UNIVERSITE CATHOLIQUE DE LOUVAIN

ECOLE POLYTECHNIQUE DE LOUVAIN

Study and Analysis of Networks Flows

Supervisor: Yves Deville
Readers: FranÃ§ois Aubry
Jean-Charles Delvenne

Thesis submitted for the Master's degree
in computer science and engineering
options: Artificial Intelligence
by Denis Genon & Victo Velghe

Louvain-la-Neuve
Academic year 2015-2016

Abstract

Ant Colony Optimization is a metaheuristic which mimics ants behavior to solve optimization problems. Ant Colony Optimization has proven to be very effective in various fields of applications. One inconvenient with ACO is the complex mapping of a problem into something that can be used by this metaheuristic. Also, adapting an ACO application from one problem to another is difficult and requires a lot of programming efforts.

The goal of this thesis is to build an application easing the use of ACO in routing problems by creating an implementation with a high level of abstraction. This is done through the building of a framework in Scala. The result is a framework that you can use to solve different vehicle routing problems. The framework permits to add specific constraints in vehicle routing problems in a quite easy way, as presented in this thesis.

In this research, tests were made on traveling salesman problems, capacitated vehicle routing problems, vehicle routing problems with time windows and capacitated vehicle routing problems with specific constraints. The results obtained with the framework are encouraging as in almost all cases they are comparable to other ACO implementations solving these problems.

Acknowledgment

This thesis represents the culmination of a long and fruitful journey. Over the years I have been blessed with friends and family who have supported and encouraged me. I would like to take the opportunity here to extend my heartfelt thanks and love to all of them. Pursuing my academic program at UCL has been a thoroughly enjoyable experience, full of learning and laughter. There were tougher moments too, and I am especially grateful to my entourage for their faithful friendship which helped me to overcome the challenges along the way.

To Tanya, my beautiful girlfriend, thank you for your love, your presence and your belief in me.

To my friends I say thank you for all of the good times we shared together. I look forward to many more of course!

To my family I would like to express my thanks for all of your love and for being a source of constant motivation.

Finally, I would like to give special thanks to my Promoter for his invaluable insights and his supervision. Your help contributed greatly towards the successful completion of this thesis.

Contents

Chapter 1

Introduction

Nowadays, solving hard combinatorial optimization problems is a very important field of artificial intelligence. Different techniques using heuristics and metaheuristics are used to solve these problems. Ant Colony Optimization (ACO) is one of these metaheuristics approaches that appeared in 1999 [1] [2]. This metaheuristic has proven to be very effective in various fields of applications such as in vehicle routing problems. Solving a problem using an ACO metaheuristic requires a lot of programming work as there is no ACO solver at the moment. One inconvenient with ACO is the complex mapping of a problem into something that can be used by this metaheuristic to build solutions. It is difficult to adapt an application of ACO from one problem to another. When using ACO, each constraint added or removed in a problem changes the whole implementation.

The goal of this master thesis is to create a framework that allows a user to use ACO to solve different routing problems by adding its own constraints in an easy way and to give a fixed representation of these constraints.

1.1 Ant Colony Optimization

The family of Ant Colony Optimization metaheuristic is originated in the observation of the behaviour of real ants and their interactions with their environment. These observations have lead to the creation of a metaheuristic now widely used.

1.1.1 Biological inspiration

The original idea comes from the observation of the exploitation of the food resources by the ants. Although ants have limited cognitive resources as individuals, when working together they are able to find the shortest route from their nest to a food source. Biologists made a series of experiments: An ant colony has the choice between two paths of unequal length that lead to a food source. After some time, they eventually use the shortest path. The biologists explain this behavior in the following way:

1. An ant (called scout) walks randomly around the colony.

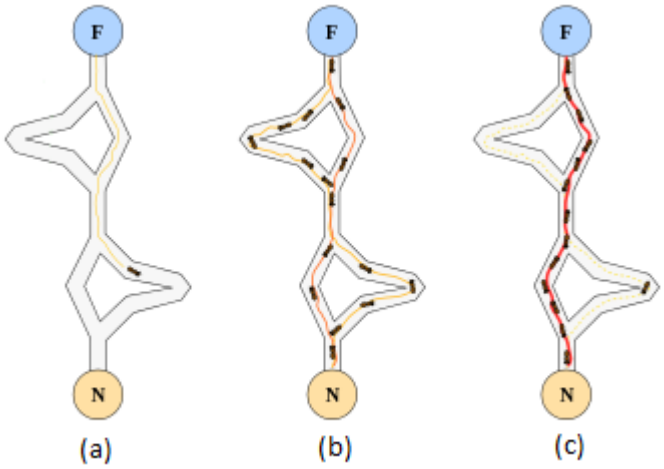


Figure 1.1: Food discovery by an ant colony

2. If that ant discovers a food source, she comes back to the nest, leaving a trail of pheromones; see figure 1.1 (a).

3. These pheromones being attractive for the other ants, the other ants will have a tendency to follow the pheromone trail; see figure 1.1 (b).

4. Coming back to the nest, these new ants will reenforce the trail.

5. If two paths are available to reach the same food source, the shortest one will be, in a fixed time interval, used by more ants than the longest one because of the evaporation.

6. The shortest path will be more and more attractive to the ants.

7. The longest path will eventually disappear, pheromones being volatile; see figure 1.1 (c).

The ants use the environment as a communication support. They exchange information by dropping pheromones on their lines of movement. The information has a local impact: only an ant that is close to the pheromones can access them and use them. The French entomologist Pierre-Paul Grasse created the word *"stigmergy"* to describe this type of communication where the actual exploration is stimulated by the performance already achieved [3]. This mechanism permits to an ant colony to solve a problem that is too complex for one single ant. It is a good example of a self-organized system. This system relies on positive feedback (the pheromone trail attracts other ants that reenforce the trail) and negative feedback (dissipation of the pheromones by evaporation over time). Theoretically, if the amount of pheromone on a path is constant over time (there is no evaporation), no path would be chosen, but the positive and negative feedback permits the amplification of the pheromone trail on a particular path and not the other. This leads to the choice of this path.

1.1.2 The Ant Colony Optimization Metaheuristic

History

After the elaboration of the stigmergy theory by Pierre-Paul Grasse in 1959, many researchers and academic experts have analyzed ants and how they behave. In 1989, the works of Goss, Aron, Deneubourg and Pasteels on the collective behavior of the argentine ants provided the necessary ground work for the development of ant colony algorithms. In 1992, Marco Dorigo proposed the Ant System in his doctoral thesis [4]. This was the first algorithm based on ant behavior. The Ant System will be improved in several ways to become in 1997 the Ant Colony System. The term Ant Colony Optimization metaheuristic (ACO) appears in 1999 to regroup the algorithms that builds solutions using the Ant (Colony) System. Until now, ACO has evolved in different ways and has been applied to a lot of different problems (see section 1.2.1).

Optimization problem

An optimization problem, in mathematics and computer sciences, is the problem of finding the best solution to a problem, using a given quantitative criterion, from the set of all feasible solutions. There are two categories of optimization problem depending on whether the variables are continuous or discrete. If the variables are discrete, the problem is called a combinatorial optimization problem. The solution to that type of problem is in the form of an integer, a permutation or a graph from a finite set. Some of these problems are known as NP-hard. To solve these NP-hard problems, heuristics and metaheuristics are used. These techniques aim to solve these optimization problems.

Metaheuristic

A metaheuristic is an optimization algorithm aiming at the solving of an optimization problem. Metaheuristic are usually iterative stochastic algorithms progressing through a global optimum. They behave like search algorithms, trying to learn some characteristics of a problem to find an approximation of the best solution. Metaheuristics use their history to guide the optimization on the following iterations. In the simplest case, they use only one previous state to determine the next iteration. We speak in that case of a method without memory. However, a lot of metaheuristics use a more evolved memory on short term or long term, using a set of parameters describing the research. The use of metaheuristics has significantly increased the ability of finding very high-quality solutions to hard, practically relevant combinatorial optimization problems in a reasonable time. [5]

Ant Colony Optimization

Ant colony optimization is a metaheuristic in which a colony of artificial ants cooperates in finding good solutions to difficult discrete optimization problems [5]. In this metaheuristic, we let simple agents (ants) build good solutions using their own channel of communication that is the pheromone trail. This type of communication permits to learn from the previous solutions constructed by the ants.

In ACO, an artificial ant is a constructive procedure that builds a solution incrementally. ACO can therefore be used in any problem where a constructive heuristic can be defined.

The ants in ACO implement a randomized construction heuristic that builds a solution by iteratively adding components to partial solutions by taking into account in a probabilistic way:

- heuristic information about the problem instance being solved.
- pheromone trails which change dynamically at run-time and reflect the ants search experience.

Problem Representation

Consider the static (non time-dependent) minimization problem (S, f, Ω) where S is the set of candidate solutions, f is the objective function that assigns a cost $f(s)$ to each candidate solution $s \in S$ and Ω is a set of constraints. The goal is to find an optimal feasible solution s^* : A minimum cost feasible solution to the problem.

This problem can be mapped in a way that will be usable by the ants. The mapping of the problem is characterized by the following items[5]:

- A finite set $C = c_1, c_2, ..., c_{N_C}$ of components is given, N_C is the number of components. Each c_i are components of the set of candidate solutions.
- The states of the problem are defined in terms of sequences $x = [c_i, c_j, ..., c_h, ...]$ of finite length over the elements of C. The set of all possible states is denoted by X. The maximum length of a sequence x is bounded.
- The set of possible solutions is denoted by S. $S \subseteq X$.
- The set of feasible solutions is denoted by $\tilde{S}$. $\tilde{S} \subseteq S$. This set of feasible solutions is obtained from S via the constraints Ω.
- A non-empty set of optimal solutions $S^* \subseteq S$.
- A cost $g(s, t)$ is associated with each solution $s \in S$. In most case $g(s) \equiv f(s)$
- Sometimes a cost can be associated with states other than candidate solutions.

With this formulation, an ant can build a solution by performing a randomized path on the completely connected graph $G_C = (C, L)$ The nodes of the graph are the components C described above and the set of arcs L fully connects the components of C. The constraints Ω are implemented in the policy followed by the ants. This permits to chose if the constraints must be implemented in a hard way (the ants can not build infeasible solutions - solutions that violate constraints) or in a soft one (the ants are allowed to build infeasible solutions that will be penalized as a function of their degree of infeasibility).

