UNIVERSITÉ LIBRE DE BRUXELLES
Faculté des Sciences
Département d'Informatique

Development of an automatically configurable ant colony optimization framework

Aldar Saranov

Promoter : Prof. Thomas Stützle

Master Thesis in Computer Sciences

Academic year 2016 - 2017+1

Dedicated to my mother Lena, who was always sincerely supporting me

"*If one does not know to which port one is sailing, no wind is favorable.*"

Lucius Annaeus Seneca, 1st century AD

"*The general, unable to control his irritation, will launch his men to the assault like swarming ants, with the result that one-third of his men are slain, while the town still remains untaken. Such are the disastrous effects of a siege.*"

Sun Tzu, "The Art of War", 5th century BC

Acknowledgment

I want to thank my promoter, prof. Thomas Stützle, whose determination and competencies were leading me to the goal of the paper, and my friend, Alain, who made this precious time of my education possible.

Contents

Chapter 1

Introduction

Some species show an extreme degree of social organization. Such species (e.g. ants) have pheromone production and detection body parts and therefore seize an ability to communicate between each other in an indirect way. This concept has inspired the development of algorithms, which are based on the social behavior of the ant colonies called ant colony optimization algorithms. These algorithms allow to solve NP-hard problems in a very efficient manner. These algorithms are considered to be metaheuristics.

At the moment, there exists a state-of-the-art software called ACOTSPQAP. As the name says, it is designed to solve Traveling Salesman Problem and Quadratic Assignment Problems with high-efficiency by means of ACO meta-heuristics. It also includes different algorithm variations as AS, EAS, MMAS, RAS, BWAS, ACS. As for the implementation, it is written in C and, therefore, it adheres to structural programming paradigm. Although switching to a different problem type context is still possible, it is nonetheless more costly than if it is was written in an objective oriented language.

The development of a flexible ACO framework is the next step of formalizing this area. Such a framework can then be used as a tool to help resolving various optimization problems. This report gives a brief overview of the current state of the ACO research area, existing framework description and some tools which can be used for the automatic configuration of the framework.

[Explain what is done]

In the chapter 2, we describe the theoretical foundations of the framework, that is required to develop. However unreasonable selection of a running configuration may lead to producing of low-quality results even after a long runtime execution. A simple decision is to perform certain configuration tuning before the actual executing. This chapter also introduces an efficient configuration software, that allows to obtain high-performing configurations for further framework exploiting.

The development of such framework would also require its application to some particular problem for testing and analysis purposes. Vehicle Routing Problem was chosen as the one. It is a widely researched NP-hard combinatorial optimization problem, therefore, it will allow us to perform benchmark comparison of the public available problem instances and its solutions with the solutions, that we obtain. In the chapter 3, we introduce

formulation of Vehicle Routing Problem and also mention several variations thereof.

In the chapter 4, we describe thorough implementation details and decisions for the ACO framework, the way we adapt it specifically to VRP-problems and also show line-by-line description of the parameter space defined for the framework. We also explain there how do we attain high flexibility of the framework in order to adapt it for various problem types besides VRP.

In the chapter 5, one can see the experimental set-up, tuning results and interpretations that are made.

[TODO: Add more?]

[TODO: fix citing]

Chapter 2

Background

2.1 Combinatorial Optimization Problems and Constructive Heuristics 5

Combinatorial optimization problems (COPs) are a large class of mathematical optimization problems. These problems can be described as a grouping, ordering, assignment or any other operations over a set of discrete objects. In practice, one may need to resolve COPs, which have a large number of extra constraints for the solutions to consider them as admissible. Many of these problems which are still being thoroughly researched at the moment, belong to NP-hard discrete optimization problems. The best performing algorithms known today to solve such problems have a worst-case run-time larger than polynomial (e.g. exponential).

