firstly the sum of the weight values of all potential outcomes is calculated; then a random number uniformly distributed between 0 and 1 is created, and multiplied by the sum of weights; the weight array is then looped through, with weights being added to a running weight total during the loop; finally, once this running weight total is greater than or equal to the random number, the outcome associated with the current loop position is selected as the next outcome, or in this case, the next city to be visited.

While the Roulette Wheel selection method works well for sequential implementations of ACO, the technique is difficult to parallelize as it is a fully sequential operation. I-Roulette (Cecilia, García, Nisbet, et al. 2013) and DSRoulette (Dawson and Stewart 2013b), both discussed in the previous chapter, were developed as parallel-friendly alternative selection methods.

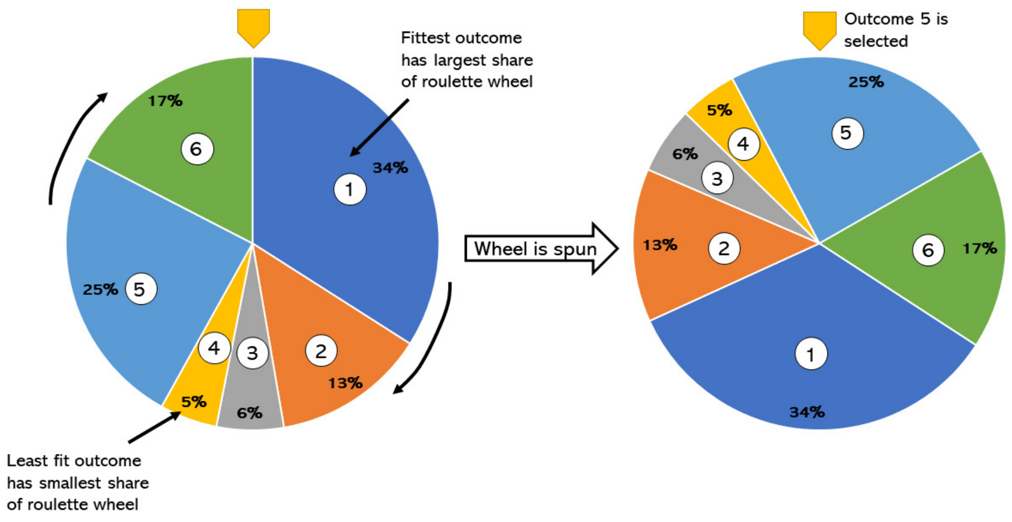

Fittest outcome has largest share of roulette wheel
Least fit outcome has smallest share of roulette wheel
Wheel is spun
Outcome 5 is selected
1 34%
2 13%
3 6%
4 5%
5 25%
6 17%

Figure 3.2: An illustration of roulette wheel selection, with each potential outcome being allocated a segment of the wheel

The Solution Construction phase ends when the current step, $j$, is greater than the number of traversable nodes in the graph, $n$. In the case that the ant is solving a TSP, the final step of the solution is set to be equal to the first step, creating a complete tour.

## 3.3 Local Search

While technically a separate technique, the pairing of Local Search with ACO is recommended for decreasing execution time, particularly with TSP (Dorigo, Birattari, and Stutzle 2006). Local search algorithms aim to reduce the length of a tour by finding local optimum in a particular neighbourhood, the contents of which vary depending on the specific Local Search algorithm used. A widely-used group of Local Search algorithms used for improving TSP solutions is the *k-opt* group. The *k-opt* local search methods eliminate crossover in a TSP tour, meaning that no edge of the tour directly crosses another edge, as the shortest route will never have crossing edges. In the 2-opt algorithm, demonstrated in Figure 3.3, 2 edges are removed and the alternate way of connecting the 4 cities while still keeping a continuous tour is analysed - if the new tour is shorter, it is saved to the solution and becomes a part of the ant's solution. This process is repeated throughout the entire solution. 2.5-opt is similar to 2-opt, but it also considers two node shifts, in which a city is removed from its current position in the

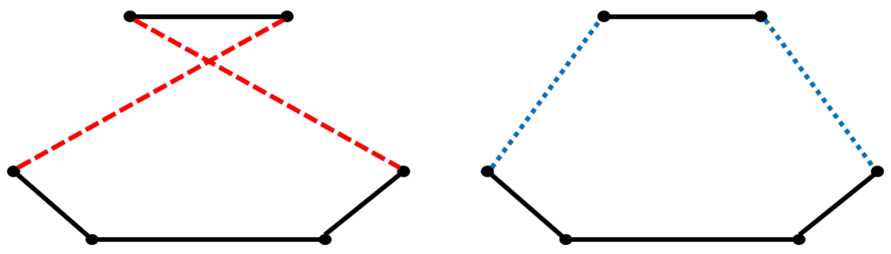


Figure 3.3: An example of the 2-opt local search procedure, deleting the red edges and replacing them with blue edges, creating a more optimal route in this case

tour and inserted between two other connected cities. This allows for a pseudo-3-opt local search, with a significant reduction in execution time vs regular 3-opt. 3-opt, demonstrated in 3.4 removes 3 edges rather than 2, and examines the 7 ways in which those edges can be reconfigured. As with 2-opt, any improvements are saved to the ant's solution.

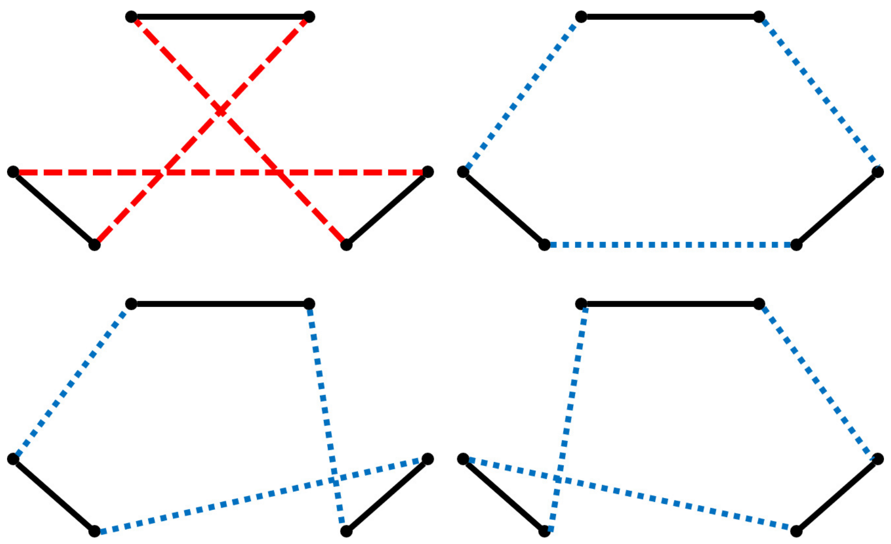


Figure 3.4: An example of the 3-opt local search procedure, deleting the red edges and replacing them with blue edges. The newly created tours are examined to determine if they are shorter than the original tour (not every potential tour is shown in this diagram)

Depending on the efficiency and execution time of the local search, the frequency at which local search takes place can vary. 2-opt is less complex than 3-opt, with a complexity of $O(n^2)$ compared to the $O(n^3)$ complexity of 3-opt (with $n$ in these cases representing the number of cities in a given TSP problem), so it could feasibly be performed after every ant completes a tour. The increased complexity of 3-opt makes it more suited to performing on an iteration-best or global-best tour once per iteration.

## 3.4 Pheromone Deposit Phase

Once each ant has constructed a solution, pheromone is deposited. This phase consists of two sub-phases: pheromone distribution and pheromone evaporation. In the original Ant System, every ant deposited pheromone at the end of its solution construction phase, with the amount of pheromone being inversely proportional to the length of the tour. This meant that shorter tours would received a higher amount of pheromone. Later variants of the algorithm, ACS and $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System instead opt to distribute pheromone only on either the global-best or iteration-best ant. In these cases, the pheromone deposit formula for ACO on TSP can be given as:

$$\tau_{i,j} = \tau_{i,j} + \Delta\tau_{i,j}\forall(i,j) \in L \tag{3.2}$$

where L is the set of edges in the ant's solution and $\Delta\tau_{i,j}$ is given as:

$$\Delta\tau_{i,j} = \begin{cases} 1/C & \text{if edge}(i,j) \in T \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

where $T$ is the set of edges in the iteration-best or global-best tour, and $C$ is the length of the tour.

Once the deposit has taken place, global pheromone evaporation is performed:

$$\tau_{i,j} = (1-\rho)\tau_{i,j}\forall(i,j) \in L \tag{3.4}$$

with the parameter $\rho$ controlling the evaporation rate. As with the previous formulae, this evaporation formula is designed for ACO on TSP.

In an effort to combat stagnation, which can be a result of large amounts of pheromone accumulating on certain edges and preventing other edges being explored, $\mathcal{MM}$AS uses a pheromone clamping mechanic, imposing a maximum and minimum pheromone value. These values, $\tau_{min}$ and $\tau_{max}$, are defined as:

$$\tau_{\text{max}} = \frac{1}{pC_{\text{best}}} \tag{3.5}$$

$$\tau_{\text{min}} = \tau_{\text{max}}\frac{2(1-a)}{a(n_{\text{neighbours}}+1)} \tag{3.6}$$

where $n_{\text{neighbours}}$ is the number of nearest neighbours in the candidate set,$a$ is an MMAS constant value defined as $a = exp(log(0.05)/n)$, and $pC_{\text{best}}$ is the solution quality of an initial greedy tour.

Once evaporation has taken place, any values higher than $\tau_{max}$ are set to $\tau_{max}$, and values lower than $\tau_{min}$ are set to $\tau_{min}$.

## 3.5 Determine Best Solution

If the current iteration $I$ is less than or equal to the pre-determined number of iterations $I_(max)$, the Solution Construction Phase is repeated with the updated pheromone levels determined in the Pheromone Deposit Phase. Otherwise, the best solution is determined. For the Pheromone Deposit Phase, each ant retains the value of the best solution, whether that be tour distance or energy usage, in local memory. This makes it straightforward to loop through every ant to determine which ant has found the best solution. This solution can then be output to the console or saved to a file.

# Chapter 4

# Vectorized Candidate Set Selection for Parallel Ant Colony Optimization

The work presented in this chapter was published in GECCO '18: Proceedings of the Genetic and Evolutionary Computation Conference Companion (Peake, Amos, et al. 2018).

Nearest Neighbour lists, or candidate sets, are frequently used in conjunction with ACO in order to reduce the search space of the given problem, and therefore the execution time. Previous work on vectorised ACO, while featuring the use of candidate sets, made no attempt to vectorise the candidate set construction or loading process, instead focusing on improving solution quality. This chapter presents an MMAS-based ACO implementation for manycore SIMD architectures featuring a candidate set system that makes full use of AVX512 instructions, enabled through a vectorisation compatible construction phase in which novel Nearest Neighbour data structures are constructed, which can then be efficiently loaded into the selection phase. These techniques were evaluated using the Traveling Salesman Problem (TSP), and experimentation was performed using the manycore Intel Xeon Phi Knight's Landing processor. The results of these experiments show a significant speedup over existing selection methods on the well-known ACOTSP set of problems.

## 4.1 Background and Motivation

Because of the inherently distributed nature of ACO (whereby ants work independently of each other, guided only by a shared global pheromone network), the ACO algorithm presents significant opportunities in terms of its implementation on high-performance parallel hardware (Cecilia, García, Ujaldón, et al. 2011; Cecilia, García, Nisbet, et al. 2013; Dawson and Stewart 2013c; Dawson 2015; Fu, Lei, and Zhou 2010; Cecilia, García, Ujaldón, et al. 2011; Skinderowicz 2016). Performance improvements are made possible by the *vector processing* capabilities of chips such as the Intel® Xeon Phi (Tian et al. 2013; Sodani et al. 2016), which have instructions that operate on one-dimensional *arrays* of data (vectors), rather than on single data items. In the case of Xeon Phi, these *Single Instruction Multiple Data (SIMD)* instructions operate simultaneously on 16 floating point registers.

### 4.1.1 Traveling Salesman Problem

The Traveling Salesman Problem (Lawler 1985) is the selected problem domain for demonstrating the techniques described in this chapter. The problem is typically used when benchmarking new algorithmic changes to ACO due to its wide use in the literature, with it being used for the initial demonstration of ACO (Colorni, Dorigo, Maniezzo, et al. 1991), as well as the ACS (Dorigo and Gambardella 1997a) and $\mathcal{MM}$AS (Stützle and Hoos 2000) variants, and many other techniques since then. TSP also has an established set of benchmarking problems, TSPLIB (Reinelt 1991), the wide use of which makes establishing context against other techniques more straightforward in cases where recreating the technique is not viable. TSP also has a number of "challenge" instances, typically very large instances where the optimal tour will reveal an image such as the 100,000 to 200,000 city Art TSPs (*TSP Art Instances* n.d.), or is based on real-world geography such as the 1,904,711 city World TSP (*World TSP* n.d.). In recent years a set of 3D challenge TSPs, based on the positions of stars, has been created (*Star Tours* n.d.). The wealth of available problem instances and the historical connection with ACO make TSP an obvious choice for the demonstration of the technique described in this chapter. The TSP is a well-known NP-hard combinatorial optimisation problem, in which the shortest "tour" , between a given number of "cities" needs to be found. Each city has associated co-ordinates, allowing the problem to be modelled as an undirected (in the case of the symmetric TSP problem) weighted graph $G = (N, E)$ with $N$ being the set of nodes, or vertices, that represent the cities and $E$ being the set of edges, representing the paths between the cities. Each edge $(i, j) \in E$ is assigned a distance value $d_{ij}$, representing the distance between cities $i$ and $j$. In the symmetric TSP, $d_{ij}$ is always equivalent to $d_{ji}$. The goal of the TSP is to find a minimum length *Hamiltonian* circuit of the graph, with each of the nodes in set $N$ is visited exactly once. An example of a TSP provided by the TSPLIB package, pr1002, is displayed in Figure 4.1.

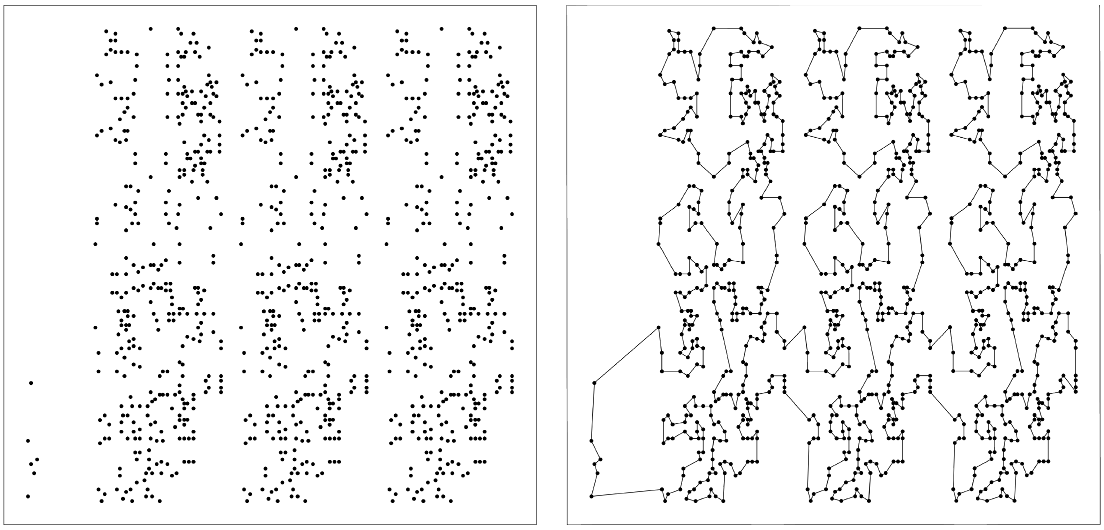


Figure 4.1: A visual representation of the pr1002 TSP problem (L), and the optimal tour for the problem (R)

### 4.1.2 Single Instruction Multiple Data

ACO features many instances of operations being performed within loops, particularly within the selection phase, which leads to a large potential performance gain from vectorisation. As the selection phase is by far the most time-complex area of the ACO algorithm, any speedup achieved will have a significant impact on the overall execution time of the algorithm. A particularly important AVX-512 instruction is the *mask_mov* instruction, which moves the contents of one vector to another vector if the corresponding lane in a given mask vector is set to true - if the lane is false, the corresponding value from a second vector is moved instead. This is particularly useful when checking the *tabu list*, a data structure which indicates whether a node has been visited in the current solution. Rather than checking every node individually against the tabu list when deciding if a move to that node is valid, the use of the masked move instruction allows for the checking of 16 nodes at a time against the tabu list. This allows for the selection phase to consider 16 nodes simultaneously (with 16 being the maximum vector width available in AVX-512), rather than each node individually. Another useful instruction is *cmp*, which allows for a comparison between two vectors and generates a mask vector based on that comparison. This functionality can be used when attempting to find the largest weight when the next node is being selected: two vectors, the previous 16 best nodes and 16 best nodes of the current loop, are compared using the *cmp* instruction, with the generated mask (indicating if the value from the new vector is greater than the value of the current vector) being used with the previously mentioned masked move instruction, with values from the new vector being moved to the results vector if they are greater than the values in the current vector, and values from the current vector being moved if they are not. These are just two examples of how useful AVX-512 instructions can be in reducing execution time, but many more are used throughout the code to essentially reduce what would be 16 individual processes to a single process.

While the AVX-512 instruction set is a significant feature of the Knights Landing processor, another key feature is the thread count. Much like the predecessor architecture Knights Corner, Knights Landing features 4-way hyperthreading, allowing for 4 logical threads per CPU core. Combined with the high number of cores (ranging from 64-72), Knights Landing processors can feature as many as 288 threads. In combination with the AVX-512 instructions allowing for 16-wide vectors of single-precision floats, KNL processors can hypothetically perform up to 1152 processes simultaneously, with a peak double-precision compute speed of up to 3456 GFLOPS. For an algorithm like ACO that has many processes running simultaneously, this parallelisation capacity can be very useful.

### 4.1.3 Selection Methods

The *independent-roulette* technique (*iRoulette*) was introduced by Cecilia *et al.* (Cecilia, García, Ujaldón, et al. 2011) as a parallel alternative to the traditional roulette-wheel selection method commonly used in sequential ACO algorithms. Roulette wheel selection is used whenever an ant must choose the next edge to traverse (and, thus, the next city to visit), with the probability of an edge being selected being proportional to its pheromone concentration. Although this is straightforward in the sequential algorithm, control flow and synchronisation issues mean that it is more challenging in a parallel setting. Dawson and Stewart subsequently proposed an alternative *double-spin roulette* (DSRoulette) technique (Dawson and Stewart 2013c). For an in-depth analysis of the properties of *iRoulette*, see (Lloyd and Amos 2017). F With the availability of the Xeon Phi came new variants of *iRoulette*, due to the potential for vectorisation offered by the its Vector Processing Unit (VPU). The algorithm described in (Lloyd and Amos 2016) (referred to as *vRoulette-1*) is one example of this; the basic principles remain the same, but this variant makes use of intrinsic instructions available on the

Xeon Phi to vectorise the *iRoulette* process, which yields improved performance over the original method (as well as over a vectorised version of the DSRoulette algorithm). Along with their *vRoulette-1* method, Lloyd and Amos (Lloyd and Amos 2016) utilised nearest neighbour information in their scheme for selecting cities. However, in this implementation, the candidate lists were used only to improve solution *quality*, and did not yield any speedup. Vectorised Candidate Set Selection (VCSS) is an amended version of the algorithm described in (Lloyd and Amos 2016), which replaces *vRoulette-1* with a properly vectorised nearest neighbour list. VCSS brings significant performance benefits in terms of execution time, especially with larger problem instances.

## 4.2 Proposed Algorithm

The key contribution in this work is a vectorised algorithm (and associated data structure) to accelerate vertex selection using candidate sets (nearest neighbour lists). The selection procedure is modified (compared to previous versions) so that only vertices in the nearest neighbour list for the current vertex are considered. Only in cases where all of these are *tabu* will the remainder of the feasible region be considered. Typically, the nearest neighbour maximum list length is set to $\sim$20; for large instances (with thousands of vertices) this can speed up the selection process significantly by reducing the number of evaluations made by the ant - for a TSP with 1000 cities, each ant would typically have to evaluate 1000 cities on each of the 1000 steps of the tour construction process - with a nearest neighbour list of size 20, each ant instead evaluates 20 cities on each of the 1000 steps of the process, reducing the total evaluations made by the ant from 1,000,000 to 20,000 (not considering the occasional fallback to a non-nearest neighbour city, which happens infrequently).

The algorithm implementation is based on the Xeon Phi code described in (Lloyd and Amos 2016). The code has been ported to use the *AVX512* vector instruction set rather than the IMCI data set, which as previously mentioned is only available on the Knights Corner architecture. The use of AVX-512 makes the algorithm more generally applicable than the IMCI variant, as while AVX-512 is currently largely exclusive to high-end processors, it will gradually be introduced to more entry-level processors in the next few years.

The main focus of this algorithm is to replace the standard selection process, which loops through every single potential city that an ant can visit next during their solution construction, and instead only consider nearest neighbours. While this would be relatively trivial in a sequential implementation, the use of AVX instructions introduces certain limitations. The AVX *load* instruction can only load in a 16-wide block of a memory structure, rather than loading 16 values from various locations in an array. This prevents simply loading the pheromone and weight values from the index of a given city's nearest neighbours, and instead necessitates the use of a novel Nearest Neighbour object which denotes which 16-wide block of the pheromone matrix relevant to one or more nearest neighbour. The algorithm itself remains largely similar to $\mathcal{MM}$AS elsewhere, with slight changes needed in places such as the selection phase and pheromone distribution phase to make use of the new Nearest Neighbour structure.

### 4.2.1 Nearest neighbour List Construction

During the *setup* phase of the algorithm, the distance matrix (an $n \times n$ matrix encoding the edge lengths of the complete TSP graph) is used to calculate a vectorised nearest neighbour list data structure. This is performed as follows:

Let the number of nearest neighbours be $N_{nn}$, and the width of a SIMD vector (in floats) be $p$. Then let

$$N_p = \lceil N_{nn}/p \rceil,$$

the maximum number of SIMD vectors required to store one line of the nearest neighbour data structure. The data structure then comprises an array (while other data structures may allow for faster searches, C++ arrays are directly compatibile with AVX512 instructions) of up to $N_p$ *Nearestneighbour* objects (one per vertex) with each *Nearestneighbour* entry containing an integer index `ivec` and a $p$-wide bitmap `mask`. To add a vertex $j$ to the nearest neighbour list, it must first be ensured that there exists an entry with `ivec` $= \lfloor j/p \rfloor$, and set the bit in `mask` corresponding to $j \mod p$.

The data structure for a vertex is filled as follows: first, the other vertices are sorted by distance, and the first $N_{nn}$ of these are processed. For each of these vertices, `ivec` is determined by dividing the index of the city by the vector width (which C++ then floors to the integer value). The remainded of this division is also calculated, as it represents the vector lane of the city. If a *Nearestneighbour* entry already exists for this value of `ivec`, the appropriate bit (indicated by the remainder) is set in `mask`. If not, a new *Nearestneighbour* is added to the end of the list, and the appropriate bit set in `mask`. The combination of the `ivec` value and the vector lane can be used to determine the index of the stored nearest neighbour cities - `ivec` multiplied by 16, added to the vector lane of the nearest neighbour bit. For example, a nearest neighbour object with an `ivec` of 2 contains cities with indexes between 32 and 47 - if the third vector lane is set to 1, this indicates that the city with the index of 34 is a nearest neighbour. The data structure is illustrated in Figure 4.2, for a vector width of 16 (the width used in AVX-512), and an example is illustrated in Figure 4.3.

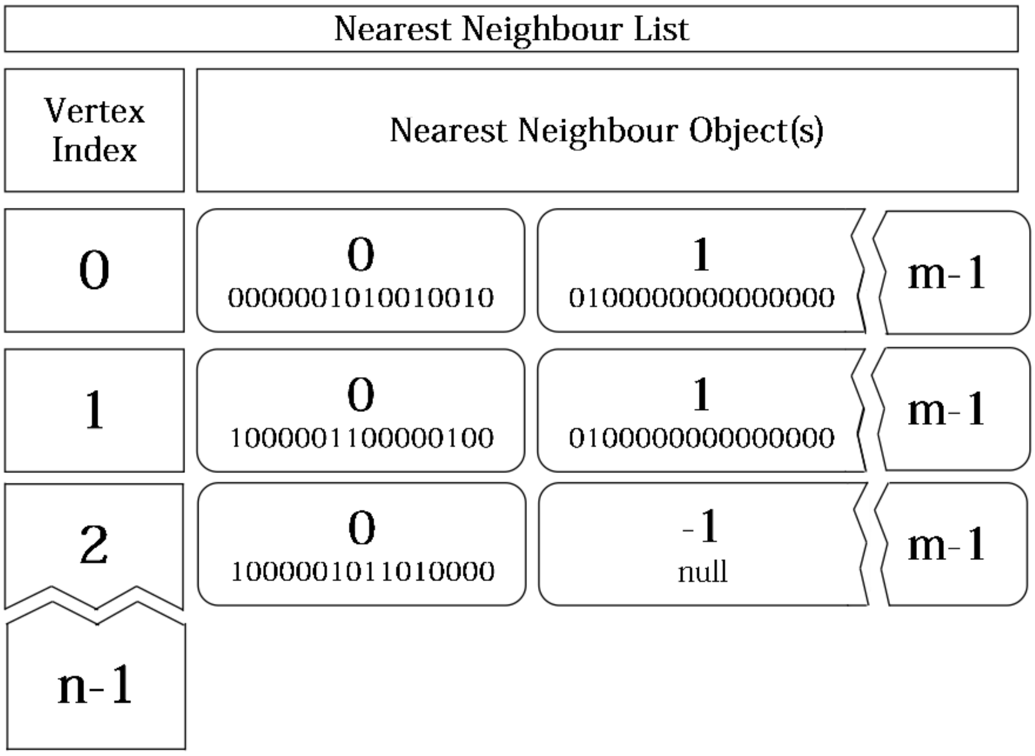


Figure 4.2: Nearest neighbour data structure, with each vertex having an associated array of nearest neighbour objects containing a vector index `ivec` and a bit mask. A sentinel value of `ivec` $= -1$ is used to indicate the end a line in the data structure. $n$ is the number of vertices and $n_{16}$ is the maximum number of entries for a vertex (for 16-wide vectors, the width used in AVX-512)

### 4.2.2 Instance Preprocessing

A potential performance problem is caused by the distribution of nearest neighbours in the problem instance. The relative proximity of vertices in space is not necessarily correlated with the vertex indices (that is, two vertices that are spatially adjacent might have indices that are widely separated, and vice versa; see Figure 4.5). If the indices of nearest neighbours tend to be close together, the nearest neighbour list can be kept short. Conversely, in the worst case, the nearest neighbour list will contain the full set

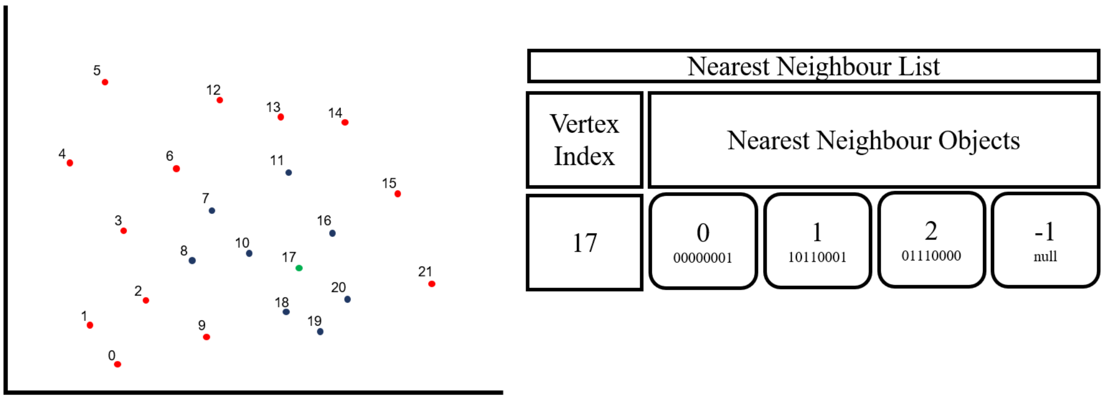


Figure 4.3: An example of the nearest neighbour list structure (with a vector width of 8 in this example) created for vertex 17 on the TSP on the left - the dark blue cities are nearest neighbours, so need to be included in the nearest neighbour list. The three objects contained in the list for vertex 17 have $ivec$ values of 0 (representing vertices 0-7), 1 (representing vertices 9-16), 2 (representing 17-24) and -1 (indicating the end of the list for vertex 17). The bit masks are set to represent the specific nearest neighbour cities, with 0 indicating that the city is not a nearest neighbour (i.e. cities 9, 12, 13, 14 and 15 for the second NN object) and 1 indicating that the city is a nearest neighbour (i.e. city 7 for the first NN object)

of $N_p$ entries. In order to keep the size of the nearest neighbour list relatively low, the problem instance is pre-processed before constructing the nearest neighbour list, by sorting the vertices into greedy tour order. This is done with the use of a basic greedy algorithm that starts at the first city given in the TSP input file and moves to the nearest city; this process repeats, with the nearest available city being selected on every step until the TSP is fully traversed. The indexes associated with each city in the TSP input file are then changed to reflect this tour order (i.e. if the tour proceeds as city 1 - city 5 - city 8 etc, city 5 and city 8 will be relabeled as city 2 and city 3.). This increases the likelihood that cities in close proximity to each other will be in the same vector index, reducing the size of the nearest neighbour object list.

### 4.2.3 Tour Construction

OpenMP is used to assign each ant's tour construction process to a single thread. As no updates are made to any of the values used by the ants until the end of an iteration, and ants only write to their own local memory, no synchronisation is required. Once a starting city has been randomly selected for an ant, it begins the Tour Construction phase. At each stage of tour construction, the ant evaluates which city to visit next (and

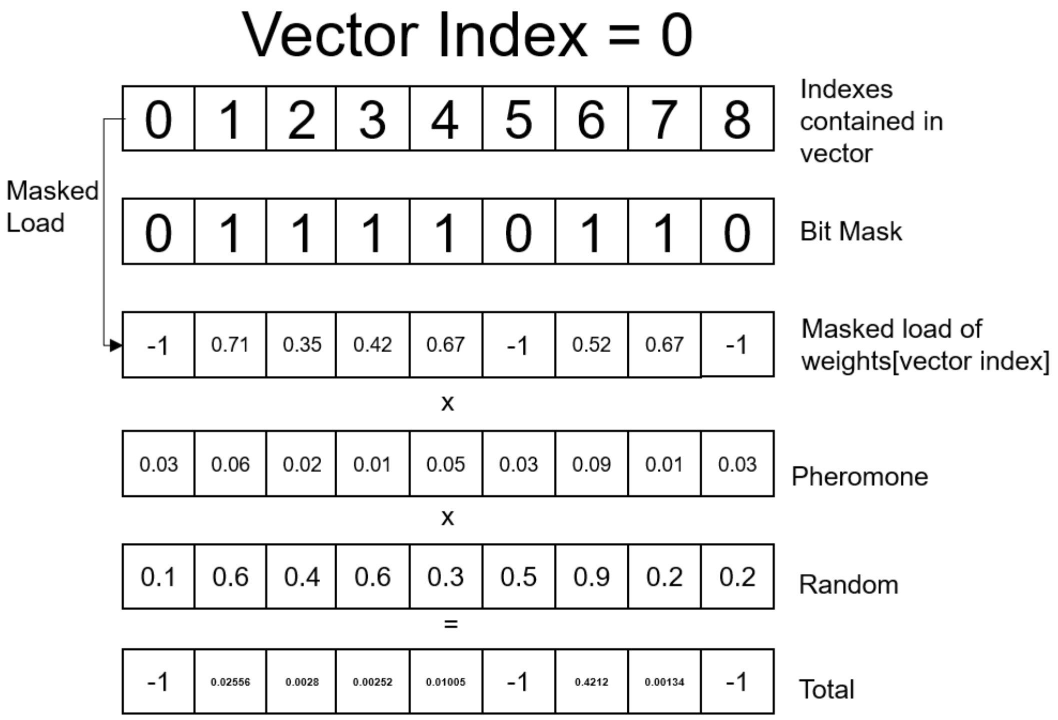


Figure 4.4: Masked load process using the nearest neighbour list to retrieve the weights of nearest neighbour vertices. The AVX2 vector width of 8 is used in this example.

hence the next edge in the graph to traverse) based on a combination of heuristic and pheromone information, only considering cities that have not already been visited by that ant in its current tour (the ant maintains a tabu list in local memory to keep track of visited cities). To make this decision, it used an edge selection function, which is called repeatedly to determine the ant's path around the graph. In the experiments described below, two vectorised edge selection functions are evaluated: *vRoulette-1* (which examines *every* vertex in the feasible region) and *Vectorised Candidate Set Selection* (*VCSS*), the new vectorised procedure, which uses nearest neighbour information.

## 4.2.4 Random Number Generation

As well as heuristic and pheromone information, random numbers are also utilised when an ant is selecting the next city to visit. Seeds for the random number generator used in this implementation are themselves generated by a RANLUX generator (Shchur and Butera 1998), seeded by an input parameter. Through the use of the RANLUX generator 16 unrelated seeds can be generated using a single input seed, which

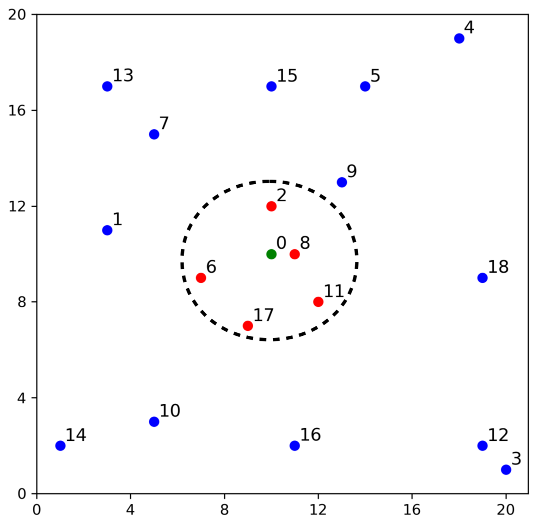


Figure 4.5: Sample TSP graph, with the current city (labelled 0) in the center, and five nearest neighbour cities highlighted in the dashed containing region.

are then used to seed 16 random number generators, one for each vector lane. The random number generator itself is a linear congruential generator (Thomson 1958), with generated seed values each being multiplied 1664525 and then added to 1013904223, the widely-used *ranqd1* values given in *Numerical Recipes* (Teukolsky et al. 1992). This generates a pseudo random float between 0 and 1. Through the use of AVX-512 add and multiply instructions, these numbers can be generated 16 at a time and then stored in a 16-wide vector. This is done by filling 3 AVX vectors, one vector with the generated seeds from RANLUX using the *load* instruction, one vector with 1664525 (C0) and one vector with 1013904223 (C1), both using the *set1* instruction. The seed vector is then multiplied by C0 using the *multiply* instruction, and the resulting vector is then added to C1 with the *add* instruction. The resulting vector has its values converted to a float between 0 and 1 using the *convert* instruction to convert the integers into floats, with the resulting vector being multiplied by a vector containing 2.3283064e-10f in order to bring the value of those floats between 0 and 1. The main benefit of using this configuration of linear congruential generator is the speed, as it only uses a single addition and multiplication operation, though the conversion from integer to float does add further operations. The fused multiply-add instruction introduced in the AVX-512 4FMAPS extension would improve execution time further, but Knights Landing does not feature this extension. The seeds themselves are only generated once by the RANLUX generator, meaning the runtime impact of seed generation is minimal.

### 4.2.5 Vectorised Candidate Set Selection (VCSS)

*Vectorised Candidate Set Selection*, based on *iRoulette* (Cecilia, García, Ujaldón, et al. 2011), selects from candidate set drawn from the nearest neighbour list data structure. If this fails to produce a selection (which will only happen when all the nearest neighbours have already been visited in the tour), the *vRoulette-1* procedure is used to select a vertex from the remaining feasible region.

*VCSS* takes the tabu list, nearest neighbour list and an array of weights and then proceeds as follows (assuming a vector width $p$): first, $p$-wide vectors representing the running maximum weight and corresponding vertex indices are initialised. The algorithm then iterates over the nearest neighbour list. For each *Nearestneighbour* object, the edge weights for the corresponding vertices are loaded as a single $p$-wide vector. The bitmask in the *Nearestneighbour* object is used to mask this weight vector such that only the vertices in the nearest neighbour list remain (illustrated in Figure 4.4).

A vector of consecutive integers is also constructed, corresponding to the vertex indices in the vector. The weight vector is multiplied by a $p$-wide vector of random deviates (produced using a simple linear congruential generator). The modified weight vector is compared, on an element-wise basis (in a single instruction), with the running maximum. This produces a bit mask which is used to update both the running maximum and the corresponding index vector. A reduction is then performed over the vector lanes to produce the maximum weight and corresponding index (in the VCSS implementation, this reduction is performed in $\log_2 p$ steps using bit-swizzling instructions). This reduction is described further in the next section. If no edge has been selected, the *vRoulette-1* algorithm is used by default on the complete set of weights.

The algorithm is formally described in Algorithm 2. Here, *Random()* is a function which returns a $p$-wide vector of uniform deviates, and *ApplyMask(mask, a, b)* is a function which returns a vector filled with values from *a* in positions where the corresponding value of *mask* is set, and values from *b* where the *mask* value is not set.

### 4.2.6 Parallel Reduction

In order to find the greatest weight value in the results vector, a reduction is performed, setting all lanes of the vector (and the corresponding vector of indexes) to the value of the highest weight. While with more modern data structures, this would merely be a case of using a *max()* function call or something similar, AVX vectors require manual

**Algorithm 2:** Pseudo-code for Vectorised Candidate Set Selection.

```
Input  : Edge Weight array $\mathbf{W}_{0...N-1}$, Tabu Mask array $\mathbf{T}_{0...n-1}$, Maximum
         number of nearest neighbours $N_p$, nearest neighbour list $L_{0...N_p-1}$
Output: Selected Edge
// Variables in bold are p-vectors, superscripts indicate vector lanes
$\mathbf{W}_{max} = (0...0)$;
$\mathbf{I}_{max} = (0...0)$;
for i = 0 to $N_p - 1$ do
    if $L[i]$.ivec $\neq -1$ then
        $\mathbf{R} = Random()$;
        $\mathbf{V} = L[i]$.mask;
        $\mathbf{I} = (pL[i]\texttt{.ivec}...pL[i]\texttt{.ivec} + p - 1)$;
        $\mathbf{w} = ApplyMask(V_i, \mathbf{W}_i \times \mathbf{R}, (-1... -1))$;
        $\mathbf{w} = ApplyMask(T_i, (-1...1), \mathbf{w})$;
        $max\_mask = \mathbf{w} > \mathbf{W}_{max}$;
        $\mathbf{I}_{max} = ApplyMask(max\_mask, \mathbf{w}, \mathbf{W}_{max})$;
    end
end
//Reduction
j = argmax($\mathbf{W}_{max}$);
return $\mathbf{I}^j_{max}$;
```

comparison between each vector lane in order to determine the max value. Fortunately, the AVX-512 instruction set contains several instructions which allow for simple "shuffling" of vectors, allowing for comparisons using the previously mentioned *mask* instruction between the original vector and shuffled vector in order to find the greatest value in each lane. The benefit of using this procedure rather than simply looping through the vector to find the highest value is the reduction in complexity - $\log 2n$ rather than $n$. This reduction can be demonstrated by considering the number of steps required to perform the parallel reduciton process, visualised for an 8 wide vector in Figure 4.6 - looping and comparing each value would require 8 evaluations (comparing each value to the smallest-so-far), while the parallel reduction method requires 3 evaluations (comparisons of swizzled vectors), with 3 being equal to $\log 2(8)$. For a 16-wide vector, 16 evaluations would be required to loop and compare each value, while only a single additional comparison would be necessary for the parallel reduction method, for a total of 4 evaluations - $\log 2(16)$. While technically 16 values are being compared with 16 other values in each of these evaluations, AVX512 instructions allows for this to happen simultaneously, making it equivalent in computation to a single function evaluation.

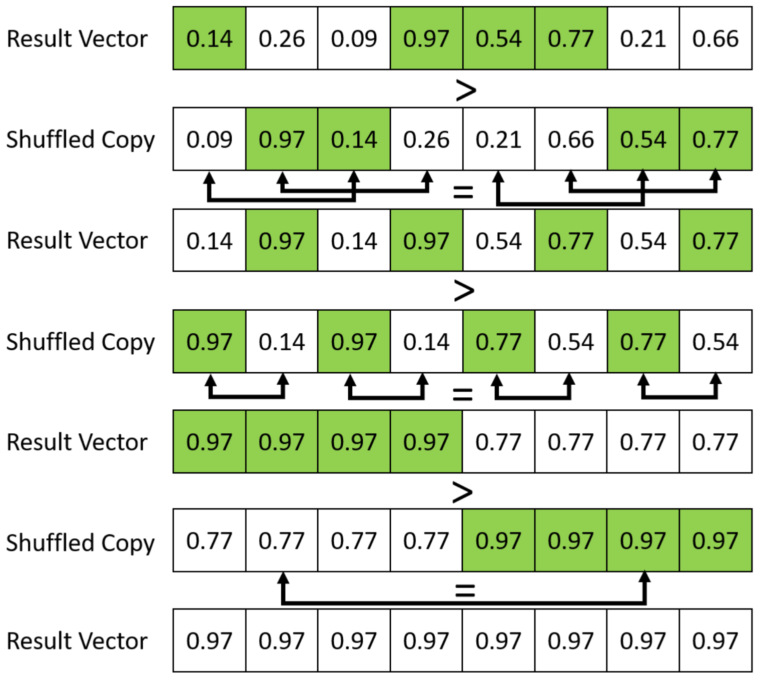


Figure 4.6: A diagram of the parallel reduction technique used to determine the maximum weight in the results vector. This is shown on an 8-wide vector and therefore features one less step than the 16-wide vector implementation.

The first step is to copy the original result vector, and then to perform a *swizzle* transformation. The AVX-512 swizzle instruction shuffles values within blocks of 4 floats, the order of which is determined by a given parameter (in this case, CDAB, which swaps float 1 with float 3 and float 2 with float 4). The original vector is then compared with this shuffled vector using the *mask* instruction, and this mask then used with the *mask_mov* instruction to create a combined vector consisting of the greatest value in each comparison. The same mask is also used to combine the original index vector with a shuffled copy, shuffled in the same way as the results vector.

The resulting vector now contains the 8 highest weight values. Another shuffled copy is created, again using the *swizzle* instruction, this time using the BADC parameter which swaps float 1 with float 2 and float 3 with float 4. The two vectors are again compared, and a mask is created. The mask is then used with the *mask_mov* method to combine the two vectors, with the resulting vector containing the 4 highest weight values. The same operations are carried on on the corresponding index vector.

For the next phase, the *swizzle* instruction is replace by the *permute4f128* instruction, which exchanges blocks of 4 floats rather than exchanging within the float. Like the *swizzle* instruction, a parameter is used to determine the order of the exchanges,

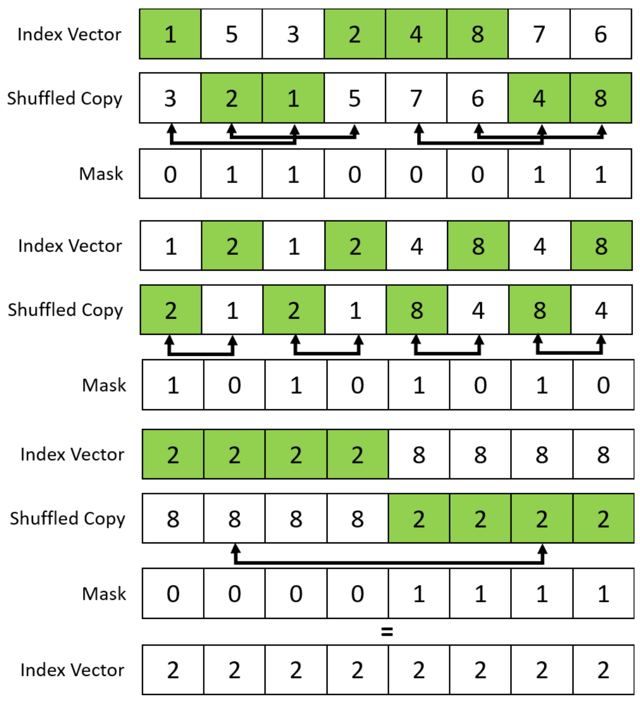


Figure 4.7: A diagram of the parallel reduction technique used to determine the index of the maximum weight determined in Figure 4.6. This is shown on an 8-wide vector and therefore features one less step than the 16-wide vector implementation. The values of the "mask" vector are determined by the comparison carried out in the weight reduction, the indices are never directly compared to each other, with the "mask" instead determining whether a value is retained or discarded - 0 indicates the retention of the original value, and 1 indicating replacement with the value in the shuffled copy.

in this case 0xB1 which represents exchanging block 1 with block 3 and block 2 with block 4. The mask and combine procedure is carried out once again, and again the operations are also applied to the index vector.