Problem type	Problem name
Routing	Traveling salseman
	Vehicle routing
	Sequential ordering
Assignment	Quadratic assignment
	Graph coloring
	Generalized assignment
	Frequency assignment
	University course timetabling
Scheduling	Job shop
	Open shop
	Flow shop
	Total tardiness
	Total weighted tardiness
	Project sheduling
	Group shop
Subset	Multiple knapsack
	Max independent set
	Redundancy allocation
	Set covering
	Weight constrained graph tree partition
	Arc-weighted l-cardinality tree
	Maximum clique
Network routing	Connection-oriented network routing
	Connectionless network routing
	Optical network routing

Table 1.1: Existing applications of ACO algorithms

1.2 State of the Art

1.2.1 Existing applications of ACO algorithms

ACO algorithms have been applied to different types of problems and have been widely studied. Table 1.1 provides a quick summary of the current applications of ACO algorithms.

In this thesis we will focus on the traveling salesman problem, the capacitated vehicle routing problem and the vehicle routing problem with time windows. These three will be explained in more detail in section 1.3.

1.2.2 Results and usage of these applications

ACO algorithms have for the moment comparable performances to the best existing methods to solve some of the problems listed here after : the sequential ordering problem [6], the vehicle routing with time window problem [7], the quadratic assignment problem [8], the group shop scheduling problem [9], the arc-weighted l-cardinality tree problem [10], the shortest common supersequence problem [11] and finally network routing problems [12]. The success of these algorithms attracted widespread attention throughout the scientific community. This encouraged research in the field of ACO applications.

1.3 Problems of Interest

This thesis focuses on three problems: the Traveling Salesman problem, the capacitated vehicle routing problem and the vehicle routing problem with time windows. These problems are described hereafter.

1.3.1 Traveling Salesman Problem

The traveling salesman problem is a mathematical problem where, given a set of cities for which the distance between each of them is known, the goal is to find the shortest path that passes through all of these cities exactly once. It's an optimization problem for which there is no known polynomial time algorithm. The decisional version (for a distance D, is there a path that passes through all the cities with a length less or equal to D?) is known as an NP-Hard problem. More formally, the TSP can be represented as a complete weighted graph $G = (N, A)$, N being the set of nodes representing the cities, and A the set of arcs. Each arc $(i, j) \in A$ is assigned a value d_{ij} (the distance between the two cities i and j $(i, j \in N)$). We will work only with symmetric TSP so $d_{ij} = d_{ji}$ for all vertices. The goal is to find the minimum length Hamiltonian cycle of this graph G. An Hamiltonian cycle is a closed path that visits each nodes $n \in N$ of G exactly once. An optimal solution to this problem is thus a permutation π of the node index such that the length $f(\pi)$ is minimal. π is thus a bijection on the set of nodes N on itself. $\pi(i)$ denotes the i^{th} element of the permutation π. $f(\pi)$ is given by [5] :

$$f(\pi) = \sum_{i=1}^{n} d_{\pi(i)\pi(i+1)} + d_{\pi(n)\pi(1)} \tag{1.1}$$

In equation 1.1, a sum of the distance between all consecutive node in permutation π is made. To that sum, is added the distance between the first and the last node of the permutation to close the cycle. This gives the value to the length of the permutation π.

1.3.2 Vehicle Routing Problem

The vehicle routing problem (VRP) is a combinatorial optimization problem. The goal is to serve a number of customers with a fleet of vehicles. It is an important problem in the field of transportation, distribution and logistics. The distribution of goods concerns the service of a set of customers by a set of vehicles which are located in one or more depots. In this thesis, we

only work with one depot. A solution of a VRP consists in the determination of a set of routes, each traveled by a single vehicle that starts and ends at the depot, such that all the needs of the customers are fulfilled and no operational constraints are violated (maximal load of a truck for example). In this thesis, we work with the capacitated Vehicle Routing Problem and with the Vehicle Routing Problem with Time-Windows. This is why we develop these two more in detail.

Capacitated Vehicle Routing Problem

In the CVRP, the customers correspond to delivery points and the demands of these customers are known in advance. The vehicles of the fleet are all the same and are based in a single central depot. In this problem, the only restriction is the capacity of the vehicles. The goal is to minimize the total cost to serve all of the customers (the travel time, the number of routes, or the length of the route). Here is the mathematical description of the problem: As for the TSP, the problem is represented with a complete graph $G = (N, A)$ where $N = 0, ..., n$ is the node set and A is the arc set. The depot corresponds to the node 0. The nodes $1, ..., n$ correspond to the customers. Again, with each arc $(i, j) \in A$ is associated a cost d_{ij} that represents the cost to travel from node i to node j. We work with symmetric graphs and thus $d_{ij} = d_{ji}\ \forall (i, j) \in A$. With each customer $i (i = 1, ..., n)$ is associated a demand $c_i > 0$. A set of K identical vehicles that have a capacity C are at the depot (we assume that $\forall i, c_i < C$ to ensure feasibility).

A CVRP solution consists of finding a collection of simple circuits (each corresponding to a vehicle route) with minimum cost, defined as the sum of the costs of the arcs belonging to the circuits. In a solution:

- each circuit visits the depot.
- each customer is visited by exactly one circuit.
- the sum of the demands of the clients visited in a circuit does not exceed the vehicle capacity C.

VRP with Time Windows

The VRPTW is an extension of the CVRP where with each customer i is associated a time interval $[a_i, b_i]$ in witch the customer has to be served that we call the time window. The cost d_{ij} in the CVRP is thus replaced by the travel time $t_{ij}\ \forall (i, j) \in A$. We add a service time s_i that represents the time needed to serve the client i. In this problem, we assume that all vehicles leave the depot at instant $t = 0$. The service of a customer must start within the time window of that customer and the vehicle must stop for s_i time instants. A VRPTW solution consists in finding a collection of simple circuits with minimum cost, and such that:

- each circuit visits the depot.
- each customer is visited by exactly one circuit.
- the sum of the demands of the clients visited in a circuit does not exceed the vehicle capacity C.

- for all customers i the service starts within the time window $[a_i, b_i]$ and the vehicle stops for s_i time.

Chapter 2

ACO abstraction for classes of problems

In this chapter, an analysis of the applications of ACO for the problems of interest are investigated. The goal of these investigations is to determine what can be abstracted to make an ACO framework. The applications made and analyzed in this chapter contribute towards the chapter 3 where the architecture of the Framework is presented along with the choices made for the implementation.

2.1 Application of ACO for the TSP

Let us start with a high level description of the application of ACO for the TSP. A TSP is formulated in the form of a fully connected graph $G = (C, L)$. Because of the fact that this graph is complete, an ant will always build a complete feasible tour (a path that visits all the nodes exactly once). The pheromone trails are associated with the arc $(i, j) \in L$ so that each arc has a value τ_{ij} corresponding to the pheromone trail. τ_{ij} gives a value to the desirability of visiting city j directly after i. This information is coupled with a heuristic information $\eta_{ij} = 1/d_{ij}$ that is inversely proportional to the distance between city i and j. A tour (a solution) is constructed with a simple constructive procedure :

- An ant is randomly positioned at a start city.
- The ant uses the pheromone and heuristic value to choose probabilistically the next city to add to her path. This is done until all cities have been visited.
- The ant goes back to her start city to complete the tour.

When all ants have constructed their tour, these tours are compared using a simple objective function that measures the total length of the tour. Pheromones are dropped on the path in a certain manner explained in section 2.1.2. The amount of pheromone dropped is a function of the tour quality. Also, an evaporation of the pheromone is applied (the concentration of pheromone diminishes), this is made to foster exploration. The representation of these pheromones τ_{ij} and

the distance d_{ij} between two cities is made with two square matrices. Note that when the ants have built a tour, it can be improved by the application of a local search procedure.

```
set parameters, initialize pheromone trails;
while termination condition not met do
    ConstructAntsSolution;
    ApplyLocalSearch //optional ;
    UpdatePheromones ;
end
```

Algorithm 1: ACO skeleton for TSP

2.1.1 Tour Construction

When the ants construct a tour, they have to choose the next city they will visit. As we said earlier, that choice is probabilistic and is ruled by the so-called *random proportional rule*. This rule is explained as : the probability that an ant k at city i goes to city j is :

$$p_{ij}^k = \begin{cases} \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in N_i^k} [\tau_{il}]^\alpha [\eta_{il}]^\beta} & \text{if } j \in N_i^k \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

In this equation 2.1, η_{ij} represents a heuristic information. In this case, as we want to minimize the travel length, $\eta_{ij} = \frac{1}{d_{ij}}$ where d_{ij} is the distance between city i and city j. N_i^k is the feasible neighborhood of ant k at city i. In other words, it is the set of cities that have not been visited yet by the ant k. τ_{ij} represents the amount of pheromone on the arc (i, j). α and β are parameters that affect the importance of the pheromone or the heuristic information. If $\alpha = 0$, the closest city will be chosen more often, and the pheromone information will not be used. If $\beta = 0$, only the pheromones will be used without heuristic. These two cases usually lead to a stagnation situation, all the ants following the same path and constructing the same tour.

Using this random proportional rule (equation 2.1) with good parameters α and β permits to balance the construction procedure between exploration and exploitation.

To make things work, each ant k has to keep in memory which cities have already been visited and the order in which they have been visited (indeed, they need to keep in memory the solution path they are building). This is also useful to determine the feasible neighborhood N_i^k.