An Optimization Problem is a tuple [**?**] (Φ, ω, f), where

- Φ is a search space consisting of all possible assignments of discrete variables x_i, with $i = 1, ..., n$
- ω is a set of constraints on the decision variables
- $f : \Phi \to R$ is an objective function, which has to be optimized

The problem formulation describes an abstract task (e.g. find the minimum spanning tree of some graph), while the instance of a problem describes a specific practical problem (e.g. find the minimum spanning tree of a given graph G). The objective function is the sum of the weights of the selected edges.

Any combination of solution components is called a candidate solution (not necessarily satisfying all problem constraints). Oppositely, solution is a candidate solution, that satisfies all problem constraints.

Since solving of NP-hard problems by trying to find provably optimal solutions is unreasonable, one can apply heuristic algorithms, which more or less provide solutions with relatively good quality consuming reasonable quantity of resources (time/power, memory etc.). An important class of such heuristic algorithms are constructive heuristics. Constructive heuristics start with an empty or partially built solution, which is then

being completed by iterative extension until a full solution is completed. Each of the iterations adds one or several solution components to the solution. For example, greedy constructive heuristics add the best-ranked components by their heuristic values.

2.2 Ant Colony Optimization 10

ACO algorithms are a subclass of constructive heuristics. Meta-heuristic is a top-level heuristic, which is used to improve the performance of an underlying, basic heuristic. ACO is such a metaheuristic that can be used to improve the performance of a construction heuristic. One has to remark several necessary features of the ACO algorithms:

- ACO algorithms are population-based algorithms. Solutions are being generated at each iteration.
- Solutions are being generated according to a probability-based mechanism, which is biased by artificial pheromones that are assigned to problem specific solution components.
- The quality of the generated solutions affect the pheromones, which are updated during the run of the algorithm.

Listing 2.1: General ACO pseudo-code

```
1  procedure ACO_Metaheuristic
2  repeat
3    foreach ant do
4      repeat
5        extend-partial-solution-probabilistically()
6      until solution-is-complete()
7
8    foreach ant in select-ants-for-local-search() do
9      apply-local-search(ant)
10
11   evaporate-pheromones()
12   deposit-pheromones()
13 until termination-criteria-met()
14 end
```

Pheromones are numeric values associated to the solution components that are meant to bias the solution construction in order to improve the quality of the generated solutions. Several ants generate the solutions by an iterative approach (see listing 2.1). After this, an optional local search is applied. Next, pheromone evaporation and deposit is done. Evaporation helps to reduce the convergence-prone behavior of the algorithm. Deposit is the part where the solutions affect the pheromone values in order to bias the future solutions.

2.2.1 Choice of pheromone trails and heuristic information

Generally, there are two mechanisms of biasing the solution construction - pheromones and heuristic values.
Hereby we introduce the following components:
C - set of solution components (a combination of which can constitute a solution).
$\tau_c \in T$ - pheromones trail values for solution component bias.
$\tau'_c \in T'$ - pheromones trail values for particular purposes.
π - candidate solution.
$\eta_c \in H$ - heuristic information (constant in time).

Higher values of τ_c stand for a higher probability that the component c will be added to a solution. Additionally, problem-specific pheromones such as τ'_c can be used for auxiliary purposes.

Similarly heuristic values are numerical attributes of the solution components. However, generally heuristic values are assigned as constants at the start of the solution construction, although in extensions one can use heuristic values, which are a function of the generated partial solution. These are called *adaptive* heuristics and normally they consume larger computer resources although they often lead to a better quality of the generated solutions. Heuristic information H is similar to the pheromone trails in terms of that they both bias the solution component choice. However, it is defined in problem-specific way and is not updated during the algorithm execution ($\forall c \in C, \exists \eta_c \in H$). Heuristic information is either static values or values which are defined by a function of the current partially constructed solution.