The resulting vector now contains the 2 highest weight values. The *permute4f128* instruction is used once again, this time exchanging block 1 with block 2 and block 3 with block 4. A final comparison is carried out, generating a final mask, and the vectors are combined again into a final results vector, containing the highest weight value in all 16 lanes. Once the procedure has been carried out on the index vector, shown in Figure 4.2.6, the final index vector will contain the index corresponding to the highest weight in all 16 lanes.

### 4.2.7 Pheromone Update

The pheromone update process is split into four phases: Deposit, evaporation, clamping and edge probability calculation. The deposit phase is carried out by a single thread, with little potential for vectorisation, but subsequent phases make further use of the AVX-512 vector instruction set.

The remaining phases are carried out in a pair of nested loops, with the outer loop, which cycles through each city individually, being parallelised with OpenMP. The inner loop, which cycles through the pheromone values between the outer loop's city and every other city, iterates over the pheromone matrix 16 values at a time. The first step is to perform the evaporation, which is straightforward. 16 pheromone values are loaded into an AVX vector and are then multiplied by another AVX vector using the *multiply* instruction, with every lane in the second vector set to $1 - \rho$ using the *set1* instruction. This evaporates all 16 pheromone values simultaneously. For the clamping phase each value is defined to be consistent with the clamping values defined in , the maximum value is defined as (Lloyd and Amos 2016).

$$\tau_{max} = \frac{1}{\rho D_s}, \tag{4.1}$$

with $D_s$ equalling distance of the shortest tour in the current iteration, and the minimum value is defined as the maximum value multiplied by the MMAS constant value, which itself is defined as

$$\tau_{min} = \tau_{max} \frac{1 - a}{0.5(n + 1)a}, \tag{4.2}$$

where

$$a = \exp(\log 0.05)/N_v), \tag{4.3}$$

$N_v$ representing the number of vertices in the current instance. These values are then loaded into AVX arrays, with the *max* and *min* instructions used. If a pheromone value is higher than $\tau_{max}$, it is set to $\tau_{max}$; if a pheromone value is lower than $\tau_{min}$, it is set to $\tau_{min}$.

The edge probability calculation is also vectorised, with the pheromone AVX vector being multiplied by an AVX vector containing the inverse of the distance between the outer loop city and the 16 cities that correspond to the section of the pheromone matrix that is currently stored in the pheromone vector. Once the multiplication is complete, both the pheromone AVX vector and weight AVX vector are stored in their respective matrices.

## 4.3 Experimental Evaluation

The aim of these experiments is to measure both the solution quality of the VCSS algorithm, and more crucially to measure the execution time. Execution time will be the main focus, as the objective of the algorithm itself is to use the novel Candidate Set structure to reduce execution time as much as possible, but it is also important to prove that in doing so the solution quality is not negatively affected at all. As there is no major competitor to this algorithm in the literature currently, comparison will instead focus on *vRoulette*, the precursor to VCSS. As the main difference between the two algorithms is the Nearest Neighbour list structure and the associated changes made to the code to accomodate it, this is a useful comparison to make in order to truly highlight the difference that VCSS makes versus a version of the algorithm which makes only limited use of candidate sets. The base ACOTSP code will also be used for comparison, running sequentially, in order to demonstrate the difference that parallelisation and vectorisation can make to the ACO algorithm.

### 4.3.1 Experimental Environment

Three variants of ACO are compared: the first is CPU reference code, the widely used ACOTSP implementation of ACO (Stützle 2004) a sequential implementation of $\mathcal{MM}$AS the second uses the previously discussed *vRoulette-1* implementation; and the third uses the *VCSS* method. Experiments were carried out on a machine with an Intel Xeon Phi 7210 processor with 64 cores and 4 threads per core (for a total of 256 threads), running at a base speed of 1.3GHz. The code was compiled with the Intel® C++ compiler (`icc`) at -O3 optimiation level. The code was run under Linux, with timings obtained using the *gettimeofday()* function. The CPU reference code used is ACOTSP version $1.03$ (Stützle 2004), compiled with `gcc` (with optimisation -O3) and run on a Linux machine containing an 8-core Intel Xeon E5-2650 v2 at a base speed of 2.6GHz.

### 4.3.2 ACO Parameters and Problem Instances

The number of Nearest Neighbours is set to 32, which is both in line with Dorigo and Stützle's recommended list size (Dorigo, Birattari, and Stutzle 2006), and a convenient power of two for the purposes of data alignment. The values of the $\mathcal{MM}$AS parameters used, based on the recommended values given in (Dorigo, Birattari, and Stutzle

2006; López-Ibáñez, Stützle, and Dorigo 2016) are as follows: $\alpha = 1, \beta = 2, \rho = 0.02$ In all cases, the number of ants is set to 256 (so that all available threads are used when assigning ants to threads). ACOTSP was run with local search switched off, as neither VCSS nor vRoulette-1 make use of local search techniques.

The problem instances used in these experiments are taken from the TSPLIB library(Reinelt 1991), and include all instances solved in (Lloyd and Amos 2016) and (Tirado, Urrutia, and Barrientos 2015). Also included are larger instances, in order to investigate the performance of the algorithm as the problem size increases. The instances used are: `lin318`, `pcb442`, `rat783`, `pr1002`, `fl1577`, `pr2392`, `fl3795`, `rl5934`, `pla7397`, and `rl11849`, all of which are symmetric TSP problems with the number of cities specified in the problem name. For each instance, 50 runs of 1024 iterations are performed, with each run using a different seed.

### 4.3.3 Execution Time

Execution time is measured on a per-iteration basis, which is calculated as the mean of the execution times for all 50 runs on the instance divided by the number of iterations, in this case 1024. Results are shown in Figure 4.8, which (log) plots the mean time per iteration over all instances for *VCSS* and *vRoulette-1* on the Xeon Phi, and ACOTSP on CPU, and Table 4.1, which gives the numerical values, along with the speedup.

While *vRoulette-1* and *VCSS* have similar execution times on the smaller instances, as the instance size grows, the execution time for *VCSS* grows more slowly than that of *vRoulette-1*. For the largest instance, `rl11849`, *VCSS* is faster than *vRoulette-1* by an order of magnitude, and is faster than the reference code by two orders of magnitude. On the other hand, *vRoulette-1*, while performing two orders of magnitude faster than the reference code on smaller instances, shows a declining speed-up compared to the CPU as the instance size grows.

### 4.3.4 Solution Quality

In Figure 4.9 box plots of solution quality of each algorithm are shown (where solution quality is measured as the ratio of the length of the best tour found to the known optimum for the problem instance). Differences between the solution quality obtained with the CPU code and the two Xeon Phi variants are expected due to the modified selection probabilities in the *iRoulette* scheme compared with those in the roulette wheel selection used by ACOTSP. It is already known that *iRoulette* can affect the solution

Table 4.1: Execution time per iteration in milliseconds, and speedup relative to CPU. Speedup is computed by dividing the VCSS time by the vRoulette-1 time.

| | CPU | *vRoulette-1* | | *VCSS* | |
|---|---|---|---|---|---|
| Instance | $t$/ms | $t$/ms | Speedup | $t$/ms | Speedup |
| `lin318` | 18.1 | 0.73 | 24.8 | 0.52 | 34.8 |
| `pcb442` | 29.2 | 1.37 | 21.3 | 0.74 | 39.5 |
| `rat783` | 68.3 | 5.65 | 12.1 | 1.37 | 49.9 |
| `pr1002` | 94.1 | 9.01 | 10.4 | 1.85 | 50.9 |
| `fl1577` | 176.7 | 20.9 | 8.45 | 3.5 | 50.1 |
| `pr2392` | 371.0 | 46.4 | 8.00 | 7.02 | 52.9 |
| `fl3795` | 785.7 | 143.3 | 5.48 | 13.5 | 58.2 |
| `rl5934` | 2088.9 | 426.8 | 4.90 | 27.9 | 74.9 |
| `pla7397` | 3388.3 | 724.3 | 4.68 | 43.04 | 78.7 |
| `rl11849` | 10578.8 | 1975.0 | 5.36 | 97.47 | 108.5 |

quality on individual instances, although its average behavior does not significantly affect the quality of solution (Lloyd and Amos 2017). There is some variation between the solution qualities obtained using *vRoulette-1* and *VCSS*. It should be noted that this experiment used a relatively small sample of instances, with 50 runs per instance. In order to measure effects on solution quality, a larger sample of instances (with one run per instance) would be better. Additionally, for the larger instances, the number of iterations (1024) is relatively small and the algorithms may not be converged at the point where the experiment is stopped. However, the focus of this chapter is the efficiency measured in terms of time per iteration: in order to investigate any effects on solution quality, more extensive experiments would be required. Given that VCSS is formally equivalent to the nearest-neighbour list algorithms already widely studied in serial ACO, it would not be expected to see large systematic effects on the solution quality arising from the use of *VCSS*, although this will be a topic for further investigation.

### 4.3.5 Discussion

Significant speedups are obtained using the *VCSS* technique. The speedup over *vRoulette-1* grows as the instance size increases. Without the nearest neighbour list, the tour construction process has time complexity $O(n^2)$ (since at each of $n$ vertices, $n-1$ vertices are included in the selection process). The nearest neighbour list reduces this complexity to $O(n)$ (since the workload per vertex is constant, determined only

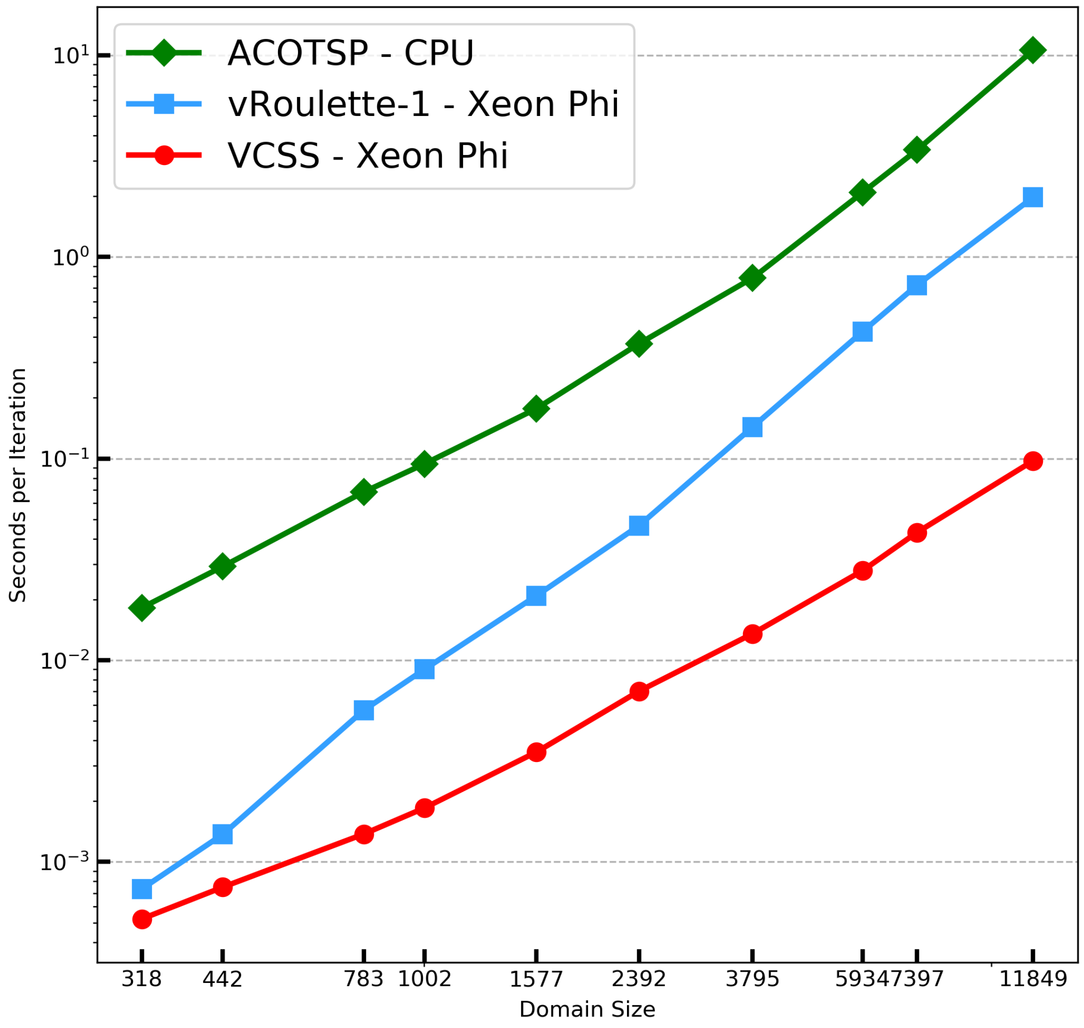


Figure 4.8: Execution times for *ACOTSP*, *vRoulette-1* and *VCSS*.

by the size of the nearest neighbour list), and though the time complexity of constructing the nearest neighbour list in itself is $O(n^2)$, this is a significantly shorter process than the tour construction phase and has little effect on the execution time. The input parameters have no effect on the time complexity of the VCSS selection method. In terms of raw time, there are two factors that are likely to effect execution time: Nearest neighbour list size, and nearest neighbour usage. As previously discussed, the nearest neighbour list object (assuming 32 nearest neighbours and a vector width of 16) has a feasible minimum size of two nearest neighbour objects, and a maximum feasible size of 32 nearest neighbour objects (though the instance pre-processing described previously reduces this maximum to 10). Though the time complexity is still linear, the

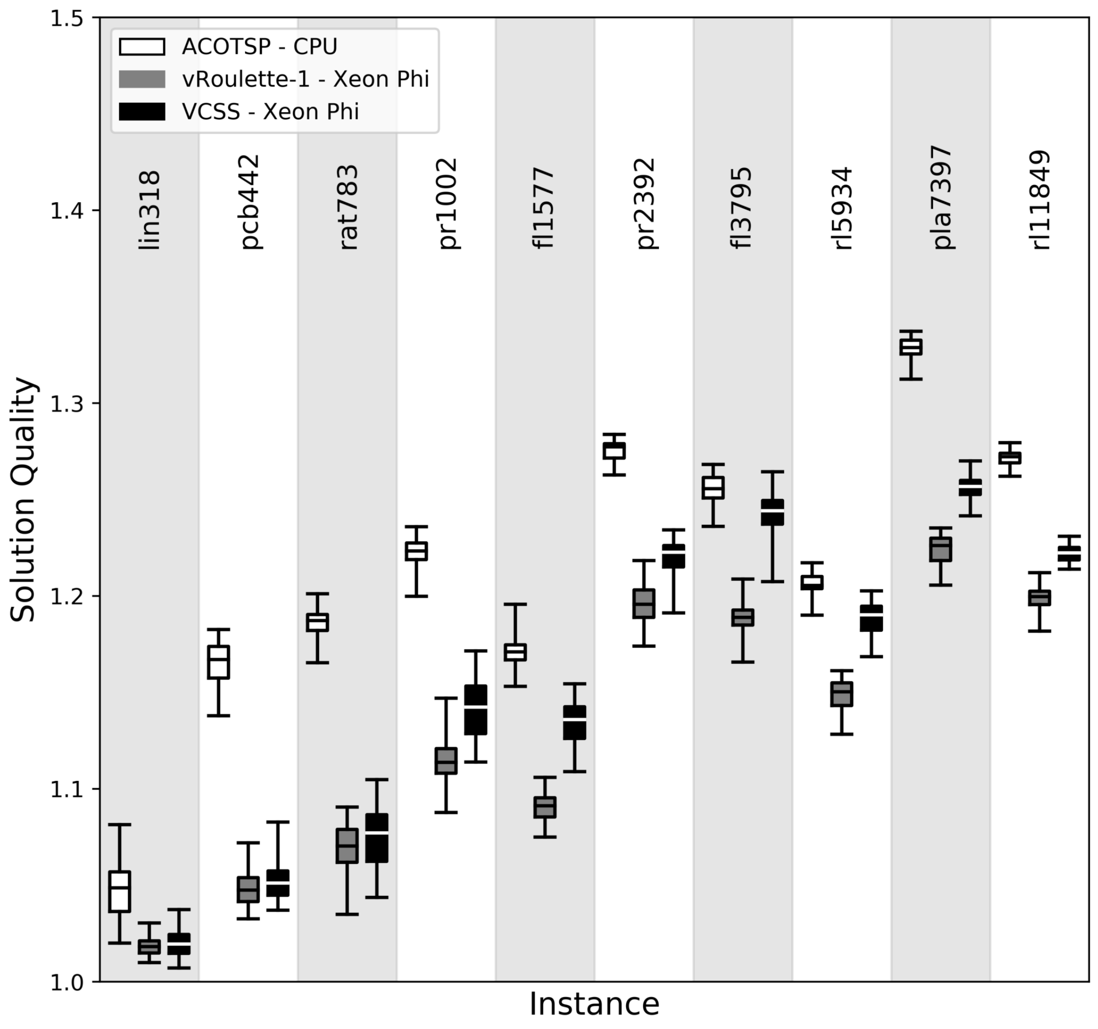


Figure 4.9: Solution quality for *ACOTSP*, *vRoulette-1* and *VCSS*.

difference between a minimum feasible and maximum feasible runtime for the selection method is around $5\times$. In terms of nearest neighbour usage, every time a nearest neighbour is unavailable, a full vRoulette-1 selection is performed, the complexity of which IS affected by the size of the input TSP. Analysis of the results during the running of the algorithm indicated that the nearest neighbour list was selected from at least 97% of the time, with the previously mentioned vRoulette-1 fallback being used in the <3% of scenarios where no nearest neighbour was available, leading to a small portion of the algorithm having $O(n^2)$ time complexity. However, the vast majority of the algorithm remains as a linear time complexity.

The speedup, relative to the CPU code, also increases with the instance size. This demonstrates the advantage of the Nearest Neighbour list structure, with the AVX-512 instructions allowing for multiple nearest neighbours to be considered simultaneously. While the ACOTSP code also has a fixed size nearest neighbour list, it is only able to process them sequentially. As the instance size increases, so does the number of selection phases required to create a full tour, which amplifies the time saving effect of VCSS and the Nearest Neighbour structure when compared to a standard nearest neighbour list. While the use of OpenMP parallelisation and AVX-512 vector instructions would be expected to always provide a significant constant speedup at the very least, the fact that the speedup grows as the instance size grows is a sign of the positive change made by VCSS. In contrast to this, the reduction of speedup in the case of vRoulette-1 demonstrates the severe execution time reduction caused by lack of a nearest neighbour list.

While it has been shown that the performance of *VCSS* improves with increasing instance size, there is a limit to how far this can be pursued. One of the inherent limitations of the general ACO algorithm is its $O(n^2)$ space complexity, due to the need to store a square pheromone matrix, which can be multiplied further due to the general usage of an additiona weight matrix. Around 500MB of memory is required for the largest instance used in these experiments, and to move to the next order of magnitude (a 100,000 city instance) around 37GB would be required. This limitation must be overcome before the speed gains obtained using the latest parallel and vector ACO techniques can be fully exploited on larger instances. In contrast to this, the space complexity of Genetic Algorithms is generally defined by the population size rather than the instance size, leading to linear growth rather than quadratic growth.

*VCSS* may also benefit further from the inclusion of *local search*, which is often used to accelerate convergence (Dorigo and Stützle 2004). While local search *may* be parallelised, there are currently no vectorised algorithms which can utilise the full power of many-core SIMD hardware.

While the focus of this work was to improve execution time rather than solution quality, the difference in solution quality between ACO-TSP and VCSS is noteworthy. The algorithms themselves are essentially identical at all levels aside from the used selection method, with ACOTSP using the traditional roulette wheel selection technique. As vRoulette-1 and VCSS are both based on the I-Roulette technique, these results suggest that I-Roulette leads to better solutions than the Roulette Wheel selection method, which is consistent with the findings of the original I-Roulette paper (Cecilia, García, Nisbet, et al. 2013). Also notable is the solution quality of pr2392, the only clustered TSP of the experiment set. The potential issues with clustered TSPs and distance-based candidate sets were discussed in section 2, but based on these results it seems that the assumption holds true even for a clustered TSP - while it isn't possible to establish a solid trend from Figure 4.9, the fact that pr2392 doesn't show a significant reduction in solution quality indicates that the VCSS technique is also applicable to clustered TSPs.

While TSP was used to demonstrate the effectiveness of the VCSS technique, it could also feasibly to applied to a wide range of problems. Any problem with a format that supports the usage of a candidate set or nearest neighbour list would be solvable with the VCSS technique, ranging from problems that are similar to the TSP such as the Vehicle Routing Problem (VRP) (Toth and Vigo 2002) and Route Inspection Problem (Thimbleby 2003), extending to any problem with a heuristic function such as the Bin-Packing Problem (which items are the closest size to the remaining capacity of a bin?) (Friesen and Langston 1986) and Job-shop Scheduling (Manne 1960). (using Johnson's rule (Cheng and Lin 2009)).

While VCSS performs well on the TSP instances used in these experiments, the instances are all static TSPs. Were the technique to be applied to dynamic TSP or any other dynamic problem, it would struggle in its current state due to the way in which the nearest neighbour list is constructed, at the beginning of execution using the initial form of the TSP instance. While this works well in a static context, the nearest neighbour list would become inaccurate as soon as a node is added or removed from the instance. As new nodes would not feature on any nearest neighbour list, they would likely be left until every nearest neighbour has been travelled to, which would lead to poor solution quality. In order to apply VCSS to dynamic TSP, the nearest neighbour list requires recalculation every time a new city is added to the TSP. Cities being removed from the TSP would cause less of an issue but a mechanism would need to be in place (similar to the $tabu$ list used currently) to prevent travel to cities that are no longer present.

## 4.4 Conclusion

This chapter has discussed VCSS, a technique that introduces a novel Nearest Neighbour structure to vectorised and parallelised $\mathcal{MM}$AS. The technique performs favourably compared to the previous best performing vectorised implementation, vRoulette-1, which used candidate sets only to improve solution quality rather than execution time. While the solution quality is largely the same as vRoulette-1, in

Table 4.2: Table listing the AVX-512 instructions used in this implementation, a brief description of the purpose of the instruction, and the AVX2 and NEON alternatives

| Description | AVX-512 | AVX2 | NEON |
|---|---|---|---|
| Add | _mm512_add_ps | _mm256_add_ps | vaddq_f32 |
| Subtract | _mm512_sub_ps | _mm256_sub_ps | vsubq_f32 |
| Multiply | _mm512_mul_ps | _mm256_mul_ps | vmulq_f32 |
| Load | _mm512_load_ps | _mm256_load_ps | vld4q_f32 |
| Store | _mm512_store_ps | _mm256_store_ps | vst4q_f32 |
| Min | _mm512_min_ps | _mm256_min_ps | vmin_f32 |
| Max | _mm512_max_ps | _mm256_max_ps | vmax_f32 |
| Broadcast Value | _mm512_set1_ps | _mm256_set1_ps | vdupq_n_f32 |
| Masked Load | _mm512_mask_load_ps | _mm256_blendv_ps | vbslq_f32 |
| Int to Mask | _mm512_int2mask | _mm256_setr_ps | vset_lane_f32 |
| Masked Move | _mm512_mask_mov_ps | _mm256_blendv_ps | vbslq_f32 |
| > Mask | _mm512_cmp_ps_mask | _mm256_cmp_ps | vcgt_f32 |
| < Mask | _mm512_cmp_ps_mask | _mm256_cmp_ps | vclt_f32 |
| Float to Int | _mm512_cvt_roundepu32_ps | _mm256_cvtepi32_ps | vcvtnq_u32_f32 |
| Swizzle | _mm512_swizzle_ps | _mm256_permute_ps | vrev64q_f32 |
| Permute Block | _mm512_permute4f128_ps | _mm256_permute2f128_ps | vrev64q_f32 |

terms of execution time speedups of almost 20x can be achieved through use of the VCSS technique. Crucially, this chapter demonstrated that the vectorisation achieved by vRoulette using the IMCI instructions restricted to the Knights Corner architecture can also be achieved using AVX-512, an instruction set with much more general applicability, especially in upcoming generations of Intel and AMD desktop CPUs. The AVX-512 instructions used by VCSS are also fairly standard instructions that have largely been ported from AVX and AVX2, meaning that the same techniques can be implemented in 8-wide vectors on an instruction set that has been included on the majority of desktop CPUs released in the last decade.

While this chapter has focused on AVX-512, the vectorisation techniques are more broadly applicable, being compatible both with AVX2 instructions and ARM's NEON instruction set, designed for ARM's Cortex range of microprocessors. NEON is particularly useful for edge computing, allowing IoT devices to be more capable at doing tasks locally rather than offloading tasks to the cloud. Table 4.2 lists the AVX-512 instructions used in the VCSS implementation along with a brief description of what the instruction does, as well as alternative instructions for both AVX2 and ARM's NEON instructions.

It should be noted that, while all of the listed instructions apply to single-precision floating point values only, integer variants of the instructions exist for each of the instruction sets, with AVX-512 integer instructions also used in the VCSS implementation. Additionally, it is important to note that these instructions may not be strictly equivalent. While many instructions, such as add, subtract and multiply, among others, perform largely identical functions for each instruction set, there are several instructions that work differently, which must be kept in mind if they are to be used: Masked load, which loads and applies the mask in one step for AVX-512, must be reproduced using a separate load instruction for AVX2 and NEON, with the listed instruction only applying the mask; Int to Mask, which is again a single instruction for AVX-512, requires the mask integer to be converted to an integer array for AVX2 and NEON, with that integer array then being used to set every lane of the mask vector individually; Finally, the swizzle and permute functions don't exist for NEON, with the listed reverse function being the closest approximation - the parallel reduction process which uses swizzle and permute would likely need to be altered in order to function correctly with NEON instructions.

# Chapter 5

# Scaling Techniques for Parallel Ant Colony Optimisation on Large Problem Instances

The work presented in this chapter was published in GECCO '19: Proceedings of the Genetic and Evolutionary Computation Conference Companion (Peake, Amos, et al. 2019).

As mentioned in the previous chapter, while the VCSS structure provided significant speedup to ACO for TSP, memory constraints are still a significant issue when it comes to solving larger TSP problems. As problem size increases, so does the memory requirement. The most significant cause of this is the pheromone matrix, and the associated weight matrix, which has to store float data between every city in the TSP. The largest TSP used as a benchmark in the previous chapter, *rl11849*, requires 140398801 float values to be stored in the pheromone matrix, and an identical amount in the weight matrix. In order to allow ACO to efficiently solve larger TSP problems in a reasonable time frame, these matrices need to be replaced by a more efficient data structure. This chapter presents the *Restricted Pheromone Matrix* (RPM), a technique which iterates upon the VCSS technique of the previous chapter. Rather than having pheromone between every single city stored in the pheromone matrix, the RPM instead only stores pheromone between a city and its nearest neighbours. In situations where no nearest neighbours are valid destinations for an ant, two fallback methods are evaluated which allows the ant to consider non-neighbour cities. These techniques lead to a significant reduction in memory requirement for large TSPs, and allow for the very large Art TSP instances to be efficiently solved by a true ACO implementation for the first time.

AVX2 instructions are also utilised to demonstrate that the SIMD techniques used in the previous chapter can also be achieved using a more widely available instruction set than AVX-512.

## 5.1 Background and Motivation

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, fundamental changes to the ACO data structure are required. Previous work in this area has focused on adopting a population-based ACO approach (Guntsch and Middendorf 2002; Chitty 2017). A combination of alternative techniques may be used to enable 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 (Dawson and Stewart 2013a), as discussed in the previous chapter. 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.

Combining candidate sets with a reduced pheromone matrix is an effective approach to increasing the scalability of ACO, 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 ACO 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. The principal contributions of this work 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.

Parallelisation and candidate sets, addressed in the previous section, are both important techniques for improving the scalability of ACO. The third highlighted opportunity for improvement is reducing the memory complexity of ACO. 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) (Guntsch and Middendorf 2002), 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 (Chitty 2017), 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, it is still possible to achieve improved solution qualities whilst retaining the core features of the traditional ACO algorithm.

### 5.1.1 AVX2 SIMD Instructions

While the AVX-512 instructions used to vectorise VCSS in the previous chapter let to significant speedups, the instruction set itself is currently only available on a limited range of hardware. While the instructions are intended to be released on AMD’s Zen 4 architecture, and already features on Intel’s recent Cascade Lake, Copper Lake, Ice Lake and Tiger Lake architectures, it is still yet to feature on a consumer-level CPU. Combined with Intel’s sunsetting of the Xeon Phi line of manycore processors, the general applicability of AVX-512 is currently low. In order to prove the general applicability of the techniques demonstrated by VCSS, the ACO implementation described in this chapter instead makes use of AVX2, an instruction set widely available on most

Intel and AMD CPUs released since 2013 (excluding the Pentium and Celeron lines, which generally do not feature AVX capability).

The most significant difference between AVX2 and AVX-512 is the vector width: AVX2's 256-bit register width allows for 8-wide vectors of single-precision floating point values, compared to the 16-wide vectors made possible by AVX-512. The instructions themselves also differ slightly: while core instructions such as *load*, *store* and arithmetic functions perform identically, certain instructions such as AVX-512's *swizzle* are not present in AVX2. Certain instructions also have syntactical differences. Therefore, certain changes were required:

- The *int2mask* instruction, which converts an integer value to a vector of 0 and 1s corresponding to the binary value of the integer, is not available as a single instruction in AVX2. Instead, the mask integer is loaded into a C++ array by looping from 0 to 8, shifting the integer bits to the right during each loop, and comparing the least significant bit with the integer 1 using the & operator (therefore checking if the LSB is 1 or 0). If the LSB is 1, the corresponding array value to the current loop is set to 1; otherwise the value is set to -1. The *_mm256_setr_ps* instruction is then used to individually set each vector lane to the corresponding value in the array.

- The *masked move* instruction, which moves values from two vectors into a single vector based on a the corresponding lanes in a mask vector, known as *_mm512_mask_mov_ps* for AVX-512, is instead called *_mm256_blendv_ps* in AVX2, though the behaviour is identical (aside from the input parameters for the instruction being in a slightly different order).

- The *swizzle* instruction, *_mm512_swizzle_ps* which exchanges within 4-wide blocks of the 16-wide AVX-512 vector, does not exist in AVX2. However, the *_mm256_permute_ps* AVX2 instruction performs largely the same operation, exchanging within the 4-wide blocks of the 8-wide AVX2 vector. Similarly, the alternative to the AVX512 *_mm512_permute4f128_ps* instruction, which exchanges the 4-wide blocks with other 4-wide blocks, is instead replaced by the *_mm256_permute2f128_ps* instruction, which functions identically but with only 2 blocks to exchange.

- The *masked load* instruction, *_mm512_mask_load_ps*, which loads data from memory if the corresponding lane of a mask vector is set to 1, does not exist as a single instruction in AVX2. The behaviour can be easily replicated by simply loading the whole vector from memory using *_mm256_load_ps*, and then performing a masked move as described previously.

Rather than using the AVX2/AVX512 instructions inline, they are abstracted to a seperate vector class file, allowing for simple switching between AVX-512, AVX2 and sequential versions of the methods. Of the 22 functions in this vector class, only 4 required changes during the creation of the AVX2 variants of the methods, demonstrating the general applicability of the techniques employed by both VCSS and RPM.

## 5.2 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 (Guntsch and Middendorf 2002) and PartialACO (Chitty 2017) focused on removing the pheromone matrix entirely, relying instead on a population of solutions. The key contribution of the work presented in this chapter is the creation of a new, candidate-set based memory structure, the *Restricted Pheromone Matrix* (RPM), 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 pheromone information between the current vertex and its nearest neighbours, as well as other vertices stored in the nearest neighbour structure for efficient vectorisation. Two other structures of the same size also exist: the distance matrix, containing the heuristic information which in this case is the inverse square of the distance between cities, and the weight matrix, containing the weight data calculated by combining pheromone and heuristic information. In a regular ACO implementation, these other matrices would also contain data between every city in a TSP instance, so memory requirement savings made on the reduced pheromone matrix are also applicable to these other matrices. If $n_{NN}$ is the number of nearest neighbours, $n$ is the number of vertices and $v$ is the vector size available on the used hardware, the restricted pheromone matrix requires $n \times n_{NN} \times v$ real numbers, compared to $n^2$ for the full pheromone matrix. For a constant vector width and nearest-neighbour list size (with the recommended nearest neighbour list size, regardless of problem size, being 20 (Dorigo, Birattari, and Stutzle 2006; López-Ibáñez, Stützle, and Dorigo 2016)), the nearest amount that can be effectively utilised by both AVX2 and AVX-512 instructions is 32, which is the value used in all experiments in this chapter), the memory complexity of the proposed algorithm is therefore $O(n)$. This significantly reduces the

Table 5.1: Memory 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 |

memory requirements of ACO, especially on very large instances, as demonstrated in Table 5.1. For a 100,000 city TSP instance, the restricted matrix occupies only $0.26\%$ of the space required by the standard pheromone matrix.

While later phases of the algorithm involving the RPM make used of the Nearest Neighbour list structure described in the previous chapter, the initial construction is fairly straightforward. An empty array of $n \times n_{NN}$ is created, and each position of the array is set as

$$\tau_{ij} = \frac{1}{\rho d_{NN}}$$

where $d_{NN}$ is the length of a greedy tour performed during the loading of the TSP problem file. The associated weight matrix, which has the same dimensions as the RPM, is filled with

$$w_{ij} = \frac{\tau_{ij}}{\eta_{ij}^2}$$

where $\eta_{ij}$ is the heuristic value, in this case the squared inverse of the distance between city *i* and city *j*. This same formula is used to recalculate the weight values after every iteration of the algorithm.

### 5.2.1 Tour Construction

The tour construction phase is parallelised using OpenMP, with each ant being allocated to an available thread. No synchronisation 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 5.1.

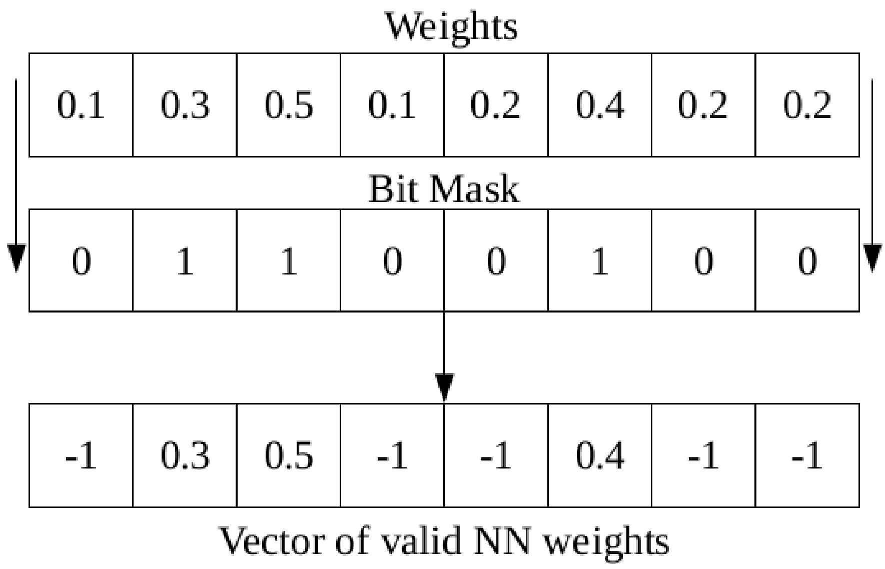


Figure 5.1: Applying the NN mask to filter out non-NN weights

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, generated in the same manner as described in the previous chapter. The randomised 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. A reduction is then performed on the maximum weights vector to find the highest overall weight, again performed in an identical manner as described in the previous chapter. One key difference between this selection method and the method used with VCSS is that the resulting index does not directly correspond to the next city in the tour, but instead points to the entry in the nearest neighbour list which contains the index of the next city to be visited. The index of this city is loaded from the nearest neighbour list, and added to the tour as the next city to be visited. 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, one of the two “fallback” methods described in Sections 5.2.2 and 5.2.3 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. While VCSS falls back to vRoulette-1 when no nearest neighbour vertices are available, here two alternative fallback methods are proposed. The proposed methods are described in Figure 5.2

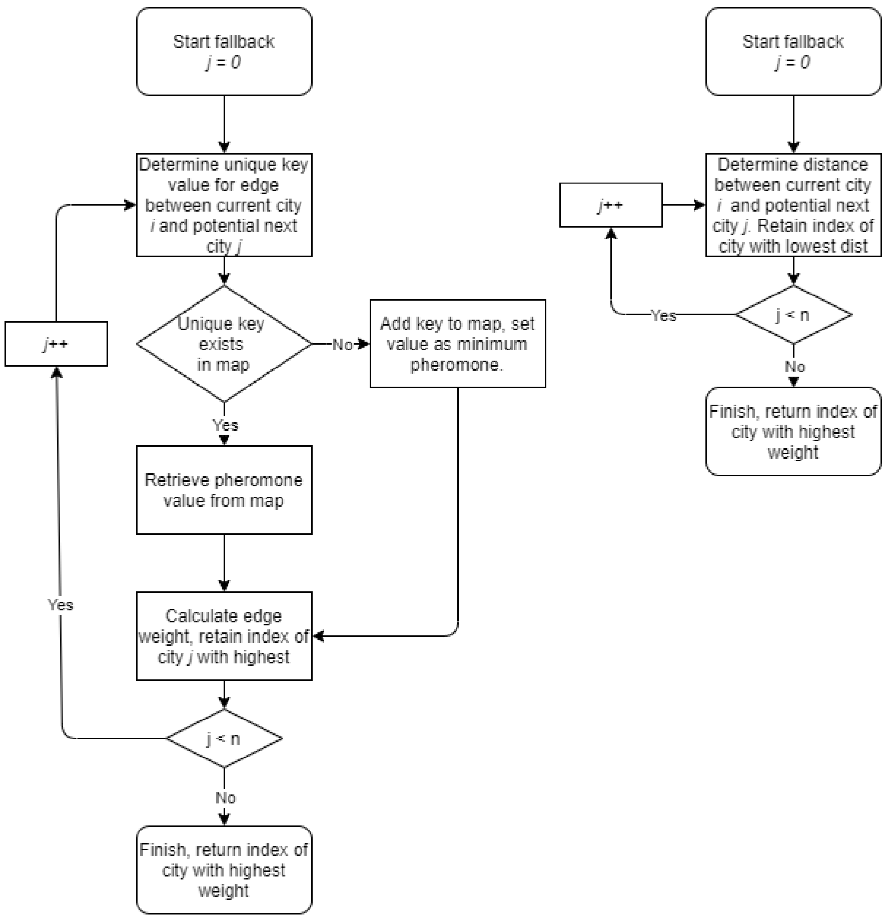

Start fallback
j = 0
Determine unique key value for edge between current city i and potential next city j
Unique key exists in map
No
Add key to map, set value as minimum pheromone.
Yes
Retrieve pheromone value from map
Calculate edge weight, retain index of city j with highest
j < n
Yes
j++
No
Finish, return index of city with highest weight
Start fallback
j = 0
Determine distance between current city i and potential next city j. Retain index of city with lowest dist
j++
j < n
Yes
No
Finish, return index of city with highest weight

Figure 5.2: Flow charts displaying the Pheromone Map fallback (L) and Heuristic Fallback (R)

## 5.2.2 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 proposed is the *Heuristic Fallback*, a greedy method that selects 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. As the different variants of calculating distance between cities in a TSP (as specified by each TSP file, though generally Euclidian Distance is used) generally involve using a costly square root function to determine the true distance, the squared distance is used when selecting the next city, as the true distance is not actually necessary and it is possible to determine the closest available city without using the square root function. While a greedy algorithm is not the best choice as a solution strategy for TSP, since a series of local optimal choices are unlikely to lead to the best global solution, this fallback will be used infrequently, only in cases where no nearest neighbours are available, so any negative effect that the greedy search will have on solution quality will be minimised. The effect that this has on solution quality will be discussed in a later subsection.

### 5.2.3 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. A C++ map object (an associative array) is used, 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 (Alba and Chicano 2007)).

The key for map entries objects is an integer that uniquely identifies one edge, and the value is the weight of that edge. The unique identifier 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 $A$). This is a straightforward way to guarantee uniqueness.

Since this fallback is used only when all nearest neighbours are tabu, it may be assumed that if an edge is found in the map then it has an associated pheromone value, otherwise the pheromone value is taken as $\tau_{min}$. 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 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.

### 5.2.4 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. 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.

If the Pheromone Map fallback method is being used, the pheromone distribution phase is largely identical to the Heuristic fallback, with one 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.

### 5.2.5 Local Search

Variants of the local search (Aarts and Lenstra 2003) technique have been successfully paired with ACO implementations on multiple occasions (Dorigo and Di Caro 1999; Chitty 2017; Mavrovouniotis, Müller, and Yang 2017). 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, the 3-opt local search code from ACOTSP Stützle 2004 is used, and this operator is applied to all tours created in an iteration. The local search phase is parallelised across the threads owned by the ants; each ant performs local search on its own thread at the end of tour construction. This local search implementation makes use of two specific techniques in order to optimise the 3-opt algorithm, which in its default form is computationally intensive, with a time complexity of $O(N^3)$, where N is the number of cities in the given TSP. Firstly, it restricts swaps to cities within a given city's nearest neighbour list, greatly reducing the computational complexity of the local search procedure. This does lead to a small reduction in final solution quality compared to the unrestricted 3-opt (0.1 or $0.2\%$ difference on average between using 20 nearest neighbours and 80 nearest neighbours), but the running time reduction is significant, with a time complexity reduction from $O(N^3)$ to $O(N)$ (Johnson and McGeoch 1997). The second technique used in this 3-opt implementation is the use of *don't-look* bits(Bentley 1992). This technique was created to take advantage of the fact that if an improving move was previously not found for a city, and it remains directly linked to the same neighbours, it is unlikely than an improving move will be found for that city in the solution's current iteration. Therefore, it will not be considered for 3-opt exchanges until it is no longer linked to the same neighbours. This is done through a list of *don't-look* flags, which are initially turned off for every city. If a search for an improvement begins in city $c$, and an improvement cannot be found, the flag for $c$ is switched on, meaning it will no longer be considered as the starting point for an exchange. However, if $c$ is involved in a later successful exchange as an endpoint for one of the exchanged edges, the bit is turned off again, meaning it can once again be considered as the start point for an exchange (Bentley 1992).

## 5.3 Experimental Evaluation

The results of experiments to evaluate the two proposed fallback methods are presented, and the results of the better-performing of the two are compared 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. Results are compared 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* optimisation enabled. The RPM implementation makes use of 3-opt local search, which each ant conducts at the end of solution construction. The PartialACO implementation described in (Chitty 2017) utilises 2-opt with each ant having a 0.001 probability of applying the local search.

Table 5.2: 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 | Time (hrs) |
| **HF** | 1.684 | 1.704 | 1.698 | 1.712 | 1.07 |
| **PMF** | 1.689 | 1.7 | 1.7 | 1.709 | 5.15 |

### 5.3.1 ACO Parameters and Problem Instances

For each experiment, 40 ants are used. Conveniently, this number is equal to both the number of threads available, and the generally recommended number of ants (López-Ibáñez, Stützle, and Dorigo 2016). The $\mathcal{MM}$AS parameter values are set to $\alpha = 1, \beta = 2, \rho = 0.02$. Each ant has a Nearest Neighbour list of size 32, in line with the recommended list size in (Dorigo, Birattari, and Stutzle 2006). Each run of the algorithm consists of 1000 iterations.

The problem instances used in these experiments are taken from the well-known Art TSP collection (*TSP Art Instances* n.d.) of Traveling Salesman Problem instances, created by Robert Bosch, a creator of TSP art. These instances are widely utilised as "challenge" TSP instances, as they are significantly larger than any of the TSPs available in the TSPLIB benchmarking set, and the unique appearance of their optimal tours, which are shown in Figure 5.3, are particularly memorable compared to the functional tours of most TSP problems. As the optimal tours are still maintained and updated for the Art TSPs, they are a good benchmark for solvers aimed at large TSP problems. Results are compared 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) (Honda, Nagata, and Ono 2013).

### 5.3.2 Fallback Comparison

The first experiment was performed to determine which of the 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*.