2.1.2 Update of pheromone trails

The pheromone trails are modified in two ways. There is both a reinforcement and an evaporation of the trails. As explained in chapter 1, the evaporation is mandatory to make the algorithm work. There are many different systems that can be used to perform these two operations. We use here the system described in the Ant Colony System (ACS) [13] in which the reenforcement of pheromones is made using only the ant k that has made the best tour so far (according to the objective function). This method has proven to be the best. The evaporation is done each time

an ant uses an arc (i, j) to move from city i to city j. This is done to increase the exploration of alternative paths. As previously said, in ACS only one ant will be used to add pheromone after an iteration. The update of the pheromone follows the equation :

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij} + \rho\Delta\tau_{ij}^{bs}, \forall(i, j) \in T^{bs} \tag{2.2}$$

Where $\Delta\tau_{ij}^{bs} = \frac{1}{C^{bs}}$, T^{bs} is the best tour so far, and C^{bs} is the length of this best tour so far. ρ is a constant chosen to represent the pheromone evaporation rate. The use of it in the pheromone update equation permits to have a new trail that is a weighted average between the old pheromone value and the new amount deposited. As said before, you can see from the equation 2.2 that only the arcs belonging to the best solution are updated. To increase the exploration of alternative paths, a local pheromone trail update is done each time an ant use an arc (i, j) during the construction of a solution. This local update is made following the rule given hereafter :

$$\tau_{ij} \leftarrow (1 - \mathcal{E})\tau_{ij} + \mathcal{E}\tau_0 \tag{2.3}$$

With $\mathcal{E}$, $0 < \mathcal{E} < 1$ and τ_0 are two parameters. τ_0 is the initial value of the pheromone trails and $\mathcal{E}$ represent the importance of this local evaporation.

2.1.3 Algorithm and data structure used

To better understand how all these concepts will work together, an analysis of a first implementation of a TSP solver using an ACO metaheuristic is made. In this section, it is explained how the implementation is done, what kind of data structures are used, what methods are implemented and how they are classified. This will be helpful to understand the choices of implementation given in chapter 3.

To understand how all of the concepts given in the previous sections are put together, an algorithm that shows how the application will work is presented in Algorithm 2.

In algorithm 2, each ant builds a complete tour (`foreach Ant k do ...`). When all the ants have built a tour, the best one is selected and kept in memory. This best path is then used to update the pheromones. This construction step is made a certain number of times, the number `MAX_ITERATION`.

To apply this algorithm 3 different classes are used:

Ant The class Ant contains the two vectors named M_k and $Path$ in the algorithm 2. This class contains also the method $choseNextCity()$ that permits to choose the next city to visit using the equation 2.1.

Colony The class Colony is used to coordinate the ants and keep the information that are used by them. Like the Tau matrix. There is also an array of the ants that is used to launch the construction of the path of each ant one by one and collect their results. It is in this class that are located the objective function to measure the quality of a solution, and the method that updates the pheromone information.

Problem The class Problem contains the information relative to the problem, in the case of a TSP, the distance matrix only.

```
Distance ← matrix containing the distance informations;
Cities ← vector containing the different cities;
Tau ← matrix containing the pheromone information;
;                                          /* all values = τ0 at the beginning */
set parameters α, β, ρ, E, τ0;
while nb_iteration < MAX_ITERATION do
    T^bs ← empty;
    ;                                  /* contain the best tour of the iteration */
    foreach Ant k do
        M_k ← Cities;
        Path ← empty;
        current_city ← randomlyChoosenCity();
        while !M_k isempty do
            next_city ← choseNextCity();
            ;                           /* choseNextCity() uses the equation 2.1 */
            remove next_city from M_k;
            add next_city to Path;
            apply local pheromone update ;
            //using equation 2.3 current_city ← next_city;
        end
        localSearch();
        ;                                                          /* optional */
    end
    T^bs ← bestPath() ;   /* bestPath() select the best path constructed by the
    ants according to an objective function (for tsp the total length of
    the tour) */
    Update Pheromones ;
    ;                                                  /* using equation 2.2 */
end
```

Algorithm 2: Complete ACO algorithm for the TSP

2.2 Application of ACO for the CVRP

The CVRP is similar to the TSP except that data is added on the client: the demand, and a constraint on the ants (representing the vehicles): their maximum load.

2.2.1 Tour Construction

The tour construction is similar to the one used in the TSP except that if the next city chosen makes the ant violate her capacity constraint, she must come back to the depot first, closing the circuit she was building and starting a new one.

In certain pieces of literature, a new random proportional rule is proposed to increase the quality of the results of the ant colony system on the CVRP [14].

$$p_{ij}^k = \begin{cases} \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta [\mu_{ij}]^\gamma [\kappa_{ij}]^\lambda}{\sum_{l \in N_i^k} [\tau_{il}]^\alpha [\eta_{il}]^\beta [\mu_{il}]^\gamma [\kappa_{il}]^\lambda} & \text{if } j \in N_i^k \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

The first new element introduced in this equation is called the *savings* , μ_{ij}. In a CVRP, it is not only the relative location of two cities which is important (information that is held in η_{ij}). The savings add an information about the relative location of two cities to the depot. The savings measure the favorability of combining two cities i and j to a tour. The quantification of μ_{ij} is given by :

$$\mu_{ij} = d_{i0} + d_{0j} - d_{ij} \tag{2.5}$$

A high value for μ_{ij} indicates that visiting city j after city i is a good choice. A new parameter γ is introduced to regulate the influence of the savings as does α and β for the pheromones and the heuristic information respectively. The second element introduced in equation 2.4 is a characterization of the capacity utilization of the vehicles. This information is held in κ_{ij} and the quantification is given by :

$$\kappa_{ij} = (Q_i + q_j)/Q \tag{2.6}$$

Q_i is the total capacity used including the capacity requirement of customer i, q_j is the capacity requirement of customer j and Q is the maximum capacity of a vehicle. A high value of κ_{ij} indicates a high capacity utilization through the visit of j after visiting i. Again a new parameter λ is introduced to determine the influence of κ_{ij}

2.2.2 Update of pheromone trails

The pheromone update is made in the same way that for the TSP. As explained in section 2.1.2, a local pheromone update is made every time an ant uses an arc and a global update is made with the best solution constructed by the ants.

2.2.3 Algorithm and Data Structures

The changes explained in section 2.2.1 imply a few modifications in the application. Indeed the information about the demand of a customer and the maximum load of a vehicle has to be kept.

To do so, a new class is created, the `Client` class.

Client The Client class is a data class that contains the information about the clients. For the CVRP, it contains an array with the quantity demanded by each client.

In addition to that, two fields are added into the class Ant. One contains the maximum load that an ant can carry, the other the current load of the ant. Also, the method $choseNextCity()$ has been modified to take into account the fact that a trip back to the depot has to be made when the maximum load is reached.

2.3 Application of ACO for the VRPTW

With the VRPTW, another constraint is added to the route. It is the time window. A vehicle must arrive at a client i in a time frame associated to him. As for the CVRP we must add a new piece of information to the client: the time frame; and to the ants: the current time.

To solve the VRPTW problem, a simple solution is to extend the CVRP model to take into account the new time constraints. This is what we call the simple solution because it leads to poor results. Another solution called Multiple Ant Colony System permits the optimization of a multiple objective function. This approach is developed by Gambardella, Taillard and Agazzi in their paper [7]. These two solutions are explained hereafter.

2.3.1 Simple Solution

Tour Construction

For the tour construction in the VRPTW, a filter is used to isolate the feasible cities between the cities that have not been visited yet. A feasible city is a city j for an ant k standing at city i such that visiting j after i does not violate any constraints (load and time). Then, the probabilistic choice is made using this subset of feasible cities. This leads to better results. The tour construction of an ant in the VRPTW is resumed in algorithm 3. In this algorithm, a loop is made until the set M_k of unvisited cities is empty. From this set of cities are filtered the cities that lead to a violation of the constraints. That is, if the current city is i, the load of the truck is Q_i (quantity demanded at city i plus the load of all the cities visited before i) and the current time is T_i (time to go to city i), a city j will be discarded if the load $Q_i + q_j$ is greater than the maximum capacity of a truck, or if the time $T_i + t_{ij}$ is greater than the upper limit of the time window of city j. From this filtered set of cities, the next city is chosen and removed from the set M_k using equation 2.1 or 2.4. Then, the pheromones are updated.

Multiple Ant Colony System

The MACS-VRPTW implementation [7] is an ACO implementation designed to solve the VRPTW with two objective functions : the minimization of the number of vehicles (or tours) and the minimization of the total travel time. Note that the introduction of these two objective functions permits to obtain better results but it is not mandatory to solve a VRPTW (as explained in section 2.3.1).

```
M_k ← Cities;
Path ← empty;
feasible_city ← empty;
current_city ← depot;
while !M_k isempty do
    feasible_city ← computeFeasible();
    if feasible_city is empty then
        next_city ← depot
    else
        next_city ← choseNextCity(feasible_city);
        remove next_city from M_k;
    end
    add next_city to Path;
    apply local pheromone update ;
    current_city ← next_city;
end
```

Algorithm 3: Tour construction for the VRPTW

In this section is presented a simplified MACS-VRPTW version to have something comparable to the implementation presented in section 2.1 and 2.2. MACS is implemented to use parallelism to the fullest degree. The previous implementation did not use the parallelism so it is removed. Some optional functions that serve to improve the results as an insertion method or a specific local search method have also been removed in the simplified version presented hereafter.

In this implementation, the construction of a solution is made in two steps. The first step concerns the minimization of the number of vehicles used. The second step concerns the minimization of the total travel time.

Vehicle minimization The first step aims to find feasible solutions (solutions that visit all the cities without violating any constraints) with the least possible number of vehicles. The algorithm used is presented in algorithm 4. The algorithm works as follows. A first solution is built with a non limited number of vehicles. This is the so-called `first_sol`. This solution has a number of vehicles v. The `min_vehicle(nb_vehicle)` method is then called with as argument a number of vehicles $nb\_vehicle = v - 1$ as the number of vehicles must be reduced. The different parameters are initialized (pheromone, time, load ...) then for a defined number of iterations, each ant will try to build a feasible solution with the number of vehicles $v - 1$ (`run_ant(nb_vehicle, IN)` call). Because the number of vehicles is limited, a solution does not especially visits all of the customers. To favor the construction of a solution with all the nodes, a special vector `IN` is used. Each time a node j is not in a solution given by an ant, the value $IN[j]$ is incremented by one. This value is then used in the construction algorithm of the ant, that is explained later in algorithm 6. If the solution built by the ant k is better (visits more clients) than the current solution, it is kept in mind. If that solution is feasible (visits

all clients), it replaces the `best_feasible_solution` and a recursive call to the `min_vehicle` method is made with the number of vehicles diminished by one. This is done until we reach the maximum number of iterations. If these two conditions are not satisfied a pheromone update is done and the iteration continues.

```
best_feasible_solution ← first_sol;
def min_vehicle(nb_vehicle);
current_sol;
initialize pheromone using nb_vehicle;
N_i^k ← feasible nodes ;
while iterations < max_iteration do
    foreach ant k do /* construct a solution Ψ^k                    */
    run_ant(nb_vehicle,IN);
    ∀ customer j ∉ Ψ^k : IN[j] ← IN[j] + 1;
    ;
    if ∃k : visited_customer(Psi^k) > visited_customer(current_sol) then
        current_sol ← Ψ^k;
        ∀j : IN_j ← 0;
        if current_sol is feasible then
            best_feasible_sol ← current_sol;
            min_vehible(nb_vehicle-1);
        end
        pheromone_update;
    end
end
```

Algorithm 4: Algorithm used to minimize the number of vehicle in a VRPTW

Time minimization The second step aims to minimize the travel time of a solution using the number of vehicles found with the vehicle minimization. The algorithm 5 is the one used to achieve that goal. The algorithm is very similar to the one used in the vehicle minimization except that in this case each feasible solution built by the ants are compared with regards to the time. A solution replaces the best one kept in memory only if it takes less time.