2.2.2 Solution component choice

The solution construction phase, as says the name, yields a new solution set. The probability of c_j to be added at a certain step can be calculated by different techniques (i.e. $Pr(c_j|T, H, s)$). Each ant starts with an empty solution s. Each ant may produce one solution at one execution of the solution construction phase. At each construction step one solution component is added. A frequently used rule is defined as follows:

$$Pr(c_j) = \frac{t_j^\alpha \times \eta_j^\beta}{\sum_{c_k \in N_i} t_k^\alpha \times \eta_k^\beta} \forall c_j \in N_i \tag{2.1}$$

α and β are parameters that determine the impact of the pheromone trails and heuristic information on the final probability. Another alternative has been proposed by Maniezzo [**?**], which combines the pheromone trails and heuristic information in a linear way.

$$Pr(c_j) = \frac{\alpha \times \tau_j + (1-\alpha) \times \eta_j}{\sum_{c_k \in N_i} \alpha \times \tau_k + (1-\alpha) \times \tau_k} \forall c_j \in N_i \tag{2.2}$$

Since it does not use exponentiation operations such choice rule is preferable for performance-targeted frameworks. However, this algorithm may cause undesired biases if the range

of the values are not taken into account. The third alternative is invented by Dorigo and Gambardella [**?**] in Ant Colony System (ACS) algorithm. The rule of solution component choice in ACS is also called pseudo-random proportional rule. A random uniform value q is generated in range $[0; 1)$ and if $q > q_0$, where q_0 is a predefined parameter, then the probability of choosing c_j is calculated according to formula (2.1). Otherwise, the solution component is picked as:

$$c_j = \underset{c_k \in N_i}{\operatorname{argmax}} t_k^\alpha \times \eta_k^\beta \tag{2.3}$$

Apparently, larger q_0 gives more greedy choice.

2.2.3 Construction extensions

Lookahead concept was introduced. It says that at each decision step several solution components should be considered at once in order to get the next solution component [**?**]. This says that in constructing a candidate solution, ants use a transition rule that incorporates complete information on past decisions (*the trail*) and local information (*visibility*) on the immediate decision that is to be made [**?**]. The look-ahead mechanism allows the incorporation of information of the anticipated decisions that are beyond the immediate choice horizon. Generally, it is worth to be implemented when the cost of making a local prediction based on the current partial solution state is much lower than the cost of the real execution of the move sequence.

A candidate list restricts the solution component set to a smaller set to be considered. The solution components in this list have to be the most promising ones at the current step [**?**]. Usually, this approach yields a significant gain of computation time, depending on the initial set-up (i.e. if this list is precalculated once before the run). Nonetheless, it can also depend on the current partial solution.

Hash table of pheromone trails. It allows to efficiently save memory when the updated pheromone trails are in a sparse set in comparison to the set of all solution components. Search and updating of the elements of the hash-table is expected to be done within linear time [**?**].

Heuristic precomputation of the values $t_j^\alpha \times \eta_j^\beta$ for each of the solution components which are used in formula (2.3). The reduction of computation time is based on the fact that all these values will be shared by the ants at each iteration.

External memory extension is based on starting the solution construction from a partially constructed solution with partial destroying of a certain good solution and reconstructing it and thus anticipating to obtain even a better solution [**?**]. This iterated greedy extension was inspired by genetic algorithms and described in [**?**]. It uses reinforcement procedures of the elite solutions with deferred reintroducing of solutions segments in following iterations (see listing 2.2). The ACO iteration is composed of two stages. First is meant to initialize the external memory. The second is the solution construction itself based on a partial solution.

Listing 2.2: External memory iteration pseudo-code

```
1  procedure ACO-external-memory
2  initialize the external memory
3  repeat
4    set m ants in the graph
5
6    ants construct a solution using neighborhood graph and the
         pheromone matrix
7
8    select k-best solutions and cut randomly positioned and
       sized segments
9
10   store the segments into the external memory
11 until the external memory is full
12
13 done = false
14
15 while not(done)
16   ants select their segments according to tournament
       selection
17
18   ants finish the solution construction
19   update the pheromone matrix
20   update the memory
21 end
22 end
```

Tournament selection procedure is usually defined as follows: from a set of solutions, one selects randomly $tSize$ solutions, finds the best solution among them and places this solution into result set. This is repeated m times.