10 runs of 1000 iterations were carried out for each fallback method, using the

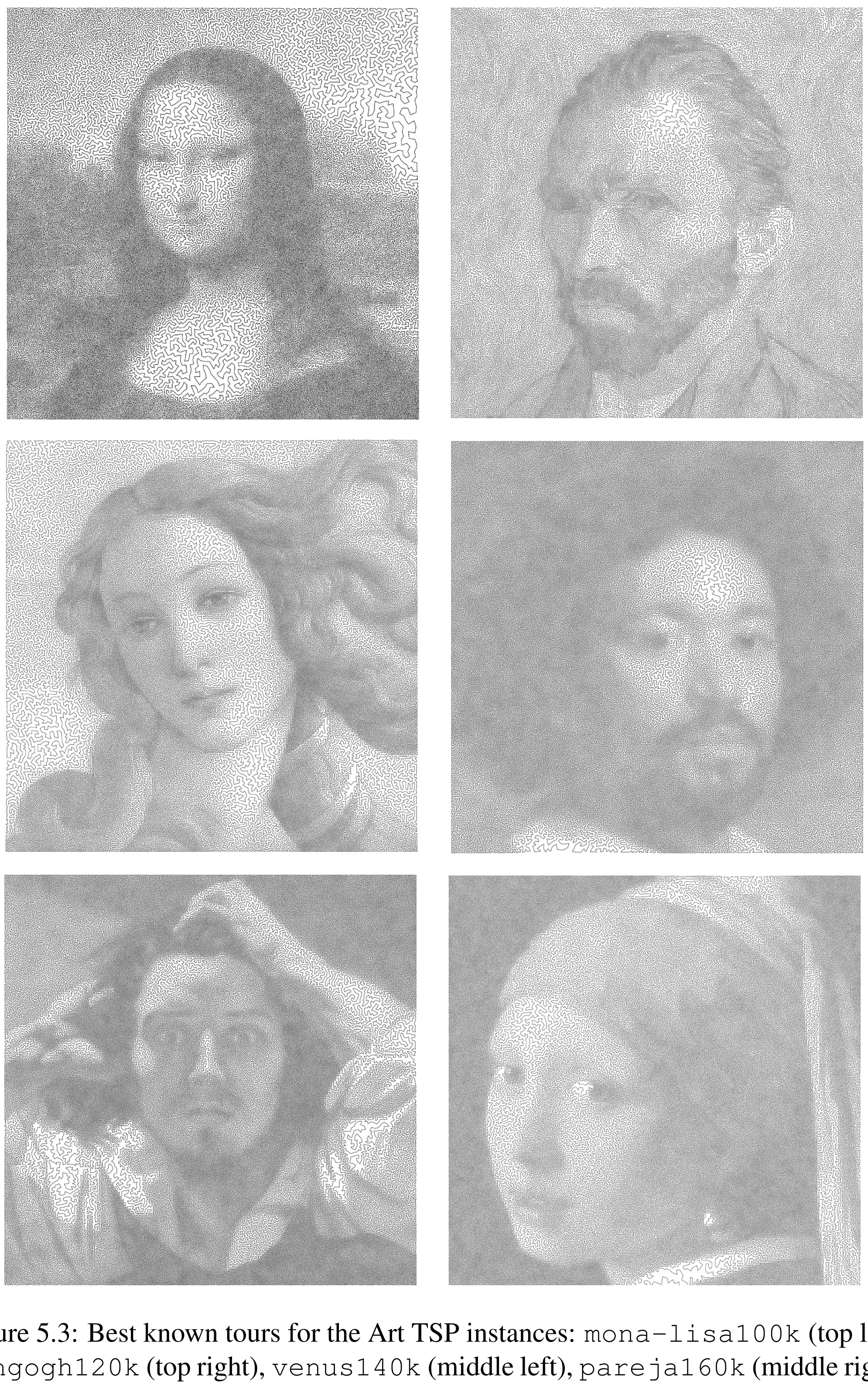


Figure 5.3: Best known tours for the Art TSP instances: `mona-lisa100k` (top left), `vangogh120k` (top right), `venus140k` (middle left), `pareja160k` (middle right), `courbet180k` (bottom left) and `earring200k` (bottom right).

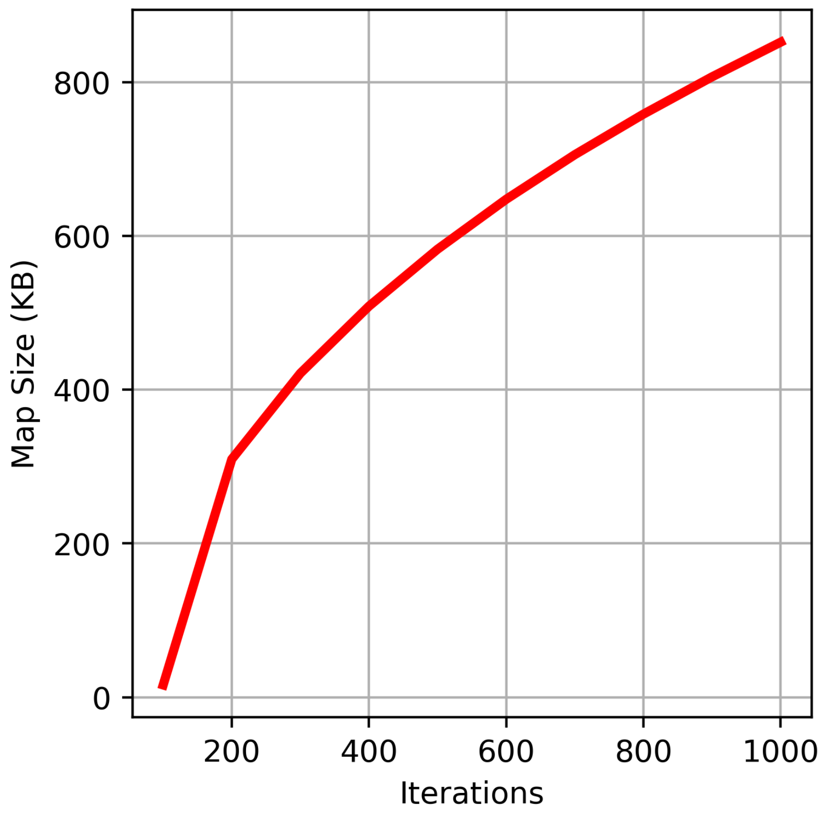


Figure 5.4: Pheromone map size over time

`mona-lisa100k` instance. The results are given in Table 5.2. 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 the 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 5.4 shows the mean memory consumption of the pheromone map as a function of iteration. Although this grows steadily, the 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). The slowing of the growth rate over time can be explained by the more explorative nature of ACO in the initial iterations (more even pheromone levels mean ants are more likely to traverse new edges), as well as the fact that the map is completely empty at the start of the algorithm, meaning that there is a higher chance that an edge traversal needs adding to the map at the beginning of the algorithm. As the map becomes full, this chance decreases, and the pheromone is less evenly distributed and more concentrated on a subset of edges, meaning new paths are less likely to be traversed and ultimately leads to fewer new edges being added to the map.

An additional observation that can be made from this comparison is the effect of the greedy fallback on solution quality: As mentioned previously, greedy tours are not the best choice as a solution strategy for TSP, but used infrequently they have little effect on solution quality compared to the pheromone map, which takes pheromone into account. This makes sense, as infrequently travelled edges will generally have a pheromone level equal to $\tau_{min}$, the minimum level of allowed pheromone, meaning that pheromone will rarely make a substantial difference in the fallback decision making process, essentially reducing both fallback methods to greedy methods in the majority of cases.

As both of the discussed fallback methods only take place in a scenario where no nearest neighbours are available to travel on, their frequency is the same. Analysing the algorithm during run time indicated a fallback rate of around 2.1% - 2.3%, regardless of instance size. This is consistent with the $<3\%$ fallback frequency of the VCSS algorithm discussed in the previous chapter, although the larger instance sizes seem to refine the fallback rate to a more consistent value than the smaller instances used in the VCSS experiments.

### 5.3.3 Results

While there is 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 the restricted pheromone matrix on the five larger Art TSP instances, and is therefore the fallback method used in the subsequent experiments.

For each instance 10 runs, with different seeds, of 1000 iterations are performed. Solutions are compared with those found by PartialACO and P-ACO (Chitty 2017), where these exist. The RPM technique produces solutions that are approximately 1-2% longer than the shortest recorded tours for these instances, which is a significantly smaller difference than P-ACO and PartialACO (see Figure 5.5 for a comparison). It is difficult to directly compare solution *times* due to hardware differences, and the

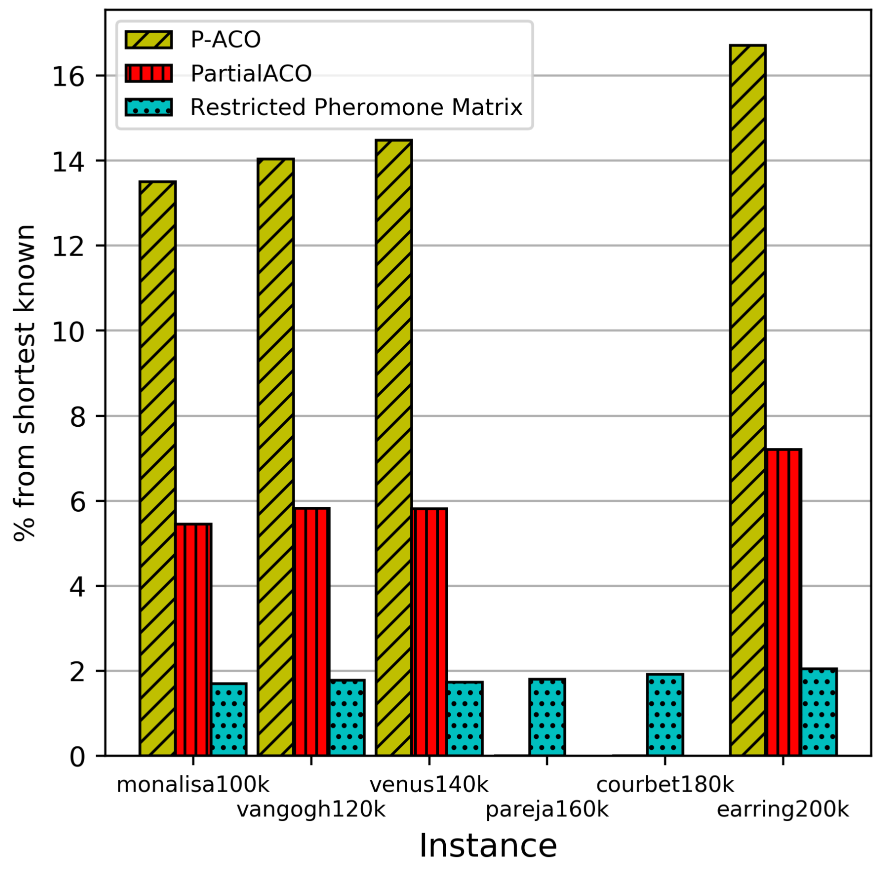


Figure 5.5: Plot of the solution quality difference between P-ACO, PartialACO (results taken from Chitty 2017 and Restricted Pheromone Matrix against shortest known tour. No P-ACO or PartialACO results are available for `pareja160k` and `courbet180k`.

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 5.3. These are broadly comparable times to solution, in both cases using recent commodity hardware.

Figure 5.6 plots solution quality over time for each of the instances. Although small reductions in tour size are still being made when the RPM runs are terminated, improvements are significantly less common than in earlier iterations of the algorithm, indicating that the runs are close to convergence.

### 5.3.4 Local Search Analysis

While the results discussed in the previous subsection made extensive use of 3-opt local search, with one local search per ant per iteration, further analysis has subsequently been performed on the configuration of local search, specifically the occurance rate.

Table 5.3: Average execution times for PartialACO (100,000 iterations) and Restricted Pheromone Matrix (1000 iterations). Note that the iterations are significantly different in terms of required processing: RPM creates an entire tour each iteration, whereas PartialACO only modifies 1% of a tour each iteration, making 1000 RPM iterations roughly equivalent to 100,000 PartialACO iterations.

| **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 |

This experimentation aims to determine the difference in both solution quality and execution time that takes place if the number of performed local searches is reduced, which in turn will indicate whether the potential reduction in solution quality can be deemed as acceptable when the time saving is taken into account.

The RPM technique was tested with different occurance rates of local search, every 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 50, or 100 iterations, with each ant performing a local search. The setup of these experiments is identical to the previous experiments in terms of hardware, parallelisation and the ACO parameters, with the only difference being the number of runs performed of each configuration, 5 rather than 10.

As demonstrated by Figure 5.7, performing a 3-opt local search every iteration leads to significantly worse worse solution qualities than less frequent local searching, even up to one local search per 100 iterations. For the Mona Lisa TSP instance, between 6 and 10 iterations per local search leads to results that are more consistently closest to the optimum, though the overall best solution quality is found when local search is only performed every 100 iterations. The potential reasoning for this is discussed in the following subsection.

Figure 5.8 demonstrates the impact that reducing Local Search occurence rate has on execution time, with a decrease of around 500 seconds between an occurence rate of 1 and 2, and a further decrease for each subsequent reduction of the occurence rate.

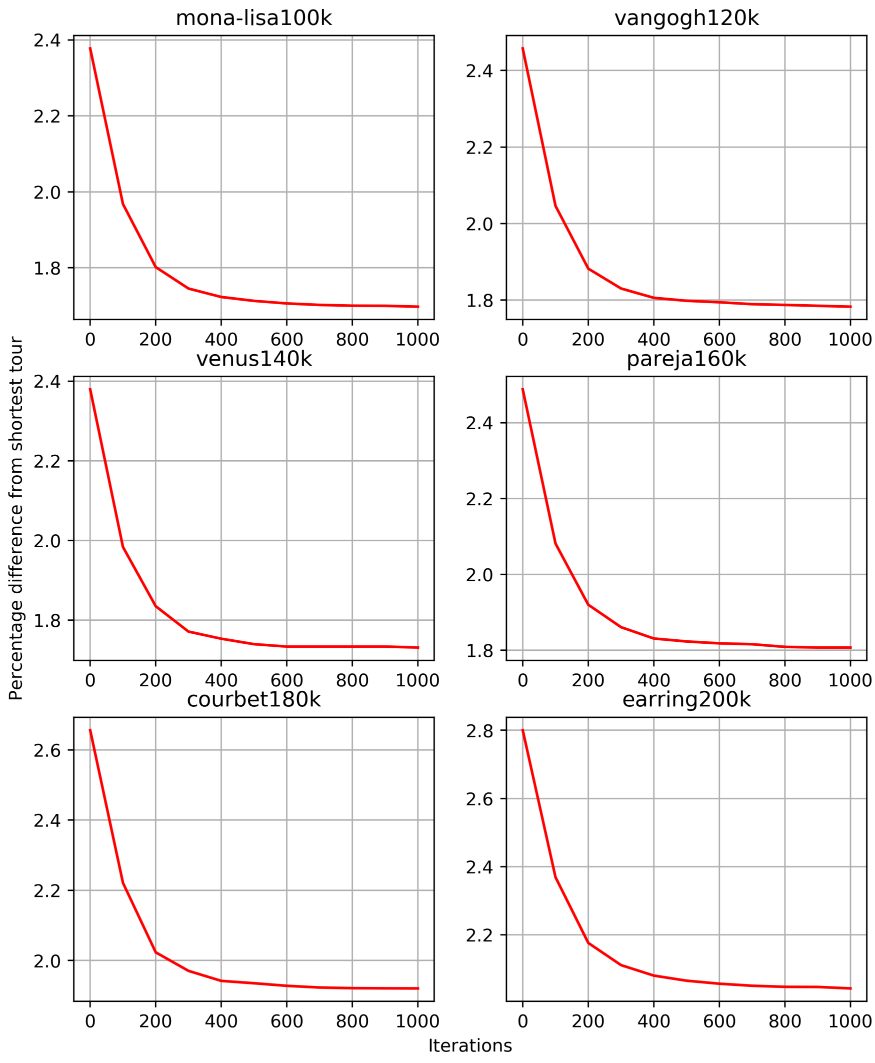


Figure 5.6: Solution quality versus iteration for the Art TSP instances using the heuristic fallback.

## 5.3.5 Discussion

The feasibility of scaling up ACO to solve large ($> 10^5$ city) instances of TSP has been demonstrated, and it has been shown that ACO can produce tours within $2\%$ of the best known on a selection of well-known large instances. Comparison of execution time between RPM and the state-of-the-art GA techique (Honda, Nagata, and Ono 2013) is difficult due to the significant differences in hardware. RPM 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 $2\%$ degradaFtion seen using PartialACO). This consistency of solution qualities suggests that the RPM

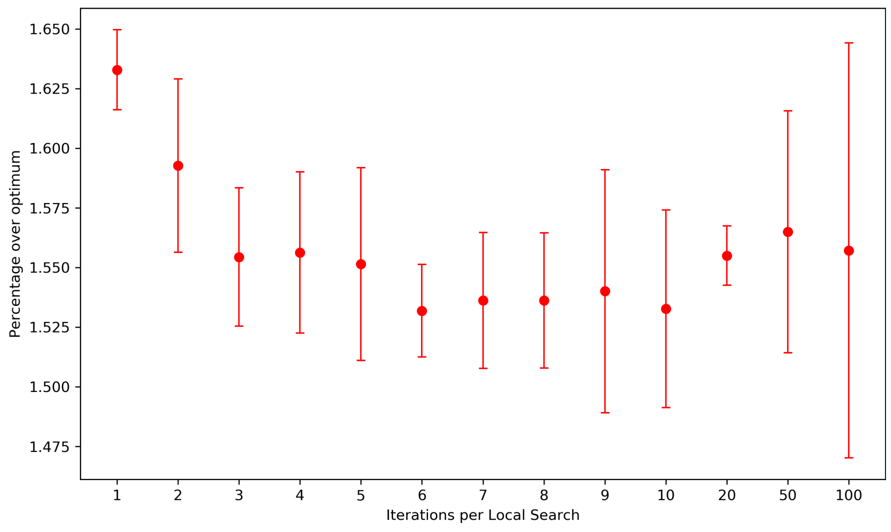


Figure 5.7: The average solution quality of 5 ACO runs with differing local search frequencies, with error bars

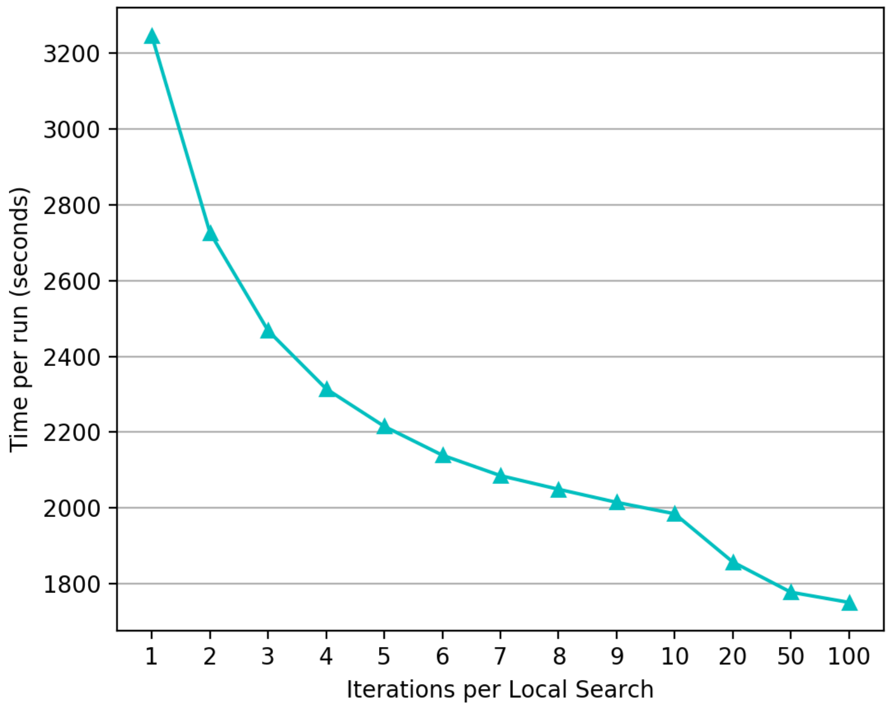


Figure 5.8: The average execution time of 5 ACO runs with differing local search frequencies

technique could potentially be used to obtain good quality tours for problem instances that are even larger than the Art TSPs.

It should be noted that local search plays a significant part in both the execution time and solution quality. RPM, with 40 ants that are guaranteed to perform 3-opt, performs 40 3-opt local searches per iteration, whereas PartialACO, with 16 ants that each have a 0.001 chance to perform a 2-opt local search, performs 0.016 local searches per iteration. It should be noted that the 3-opt local search used in the RPM implementation, as discussed previously, makes use of the nearest neighbour list and don't-look bit optimisation techniques, meaning that the 3-opt used by RPM is actually more efficient than the 2-opt used by PartialACO and P-ACO, which appears to use neither.

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), ignoring pheromone on edges outside the candidate set has little impact. It should be noted that, as previously mentioned, the fallback rate is very low, with fewer than 3% of tour construction selections being made using either fallback method. While pheromone is an integral part of ACO, these 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.

In terms of Local Search, the fact that performing a 3-opt local search every iteration leads to poor results compared to a lower Local Search occurance rate demonstrates that performing a local search for every single iteration is leading the ACO algorithm to become stuck in local minima, while allowing the algorithm to initially find poorer solutions to improve upon using ACO with no local search interference leads to a better quality of solution in the long run. The exact occurance rate is likely to vary depending on the TSP instance used, with the Mona Lisa TSP having an ideal occurance rate of 6 for pure solution quality, or10 for a solution quality that is almost as good with the added benefit of a shorter execution time. The erratic nature of the solution quality when local searches are performed every 100 iterations is potentially caused by the dependence of local search on the ACO algorithm itself - if the ACO performs well, the local search improves it further, whereas the lack of local search more regularly can clearly lead to poor quality solutions. While the execution time results for the main experimentation in which RPM was compared with PartialACO indicates the RPM is unable to improve upon the execution time of PartialACO, the results of the subsequent Local Search experiments indicate that RPM is in fact able to comfortably out-perform PartialACO in terms of execution time as well as solution quality. Further experimentation is required to determine the full reasoning behind the superior performance of the 6-10 range of occurance rates, and how the optimal occurance rate relates to the properties of the TSP instance itself.

## 5.4 Conclusions

While the substantial reduction in memory size allows for the solving 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 the fallback techniques currently uses the vector instructions employed by, for example, I-Roulette and VCSS, and a significant speedup could be obtained by vectorising the fallback algorithms.

Finally, it should be noted 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 (López-Ibáñez, Stützle, and Dorigo 2016), Resource-constrained project scheduling problems (Merkle, Middendorf, and Schmeck 2002), and vehicle routing problems (Bell and McMullen 2004). The methods presented in this chapter could be also be applied in these cases, where the solution of large instances is limited by memory.

# Chapter 6

# PACO-VMP: Parallel Ant Colony Optimisation for Virtual Machine Placement

The work presented in this chapter is currently in submission (Peake, Costen, et al. 2021).

The improvements made to ACO using OpenMP parallelisation and AVX vector instructions are promising, though limited to TSP, which can be considered the traditional testing ground for improvements to the core ACO algorithm. In order to demonstrate the general applicability of these techniques, a different problem with a more significant real-world impact must be solved with ACO. In this case, the previously discussed Virtual Machine Problem has been selected.

## 6.1 Background & Related Work

This Section describes the Virtual Machine Placement Problem, discusses a range of existing methods for its solution, and provide some motivating background on Ant Colony Optimisation, which forms the basis of this own algorithm.

### 6.1.1 Virtual Machine Placement Problem

The real world impact of VMP becomes more significant as time goes on. Cloud computing (Hayes 2008) is an increasingly prevalent paradigm, especially as COVID-19 is forcing companies to more heavily rely on cloud-based software (Alashhab et al.

2020). The cloud computing paradigm is a key enabler for a number of recent developments, such as the Internet of Things (Botta et al. 2014), Edge Computing (Satyanarayanan 2017), and Big Data Analytics (Al-Fuqaha et al. 2015), all of which in turn enable important societal developments such as Smart Cities (Zanella et al. 2014) and Intelligent Transportation Systems (Guerrero-ibanez, Zeadally, and Contreras-Castillo 2015). However, data centres now represent a significant proportion of global energy usage; this figure currently stands at around $2\%$, and it is set to rise (Fernández-Cerero, Fernández-Montes, and Jakóbik 2020). There is, therefore, an urgent need to optimise the software infrastructure underpinning modern data centres.

Resource requirements are expressed in terms of Virtual Machine (VM) instances, each of which carries its own overhead. A key benefit of cloud computing for users is its *scalability*, which is derived from the ability to dynamically increase and reduce resource usage depending on demand. While this *elasticity* is beneficial for users, it provides challenges for cloud computing providers. With constantly changing demand, the assignment of VMs to servers (or Physical Machines, PMs) can quickly become inefficient, leading to unnecessary usage of servers. This can cause providers to use more of their hardware resources than are necessary, which has both an economic and environmental impact. The solution to this is *virtual machine consolidation*, which allocates currently in-use VMs to as few PMs as possible. This increases server utilisation and energy efficiency, and lower power consumption equates to lower energy costs for the host. This also incentivises efficient re-allocation of servers to ensure that they operate in an efficient configuration for a longer amount of time, which leads to a further reduction in energy usage. A number of algorithms have been proposed to address this problem; here, the focus is on methods based on *Ant Colony Optimisation*, specifically *parallel* Ant Colony Optimisation, which takes advantage of modern multi-core hardware to significantly reduce the time required to find satisfactory solutions. The use of the AVX2 instruction set, available on the vast majority of modern CPUs, further reduces execution time in an already parallelised approach.

An instance of the VMP is defined by a set $V$ of virtual machines $V = \{V_i, i \in [1, N_{vm}]\}$ with *CPU requirements* and *RAM requirements* $C_i^{\text{req}}, R_i^{\text{req}} \forall i \in [1, N_{vm}]$, and a set $P$ of physical machines $P = \{P_j, j \in [1, N_{pm}]\}$ with *CPU capacities* and *RAM capacities* $C_j^{\text{cap}}, R_j^{\text{cap}} \forall j \in [1, N_{pm}]$. A *feasible solution* to an instance of the VMP is a mapping of the indices of virtual machines $i$ to physical machines $j$ such that $\forall j$, $\sum_i C_i^{\text{req}} \leq C_j^{\text{cap}}$ and $\sum_i R_i^{\text{req}} \leq R_j^{\text{cap}}$ where the sums are taken over the indices of all virtual machines $i$ which are mapped to the physical machine $j$. The optimisation problem seeks to find a feasible solution which maximises the number of *e*mpty physical machines, which is equal to the cardinality of the set of indices $j$ which are not mapped from any virtual machines $i$.

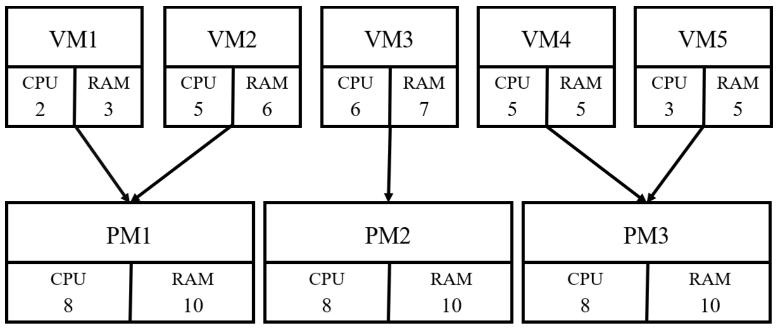


Figure 6.1: Instance of the Virtual Machine Placement problem, with arrows showing allocation of VMs (top) to PMs (bottom). Virtual Machine requests are efficiently allocated to Physical Machines

As the method presented in this chapter is based on Ant Colony Optimisation (ACO), another ACO-based VMP solver, OEMACS (Liu et al. 2016), is selected for comparison. OEMACS treats the VMP problem as a Variable-Sized Bin Packing Problem (VSBPP), a variant of the Bin Packing Problem in which the container elements have differing capacities. OEMACS significantly outperforms FellerACO (Feller, Rilling, and Morin 2011), the first ACO-based VMP solver, in terms of both solution quality and execution time. A further description of OEMACS is available in 2.

In addition to OEMACS, IGA-POP (Abohamama and Hamouda 2020) is also selected for comparison. IGA-POP also frames the VMP as a VSBPP. In IGA-POP, a solution (contained within a *chromosome*) encodes an ordering of VM assignments to PMs. The fitness function for this algorithm prioritises low power usage, and it performs competitively in terms of solution quality against the BF and First-Fit (FF) greedy algorithms, the Sine-Cosine Optimisation Algorithm (SCA) (Mirjalili 2016) and a generic GA. For this reason, IGA-POP is selected as being representative of the "evolutionary" algorithm class of solutions for VMP. A full description of IGA-POP can be found in 2.

## 6.2 Parallel ACO for Virtual Machine Placement

Parallel ACO for Virtual Machine Placement (PACO-VMP) is a novel ACO variant that uses parallelisation techniques and SIMD vector operations to efficiently solve VMP problems. The algorithm uses the clamping and pheromone behaviour of the $\mathcal{MM}$AS variant of ACO, as this is most amenable to parallelisation (due to the absence of communication between ants during an iteration). Complete reference code is available online [1]. The notation used in this chapter is defined in Table 6.1, and the algorithm is summarised in Figure 6.3.

The key underlying data structure for the vast majority of ACO implementations is the *pheromone matrix*, which in this case stores the pheromone information indicating the preferability of two given virtual machines being allocated to the same physical machine as each other, stored as a floating-point number. The size of this matrix is dependant on the number of VMs in the given VMP instance, always being the square of the number of VMs. The pheromone matrix as well as the majority of the more minor data structures in the algorithm (such as PM capacity list, VM capacity list, VM demand list etc.) are stored using a standard C++ array for compatibility with the AVX2 vector instructions, which allow the loading of data from C++ arrays to AVX vectors, and vice-versa.

### 6.2.1 Initialisation Phase

In this phase, the parameters and structures required by the algorithm are created and initalised. An important step is to ensure that all of the arrays that will later be vectorised are padded correctly, which prevents errors when they are loaded into vectors. As this implementation uses the Intel AVX2 instructions, which operate on 8 32-bit values at a time, the size of the arrays must be a multiple of 8. The arrays also need to be aligned in memory correctly in order to be correctly loaded into AVX2 vectors. The pheromone matrix is a matrix of size $N_{\mathrm{VM}} \times N_{\mathrm{VM}}$ , with $N_{\mathrm{VM}}$ being the number of Virtual Machines in the problem instance. The values of the pheromone matrix are initially set to $\tau_0 = 1/N_{\mathrm{PM}}$, where $N_{\mathrm{PM}}$ is the number of Physical Machines. In this phase the value of the $\mathcal{MM}$AS constant $a$ (see equation 6.8) is also set, which is later used to determine the maximum and minimum pheromone values. The number of PMs is initially set to be equal to the number of VMs.

[1] `https://github.com/jnpeake/PACO-VMP`

### 6.2.2 Solution Construction

The first step of the solution construction phase is to randomly shuffle the VMs. This happens at the beginning of each iteration in order to prevent VMs being allocated the same PM purely due to their position in the array. OpenMP is used to allocate each ant's construction process to a separate thread. As each ant only reads from global pheromone memory during the construction phase and does not write to memory, synchronisation is not required. During the construction phase, the ants loop through every VM and allocate it to a PM, unless the current VM is unable to fit in any remaining PM. Any VMs left un-allocated at the end of the loop are then allocated to the PM with the most available capacity, creating an *infeasible* solution. A Local Search procedure, which will be fully described in a later section, is applied to the solution in an attempt to make it *feasible*.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Heuristic | 0.02 | 0.02 | 0.07 | 0.03 | 0.04 | 0.12 | 0.35 | 0.1 |
| | | | | | x | | | |
| Pheromone | 0.06 | 0.05 | 0.09 | 0.08 | 0.06 | 0.03 | 0.02 | 0.07 |
| | | | | | x | | | |
| Random | 0.1 | 0.6 | 0.4 | 0.6 | 0.3 | 0.5 | 0.9 | 0.2 |
| | | | | | = | | | |
| Total | 0.00012 | 0.0006 | 0.00252 | 0.00144 | 0.0072 | 0.0018 | 0.063 | 0.00014 |

Figure 6.2: A demonstration of how the vRoulette-1 technique combines the heuristic and pheromone values of a PM with a random number between 0 and 1. AVX2 instructions allows operations to be carried out on each Vector lane (numbered 0-7) simultaneously.

The selection procedure used to allocate VMs is based on the vRoulette-1 technique developed by Lloyd & Amos (Lloyd and Amos 2016). This is demonstrated in Figure 6.2, which shows how the Heuristic and Pheromone values (which is described in detail later) of the PM in each vector lane are combined with a random number between 0 and 1. This is then multiplied by a *Tabu* value, which is set to 0 or 1 (the value is only set to 0 in the instance that the "PM" in that lane is actually just a placeholder used to pad the PM list to a multiple of 8), and then masked by vectors (denoted *MaxCPUMask* and *MaxRAMMask*) that filter out any PMs that do not have enough available capacity for the current VM. This is done on a vector-by-vector basis, with 8

Table 6.1: List of symbols and notations used in this chapter

| Symbol | Definition |
|---|---|
| $S_{gb}$ | The global best solution |
| $S_{ib}$ | The best solution from the current iteration |
| $P_{gb}$ | Power usage of the global best solution |
| $P_{ib}$ | Power usage of the iteration best solution |
| $\tau_0$ | The initial pheromone value |
| $N_s$ | The number of PMs ants are able to use |
| $N_{gb}$ | The number of PMs used in $S_{gb}$ |
| $N_{ib}$ | The number of PMs used in $S_{ib}$ |
| $N_{vm}$ | The number of VMs in the current instance |
| $N_{pm}$ | The number of PMs in the current instance |
| $k$ | Current iteration number |
| $k_{\max}$ | Maximum number of iterations permitted |
| $i_{\text{cur}}$ | The current VM |
| $j_{\text{cur}}$ | The current PM |
| $\alpha$ | Pheromone influence |
| $\beta$ | Heuristic influence |
| $\rho$ | Pheromone decay rate |
| $\eta_{ij}$ | Heuristic value between VM $i$ and PM $j$ |
| $\tau_{ij}$ | Pheromone value between VMs $i$ and $j$ |
| $P$ | Power usage of current solution |
| $P_j^{\max}$ | Maximum power usage of PM $j$ |
| $P_j^{\text{idle}}$ | Idle power usage of PM $j$ |
| $f_C$ | CPU usage ratio for current PM |
| $f_R$ | RAM usage ratio for current PM |
| $C_j^{\text{used}}$ | Current CPU usage on PM $j$ |
| $R_j^{\text{used}}$ | Current RAM usage on PM $j$ |
| $C_i^{\text{req}}$ | CPU requirement of VM $i$ |
| $R_i^{\text{req}}$ | RAM requirement of VM $i$ |
| $C_j^{\text{cap}}$ | Total CPU capacity on PM $j$ |
| $R_j^{\text{cap}}$ | Total RAM capacity on PM $j$ |
| $\tau_{\max}$ | Maximum pheromone value |
| $\tau_{\min}$ | Minimum pheromone value |
| $a$ | $\mathcal{MM}$AS constant value |
| $N_{\min}$ | Theoretical lower limit of current VMP |
| $N_B$ | Number of type B servers |
| $C_A^{\text{cap}}$ | CPU capacity of type A servers |
| $C_B^{\text{cap}}$ | CPU capacity of type B servers |
| $R_A^{\text{cap}}$ | RAM capacity of type A servers |
| $R_B^{\text{cap}}$ | RAM capacity of type B servers |

PMs being processed for selection in parallel. The 8 current PM values are compared lane-by-lane with a vector of the highest PM values in the current selection process.

Once every PM has been processed, a parallel reduction (with the *max* operator) is carried out on this vector and the PM corresponding to the highest value is then assigned the current VM. If the highest value is lower than 0, this indicates that no PMs had enough capacity available for the current VM, and the VM is added to the unassigned list to be allocated once the solution construction procedure has been completed.

While the original vRoulette-1 implementation made use of the AVX512 instruction set, which allows for 16-wide vectors and features additional instructions compared to AVX2, it is not currently as widely available as the AVX2 instruction set, which is available on most Intel CPUs released since 2013, and most AMD CPUs released since 2015. For the implementation evaluated here, AVX2 instructions were used.

### 6.2.3 Pheromone & Heuristic Definition

As with any ACO implementation, the definition of the pheromone and heuristic values is crucial for the consistent construction of good-quality solutions.

The heuristic is a problem-specific value which indicates the favourability of assigning a VM to a PM. The definition of the heuristic value can differ significantly even within ACO implementations that aim to solve the same problem. A key difference between calculating the heuristic value for VMP is the need for a dynamically calculated heuristic which differs depending on the current state of the PM that is being assigned to, and this requires the heuristic to be calculated at every step of the solution for every VM, which increases the solution time compared to the more static heuristic values of problems such as the Traveling Salesman Problem.

The heuristic definition is designed to ensure that the fewest possible number of PMs are used, by prioritising both resource utilisation balance and total resource utilisation. The prioritisation of total utilisation makes it more likely that an ant will allocate the current VM to a PM that already contains other VMs, while the resource balance will attempt to keep the available RAM and CPU on a PM as even as possible, which will prevent PMs exhausting one resource capacity while still having a large available capacity for the other resource. The heuristic value, $\eta_{ij}$, associated with placement of virtual machine $i$ on physical machine $j$ is given by

$$\eta_{ij} = \frac{1 - |f_C - f_R|}{1 + f_C + f_R} \tag{6.1}$$

where

$$f_C = \frac{C_j^{\text{used}} + C_i^{\text{req}}}{C_j^{\text{cap}}} \tag{6.2}$$

and

$$f_R = \frac{R_j^{\text{used}} + R_i^{\text{req}}}{R_j^{\text{cap}}}. \tag{6.3}$$

Here, $C_j^{\text{used}}$ and $R_j^{\text{used}}$ are, respectively, the current CPU and RAM usage of physical machine $j$, $C_i^{\text{req}}$ and $R_i^{\text{req}}$ are the CPU and RAM requirements of virtual machine $i$, and $C_j^{\text{cap}}$ and $R_j^{\text{cap}}$ are the CPU and RAM capacities of physical machine $j$.

Implementations of ACO for VMP generally use one of two pheromone trail definitions: the first defines the trail as being between VMs and the PMs to which they are allocated, and the second defines trails as being between VMs that are allocated the same PMs, meaning that VMs are more likely to be allocated to a PM with VMs that they have previously shared with in good solutions. For PACO-VMP, the pheromone trail associates VMs with other VMs. The pheromone distributed is based on solution quality, which in this case is the energy consumption of the solution. Pheromone is updated as

$$\tau_{ij} \leftarrow \tau_{ij} + \frac{K_P}{P} \tag{6.4}$$

where $K_P$ is a constant, for all pairs of VMs $i$, $j$ which are allocated to the same PM in the global best solution, where $P$ is the power usage of the solution. In these experiments, $K_P$ is set to 365, which brings the power usage down from an annual rate to a daily rate. In practice, this is a constant which scales power usage into the typical range of values for $\tau_{min}$ and $\tau_{max}$, defined below. The actual value of this parameter may be varied depending on the units used for power in the instance definition, and can be considered a hyper-parameter of the method. The power usage is defined as:

$$P = \sum_{j=1}^{N_{\text{PM}}} \left( (P_j^{\text{max}} - P_j^{\text{idle}}) \frac{C_j^{\text{used}}}{C_j^{\text{cap}}} + P_j^{\text{idle}} \right) \tag{6.5}$$

where $N_{\text{PM}}$ is the number of PMs in the current instance and $P_j^{\text{max}}$ and $P_j^{\text{idle}}$ are the maximum and idle power usage of physical machine $j$ respectively. Once pheromone has been deposited, the trail is globally decayed by a fixed factor, $\rho$. The choice of value of $\rho$ will be discussed in Section 6.3. The definition of pheromone is based on power usage as it will reflect the positive impact of a lower number of PMs while still measuring differences between solutions with the same number of PMs used.

As the selection process of the PACO-VMP algorithm attempts to allocate VMs to PMs, directly loading pheromone from the pheromone matrix is insufficient when it comes to deciding which PM to allocate a VM to. Instead, the average amount of pheromone between the current VM and the VMs that are currently allocated to the PM being evaluated is calculated (the amount of pheromone between VM and PM is initially set to $\tau_0$, and remains at that level until a VM is added to the PM).

### 6.2.4 Local Search

The Local Search used in the presented algorithm is based on a technique developed by Alvim et al. (Alvim et al. 1999) for the Bin Packing Problem, and also utilised by Liu et al. (Liu et al. 2016) for the VMP. In this algorithm, after each solution is found, one bin is destroyed, or a PM in the case of the VMP problem. If a subsequent solution is then able to successfully fit all items in the remaining bins, it is considered *feasible*. However, if no feasible solution can be found, the local search technique is applied. There are two phases of the local search technique, the swap phase and the insertion phase. Any PM that has been allocated more VMs than it has capacity for is marked as *overloaded*. In the swap phase an overloaded PM is compared with every non-overloaded PM, and the algorithm attempts to swap each VM in the overloaded PM with each VM in the non-overloaded PM. This continues until either a successful swap takes place, or every non-overloaded PM has been compared to the overloaded PM. Regardless of the outcome, the process is carried out again for the next PM, and this continues until every overloaded PM has been compared. If the swap phase is unable to successfully find a feasible solution, the insertion phase is then performed. In this phase, each overloaded PM attempts to allocate each of its VMs to a non-overloaded PM. While this is far less likely to produce positive results than the swap phase, it is still able to occasionally make progress where the swap phase cannot. A drawback of this Local Search technique is the fact that it is non-trivial, leading to significant cost in processing time, and it runs serially, rather than in parallel. Due to this, local search is performed only on the iteration-best solution.

### 6.2.5 Pheromone Distribution

The final phase of the PACO-VMP algorithm is the pheromone distribution. As the algorithm is based on the $\mathcal{MM}$AS ACO variant, pheromone is only deposited by either the global-best or iteration-best ant. As mentioned previously, pheromone is distributed between VMs allocated to the same PM. The global amount of pheromone then decays by a static amount. $\mathcal{MM}$AS utilises a clamping procedure to prevent stagnation, by restricting the level of pheromone to be between maximum and minimum values. The maximum and minimum values are defined as

$$\tau_{\max} = \frac{1}{\rho N_{\text{best}}^{\text{global}}} \tag{6.6}$$

$$\tau_{\min} = \tau_{\max} \frac{2(1-a)}{(N_{\text{VM}}+1)a} \tag{6.7}$$

where $N_{\text{VM}}$ is the number of VMs in the current instance, $P_{\min}$ is the global lowest PM usage, and

$$a = \exp(\ln(0.05)/N_{\text{VM}}). \tag{6.8}$$

## 6.3 Experimental Results

The performance of the PACO-VMP algorithm is investigated by comparing with an implementation of the OEMACS algorithm, which is an ACO-based method that generally out-performs conventional heuristics and evolutionary algorithms for this problem (Liu et al. 2016), and a state-of-the-art genetic algorithm, IGA-POP (Abohamama and Hamouda 2020). Code for OEMACS is publicly available[2]. All algorithms were implemented in C++, and all tests were carried out on a machine with an Intel® Xeon E5-2640 v4 processor with 20 cores running at a base frequency of 2.4 GHz and a maximum frequency of 3.4 GHz. Code was compiled using the GNU C++ compiler (g++), with *O2* optimisation enabled. The initial comparative tests will compare a sequential implementation of PACO-VMP with OEMACS and IGA-POP, as OEMACS is unable to be parallelised in its current form without synchronisation issues. Two variants of IGA-POP will be used: the first, referred to as GA1, uses the fitness function also used by PACO-VMP and OEMACS; the second, referred to as GA2, uses a slightly modified version of the fitness function used in the initial IGA-POP experiments (Abohamama and Hamouda 2020). To demonstrate the impact of OpenMP parallelisation, a parallelised PACO-VMP using OpenMP was also used for comparison, using one ant per each of the 20 available cores. While the execution time differs, solution quality is identical to the sequential version.

All problem instances used in these experiments were randomly generated. For

[2]https://github.com/Budding0828/OEMACS

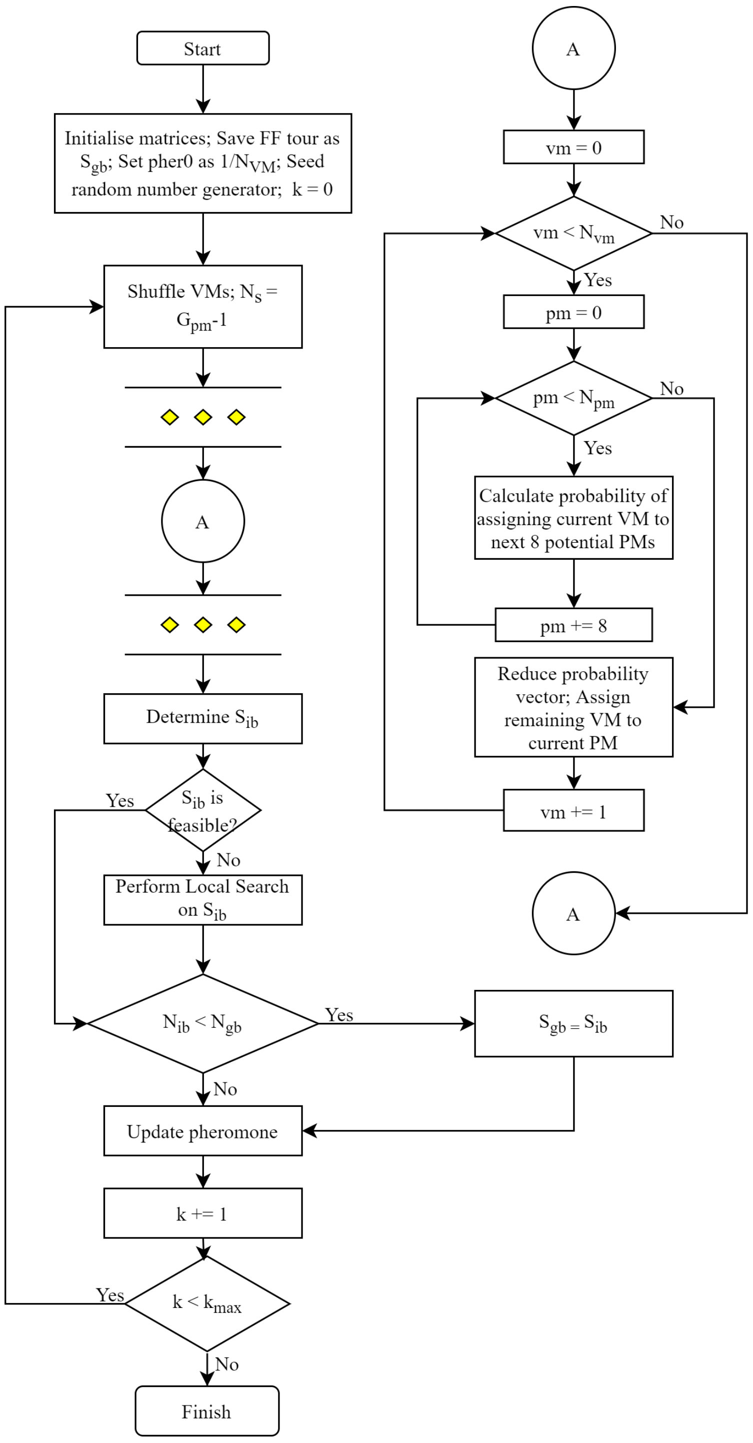

Start
Initialise matrices; Save FF tour as $S_{gb}$; Set pher0 as $1/N_{VM}$; Seed random number generator; k = 0
Shuffle VMs; $N_s = G_{pm}$-1
A
Determine $S_{ib}$
$S_{ib}$ is feasible?
Yes
No
Perform Local Search on $S_{ib}$
$N_{ib} < N_{gb}$
Yes
$S_{gb} = S_{ib}$
No
Update pheromone
k += 1
$k < k_{max}$
Yes
No
Finish
A
vm = 0
$vm < N_{vm}$
No
Yes
pm = 0
$pm < N_{pm}$
No
Yes
Calculate probability of assigning current VM to next 8 potential PMs
pm += 8
Reduce probability vector; Assign remaining VM to current PM
vm += 1
A

Figure 6.3: A flow chart detailing the PACO-VMP algorithm

each instance, the initial number of Physical Machines is set to be equal to the number of Virtual Machines. The problem instance dataset consists of 600 VMP instances, split into 6 sets of 100 instances with 100, 200, 300, 400, 500 and 1000 VMs. One run is performed using each instance. One run per instance is used with a larger number of instances, rather than multiple runs on a smaller number of instances; a proof in (Birattari 2004) shows that given a budget of $N$ runs, selecting a $K$ instances and performing $n$ runs on each with $N = Kn$ is a suboptimal choice and that the best statistical estimate of algorithm performance is obtained from a single run on each of $N$ independently selected instances, contrary to popular belief.