In the vehicle minimization and time minimization, an `run_ant(nb_vehicles, IN)` method is used. Algorithm 6 presents that method. From the CVRP implementation, the first difference that stands out is the arguments that are used: `nb_vehicle` and `IN`. In MACS, as explained earlier, we try to minimize the number of vehicle used. To do that, the ant that builds a solution has to know the number of tours it can make, this is why we use the `nb_vehicle` parameter. `IN` is an array that has an entry for each cities of the problem. This vector is used in the `select_next_node` method where the proportional rule presented in equation 2.4 is used. Indeed, the attractiveness η_{ij} used in that equation is not anymore given by $1/d_{ij}$. The

$best\_feasible\_solution \leftarrow first\_sol$;
def min_time(nb_vehicle);
$current\_sol$;
initialize pheromone using nb_vehicle;
$N_i^k \leftarrow$ feasible nodes ;
while *iterations* $<$ *max_iteration* **do**
  foreach *ant k* **do** `/* construct a solution` Ψ^k `*/`
  run_ant(nb_vehicle,IN);
  if $\exists k : Psi^k$ *is feasible &&* $Time_{\Psi^k} < Time_{\Psi^{best\_feasible\_solution}}$ **then**
    $best\_feasible\_sol \leftarrow \Psi^k$;
  end
  pheromone_update;
  ;
end

Algorithm 5: Algorithm used to minimize the travel time in a VRPTW

computation of η_{ij} is done as follows:

$$
\begin{aligned}
delivery\_time_j &\leftarrow max(current\_time_k + t_{ij}, a_j) \\
delta\_time_{ij} &\leftarrow delivery\_time - current\_time_k \\
distance_{ij} &\leftarrow delta\_time_{ij} * (b_j - current\_time_k) \\
distance_{ij} &\leftarrow max(1.0, (distance_{ij} - IN_j)) \\
\eta_{ij} &\leftarrow 1/distance_{ij}
\end{aligned}
$$

The delivery_time is computed as the maximum value between the current time of the ant k plus the time it takes to go from client i (the client where the ant k is) to client j and the beginning of the time window associated to client j. This comparison is done because the client can not be delivered to before the beginning of its time window. The delta_time is computed as the difference of the delivery time and the current time of the ant k. The distance is then computed as the product of the delta_time and the end of the time windows of client j minus the current time of the ant k. The IN vector is taken into account to diminish the distance value. Every time a city j is not visited in a path constructed by the ants, the value $IN[j]$ is incremented. So the distance is virtually lowered to permit the cities that are not often in a path to have a higher probability of being chosen. Finally η_{ij} is computed.

def run_ant(nb_vehicle, IN): ;
put ant k at depot i;
$\Psi^k \leftarrow i$;
$current\_time_k \leftarrow 0$, $load_k \leftarrow 0$;
$N_i^k \leftarrow$ feasible nodes ;
while $N_i^k != empty$ **do**
  $\forall j \in N_i^k$ compute attractiveness η_{ij};
  select_next_node(IN);
  //Choose probabilistically the next node j using η_{ij} $\Psi^k \leftarrow \Psi^k + j$;
  $current\_time_k \leftarrow delivery_time_j$;
  $load_k \leftarrow load_k + q_j$;
  if *j is depot* **then**
    $current\_time_k \leftarrow 0$, $load_k \leftarrow 0$;
    $nb_vehicle - -$;
  end
  local pheromone update ;
  $i \leftarrow j$;
  $N_i^k \leftarrow$ feasible nodes ;
end

Algorithm 6: Algorithm used by the ant to build a solution in MACS

Chapter 3

An ACO Framework

This chapter concerns the construction of the ACO framework. We will first discover how the framework is built and why. Then we will discover the choices of implementation and why they have been made. Finally, a complete explanation on how to use the framework is given with some examples that are used in chapter 4 for the tests.

3.1 Analysis of the Implementations

There are many similarities and differences that can be identified in the different implementations discovered in chapter 2. In this section, these common characteristics as well as the differences will be explained and developed. These observations will help to understand how the framework is built.

3.1.1 Common characteristics

A first point that is common to the different implementations of chapter 2 is the use of the imput file. Each time, an input file containing information about the different cities is used. This information relates to the coordinates of the cities, the demand of a client in a CVRP case or the time windows for a VRPTW. This information can be easily parsed if it is normalized. **The reading of the input file** is thus considered as a first point that will be implemented in the framework.

Another common point is the way in which the **constraints that concern each client** are handled. As an example, for the CVRP, the demands of all clients are obviously associated with each client. The same goes for the Time windows in the VRPTW. Every time a constraint that concerns each client is added, that constraint is linked to the client. This link can be done in a generic way.

Another common point is the way of **representing the problem**. As explained in chapter 1, the problem representation for the TSP, CVRP and VRPTW is based on a graph. In all cases, the graph contains information about the cost (in time or length) to go from one city to another. Also, as explained in section 1.1, the ants uses stygmergy to communicate. **The**

representation of the pheromones is also made using a graph. Indeed the pheromones are dropped on the arcs between the different cities.

Last but not least, the **construction algorithm** is very similar from one implementation to another. The core method used by the ant to build a path and the method that coordinates all the ants are the same in the application of the CVRP and the TSP. For the MACS-VRPTW, that is a more complicated implementation (see section 2.3), these methods are slightly different but there are still some similarities that are exploited in the framework.

3.1.2 Differences

In the implementation developed in sections 2.1, 2.2 and 2.3, some differences have been pointed out. For example, when the capacity constraints are added with the CVRP, some aspects have to be modified from the TSP to have a working algorithm.

First, the capacity constraints that are added require that an ant keeps in memory her current load and the maximum load she can get. To do this, we need to create new variables for the ant.

Then, the method to choose the next node must also be modified because of the capacity constraint. Indeed, an ant can not go to every city j if she is at city i. Some cities will lead to a violation of the capacity constraint and thus can not be chosen.

Next, the way the pheromones are updated on a path is different from one problem to another. This also leads to different implementations.

Finally, all the methods that permit to initialize or reset a path are slightly different in all the implementations because of the points listed above.

3.2 Architecture of the Framework

From section 3.1, we can see that two big elements change with the problem and the constraints of the problem. This is the behavior of an ant and the synchronization between them via the pheromones. This is why two base classes that will handle these two behaviors are created. These two base classes will be extended by a user of the framework so that it has the desired behavior. The rest of the methods and information useful for the ACO application that we presented in section 3.1.1 will be held in four different other classes that do not need to be modified, whatever constraints or changes are made to a VRP. In the figure 3.1, is the architecture of the framework described with an UML diagram. First, the four classes that will be constants in all the different possible implementations using the ACO framework are described: `VehicleRouting`, `FileReader`, `Graph` and `Client`. Then, the two classes that need to be extended by a user of the framework are described: `Ant` and `Colony`.

3.2.1 VehicleRouting object

The figure 3.2 is an UML representation of that object. This object is the "main" object of the framework. It contains a method `launch(file:String,...)` that has for purpose of coordinating all the work done by the other classes. It uses the `FileReader` object methods to read the input file `file`. Then it transforms the data read into a matrix that will be used to