Iterated ants also uses the partially constructed solutions. Based on the following additional notions. Destruct() removes some solution components from a complete solution [**?**]. Constuct() constructs a complete solution from initially partial solution. An acceptance-criterion chooses one of two complete solutions in order to continue the construction with it. Concrete implementations of these strategies are defined in problem-specific way. The algorithm of the extension is showed in listing 2.3.

Listing 2.3: General iterated ants pseudo-code

```
1  procedure iterated-ants
2    s0 = initial-solution()
3    s = local-search(s0)
4    repeat
5      sp = destruct(s)
6      s' = construct(sp)
7      s' = local-search(s')    // optional
8      s = acceptances-criterion(s,s')
9    until termination criterion met
10 end
```

Cunning ants algorithm tackles to the solution generation by iterated producing of new ant population. The algorithm has a pheromone database and an ant population of fixed size. For every existing ant, a new one is produced which borrows some solution parts from its parent [**?**]. Then in each such ant pair a winner is selected and all winners continue their work in the next iteration. After each iteration all winners jointly update the pheromone database and stop if the termination criteria is met. Similarly the solution component inheritance process is problem-specific.

As an extension of the original cunning ants variation, one may use local search after the reconstruction step and also a custom pheromone update algorithm.

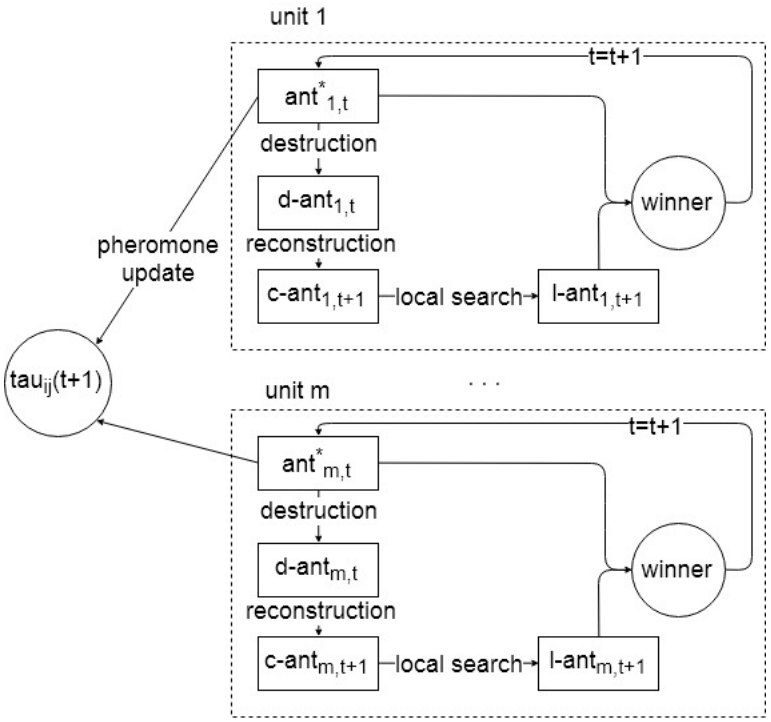


Figure 2.1: Cunning ants

2.2.4 Global pheromone update

As it was said before, the key idea of the ACO algorithm is the pheromone trail biasing. It is composed of two parts - pheromone evaporation and pheromone deposit. Pheromone evaporation decreases the values in order to reduce the impact of the previously deposited solutions. The general form formula is as follows.

$$\tau_{new} = evaporation(\tau_{old}, \rho, S^{eva}) \tag{2.4}$$

where:

- τ_{new}, τ_{old} - new and old pheromone trail values
- $\rho \in (0, 1)$ - evaporation rate
- S^{eva} - selected solution set for evaporation

The classic linear reduction is as follows:

$$\tau_j = (1 - \rho) \times \tau_j \quad \forall \tau_j \in T | c_j \in S^{eva} \tag{2.5}$$

Hence $\rho = 1$ stands for the pheromone trails are being reset completely to zero whereas $\rho = 0$ means that pheromone trail remains exactly the same. Other values cause a geometrically decreasing sequence of the pheromone trails with a number of iterations. Usually, all the solution components are being selected for the evaporation, however, some modifications perform distinctive selection of the components. The generic intention of the evaporation is to slow down the convergence of solution components, which can be selected with some reasonable probability, to a limited subset of all solution components.