For each experiment 20 ants are used for PACO-VMP, as this allows for one ant to be allocated to each available core in the OpenMP-enabled variant. For the PACO-VMP algorithm, the ACO parameter values specified in the next section are used, and for OEMACS the default values as specified in (Liu et al. 2016) are used. Both PACO-VMP and OEMACS are run for 50 iterations. For GA1 and GA2, the parameters specified in (Abohamama and Hamouda 2020), 200 iterations and a population size of the number of PMs multiplied by 4, are used. These results are compared against the First Fit (FF) algorithm in order to provide a baseline greedy algorithm implementation. FF was selected over the more widely used First Fit Decreasing(FFD) algorithm due to better results on the problem instance data sets. It should be noted that the solution construction time for FF is near-instantaneous for all instance sizes, and thus has been omitted from all execution time plots.

The First Fit algorithm is a straightforward algorithm initially developed for the Bin Packing Problem (Ullman 1971). The algorithm maintains 2 lists, PMs and VMs. It loops through the VMs, allocating them to the first available PM that has sufficient capacity for the demand of the VM. While this greedy approach is unlikely to lead to the best solution for a problem, it can provide a solid baseline for comparison against other algorithms.

### 6.3.1 Parameter Tuning

Before running the experiments, parameter tuning is performed on the three main ACO parameters ($\alpha$, $\beta$ and $\rho$) in order to determine the optimum values of these values for the PACO-VMP algorithm. For both $\alpha$ and $\beta$ integer values in the range 0-6 inclusive were used. This range was selected as values higher than 5 lead to floating point underflows in the code, and typical values of $\alpha$ and $\beta$ in other problem domains are $< 5$. Values are restricted to integers as the powers are evaluated by multiplication to avoid the use of the *pow* function; no AVX2 *pow* function exists. For $\rho$, the available values were 0.1, 0.2, 0.3, 0.4, 0.5 and 0.6, complying with the range suggested in (López-Ibáñez, Stützle, and Dorigo 2016).

For the parameter tuning experiments problem instances with 1000 VMs were used from instance set B, which will be described in detail later in this section.. The larger variance in solution quality for this problem set allows for the impact of changing parameters to be more easily observed. Each possible combination of parameters were used, totalling 216 sets, and each set was run with 5 different random seeds on 8 problem instances, meaning that 8640 tests were run in total. Consistent with the experiments described in the next subsection, 20 ants and 50 iterations were used per run.

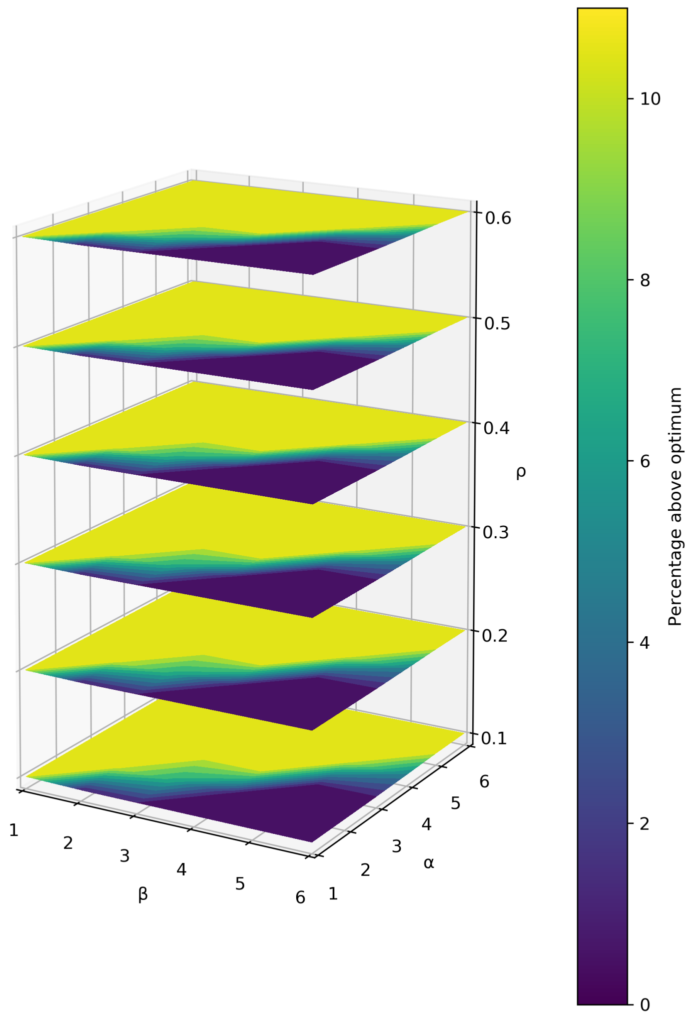


Figure 6.4: Solution quality is displayed, presented as a percentage value over the minimum possible value, with each z-axis plane of the plot displaying the results of a set $\rho$ value.

The results shown in Figure 6.4 provide several points of discussion. Firstly is the optimal values of the parameters themselves - it is clear that a high $\beta$ value combined with a low $\alpha$ value lead to the best solution quality, while $\rho$ has little, if any, perceivable impact. Referencing the raw data of the experiments indicates that an $\alpha$ value of 1, $\beta$ value of 6 and $\rho$ value of 0.6 lead to the best solutions on average.

The parameter tuning process also shows an interesting trend of poorly configured parameters leading to very poor quality solutions, and investigation of the raw data reveals that ACO is incapable of improving upon the initial baseline tour, set by the greedy FF algorithm, with low $\beta$ and high $\alpha$ values. In the context of these experiments, this makes sense; 50 iterations is a very small number of iterations for an ACO solver, as it does not allow the pheromone mechanic sufficient time to make an impact. Higher $\beta$ values allow the heuristic to have a higher influence on the decisions made by ants, which leads to better solutions in a low iteration count scenario. The justification for this low iteration count is given in the next subsection.

### 6.3.2 Instance Set A: Large-scale Homogeneous Environment with Bottleneck

Set A is designed to test the performance of PACO-VMP in a straightforward scenario where the PMs are identical and the demands of the VMs are fairly evenly divided between RAM and CPU. This set consists of 1000 VMP instances equally divided between 100, 200, 300, 400, 500 and 1000 VM instances. This dataset is similar to the Set A data used by Liu *et al.* (Liu et al. 2016) in evaluating OEMACS, which was initially created to benchmark the Reordering Grouping Genetic Algorithm (RGGA) (Wilcox, McNabb, and Seppi 2011); however, this data is no longer publicly available. In comparison to this data, a larger number of smaller instances were used in these experiments. Additionally, the Set A data used for in (Liu et al. 2016) was deliberately created to have equal resource distribution, despite the different ranges of CPU and RAM demands; in contrast, the randomly generated nature of the Set A used in these experiments leads to a CPU bottleneck, with CPU demand ratio of approximately 5:4 against RAM demand.

The VM requirements for this instance set are randomly generated in ranges of [1,128] for CPU and [1,100] for RAM. Each PM has a capacity of 500 for both CPU and RAM, leading to slightly higher average CPU utilisation than RAM but still close to 1:1. As these instances are randomly generated, there is no known optimum, but a lower limit to the number of PMs used in the solution, $N_{\min}$, is calculated

$$N_{\min} = \max \left\{ \frac{\sum_{j=1}^{N_{\mathrm{VM}}} C_j^{\mathrm{req}}}{C_i^{\mathrm{cap}}}, \frac{\sum_{j=1}^{N_{\mathrm{VM}}} R_j^{\mathrm{req}}}{R_i^{\mathrm{cap}}} \right\} \tag{6.9}$$

where $i$ is the index of any physical machine; as the servers in Instance Set A are homogeneous, it does not matter which physical machine is used to evaluate this quantity.

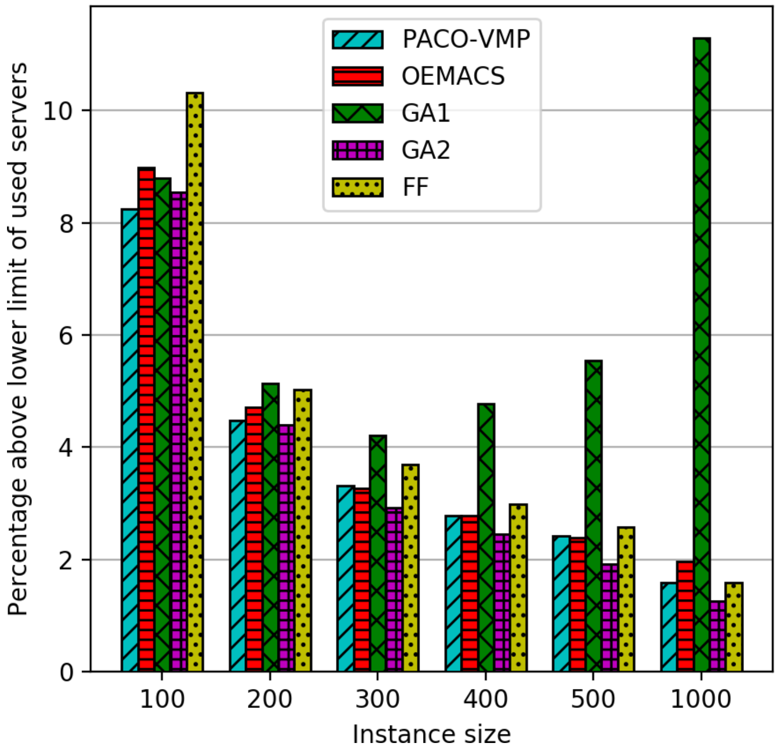


Figure 6.5: Solution difference measured as percentage over theoretical optimum for PACO-VMP, OEMACS, GA1, GA2 and FF for instance set A.

The results for instance set A in terms of solution quality are displayed in Figure 6.5. FF shows good results throughout, improving as the problem instances get larger, which indicates that it is fairly simple for a the greedy FF solver to create good-quality solutions for the large-scale version of the VMP problem. In all but one instance, OEMACS is able to match or exceed the solutions created by FF. Likewise, PACO-VMP outperforms or matches OEMACS on 5 sizes of instances including the largest instances. It should be noted that the PACO-VMP algorithm utilises the FF result as its initial best tour, meaning that it is not able to find worse tours than FF. A distinction between the results of set A and the other instance sets is that FF is competitive with the two ACO algorithms. For the other instance sets this is not the case, but

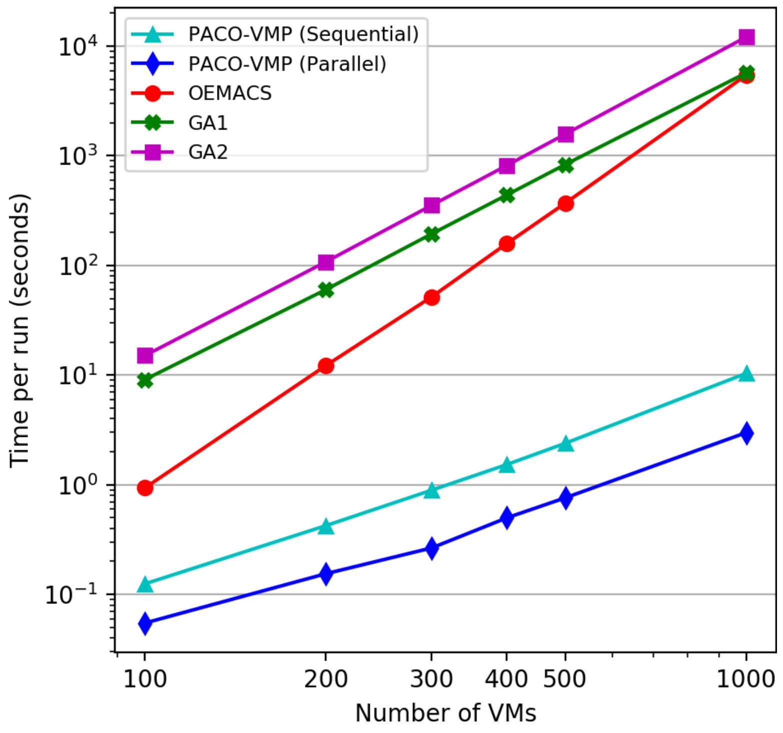


Figure 6.6: Average time to solve (seconds) for 100 instances of a given instance size for PACO-VMP, OEMACS, GA1 and GA2 for instance set A.

as the large-scale problem is fairly straightforward, it allows FF to find good quality solutions. GA2 also performs well on this dataset, outperforming PACO-VMP on all but a single dataset. On the other hand, GA1 struggles, remaining moderately competitive for the smaller instances but performing dramatically worse on the 400, 500 and 1000 VM instances.

Execution time results for instance set A are shown in Figure 6.6. From this plot it is clear to see that PACO-VMP has a significant advantage over OEMACS when it comes to execution time, beginning at around 1 order of magnitude for the size 100 instances, and increasing to an advantage of around 3 orders of magnitude for the 1000 VM instance sets. An even larger advantage is held over the two IGA-POP algorithms, beginning at around 2 orders of magnitude for the 100 VM instances and increasing to around 3 orders of magnitude for the 1000 VM instances. Interestingly, despite beginning with a sizeable time advantage over IGA-POP, OEMACS performs similarly to GA1 for the 1000 VM instance set. The parallelised version of PACO-VMP increases the time difference between it and the sequential variant of PACO-VMP, increasing from a speedup of $2.2\times$ for the 100 VM instances to a speedup of $3.47\times$ for 1000 VM instances.

### 6.3.3 Instance Set B: Small-scale Homogeneous Environment with Bottleneck

Set B introduces a bottleneck resource to the problem instances, testing the performance of PACO-VMP in a slightly more complicated scenario which will lead to more overloaded servers. As with the previous instance set, Set B consists of 1000 VMP instances equally divided between 100, 200, 300, 400, 500 and 1000 VM instances. VM requirements are randomly generated, in the range of [1-4] for CPU (measured in cores) and [1-8] for RAM (measured in GB). PM capacity is 16 cores for CPU and 32GB for RAM. As the VM requirements are evenly distributed, the probability of a 4 core CPU requirement is 0.25, whereas the probability of an 8GB RAM requirement is 0.125, leading to a slight CPU bottleneck, though less severe than Set A, with a ratio of approximately 10:9. The main differing factor between Set B and Set A is the scale of the capacity and demand - smaller capacities lead to less flexibility in the local search process, though the reduced demands still lead to some flexibility. As with Set A, these instances have no known optimum, and a lower limit is calculated using the same formula.

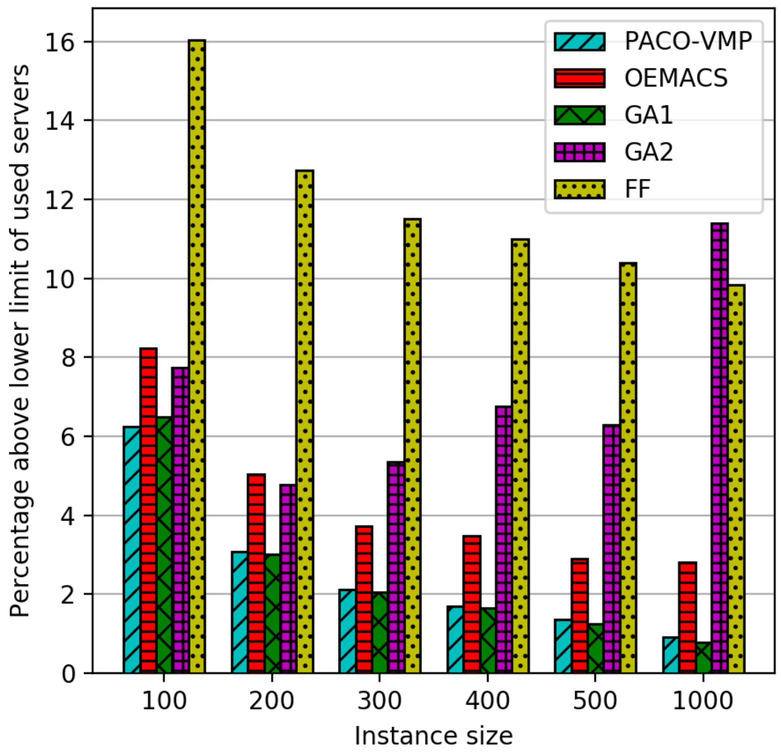


Figure 6.7: Solution difference measured in percentage over theoretical optimum for PACO-VMP, OEMACS, GA1, GA2 and FF for instance set B.

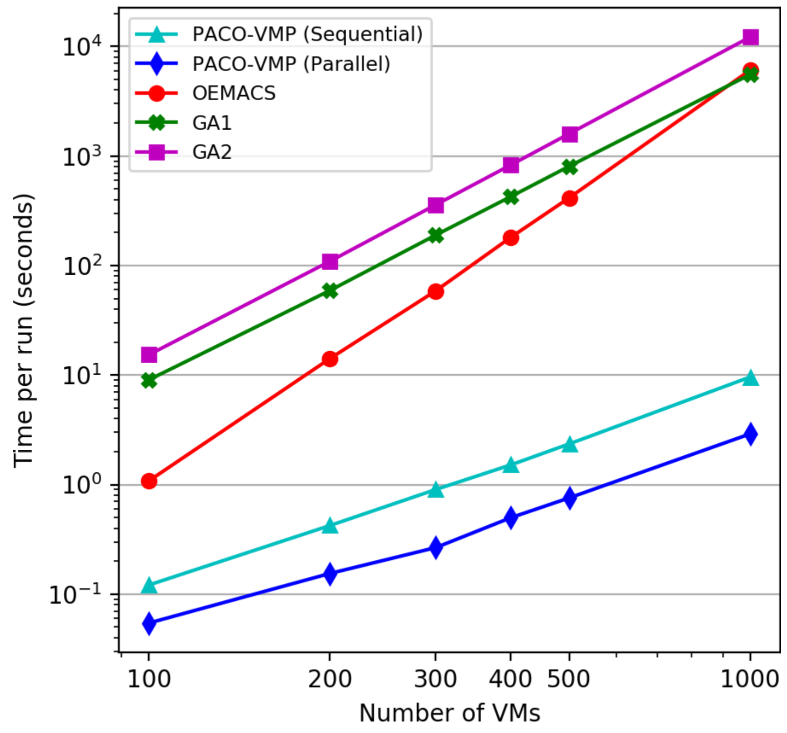


Figure 6.8: Average time to solve (seconds) for 100 instances of a given instance size for PACO-VMP, OEMACS, GA1 and GA2 for instance set B.

Unlike instance set A, the results for instance set B displayed in Figure 6.7 show a clear difference between PACO-VMP and OEMACS. FF's poor results also indicate that a greedy solver has more difficulty finding a good solution for the bottlenecked VMP problem than for the non-bottlenecked variant. While OEMACS significantly outperforms FF, PACO-VMP outperforms it for every problem size, finding solutions that range from 1%-2% closer to the theoretical lower limit. Additionally, the solution quality in terms of percentage is actually worse for the size 1000 instances with OEMACS, whereas PACO-VMP continues to improve. In contrast to Set A, GA1 performs very well in this bottle-necked scenario, with PACO-VMP returning better results for the 100 VM instances but then returning very slightly worse results for the larger instances. Conversely, GA2 performs poorly, initially returning similar results to OEMACS before worsening on the larger instances, and even being outperformed by FF for the 1000 VM instances.

In terms of execution time, displayed in Figure 6.8, the results for PACO-VMP are near-identical to the results for instance set A, demonstrating that the bottleneck led to no additional execution time. This is also the case for OEMACS, which also achieved near-identical execution times to the instance set A results. The execution time advantage held by PACO-VMP is maintained, with OEMACS once again losing the advantage it holds over IGA-POP as the solution size increases. The difference between sequential and parallel PACO-VMP is also near-identical to instance set A, though the speedup increase is slightly smaller, from $2.2\times$ to $3.27\times$.

### 6.3.4 Instance Set C: Heterogeneous Environment with Bottleneck

Set C further complicates the problem instances by introducing non-identical servers, simulating a scenario in which a cloud host has multiple server types. Two types of server are defined, *A* and *B*. Server type *A* has a CPU capacity of 16 cores and a RAM capacity of 32GB. Server type *B* has a CPU capacity of 32 cores and a RAM capacity of 64GB. However, type *B* servers only make up 10% of the total PMs in each problem instance, meaning that VMs will have to use both types of servers. This will test the ability of PACO-VMP to prioritise the high capacity servers while still allocating the VMs efficiently. The VM requirements are in the range of [1,8] for CPU and [1,32] for RAM, meaning that the bottleneck resource in this case is RAM. Set C utilises the same instance sizes as the previous sets.

Due to the heterogeneous servers in this instance set, an alternative formula is required for calculating the lower limit to the number of PMs

$$N_{\min} = N_B + \max\left\{\frac{\sum_{j=1}^{N_{\text{VM}}} C_j^{\text{req}} - N_B C_B^{\text{cap}}}{C_A^{\text{cap}}}, \frac{\sum_{j=1}^{N_{\text{VM}}} R_j^{\text{req}} - N_B R_B^{\text{cap}}}{R_A^{\text{cap}}}\right\} \tag{6.10}$$

where $N_B$ is the number of type *B* servers, $C_A^{\text{cap}}$ and $C_B^{\text{cap}}$ are the CPU capacities of type A and B servers respectively, $R_A^{\text{cap}}$ and $R_B^{\text{cap}}$ are the RAM capacities of type A and B servers respectively, and $C_j^{\text{req}}$ is the CPU requirement of virtual machine $j$.

The results shown in Figure 6.9 indicate that FF performs very poorly on instance set C, with the heterogeneous servers causing issues for the greedy technique. OEMACS significantly outperforms FF once again, but is itself outperformed by PACO-VMP, with solution quality improvement ranging from 5% for 100 VM instances to around 10% for 1000 VM instances. While OEMACS performs significantly worse on instance set C than the other sets, PACO-VMP is able to capably solve the heterogeneous instances. As with instance set B, while OEMACS begins to return worse solution qualities for the size 1000 instances, PACO-VMP continues to improve as the instance size increases. The performance of the GA variants is also consistent with set B, with GA1 slightly outperforming PACO-VMP in all but one instance size,

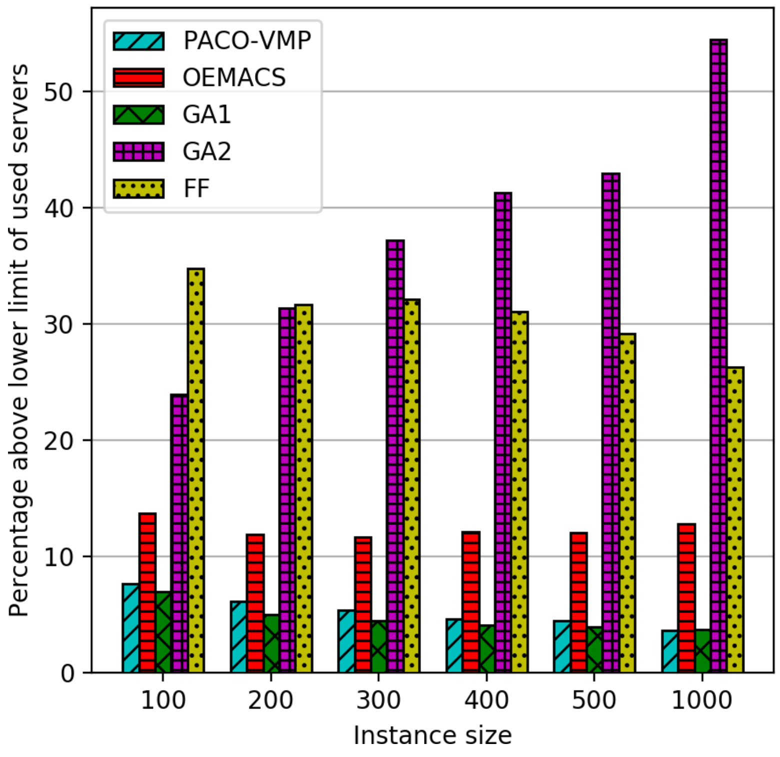


Figure 6.9: Solution difference measured in percentage over theoretical optimum for PACO-VMP, OEMACS, GA1 and GA2 and FF for instance set C.

and GA2 performing poorly, showing even poorer results on instance set C.

Execution time for instance set C, as displayed in Figure 6.10, is similar to the other instance sets, with the execution time of PACO-VMP being near identical. However, OEMACS takes slightly longer to solve the instances in set C, further increasing the execution time advantage held by PACO-VMP. Additionally, the execution time of OEMACS is now closer to the time of GA2 than GA1 for 1000 VM instances, emphasising the increased difficulty that OEMACS has when trying to solve the bottlenecked, homogeneous problem. As with the previous instance sets, the time difference between the sequential and parallel PACO-VMP increases slightly as the instance sizes increase, from $2.6\times$ to $3.78\times$.

### 6.3.5 Discussion

The significant execution time reduction that can be enabled through the use of parallelisation and vectorisation techniques on a wide range of different problem instance sets that represent three realistic Cloud Computing scenarios has been demonstrated.

Table 6.2: Results of the experiments on FF, OEMACS and PACO-VMP. Entries in the Set column represent the 100 instances of the specified size from the specific instance set. Solution Quality is the average percentage over the theoretical minimum for all 100 problem instances for each size within each set with values in **bold** being the best result, on the condition that it is significantly different when results are analysed with the Wilcoxon signed-rank test. Execution Time is the average time per run in seconds for all 100 problem instances for each size within each set. The sequential and parallel versions of PACO-VMP are included as PACO(S) and PACO(P) respectively

| | **Solution Quality (%)** | | | | | **Execution Time (seconds)** | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Set | FF | OEMACS | GA1 | GA2 | PACO | OEMACS | GA1 | GA2 | PACO(S) | PACO(P) |
| A100 | 10.3 | 8.98 | 8.70 | 8.54 | 8.25 | 0.936 | 9.098 | 15.05 | 0.125 | 0.055 |
| A200 | 5.03 | 4.71 | 5.13 | 4.4 | 4.47 | 12.22 | 60.23 | 107.9 | 0.423 | 0.154 |
| A300 | 3.68 | 3.26 | 4.21 | **2.92** | 3.32 | 51.81 | 194.4 | 354.3 | 0.892 | 0.265 |
| A400 | 2.98 | 2.78 | 4.78 | **2.46** | 2.78 | 158.8 | 440.0 | 817.7 | 1.528 | 0.499 |
| A500 | 2.58 | 2.39 | 5.54 | **1.92** | 2.42 | 369.9 | 832.8 | 1575 | 2.387 | 0.759 |
| A1000 | 1.58 | 1.96 | 11.3 | **1.26** | 1.58 | 5450 | 5711 | 10478 | 10.37 | 2.986 |
| B100 | 16.0 | 8.24 | 6.49 | 7.75 | 6.24 | 1.082 | 9.044 | 15.32 | 0.121 | 0.054 |
| B200 | 12.7 | 5.05 | 3.01 | 4.78 | 3.08 | 14.14 | 59.15 | 109.2 | 0.422 | 0.154 |
| B300 | 11.5 | 3.72 | 2.05 | 5.36 | 2.11 | 58.75 | 189.7 | 358.6 | 0.902 | 0.266 |
| B400 | 11.0 | 3.47 | 1.64 | 6.77 | 1.69 | 181.1 | 426.2 | 829.1 | 1.514 | 0.499 |
| B500 | 10.4 | 2.89 | 1.25 | 6.30 | 1.37 | 414.8 | 805.4 | 1596 | 2.351 | 0.756 |
| B1000 | 9.83 | 2.82 | **0.78** | 11.39 | 0.91 | 6080 | 5546 | 10321 | 9.594 | 2.904 |
| C100 | 34.8 | 13.7 | **6.96** | 23.9 | 7.67 | 1.656 | 9.733 | 16.98 | 0.135 | 0.056 |
| C200 | 31.6 | 11.9 | **5.03** | 31.4 | 6.10 | 22.14 | 63.66 | 116.7 | 0.464 | 0.158 |
| C300 | 32.1 | 11.7 | **4.46** | 37.2 | 5.39 | 88.56 | 205.0 | 383.7 | 0.980 | 0.274 |
| C400 | 31.0 | 12.1 | **4.07** | 41.3 | 4.66 | 270.3 | 456.8 | 874.7 | 1.663 | 0.511 |
| C500 | 29.2 | 12.0 | **3.93** | 43.0 | 4.44 | 633.0 | 858.4 | 1665 | 2.623 | 0.771 |
| C1000 | 26.3 | 12.8 | 3.70 | 54.5 | 3.62 | 8873 | 5753 | 10347 | 10.69 | 2.873 |
| **Average** | 15.7 | 6.92 | 4.61 | 16.4 | 3.89 | 1260 | 1201 | 2221 | 2.621 | 0.777 |
| **Rank** | 4 | 3 | 2 | 5 | 1 | 4 | 3 | 5 | 2 | 1 |

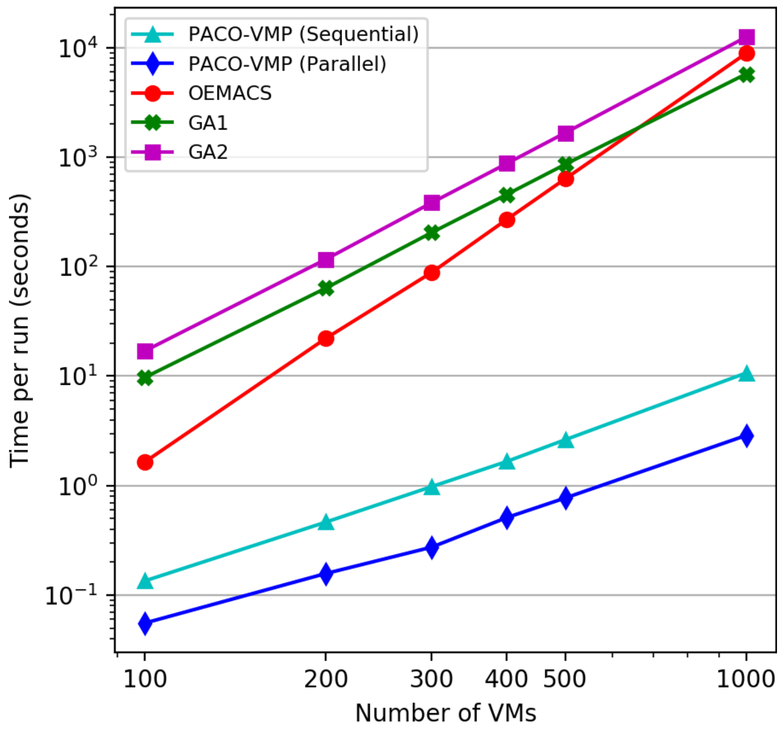


Figure 6.10: Average time to solve (seconds) for 100 instances of a given instance size for PACO-VMP, OEMACS, GA1 and GA2 for instance set C.

PACO-VMP outperforms OEMACS in each instance set, very slightly in set A and significantly in sets B and C, and while it is matched by GA1 in set B and C, it performs significantly better in instance set A. The opposite is true of GA2, which outperforms PACO-VMP in set A, but significantly underperforms in sets B and C. The consistency demonstrates the versatility of ACO compared to IGA-POP: while IGA-POP performs well, it requires two separate fitness functions in order to match PACO-VMP. This is confirmed by the averages displayed in Table 6.2, which clearly show that PACO has the best average solution quality in terms of percentage above optimum. Upon further analysis of the IGA-POP results, the issue stems from the tendency of the algorithm to assign VMs to empty PMs even when currently used PMs have enough capacity remaining, which happens regardless of fitness function. Both PACO-VMP and OEMACS enforce a limit on the number of PMs than can be used (the previous best number of PMs) which prevents this behaviour. Additionally, it is worth nothing that despite a 10x increase of instance size in the experiments, the quality of the solutions produced by PACO-VMP remains consistent. The percentages above the lower limit of PM utilisation decrease with each increase in instance size, which in terms of raw numbers indicates a fairly consistent number of PMs over the minimum. This suggests that the PACO-VMP implementation could still produce good results for even larger VMP problem instances. This is an advantage over OEMACS, which provides degrading solution quality for size 1000 instances, a trend which would potentially continue as instance sizes increase.

The main focus of PACO-VMP is to improve execution time, and it succeeds at this objective. While PACO-VMP and OEMACS use similar pheromone definitions and local search techniques, PACO-VMP produces better results both in terms of execution time and solution quality. This may be caused partly by the choice of $\mathcal{MM}$AS algorithm over ACS, and also by differences in the selection probabilities due to the use of independent roulette. It has been shown (Lloyd and Amos 2016) that independent roulette algorithms (such as *vRoulette*) tend to make greedier selections than the traditional roulette wheel algorithm, which may be a factor in the different solution qualities found between PACO-VMP and OEMACS. Clearly the areas in which PACO-VMP and OEMACS differ are significant in terms of execution time, as PACO-VMP has a time complexity of $O(n^2)$, whereas OEMACS is, experimentally, closer to $O(n^4)$. The main contributing factor to this is the probability calculation: whereas PACO-VMP uses the resource wastage formula as given in Formula 1 as the heuristic value, OEMACS uses a much more complex formula that includes the resource wastage, but also has extra sums over the VMs in both the numerator and denominator of the formula. Experimentally, the time complexity of the IGA-POP variants is approximately $O(n^3)$.

The results are summarised in Table 6.1. This table shows the results for solution quality and execution time, and for the solution quality results it is indicated which results are statistically significant. The results of each algorithm on each instance of a given size can be paired, and compared to each other using the Wilcoxon signed-rank test, a non-parametric test which can be used to compare paired sets of readings. Since an all-vs-all comparison of three tests is performed (all possible pairs of algorithms) the Bonferroni correction is applied, and divide the significance threshold by the number of tests (in this case 3). Bold values in the table show the solution quality values which are significantly better than the other four algorithms, using a significance threshold of $0.002$ (that is, $0.01$ after application of the Bonferroni correction). In general, one of the two GA versions tends to produce the best solutions, however although the differences are in many cases statistically significant, the magnitude of the effect is small. For example, in the case of the A1000 instances, a comparison between GA2 and PACO shows that out of the 100 trials, GA2 is superior for 46 instances whereas ACO is superior for 3, with 51 ties. Although this is a statistically highly significant difference, the *magnitude* of the difference is only 0.32% in solution quality. This demonstrates that the experiments are very sensitive in detecting significant, but small, differences in performance. Qualitatively, the results show that one of the two GAs generally performs the best for any set of instances, but this is often accompanied by the other GA performing the worst. Since the GA is used with the recommended parameters for the population size and number of generations, this performance also comes at a significant cost; for example in the 1000 VM instances, GA1 and GA2 will perform 4000 evaluations per generation for 200 generations, compared to 20 evaluations for 50 iterations in PACO. Furthermore, the difference in performance between the two cost functions is very clear; using the original cost function proposed by (Abohamama and Hamouda 2020) leads to poor performance on the B and C instance sets. PACO-VMP on the other hand, achieves solution quality close to best (or best) across all instance types, without any sensitivity to the algorithm parameters, and achieves better average solution quality than the other ACO algorithm (OEMACS) in 15 out of the 18 instance categories. There is also a clear advantage for PACO-VMP in both scalability and execution time. The computational complexity of PACO is superior to both OEMACS and GA1/2, and the execution time of the parallel version is several orders of magnitude less in most cases. For the C1000 instances, the most challenging instance set, PACO-VMP achieves the best solution quality of all algorithms in an average time of 2.873s, while GA1/2 and OEMACS require several hours of CPU time to reach a solution.

As with the TSP problem discussed in previous sections, these experiments have focused on the static variant of the VMP problem. The current algorithm would require alteration in order to be able to solve the dynamic VMP problem, which more realistically represents the scenario of Virtual Machine Placement. The main change that would be required is to the pheromone and weight matrices; these are currently a static size, and are restricted to standard C++ arrays in order to be compatible with AVX2 instructions, meaning they are inflexible. In the dynamic problem, VMs are added to represent a new user starting up a VM, and removed to represent a user shutting down a VM - each time this happens, the size of the array would have to be changed, which for the C++ array means building a completely new array and moving all relevant information over to it. While this is possible, it is time consuming, and investigation into using more flexible data types such as C++ vectors or maps with AVX vectors could lead to a more efficient method of increasing and decreasing matrix sizes.

## 6.4 Conclusions & Future Work

In this chapter PACO-VMP was presented, a parallelised and vectorised implementation of $\mathcal{MM}$AS for solving the Virtual Machine Placement problem. The method is several orders of magnitude faster than two current state-of-the-art ACO solvers, OEMACS and IGA-POP while producing comparable or superior results. Since virtual machine placement in the real world is a problem in which reducing time to solution can have significant cost benefits, the improved execution time performance of PACO-VMP

While PACO-VMP is capable of solving the static VMP problem, in reality this problem is rarely static. Real-world cloud workloads have constantly changing demand, with Virtual Machines being added and removed from the workload constantly. As with the static VMP, execution time is vital for dynamic VMPs in order to minimise time spent in an inefficient configuration, and PACO-VMP's positive results on the static problem indicate that it could also be effectively used to solve the dynamic problem. This is an area for further investigation.

The parameter tuning phase of the experiments revealed that the performance of the algorithm is relatively insensitive to the parameter governing the importance of pheromone information, further suggesting that analysing and improving pheromone definition may lead to better solution quality from the underlying ACO mechanism. This is a potentially fruitful area of further work.

Many assumptions were made in this implementation regarding the VMP problem, including that there will always be as many PMs available as VMs, that performance doesn't degrade when the PMs reach 100% capacity, and that CPU and RAM are the only requirements. These assumptions are commonly made to simplify the problem solving process rather than having to consider a vast array of additional variables. Another potentially fruitful area that is fairly to investigate is the use of additional parameters for the VMP problem, rather than just CPU and RAM. Further work is required to investigate the inclusion of these additional parameters.

# Chapter 7

# Conclusions and Future Work

Within this thesis, several novel contributions to Ant Colony Optimisation, particularly in terms of execution time, have been presented. At the time that the publications described throughout the thesis were written, the presented work represented the best performing ACO implementation for the traveling salesman problem in terms of execution time, the first true ACO implementation capable of solving the Art TSP instances, and the best performing ACO implementation for the Virtual Machine Placement problem, in terms of execution time (while remaining competitive with a contemporary Genetic Algorithm based solver in terms of solution quality). While parallelisation is a straightforward step that could fairly simply be reproduced for most ACO implementations, especially MMAS implementations with the lack of necessary synchronisation, the AVX vector instructions used are more difficult to implement effectively, not to mention the task of identifying areas of the code that are particularly amenable to vectorisation.

A key element of this thesis was the demonstration of the versatility of the vectorised techniques described in Chapter 4. Due to the low current availability of AVX-512 on contemporary hardware, and the discontinuation of the Xeon Phi range of manycore CPUs used for the experimentation in Chapter 4, the presented work risked looking somewhat restricted to a high end platform. As shown in Chapter 5, the vectorised techniques can be ported to AVX2, an instruction set with a significantly larger availability on current hardware, with only a small amount of work, largely centered around the replacement of AVX512 instructions that have no AVX2 equivalent.

Versatility was also a key factor in the motivation for Chapter 6, as while TSP is ACO's traditional "proving ground" for new techniques, it is necessary to demonstrate a technique's effectiveness on other problems to demonstrate that the technique can be utilised in other contexts. While the Vectorised Candidate Set Selection technique and, by extension, the Restricted Pheromone Matrix were not carried over from TSP due to no clear definition of what would constitute a "nearest neighbour" definition for that problem, the parallelisation and vectorisation ideas used in previous work were used again for VMP, again through the use of AVX2 vector instructions. The fact that PACO-VMP was able to significantly outperform OEMACS, the previous best ACO VMP implementation, demonstrates the effectiveness of vectorisation once again. The key element of the AVX2 instructions is that they are not problem-specific - anywhere an array exists in code, they can be used. This has been demonstrated on two seperate problems in this thesis, and hopefully in future AVX instructions will see more use in ACO implementations.

## 7.1 Subsequent Developments

In the time since the publication of the Restricted Pheromone Matrix technique, described in Chapter 5, further work has been done to improve the scalability of ACO, increasing the effectiveness on large problem instances. While RPM was designed to allow ACO to solve the Art TSPs without substantially altering the core algorithm, the execution time for the initial experimentation was slightly worse than other contemporary ACO techniques. The Memory-Friendly ACOTSP technique (Martínez and García 2021, which cites Peake, Amos, et al. 2019) improves further upon RPM, introducing the concept of "backup cities".

In the ACOTSP-MF algorithm, cities are grouped into three categories: Closest neighbouring cities, the aforementioned backup cities, and other cities. The first structure is identical to the nearest neighbour list used in RPM, with the $n$ nearest cities to a given city being considered as its "nearest neighbours". As with RPM, these cities have full pheromone, weight and distance information stored in global memory. The backup cities function as an extension to the nearest neighbour cities, with the memory situation of those cities varying depending on which approach is selected: The *conservative* mode treats the backup cities identically to the nearest neighbour cities, storing weight, pheromone and distance information in global memory; the *aggressive* mode retains neither weight nor pheromone data for the backup cities, instead only storing distance information in global memory. In the aggressive case, if no nearest neighbour city is available, the distance matrix will be used to find the closest backup city. This functions similarly to the heuristic fallback method used in RPM, though with a much smaller pool of cities to select from and distance information able to be loaded from the matrix rather than calculated dynamically. In the rare occasion where neither a nearest neighbour city nor a backup city is available, the closest other city is selected, with the distance calculated dynamically. The use of the backup city data structure increases the used memory when compared to RPM, though it is still much closer to RPM's memory usage than the default ACO memory usage.

One significant difference between ACOTSP-MF and RPM is the method of vectorisation - while RPM used hand-vectorised methods using the AVX2 instruction set, ACOTSP-MF uses *#pragma ivdep* to automatically vectorise the loop. The differences in execution time between these approaches are an area for future investigation. As with RPM, the number of nearest neighbours used by ACOTSP-MF is set to 32, with 1024 backup cities. While this is quite a significant number of backup cities, the search space of cities is still only 1% the size of the search space for the default ACO algorithm. Another parallel with RPM is the use of 3-opt local search, which ACOTSP-MF uses to improve solution quality. However, unlike RPM, with a default configuration of one local search per ant per iteration, ACOTSP-MF instead opts to perform one local search every 100 iterations, which reduces the performance impact of the operation.

Three factors have to be considered in the context of the ACOTSP-MF results: Memory complexity, solution quality and execution time. In terms of memory complexity, the use of the backup city structure is actually a backwards step compared to RPM, although as previously mentioned the ACOTSP-MF memory requirement of 1024 backup cities is still significantly closer to the RPM memory requirement than the standard ACO requirement. This increased memory usage does have a positive impact on execution time though, as the backup city structure significantly reduces the amount of time taken when no nearest neighbours are available. While the RPM fallback involved calculating the distance for every single city in the TSP, ACOTSP-MF has a much smaller search space, and every city in this search space already has a distance calculated and stored in a matrix, making the time taken to retrieve that information significantly less than RPM. In fact, ACOTSP-MF is able to find a solution to the Mona-Lisa100k TSP in just 85 seconds, compared to the 1.36 hours needed by RPM, a significant speedup. For solution quality, the ACOTSP-MF results were very similar to the RPM results, which can be expected given the similarity of the core algorithm.