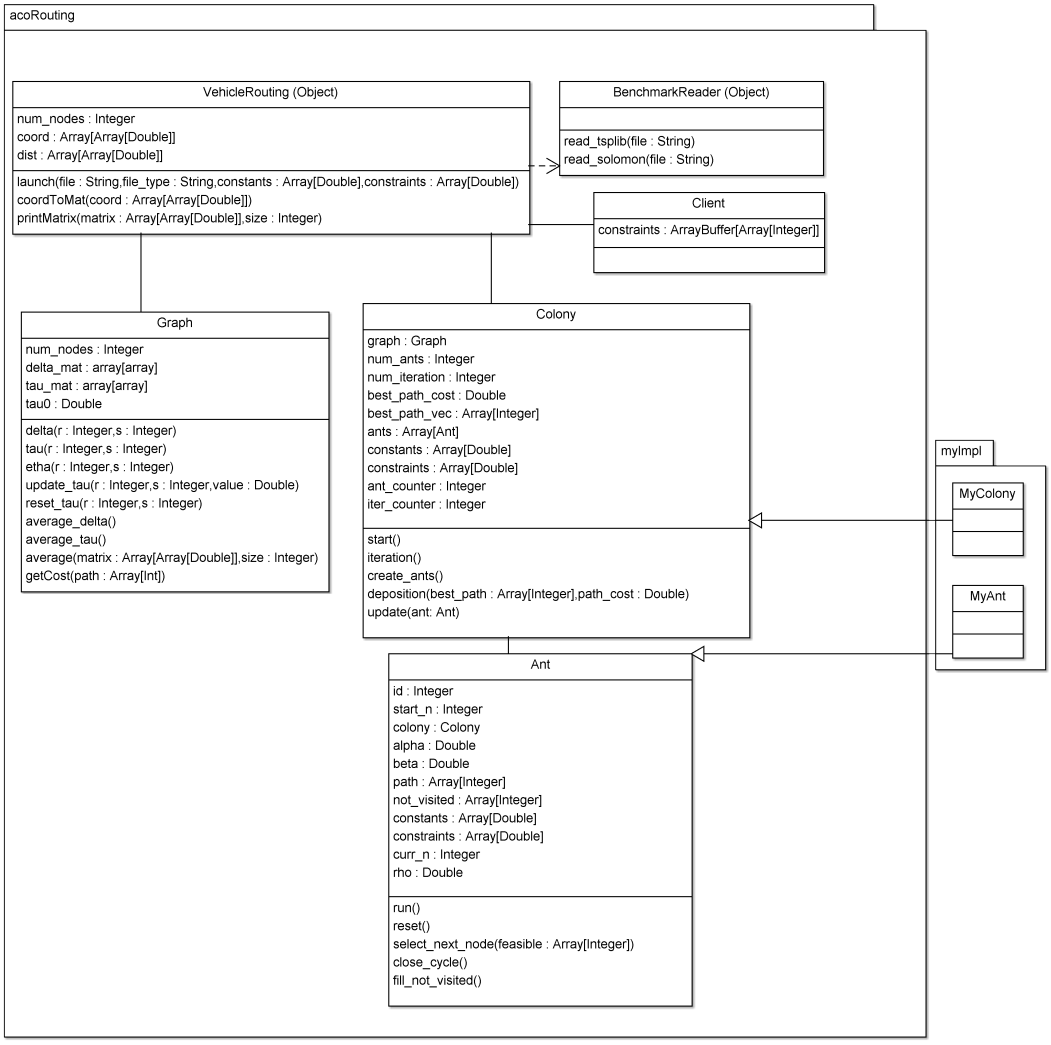


Figure 3.1: UML diagram of the Framework developed.

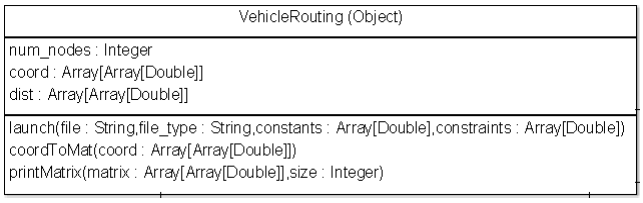


Figure 3.2: UML diagram of the VehicleRouting object.

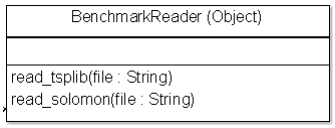


Figure 3.3: UML diagram of the BenchmarkReader object.

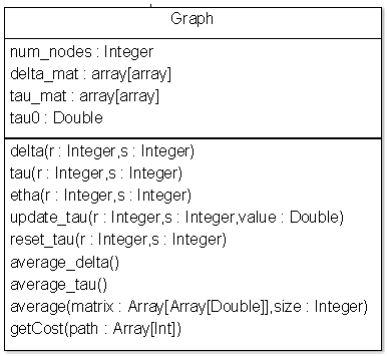


Figure 3.4: UML diagram of the Graph class.

create an instance of the Graph class. Finally it creates an instance of the `MyColony` class that will be used to find a solution to the problem. The method `coordToMat()` is used to transform the information contained in the input file that are cartesian coordinates into a distance matrix. A distance matrix is a square matrix M that contains the distance between the city i and j at M_{ij}. The parameters of the launch method are used to fix the number of iterations, the number of ants and other constants that are used in the program.

3.2.2 BenchmarkReader object

`BenchmarkReader` is an object that contains two methods `read_tsplib(file)` and `read_solomon(file)`. Each method can read a TSPLib file or a SolomonLib file respectively. The way a TSPLib file or SolomonLib file should be encoded is explained in section 3.4. The figure 3.3 is the representation of that object.

3.2.3 Graph class

Graph is a class containing all the information about the graph that represents the problem. This information is stored into the `delta_mat` matrix. A Graph object is created by the VehicleRouting object after the input file has been read. The graph class also contains information about the pheromones because they are held in a similar matrix, `tau_mat`. The methods defined in the Graph class are used to access and modify these matrices.

3.2.4 Client object

Client is a data object containing an ArrayBuffer `constraints` whose goal is to keep all the clients demands and constraints in that single array. The UML representation is in figure 3.5.

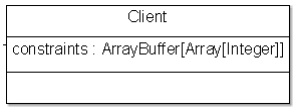


Figure 3.5: UML diagram of the Client object.

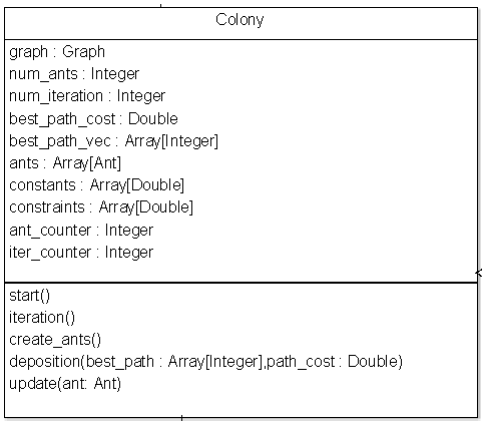


Figure 3.6: UML diagram of the Colony class.

3.2.5 Colony class

The class Colony is an important class because it is the class that coordinates all of the ants that are used to solve a problem. The method `start()` creates all the ants that populate the colony (each ant is an instance of the `MyAnt class`). Then a loop that runs the ants is done until we reach a maximum fixed number of iterations. In that loop the method `iteration()` is called. This method runs each ant created using the `run()` method of these ants. When all ants have built their path, their results are compared and pheromones are dropped using the `deposition()` method. Finally, the best path is given to the `VehicleRouting` object that will print it.

3.2.6 Ant class

The Ant class is also an important one because it is the class that controls the behavior of the ant. The core method of this class is the `run()` method. This method aims to make a path that is a solution to the problem. It will visit the cities one after another until there is no more city to visit. The choice of the next city to visit is made using the `select_next_node()` method.

3.2.7 MyColony - MyAnt

These classes are intended to the user of the framework. These classes inherit from the `Colony` and `Ant` class respectively. The explanations about the way of completing these classes will be given in the next section (section 3.4).

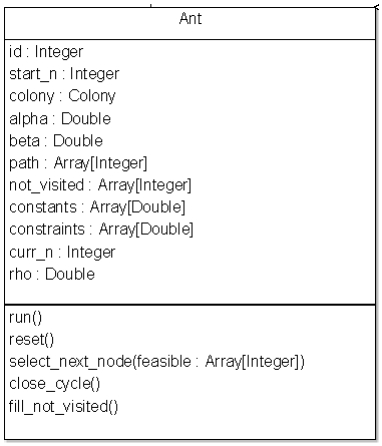


Figure 3.7: UML diagram of the Ant class.

3.3 Implementation Choices

In these sections the choices made for the implementation of the framework are explained. This covers the language used, the representation of the graph, the representation of the clients and the constraints associated to them, the parameters given to the different methods and the base case of the Framework.

3.3.1 Language used

The Scala language was carefully chosen. Scala is high level and a mixed paradigm language, it uses pure object-oriented and functional programming. The fact that it is object oriented allows for the creation of a framework that will be extended by inheritance by the user.

3.3.2 Representation of the graph

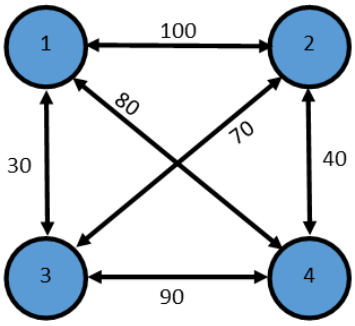
In routing problems, we always work with weighted graphs, graphs that have values attached to the edges. These values represent the cost of traveling from one node to the next. When we work with ACO, each ant has to access these values quite frequently. The weighted graph is thus represented as a weighted adjacency matrix. This choice permits to have access to the different value with a $O(1)$ complexity.

Let v_i and v_j be numbered vertices for $1 \leq i, j \leq n$. Let w_{ij} be the weight of an edge e and assume that all edge weights are positive ($w_{ij} \geq 0$). Let $M[i, j]$ be an adjacency matrix where $M[i, j] = w_{ij}$ for each edge $e = (v_i, v_j)$. For each vertex i in the graph, $M[i, i] = 0$.

Figure 3.8 shows an example of a graph with its matrix representation.

3.3.3 Client constraints

To represent the clients and the constraints associated to them, there are two choices: creating a Client class where each client is an instance of that class or creating a single Client object



(a) A simple weighted Graph

0	100	30	80
100	0	70	40
30	70	0	90
80	40	90	0

(b) Weighted adjacency matrix

Figure 3.8: A simple graph with its matrix representation

constraints		
0	1	2
Q_0	a_0	b_0
Q_1	a_1	b_1
...	...	...
Q_n	a_n	b_n

Figure 3.9: An example of the constraints arraybuffer

that contains all the information about all the clients of the problem in a single array. Because of the input file that presents the constraints in a sort of array, and because of the fact that an element in an array is accessed in $O(1)$, the second option is chosen.

In the framework, a client Object is used, this object contains an ArrayBuffer named `constraints`. This ArrayBuffer contains an array for each constraints on the clients. For example, with a VRPTW of size n (i.e.: with $n-1$ clients and 1 depot) the arraybuffer `constraints` is represented in figure 3.9. Q_i represents the demand of client i. a_i and b_i represent respectively the begining and the end of the time window of the client i. If another constraint is added, a new array is added to the ArrayBuffer.

3.3.4 Parameters of the launch method

As explained in section 3.2.1, the launch method has different parameters. The first is the location of the input file `file`. Then a string `file_type` that says wich type of file is given in parameter (i.e.: Solomon or TSPLIB). The third and fourth are two arrays named `constants` and `constraints`. These arrays, as their names indicates, will contain the values of the constants used by the programs and the values associated to the constraints.

The choice of an array to keep these values is made because it is flexible. If the user of the framework needs to use constants, he just has to pass an array that defines these constants. The same goes for the constraints. These two arrays are passed to the Colony and Ant instances. This also permits to the user to make a program that takes these arrays as parameters, so it is easy to manipulate and modify these constants.

# Ants	# Iterations	α	β	...

Table 3.1: Representation of the `constants` array.

The `constants` array has a fixed minimum size. Some values, such as the number of ants to be used and the maximum number of iterations, must be the first two in the array. See figure 3.1 for an example of the `constants` array.