In contrast, the pheromone deposit increases the pheromone trail values of selected solution components. The solution components may belong to several solutions at once. The general deposit formula is described as:

$$\tau_j = \tau_j + \sum_{s_k \in S^{upd}} w_k \times F(s_k) \tag{2.6}$$

- $S^{upd} \subseteq S^{eva}$ - the set of solutions selected for the pheromone deposit
- F - non-decreasing function with respect to the solution quality
- w_k - multiplication weight of the k-th solution.

The following update selection techniques can be used:

1. **Ant system** - selects all solution from the last iteration
2. Single update selections:

(a) **iteration-based** update - selects the best solution from the last iteration

(b) **global-based** update - selects the best solution recorded since the start of the run. Provides fast convergence but may lead to a state called stagnation. Stagnation is a state, where the pheromone trails are defined in such way, that some solution components are selected regularly from one iteration to another, so that virtually no other solution components can be selected.

(c) **restart-based update - selects the best solution since last pheromone reset.**

In the minimization case, which is the case of VRP, typically one adds a value inversely proportional to a value of the solution quality function.

$$w_k \times F(s_k) = 1/f(s_k) \tag{2.7}$$

For the mentioned update techniques several variants are possible:

1. **Ant System** - i.e. without extensions. Every pheromone trail is evaporated.

2. **Ant Colony System** - uses formulas 2.1 or 2.3 for solution construction. It also uses local pheromone update according to formula 2.15. Thus, it makes the components, that have been already chosen, less attractive for the rest of the ants. After the solution construction is finished, the ants deposit the pheromone trails according to formula 2.8

$$\tau_j = (1-\rho) \times \tau_j + \rho \times \Delta\tau_j^*, \forall j \in S^* \tag{2.8}$$

   where S^* is the best solution so far, and $\Delta t_j^* = 1/z^*$, z^* is the cost of S^*.

3. **Max-Min Ant System**. The pheromone values are updated by evaporating all pheromone trails according to 2.10 with consequent deposit of $\Delta\tau = 1/z$ to the solution components in global-best or iteration-best or reset-based solution, where z is the cost of this solution. The amount of pheromones per component is bounded $\tau_i \in [\tau_{max}; \tau_{min}]$. The update schedule switches between ib, gb and rb depending on the value called branching factor [**?**].

$$\tau_i' = \rho\tau_i + \sum_{\forall ants} \delta t_i^k \tag{2.9}$$

   where

$$\delta t_i^k = \begin{cases} \frac{1}{L^k(t)} & \textit{if i-th component is used by ant k at the iteration} \\ 0 & \textit{otherwise} \end{cases}$$

   $L^k(t)$ - *is the length of k-th ant tour*

$$\tau_j = (1-\rho) \times \tau_j, \forall j \in N \tag{2.10}$$

A special *pheromone trail smoothing* mechanism is often used hand in hand with MMAS update. It is a tool to oppose the effect of convergence, thus, it is useful for long runs that are excessively exploiting. It is applied in case, if fraction Ω (branching factor) of solution components, whose pheromone trail values surpass the computed threshold τ_t, exceeds predefined fraction Ω'. Higher Ω values correspond to higher grade of convergence. Computing of τ_t is shown in formula 2.11.

$$\tau_t = \tau_{min} + (\tau_{max} - \tau_{min}) \times \lambda \tag{2.11}$$

where τ_{min} - minimum pheromone trail border, τ_{max} - maximum pheromone trail border. λ - threshold determining factor in [0;1].