The existence of the ACOTSP-MF method shows that RPM has had a definite impact in scalable ACO research, and hopefully it will lead to further techniques being created that improve the scalability of ACO even further.

## 7.2 Suggested Future Work

While the work presented in this thesis demonstrated the effectiveness of parallel and vector techniques, further work is possible for each of the main contributions.

The Vectorised Candidate Set Selection method has currently only been demonstrated using the Traveling Salesman Problem. While the technique is not applicable to every potential problem domain due to the dependence on the concept of a nearest neighbour, many problems do exist that could make use of the VCSS technique. Problems that utilise distance as a heuristic, such as the Vehicle Routing Problem (Du and He 2012), could be used as a benchmark for testing the VCSS problem. Nearest neighbour lists are not restricted to distance, however, so the technique could also be applied to problems such as Pattern Recognition (Cover and Hart 1967), Computer Vision (Muja and Lowe 2014), and classification (Cunningham and Delany 2020), all of which have previously utilised nearest neighbour techniques. Some way of determining what constitutes a "nearest neighbour", or at least the use of some form of candidate set, for problems such as Virtual Machine Placement would also allow the technique to be more widely applicable.

The Restricted Pheromone Matrix technique is bound to problem domains in which VCSS is also applicable, as the reduced size of the pheromone matrix itself is dependant on the existence of a nearest neighbour list. Therefore, like VCSS, expansion to other problem domains would help to prove the viability of RPM on a wide range of problems. The previously discussed ACOTSP-MF technique has already improved the technique fairly substantially, with only solution quality still being a relative weak point. However, the heuristic and pheromone definitions for solving TSP with ACO are well-established at this point, so any improvement to the solution quality of RPM would require a substantial development to be made in discovering a more optimal pheromone/heuristic definition.

Parallel Ant Colony Optimisation for Virtual Machine Placement (PACO-VMP) has a lot of potential for future work, largely due to how fluid the current definition of the VMP problem is.The definitions of the virtual machine problem in the literature vary considerably; work on unifying the problem into one more widely-accepted definition would be useful for future VMP research. The creation of a set of test instances, similar to the widely used TSPLIB set of TSP problems, would also be a significant contribution to current VMP research as no public instance sets currently exist. In terms of PACO-VMP itself, assumptions were made when the algorithm was created: firstly, that RAM and CPU were the only two parameters used by both PMs and VMs;

secondly, that utilising 100% of the capacity of a PM would cause no performance issues; and finally, that enough PMs would always be available to allow for the least optimal configuration of PMs, i.e. one PM per VM. Eliminating these assumptions and solving a more constrained version of the VMP would further demonstrate the benefit of the PACO-VMP technique on a real-world VMP scenario.

Each of the three algorithms described in this thesis make use of either the AVX-512 or AVX2 instruction set, the availability of which covers most modern desktop and HPC CPUs. However, the expansion of the vector techniques used in these algorithms to other instruction sets, such as Arm's NEON instructions, would allow these algorithms to be used in embedded systems. Increasing the efficiency of processes on microprocessors is a key enabler of Edge Computing, a computing paradigm in which devices associated with an IoT network perform computation locally rather than relying on sending and receiving data to and from the cloud. This reduces memory bandwidth and improves response times. The potential porting of the VCSS technique to the NEON instruction set is briefly discussed at the end of Chapter 4.

A useful development during the creation of all three of these techniques was the vector class, which abstracted the use of vector instructions into a separate class and allows for vector instructions to be implemented as operators, rather than using the AVX instructions inline. This allows for the definition of each vector function to be changed depending on the AVX instructions available on any given hardware, with AVX-512 compatible hardware using AVX-512, AVX2 compatible hardware using AVX2, and hardware compatible with neither instruction set using sequential versions of the functions. This further allows for truly cross-platform vectorisation, whereas normally the platform would have to be taken into consideration when using vector instructions. This class could be extended to use the NEON instruction set and potentially even the SSE instruction sets to allow compatibility for older hardware. A public release of this class as a Vectorisation library would allow for simple, cross-platform hand-coded vectorisation without needing to be intimately familiar with the instructions themselves. This vector class is included in Appendix A, which demonstrates the different definitions of functions depending on the available instruction set. It should be noted that while AVX-512 instructions are not included for every method, it is easily possible to port over the AVX2 methods to AVX-512. The use of this vector class is demonstrated in Appendix B, the selection method used for both the VCSS technique and the RPM technique.

# Bibliography

Aarts, Emile and Jan Karel Lenstra (2003). *Local Search in Combinatorial Optimization*. Princeton University Press.

Abohamama, A.S. and Eslam Hamouda (2020). “A Hybrid Energy–Aware Virtual Machine Placement Algorithm for Cloud Environments”. In: *Expert Systems with Applications* 150, p. 113306. ISSN: 0957-4174. DOI: `https://doi.org/10.1016/j.eswa.2020.113306`. URL: `http://www.sciencedirect.com/science/article/pii/S0957417420301317`.

Ahmed, Zakir H (2010). “Genetic algorithm for the traveling salesman problem using sequential constructive crossover operator”. In: *International Journal of Biometrics & Bioinformatics (IJBB)* 3.6, p. 96.

Alashhab, Ziyad R., Mohammed Anbar, Manmeet Mahinderjit Singh, Yu-Beng Leau, Zaher Ali Al-Sai, and Sami Abu Alhayja’a (2020). “Impact of Coronavirus Pandemic Crisis on Technologies and Cloud Computing Applications”. In: *Journal of Electronic Science and Technology*, p. 100059. ISSN: 1674-862X. DOI: `https://doi.org/10.1016/j.jnlest.2020.100059`. URL: `https://www.sciencedirect.com/science/article/pii/S1674862X20300665`.

Alba, Enrique and Francisco Chicano (2007). “ACOhg: Dealing with Huge Graphs”. In: *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*. GECCO ’07. London, England: ACM, pp. 10–17. ISBN: 978-1-59593-697-4. DOI: `10.1145/1276958.1276961`. URL: `http://doi.acm.org/10.1145/1276958.1276961`.

Alvim, Adriana, Fred S Glover, Celso C Ribeiro, and Dario J Aloise (1999). *Local Search for the Bin Packing Problem.*

Amos, Martyn, Matthew Crossley, and Huw Lloyd (2019). “Solving Nurikabe with Ant Colony Optimization”. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 129–130.

Anderson, Edward J and Michael C Ferris (1994). “Genetic algorithms for combinatorial optimization: the assemble line balancing problem”. In: *ORSA Journal on Computing* 6.2, pp. 161–173.

Bai, Hongtao, Dantong OuYang, Ximing Li, Lili He, and Haihong Yu (2009). “MAX-MIN Ant System on GPU with CUDA”. In: *2009 Fourth International Conference on Innovative Computing, Information and Control (ICICIC)*. IEEE, pp. 801–804.

Bell, John E. and Patrick R. McMullen (2004). “Ant Colony Optimization Techniques for the Vehicle Routing Problem”. In: *Advanced Engineering Informatics* 18.1, pp. 41–48. ISSN: 1474-0346. DOI: `https://doi.org/10.1016/j.aei.2004.07.001`. URL: `http://www.sciencedirect.com/science/article/pii/S1474034604000060`.

Bentley, Jon Jouis (1992). “Fast Algorithms for Geometric Traveling Salesman Problems”. In: *ORSA Journal On Computing* 4.4, pp. 387–411.

Birattari, Mauro (2004). *On the Estimation of the Expected Performance of a Metaheuristic on a Class of Instances. How Many Instances, How Many Runs?* Tech. rep. TR/IRIDIA/2004-001. Brussels, Belgium: IRIDIA, Université Libre de Bruxelles.

Boeringer, Daniel W and Douglas H Werner (2004). “Particle Swarm Optimization Versus Genetic Algorithms for Phased Array Synthesis”. In: *IEEE Transactions on Antennas and Propagation* 52.3, pp. 771–779.

Botta, A., W. de Donato, V. Persico, and A. Pescapé (2014). “On the Integration of Cloud Computing and Internet of Things”. In: *2014 International Conference on Future Internet of Things and Cloud*, pp. 23–30.

Braun, Heinrich (1990). “On solving travelling salesman problems by genetic algorithms”. In: *International Conference on Parallel Problem Solving from Nature*. Springer, pp. 129–133.

Bremermann, Hans J et al. (1962). “Optimization Through Evolution and Recombination”. In: *Self-organizing Systems* 93, p. 106.

Bullnheimer, Bernd, Richard F Hartl, and Christine Strauss (1997). “A New Rank Based Version of the Ant System. A Computational Study.” In.

Bullnheimer, Bernd, Gabriele Kotsis, and Christine Strauß (1998a). “Parallelization Strategies for the Ant System”. In: *High Performance Algorithms and Software in Nonlinear Optimization*. Springer, pp. 87–100.

— (1998b). “Parallelization Strategies for the Ant System”. In: *High Performance Algorithms and Software in Nonlinear Optimization*. Springer, pp. 87–100.

Cecilia, José M, José M García, Manuel Ujaldón, Andy Nisbet, and Martyn Amos (May 2011). "Parallelization Strategies for Ant Colony Optimisation on GPUs". In: *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pp. 339–346. DOI: 10.1109/IPDPS.2011.170.

Cecilia, José M., José M. García, Andy Nisbet, Martyn Amos, and Manuel Ujaldón (2013). "Enhancing Data Parallelism for Ant Colony Optimization on GPUs". In: *Journal of Parallel and Distributed Computing* 73.1, pp. 42–51. ISSN: 07437315. DOI: 10.1016/j.jpdc.2012.01.002.

Cecilia, José M., Andy Nisbet, Martyn Amos, José M. García, and Manuel Ujaldón (2013). "Enhancing GPU Parallelism in Nature-inspired Algorithms". In: *The Journal of Supercomputing* 63.3, pp. 773–789. ISSN: 0920-8542. DOI: 10.1007/s11227-012-0770-1. URL: http://link.springer.com/10.1007/s11227-012-0770-1.

Chen, Ling and Chunfang Zhang (2005). "Adaptive Parallel Ant Colony Algorithm". In: *International Conference on Natural Computation*. Springer, pp. 1239–1249.

Cheng, TC Edwin and Bertrand MT Lin (2009). "Johnson's rule, composite jobs and the relocation problem". In: *European Journal of Operational Research* 192.3, pp. 1008–1013.

Chitty, Darren M (2017). "Applying ACO to Large Scale TSP Instances". In: *UK Workshop on Computational Intelligence*. Springer, pp. 104–118.

Chrysos, George (2014). "Intel® Xeon Phi™ Coprocessor - the Architecture". In: *Intel Whitepaper* 176, p. 43.

Clark, Christopher, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield (2005). "Live Migration of Virtual Machines". In: *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pp. 273–286.

Colorni, Alberto, Marco Dorigo, Vittorio Maniezzo, et al. (1991). "Distributed Optimization by Ant Colonies". In: *Proceedings of the First European Conference on Artificial Life*. Vol. 142. Paris, France, pp. 134–142.

Colorni, Alberto, Marco Dorigo, Vittorio Maniezzo, and Marco Trubian (1994). "Ant System for Job-shop Scheduling". In: *JORBEL-Belgian Journal of Operations Research, Statistics, and Computer Science* 34.1, pp. 39–53.

Cordón García, Oscar, Iñaki Fernández de Viana, and Francisco Herrera Triguero (2002). "Analysis of the Best-Worst Ant System and its Variants on the TSP". In: *Mathware & Soft Computing. 2002 Vol. 9 Núm. 2 [-3].*

Cover, Thomas and Peter Hart (1967). "Nearest Neighbor Pattern Classification". In: *IEEE Transactions on Information Theory* 13.1, pp. 21–27.

Cunningham, Padraig and Sarah Jane Delany (2020). "K-Nearest Neighbour Classifiers". In: *arXiv preprint arXiv:2004.04523*.

Dawson, Laurence (2015). "Generic Techniques in General Purpose GPU Programming with Applications to Ant Colony and Image Processing Algorithms". PhD thesis. Durham University, UK.

Dawson, Laurence and Iain Stewart (2013a). "Candidate Set Parallelization Strategies for Ant Colony Optimization on the GPU". In: *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, pp. 216–225.

— (2013b). "Improving Ant Colony Optimization Performance on the GPU using CUDA". In: *Evolutionary Computation (CEC), 2013 IEEE Congress on*. IEEE, pp. 1901–1908.

Dawson, Laurence and Iain A Stewart (June 2013c). "Improving Ant Colony Optimization performance on the GPU using CUDA". In: *2013 IEEE Congress on Evolutionary Computation*, pp. 1901–1908. DOI: 10.1109/CEC.2013.6557791.

Díaz, Diego, Pablo Valledor, Borja Ena, Miguel Iglesias, and César Menéndez Fernández (Sept. 2020). "Improved Method for Parallelization of Evolutionary Metaheuristics". In: *Mathematics* 8, p. 1476. DOI: 10.3390/math8091476.

Doerner, Karl F, Richard F Hartl, Siegfried Benkner, and Maria Lucka (2006). "Parallel Cooperative Savings Based Ant Colony Optimization—Multiple Search and Decomposition Approaches". In: *Parallel Processing Letters* 16.03, pp. 351–369.

Donelli, Massimo, Renzo Azaro, Francesco GB De Natale, and Andrea Massa (2006). "An Innovative Computational Approach Based on a Particle Swarm Strategy for Adaptive Phased-arrays Control". In: *IEEE Transactions on Antennas and Propagation* 54.3, pp. 888–898.

Dorigo, Marco and Mauro Birattari (2011). "Ant Colony Optimization". In: *Encyclopedia of Machine Learning*. Springer, pp. 36–39.

Dorigo, Marco, Mauro Birattari, and Thomas Stutzle (2006). "Ant Colony Optimization". In: *IEEE Computational Intelligence Magazine* 1.4, pp. 28–39.

Dorigo, Marco and Gianna Di Caro (1999). “Ant Colony Optimization: a New Meta-heuristic”. In: *Proceedings of the 1999 Congress on Evolutionary Computation*. Vol. 2.

Dorigo, Marco and Luca Maria Gambardella (1997a). “Ant Colonies for the Travelling Salesman Problem”. In: *BioSystems* 43.2, pp. 73–81.

— (1997b). “Ant Colony System: a cooperative learning approach to the Traveling Salesman Problem”. In: *Evolutionary Computation, IEEE Transactions on* 1.1, pp. 53–66.

Dorigo, Marco, Vittorio Maniezzo, and Alberto Colorni (1991). “The Ant System: An Autocatalytic Optimizing Process”. In.

— (1996). “Ant System: Optimization by a Colony of Cooperating Agents”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 26.1, pp. 29–41.

Dorigo, Marco and Thomas Stützle (2004). *Ant Colony Optimization*. Scituate, MA, USA: Bradford Company.

Du, Lingling and Ruhan He (2012). “Combining Nearest Neighbor Search with Tabu Search for Large-scale Vehicle Routing Problem”. In: *Physics Procedia* 25, pp. 1536–1546.

Eberhart, Russell C and Yuhui Shi (2001). “Tracking and Optimizing Dynamic Systems with Particle Swarms”. In: *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No. 01TH8546)*. Vol. 1. IEEE, pp. 94–100.

Elsheikh, AH and M Abd Elaziz (2019). “Review on Applications of Particle Swarm Optimization in Solar Energy Systems”. In: *International Journal of Environmental Science and Technology* 16.2, pp. 1159–1170.

Farmer, J Doyne, Norman H Packard, and Alan S Perelson (1986). “The Immune System, Adaptation, and Machine Learning”. In: *Physica D: Nonlinear Phenomena* 22.1-3, pp. 187–204.

Feller, Eugen, Louis Rilling, and Christine Morin (2011). “Energy-aware Ant Colony Based Workload Placement in Clouds”. In: *2011 IEEE/ACM 12th International Conference on Grid Computing*. IEEE, pp. 26–33.

Fernández-Cerero, Damián, Alejandro Fernández-Montes, and Agnieszka Jakóbik (2020). “Limiting Global Warming by Improving Data-centre Software”. In: *IEEE Access* 8, pp. 44048–44062.

Fogel, Lawrence J, Aalvin J. Owens, and Michael John Walsh (1966). *Artificial Intelligence through Simulated Evolution*.

Friedberg, Richard M (1958). "A Learning Machine: Part I". In: *IBM Journal of Research and Development* 2.1, pp. 2–13.

Friesen, Donald and Michael Langston (Feb. 1986). "Variable Sized Bin Packing". In: *SIAM Journal on Computing* 15, pp. 222–230. DOI: 10.1137/0215016.

Fu, Jie, Lin Lei, and Guohua Zhou (2010). "A Parallel Ant Colony Optimization algorithm with GPU-acceleration based on All-In-Roulette selection". In: *Advanced Computational Intelligence (IWACI), 2010 Third International Workshop on*, pp. 260–264.

Al-Fuqaha, A., M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash (2015). "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications". In: *IEEE Communications Surveys Tutorials* 17.4, pp. 2347–2376.

Gambardella, Luca M and Marco Dorigo (1995). "Ant-Q: A Reinforcement Learning Approach to the Traveling Salesman Problem". In: *Machine Learning Proceedings 1995*. Elsevier, pp. 252–260.

Gambardella, Luca Maria, Éric Taillard, and Giovanni Agazzi (1999). "MACS-VRPTW: A Multiple Colony System for Vehicle Routing Problems with Time Windows". In: *New Ideas in Optimization*. Citeseer.

Gao, Yongqiang, Haibing Guan, Zhengwei Qi, Yang Hou, and Liang Liu (2013). "A Multi-objective Ant Colony System Algorithm for Virtual Machine Placement in Cloud Computing". In: *Journal of computer and system sciences* 79.8, pp. 1230–1242.

Ghosh, Manosij, Ritam Guha, Ram Sarkar, and Ajith Abraham (2019). "A Wrapper-filter Feature Selection Technique Based on Ant Colony Optimization". In: *Neural Computing and Applications*, pp. 1–19.

Goldberg, David E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201157675.

Grassé, Pierre-Paul (1986). *Termitologia. Anatomie, physiologie, biologie-systematique des termites. Tome III: Comportement, socialite, ecologie, evolution, systematique*.

Guerrero-ibanez, J. A., S. Zeadally, and J. Contreras-Castillo (2015). "Integration Challenges of Intelligent Transportation Systems with Connected Vehicle, Cloud Computing, and Internet of Things Technologies". In: *IEEE Wireless Communications* 22.6, pp. 122–128.

Guntsch, Michael and Martin Middendorf (2002). “A Population Based Approach for ACO”. In: *Workshops on Applications of Evolutionary Computation*. Springer, pp. 72–81.

Hajihassani, M, D Jahed Armaghani, and R Kalatehjari (2018). “Applications of Particle Swarm Optimization in Geotechnical Engineering: a Comprehensive Review”. In: *Geotechnical and Geological Engineering* 36.2, pp. 705–722.

Hayes, Brian (2008). *Cloud Computing*.

Helms, Michael, Swaroop S Vattam, and Ashok K Goel (2009). “Biologically Inspired Design: Process and Products”. In: *Design studies* 30.5, pp. 606–622.

Helsgaun, Keld (2000). “An effective implementation of the Lin–Kernighan traveling salesman heuristic”. In: *European journal of operational research* 126.1, pp. 106–130.

Hendtlass, Tim (2001). “A Combined Swarm Differential Evolution Algorithm for Optimization Problems”. In: *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*. Springer, pp. 11–18.

Holland, John H (1975). “Adaptation in Natural and Artificial Systems. An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence”. In: *anas*.

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

Jiening, Wang, Dong Jiankang, and Zhang Chunfeng (Aug. 2009). “Implementation of Ant Colony Algorithm Based on GPU”. In: *2009 Sixth International Conference on Computer Graphics, Imaging and Visualization*. IEEE, pp. 50–53. ISBN: 978-0-7695-3789-4. DOI: `10.1109/CGIV.2009.20`. URL: `http://ieeexplore.ieee.org/document/5298367/`.

Johnson, David S and Lyle A McGeoch (1997). “The Traveling Salesman Problem: A Case Study in Local Optimization”. In: *Local Search in Combinatorial Optimization* 1.1, pp. 215–310.

Jr, EG Co man, MR Garey, and DS Johnson (1996). “Approximation Algorithms for Bin Packing: A Survey”. In: *Approximation Algorithms for NP-hard Problems*, pp. 46–93.

Karaboga, D. (2005). *An Idea Based on Honey Bee Swarm for Numerical Optimization*. Tech. rep.

Karaboga, Dervis and Bahriye Akay (2007). “Artificial Bee Colony (ABC) Algorithm on Training Artificial Neural Networks”. In: *2007 IEEE 15th Signal Processing and Communications Applications*. IEEE, pp. 1–4.

Karaboga, Dervis, Bahriye Akay, and Celal Ozturk (2007). “Artificial Bee Colony (ABC) Optimization Algorithm for Training Feed-forward Neural Networks”. In: *International Conference on Modeling Decisions for Artificial Intelligence*. Springer, pp. 318–329.

Karaboga, Dervis and Beyza Gorkemli (2011). “A Combinatorial Artificial Bee Colony Algorithm for Traveling Salesman Problem”. In: *2011 International Symposium on Innovations in Intelligent Systems and Applications*. IEEE, pp. 50–53.

Karaboga, Dervis and Celal Ozturk (2009). “Neural Networks Training by Artificial Bee Colony Algorithm on Pattern Classification”. In: *Neural Network World* 19.3, p. 279.

Kennedy, J. and R. Eberhart (1995). “Particle Swarm Optimization”. In: *Proceedings of ICNN’95 - International Conference on Neural Networks*. Vol. 4, 1942–1948 vol.4.

Kennedy, James and Russell C Eberhart (1997). “A Discrete Binary Version of the Particle Swarm Algorithm”. In: *1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation*. Vol. 5. IEEE, pp. 4104–4108.

Khodier, Majid M and Christos G Christodoulou (2005). “Linear Array Geometry Synthesis with Minimum Sidelobe Level and Null Control Using Particle Swarm Optimization”. In: *IEEE Transactions on Antennas and Propagation* 53.8, pp. 2674–2679.

Lawler, Eugene L (1985). “The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization”. In: *Wiley-Interscience Series in Discrete Mathematics*.

Leguizamon, Guillermo and Zbigniew Michalewicz (1999). “A New Version of Ant System for Subset Problems”. In: *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*. Vol. 2. IEEE, pp. 1459–1464.

Li, Wei Hua, Wei Jia Li, Yuan Yang, Hai Qiang Liao, Ji Long Li, and Xi Peng Zheng (2011). “Artificial Bee Colony Algorithm for Traveling Salesman Problem”. In: *Advanced Materials Research*. Vol. 314. Trans Tech Publ, pp. 2191–2196.

Li, Yingmin, David Brooks, Zhigang Hu, Kevin Skadron, and Pradip Bose (2004). “Understanding the Energy Efficiency of Simultaneous Multithreading”. In: *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, pp. 44–49.

Liu, Xiao-Fang, Zhi-Hui Zhan, Jeremiah D Deng, Yun Li, Tianlong Gu, and Jun Zhang (2016). “An Energy Efficient Ant Colony System for Virtual Machine Placement in Cloud Computing”. In: *IEEE Transactions on Evolutionary Computation* 22.1, pp. 113–128.

Lloyd, Huw and Martyn Amos (2016). “A Highly Parallelized and Vectorized Implementation of Max-Min Ant System on Intel® Xeon Phi™”. In: *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, pp. 1–6. ISBN: 978-1-5090-4240-1. DOI: `10.1109/SSCI.2016.7850085`. URL: `http://ieeexplore.ieee.org/document/7850085/`.

— (2017). “Analysis of Independent Roulette Selection in Parallel Ant Colony Optimization”. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO ’17. Berlin, Germany: ACM, pp. 19–26. ISBN: 978-1-4503-4920-8. DOI: `10.1145/3071178.3071308`. URL: `http://doi.acm.org/10.1145/3071178.3071308`.

— (2020). “Solving Sudoku With Ant Colony Optimization”. In: *IEEE Transactions on Games* 12.3, pp. 302–311.

Lomont, Chris (2011). “Introduction to Intel Advanced Vector Extensions”. In: *Intel White Paper* 23.

López-Ibáñez, Manuel, Thomas Stützle, and Marco Dorigo (2016). “Ant Colony Optimization: A Component-Wise Overview”. In: *Handbook of Heuristics*. Ed. by Rafael Martí, Pardalos Panos, and Mauricio G.C. Resende. Cham: Springer International Publishing, pp. 1–37. ISBN: 978-3-319-07153-4. DOI: `10.1007/978-3-319-07153-4_21-1`. URL: `https://doi.org/10.1007/978-3-319-07153-4_21-1`.

Manfrin, Max, Mauro Birattari, Thomas Stützle, and Marco Dorigo (2006). “Parallel Ant Colony Optimization for the Traveling Salesman Problem”. In: *International Workshop on Ant Colony Optimization and Swarm Intelligence*. Springer, pp. 224–234.

Maniezzo, V, L Muzio, A Colorni, and M Dorigo (1994). *Il Sistema Formiche Applicato al Problema Dell’assegnamento Quadratico*. Tech. rep. Technical Report.

Maniezzo, Vittorio and Alberto Colorni (1999). “The Ant System Applied to the Quadratic Assignment Problem”. In: *IEEE Transactions on knowledge and data engineering* 11.5, pp. 769–778.

Manne, Alan S (1960). “On the job-shop scheduling problem”. In: *Operations research* 8.2, pp. 219–223.

Martínez, Pablo A and José M García (2021). “ACOTSP-MF: A Memory-friendly and Highly Scalable ACOTSP Approach”. In: *Engineering Applications of Artificial Intelligence* 99, p. 104131.

Masoumzadeh, Seyed Saeid and Helmut Hlavacs (2013). “Integrating VM Selection Criteria in Distributed Dynamic VM consolidation using Fuzzy Q-Learning”. In: *Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013)*. IEEE, pp. 332–338.

Mavrovouniotis, Michalis, Charalambos Menelaou, Stelios Timotheou, Georgios Ellinas, Christos Panayiotou, and Marios Polycarpou (July 2020). “A Benchmark Test Suite for the Electric Capacitated Vehicle Routing Problem”. In: DOI: 10.1109/CEC48606.2020.9185753.

Mavrovouniotis, Michalis, Felipe M. Müller, and Shengxiang Yang (2017). “Ant Colony Optimization With Local Search for Dynamic Traveling Salesman Problems”. In: *IEEE Transactions on Cybernetics* 47.7, pp. 1743–1756. ISSN: 2168-2267. DOI: 10.1109/TCYB.2016.2556742.

Merkle, Daniel, Martin Middendorf, and Hartmut Schmeck (2002). “Ant Colony Optimization for Resource-constrained Project Scheduling”. In: *IEEE Transactions on Evolutionary Computation* 6.4, pp. 333–346. ISSN: 1089-778X. DOI: 10.1109/TEVC.2002.802450.

Mi, Haibo, Huaimin Wang, Gang Yin, Yangfan Zhou, Dianxi Shi, and Lin Yuan (2010). “Online Self-reconfiguration with Performance Guarantee for Energy-efficient Large-scale Cloud Computing Data Centers”. In: *2010 IEEE International Conference on Services Computing*. IEEE, pp. 514–521.

Middendorf, Martin, Frank Reischle, and Hartmut Schmeck (2002). “Multi Colony Ant Algorithms”. In: *Journal of Heuristics* 8, pp. 305–320. URL: https://link.springer.com/content/pdf/10.1023%7B%5C%%7D2FA%7B%5C%%7D3A1015057701750.pdf.

Mirjalili, Seyedali (2016). “SCA: A Sine Cosine Algorithm for Solving Optimization Problems”. In: *Knowledge-Based Systems* 96, pp. 120–133. ISSN: 0950-7051. DOI: https://doi.org/10.1016/j.knosys.2015.12.022.

URL: `http://www.sciencedirect.com/science/article/pii/S0950705115005043`.

Mirjalili, Seyedali, Seyed Mohammad Mirjalili, and Andrew Lewis (2014). "Grey Wolf Optimizer". In: *Advances in Engineering Software* 69, pp. 46–61. ISSN: 0965-9978. DOI: `https://doi.org/10.1016/j.advengsoft.2013.12.007`. URL: `http://www.sciencedirect.com/science/article/pii/S0965997813001853`.

Moscato, Pablo et al. (1989). "On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts: Towards Memetic Algorithms". In: *Caltech Concurrent Computation Program, C3P Report* 826, p. 1989.

Muja, Marius and David G Lowe (2014). "Scalable Nearest Neighbor Algorithms for High Dimensional Data". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36.11, pp. 2227–2240.

Oliveira, Iona Maghali S de, Roberto Schirru, and Jose ACC Medeiros (2009). *On the Performance of an Artificial Bee Colony Optimization Algorithm Applied to the Accident Ciagnosis in a PWR Nuclear Power Plant.*

Peake, Joshua, Martyn Amos, Paraskevas Yiapanis, and Huw Lloyd (2018). "Vectorized Candidate Set Selection for Parallel Ant Colony Optimization". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM, pp. 1300–1306.

— (2019). "Scaling Techniques for Parallel Ant Colony Optimization on Large Problem Instances". In: *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 47–54.

Peake, Joshua, Nicholas Costen, Giovanni Masala, Martyn Amos, and Huw Lloyd (2021). *PACO-VMP: Parallel Ant Colony Optimization for Virtual Machine Placement.*

Randall, Marcus and Andrew Lewis (2002). "A Parallel Implementation of Ant Colony Optimization". In: *Journal of Parallel and Distributed Computing* 62.9, pp. 1421–1432.

Randall, Marcus and James Montgomery (2002). "Candidate Set Strategies for Ant Colony Optimisation". In: *International Workshop on Ant Algorithms*. Springer, pp. 243–249.

Reinelt, Gerhard (1991). "TSPLIB - A Traveling Salesman Problem library". In: *ORSA Journal on Computing* 3.4, pp. 376–384. DOI: `10.1287/ijoc.3.4.376`.

Reynolds, Robert G (1994). “An Introduction To Cultural Algorithms”. In: *Proceedings of the Third Annual Conference on Evolutionary Programming*. World Scientific, pp. 131–139.

Robinson, Jacob, Seelig Sinton, and Yahya Rahmat-Samii (2002). “Particle Swarm, Genetic Algorithm, and Their Hybrids: Optimization of a Profiled Corrugated Horn Antenna”. In: *IEEE Antennas and Propagation Society International Symposium (IEEE Cat. No. 02CH37313)*. Vol. 1. IEEE, pp. 314–317.

Sato, Mikiko, Shigeyoshi Tsutsui, Noriyuki Fujimoto, Yuji Sato, and Mitaro Namiki (2014). “First Results of Performance Comparisons on Many-Core Processors in Solving QAP with ACO: Kepler GPU Versus Xeon Phi”. In: *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM, pp. 1477–1478.

Satyanarayanan, M. (2017). “The Emergence of Edge Computing”. In: *Computer* 50.1, pp. 30–39.

Selvan, S Easter, C Cecil Xavier, Nico Karssemeijer, Jean Sequeira, Rekha A Cherian, and Bharathi Y Dhala (2006). “Parameter Estimation in Stochastic Mammogram Model by Heuristic Optimization Techniques”. In: *IEEE Transactions on Information Technology in Biomedicine* 10.4, pp. 685–695.

Shchur, Lev N and Paolo Butera (1998). “The RANLUX Generator: Resonances in a Random Walk Test”. In: *International Journal of Modern Physics C* 9.04, pp. 607–624.

Shi, Xiaohu, Yanwen Li, Haijun Li, Renchu Guan, Liupu Wang, and Yanchun Liang (2010). “An Integrated Algorithm Based on Artificial Bee Colony and Particle Swarm Optimization”. In: *2010 Sixth International Conference on Natural Computation*. Vol. 5. IEEE, pp. 2586–2590.

Shi, Yuhui and Russell C. Eberhart (1998). “Parameter Selection in Particle Swarm Optimization”. In: *Evolutionary Programming VII*. Ed. by V. W. Porto, N. Saravanan, D. Waagen, and A. E. Eiben. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 591–600. ISBN: 978-3-540-68515-9.

Shmygelska, Alena and Holger H Hoos (2005). “An Ant Colony Optimisation Algorithm for the 2D and 3D Hydrophobic Polar Protein Folding Problem”. In: *BMC Bioinformatics* 6.1, p. 30.

Skinderowicz, Rafał (2012). “Ant Colony System with Selective Pheromone Memory for TSP”. In: *International Conference on Computational Collective Intelligence*. Springer, pp. 483–492.

Skinderowicz, Rafał (2016). “The GPU-based parallel Ant Colony System”. In: *Journal of Parallel and Distributed Computing* 98.Supplement C, pp. 48–60. ISSN: 0743-7315. DOI: `https://doi.org/10.1016/j.jpdc.2016.04.014`. URL: `http://www.sciencedirect.com/science/article/pii/S0743731516300284`.

Sodani, Avinash, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu (2016). “Knights Landing: Second-generation Intel Xeon Phi Product”. In: *IEEE Micro* 36.2, pp. 34–46.

*Star Tours* (n.d.). `http://www.math.uwaterloo.ca/tsp/star/`. Accessed: 2021-10-22.

Stillwell, Mark, David Schanzenbach, Frédéric Vivien, and Henri Casanova (2010). “Resource allocation algorithms for virtualized service hosting platforms”. In: *Journal of Parallel and distributed Computing* 70.9, pp. 962–974.

Stützle, Thomas (1998). “Parallelization Strategies for Ant Colony Optimization”. In: *International Conference on Parallel Problem Solving from Nature*. Springer, pp. 722–731.

— (2004). *ACOTSP*. Available from http://www.aco-metaheuristic.org/aco-code, 2004.

Stützle, Thomas and Marco Dorigo (1999). “ACO Algorithms for the Traveling Salesman Problem”. In: *Evolutionary Algorithms in Engineering and Computer Science* 4, pp. 163–183.

Stützle, Thomas and Holger H Hoos (2000). “MAX-MIN Ant System”. In: *Future Generation Computer Systems* 16.8, pp. 889–914.

Szeto, Wai Yuen, Yongzhong Wu, and Sin C Ho (2011). “An Artificial Bee Colony Algorithm for the Capacitated Vehicle Routing Problem”. In: *European Journal of Operational Research* 215.1, pp. 126–135.

Taillard, Éric D. and Keld Helsgaun (2019). “POPMUSIC for the travelling salesman problem”. In: *European Journal of Operational Research* 272.2, pp. 420–429. ISSN: 0377-2217. DOI: `https://doi.org/10.1016/j.ejor.2018.06.039`. URL: `https://www.sciencedirect.com/science/article/pii/S0377221718305745`.

Tao, Fei, Chen Li, T Warren Liao, and Yuanjun Laili (2015). “BGM-BLA: a New Algorithm for Dynamic Migration of Virtual Machines in Cloud Computing”. In: *IEEE Transactions on Services Computing* 9.6, pp. 910–925.

Tereshko, Valery and Andreas Loengarov (2005). “Collective decision making in honey-bee foraging dynamics”. In: *Computing and Information Systems* 9.3, p. 1.

Teukolsky, Saul A, Brian P Flannery, William H Press, and William T Vetterling (1992). *Numerical Recipes in C*.

Thimbleby, Harold (2003). “The directed chinese postman problem”. In: *Software: Practice and Experience* 33.11, pp. 1081–1096.

Thomson, WE (1958). “A Modified Congruence Method of Generating Pseudo-random Numbers”. In: *The Computer Journal* 1.2, pp. 83–83.

Tian, Xinmin, Hideki Saito, Serguei V. Preis, Eric N. Garcia, Sergey S. Kozhukhov, Matt Masten, Aleksei G. Cherkasov, and Nikolay Panchenko (2013). “Practical SIMD Vectorization Techniques for Intel® Xeon Phi Coprocessors”. In: *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, pp. 1149–1158. ISBN: 978-0-7695-4979-8. DOI: 10.1109/IPDPSW.2013.245. URL: http://ieeexplore.ieee.org/document/6651001/.

Tirado, Felipe, Ricardo J. Barrientos, Paulo González, and Marco Mora (2017). “Efficient Exploitation of the Xeon Phi Architecture for the Ant Colony Optimization (ACO) Metaheuristic”. In: *The Journal of Supercomputing* 73.11, pp. 5053–5070. ISSN: 0920-8542. DOI: 10.1007/s11227-017-2124-5. URL: http://link.springer.com/10.1007/s11227-017-2124-5.

Tirado, Felipe, Angelica Urrutia, and Ricardo J. Barrientos (Nov. 2015). “Using a Co-processor to Solve the Ant Colony Optimization Algorithm”. In: *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*, pp. 1–6. DOI: 10.1109/SCCC.2015.7416584.

Toth, Paolo and Daniele Vigo (2002). *The vehicle routing problem*. SIAM.

*TSP Art Instances* (n.d.). http://www.math.uwaterloo.ca/tsp/data/art/. Accessed: 2019-01-30.

Turing, AM (1950). “Computing Machinery and Intelligence”. In: *Mind* 59, pp. 433–460.

Twomey, Colin, Thomas Stützle, Marco Dorigo, Max Manfrin, and Mauro Birattari (2010). “An Analysis of Communication Policies for Homogeneous Multi-colony ACO algorithms”. In: *Information Sciences* 180.12, pp. 2390–2404.

Ullman, Jeffrey D. (1971). “The performance of a memory allocation algorithm”. In.

Uthayakumar, J, Noura Metawa, K Shankar, and SK Lakshmanaprabu (2020). “Financial Crisis Prediction Model using Ant Colony Optimization”. In: *International Journal of Information Management* 50, pp. 538–556.

Wang, Jiquan, Okan K Ersoy, Mengying He, and Fulin Wang (2016). “Multi-offspring genetic algorithm and its application to the traveling salesman problem”. In: *Applied Soft Computing* 43, pp. 415–423.

Wang, Kang-Ping, Lan Huang, Chun-Guang Zhou, and Wei Pang (2003). “Particle Swarm Optimization for Traveling Salesman Problem”. In: *Proceedings of the 2003 International Conference on Machine Learning and Cybernetics (IEEE cat. no. 03ex693)*. Vol. 3. IEEE, pp. 1583–1585.

Wang, Shangguang, Zhipiao Liu, Zibin Zheng, Qibo Sun, and Fangchun Yang (2013). “Particle Swarm Optimization for Energy-aware Virtual Machine Placement Optimization in Virtualized Data Centers”. In: *2013 International Conference on Parallel and Distributed Systems*. IEEE, pp. 102–109.

Watkins, Christopher JCH and Peter Dayan (1992). “Q-Learning”. In: *Machine Learning* 8.3-4, pp. 279–292.

Wilcox, David, Andrew McNabb, and Kevin Seppi (2011). “Solving Virtual Machine Packing with a Reordering Grouping Genetic Algorithm”. In: *2011 IEEE Congress of Evolutionary Computation (CEC)*. IEEE, pp. 362–369.

Wilson, Edward O. (1962). “Chemical Communication Among Workers of the Fire Ant Solenopsis Saevissima (Fr. Smith) 1. The Organization of Mass-Foraging”. In: *Animal Behaviour* 10.1, pp. 134–147. ISSN: 0003-3472. DOI: `https://doi.org/10.1016/0003-3472(62)90141-0`. URL: `http://www.sciencedirect.com/science/article/pii/0003347262901410`.

*World TSP* (n.d.). `https://www.math.uwaterloo.ca/tsp/world/`. Accessed: 2021-10-22.

Yao, B, C Yang, J Hu, and B Yu (2010). “The Optimization of Urban Subway Routes Based on Artificial Bee Colony Algorithm”. In: *Key Technologies of Railway Engineering—High Speed Railway, Heavy Haul Railway and Urban Rail Transit. Beijing Jiaotong University, Beijing*, pp. 747–751.

You, Ying-Shiuan (2009). “Parallel Ant System for Traveling Salesman Problem on GPUs”. In: *Eleventh Annual Conference on Genetic and Evolutionary Computation*. sn, pp. 1–2.

Zanella, A., N. Bui, A. Castellani, L. Vangelista, and M. Zorzi (2014). “Internet of Things for Smart Cities”. In: *IEEE Internet of Things Journal* 1.1, pp. 22–32.

Zhang, Hong, Hoang Nguyen, Xuan-Nam Bui, Trung Nguyen-Thoi, Thu-Thuy Bui, Nga Nguyen, Diep-Anh Vu, Vinyas Mahesh, and Hossein Moayedi (2020). "Developing a Novel Artificial Intelligence Model to Estimate the Capital Cost of Mining Projects using Deep Neural Network-based Ant Colony Optimization Algorithm". In: *Resources Policy* 66, p. 101604.

Zhao, Fuqing, Qiuyu Zhang, Dongmei Yu, Xuhui Chen, and Yahong Yang (2005). "A Hybrid Algorithm Based on PSO and Simulated Annealing and its Applications for Partner Selection in Virtual Enterprise". In: *International Conference on Intelligent Computing*. Springer, pp. 380–389.