The `constraints` array is an array that is used by the ants to fix the limits that are given on the constraints. For example, in a CVRP, the ants (i.e. the trucks) have a maximum capacity limit. This limit is given in the constraints tabular. If other constraints of the same type must be added, this array is extended.

3.3.5 Colony and Ant class

The ants that build solutions share common information, the pheromone matrix. This matrix is modified by the ants when they use an arc, and also when they all have completed a tour. This is where the colony class is important. The colony class collects the information about all the tours that the ants built, compares them and updates the pheromones using the best solution built by the ants. These two classes are also the base classes that will be extended by an user of the framework. The question is what contains these classes? To answer that, we can make a first observation : The CVRP and the VRPTW are two generalizations of the TSP. Indeed, a TSP is a CVRP that has one vehicle with no capacity limit and customer with no demand. The same goes for the VRPTW, a TSP is a VRPTW with one vehicle, no capacity limit, no time limit and customer with no demand and infinite time-windows. This observation leads to the implementation of the base problem (the TSP) in the base classes. This permits a user to directly use the framework without coding anything on a TSP problem. In these classes can be found the methods that apply the base algorithm of an ACO system. These methods should not (or rarely) be modified. This is the `start()` and `iteration()` methods of the colony that are presented in the code block 3.1 and the `run()` method of the ant presented in the code block 3.2.

Colony

```
def start() {
  this.create_ants
  while (!this.done) {
    this.iteration()
  }
}

def iteration() {
  this.ant_counter = 0
  this.iter_counter += 1
  for (i <- 0 to num_ants - 1) {
```

```
    ants(i).run
  }
  this.deposition(this.best_path_vec, this.best_path_cost)
  if (this.best_path_cost < VehicleRouting.best_cost){
    VehicleRouting.best_tour = this.best_path_vec
    VehicleRouting.best_cost = this.best_path_cost
  }
  this.best_path_vec = Array(0)
  this.best_path_cost = Double.MaxValue
}
```

Code 3.1: Source code of the start and iteration method in Colony class

This fraction of code is a scala implementation of the ACO algorithm given in algorithm 1. The Colony starts by creating the artificial ants. Then each ant is launched and does its work. When all the ants have made their tour, the best tour is used to drop pheromone with the `this.deposition()` method. If the best path found during this iteration is better than the best path found at the moment, it is replaced.

Ant

```
def run() {
  var new_node = 0
  while (this.not_visited.length != 0) {
    new_node = this.select_next_node(this.not_visited, null)
    this.path_cost += graph.delta(this.curr_n, new_node)
    this.path = this.path :+ new_node
    (this.graph.update_tau(curr_n, new_node, (1 - this.rho)
                * this.graph.tau(curr_n, new_node)
                       + (this.rho * this.graph.tau0)))
    this.curr_n = new_node
  }
  close_cycle()


  this.colony.update(this)
  this.reset
}
```

Code 3.2: Source code of the run method in Ant class

The fraction of code presented in code block 3.2 is the core method of the Ant. The `run()` method goal is to build a path that is a solution to the problem. The method iterates while the `not_visited()` array is not empty. At each iteration:

- a new node is selected
- the node is added to the path

- a local pheromone update is done according to equation 2.3.
- the current node is replaced by the new selected node.

When the iteration is done, the cycle is closed (the salesman goes back to his start point) and the results are sent to the Colony.

3.4 How to use the framework

Now that we have a clear idea of the design of the framework, we will discover the convention governing its proper use. To do that, the correct way to use the framework to solve a CVRP problem will be elaborated. Some aspects change from the TSP base implementation of the framework. A capacity constraint is added. The Input file, the Client class and obviously the Colony and Ant classes change. The modifications are explained hereafter.

3.4.1 Input File

The input file describing the problem must follow the convention of the TSPLIB file or the convention of the Solomon benchmark.

TSPLIB

The TSPLIB input file contains two part : a *specification part* containing information on the file format and its content, and a *data part* containing explicit data as the distances between cities. [15]

The specification part contains :

- the *name* of the file.
- the *type* of problem it describes (TSP, CVRP,...).
- the *dimension* of the problem (Total number of nodes and depot in the case of a CVRP).
- the *capacity* that specifies the truck capacity in a CVRP.
- the other constraints that can be added and their limitation on a truck.
- the *edge weight type* that specifies how the distances are given.

The Data part contains :

- the *node_coord_section* where node coordinates are given in the form:

  `<integer> <real> <real>`

- the *depot_section* that contains the list of depot nodes (always node 0 in our case)

Client_Number	X_coordinate	Y_coordinate	Demand	Ready_Time	Due_Date	Service_Time	...

Table 3.2: A row of a Solomon benchmark file

- the *demand_section* that contains the demands of all nodes of a CVRP. It is given in the form:

  `<integer><integer>`

- the other constraints per node.

Solomon Benchmark file

The Solomon benchmark takes the form of a matrix. Each row contains all the information about a node. Table 3.2 demonstrates the contents of the rows of that matrix.

More columns can be added if there are more constraints to represent. [16]

3.4.2 Constraints

The CVRP is a TSP augmented with capacity constraints on the vehicle and demand of the client. In this section, we discover how to adapt the framework to solve the CVRP problem.

Modelization of the constraints

As outlined above, the CVRP adds one constraint to the TSP. It is the constraint bound to the capacity of the vehicle. The first change to make is of course in the input file. As explained before, a new set of data is added for each client: the demand. The next changes we have to do are in the implementation of the Colony and the Ant. To do that, the class `MyAnt` and `MyColony` that **extends** the `Ant` and `Colony` class have to be implemented.

The MyColony class does not need many new methods. As the Ants do not start from a random city like in the TSP, but rather from the depot, the method `create_ants()` is modified. The code of the `MyColony` class is presented in code block : 3.3.

```
class MyColony(graphp: Graph, num_antsp: Int, num_iterationp: Int)
      extends Colony(graphp, num_antsp, num_iterationp) {

  override def create_ants() {
    var startcity = 0
    this.ants = Array.empty[MyAnt]
    for (i <- 0 to (num_ants - 1)) {
        startcity = 0
    var ant = new MyAnt(i, startcity, this, VehicleRouting.capacity)
        ants = ants :+ ant
    }
    ants
  }
```

```
}
```

Code 3.3: MyColony implementation for a CVRP

The MyAnt class needs more modifications. The capacity constraint has to be taken into account when selecting the next node. To do that, different methods are possible. The simplest is to follow the selection process as for the TSP and send the truck back to the depot if the selected node violates the constraints. In that case, only the `select_next_node()` method must be overridden. Below is the new implementation of the `select_next_node()` method.

```
override def select_next_node(feasable: Array[Int]): Int = {
    val graph = this.graph
    var new_node = -1
    var denominator = 0.0

    if (feasable(0) == 0) {
      return 0
    }
    if (!feasable.isEmpty) {
      denominator = compute_denominator(feasable)
      new_node = choose_new_node(denominator, feasable)
      if (new_node == -1) {
        new_node = this.curr_n
      }
      if (this.capacity < Client.constraints(0)(new_node) + this.load) {
        new_node = 0
        this.load = 0
      } else {
        this.load += Client.constraints(0)(new_node)
        this.not_visited = this.not_visited.filterNot(_ == new_node)
      }
      return new_node
    } else {
      new_node = 0
      this.load = 0
      return new_node
    }
  }
```

Code 3.4: MyAnt implementation for a CVRP

3.4.3 Adding new Constraints

In this section, through two other examples, it is shown how to add other constraints to a CVRP. The two examples that are described in this section will be tested in chapter 4.

The first example concerns the adding of a constraint to a CVRP that is verified after a tour is built. In other words, an ant constructs a valid path according to the capacity constraint.

This path must then be validated to be sure it does not violate other constraints. This type of constraint is used in some vehicle routing problems with black box feasibility (3D loading ...) [17]. This type of constraint is called "off-parcour constraint".

The second example concerns the adding of a soft constraint that is used in the construction of a tour. We talk about a soft constraint because the constraint will be used in the evaluation of the quality of a solution provided (i.e. in the objective function). A soft constraint can not be violated but must be taken into account in the building of a solution to have a correct solution. This type of constraint is called "in-parcour constraint".

The third example is a hard "in-parcour constraint". What is called a hard constraint is, as the capacity of a CVRP, a constraint that can not be violated.

Off-parcour constraint

In this section is explained the adding of an off-parcour constraint. In this example, a simple rule is implemented but the case can be generalized to any more complex rules. In this example, the rule forbids two cities that have consecutive numbers to be visited one after the other. In other words, a city i can not be visited after city $i-1$ or city $i+1$.

To implement this rule, from the CVRP implementation given above, we only need to modify the `MyColony`. Indeed, the `MyAnt` class does not need modifications because the constraint that we add can not be verified in the path construction but only when the path is finished.

When an ant has finished building a path, the colony has to test whether or not the path is the shortest found in this iteration and whether or not the path is valid according to the rule we defined. This is done with both methods `update(ant:Ant)` and `is_valid(ant:Ant)`. The code of these functions is given in code block 3.5.

```
override def update(ant:Ant){
   this.ant_counter += 1
   if(is_valid(ant)){
     if (ant.path_cost < this.best_path_cost) {
       this.best_path_cost = ant.path_cost
       this.best_path_vec = ant.path
     }
   }
 }
 def is_valid(ant:Ant):Boolean={
   var last=0
   for(c <- ant.path ){
     if (last != 0){
       if(last+1 == c || last-1 == c){
         return false
       }
     }
     last=c
   }
   return true
 }
```

Code 3.5: Method used for the off-path constraints

The method `update(ant:Ant)` is called every time an ant has build a path. The update method checks first if the path is valid using the `is_valid(ant:Ant)` then keeps the solution of the ant if it is the best of the iteration. If the path is not valid, it is discarded. In this case the method `is_valid(ant:Ant)` is quite simple but it can be replaced by something different easily.

In-parcour constraint

The soft constraint that is implemented is a preference for the client to be placed first in a subtour. Each client has a value assigned to him that represents how important it is to be the first in a subtour. The value of that preference is an integer in the range 0 through 10, 10 symbolizing a great need to be first and 0 symbolizing the fact that a client does not want to be first.