When applied, the value of each pheromone trail is increased according to formula 2.12, and thus, exploration is boosted.

$$\tau^*_{ij} = \tau_{ij} + \delta \times (\tau_{max} - \tau_{ij}) \tag{2.12}$$

where δ - is elevation parameter in [0;1].

4. **Rank-based Ant System** uses the notion of rank by depositing [**?**] to the pheromone trail the following value: $w_k \times F(s_k) = \frac{max(0,w-r)}{f(s_k)}$ where: $w = |S^{upd}|$, r-solution rank in the current iteration

5. **Elitist Ant System** - all solutions from the current iteration deposit pheromones as well as the global-best solution s_{gb} deposits an amount $w_{gb} \times F(s_{gb}) = Q \times \frac{m_{elite}}{f(s_{gb})}$ [**?**], where m_{elite} is the multiplicative factor that increases the weight of s_{gb} solution, Q is the same multiplication factor from the standard AS.

6. **Best-Worst Ant System** denotes that pheromone trail values are deposited to the solution components that constitute the global-best solution but also evaporation is applied to the components from the worst solution of the current iteration (which are in the global-best solution) [**?**]

   Mutation mechanism is used hand in hand with BWAS. It is inspired by natural evolution process of gene mutation. It provides certain amount of diversity by modifying pheromone values of some components. The mutation mechanism is illustrated in formula 2.13. Every component is mutated with probability p_{mut}. Uniform random q is generated within $[0; 1]$

$$\tau' = \begin{cases} \tau + 4 \times \tau_{avg} \times f, & \text{if } q < 0.5 \\ \tau - 4 \times \tau_{avg} \times f, & \text{if } q \geq 0.5 \end{cases} \tag{2.13}$$

   where τ_{avg} - is the average pheromone trail value before global update execution, f - is BWAS mutation factor defined in formula 2.14 for the case of time based execution.

$$f = \frac{t - t_{reinit}}{t_{end} - t_{reinit}} \tag{2.14}$$

   t - is current time, t_{reinit} - is time of last pheromone reinitialization, t_{end} - is expected time of the run finish.

The pheromone update schedule allows to dynamically switch between different solution selections. For example, an ACO algorithm may start from ib and then converge to gb or rb. If one wants to start the problem solution process with exploratory behavior and end up with exploiting one, then it is possible to start solution process with ib-update and then switch to gb or ib updates at the later solution iterations. The update schedule has a major influence on the ACO algorithm performance, since it determines the balance between convergence and exploration traits of the algorithm in a global scale.

2.2.5 Initialization and reinitialization of pheromones

The solution selection algorithm plays a critical role in determining the ACO algorithm behavior. However, it is also important how one initializes and reinitializes the pheromones. Typically, for ACS and BWAS very small initial values are assigned in the beginning and large ones for MMAS. Small values stand for exploitative bias, whereas large stand for exploration one, because in the first case better solution components will rapidly gain pheromone values advantage over the rest ones, and in the second case the high values will oppose such fast convergence. Pheromone reinitialization in often applied in ACO algorithms since otherwise the run may converge rapidly. MMAS uses a notion of branching factor in such a way that when it falls below a certain value, all pheromone trails are reset to the initial pheromone trail value. BWAS resets whenever the distance between global-best and global-worst solution for a certain iteration plummets down lower than a predefined value.

2.2.6 Local pheromone update

For sake of making the behavior more exploratory one can apply [**?**] local pheromone update in ACS according to the formula:

$$\tau_j = (1 - \epsilon) \times \tau_j + \epsilon \times \tau_0 \quad , \forall \tau_j | c_j \tag{2.15}$$

ϵ - update strength. τ_0 - initial pheromone trail.