# Appendix A

# Vector Class

## A.1 Vector.h

```
#pragma once

#ifndef _VECTOR_INC_
#define _VECTOR_INC_
#include <immintrin.h>
#include <cstdio>
#include "platform.h"
#include <iostream>


class Vector
{

/*
  The specific types of fields of the Vector class change depending
    on the
  AVX instructions being used, but Vectors have three possible uses:
     storing
  floats, storing integers, or storing a mask. Each Vector,
    regardless of
  it's intended purpose, contains all three of the following:

  A float array/vector (values/AVXVec)
  An integer array/vector (iValues/AVXIntVec)
  A mask value. The SISD Vector stores masks as an iValues array,
  AVX512 uses the exclusive mmask16 type, and AVX2 uses an integer.
```

```
*/
public:
#ifdef SISD //fields used for the non-AVX vector implementation
  float AVXVec[8];
  int AVXIntVec[8];
  int vectorSize = 8;
#elif defined AVX512 //fields used for AVX512 compatible hardware
  __m512 AVXVec;
  __m512i AVXIntVec;
  __mmask16 maskVec;
#elif defined AVX2 //fields used for AVX2 compatible hardware
  __m256 AVXVec;
  __m256i AVXIntVec;
  int mask;
#endif
/*
  The + function for Vectors, adding the values of two Vector
  objects together lane-by-lane

  @param v | The Vector that will be added to the Vector that called
     the method
  @return | Returns a vector containing the results of adding the
    vectors
*/
  Vector operator+(const Vector& v) const
  {
    Vector vector;
#ifdef SISD
    for (int i = 0; i < vectorSize; i++)
    {
      vector.AVXVec[i] = this->AVXVec[i] + v.AVXVec[i];
    }
    return vector;
#elif defined AVX512
    vector.AVXVec = _mm512_add_ps(this->AVXVec, v.AVXVec);
    return vector;
#elif (defined AVX || defined AVX2)
    vector.AVXVec = _mm256_add_ps(this->AVXVec, v.AVXVec);
    return vector;
#endif
  }

```

```
/*
  The - function for Vectors, subtracting the values of the
    parameter Vector
  from the values of the calling Vector lane-by-lane

  @param v | The Vector that will be subtracted from the Vector that
     called the method
  @return | Returns a vector containing the results of subtracting
    the vectors
*/
  Vector operator-(const Vector& v) const
  {
    Vector vector;
#ifdef SISD
    for (int i = 0; i < vectorSize; i++)
    {
      vector.AVXVec[i] = (this->AVXVec[i] - v.AVXVec[i]);
    }
    return vector;
#elif defined AVX512
    vector.AVXVec = _mm512_sub_ps(this->AVXVec, v.AVXVec);
#elif (defined AVX || defined AVX2)
    vector.AVXVec = _mm256_sub_ps(this->AVXVec, v.AVXVec);
    return vector;
#endif
  }

/*
  The * function for Vectors, multiplying the values of two Vector
  objects together lane-by-lane

  @param v | The Vector that will be multiplied by the Vector that
    called the method
  @return | Returns a vector containing the results of multiplying
    the vectors
*/
  Vector operator*(const Vector& v) const
  {
    Vector vector;
#ifdef SISD
    for (int i = 0; i < vectorSize; i++)
    {
      vector.AVXVec[i] = (this->AVXVec[i] * v.AVXVec[i]);
    }
    return vector;
#elif defined AVX512
    vector.AVXVec = _mm512_mul_ps(this->AVXVec, v.AVXVec);
    return vector;
#elif (defined AVX || defined AVX2)
    vector.AVXVec = _mm256_mul_ps(this->AVXVec, v.AVXVec);
    return vector;
#endif
  }

  Vector operator*(const int v) const
  {
    Vector vector;
    Vector intVec;
    intVec.set1(v);
#ifdef SISD
    for (int i = 0; i < vectorSize; i++)
    {
      vector.AVXVec[i] = (this->AVXVec[i] * intVec.AVXVec[i]);
    }
    return vector;
#elif defined AVX512
    vector.AVXVec = _mm512_mul_ps(this->AVXVec, intVec.AVXVec);
    return vector;
#elif (defined AVX || defined AVX2)
    vector.AVXVec = _mm256_mul_ps(this->AVXVec, intVec.AVXVec);
    return vector;
#endif
  }

  Vector operator/(const Vector& v) const
  {
    Vector vector;
#ifdef SISD
    for (int i = 0; i < vectorSize; i++)
    {
      vector.AVXVec[i] = (this->AVXVec[i] / v.AVXVec[i]);
    }
    return vector;
#elif defined AVX512
```

```
    vector.AVXVec = _mm512_div_ps(this->AVXVec, v.AVXVec);
    return vector;
#elif (defined AVX || defined AVX2)
    vector.AVXVec = _mm256_div_ps(this->AVXVec, v.AVXVec);
    return vector;
#endif
  }

/*
  Enforces a maximum value upon a vector, reducing all values that
   are above
  the maximum to the maximum.

  @param maxVal | The maximum allowed value
*/
  void vecMax(float maxVal)
  {
#ifdef SISD
    for (int i = 0; i < vectorSize; i++)
    {
      if (this->AVXVec[i] > maxVal)
      {
        this->AVXVec[i] = maxVal;
      }
    }
#elif defined AVX512
    __declspec(align(64)) float maxValAr[16] = { maxVal,maxVal,
   maxVal,maxVal,maxVal,maxVal,maxVal,maxVal,maxVal,maxVal,maxVal,
   maxVal,maxVal,maxVal,maxVal,maxVal };
    __m512 maxValVec = _mm512_load_ps(maxValAr);
    this->AVXVec = _mm512_min_ps(this->AVXVec, maxValVec);
#elif (defined AVX || defined AVX2)
    float maxValAr[8] __attribute__ ((aligned (32))) = { maxVal,
   maxVal,maxVal,maxVal,maxVal,maxVal,maxVal,maxVal};
    __m256 maxValVec = _mm256_load_ps(maxValAr);
    this->AVXVec = _mm256_min_ps(this->AVXVec, maxValVec);
#endif
  }
/*
  Enforces a minimum value upon a vector, increasing all values that
     are below
  the minimum to the minimum.

  @param minVal | The minimum allowed value
*/
  void vecMin(float minVal)
  {
#ifdef SISD
    for (int i = 0; i < vectorSize; i++)
    {
      if (this->AVXVec[i] < minVal)
      {
        this->AVXVec[i] = minVal;
      }
    }
#elif defined AVX512
    __declspec(align(64)) float minValAr[16] = { minVal,minVal,
    minVal,minVal,minVal,minVal,minVal,minVal,minVal,minVal,minVal,
    minVal,minVal,minVal,minVal,minVal };
    __m512 minValVec = _mm512_load_ps(minValAr);
    this->AVXVec = _mm512_max_ps(this->AVXVec, minValVec);
#elif (defined AVX || defined AVX2)
    float minValAr[8] __attribute__ ((aligned (32))) = { minVal,
    minVal,minVal,minVal,minVal,minVal,minVal,minVal};
    __m256 minValVec = _mm256_load_ps(minValAr);
    this->AVXVec = _mm256_max_ps(this->AVXVec, minValVec);
#endif
  }
/*
  Sets each Vector lane to one value

  @param setValue | The value that the Vector lanes will be set to
*/
  void set1(float setValue)
  {
#ifdef SISD
    for (int i = 0; i < vectorSize; i++)
    {
      this->AVXVec[i] = setValue;
    }
#elif defined AVX512
    this->AVXVec = _mm512_set1_ps(setValue);
#elif (defined AVX || defined AVX2)
    this->AVXVec = _mm256_set1_ps(setValue);
#endif
  }

/*
  Loads float data from a location in memory (normally an array)
    into a Vector

  @param source | The memory location of the data to be loaded
*/
  void load(float* source)
  {
#ifdef SISD
    for (int i = 0; i < vectorSize; i++)
    {
      this->AVXVec[i] = source[i];
    }
#elif defined AVX512
    this->AVXVec = _mm512_load_ps(source);
#elif (defined AVX || defined AVX2)
    this->AVXVec = _mm256_load_ps(source);
#endif
  }

  void load_u(float* source)
  {
#ifdef SISD
    for (int i = 0; i < vectorSize; i++)
    {
      this->AVXVec[i] = source[i];
    }
#elif defined AVX512
    this->AVXVec = _mm512_load_ps(source);
#elif (defined AVX || defined AVX2)
    this->AVXVec = _mm256_loadu_ps(source);
#endif
  }

/*
  Loads integer data from a location in memory (normally an array)
    into a Vector

  @param source | The memory location of the data to be loaded
```

```
*/

  void load(int* source)
  {
#ifdef SISD
    for (int i = 0; i < vectorSize; i++)
    {
      this->AVXVec[i] = source[i];
    }
#elif defined AVX512
    this->AVXIntVec = _mm512_load_epi32(source);
#elif (defined AVX || defined AVX2)
    this->AVXIntVec = _mm256_load_si256(((const __m256i *)source));
#endif
  }

/*
  Loads unsigned integer data from a location in memory (normally an
     array) into a Vector

  @param source | The memory location of the data to be loaded
*/

  void load(unsigned int * source)
  {
#ifdef SISD
    for (int i = 0; i < vectorSize; i++)
    {
      this->AVXVec[i] = source[i];
    }
#elif defined AVX512
    this->AVXIntVec = _mm512_load_epi32(source);
#elif defined AVX2
#endif
  }

/*
  Loads float data from a location in memory (normally an array)
    into a Vector. Applies
  a mask to the data, loading data from the calling Vector when the
    corresponding
  mask value is 1, and data from the src Vector when the
   corresponding value is 0.

  @param src | A vector containing data that will be added to the
   new vector is the
  corresponding mask value is 0
  @param mask | A vector of mask values (0 or 1) that determines
   which values will come
  from the calling Vector, and which values will come from src
  @param mem_addr | The memory location of the data to be loaded
*/

  void maskLoad(Vector &src, Vector &mask, float* mem_addr)
  {
#ifdef SISD

    for (int i = 0; i < vectorSize; i++)
    {
      if (mask.AVXVec[i] == 1)
        this->AVXVec[i] = mem_addr[i];
      else
        this->AVXVec[i] = src.AVXVec[i];
    }
#elif defined AVX512
    this->AVXVec = _mm512_mask_load_ps(src.AVXVec, mask.maskVec,
   mem_addr);
#elif (defined AVX || defined AVX2)
    Vector resultVector;
    resultVector.AVXVec = _mm256_load_ps(mem_addr);
    this->AVXVec = _mm256_blendv_ps(resultVector.AVXVec, src.AVXVec,
     mask.AVXVec);
#endif
  }
};

Vector int2mask(int maskInt);
Vector mask_mov(const Vector& v1, const Vector& bitMask, const
   Vector& v2);
Vector gtMask(const Vector& v1, const Vector& v2);
Vector ltMask(const Vector& v1, const Vector& v2);
Vector vecRandom(Vector& rC0, Vector& rC1, Vector& factors, Vector&
   rSeed);
```

```
Vector pow(const Vector& v1, const Vector& v2);
Vector max (const Vector& v1, const Vector& v2);
Vector abs(const Vector& v1);
void seedVecRandom(Vector& rC0, Vector& rC1, Vector& factors, int *
    seeds, Vector& rSeed);
void maxLocStep(Vector &oldWeights, Vector &oldIndices, Vector &
    newWeights, Vector &newIndices);
int reduceMax(Vector &curWeights, Vector &curIndices);
void store(float* loc, const Vector& v1);
void store(int* loc, const Vector& v1);
void printVec(Vector v1);

#endif
```

## A.2 Vector.cpp

```
/**
    vector.cpp
    Purpose: Contains all vector instructions for AVX512 and AVX2,
   as well as
  SISD variants for incompatible hardware.

    @author Joshua Peake
    @version 1.0
*/

#include "vector.h"
#include <iostream>
#include "platform.h"

/*
  Converts an integer to either an 8-wide or 16-wide mask vector,
   which
  is equivalent to the binary representation of that integer, for
   example:
  14291 becomes:
  0011011111010011 (AVX512) or 11010011 (AVX2)

  @param maskInt | The integer to be converted to a mask
  @return | The vector object containing the mask

```

```
*/
Vector int2mask(int maskInt) // NO AVX ---------------------------
{
#ifdef SISD
  Vector mask;
  for (int i = 0; i < _VECSIZE; ++i) {
    mask.AVXVec[i] = (maskInt >> i) & 1;
  }
  return mask;
#elif defined AVX512
  Vector resultVector;
  resultVector.maskVec = _mm512_int2mask(maskInt);
  return resultVector;
#elif (defined AVX || defined AVX2)
  Vector resultVector;
  float mask[8];
  for (int i = 0; i < 8; i++) {
    if ((maskInt >> i) & 1)
    {
      mask[i] = 1.0f;
    }
    else
    {
      mask[i] = -1.0f;
    }
  }
  resultVector.AVXVec = _mm256_setr_ps(mask[0], mask[1], mask[2],
     mask[3], mask[4], mask[5], mask[6], mask[7]);
  return resultVector;

#endif
}

/*
  Moves a vector from one vector to another, using a mask. Values
     are moved from
  v1 to resultVector if corresponding lane in bitmask is 0, or moved
     from v2 to
  resultVector if corresponding lane in bitmask is 1.

  @param v1 | A vector of values, usually weight data
  @param bitMask | A vector containing a bitMask, used to filter
   values from v1
  @param v2 | A vector usually containing one value multiple times,
   used for
  when values from v1 are masked
  @return | The filtered vector containing a combination of values
   from v1 and v2

*/
Vector mask_mov(const Vector& v1, const Vector& bitMask, const
    Vector& v2) // NO AVX -----------------
{
  Vector resultVector;
#ifdef SISD
  for (int i = 0; i < _VECSIZE; i++)
  {
    if (bitMask.AVXVec[i] == 0)
    {
      resultVector.AVXVec[i] = v1.AVXVec[i];
    }
    else
    {
      resultVector.AVXVec[i] = v2.AVXVec[i];
    }
  }
  return resultVector;
#elif defined AVX512
  resultVector.AVXVec = _mm512_mask_mov_ps(v1.AVXVec, bitMask.
   maskVec, v2.AVXVec);
  return resultVector;
#elif (defined AVX || defined AVX2)
  resultVector.AVXVec = _mm256_blendv_ps(v2.AVXVec, v1.AVXVec,
   bitMask.AVXVec);
  return resultVector;
#endif
}

/*
  Combines two vectors into a new vector, comparing vectors lane-by-
   lane and retaining the
  largest value.

  @param v1 | A vector of float values
  @param v2 | A vector of float values
  @return | The filtered vector containing a combination of values
    from v1 and v2

*/
Vector gtMask(const Vector& v1, const Vector& v2) // NO AVX
    --------------------
{
  Vector resultMask;
#ifdef SISD
  for (int i = 0; i < _VECSIZE; i++)
  {
    if (v1.AVXVec[i] > v2.AVXVec[i])
    {
      resultMask.AVXVec[i] = 1;
    }
    else
    {
      resultMask.AVXVec[i] = 0;
    }
  }
  return resultMask;
#elif defined AVX512
  resultMask.maskVec = _mm512_cmp_ps_mask(v1.AVXVec, v2.AVXVec,
    _MM_CMPINT_GT);
  return resultMask;
#elif (defined AVX || defined AVX2)
  resultMask.AVXVec = _mm256_cmp_ps(v1.AVXVec, v2.AVXVec, _CMP_GT_OS
    );
  return resultMask;
#endif
}

/*
  Combines two vectors into a new vector, comparing vectors lane-by-
    lane and retaining the
  lowest value.

  @param v1 | A vector of float values
  @param v2 | A vector of float values
  @return | The filtered vector containing a combination of values from v1 and v2

*/
Vector ltMask(const Vector& v1, const Vector& v2) // NO AVX ---------------------------
{
  Vector resultMask;
#ifdef SISD
  for (int i = 0; i < _VECSIZE; i++)
  {
    if (v1.AVXVec[i] < v2.AVXVec[i])
    {
      resultMask.AVXVec[i] = 1;
    }
    else
    {
      resultMask.AVXVec[i] = 0;
    }
  }
  return resultMask;
#elif defined AVX512
  resultMask.maskVec = _mm512_cmp_ps_mask(v1.AVXVec, v2.AVXVec, _MM_CMPINT_LT);
  return resultMask;
#elif (defined AVX || defined AVX2)
  resultMask.AVXVec = _mm256_cmp_ps(v1.AVXVec, v2.AVXVec, _CMP_LE_OS);
  return resultMask;
#endif
}

/*
  Initialises variables required for random number generation. The constant values 1664525 and 1013904223
  cause integer overflow, allowing for true random number generation. The value 2.3283064e-10f will be used
  in a multiplication to reduce very large random numbers to a value between 0 and 1.

  @param rC0 | An empty vector that will be filled with the first constant int, 1664525
  @param rC1 | An empty vector that will be filled with the second
    constant int, 1013904223
  @param factor | An empty vector that will be filled with the
    constant float, 2.3283064e-10f
  @param seeds | An array of seeds generated from the input seed by
    ranluxgen
  @param rSeed | An empty vector that will be filled with seeds from
     the seeds parameter
*/
void seedVecRandom(Vector& rC0, Vector& rC1, Vector& factor, int *
    seeds, Vector& rSeed)
{
#ifdef SISD
  for (int i = 0; i < 8; i++)
  {
    rSeed.AVXIntVec[i] = seeds[i];
  }
#elif defined AVX512
  __declspec(align(64)) unsigned int c0[16] = { 1664525L, 1664525L,
    1664525L, 1664525L, 1664525L, 1664525L, 1664525L, 1664525L
    ,1664525L, 1664525L, 1664525L, 1664525L, 1664525L, 1664525L,
    1664525L, 1664525L };
  __declspec(align(64)) unsigned int c1[16] = { 1013904223L,
    1013904223L, 1013904223L, 1013904223L, 1013904223L,  1013904223L,
     1013904223L, 1013904223L,1013904223L, 1013904223L, 1013904223L,
    1013904223L, 1013904223L, 1013904223L, 1013904223L, 1013904223L
    };
  __declspec(align(64)) float factors[16] = { 2.3283064e-10f,
    2.3283064e-10f, 2.3283064e-10f, 2.3283064e-10f, 2.3283064e-10f,
    2.3283064e-10f, 2.3283064e-10f, 2.3283064e-10f,2.3283064e-10f,
    2.3283064e-10f, 2.3283064e-10f, 2.3283064e-10f, 2.3283064e-10f,
    2.3283064e-10f, 2.3283064e-10f, 2.3283064e-10f };

  rSeed.AVXIntVec = _mm512_load_epi32(seeds);
  rC0.AVXIntVec = _mm512_load_epi32(&c0); //m512i
  rC1.AVXIntVec = _mm512_load_epi32(&c1); //m512i
  factor.AVXVec = _mm512_load_ps(factors);
#elif (defined AVX || defined AVX2)
  int c0 = 1664525L;
  int c1 = 1013904223L;
  float factors[8] __attribute__ ((aligned (32))) = { 2.3283064e-10f, 2.3283064e-10f, 2.3283064e-10f, 2.3283064e-10f, 2.3283064e-10f, 2.3283064e-10f, 2.3283064e-10f, 2.3283064e-10f};
  rSeed.AVXIntVec = _mm256_load_si256((__m256i*)seeds);
  rC0.AVXIntVec = _mm256_set1_epi32(c0); //m512i
  rC1.AVXIntVec = _mm256_set1_epi32(c1); //m512i
  factor.AVXVec = _mm256_load_ps(factors);
#endif
}

/*
  Generates a vector of random numbers between 0 and 1, created using the seeds created by the ranluxgen and
  3 constants (rC0, rC1 and factor). Used by ant.cpp when performing edge selection in csRoulette().

  @param rC0 | A vector containing the first constant int, 1664525
  @param rC1 | A vector containing the second constant int, 1013904223
  @param factor | A vector containing the constant float, 2.3283064e-10f
  @param rSeed | A vector containing seeds generated by ranluxgen

  @return | A vector contained with 8 or 16 random numbers between 0 and 1
*/

Vector vecRandom(Vector& rC0, Vector& rC1, Vector& factor, Vector& rSeed)
{
  Vector r;
#ifdef SISD
  for (int i = 0; i < 8; i++)
  {
    rSeed.AVXIntVec[i] = rSeed.AVXIntVec[i] * 1664525L + 1013904223L;
    r.AVXVec[i] = (float)rSeed.AVXIntVec[i] * 2.328306437087974e-10;
    r.AVXVec[i] += 0.5f;
  }
  return r;
#elif defined AVX512
```

```
  rSeed.AVXIntVec = _mm512_mullo_epi32(rC0.AVXIntVec, rSeed.AVXIntVec);
  rSeed.AVXIntVec = _mm512_add_epi32(rC1.AVXIntVec, rSeed.AVXIntVec);
  // convert to float in range 0 to 1 and return
  __m512 returnValue = _mm512_cvt_roundepu32_ps(rSeed.AVXIntVec, _MM_FROUND_TO_NEAREST_INT | _MM_FROUND_NO_EXC);
  r.AVXVec = _mm512_mul_ps(returnValue, factor.AVXVec);
  return r;
#elif defined AVX2
  __m256 addedVal;
  addedVal =_mm256_set1_ps(0.5f);
  rSeed.AVXIntVec = _mm256_mullo_epi32(rC0.AVXIntVec, rSeed.AVXIntVec);
  rSeed.AVXIntVec = _mm256_add_epi32(rC1.AVXIntVec, rSeed.AVXIntVec);
  // convert to float in range 0 to 1 and return
  __m256 returnValue = _mm256_cvtepi32_ps(rSeed.AVXIntVec);
  returnValue = _mm256_mul_ps(returnValue, factor.AVXVec);
  r.AVXVec = _mm256_add_ps(returnValue,addedVal);
  return r;

#endif
}

/*
  Compares values in the oldWeights vector (the vector of the previous largest values) with
  values in the newWeights vector (the vector of values from the current ACO iteration) and
  replaces values in oldWeights with larger values in the same vector lane from newWeights.
  The corresponding indexes for each weight are stored in oldIndices and newIndices
  respectively.

  @param oldWeights | A vector containing weight values for the previous largest weights
  @param oldIndices | A vector containing the corresponding indices for weights in oldWeights
  @param newWeights | A vector containing weight values for the largest weights in the current
  ACO iteration
  @param newIndices | A vector containing the corresponding indices
    for weights in newWeights

*/
void maxLocStep(Vector &oldWeights, Vector &oldIndices, Vector &
    newWeights, Vector &newIndices)
{
#ifdef SISD
  Vector maxMask = gtMask(newWeights, oldWeights);
  oldWeights = mask_mov(oldWeights, maxMask, newWeights);
  oldIndices = mask_mov(oldIndices, maxMask, newIndices);
#elif defined AVX512
  Vector maxMask;
  maxMask.maskVec = _mm512_cmp_ps_mask(newWeights.AVXVec, oldWeights
    .AVXVec, _MM_CMPINT_GT);
  oldWeights.AVXVec = _mm512_mask_mov_ps(oldWeights.AVXVec, maxMask.
    maskVec, newWeights.AVXVec);
  oldIndices.AVXVec = _mm512_mask_mov_ps(oldIndices.AVXVec, maxMask.
    maskVec, newIndices.AVXVec);
#elif (defined AVX || defined AVX2)
  Vector maxMask;
  maxMask = gtMask(newWeights, oldWeights);
  oldWeights = mask_mov(newWeights, maxMask, oldWeights);
  oldIndices = mask_mov(newIndices, maxMask, oldIndices);
#endif
}

/*
  Reduces the vector containing the largest weight values from every
     iteration (curWeights),
  determining the largest value in the vector and changing all
    vector lanes to that value, with
  identical operations being performed on the vector of the indeces
    associated with those weights

  @param curWeights | A vector containing weight values for largest
    weights from all iterations
  @param curIndices | A vector containing the corresponding indices
    for weights in curWeights
  @return | A vector with each lane containing the largest weight
    value in curWeights
```

```
*/
int reduceMax(Vector &curWeights, Vector &curIndices)
{
#ifdef SISD

  int highestIndex = -1;
  float highestValue = -1.0f;
  for (int i = 0; i < _VECSIZE; i++)
  {
    if (curWeights.AVXVec[i] > highestValue)
    {
      highestValue = curWeights.AVXVec[i];
      highestIndex = curIndices.AVXVec[i];
    }
  }

  return highestIndex;
#elif defined AVX512
  // return a vector with all elements equal to ivec[imax] where
  // valvec[imax] is largest element of valvec
  __m512 permVal;
  __m512 permInd;
  __mmask16 maxMask;
  // swap with neighbour
  permVal = _mm512_swizzle_ps(curWeights.AVXVec, _MM_SWIZ_REG_CDAB);
  permInd = _mm512_swizzle_ps(curIndices.AVXVec, _MM_SWIZ_REG_CDAB);
  maxMask = _mm512_cmp_ps_mask(curWeights.AVXVec, permVal,
    _MM_CMPINT_GT);
  curWeights.AVXVec = _mm512_mask_mov_ps(permVal, maxMask,
    curWeights.AVXVec);
  curIndices.AVXVec = _mm512_mask_mov_ps(permInd, maxMask,
    curIndices.AVXVec);
  // swap pairs
  permVal = _mm512_swizzle_ps(curWeights.AVXVec, _MM_SWIZ_REG_BADC);
  permInd = _mm512_swizzle_ps(curIndices.AVXVec, _MM_SWIZ_REG_BADC);
  maxMask = _mm512_cmp_ps_mask(curWeights.AVXVec, permVal,
    _MM_CMPINT_GT);
  curWeights.AVXVec = _mm512_mask_mov_ps(permVal, maxMask,
    curWeights.AVXVec);
  curIndices.AVXVec = _mm512_mask_mov_ps(permInd, maxMask,
    curIndices.AVXVec);
  // swap lanes
  permVal = _mm512_permute4f128_ps(curWeights.AVXVec, 0xB1); // 2,
    3, 0, 1
  permInd = _mm512_permute4f128_ps(curIndices.AVXVec, 0xB1);
  maxMask = _mm512_cmp_ps_mask(curWeights.AVXVec, permVal,
    _MM_CMPINT_GT);
  curWeights.AVXVec = _mm512_mask_mov_ps(permVal, maxMask,
    curWeights.AVXVec);
  curIndices.AVXVec = _mm512_mask_mov_ps(permInd, maxMask,
    curIndices.AVXVec);
  // swap pairs of lanes
  permVal = _mm512_permute4f128_ps(curWeights.AVXVec, 0x4E); // 1,
    0, 3, 2
  permInd = _mm512_permute4f128_ps(curIndices.AVXVec, 0x4E);
  maxMask = _mm512_cmp_ps_mask(curWeights.AVXVec, permVal,
    _MM_CMPINT_GT);
  curIndices.AVXVec = _mm512_mask_mov_ps(permInd, maxMask,
    curIndices.AVXVec);
  // all elements of ivec now contain index of maximum
  return curIndices.AVXVec[0];
#elif (defined AVX || defined AVX2)
  __m256 permVal;
  __m256 permInd;
  __m256 maxMask;
  float result[8] __attribute__ ((aligned (32)));
  permVal = _mm256_permute_ps(curWeights.AVXVec, _MM_SHUFFLE
    (2,3,0,1)); //01001110
  permInd = _mm256_permute_ps(curIndices.AVXVec, _MM_SHUFFLE(2, 3,
    0, 1)); //01001110
  maxMask = _mm256_cmp_ps(curWeights.AVXVec, permVal, _CMP_GT_OS);
  curWeights.AVXVec = _mm256_blendv_ps(permVal, curWeights.AVXVec,
    maxMask);
  curIndices.AVXVec = _mm256_blendv_ps(permInd, curIndices.AVXVec,
    maxMask);

  permVal = _mm256_permute_ps(curWeights.AVXVec, _MM_SHUFFLE
    (1,0,3,2)); //01001110
  permInd = _mm256_permute_ps(curIndices.AVXVec, _MM_SHUFFLE(1, 0,
    3, 2)); //01001110
  maxMask = _mm256_cmp_ps(curWeights.AVXVec, permVal, _CMP_GT_OS);
  curWeights.AVXVec = _mm256_blendv_ps(permVal, curWeights.AVXVec,
    maxMask);
  curIndices.AVXVec = _mm256_blendv_ps(permInd, curIndices.AVXVec,
    maxMask);

  permVal = _mm256_permute2f128_ps(curWeights.AVXVec, curWeights.
    AVXVec, 0b0001);
  permInd = _mm256_permute2f128_ps(curIndices.AVXVec, curIndices.
    AVXVec, 0b0001);
  maxMask = _mm256_cmp_ps(curWeights.AVXVec, permVal, _CMP_GT_OS);
  curWeights.AVXVec = _mm256_blendv_ps(permVal, curWeights.AVXVec,
    maxMask);
  curIndices.AVXVec = _mm256_blendv_ps(permInd, curIndices.AVXVec,
    maxMask);
  store(result, curIndices);
  return result[0];
#endif

}

/*
  Takes float values that are currently stored in a vector (most
    likely an array) and stores them
  in a memory (most likely an array).

  @param loc | The memory location where the data is to be stored
  @param v1 | A vector containing data that will be stored
*/
void store(float* loc, const Vector& v1)
{
#ifdef SISD

  for (int i = 0; i < _VECSIZE; i++)
  {
    loc[i] = v1.AVXVec[i];
  }

#elif defined AVX512
  _mm512_store_ps(loc, v1.AVXVec);
#elif (defined AVX || defined AVX2)
  _mm256_store_ps(loc, v1.AVXVec);
#endif
}

```

```
/*
  Takes integer values that are currently stored in a vector (most
    likely an array) and stores them
  in a memory (most likely an array).

  @param loc | The memory location where the data is to be stored
  @param v1 | A vector containing data that will be stored
*/
void store(int* loc, const Vector& v1)
{
#ifdef SISD
  for (int i = 0; i < _VECSIZE; i++)
  {
    loc[i] = v1.AVXVec[i];
  }
#elif defined AVX512
  _mm512_store_ps(loc, v1.AVXVec);
#elif (defined AVX || defined AVX2)
  _mm256_store_si256((__m256i *)loc, v1.AVXIntVec);
#endif
}

/*
  Compares two vectors lane-by-lane, with the larger value being
    saved to a new vector

  @param v1 | The first vector of values to be compared
  @param v2 | The second vector of values to be compared
*/
Vector max (const Vector& v1, const Vector& v2)
{
  #ifdef SISD
  Vector result;
  for(int i = 0; i < _VECSIZE; i++)
  {
    if(v1.AVXVec[i] > v2.AVXVec[i])
    {
      result.AVXVec[i] = v1.AVXVec[i];
    }
    else
    {
      result.AVXVec[i] = v2.AVXVec[i];
    }
  }

  return result;
  #elif (defined AVX || defined AVX2)
  Vector result;
  result.AVXVec = _mm256_max_ps(v1.AVXVec, v2.AVXVec);
  return result;
  #endif
}

/*
  Performs the abs function on a vector, converting every negative
    value
  to the equivalent positive value

  @param v1 | A vector containing data that will have the abs
    function applied to it
*/
Vector abs(const Vector& v1)
{
  #ifdef SISD
  Vector result;
  for(int i = 0; i < _VECSIZE; i++)
  {
    result.AVXVec[i] = std::abs(v1.AVXVec[i]);
  }
  return result;

  #elif (defined AVX || defined AVX2)
  Vector signMask, resultVector;
    signMask.set1(-0.0f);
  resultVector.AVXVec =  _mm256_andnot_ps(signMask.AVXVec, v1.AVXVec
    );
  return resultVector;
  #endif
}
```

# Appendix B

# Candidate Set Roulette

```
int Ant::csRoulette(float *weights, int *tabu, int nVerts, nearestNeighbour *nnList, int numNN)
{

#ifdef AVX2
  ALIGN(float indexSeed[_VECSIZE]) = { 0.0f, 1.0f, 2.0f, 3.0f, 4.0f, 5.0f, 6.0f, 7.0f};
  ALIGN(float indexStep[_VECSIZE]) = { 8.0f, 8.0f, 8.0f, 8.0f, 8.0f, 8.0f, 8.0f, 8.0f};
  ALIGN(float minusOnes[_VECSIZE]) = { -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f};
  ALIGN(float nextIndicesArray[_VECSIZE]) = {0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f};

#elif defined AVX512
  ALIGN(float indexSeed[_VECSIZE]) = { 0.0f, 1.0f, 2.0f, 3.0f, 4.0f, 5.0f, 6.0f, 7.0f, 8.0f, 9.0f, 10.0f, 11.0f, 12.0f, 13.0f, 14.0f, 15.0f};
  ALIGN(float indexStep[_VECSIZE]) = { 16.0f, 16.0f, 16.0f, 16.0f, 16.0f, 16.0f, 16.0f, 16.0f, 16.0f, 16.0f, 16.0f, 16.0f, 16.0f, 16.0f, 16.0f, 16.0f};
  ALIGN(float minusOnes[_VECSIZE]) = { -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f ,-1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f};
  ALIGN(float nextIndicesArray[_VECSIZE]) = {0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f};
#endif
  Vector minusOne;
  minusOne.load(minusOnes);
```

```
Vector runningIndex;
runningIndex.load(indexSeed);
Vector delta16;
delta16.load( indexStep );
Vector curIndices = minusOne;
Vector curWeights = minusOne;

for(int i = 0; i < numNN; i++)
{
  if(nnList[i].vectIndex != -1)
  {
    Vector randoms = vecRandom(rC0,rC1,factor,rSeed);
    Vector nextIndices = runningIndex;
    Vector tabuMask = int2mask(tabu[nnList[i].vectIndex]);

    Vector nnMask = int2mask(nnList[i].nnMask);
    Vector nextWeights;
    Vector nextDemand;
    Vector capacityVec;
    Vector capacityMask;

    nextWeights.load(weights + (i *_VECSIZE));
    nextWeights = nextWeights * randoms;
    nextWeights = mask_mov(minusOne, nnMask, nextWeights);
    nextWeights = mask_mov(nextWeights, tabuMask, minusOne);
    if(problemType == ProblemType::CVRP)
    {
      capacityVec.set1(capacity);
      nextDemand.load(m_as->demand + (nnList[i].vectIndex *
 _VECSIZE));
      capacityMask = gtMask(capacityVec, nextDemand);
      nextWeights = mask_mov(nextWeights, capacityMask, minusOne);
    }
    maxLocStep(curWeights, curIndices, nextWeights, nextIndices);
    runningIndex = runningIndex + delta16 ;
  }
  else
  {
    break;
  }
}

```

```
  // now reduce the elements of curWeights
  int reduced = reduceMax(curWeights, curIndices);
  if (reduced < 0) {
    reduced = -1;
  }

  return reduced;
}
```

# Appendix C

# Paper: Vectorised Candidate Set Selection for Ant Colony Optimisation

# Vectorized Candidate Set Selection for Parallel Ant Colony Optimization

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

Martyn Amos
Centre for Advanced Computational Science
Manchester Metropolitan University
Manchester, United Kingdom
M.Amos@mmu.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 well-established nature-inspired heuristic, and parallel versions of the algorithm now exist to take advantage of emerging high-performance computing processors. However, careful attention must be paid to parallel components of such implementations if the full benefit of these platforms is to be obtained. One such component of the ACO algorithm is *next node selection*, which presents unique challenges in a parallel setting. In this paper, we present a new node selection method for ACO, Vectorized Candidate Set Selection (VCSS), which achieves significant speedup over existing selection methods on a test set of Traveling Salesman Problem instances.

**ACM Reference Format:**
Joshua Peake, Martyn Amos, Paraskevas Yiapanis, and Huw Lloyd. 2018. Vectorized Candidate Set Selection for Parallel Ant Colony Optimization. In *GECCO '18 Companion: Genetic and Evolutionary Computation Conference Companion, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3205651.3208274

## 1 INTRODUCTION

Ant Colony Optimization (ACO) [10, 13] is a population-based optimization technique inspired by the foraging behavior of ants, and it has been successfully applied in a wide variety of domains [28]. The fundamental principle of the algorithm is that agents representing ants traverse some structure (such as a graph), constructing a solution to the given problem and laying virtual "pheromone trails" as they proceed. The amount of pheromone deposited is proportionate to the "quality" of the solution; as path choices made by individual ants are informed by pheromone concentrations, this leads the population to converge on high-quality solutions [25]. In this paper, we focus on the $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System variant of the algorithm [27], which allows only the "best performing" ant to deposit pheromone (and also restricts the range of pheromone concentrations, in order to prevent stagnation).

Because of the inherently distributed nature of ACO (whereby ants work independently of each other, guided only by a shared global pheromone network), the ACO algorithm presents significant opportunities in terms of its implementation on high-performance parallel hardware [2–4, 6, 8, 14, 21]. In terms of this paper, we are specifically interested in performance improvements that are made possible by the *vector processing* capabilities of chips such as the Intel® Xeon Phi [22, 30], which have instructions that operate on one-dimensional *arrays* of data (vectors), rather than on single data items. In the case of Xeon Phi, these *Single Instruction Multiple Data (SIMD)* instructions operate simultaneously on 16 floating point registers.

A significant bottleneck can arise in ACO-based algorithms when ants are required to select their next "move" [19]. Such algorithms generally (but not exclusively) work on graph-based representations of problems, where ants traverse edges, moving from vertex to vertex (as in the Traveling Salesman Problem (TSP) [11, 26]). Because the number of possible "next moves" can be extremely large, many algorithms use *candidate sets* (or candidate lists) to restrict movement to a select subset of vertices [15], and this has been successfully used in the Ant Colony System variant of ACO [12].

More recently, the use of *nearest neighbour* candidate lists has shown significant promise in solving large instances of the TSP using ACO [4, 7]. This refinement is based on the assumption that good solutions to the TSP avoid large "jumps", and that they can generally be found by making only relatively *local* transitions from vertex to vertex. Although candidate lists are now a standard component of parallel ACO-based algorithms [4, 7], previous implementations of this feature have failed to take advantage of the vector processing capabilities of processors such as the Xeon Phi. In this paper, we show how a modified representation of the nearest neighbour list can fully utilize vector processing, yielding significant performance improvements. Moreover, the speedup obtained increases as the problem size grows, suggesting that our method

ACM ISBN 978-1-4503-5764-7/18/07…$15.00
https://doi.org/10.1145/3205651.3208274

will be a required component of future ACO-based algorithms for large-scale instances of similar problems.

The rest of the paper is organized as follows: in Section 2 we discuss relevant earlier work, before presenting our algorithm in Section 3. We give and discuss our experimental results in Section 4, before concluding in Section 5 with a brief consideration of possible future work in this area.

## 2 BACKGROUND AND RELATED WORK

Early work on parallelizing the basic ACO algorithm used multiple independent runs of the sequential algorithm, with the best result from all runs being selected [24]. Later work [16] investigated its implementation on Graphics Processing Units (GPUs), which offered significant acceleration of the basic algorithm [1, 2, 4, 7, 9]. We now focus on key developments that contribute to the work described in the current paper.

The *independent-roulette* technique (*iRoulette*) was introduced by Cecilia *et al.* [4] as a parallel alternative to the traditional roulette-wheel selection method commonly used in sequential ACO algorithms. Roulette wheel selection is used whenever an ant must choose the next edge to traverse (and, thus, the next city to visit), with the probability of an edge being selected being proportional to its pheromone concentration. Although this is straightforward in the sequential algorithm, control flow and synchronization issues mean that it is more challenging in a parallel setting (Dawson and Stewart subsequently proposed an alternative *double-spin roulette* (DSRoulette) technique [8]). For an in-depth analysis of the properties of *iRoulette*, see [18].

With the availability of the Xeon Phi came new variants of *iRoulette*, due to the potential for vectorization offered by the its Vector Processing Unit (VPU). The algorithm described in [17] (which we refer to as *vRoulette-1*) is one example of this; the basic principles remain the same, but this variant makes use of intrinsic instructions available on the Xeon Phi to vectorize the *iRoulette* process, which yields improved performance over the original method (as well as over a vectorized version of the DSRoulette algorithm). We also note the existence of another vectorized version of *iRoulette*, *UVRoulette*, due to Tirado, *et al.* [31]. Along with their *vRoulette-1* method, Lloyd and Amos [17] utilized nearest neighbour information in their scheme for selecting cities. However, in this implementation, the candidate lists were used only to improve solution *quality*, and did not yield any speedup. In this paper, we describe an amended version of the algorithm described in [17], which replaces *vRoulette-1* with a properly vectorized nearest neighbour list (which we call *Vectorized Candidate Set Selection* (*VCSS*). As we will demonstrate, *VCSS* brings significant performance benefits in terms of execution time, especially with larger problem instances.

In the rest of this Section, we give a brief description of the base $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System (MMAS), and more details of the *iRoulette* method (both of which provide a foundation for our own contributions described in this paper).

### 2.1 MAX-MIN Ant System

ACO is an iterative algorithm, in which each step comprises two main phases: *Tour Construction* and *Pheromone Update*. During the tour construction phase, ants construct tours of the graph, making probabilistic choices based on heuristic weights derived from the lengths and pheromone values associated with edges in the complete graph. Each of the $m$ ants starts on a different randomly-selected vertex of the graph. At each stage in the construction of a tour, the probability of ant $k$ on vertex $i$ choosing to move to vertex $j$ is given by:

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

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 what we call the *feasible region* for vertex $i$. One constraint inherent to TSP problems is that cities can only be visited once. This is enforced in the ACO algorithm through the *tabu list*, which keeps track of every vertex visited during the current tour. During tour construction, vertices on the tabu list are ignored when making the choices. Thus, the *feasible region*, $N_i$ is the set of all vertices *not* in the tabu list which are adjacent to vertex $i$.

During the pheromone update phase, ants deposit pheromone on the edges traversed in their tours in an amount that is proportional to the objective quality of the tour (measured in terms of its length). In $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System, only one ant (the iteration best or best-so-far ant) deposits pheromone. This makes $\mathcal{MM}$AS well-suited to a parallel implementation, since we do not require concurrent write access to the pheromone matrix in order to allow multiple agents to deposit pheromone in parallel.

The amount of pheromone deposited is given by

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

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

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

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*, using the rule

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

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

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

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

where $n_{\text{neighbours}}$ is the number of nearest neighbours and $a = \exp(\log(0.05)/n)$. These limits are used to prevent solution *stagnation*, where one edge dominates others in its vicinity, due to it having a significantly higher pheromone concentration than the others (such edges will effectively become "locked in" to solutions, leading to a rapid loss of diversity). This restriction exists in conjunction with the standard pheromone *evaporation* operation, which is used at every iteration to allow concentrations to gradually decay (and thus allow the algorithm to "forget" sub-optimal solutions, over time).

### 2.2 Independent Roulette

In serial implementations of ACO, the probabilistic selection of edges (according to Equation 1) is performed using the *Roulette Wheel* algorithm. Various approaches have been used to parallelize this algorithm: here we concentrate on *independent-roulette (iRoulette)* [4], which forms a component of our *VCSS* algorithm described in Section 3.4.

In *iRoulette*, a choice between $N$ weights, $W_i, i \in [1, N]$ is made by multiplying the edge weights by uniform random deviates $R_i, i \in [1, N]$ with $R_i \in [0, 1]$. The selected edge is then

$$i_{\text{sel}} = \underset{i \in [1,N]}{\operatorname{argmax}} W_i R_i .$$

In this scheme, the probability of selecting an edge is no longer proportional to the weights, although higher-weighted edges are more likely to be selected than those with lower weights. A detailed analysis of *iRoulette* [18] shows that this produces greedier selection than standard Roulette Wheel selection, but the effect on solution quality is small. Furthermore, for $\mathcal{MM}$AS using this scheme demonstrably speeds up convergence to a solution.

A vectorized version of *iRoulette*, *vRoulette-1* [17], forms part of a Xeon Phi implementation of $\mathcal{MM}$AS and is the inspiration for the work presented here. In this original algorithm, the generation of random deviates and weight multiplication is vectorized across 16 lanes, with a final reduction over the vector producing the maximum.

## 3 VECTORIZED CANDIDATE SET SELECTION

The key contribution in this work is a vectorized algorithm (and associated data structure) to accelerate vertex selection using candidate sets (nearest neighbour lists). The selection procedure is modified (compared to previous versions) so that *only vertices in the nearest neighbour list for the current vertex are considered*. Only in cases where all of these are *tabu* will the remainder of the feasible region be considered. Typically, the nearest neighbour maximum list length is set to ~20; for large instances (with thousands of vertices) this can speed up the selection process by an order of magnitude or more.

We base our algorithm implementation on the Xeon Phi code described in [17]. The code has been ported to use the *AVX512* vector instruction set of the *Knight's Landing* architecture (but it can be ported to any multi-core SIMD architecture). The Intel® Xeon Phi range of processors are designed for use in high-performance computing, containing between 57-72 cores depending on the processor. These cores provide 4 threads (concurrent processes) each, which enables a high level of parallelization and vectorization. The latest generation of Xeon Phi, Knight's Landing, has several benefits over the previous generation, Knight's Corner, including a higher number of processor, and support for AVX-512 Single Instruction Multiple Data (SIMD) instructions, which allows for highly efficient vectorization of the ACO algorithm.

### 3.1 Instance Preprocessing

A potential performance problem is caused by the distribution of nearest neighbours in the problem instance. The relative proximity of vertices in space is not necessarily correlated with the vertex indices (that is, two vertices that are spatially adjacent might have indices that are widely separated, and vice versa; see Figure 1). If the indices of nearest neighbours tend to be close together, the nearest neighbour list can be kept short. Conversely, in the worst case, the nearest neighbour list will contain the full set of $N_p$ entries. In order to keep the size of the nearest neighbour list relatively low, we pre-process the problem instance before constructing the nearest neighbour list, by sorting the vertices into greedy tour order.

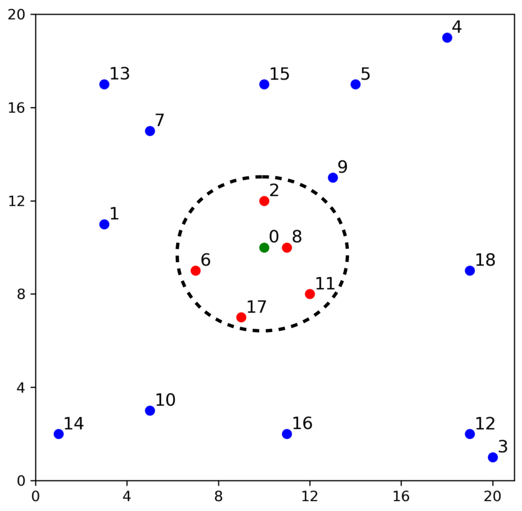


**Figure 1: Sample TSP graph, with the current city (labelled 0) in the center, and five nearest neighbour cities highlighted in the dashed containing region.**

### 3.2 Nearest neighbour List Construction

During the *setup* phase of the algorithm, the distance matrix (an $n \times n$ matrix encoding the edge lengths of the complete TSP graph) is used to calculate a vectorized nearest neighbour list data structure. This is performed as follows:

Let the number of nearest neighbours be $N_{nn}$, and the width of a SIMD vector (in floats) be $p$. We then let

$$N_p = \lceil N_{nn}/p \rceil,$$

the maximum number of SIMD vectors required to store one line of the nearest neighbour data structure. The data structure then comprises a list of up to $N_p$ *Nearestneighbour* objects (one per vertex) with each *Nearestneighbour* entry containing an integer index `ivec` and a $p$-wide bitmap `mask`. To add a vertex $j$ to the nearest neighbour list, we first ensure that there exists an entry with `ivec` $= \lfloor j/p \rfloor$, and set the bit in `mask` corresponding to $j$ mod $p$.

The data structure for a vertex is filled as follows: first, the other vertices are sorted by distance, and the first $N_{nn}$ of these are processed. For each of these vertices, `ivec` is calculated. If a *Nearestneighbour* entry already exists for this value of `ivec`, the appropriate bit is set in `mask`. If not, a new *Nearestneighbour* is added to the end of the list. The data structure is illustrated in Figure 2, for a vector width of 16.

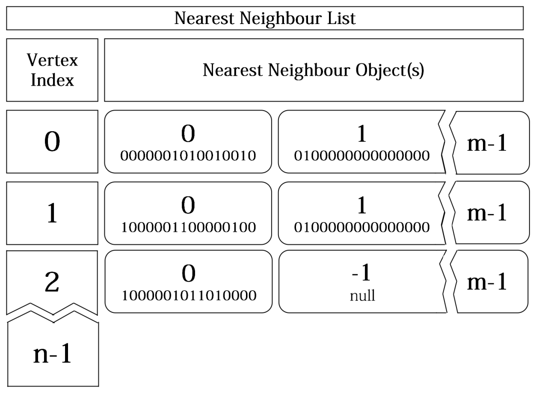


**Figure 2: Nearest neighbour data structure, with each vertex having an associated array of nearest neighbour objects containing a vector index `ivec` and a bit mask. A sentinel value of `ivec = −1` is used to indicate the end a line in the data structure. $n$ is the number of vertices and $n_{16}$ is the maximum number of entries for a vertex (for 16-wide vectors)**

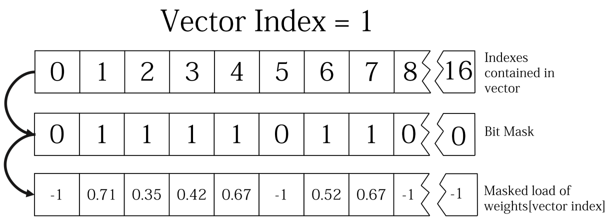


**Figure 3: Masked load process using the nearest neighbour list to retrieve the weights of nearest neighbour vertices.**

## 3.3 Tour Construction

We use OpenMP to assign each ant's tour construction process to a single thread. As no updates are made to any of the values used by the ants until the end of an iteration, and ants only write to their own local memory, no synchronization is required. We randomly select the starting vertex for each tour. We then repeatedly call the edge selection function to determine the ant's path around the graph. Each ant maintains its own *tabu* list, which keeps track of visited vertices.