To implement that constraint, the first thing to do is to add the preference of each client in the input file. Once this is done, the `MyAnt` class must be modified to take the constraint into account. A different random proportional rule will be used when the ant is at the depot (that is the city $i = 0$). Instead of using the classical one presented in equation 2.4, when at city $i = 0$ a new factor is added to the equation to take into account the preferences of each client. The new rule that is used is :

$$p_{0j}^k = \begin{cases} \frac{[\tau_{0j}]^\alpha [\eta_{0j}]^\beta [\xi_j]^\epsilon}{\sum_{l \in N_i^k} [\tau_{0l}]^\alpha [\eta_{0l}]^\beta [\xi_l]^\epsilon} & \text{if } j \in N_i^k \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

where ξ_j is the value associated with client j and ϵ is a constant as α and β that affects the importance of ξ in the building of a solution. As ϵ is a constant, the value of that constant is added into the array `constants` presented in section 3.3.4.

The second thing to change is the objective function that measures the quality of a tour in the `MyColony` class. To do that, instead of taking only the length of the path to give a cost to a solution, we add to that cost a value related to the preference of each first client in a subtour (see code block 3.6).

```
override def update(ant: Ant) {
  this.ant_counter += 1
  var first_in_tour = new ArrayBuffer[Int]
  var cost = ant.path_cost
  for (i <- 0 to ant.path.length-1){
    if(ant.path(i)==0){
      first_in_tour += ant.path(i+1)
    }
  }
  for (a<-first_in_tour){
    cost += ((math.abs(Client.constraints(1)(a)-10))*100)
```

```
    }
    if (cost < this.best_path_cost) {
      this.best_path_cost = cost
      this.best_path_vec = ant.path
    }
  }
```

Code 3.6: Taking the preference into account

Again this example is a simple one. The objective function and the proportional rule used are maybe not the best in that case, but the goal of this example is just to show how to add constraints and what it implies in the implementation.

The hard constraint that is implemented here is a limit on a sub-path of a CVRP solution. As a reminder, a CVRP solution is a set of paths where each path begins and ends at the depot and visits some cities. To the classic CVRP that limits only the capacity of a vehicle, we add here a limit on the maximum length of a subpath. This limit can represent, for example, the size of the fuel tank of a truck. Adding that kind of constraint is done by modifying the ant behaviour. This is done in the implementation of the `MyAnt` class by overriding the `select_next_node(feasable:Array[Int])` method. In the implementation of the CVRP given above, we force the ant to go back to the depot if the next node selected in the path violate the constraint capacity. To take the new constraint into account, we simply need to calculate and keep in memory the length of the subtour, and send the truck back to the depot if the constraint is violated (if the subtour is too long). In this example the `constraints` array that is defined as an argument in the `launch` method of the VehicleRouting object (see section 3.3.4) will contain a new value that will represent the maximum length of a tour.

Chapter 4

Experimental Results

This chapter presents the tests made on the framework and their results. First, the TSP is tested, which is the base case of the framework. Next, a classical CVRP implementation and a VRPTW implementation are tested. After that, an example of CVRP with new constraints relative to the client is tested, and finally, an example with path constraint is tested. These tests have two goals, the first is to compare the efficiency of the ACO implementation, the second is to test the framework by giving tests examples on extended CVRP (CVRP with new constraints). To achieve that, the tests are run with a configuration that is a good compromise between the computation time and the solution quality. The configuration is the value that is given to different constants like the number of ants, iterations ,α, β ... The values given have been found in the literature and by testing.

All the tests are made on a personal computer with Intel Core i7-3610 CPU that runs at 2.30 GHz.

4.1 The Traveling Salesman Problem

These first series of tests permit to determine the efficiency of the base implementation of the framework. The tests are made with problems of different sizes that we can find in TSPLIB [15]. Four different tests of different size are made. The TSP problems that are used are :

eil51 A TSP with 51 cities by Christofides

berlin52 A TSP that represents 52 locations in Berlin.

eil76 A TSP with 76 cities by Christofides

These instances of TSP are chosen over the others from the TSPLIB because they have been tested with other ACO implementations and thus we can compare similar results together. Each instance of the TSP problem is tested a hundred times. In table 4.1, the results of the tests are listed. In table 4.2, the results obtained by two other implementations [13] [18] are presented. In the implementation [13] 1250 iterations and a number N of ants that is equal to the number of cities in the problem are used. No similar information has been found for the implementation [18].

problem	#iterations	#ants	α	β	mean	best
eil51	500	51	5.0	5.0	451.48	430.34
berlin52	500	52	5.0	5.0	7730.00	7544.36
eil76	500	76	5.0	5.0	592.27	567.96

Table 4.1: results of the tests for the four TSP instances

Implementation	problem	best
ACS [13]	eil51	427.96
	eil76	542.37
ACO for TSP [18]	eil51	– (426)
	berlin52	7544.36 (7542)
	eil76	– (538)

Table 4.2: results of other ACO implementation for the TSP

As a reminder, here is the signification of the names of the columns:

#iterations denotes the number of iterations made.

#ants denotes the number of ants used.

α is a parameter influencing the use of the pheromone

β is a parameter influencing the importance of the heuristic function.

mean is the mean value of the path length of the 100 trials.

best is the best value of the path length of the 100 trials.

The values that are in brackets are rounded values. They are calculated with rounded distances.

The results of the TSP implementation are quite good. Compared with the ACS algorithm developed by M.Dorigo and L.M.Gambardella, the results are almost the same. The same goes for the implementation of Hlaing and Khine.

Tests are not made on TSP instances with more cities because, in order to have good results, new functions have to be made to improve the results. For example, a local search method or a structure known as candidate list must be implemented. A candidate list is a preselection of nodes that can be elected at each step by an ant. With the framework, if these functions are encoded, it can be easily added by modifying the class `MyAnt`.

4.2 The Capacitated Vehicle Routing Problem

As for the TSP, the instances of CVRP that are tested are choosen so that they can be compared to other ACO implementations. The instances that are used come from the problem described by Christofides, Mingozzi and Toth. In particular their problem:

C1 A CVRP with 51 cities and a truck capacity of 160.

C3 A CVRP with 101 cities and a truck capacity of 200.

C4 A CVRP with 151 cities and a truck capacity of 200.

The results of the framework are compared with the results of J.Bell and P.McMullen [19] and Y.Bin, Y.Zhong-Zhen and Y.Baozhen [20]. These two tests are run with a number of 5000 iterations and a fixed number of 30 ants. In table 4.3 are listed the results obtained with the ACO framework and in table 4.4 the results of the scientists given above.

problem	#iteration	#ants	α	β	mean	best
C1	500	51	5.0	5.0	587.23	556.75
C3	500	100	5.0	5.0	983.26	942.38
C4	500	100	5.0	5.0	1272.34	1269.65

Table 4.3: results of the tests for the CVRP

implementation	problem	mean	best
[19]	C1	528.90	524.80
	C3	865.65	854.20
	C4	1143.43	1131.83
[20]	C1	524.61	524.61
	C3	844.32	830.00
	C4	1042.52	1028.42

Table 4.4: results of other ACO implementation of CVRP

The solutions provided by the framework are within 10% of the optimum of other ACO algorithms. This can be explained by several factors. First, the number of iterations used is much greater in the implementations [19] [20] where they set the number of iterations to 5000 than here in the tests where we set the number of iterations to 500. Next, these implementations have special local search algorithms and use candidate lists to improve the results, the framework implementation tested here does not (but again, if the functions are encoded, the framework can easily be extended). Finally, the test with 151 cities is already quite large for a basic implementation thus it is normal to observe bad results.

4.3 Vehicle Routing Problem with Time Windows

The VRPTW benchmark tested are instances of the Solomon benchmark [16]. The implementation made in the framework is a simplified version of the MACS-VRPTW [7]. The results obtained are compared to the original MACS-VRPTW version. In the simplified version, we omit the local search function and an insertion function that are implemented in the MACS-VRPTW. The results obtained with the framework are presented in table 4.5, the ones obtained

with the MACS implementation are presented in table 4.6. Values in brackets represent the number of vehicles used. The number of iterations used by [7] is fixed with a computational time limit. As it is implemented in C++ and uses multi-threading at its maximum, the comparaison is hard to make.

problem	#iterations	#ants	α	β	mean	best
R101.25	500	25	5.0	5.0	889.51 (10)	802.95 (10)
R101.50	500	50	5.0	5.0	1567.08 (18)	1446.07 (18)

Table 4.5: results of the tests for the VRPTW

implementation	problem	best
MACS-VRPTW	R101.25	617.1 (8)
	R101.50	1044.0 (12)

Table 4.6: results of the MACS implementation of VRPTW

As can be seen, the simplified implementation of the framework has bad results. This can be explained by the fact that the insertion function plays an important role in the construction of solutions that minimize the number of vehicles used. That function was not clearly described in the paper of Gambardella, but again, if we have that function, it can be easily added to the framework and thus improve the results.

4.4 CVRP with in-path constraints

This section will present two CVRP problems with new constraints added in the construction of a solution. These constraints are called "in-path" constraints because they have an effect on the construction of the solution by the ant. This type of constraint is opposed to the "off-path" type where a path is validated or not based on the finished path.

The first test simulates a constraint on the truck. The trucks used for the vehicle problem have a limited fuel capacity. Thus a sub-path can not have a length greater than a certain value $tour\_length$. The adding of that constraint is done as explained in section 3.4.

The second test simulates a preference of the clients who want to be first in a tour. Each client is assigned a value $pref$ from 1 to 10. Clients with high value are clients that need to be first in a tour and clients with low values are clients that do not want to be first in the tour. Both tests are performed on the C1 instance of the CVRP presented in section 4.2. The results of these tests are listed in the table 4.7.

Limited subpath For the test, the maximum subpath is set to 50. In the best case there are eleven vehicles. The best result of the problem C1 given in section 4.2 had 10 vehicles and a total length of 556. Here with a limitation of the subtours of 50, with a simple calculation, it points out that at least 11 vehicles are needed. The length of the complete path is almost doubled. This can be explained by the fact that when new vehicle are used, there are more trips

problem	#iterations	#ants	α	β	mean	best
limited sub-path	500	50	5.0	5.0	919.66	881.37 (18)
First client pref.	500	50	5.0	5.0	710.14	614.00

Table 4.7: results of the CVRP implementation with in-path constraints

best path	preference
0	
46	6
47	
...	
0	
48	7
8	
...	
0	
17	10
37	
...	
0	
4	10
44	
...	