In our experiments, described below, we evaluate two vectorized edge selection functions: *vRoulette-1* (which examines *every* vertex in the feasible region) and *Vectorized Candidate Set Selection* (*VCSS*), our new vectorized procedure, which uses nearest neighbour information. We now describe *VCSS* in more detail.

## 3.4 Vectorized Candidate Set Selection (VCSS)

Here, we propose a new selection procedure, *Vectorized Candidate Set Selection*, based on *iRoulette* [4], in which *iRoulette* selection is performed on a candidate set drawn from the nearest neighbour list data structure. If this *fails* to produce a selection (which will only happen when all the nearest neighbours have already been visited in the tour), the *vRoulette-1* procedure is used to select a vertex from the remaining feasible region.

*VCSS* takes the tabu list, an array of weights and the nearest neighbour list for the current vertex and then proceeds as follows (assuming a vector width $p$): first, we initialize $p$-wide vectors representing the running maximum weight and corresponding vertex indices. The algorithm then iterates over the nearest neighbour list. For each *Nearestneighbour* object, we load the edge weights for the corresponding vertices as a single $p$-wide vector. We use the bitmask in the *Nearestneighbour* object to mask this weight vector such that only the vertices in the nearest neighbour list remain (illustrated in Figure 3).

We also construct a vector of consecutive integers, corresponding to the vertex indices in the vector. We multiply the weight vector by a $p$-wide vector of random deviates (produced using a simple linear congruential generator). We then compare the modified weight vector, on an element-wise basis (in a single instruction), with the running maximum. This produces a bit mask which is used to update both the running maximum and the corresponding index vector. A reduction is then performed over the vector lanes to produce the maximum weight and corresponding index (in our implementation, this reduction is performed in $\log_2 p$ steps using bit-swizzling instructions). If no edge has been selected, the *vRoulette-1* algorithm is used by default on the complete set of weights.

The algorithm is formally described in Algorithm 1. Here, *Random()* is a function which returns a $p$-wide vector of uniform deviates, and *ApplyMask(mask, a, b)* is a function which returns a vector filled with values from $a$ in positions where the corresponding value of *mask* is set, and values from $b$ where the *mask* value is not set.

**Algorithm 1:** Pseudo-code for Vectorized Candidate Set Selection.

```
Input  : Edge Weight array $\mathbf{W}_{0...N-1}$, Tabu Mask array
         $\mathbf{T}_{0...n-1}$, Maximum number of nearest neighbours
         $N_p$, nearest neighbour list $L_{0...N_p-1}$
Output : Selected Edge
// Variables in bold are p-vectors, superscripts indicate vector
  lanes
$\mathbf{W}_{max}$ = (0...0);
 $\mathbf{I}_{max}$ = (0...0);
for $i = 0$ to $N_p - 1$ do
    if $L[i]$.ivec $\neq -1$ then
        $\mathbf{R}$ = Random();
        $\mathbf{V} = L[i]$.mask;
        $\mathbf{I} = (pL[i]$.ivec...$pL[i]$.ivec $+ p - 1)$;
        $\mathbf{w} = ApplyMask(V_i, \mathbf{W}_i \times \mathbf{R}, (-1... - 1))$;
        $\mathbf{w} = ApplyMask(T_i, (-1...1), \mathbf{w})$;
        $max\_mask = \mathbf{w} > \mathbf{W}_{max}$;
        $\mathbf{I}_{max} = ApplyMask(max\_mask, \mathbf{w}, \mathbf{W}_{max})$;
    end
end
//Reduction
j = argmax($\mathbf{W}_{max}$);
return $\mathbf{I}^{j}_{max}$;
```

## 3.5 Pheromone Update

The pheromone update process is split into four phases: Deposit, evaporation, clamping and edge probability calculation. The deposit phase is carried out by a single thread, with little potential for vectorization. The remaining phases are carried out in a pair of nested loops, with the outer loop being parallelized with OpenMP. The inner loop iterates over the pheromone matrix 16 values at a time, using vector instructions to perform the evaporation, clamping and probability calculations.

## 4 EXPERIMENTAL RESULTS

In what follows, we measure the performance three variants of ACO: the first is CPU reference code, the second uses only *vRoulette-1*, and the third uses our new *VCSS* method. Experiments were carried out on a machine with an Intel® Xeon Phi 7210 processor with 64 cores and 4 threads per core (for a total of 256 threads), running at a base speed of 1.3GHz. The code was compiled with the Intel® C++ compiler (`icc`) at -O3 optimization level. The code was run under Linux, with timings obtained using the *gettimeofday()* function. The CPU reference code used is ACOTSP version 1.03 [23], compiled with `gcc` (with optimization -O3) and run on a Linux machine containing an 8-core Intel® Xeon E5-2650 v2 at a base speed of 2.6GHz.

### 4.1 ACO Parameters and Problem Instances

We use 32 nearest neighbours for our experiments, which is both in line with Dorigo and Stützle's recommended list size [29], and a convenient power of two for the purposes of data alignment. The values of the $\mathcal{MM}$AS parameters used are as follows: $\alpha = 1, \beta = 2, \rho = 0.02$. In all cases, the number of ants is set to 256 (so that all available threads are used when assigning ants to threads).

The problem instances used in our experiments are taken from the TSPLIB library[20], and include all instances solved in [17] and [32]. We also include larger instances, in order to investigate the performance of the algorithm as the problem size increases. The instances used are: `lin318`, `pcb442`, `rat783`, `pr1002`, `fl1577`, `pr2392`, `fl3795`, `rl5934`, `pla7397`, and `rl11849`. For each instance, we performed 50 runs of 1024 iterations.

### 4.2 Execution Time

We measure execution time on a per-iteration basis. Results are shown in Figure 4, which (log) plots the mean time per iteration over all instances for *VCSS* and *vRoulette-1* on the Xeon Phi, and ACOTSP on CPU, and Table 1, which gives the numerical values, along with the speedup.

While *vRoulette-1* and *VCSS* have similar execution times on the smaller instances, as the instance size grows, the execution time for *VCSS* grows more slowly than that of *vRoulette-1*. For the largest instance, `rl11849`, *VCSS* is faster than *vRoulette-1* by an order of magnitude, and is faster than the reference code by two orders of magnitude. On the other hand, *vRoulette-1*, while performing two orders of magnitude faster than the reference code on smaller instances, shows a declining speed-up compared to the CPU as the instance size grows.

**Table 1: Execution time per iteration in milliseconds, and speedup relative to CPU.**

| | CPU | *vRoulette-1* | | *VCSS* | |
|---|---|---|---|---|---|
| Instance | $t$/ms | $t$/ms | Speedup | $t$/ms | Speedup |
| `lin318` | 18.1 | 0.73 | 24.8 | 0.52 | 34.8 |
| `pcb442` | 29.2 | 1.37 | 21.3 | 0.74 | 39.5 |
| `rat783` | 68.3 | 5.65 | 12.1 | 1.37 | 49.9 |
| `pr1002` | 94.1 | 9.01 | 10.4 | 1.85 | 50.9 |
| `fl1577` | 176.7 | 20.9 | 8.45 | 3.5 | 50.1 |
| `pr2392` | 371.0 | 46.4 | 8.00 | 7.02 | 52.9 |
| `fl3795` | 785.7 | 143.3 | 5.48 | 13.5 | 58.2 |
| `rl5934` | 2088.9 | 426.8 | 4.90 | 27.9 | 74.9 |
| `pla7397` | 3388.3 | 724.3 | 4.68 | 43.04 | 78.7 |
| `rl11849` | 10578.8 | 1975.0 | 5.36 | 97.47 | 108.5 |

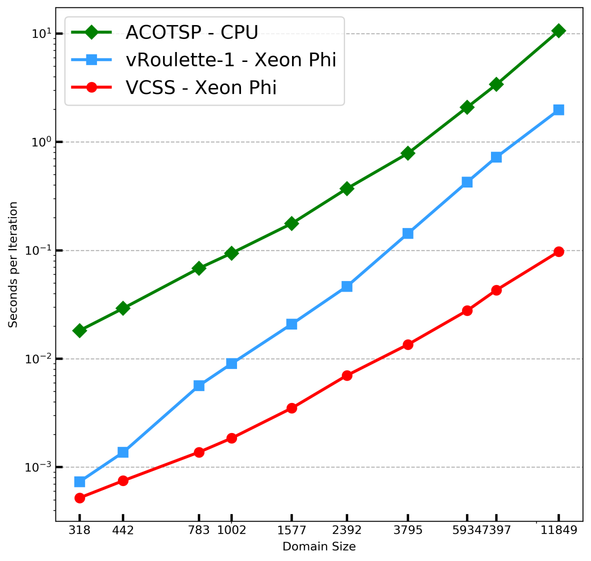


**Figure 4: Execution times for *ACOTSP*, *vRoulette-1* and *VCSS*.**

### 4.3 Solution Quality

In Figure 5 we show box plots of solution quality of each algorithm (where solution quality is measured as the ratio of the length of the best tour found to the known optimum for the problem instance). We would expect differences between the solution quality obtained with the CPU code and the two Xeon Phi variants due to the modified selection probabilities in the *iRoulette* scheme compared with those in the roulette wheel selection used by ACOTSP. It is already known that *iRoulette* can affect the solution quality on individual instances, although its average behavior does not significantly affect the quality of solution [18]. There is some variation between the solution qualities obtained using *vRoulette-1* and *VCSS*. It should be noted that this experiment used a relatively small sample of instances, with 50 runs per instance. In order to measure effects on solution quality, a larger sample of instances (with one run per instance) would be better. Additionally, for the larger instances, the number of iterations (1024) is relatively small and the solutions may not be converged. However, the focus of this paper is the efficiency measured in terms of time per iteration: in order to investigate any effects on solution quality, more extensive experiments would be required. Given that VCSS is formally equivalent to the nearest-neighbour list algorithms already widely studied in serial ACO, we would not expect to see large systematic effects on the solution quality arising from the use of *VCSS*, although this will be a topic for further investigation.

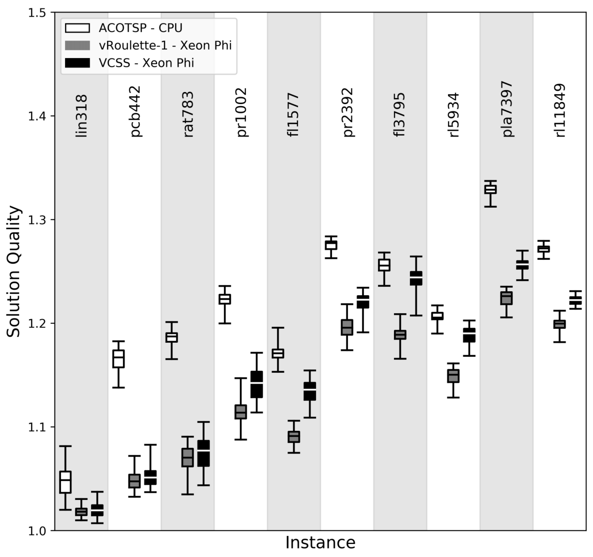


**Figure 5: Solution quality for *ACOTSP*, *vRoulette-1* and *VCSS*.**

## 4.4 Discussion

We have demonstrated that significant speedups are obtained using our *VCSS* scheme. The speedup over *vRoulette-1* grows as the instance size increases. Without the nearest neighbour list, the tour construction process has time complexity $O(n^2)$ (since at each of $n$ vertices, $n - 1$ vertices are included in the selection process). The nearest neighbour list reduces this complexity to $O(n)$ (since the workload per vertex is constant, determined only by the size of the nearest neighbour list). The speedup, relative to the CPU code, also increases with the instance size. This is more difficult to explain, since the CPU code also uses a fixed size nearest neighbour list, and the execution time should therefore scale in the same way. However, considering the number of 16-wide vectors which are processed in making a selection with *vRoulette-1*, we see that - for instances up to around 500 vertices - this is less than or equal to the size of the nearest neighbour list. The increase in speed up is, therefore, due to the nearest neighbour list conferring minimal benefit with smaller instances. In this case, we would expect the speedup to eventually level off.

## 5 CONCLUSION

In this paper, we presented *VCSS*, a novel nearest neighbour vectorization technique and selection method for ant colony optimization. This method is an order of magnitude faster than the previous best-performing algorithm on the Xeon Phi platform, *vRoulette-1*, and two orders of magnitude faster than the reference CPU code.

While we have shown that the performance of *VCSS* improves with increasing instance size, there is a limit to how far this can be pursued. One of the inherent limitations of the general ACO algorithm is its $O(n^2)$ memory complexity, due to the need to store a square pheromone matrix. Around 500MB of memory is required for our largest instance, and to move to the next order of magnitude (a 100,000 city instance) we would require around 37GB. This limitation must be overcome before the speed gains obtained using the latest parallel and vector ACO techniques can be fully exploited on larger instances. The PartialACO [5] method is a promising development in overcoming this barrier.

Our solution may also benefit further from the inclusion of *local search*, which is often used to accelerate convergence [13]. While local search *may* be parallelized, there are currently no vectorized algorithms which can utilize the full power of many-core SIMD hardware. The possibility of using local search with parallel ACO requires further investigation.

There are a number of areas for further investigation in terms of the details of our *VCSS* algorithm. Firstly, the fall-back to *vRoulette-1* when all nearest neighbours are tabu introduces a potential load-balance issue (although, in practice, this happens very rarely). The work carried out by each thread will differ, depending on how many times *vRoulette-1* is used, and all threads must wait for the slowest to complete. This could be alleviated either by using a faster fall-back algorithm, or by organizing the workload differently, with ants sending work to a pool of threads, rather than decomposing the work strictly by ant.

Secondly, the distribution of the lengths of the nearest neighbour lists (in terms of the number of vectors required to store the nearest neighbours) is another potential source of load imbalance. The greedy tour scheme used here to reorder the vertices may be improved upon. It is possible, for example, that a distribution of list lengths with a larger mean, but smaller variance, could give shorter execution times, due to improved load balancing. A full analysis of the relationship between this distribution and the load balance is another area for future work.

## REFERENCES

[1] Alberto Cano, Juan Luis Olmo, and Sebastián Ventura. Parallel multi-objective ant programming for classification using GPUs. *Journal of Parallel and Distributed Computing*, 73(6):713 – 728, 2013.
[2] José M Cecilia, José M García, Manuel Ujaldón, Andy Nisbet, and Martyn Amos. Parallelization strategies for Ant Colony Optimisation on GPUs. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 339–346, May 2011.
[3] José M Cecilia, Andy Nisbet, Martyn Amos, José M García, and Manuel Ujaldón. Enhancing GPU parallelism in nature-inspired algorithms. *The Journal of Supercomputing*, 63(3):773–789, 2013.
[4] 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.
[5] Darren M Chitty. Applying ACO to large scale TSP instances. In *UK Workshop on Computational Intelligence*, pages 104–118. Springer, 2017.
[6] Laurence Dawson. *Generic Techniques in General Purpose GPU Programming with Applications to Ant Colony and Image Processing Algorithms.* PhD thesis, Durham

University, UK., 2015.

[7] Laurence Dawson and Iain A 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.

[8] Laurence Dawson and Iain A Stewart. Improving Ant Colony Optimization performance on the GPU using CUDA. In *2013 IEEE Congress on Evolutionary Computation*, pages 1901–1908, June 2013.

[9] Audrey Delévacq, Pierre Delisle, Marc Gravel, and Michaël Krajecki. Parallel ant colony optimization on graphics processing units. *Journal of Parallel and Distributed Computing*, 73(1):52–61, 2013.

[10] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39, 2006.

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

[12] Marco Dorigo and Luca Maria Gambardella. Ant colony system: a cooperative learning approach to the Traveling Salesman Problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, Apr 1997.

[13] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. Bradford Company, Scituate, MA, USA, 2004.

[14] Jie Fu, Lin Lei, and Guohua Zhou. A parallel Ant Colony Optimization algorithm with GPU-acceleration based on All-In-Roulette selection. In *Advanced Computational Intelligence (IWACI), 2010 Third International Workshop on*, pages 260–264, 2010.

[15] Fred Glover. Tabu search – Part I. *ORSA Journal on computing*, 1(3):190–206, 1989.

[16] Wang Jiening, Dong Jiankang, and Zhang Chunfeng. Implementation of Ant Colony Algorithm based on GPU. In *CGIV '09: Proceedings of the 2009 Sixth International Conference on Computer Graphics, Imaging and Visualization*, pages 50–53, Washington, DC, USA, 2009. IEEE Computer Society.

[17] 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, Dec 2016.

[18] Huw Lloyd and Martyn Amos. Analysis of independent roulette selection in parallel Ant Colony Optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, pages 19–26, New York, NY, USA, 2017. ACM.

[19] Marcus Randall and James Montgomery. Candidate set strategies for Ant Colony Optimisation. In *International Workshop on Ant Algorithms*, pages 243–249. Springer, 2002.

[20] Gerhard Reinelt. TSPLIB - a Traveling Salesman Problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

[21] Rafał Skinderowicz. The GPU-based parallel Ant Colony System. *Journal of Parallel and Distributed Computing*, 98(Supplement C):48 – 60, 2016.

[22] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights Landing: Second-generation Intel® Xeon Phi product. *IEEE Micro*, 36(2):34–46, 2016.

[23] Thomas Stützle. ACOTSP. Available at http://iridia.ulb.ac.be/~mdorigo/ACO/downloads/ACOTSP-1.03.tgz (2005/06/12).

[24] Thomas Stützle. Parallelization strategies for Ant Colony Optimization. In *PPSN V: Proceedings of the 5th International Conference on Parallel Problem Solving from Nature*, pages 722–731, London, UK, 1998. Springer-Verlag.

[25] Thomas Stutzle and Marco Dorigo. A short convergence proof for a class of ant colony optimization algorithms. *IEEE Transactions on Evolutionary Computation*, 6(4):358–365, 2002.

[26] Thomas Stützle and Holger Hoos. MAX-MIN ant system and local search for the Traveling Salesman Problem. In *Evolutionary Computation, 1997., IEEE International Conference on*, pages 309–314, Apr 1997.

[27] Thomas Stützle and Holger H Hoos. Max–min ant system. *Future Generation Computer Systems*, 16(8):889–914, 2000.

[28] Thomas Stützle, Manuel López-Ibáñez, and Marco Dorigo. A concise overview of applications of Ant Colony Optimization. In James J. Cochran, Louis A. Cox, Pinar Keskinocak, Jeffrey P. Kharoufeh, and J. Cole Smith, editors, *Wiley Encyclopedia of Operations Research and Management Science*. John Wiley & Sons, Inc., 2010.

[29] Thomas Stützle and Marco Dorigo. Ant Colony Optimization. 01 2004.

[30] Xinmin Tian, Hideki Saito, Serguei V Preis, Eric N Garcia, Sergey S Kozhukhov, Matt Masten, Aleksei G Cherkasov, and Nikolay Panchenko. Practical SIMD vectorization techniques for Intel® Xeon Phi coprocessors. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1149–1158. IEEE, 2013.

[31] 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, Nov 2017.

[32] 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, Nov 2015.

# Appendix D

# Paper: Scaling Techniques for Parallel Ant Colony Optimization on Large Problem Instances

# 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 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, $\tau_{min}$ and $\tau_{max}$, 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,

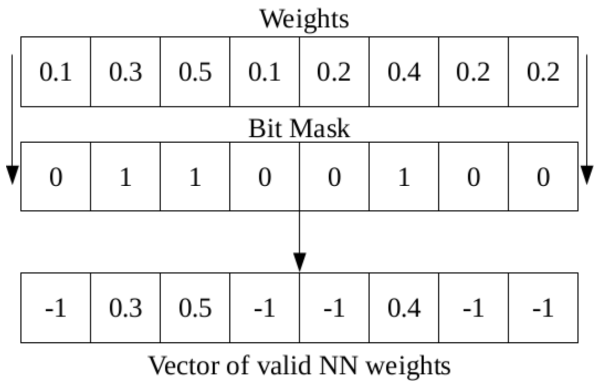


**Figure 1: Applying the NN mask to filter out non-NN weights**

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 $\tau_{min}$. 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

**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 |

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.

## 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

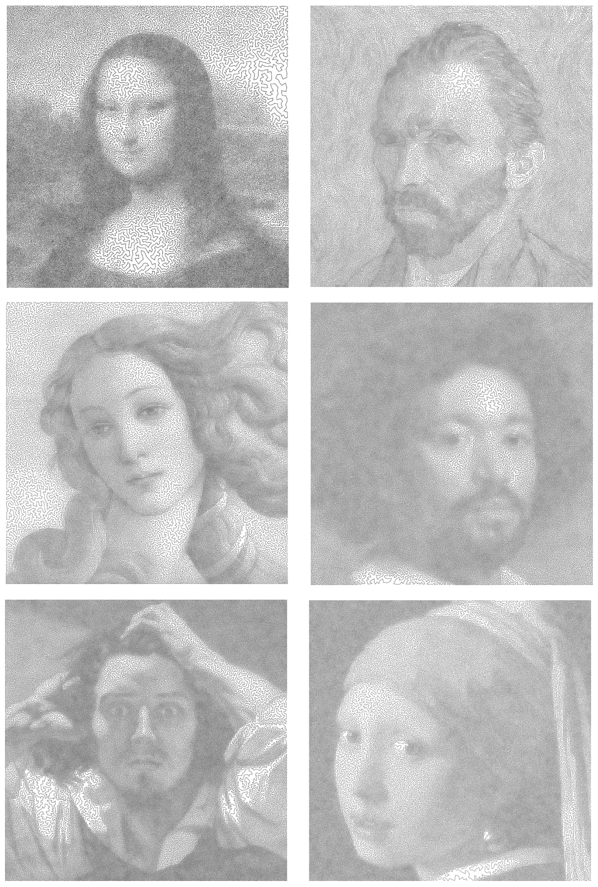


**Figure 2: Best known tours for the Art TSP instances: `mona-lisa100k` (top left), `vangogh120k` (top right), `venus140k` (middle left), `pareja160k` (middle right), `courbet180k` (bottom left) and `earring200k` (bottom right).**

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 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.

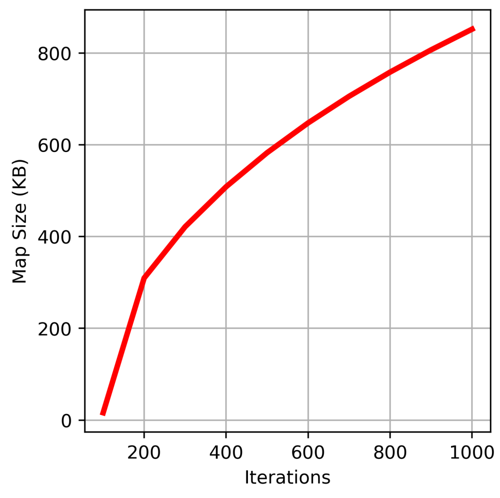


**Figure 3: Pheromone map size over time**

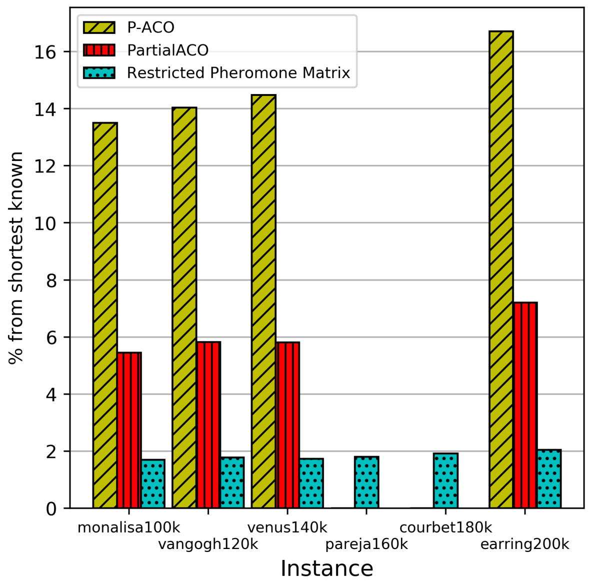


**Figure 4: Plot of the solution quality difference between P-ACO, PartialACO (results taken from [9] and Restricted Pheromone Matrix (our experiments) against shortest known tour. No P-ACO or PartialACO results are available for `pareja160k` and `courbet180k`.**

**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.

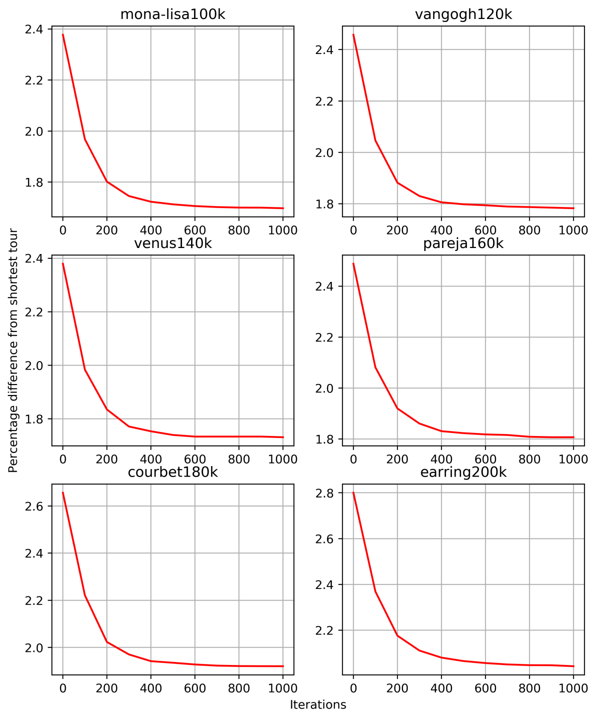


**Figure 5: Solution quality versus iteration for the Art TSP instances using the heuristic fallback.**

## 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.

# Appendix E

# Paper: PACO-VMP: Parallel Ant Colony Optimization for Virtual Machine Placement

# PACO-VMP: Parallel Ant Colony Optimization for Virtual Machine Placement

Joshua Peake[a,*], Martyn Amos[b], Nicholas Costen[a], Giovanni Masala[a], Huw Lloyd[a]

[a]*Department of Computing and Mathematics, Manchester Metropolitan University, Manchester, United Kingdom.*
[b]*Department of Computer and Information Sciences, Northumbria University, Newcastle upon Tyne, United Kingdom.*

**Abstract**

The Virtual Machine Placement (VMP) problem is a challenging optimization task that involves the assignment of virtual machines to physical machines in a cloud computing environment. The effective placement of virtual machines has a significant impact on the use of resources in a cluster, with a subsequent impact on operational cost and the environment. In this paper, we present an improved algorithm for VMP, based on Parallel Ant Colony Optimization (PACO), which makes effective use of parallelization techniques and modern processor technologies. We achieve solution qualities that are comparable with or superior to those obtained by other nature-inspired methods, with our parallel implementation obtaining a speed-up of up to 2002x over recent serial algorithms in the literature. This allows us to rapidly find high-quality solutions that are close to the theoretical minimum number of Virtual Machines.

*Keywords:* Virtual Machine Placement, Ant Colony Optimization, Swarm Intelligence, Parallel MAX-MIN Ant System, Parallel Ant Colony Optimization

## 1. Introduction

Cloud computing [22] is an increasingly prevalent computing paradigm, in which on-demand computing services (such as compute or storage) are provided, either privately or commercially, as a service to remote users and organizations. As well as providing the foundation for modern electronic commerce, the paradigm is a key enabler for a number of recent developments, such as the Internet of Things [6], Edge Computing [29], and Big Data Analytics [2], all of which in turn enable important societal developments such as Smart Cities [38] and Intelligent Transportation Systems [21]. However, data centres now represent a significant proportion of global energy usage; this figure currently stands at around 2%, and it is set to rise [19]. There is, therefore, an urgent need to optimize the software infrastructure underpinning modern data centres.

Resource requirements are expressed in terms of Virtual Machine (VM) instances, each of which carries its own overhead. A key benefit of cloud computing for users is its *scalability*, which is derived from the ability to dynamically increase and reduce resource usage depending on demand. While this *elasticity* is beneficial for users, it provides challenges for cloud computing providers. With constantly changing demand, the assignment of VMs to servers (or Physical Machines, PMs) can quickly become inefficient, leading to unnecessary usage of servers. This can cause providers to use more of their hardware resources than are necessary, which has both an economic and environmental impact. The solution to this is *virtual machine consolidation*, which allocates currently in-use VMs to as few PMs as possible. This increases server utilisation and energy efficiency, and lower power consumption equates to lower energy costs for the host. This also incentivizes efficient re-allocation of servers to ensure that they operate in an efficient configuration for a longer amount of time, which leads to a further reduction in energy usage. A number of algorithms have been proposed to address this problem; here, we focus on methods based on *Ant Colony Optimization*. Importantly, we focus on *parallel* Ant Colony Optimization, which takes advantage of modern multi-core hardware to significantly reduce the time required to find satisfactory solutions. We make use of the AVX2 instruction set, available on the vast majority of modern CPUs, to further reduce execution time in an already parallelised approach.

The rest of the paper is organised as follows: In Section 2 we provide background to the Virtual Machine Placement problem and existing methods for its solution; in Section 3 we describe Ant Colony Optimization and our own improved algorithm; in Section 4 we present the results of evaluating our algorithm against competing techniques, and in Section 5 we discuss our findings and suggest possible further work.

## 2. Background & Related Work

In this Section we first describe the Virtual Machine Placement Problem and discuss a range of existing methods for its solution.

### 2.1. Virtual Machine Placement Problem

*Hardware virtualization* in cloud computing allows for many separate machine instances to be created that are distinct from

[*]Corresponding author: j.peake@mmu.ac.uk

the host machine on which they are running. These instances, known as Virtual Machines (VMs), essentially act as completely separate computers, distinct from other VMs running on the same host. Each VM may have its own specific resource requirements (in terms of memory, and so on), and thus occupies a specific "footprint". These VMs are managed by a *hypervisor* running on the host server (also known as a Virtual Machine Monitor, VMM) which creates, optimizes and monitors the performance of VMs.

Many companies such as Amazon (AWS) and Microsoft (Azure) provide access to Virtual Machines hosted on their own servers. Due to the sheer size of these operations, a significant amount of hardware is required to offer these services to customers. In order to minimize, as far as possible, the amount of costly physical infrastructure required, a process known as *Virtual Machine Migration* is often used to move Virtual Machines from one Physical Machine to another in a seamless fashion, without disruption for the user [12]. This ability to migrate VMs therefore offers the possibility of *optimization* of their placement on servers. Given a set of VMs, each with a specific resource footprint, and a number of PMs with individual capacities, what is the most efficient allocation of VMs to PMs, such that the number of PMs is minimised? For scenarios where a small number of servers are available, determining the most efficient allocation for a small number of VMs can be trivial. However, services such as AWS have millions of users and hundreds of thousands of servers, which significantly complicates this process.

Virtual Machine Placement (VMP) is an NP-hard problem [31], in which the aim is to allocate Virtual Machines (VMs) to Physical Machines (PMs) as efficiently as possible. While VM features differ between variants of the problem, the two most typical attributes are *memory* (RAM) and *processing* (CPU). RAM requirements are generally measured in Gigabytes (GB), while CPU requirements are generally measured in either processor cores or MIPS (million instructions per second). Every VM has its own specific requirement for each, which means it occupies its own resource "footprint". The aim of the problem is to "legally" allocate every VM to a PM in such a way that the number of PMs required is minimised (that is, this version of the VMP is a variant of a bin packing problem). Here, we focus on the *static* variant of the VMP problem, where we need to allocate a fixed set of VMs to PMs.

### 2.1.1. Problem definition

We now formally define the VMP in the form considered in this paper. An instance of the VMP is defined by a set $V$ of virtual machines $V = \{V_i, i \in [1, N_{vm}]\}$ with *CPU requirements* and *RAM requirements* $C_i^{\text{req}}, R_i^{\text{req}} \forall i \in [1, N_{vm}]$, and a set $P$ of physical machines $P = \{P_j, j \in [1, N_{pm}]\}$ with *CPU capacities* and *RAM capacities* $C_j^{\text{cap}}, R_j^{\text{cap}} \forall j \in [1, N_{pm}]$. A *feasible solution* to an instance of the VMP is a mapping of the indices of virtual machines $i$ to physical machines $j$ such that $\forall j$, $\sum_i C_i^{\text{req}} \leq C_j^{\text{cap}}$ and $\sum_i R_i^{\text{req}} \leq R_j^{\text{cap}}$ where the sums are taken over the indices of all virtual machines $i$ which are mapped to the physical machine $j$. The optimization problem seeks to find a feasible solution which maximizes the number of *e*mpty physical machines, which is equal to the cardinality of the set of indices $j$ which are not mapped from any virtual machines $i$.

An illustrative example of an instance of the static VMP problem is shown in Figure 1, along with its solution. We have an initially unbounded number of PMs, each with a fixed CPU and RAM resource, and a number of VMs (VM1-5) to be allocated to PMs such that the total resource requirement on each PM does not exceed its capacity, and the number of PMs is minimised (in this case, to three).

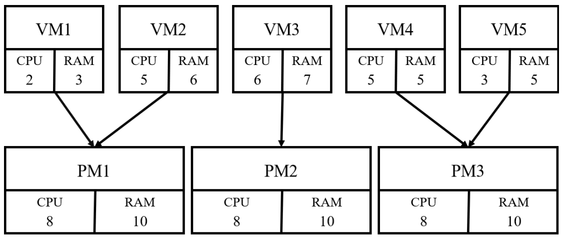


Figure 1: Instance of the Virtual Machine Placement problem, with arrows showing the solution (i.e., the allocation of VMs to PMs). Virtual Machine requests are efficiently allocated to Physical Machines

## 2.2. Existing Methods

As with many combinatorial optimisation problems, a wide range of techniques exist to find solutions to VMP instances. As well as heuristic-based approaches such as Next Fit, First Fit, First Fit Decreasing (FFD) and Best Fit (BF) [13], more advanced optimization techniques have been successfully applied to the VMP problem; these include Genetic Algorithms (GA) [20, 27, 33], Particle Swarm Optimization [36], Q-Learning [26] and Ant Colony Optimization (ACO) [3, 18, 20, 23].

A recently-published method for VMP, which provides one of the comparison baselines for the work presented here, is the IGA-POP genetic algorithm (GA) [1]. This frames the VMP as a Variable-Sized Bin Packing Problem (VSBPP), a variant of the Bin Packing Problem in which the container elements have differing capacities. In IGA-POP, a solution encodes an ordering of VM assignments to PMs.

The fitness function for this algorithm prioritises low power usage, and it performs competitively in terms of solution quality against the BF and First-Fit (FF) greedy algorithms, the Sine-Cosine Optimization Algorithm (SCA) [28] and a generic GA. For this reason, we select IGA-POP as being representative of the "evolutionary" algorithm class of solutions for VMP.

Our own method is based on Ant Colony Optimization (ACO), [15] which is an optimization metaheuristic modelled on the foraging behaviour of ants [14]. When ants leave the nest to look for food, they initially explore the local area; once food is located by an ant, it returns to the nest. On the return journey, the ant leaves a *pheromone trail*, which increases the probability that other members of the colony will take that path to the food source. As each ant follows a trail, it also lays its own trail, strengthening the pheromone over time and causing more ants to follow it, in a process of positive feedback reinforcement.

This phenomenon is abstractly replicated by the ACO algorithm, with problem instances generally represented by graph structures, and with multiple "ants" searching for a solution by traversing its edges according to pheromone concentrations. As an ant traverses a problem graph, it uses a weighted random process to select its next move, in which solution components with a better combination of pheromone and heuristic are more likely to be selected. Pheromone also evaporates over time, meaning that unproductive paths are eventually erased (this preventing premature convergence).

Feller et al. [18] presented an early implementation of ACO for VMP. This treats VMP as a multi-dimensional bin-packing problem (MDBP), with the Physical Machines representing the bins, and the VMs representing the item to be packed. As reducing the number of PMs is the most effective way of reducing energy usage, the objective of the algorithm is to minimize the number of bins used. The algorithm fills PMs one-at-a-time, with each bin being closed when no remaining VMs can fit inside. Pheromone is deposited on Item-Bin pairs, with VMs being linked to the specific PM to which they are allocated. The heuristic is based on the total resource utilisation of the PM if the current VM were to be assigned to it, and the pheromone deposition is based on the average utilisation of all utilised PMs. The Feller ACO technique outperforms First Fit Decreasing (FFD) in terms of energy usage, saving 4.1% on average. However, execution time for the algorithm is significant, ranging from 37.46 seconds for 100 VMs to 2.01 hours for 600 VMs.

A more recent ACO-based VMP algorithm is OEMACS [23]. This adds two Local Search procedures: an exchange procedure similar to a local search procedure used for Bin Packing Problems [4], which swaps VMs between PMs in an attempt to find a more efficient configuration, and an insertion procedure, which attempts to remove a VM from one PM and insert it into another. OEMACS outperforms FellerACO in terms of both solution quality and execution time. We therefore compare our ACO algorithm against OEMACS, as it stands as a representative modern ACO algorithm for the VMP. To summarize, we select for comparison with our algorithm OEMACS and IGA-POP as representative state of the art ACO and GA solvers for the VMP, along with a standard heuristic, *first-fit*.

## 3. Our ACO algorithm for VMP

### 3.1. Overview of ACO

As we base our algorithm on ACO, we now provide a brief overview of its operation. The first ACO algorithm (Ant System) was described by [16]. Many variants of, and applications for the algorithm have since been developed, however the key features of ACO which all the variants share are that the algorithm uses a number of *agents* (ants) which independently construct solutions guided by a global *pheromone matrix* data structure and in some cases, a problem-specific heuristic. The representation of the solution typically takes the form of a subset of edges from a graph, and the *solution construction* phase of the algorithm involves each ant iteratively traversing the graph, building a feasible solution by selecting from the available edges at each step. Edges are selected using a random choice weighted by the pheromone and heuristic values associated with the available edges. The pheromone update phase of the algorithm aims to associate higher pheromone values with edges which are included in good solutions, and to evaporate pheromone from older, less successful, solution components. We base our algorithm on the $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS) variant [32].

Due to the inherently distributed nature of ACO (many "ants" effectively act independently, informed by their environment), parallelizing the algorithm is a well-researched subject area. While the early implementations of parallel ACO made use of distributed systems [7, 10], the development of Nvidia's Computed Unified Device Architecture (CUDA) framework led to a significant number of GPU-based implementations [8, 9, 30]. The CUDA framework gives access to the powerful parallelization architecture offered by GPUs for graphics processing, and utilise it for other purposes. This is known as General-Purpose computing on Graphics Processing Units (GPGPU). Before CUDA, distributed systems were the only realistic method of parallelizing an algorithm, but CUDA allows for parallelization to be performed on a single machine. The rising thread count on modern processors has also increased the viability of parallel ACO on CPUs [11, 17, 24, 34, 35], along with the availability of Single Instruction Multiple Data (SIMD) vector operations such as the AVX512 instruction set on Intel Xeon Phi and Xeon processors. SIMD is a class of parallel computing in which the same operation is performed on multiple data points simultaneously, allowing multiple operations to be performed in parallel. In the case of AVX512 and the previous AVX2 instruction set, 16 and 8 operations respectively can be performed simultaneously. These instructions can be applied to code that is already parallelized, essentially reducing the number of operations required by 16x, allowing for an even deeper level of parallel processing.

While running ants in parallel across a graph seems like a straightforward task, certain aspects of the ACO algorithm make parallelization difficultm in particular the fundamental *roulette-wheel* selection technique used by ACO, where different paths are allocated a "slice" of the roulette wheel that is proportional to their favourability, is not amenable to parallelization. To overcome this issue, Cecilia *et al.* developed a new data parallel approach to ACO edge selection known as I-Roulette [8]. While the exact proportionality between probability and edge weights is not fully maintained, I-Roulette still accurately replicates the behaviour of the Roulette Wheel selection in a parallel compatible manner. I-Roulette was later adapted into vRoulette [24], which made use of the AVX512 vector instructions to further increase the efficiency of the algorithm.

In what follows we describe an alternative ACO algorithm for VMP which dramatically reduces run-time by harnessing *parallel computing* techniques described above, and modern processor features.

### 3.2. Data structures and algorithm design

Our algorithm for VMP is a novel ACO variant that uses parallelization techniques and SIMD vector operations to efficiently solve VMP problems. The algorithm is based the $\mathcal{MM}$AS variant of ACO, as this is most amenable to parallelization (due to the absence of communication between ants during an iteration). In this Section we describe our Parallel Ant Colony Optimization for Virtual Machine Placement (PACO-VMP) algorithm. Complete reference code is available online [1]. The notation used in this paper is defined in Table 1, and the algorithm is shown in detail in Figure 3.

The key data structures used by the algorithm are:

1. The *pheromone matrix*, a square ($N_{vm} \times N_{vm}$) matrix of floating point numbers which holds the pheromone values associated with including a particular pair of VMs in the same PM in a solution.
2. An array of *ant* data structures, which each contain a representation of a *solution* (an array of arrays of integers, which lists the indices of the VMs assigned to each PM).

At the highest level of description, the algorithm proceeds as shown in Algorithm 1; each phase of the algorithm is discussed in detail in subsections which follow.

**Algorithm 1** High-level description of proposed algorithm

```
Initialization Phase
for each Iteration do
    for each Ant do
        Solution Construction
    end for
    Apply Local Search to iteration-best Ant
    Update Global Best Solution
    Pheromone Update
end for
```

### 3.2.1. *Initialization Phase*

In this phase, the parameters and structures required by the algorithm are created and initalized. An important step is to ensure that any arrays that will be used for vectorized computations are padded correctly, which prevents errors when they are loaded into vectors. As our implementation uses the Intel AVX2 instructions, which operate on 8 32-bit values at a time, the size of the arrays must be a multiple of 8. The arrays also need to be aligned in memory correctly in order to be correctly loaded into AVX2 vectors. The pheromone matrix is a matrix of size $N_{\mathrm{v,}} \times N_{\mathrm{vm}}$ , with $N_{\mathrm{v,}}$ being the number of Virtual Machines in the problem instance. The values of the pheromone matrix are initially set to $\tau_0 = 1/N_{\mathrm{pm}}$, where $N_{\mathrm{pm}}$ is the number of Physical Machines. In this phase we also set the value of the $\mathcal{MM}$AS constant $a$ (see equation 11), which is later used to determine the maximum and minimum pheromone values. The number of PMs is initially set to be equal to the number of VMs.

### 3.2.2. *Solution Construction*

The first step of the solution construction phase is to randomly shuffle the VMs. This happens at the beginning of each iteration in order to prevent VMs being allocated the same PM purely due to their position in the array. OpenMP is used to allocate each ant's construction process to a separate thread. As each ant only reads from global pheromone memory during the construction phase and does not write to memory, synchronisation is not required. During the construction phase, the ants loop through every VM and allocate it to a PM, unless the current VM is unable to fit in any remaining PM. Any VMs left un-allocated at the end of the loop are then allocated to the PM with the most available capacity, creating an *infeasible* solution. A Local Search procedure, which will be fully described in a later section, is applied to the solution in an attempt to make it *feasible*.

Table 1: List of symbols and notations used in this paper

| **Symbol** | **Definition** |
|---|---|
| $S_{gb}$ | The global best solution |
| $S_{ib}$ | The best solution from the current iteration |
| $P_{gb}$ | Power usage of the global best solution |
| $P_{ib}$ | Power usage of the iteration best solution |
| $\tau_0$ | The initial pheromone value |
| $N_s$ | The number of PMs ants are able to use |
| $N_{gb}$ | The number of PMs used in $S_{gb}$ |
| $N_{ib}$ | The number of PMs used in $S_{ib}$ |
| $N_{vm}$ | The number of VMs in the current instance |
| $N_{pm}$ | The number of PMs in the current instance |
| $k$ | Current iteration number |
| $k_{\max}$ | Maximum number of iterations permitted |
| $i_{\mathrm{cur}}$ | The current VM |
| $j_{\mathrm{cur}}$ | The current PM |
| $\alpha$ | Pheromone influence |
| $\beta$ | Heuristic influence |
| $\rho$ | Pheromone decay rate |
| $\eta_{ij}$ | Heuristic value between VM $i$ and PM $j$ |
| $\tau_{ij}$ | Pheromone value between VMs $i$ and $j$ |
| $P$ | Power usage of current solution |
| $P_j^{\max}$ | Maximum power usage of PM $j$ |
| $P_j^{\mathrm{idle}}$ | Idle power usage of PM $j$ |
| $f_C$ | CPU usage ratio for current PM |
| $f_R$ | RAM usage ratio for current PM |
| $C_j^{\mathrm{used}}$ | Current CPU usage on PM $j$ |
| $R_j^{\mathrm{used}}$ | Current RAM usage on PM $j$ |
| $C_i^{\mathrm{req}}$ | CPU requirement of VM $i$ |
| $R_i^{\mathrm{req}}$ | RAM requirement of VM $i$ |
| $C_j^{\mathrm{cap}}$ | Total CPU capacity on PM $j$ |
| $R_j^{\mathrm{cap}}$ | Total RAM capacity on PM $j$ |
| $\tau_{\max}$ | Maximum pheromone value |
| $\tau_{\min}$ | Minimum pheromone value |
| $a$ | $\mathcal{MM}$AS constant value |
| $N_{\min}$ | Theoretical lower limit of current VMP |
| $N_B$ | Number of type B servers |
| $C_A^{\mathrm{cap}}$ | CPU capacity of type A servers |
| $C_B^{\mathrm{cap}}$ | CPU capacity of type B servers |
| $R_A^{\mathrm{cap}}$ | RAM capacity of type A servers |
| $R_B^{\mathrm{cap}}$ | RAM capacity of type B servers |

The selection procedure used to allocate VMs is based on the vRoulette-1 technique developed by Lloyd & Amos [24]. This is demonstrated in Figure 2, which shows how the Heuris-

[1] `https://github.com/jnpeake/PACO-VMP`

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Heuristic | 0.02 | 0.02 | 0.07 | 0.03 | 0.04 | 0.12 | 0.35 | 0.1 |
|  |  |  |  |  | x |  |  |  |
| Pheromone | 0.06 | 0.05 | 0.09 | 0.08 | 0.06 | 0.03 | 0.02 | 0.07 |
|  |  |  |  |  | x |  |  |  |
| Random | 0.1 | 0.6 | 0.4 | 0.6 | 0.3 | 0.5 | 0.9 | 0.2 |
|  |  |  |  |  | = |  |  |  |
| Total | 0.00012 | 0.0006 | 0.00252 | 0.00144 | 0.0072 | 0.0018 | 0.063 | 0.00014 |

Figure 2: A demonstration of how the vRoulette-1 technique combines the heuristic and pheromone values of a PM with a random number between 0 and 1. AVX2 instructions allows operations to be carried out on each Vector lane (numbered 0-7) simultaneously.

tic and Pheromone values (which we describe in detail later) of the PM in each vector lane are combined with a random number between 0 and 1. This is then multiplied by a *Tabu* value, which is set to 0 or 1 (the value is only set to 0 in the instance that the "PM" in that lane is actually just a placeholder used to pad the PM list to a multiple of 8), and then masked by vectors (denoted *MaxCPUMask* and *MaxRAMMask*) that filter out any PMs that do not have enough available capacity for the current VM. This is done on a vector-by-vector basis, with 8 PMs being processed for selection in parallel. The 8 current PM values are compared lane-by-lane with a vector of the highest PM values in the current selection process. Once every PM has been processed, a parallel reduction (with the *max* operator) is carried out on this vector and the PM corresponding to the highest value is then assigned the current VM. If the highest value is lower than 0, this indicates that no PMs had enough capacity available for the current VM, and the VM is added to the unassigned list to be allocated once the solution construction procedure has been completed.

While the original vRoulette-1 implementation made use of the AVX512 instruction set, which allows for 16-wide vectors and features additional instructions compared to AVX2, it is not currently as widely available as the AVX2 instruction set, which is available on most Intel CPUs released since 2013, and most AMD CPUs released since 2015. For the implementation evaluated here, we used AVX2.

### *3.2.3. Heuristic & Pheromone Definition*

As with any ACO implementation, the definition of the pheromone and heuristic values is crucial for the consistent construction of good-quality solutions.

The heuristic is a problem-specific value which indicates the favourability of assigning a VM to a PM. The definition of the heuristic value can differ significantly even within ACO implementations that aim to solve the same problem. A key difference between calculating the heuristic value for VMP is the need for a dynamically calculated heuristic which differs depending on the current state of the PM that is being assigned to, and this requires the heuristic to be calculated at every step of the solution for every VM, which increases the solution time compared to the more static heuristic values of problems such as the Traveling Salesman Problem.

Our heuristic definition is designed to ensure that the fewest possible number of VMs are used, by prioritising both resource utilisation balance and total resource utilisation. The prioritisation of total utilisation makes it more likely that an ant will allocate the current VM to a PM that already contains other VMs, while the resource balance will attempt to keep the available RAM and CPU on a PM as even as possible, which will prevent PMs exhausting one resource capacity while still having a large available capacity for the other resource. The heuristic value, $\eta_{ij}$, associated with placement of virtual machine $i$ on physical machine $j$ is given by

$$\eta_{ij} = \frac{1 - |f_C - f_R|}{1 + f_C + f_R} \tag{1}$$

where

$$f_C = \frac{C_j^{\text{used}} + C_i^{\text{req}}}{C_j^{\text{cap}}} \tag{2}$$

and

$$f_R = \frac{R_j^{\text{used}} + R_i^{\text{req}}}{R_j^{\text{cap}}}. \tag{3}$$

Here, $C_j^{\text{used}}$ and $R_j^{\text{used}}$ are, respectively, the current CPU and RAM usage of physical machine $j$, $C_i^{\text{req}}$ and $R_i^{\text{req}}$ are the CPU and RAM requirements of virtual machine $i$, and $C_j^{\text{cap}}$ and $R_j^{\text{cap}}$ are the CPU and RAM capacities of physical machine $j$.

Implementations of ACO for VMP generally use one of two pheromone trail definitions: the first defines the trail as being between VMs and the PMs to which they are allocated, and the second defines trails as being between VMs that are allocated the same PMs, meaning that VMs are more likely to be allocated to a PM with VMs that they have previously shared with in good solutions. In this paper, we choose to define the pheromone trail to associate VMs with other VMs. The pheromone distributed is based on solution quality, which in our case is the energy consumption of our solution.

As the selection process of our algorithm attempts to allocate VMs to PMs, we are unable to load pheromone information directly from the matrix as pheromone is distributed between VMs rather than between VM and PM. Instead, we calculate the mean value of pheromone which links the current VM and the VMs that are currently allocated to the PM that we are evaluating (the amount of pheromone between VM and PM is initially set to $\tau_0$, and remains at that level until a VM is added to the PM).

Therefore, the pheromone associated with a physical machine $j$ when placing a virtual machine $i$ is given by

$$T_{ij} = \begin{cases} \tau_0, & \text{if} N_{vm}^j = 0 \\ \frac{1}{N_{vm}^j} \sum_k \tau_{ik}, & \text{otherwise} \end{cases} \tag{4}$$

where $N_{vm}^j$ is the number of VMs already assigned to PM $j$, and the sum is taken over all VMs $k$ which are assigned to PM $j$.

Finally, the weight associated with a particular choice of PM, $j$, for a given VM, $i$, during the solution construction phase is determined by combining the pheromone and heuristic values according to

$$W_{ij} = T_{ij}^{\alpha}\eta_{ij}^{\beta} \tag{5}$$

where $\alpha$ and $\beta$ are parameters controlling the relative influence of pheromone and heuristic information. The choice of PM is made with probability $p$ which is proportional to $W_{ij}$.

### 3.2.4. Local Search

Local Search is a procedure which takes a candidate solution generated by an optimisation algorithm and makes small changes in order to find a local minimum with respect to some neighbourhood. The term "Local Search" refers to a vast array of usually problem-specific techniques that carry out these small changes. Local Search techniques are widely used in conjunction with ACO to good effect, and they are essential for creating solutions that are optimal or near-optimal. Local Search takes place after the solution construction phase, once the iteration-best solution has been determined. A drawback of this Local Search technique is the fact that it is non-trivial, leading to significant cost in processing time, and it runs serially, rather than in parallel. Due to this, we perform local search only on the iteration-best solution.

Our Local Search is based on a technique developed by Alvim et al. [4] for the Bin Packing Problem, and also utilised by Liu et al. [23] for the VMP. In this algorithm, after each solution is found, one bin is destroyed, or a PM in the case of the VMP problem. If a subsequent solution is then able to successfully fit all items in the remaining bins, it is considered *feasible*. However, if no feasible solution can be found, the local search technique is applied. There are two phases of our technique, the swap phase and the insertion phase. Any PM that has been allocated more VMs than it has capacity for is marked as *overloaded*. In the swap phase an overloaded PM is compared with every non-overloaded PM, and the algorithm attempts to swap each VM in the overloaded PM with each VM in the non-overloaded PM. This continues until either a successful swap takes place, or every non-overloaded PM has been compared to the overloaded PM. Regardless of the outcome, the process is carried out again for the next PM, and this continues until every overloaded PM has been compared. If the swap phase is unable to successfully find a feasible solution, the insertion phase is then performed. In this phase, each overloaded PM attempts to allocate each of its VMs to a non-overloaded PM. While this is far less likely to produce positive results than the swap phase, it is still able to occasionally make progress where the swap phase cannot.

### 3.2.5. Pheromone Update

The final phase of our ACO algorithm is the pheromone distribution. As our algorithm is based on the $\mathcal{MM}$AS ACO variant, pheromone is only deposited only by the global-best and. As mentioned previously, pheromone is distributed between VMs allocated to the same PM. The pheromone matrix is updated using

$$\tau_{ij} \leftarrow \tau_{ij}\left(1 + \frac{365}{P}\right) \tag{6}$$

for all pairs of VMs $i, j$ which are allocated to the same PM in the global best solution, and where $P$ is the power usage of the solution. The power usage is defined as

$$P = \sum_{j=1}^{N_{\text{PM}}}\left((P_j^{\max} - P_j^{\text{idle}})\frac{C_j^{\text{used}}}{C_j^{\text{cap}}} + P_j^{\text{idle}}\right) \tag{7}$$

where $N_{\text{PM}}$ is the number of PMs in the current instance and $P_j^{\max}$ and $P_j^{\text{idle}}$ are the maximum and idle power usage of physical machine $j$ respectively. We choose a definition of pheromone based on power usage as it will reflect the positive impact of a lower number of PMs while still measuring differences between solutions with the same number of PMs used.

The global amount of pheromone then decays by a static amount, controlled by the parameter $\rho$,

$$\tau_{ij} \leftarrow \tau_{ij}(1-\rho) \ \ \forall i, j \in [1, N_{vm}]. \tag{8}$$

The choice of value of $\rho$ will be discussed in Section 4.

$\mathcal{MM}$AS utilises a clamping procedure to prevent stagnation, by restricting the level of pheromone to be between maximum and minimum values. The maximum and minimum values are defined as

$$\tau_{\max} = \frac{1}{\rho N_{\text{best}}^{\text{global}}} \tag{9}$$

$$\tau_{\min} = \tau_{\max}\frac{2(1-a)}{(N_{\text{VM}}+1)a} \tag{10}$$

where $N_{\text{VM}}$ is the number of VMs in the current instance, $P_{\min}$ is the global lowest PM usage, and

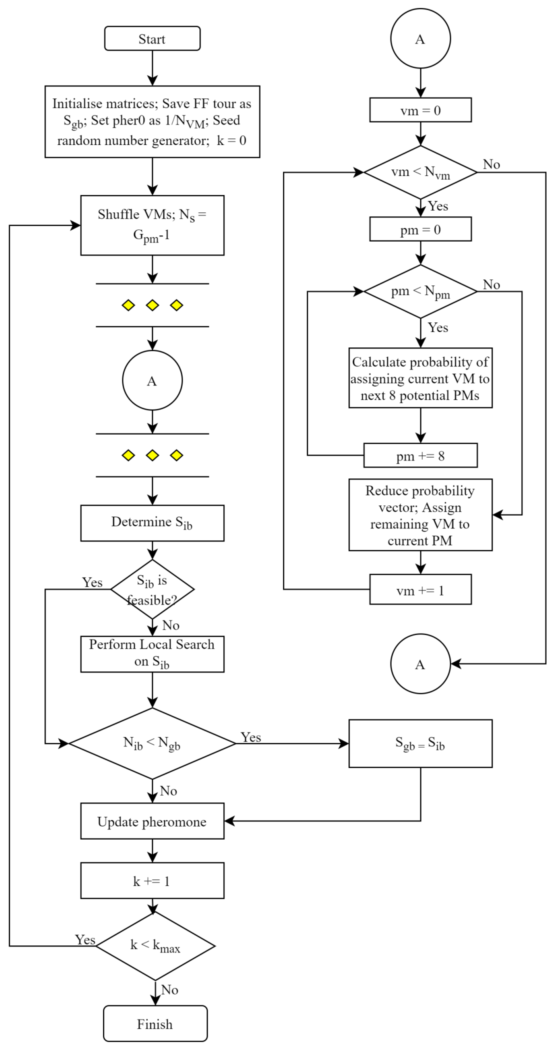
$$a = \exp(\ln(0.05)/N_{\text{VM}}). \tag{11}$$

This clamping is applied to the whole matrix after evaporation.

## 4. Experimental Evaluation

We investigate the performance of our algorithm by comparing with an implementation of the OEMACS algorithm, which is an ACO-based method that generally out-performs conventional heuristics and evolutionary algorithms for this problem [23], and a state-of-the-art genetic algorithm, IGA-POP [1]. Code for OEMACS is publicly available[2]. All algorithms were implemented in C++, and all tests were carried out on a machine with an Intel® Xeon E5-2640 v4 processor with 20 cores running at a base frequency of 2.4 GHz and a maximum frequency of 3.4 GHz. Code was compiled using the GNU C++ compiler (g++), with *O2* optimization enabled. The initial comparative tests will compare a serial implementation of PACO-VMP with the serial algorithms OEMACS and IGA-POP, with the aim of comparing the quality of solutions obtained. Two variants of IGA-POP will be used: the first, referred to as GA1, uses the fitness function also used by PACO-VMP and OEMACS; the second, referred to as GA2, uses a slightly modified version

[2]https://github.com/Budding0828/OEMACS



Start
Initialise matrices; Save FF tour as $S_{gb}$; Set pher0 as $1/N_{VM}$; Seed random number generator; k = 0
Shuffle VMs; $N_s$ = $G_{pm}$-1
A
Determine $S_{ib}$
$S_{ib}$ is feasible?
Yes
No
Perform Local Search on $S_{ib}$
$N_{ib} < N_{gb}$
Yes
No
$S_{gb} = S_{ib}$
Update pheromone
k += 1
$k < k_{max}$
Yes
No
Finish
A
vm = 0
vm < $N_{vm}$
Yes
No
pm = 0
pm < $N_{pm}$
Yes
No
Calculate probability of assigning current VM to next 8 potential PMs
pm += 8
Reduce probability vector; Assign remaining VM to current PM
vm += 1
A

Figure 3: A flow chart detailing the PACO-VMP algorithm. Section A, between the yellow diamonds, is executed in parallel for each ant.

of the fitness function used in the initial IGA-POP experiments [1]. Finally, to evaluate the impact of our OpenMP parallelization, we also ran a parallelized PACO-VMP using OpenMP, assigning one ant to each of our 20 cores. While the execution time differs, solution quality is identical to the serial version.

## 4.1. *Problem instances*

All problem instances used in our experiments were randomly generated. For each instance, the initial number of Physical Machines is set to be equal to the number of Virtual Machines. Our dataset consists of three sets of 600 VMP instances, each further split into 6 subsets of 100 instances with 100, 200, 300, 400, 500 and 1000 VMs. The three sets (A, B and C) are described in the following subsections, and differ by the inclusion of bottlenecks in one or other resource, or the homogeneity of the physical machines in the instance setup. One run is performed using each instance. We use one run per instance with a larger number of instances, rather than multiple runs on a smaller number of instances; a proof in [5] shows that given a budget of $N$ runs, selecting a $K$ instances and performing $n$ runs on each with $N = Kn$ is a suboptimal choice and that the best statistical estimate of algorithm performance is obtained from a single run on each of $N$ independently selected instances, contrary to popular belief.

### 4.1.1. *Instance Set A: Homogeneous Environment*

Set A is designed to evaluate the performance of the algorithms in a straightforward scenario where the PMs are identical and the demands of the VMs are fairly evenly divided between RAM and CPU. This set consists of 600 VMP instances equally divided between 100, 200, 300, 400, 500 and 1000 VM instances. This dataset is similar to the Set A data used by Liu *et al.* [23] in evaluating OEMACS, which was initially created to benchmark the Reordering Grouping Genetic Algorithm (RGGA) [37]; however, this data is no longer publicly available. In comparison to this data, we used a larger number of smaller instances.

The VM requirements for this instance set are randomly generated in ranges of [1,128] for CPU and [1,100] for RAM. Each PM has a capacity of 500 for both CPU and RAM, leading to slightly higher average CPU utilisation than RAM but still close to 1:1. As these instances are randomly generated, there is no known optimum, but a lower limit to the number of PMs used in the solution, $N_{\text{min}}$, is calculated

$$N_{\text{min}} = \max\left\{\frac{\sum_{j=1}^{N_{\text{VM}}} C_j^{\text{req}}}{C_i^{\text{cap}}}, \frac{\sum_{j=1}^{N_{\text{VM}}} R_j^{\text{req}}}{R_i^{\text{cap}}}\right\} \tag{12}$$

where $i$ is the index of any physical machine; as the servers in Instance Set A are homogeneous, it does not matter which physical machine is used to evaluate this quantity.

### 4.1.2. *Instance Set B: Homogenous Environment with Bottleneck*

Set B introduces a bottleneck resource to the problem instances, evaluating the performance of the algorithms in a slightly more complicated scenario which will lead to more overloaded servers. As with the previous instance set, Set B consists of 600 VMP instances equally divided between 100, 200, 300, 400, 500 and 1000 VM instances. VM requirements are randomly generated, in the range of [1-4] for CPU (measured in cores) and [1-8] for RAM (measured in GB). PM capacity is 16 cores for CPU and 32GB for RAM. As the probability of a 4 core VM requirement is higher than the probability of an 8GB RAM requirement, CPU is the bottleneck resource. As with Set A, these instances have no known optimum, and a lower limit is calculated using the same formula.

### 4.1.3. *Instance Set C: Heterogeneous Environment with Bottleneck*

Set C further complicates the problem instances by introducing non-identical servers, simulating a scenario in which a cloud host has multiple server types. We define two types of server, *A* and *B*. Server type *A* has a CPU capacity of 16 cores and a RAM capacity of 32GB. Server type *B* has a CPU capacity of 32 cores and a RAM capacity of 64GB. However, type

$B$ servers only make up 10% of the total PMs in each problem instance, meaning that VMs will have to use both types of servers. This will evaluate the ability of the algorithms to prioritise the high capacity servers while still allocating the VMs efficiently. The VM requirements are in the range of [1,8] for CPU and [1,32] for RAM, meaning that the bottleneck resource in this case is RAM. Set C utilises the same instance sizes as the previous sets.

Due to the heterogenous servers in this instance set, an alternative formula is required for calculating the lower limit to the number of PMs

$$N_{\min} = N_B + \max\left\{\frac{\sum_{j=1}^{N_{\mathrm{VM}}} C_j^{\mathrm{req}} - N_B C_B^{\mathrm{cap}}}{C_A^{\mathrm{cap}}}, \frac{\sum_{j=1}^{N_{\mathrm{VM}}} R_j^{\mathrm{req}} - N_B R_B^{\mathrm{cap}}}{R_A^{\mathrm{cap}}}\right\} \tag{13}$$

where $N_B$ is the number of type $B$ servers, $C_A^{\mathrm{cap}}$ and $C_B^{\mathrm{cap}}$ are the CPU capacities of type A and B servers respectively, and $R_A^{\mathrm{cap}}$ and $R_B^{\mathrm{cap}}$ are the RAM capacities of type A and B servers respectively.

### *4.2. Experimental configuration*

For each experiment we use 20 ants for PACO-VMP, as this allows for one ant to be allocated to each core available on our hardware in the OpenMP-enabled variant. For our algorithm, we use the ACO parameter values specified in the next section, and for OEMACS we use the default values as specified in [23]. Both PACO-VMP and OEMACS are run for 50 iterations. For GA1 and GA2, we use the parameters specified in [1], 200 iterations and a population size of the number of PMs multiplied by 4. We compare these results against the First Fit (FF) algorithm in order to provide a baseline greedy algorithm implementation. FF was selected over the more widely used FFD algorithm due to better results on our data sets. It should be noted that the solution construction time for FF is near-instantaneous for all instance sizes, and thus has been omitted from all execution time plots. For the ACO parameters, we selected values of $\rho = 0.8, \alpha = 1, \beta = 6$ on the basis of some tuning experiments on a small sample of 1000 VM instances, although we found the performance was generally insensitive to these paraemeters.

### *4.3. Results*

The results for instance set A in terms of solution quality are displayed in Figure 4. FF shows good results throughout, improving as the problem instances get larger, which indicates that it is fairly simple for a greedy solver to create good-quality solutions for the non-bottlenecked version of the VMP problem. In all but one instance, OEMACS is able to match or exceed the solutions created by FF. Likewise, PACO-VMP outperforms or matches OEMACS on 5 sizes of instances including the largest instances. It should be noted that the PACO-VMP algorithm utilises the FF result as its initial best tour, meaning that it is not able to find worse tours than FF. A distinction between the results of set A and our other instance sets is that FF is competitive with the two ACO algorithms. For our other instance sets this is not the case, but as the non-bottlenecked problem is fairly straightforward, it allows FF to find good quality solutions. GA2 also performs well on this dataset, outperforming PACO-VMP on all but a single dataset. On the other hand, GA1 struggles, remaining moderately competitive for the smaller instances but performing dramatically worse on the 400, 500 and 1000 VM instances.

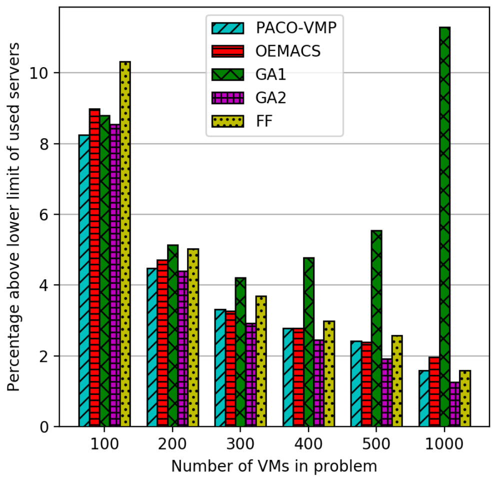


Figure 4: Solution difference measured as percentage over theoretical optimum for PACO-VMP, OEMACS, GA1, GA2 and FF for instance set A.

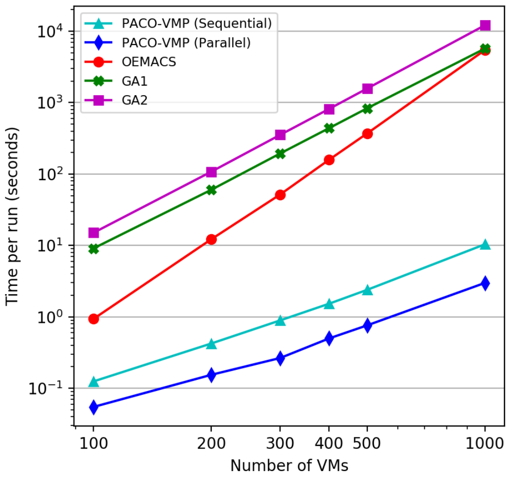


Figure 5: Average time to solve (seconds) for 100 instances of a given instance size for PACO-VMP, OEMACS, GA1 and GA2 for instance set A.

Execution time results for instance set A are shown in Figure 5. From this plot it is clear to see that PACO-VMP has a significant advantage over OEMACS when it comes to execution time, beginning at around 1 order of magnitude for the size 100 instances, and increasing to an advantage of around 3 orders of magnitude for the 1000 VM instance sets. An even larger advantage is held over the two IGA-POP algorithms, beginning at around 2 orders of magnitude for the 100 VM instances and increasing to around 3 orders of magnitude for the 1000

VM instances. Interestingly, despite beginning with a sizeable time advantage over IGA-POP, OEMACS performs similarly to GA1 for the 1000 VM instance set. The parallelized version of PACO-VMP increases the time difference between it and the sequential variant of PACO-VMP, increasing from a speedup of 2.2× for the 100 VM instances to a speedup of 3.47× for 1000 VM instances.

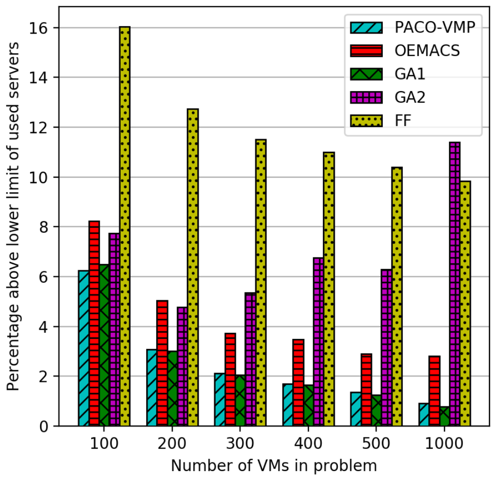


Figure 6: Solution difference measured in percentage over theoretical optimum for PACO-VMP, OEMACS, GA1, GA2 and FF for instance set B.

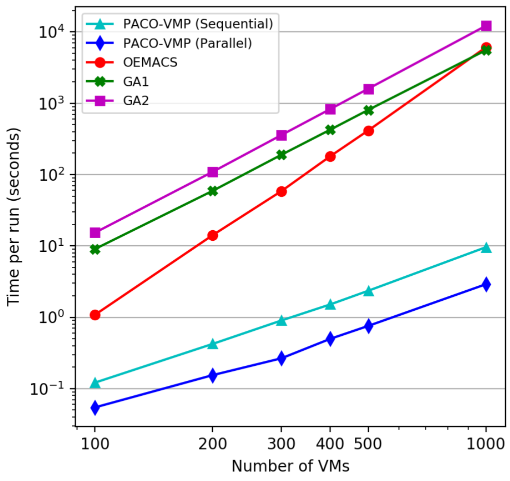


Figure 7: Average time to solve (seconds) for 100 instances of a given instance size for PACO-VMP, OEMACS, GA1 and GA2 for instance set B.

Unlike instance set A, the results for instance set B displayed in Figure 6 show a clear difference between PACO-VMP and OEMACS. FF's poor results also indicate that a greedy solver has more difficulty finding a good solution for the bottlenecked VMP than for the non-bottlenecked variant. While OEMACS significantly outperforms FF, PACO-VMP outperforms it for every problem size, finding solutions that range from 1%-2% closer to the theoretical lower limit. Additionally, the solution quality in terms of percentage is actually worse for the size 1000 instances with OEMACS, whereas PACO-VMP continues to improve. In contrast to Set A, GA1 performs very well in this bottle-necked scenario, with PACO-VMP returning better results for the 100 VM instances but then returning very slightly worse results for the larger instances. Conversely, GA2 performs poorly, initially returning similar results to OEMACS before worsening on the larger instances, and even being outperformed by FF for the 1000 VM instances.

In terms of execution time, displayed in Figure 7, the results for PACO-VMP are near-identical to the results for instance set A, demonstrating that the bottleneck led to no additional execution time. This is also the case for OEMACS, which also achieved near-identical execution times to the instance set A results. The execution time advantage held by PACO-VMP is maintained, with OEMACS once again losing the advantage it holds over IGA-POP as the solution size increases. The difference between sequential and parallel PACO-VMP is also near-identical to instance set A, though the speedup increase is slightly smaller, from 2.2× to 3.27×.

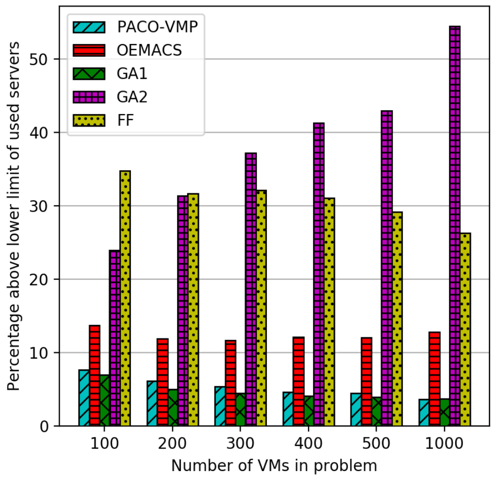


Figure 8: Solution difference measured in percentage over theoretical optimum for PACO-VMP, OEMACS, GA1 and GA2 and FF for instance set C.

The results shown in Figure 8 indicate that FF performs very poorly on instance set C, with the heterogeneous servers causing issues for the greedy technique. OEMACS significantly outperforms FF once again, but is itself outperformed by PACO-VMP, with solution quality improvement ranging from 5% for 100 VM instances to around 10% for 1000 VM instances. While OEMACS performs significantly worse on instance set C than the other sets, PACO-VMP is able to capably solve the heterogeneous instances. As with instance set B, while OEMACS begins to return worse solution qualities for the size 1000 instances, PACO-VMP continues to improve as the instance size increases. The performance of the GA variants is also consistent with set B, with GA1 slightly outperforming PACO-VMP

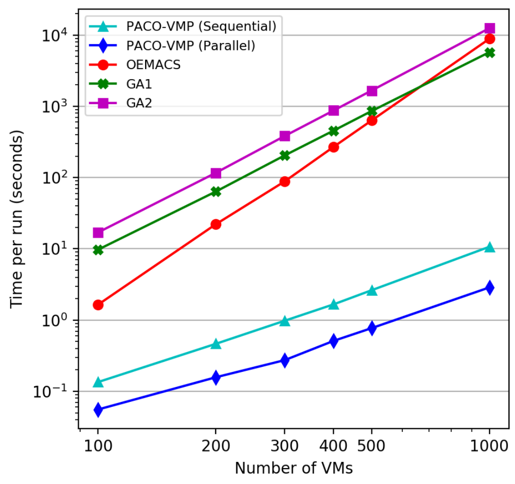


Figure 9: Average time to solve (seconds) for 100 instances of a given instance size for PACO-VMP, OEMACS, GA1 and GA2 for instance set C.

in all but one instance size, and GA2 performing poorly, showing even poorer results on instance set C.

Execution time for instance set C, as displayed in Figure 9, is similar to the other instance sets, with the execution time of PACO-VMP being near identical. However, OEMACS takes slightly longer to solve the instances in set C, further increasing the execution time advantage held by PACO-VMP. Additionally, the execution time of OEMACS is now closer to the time of GA2 than GA1 for 1000 VM instances, emphasising the increased difficulty that OEMACS has when trying to solve the bottlenecked, homogeneous problem. As with the previous instance sets, the time difference between the sequential and parallel PACO-VMP increases slightly as the instance sizes increase, from 2.6× to 3.78×.

## *4.4. Discussion*

We have demonstrated the significant execution time reduction that can be enabled through the use of parallelization and vectorization techniques on a wide range of different problem instance sets that represent three realistic Cloud Computing scenarios. PACO-VMP outperforms OEMACS in each instance set, very slightly in set A and significantly in sets B and C, and while it is matched by GA1 in set B and C, it performs significantly better in instance set A. The opposite is true of GA2, which outperforms PACO-VMP in set A, but significantly underperforms in sets B and C. The consistency demonstrates the versatility of ACO compared to IGA-POP: while IGA-POP performs well, it requires two separate fitness functions in order to match PACO-VMP. Upon further analysis of the IGA-POP results, the issue stems from the tendency of the algorithm to assign VMs to empty PMs even when currently used PMs have enough capacity remaining, which happens regardless of fitness function. Both PACO-VMP and OEMACS enforce a limit on the number of PMs than can be used (the previous best number of PMs) which prevents this behaviour. Additionally, it is worth nothing that despite a 10x increase of instance size in our experiments, the quality of the solutions produced by PACO-VMP remains consistent. The percentages above the lower limit of PM utilization decrease with each increase in instance size, which in terms of raw numbers indicates a fairly consistent number of PMs over the minimum. This suggests that our implementation could still produce good results for even larger VMP instances. This is an advantage over OEMACS, which provides degrading solution quality for size 1000 instances, a trend which would potentially continue as instance sizes increase.

The main focus of PACO-VMP is to improve execution time, and it succeeds at this objective. While PACO-VMP and OEMACS use similar pheromone definitions and local search techniques, PACO-VMP produces better results both in terms of execution time and solution quality. This may be caused partly by the choice of $\mathcal{MM}$AS algorithm over ACS, and also by differences in the selection probabilities due to the use of independent roulette. It has been shown [25] that independent roulette algorithms (such as *vRoulette*) tend to make greedier selections than the traditional roulette wheel algorithm, which may be a factor in the different solution qualities found between PACO-VMP and OEMACS. Clearly the areas in which PACO-VMP and OEMACS differ are significant in terms of execution time, as PACO-VMP has a time complexity of $O(n^2)$, whereas OEMACS is, experimentally, closer to $O(n^4)$. The main contributing factor to this is the probability calculation: whereas PACO-VMP uses the resource wastage formula as given in Formula 1 as the heuristic value, OEMACS uses a much more complex formula that includes the resource wastage, but also has extra sums over the VMs in both the numerator and denominator of the formula. Experimentally, the time complexity of the IGA-POP variants is approximately $O(n^3)$.

We summarize the results in Table 2. This table shows the results for solution quality and execution time, and for the solution quality results we indicate which results are statistically significant. The results of each algorithm on each instance of a given size can be paired, and compared to each other using the Wilcoxon signed-rank test, a non-parametric test which can be used to compare paired sets of readings. Since we perform an all-vs-all comparison of three tests (all possible pairs of algorithms) we apply the Bonferroni correction, and divide the significance threshold by the number of tests (in this case 3). Bold values in the table show the solution quality values which are significantly better than the other four algorithms, using a significance threshold of 0.002 (that is, 0.01 after application of the Bonferroni correction). In general, one of the two GA versions tends to produce the best solutions, however although the differences are in many cases statistically significant, the magnitude of the effect is small. For example, in the case of the A1000 instances, a comparison between GA2 and PACO shows that out of the 100 trials, GA2 is superior for 46 instances whereas ACO is superior for 3, with 51 ties. Although this is a statistically highly significant difference, the *magnitude* of the difference is only 0.32% in solution quality. This demonstrates that the experiments are very sensitive in detecting significant, but small, differences in performance. Qualitatively, the results show that one of the two GAs generally performs the best for any set of in-

Table 2: Results of our experiments on FF, OEMACS and PACO-VMP. Entries in the Set column represent the 100 instances of the specified size from the specific instance set. Solution Quality is the average percentage over the theoretical minimum for all 100 problem instances for each size within each set with values in **bold** being the best result, on the condition that it is significantly different when results are analysed with the Wilcoxon signed-rank test. Execution Time is the average time per run in seconds for all 100 problem instances for each size within each set. The sequential and parallel versions of PACO-VMP are included as PACO(S) and PACO(P) respectively

| | **Solution Quality (%)** | | | | | **Execution Time (seconds)** | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Set | FF | OEMACS | GA1 | GA2 | PACO | OEMACS | GA1 | GA2 | PACO(S) | PACO(P) |
| A100 | 10.3 | 8.98 | 8.70 | 8.54 | 8.25 | 0.936 | 9.098 | 15.05 | 0.125 | 0.055 |
| A200 | 5.03 | 4.71 | 5.13 | 4.4 | 4.47 | 12.22 | 60.23 | 107.9 | 0.423 | 0.154 |
| A300 | 3.68 | 3.26 | 4.21 | **2.92** | 3.32 | 51.81 | 194.4 | 354.3 | 0.892 | 0.265 |
| A400 | 2.98 | 2.78 | 4.78 | **2.46** | 2.78 | 158.8 | 440.0 | 817.7 | 1.528 | 0.499 |
| A500 | 2.58 | 2.39 | 5.54 | **1.92** | 2.42 | 369.9 | 832.8 | 1575 | 2.387 | 0.759 |
| A1000 | 1.58 | 1.96 | 11.3 | **1.26** | 1.58 | 5450 | 5711 | 10478 | 10.37 | 2.986 |
| B100 | 16.0 | 8.24 | 6.49 | 7.75 | 6.24 | 1.082 | 9.044 | 15.32 | 0.121 | 0.054 |
| B200 | 12.7 | 5.05 | 3.01 | 4.78 | 3.08 | 14.14 | 59.15 | 109.2 | 0.422 | 0.154 |
| B300 | 11.5 | 3.72 | 2.05 | 5.36 | 2.11 | 58.75 | 189.7 | 358.6 | 0.902 | 0.266 |
| B400 | 11.0 | 3.47 | 1.64 | 6.77 | 1.69 | 181.1 | 426.2 | 829.1 | 1.514 | 0.499 |
| B500 | 10.4 | 2.89 | 1.25 | 6.30 | 1.37 | 414.8 | 805.4 | 1596 | 2.351 | 0.756 |
| B1000 | 9.83 | 2.82 | **0.78** | 11.39 | 0.91 | 6080 | 5546 | 10321 | 9.594 | 2.904 |
| C100 | 34.8 | 13.7 | **6.96** | 23.9 | 7.67 | 1.656 | 9.733 | 16.98 | 0.135 | 0.056 |
| C200 | 31.6 | 11.9 | **5.03** | 31.4 | 6.10 | 22.14 | 63.66 | 116.7 | 0.464 | 0.158 |
| C300 | 32.1 | 11.7 | **4.46** | 37.2 | 5.39 | 88.56 | 205.0 | 383.7 | 0.980 | 0.274 |
| C400 | 31.0 | 12.1 | **4.07** | 41.3 | 4.66 | 270.3 | 456.8 | 874.7 | 1.663 | 0.511 |
| C500 | 29.2 | 12.0 | **3.93** | 43.0 | 4.44 | 633.0 | 858.4 | 1665 | 2.623 | 0.771 |
| C1000 | 26.3 | 12.8 | 3.70 | 54.5 | 3.62 | 8873 | 5753 | 10347 | 10.69 | 2.873 |

stances, but this is often accompanied by the other GA performing the worst. Since we used the GA with the recommended parameters for the population size and number of generations, this performance also comes at a significant cost; for example in the 1000 VM instances, GA1 and GA2 will perform 4000 evaluations per generation for 200 generations, compared to 20 evaluations for 50 iterations in PACO. Furthermore, the difference in performance between the two cost functions is very clear; using the original cost function proposed by [1] leads to poor performance on the B and C instance sets. PACO-VMP on the other hand, achieves solution quality close to best (or best) across all instance types, without any sensitivity to the algorithm parameters, and achieves better average solution quality than the other ACO algorithm (OEMACS) in 15 out of the 18 instance categories. There is also a clear advantage for PACO-VMP in both scalability and execution time. The computational complexity of PACO is superior to both OEMACS and GA1/2, and the execution time of the parallel version is several orders of magnitude less in most cases. For the C1000 instances, the most challenging instance set, PACO-VMP achieves the best solution quality of all algorithms in an average time of 2.873s, while GA1/2 and OEMACS require several hours of CPU time to reach a solution.

## 5. Conclusions & Future Work

In this paper we presented PACO-VMP, a parallelized and vectorized implementation of $\mathcal{MM}$AS for solving the Virtual Machine Placement problem. The method is several orders of magnitude faster than two current state-of-the-art ACO solvers, OEMACS and IGA-POP while producing comparable or superior results. Since virtual machine placement in the real world is a problem in which reducing time to solution can have significant cost benefits, the improved execution time performance of PACO-VMP

While PACO-VMP is capable of solving the static VMP problem, in reality this problem is rarely static. Real-world cloud workloads have constantly changing demand, with Virtual Machines being added and removed from the workload constantly. As with the static VMP, execution time is vital for dynamic VMPs in order to minimise time spent in an inefficient configuration, and PACO-VMP's positive results on the static problem indicate that it could also be effectively used to solve the dynamic problem. This is an area for further investigation.

The parameter tuning phase of our experiments revealed that the performance of the algorithm is relatively insensitive to the parameter governing the importance of pheromone information, further suggesting that analysing and improving our pheromone definition may lead to better solution quality from the underlying ACO mechanism. This is a potentially fruitful area of further work.

Many assumptions were made in our implementation regarding the VMP problem, including that there will always be as many PMs available as VMs, that performance doesn't degrade when the PMs reach 100% capacity, and that CPU and RAM are the only requirements. These assumptions are commonly made to simplify the problem solving process rather than having to consider a vast array of additional variables. Another potentially fruitful area that is fairly to investigate is the use of additional parameters for the VMP problem, rather than just CPU and RAM. Further work is required to investigate the inclusion of these additional parameters.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Acknowledgements

JP is funded by the Centre for Advanced Computational Science at Manchester Metropolitan University.

## References

[1] A.S. Abohamama and Eslam Hamouda. A hybrid energy–aware virtual machine placement algorithm for cloud environments. *Expert Systems with Applications*, 150:113306, 2020.

[2] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376, 2015.

[3] Fares Alharbi, Yu-Chu Tian, Maolin Tang, Wei-Zhe Zhang, Chen Peng, and Minrui Fei. An Ant Colony System for Energy-efficient Dynamic Virtual Machine Placement in Data Centers. *Expert Systems with Applications*, 120:228–238, 2019.

[4] Adriana Alvim, Fred S Glover, Celso C Ribeiro, and Dario J Aloise. Local Search for the Bin Packing Problem. 1999.

[5] Mauro Birattari. On the estimation of the expected performance of a metaheuristic on a class of instances. how many instances, how many runs? Technical Report TR/IRIDIA/2004-001, IRIDIA, Université Libre de Bruxelles, Brussels, Belgium, 2004.

[6] A. Botta, W. de Donato, V. Persico, and A. Pescapé. On the Integration of Cloud Computing and Internet of Things. In *2014 International Conference on Future Internet of Things and Cloud*, pages 23–30, 2014.

[7] 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.

[8] 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.

[9] José M Cecilia, José M Garcia, Manuel Ujaldón, Andy Nisbet, and Martyn Amos. Parallelization Strategies for Ant Colony Optimisation on GPUs. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 339–346. IEEE, 2011.

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

[11] Darren M. Chitty. Applying ACO to Large Scale TSP Instances. In Fei Chao, Steven Schockaert, and Qingfu Zhang, editors, *Advances in Computational Intelligence Systems*, pages 104–118, Cham, 2018. Springer International Publishing.

[12] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286, 2005.

[13] EG Coffman Jr, MR Garey, and DS Johnson. Approximation Algorithms for Bin Packing: A Survey. *Approximation Algorithms for NP-hard Problems*, pages 46–93, 1996.

[14] 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.

[15] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant Colony Optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39, 2006.

[16] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. Ant System: Optimization By a Colony of Cooperating Agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 26(1):29–41, 1996.

[17] Ivars Dzalbs and Tatiana Kalganova. Accelerating Supply Chains with Ant Colony Optimization across a Range of Hardware Solutions. *Computers & Industrial Engineering*, 147:106610, 2020.

[18] Eugen Feller, Louis Rilling, and Christine Morin. Energy-aware Ant Colony Based Workload Placement in Clouds. In *2011 IEEE/ACM 12th International Conference on Grid Computing*, pages 26–33. IEEE, 2011.

[19] Damián Fernández-Cerero, Alejandro Fernández-Montes, and Agnieszka Jakóbik. Limiting global warming by improving data-centre software. *IEEE Access*, 8:44048–44062, 2020.

[20] Yongqiang Gao, Haibing Guan, Zhengwei Qi, Yang Hou, and Liang Liu. A Multi-objective Ant Colony System Algorithm for Virtual Machine Placement in Cloud Computing. *Journal of Computer and System Sciences*, 79(8):1230–1242, 2013.

[21] J. A. Guerrero-ibanez, S. Zeadally, and J. Contreras-Castillo. Integration Challenges of Intelligent Transportation Systems with Connected Vehicle, Cloud Computing, and Internet of Things Technologies. *IEEE Wireless Communications*, 22(6):122–128, 2015.

[22] Brian Hayes. Cloud computing, 2008.

[23] Xiao-Fang Liu, Zhi-Hui Zhan, Jeremiah D Deng, Yun Li, Tianlong Gu, and Jun Zhang. An Energy Efficient Ant Colony System For Virtual Machine Placement in Cloud Computing. *IEEE Transactions on Evolutionary Computation*, 22(1):113–128, 2016.

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

[25] Huw Lloyd and Martyn Amos. Analysis of independent roulette selection in parallel ant colony optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, page 19–26, New York, NY, USA, 2017. Association for Computing Machinery.

[26] Seyed Saeid Masoumzadeh and Helmut Hlavacs. Integrating VM Selection Criteria in Distributed Dynamic VM Consolidation using Fuzzy Q-Learning. In *Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013)*, pages 332–338. IEEE, 2013.

[27] Haibo Mi, Huaimin Wang, Gang Yin, Yangfan Zhou, Dianxi Shi, and Lin Yuan. Online Self-reconfiguration with Performance Guarantee for Energy-efficient Large-scale Cloud Computing Data Centers. In *2010 IEEE International Conference on Services Computing*, pages 514–521. IEEE, 2010.

[28] Seyedali Mirjalili. Sca: A sine cosine algorithm for solving optimization problems. *Knowledge-Based Systems*, 96:120 – 133, 2016.

[29] M. Satyanarayanan. The Emergence of Edge Computing. *Computer*, 50(1):30–39, 2017.

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

[31] Mark Stillwell, David Schanzenbach, Frédéric Vivien, and Henri Casanova. Resource allocation algorithms for virtualized service hosting platforms. *Journal of Parallel and distributed Computing*, 70(9):962–974, 2010.

[32] Thomas Stützle. MAX-MIN Ant System for Quadratic Assignment Problems. 1997.

[33] Fei Tao, Chen Li, T Warren Liao, and Yuanjun Laili. BGM-BLA: A New Algorithm For Dynamic Migration of Virtual Machines in Cloud Computing. *IEEE Transactions on Services Computing*, 9(6):910–925, 2015.

[34] 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.

[35] 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.

[36] Shangguang Wang, Zhipiao Liu, Zibin Zheng, Qibo Sun, and Fangchun Yang. Particle Swarm Optimization for Energy-aware Virtual Machine Placement Optimization in Virtualized Data Centers. In *2013 International Conference on Parallel and Distributed Systems*, pages 102–109. IEEE, 2013.

[37] D. Wilcox, A. McNabb, and K. Seppi. Solving Virtual Machine Packing With a Reordering Grouping Genetic Algorithm. In *2011 IEEE Congress of Evolutionary Computation (CEC)*, pages 362–369, 2011.

[38] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi. Internet of things for smart cities. *IEEE Internet of Things Journal*, 1(1):22–32, 2014.