Table 4.8: results of the CVRP implementation with in-path constraints

between the depot and the cities. An analysis of the different paths given by the 100 executions of the problem reveals that all the constraints are always verified as expected.

Client preference The constraint has been implemented in a soft way. The goal here is to satisfy the most clients but it is not always possible. The objective function that gives a score to a solution has thus been modified (see section 3.4.3). The score takes into account the total distance (as in the usual CVRP) and the respect of the preference. The best path and the preference associated with client of interest are given in table 4.8 (as a reminder 0 represent the depot). As can be seen in that table, the first clients of each tour have high preferences. The other nodes and preferences of each subtour are not reported as the preference is only taken into account for the first client visited after a depot.

This test shows that it is possible to add a constraint on the path in an easy way. It also shows that the constraint tends to be respected. The path is different from the classic CVRP presented earlier, and the constraint added modifies the path. In table 4.8, we can see that the first cities of a tour have a high preference as expected.

problem	#iterations	#ants	Q_0	α	β	mean	best
off-parcour	500	50	0.9	5.0	5.0	634.09	601.55

Table 4.9: results of the CVRP implementation with off-path constraints

4.5 CVRP with off-parcour constraints

The "off-parcour" constraints are constraints that can only be verified after the construction of a tour. As for example in a VRP with 3D-loading constraints where some rules must be verified to validate a path. Here a simple rule is implemented but this simple rule can be replaced by a more complex one. The rule forbids two cities having consecutive numbers to be visited one after the other. In other words, a city i can not be visited after city $i-1$ or city $i+1$. A simple function verifies a path and indicates if it is valid or if it must be discarded. The result of that implementation on the C1 CVRP instance is given in table 4.9. An analysis on the path given after the execution of the test proves that the constraint is verified. The best result is different of the normal C1 instance because, in the best solution of that problem, there are several consecutive nodes.

Interestingly, the problem has been modified with a new constraint and so the result slightly differs from the original instance of the CVRP. It shows again that the constraint added have an effect on the results produced and are respected.

Chapter 5

Conclusion

The goal of this master thesis was to ease the use of Ant Colony Optimization in routing problems by creating an implementation with a high level of abstraction. After the analysis of existing ACO implementations for routing problems, it was found that implementing a framework was the most natural way to proceed. Because of the particularity of ACO, a high level of abstraction is hard to reach.

In chapter 2, an analysis of the implementations of ACO for different routing problems (TSP, CVRP an VRPTW) shows that even if there is a lot of differences in the implementations, the principle of the algorithm is always the same. The way to solve the problems are very similar from a high-level perspective, but they differ significantly in implementation details. Some ants build solutions and these solutions are compared and used to update the pheromone trails. This is the principle that is used in chapter 3 to build the framework. An `Ant` class is build to describe the behavior of the ants and a `Colony` class has the purpose of coordinating these ants. The inherintance capacity of Scala is used on these classes to change the implementation details that change from one problem to another.

As presented in chapter 3, when very specific implementations (as the MACS for the VRPTW) are made, the `MyAnt` and `MyColony` classes override almost everything from the parent classes `Ant` and `Colony`. On the other hand, when different types of constraints are added to a CVRP, the cores method of the `Ant` and `Colony` classes are not modified but only the methods that describe how an ant chooses the cities to visit. Also, the other classes of the framework are well generalized as they permit to easily add constraints to the problem using the input file and the arguments of the main function.

Chapter 4 presents the results obtained with the framework on different problems. The first set of tests concern the resolutions of classical TSP, CVRP and VRPTW problems. It can be seen that, compared to other ACO implementations, there are good results with TSP and CVRP. The differences that are observable can be explained because of the fact that, in this thesis, we focus on the abstraction of ACO and not on the implementation of the most efficient method to solve these problems. A lot of different functions such as local search functions are used in the compared implementations. The tests with VRPTW instances give unsatisfactory results, but the interesting aspect to put forward is that all these implementations are made with the same framework, using the same classes, and where the implementation details which

differ are contained in the two extended classes `MyAnt` and `MyColony`. The second set of tests presents the solving of "tuned" CVRP with new constraints. These tests aim to show that a user can modify and use the ACO framework to solve different specific problems and have consistent results.

The goal of this master thesis, which was to abstract the ACO metaheuristic, is achieved through the framework developed. Nevertheless, some improvements can be done in future work. One of the reasons scala is used in this thesis is because it is a high level language with mixed paradigms: it uses pure object oriented and functional programming. As it is statically typed, it allows the writing of Domain Specific Language (DSL). Working on a DSL that permits to specify a routing problem in a way that can be used for the ants could be a serious improvement of this implementation. Scala, being high level, permits to do very specific things. A Scala expert could probably have implemented some specific details in another way. Finally, a great improvement that can be done is the use of parallelism in the implementation. The structure of the colony that runs a certain number of ants is particularly suitable for a parallel implementation where each ant has its own thread to build a solution. This could be a very interesting improvement as almost every CPU and GPU are currently built with multi-cores and multi-threads.

Bibliography

[1] M. Dorigo, G. Di Caro, and L. M. Gambardella, "Ant algorithms for discrete optimization," *Artificial life*, vol. 5, no. 2, pp. 137–172, 1999.

[2] D. Corne, M. Dorigo, and F. Glover, "The ant colony optimization meta-heuristic," *New Ideas in Optimization. McGraw Hill, New York*, 1999.

[3] M. Dorigo, M. Birattari, and T. Stutzle, "Ant colony optimization," *Computational Intelligence Magazine, IEEE*, vol. 1, no. 4, pp. 28–39, 2006.

[4] M. Dorigo, V. Maniezzo, and A. Colorni, "The ant system: An autocatalytic optimizing process," -, 1991.

[5] M. Dorigo and T. Stutzle, *Ant colony optimization*. MIT Press, 2004.

[6] L. M. Gambardella and M. Dorigo, "An ant colony system hybridized with a new local search for the sequential ordering problem," *INFORMS Journal on Computing*, vol. 12, no. 3, pp. 237–255, 2000.

[7] L. M. Gambardella, É. Taillard, and G. Agazzi, "Macs-vrptw: A multiple colony system for vehicle routing problems with time windows," in *New ideas in optimization*, Citeseer, 1999.

[8] T. Stützle and H. H. Hoos, "Max–min ant system," *Future generation computer systems*, vol. 16, no. 8, pp. 889–914, 2000.

[9] C. Blum, "Aco applied to group shop scheduling: A case study on intensification and diversification," in *Ant algorithms*, pp. 14–27, Springer, 2002.

[10] C. Blum and M. J. Blesa, "New metaheuristic approaches for the edge-weighted k-cardinality tree problem," *Computers & Operations Research*, vol. 32, no. 6, pp. 1355–1377, 2005.

[11] R. Michel and M. Middendorf, "An aco algorithm for the shortest common supersequence problem," 1999.

[12] G. N. Varela and M. C. Sinclair, "Ant colony optimisation for virtual-wavelength-path routing and wavelength allocation," in *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, vol. 3, IEEE, 1999.

[13] M. Dorigo and L. M. Gambardella, "Ant colony system: a cooperative learning approach to the traveling salesman problem," *Evolutionary Computation, IEEE Transactions on*, vol. 1, no. 1, pp. 53–66, 1997.

[14] B. Bullnheimer, R. F. Hartl, and C. Strauss, "Applying the ant system to the vehicle routing problem," in *Meta-Heuristics*, pp. 285–296, Springer, 1999.

[15] G. Reinelt, "Tsplib - a traveling salesman problem library," *ORSA journal on computing*, vol. 3, no. 4, pp. 376–384, 1991.

[16] M. M. Solomon, "Benchmark problems and solutions." `http://w.cba.neu.edu/~msolomon/problems.htm`, 2005. [Online; accessed 01-february-2014].

[17] F. Massen, *Optimization Approaches for Vehicle Routing Problems with Black Box Feasibility*. Universite Catholique de Louvain, 2013.

[18] Z. C. S. S. Hlaing and M. A. Khine, "An ant colony optimization algorithm for solving traveling salesman problem," in *International Conference on Information Communication and Management, Singapore, IPCSIT*, vol. 16, 2011.

[19] J. E. Bell and P. R. McMullen, "Ant colony optimization techniques for the vehicle routing problem," *Advanced Engineering Informatics*, vol. 18, no. 1, pp. 41–48, 2004.

[20] B. Yu, Z.-Z. Yang, and B. Yao, "An improved ant colony optimization for vehicle routing problem," *European journal of operational research*, vol. 196, no. 1, pp. 171–176, 2009.

[21] O. Rossi-Doria, M. Sampels, M. Birattari, M. Chiarandini, M. Dorigo, L. M. Gambardella, J. Knowles, M. Manfrin, M. Mastrolilli, B. Paechter, *et al.*, "A comparison of the performance of different metaheuristics on the timetabling problem," *Practice and Theory of Automated Timetabling IV*, pp. 329–351, 2003.

[22] M. Manfrin, M. Birattari, T. Stützle, and M. Dorigo, "Parallel ant colony optimization for the traveling salesman problem," *Ant Colony Optimization and Swarm Intelligence*, pp. 224–234, 2006.

[23] M. Dorigo and T. Stützle, "Ant colony optimization: overview and recent advances," *Handbook of metaheuristics*, pp. 227–263, 2010.

[24] T. Stützle and M. Dorigo, "Aco algorithms for the traveling salesman problem," *Evolutionary Algorithms in Engineering and Computer Science*, pp. 163–183, 1999.

[25] P. Toth and D. Vigo, *The vehicle routing problem*. Siam, 2001.

[26] M. Dorigo and L. M. Gambardella, "Ant colonies for the travelling salesman problem," *BioSystems*, vol. 43, no. 2, pp. 73–81, 1997.

[27] W. F. Tan, L. S. Lee, Z. Majid, and H. V. Seow, "Ant colony optimization for capacitated vehicle routing problem.," *Journal of Computer Science*, vol. 8, no. 6, 2012.

List of Figures

List of Tables