This local evaporation is applied to solution components that are used by some ant and future ants will choose these components with smaller probability. Therefore, already explored components become less attractive. An important algorithmic detail is whether the algorithm will work sequentially or in parallel, since in parallel case ants will have to modify the pheromone trails simultaneously, whereas in sequential they have no such a problem. However it does not behave very efficiently with other ACO algorithms but ACS, because it relies on a strong gb-update and a lack of pheromone evaporation.

2.2.7 Pheromone trail limits

As it was said before, MMAS is based on restricting the pheromone values in a certain range. Parameter p_{dec} is the probability that an ant chooses exactly the solution components that reproduce best solution found so far.

$$\tau_{min} = \frac{\tau_{max} \times (1 - \sqrt[n]{p_{dec}})}{n' \times \sqrt[n]{p_{dec}}} \tag{2.16}$$

where n' - is an estimation of the number of solution components available to each ant at each construction step (often corresponds to $\frac{n}{2}$). The p_{dec} allows to estimate the reasonable values for τ_{min} according to formula 2.16. This formula relies on two facts. First is that the best solutions are found shortly before search stagnation occurs. The second fact is that the relative difference between upper and lower pheromone trail limits has more influence on the solution construction than the relative differences of the heuristic information.

2.2.8 Local search

Local search allows to significantly increase the obtained solutions quality for specific problems. It is based on small iterative solution changes obtained by applying a so-called neighborhood operator, followed by evaluation of the solutions. Generally there are two neighborhood scanning strategies:

- **best-improvement** scans all the neighborhood and chooses the best solution.
- **first-improvement** takes the first found improving solution in the neighborhood.

In the general ACO algorithm this stage is optional. If it is fast and effective it may be used to all solutions at each iteration. It is not guaranteed to converge on an optimal solution, however, it may converge on a relatively good local optimum. In other cases applying it in iteration-best ants may be the optimal strategy. A neighborhood operator is always defined in a problem-specific way. More thoroughly this is explained in chapter 3.

In many COPs it is possible to optimize computation of objective function value of a newly obtained solution, based on objective function value of an old solution. From feasibility point of view one can apply a local move and check whether this is a solution after. Another option is to define a neighborhood operator in such way, that all solution's neighbors are also feasible solutions.

In many problem-depending cases perturbative search can be embedded into the local search heuristics. The aim of such mechanism is to allow escaping from local search space "peaks" which might be not as good as the others.

One of the most efficient metaheuristics in local search domain is Iterated Local Search (ILS) [**?**]. Usually it leads to much better solutions, than the ones obtained by repeated random trials. It uses custom components as local search procedure, perturbation procedure and acceptance criterion. The pseudo-code is shown in 2.4.

Listing 2.4: Iterated Local Search pseudo-code

```
1 procedure IteratedLocalSearch
2   s_0 = GenerateInitialSoluition()
3   s* = LocalSearch(s_0)
4   repeat
5     s' = Perturbation(s*, history)
6     s*' = LocalSearch(s')
7     s* = AcceptanceCriterion(s*, s*', history)
8   until TerminationCondition()
9 end
```

The simplest acceptance criteria is choice of a better solution, however one may want to use a probabilistic approach. It prevents major perturbation moves, that do not effectively contribute to objective goal improving, and thus, attains good trade-off between exploration and intensification. History can be taken into account in this heuristics, so the whole procedure is in fact a "Markovian" process, which implies that further moves completely depend on the current state.

2.3 Iterated F-Race 5

Automatic configuration is a process that optimizes the performance of a certain algorithm as a goal function based on input parameters of the algorithm. The general parameter types are:

- **categorical** parameters - represent discrete values without any implicit order or distance measure between its possible values. Define the choice of constructive procedure, choice of branching strategies (i. e. algorithmic blocs) and so on.
- **ordinal** parameters - are also assigned to finite discrete values, but they have implicit value order (such as *low, middle, high*). Define lower bounds, neighborhoods.
- **numerical** parameters - define integer or real values/constants such as weighting factors, population sizes.

Some of the parameters maybe subordinate to other ones. This means that their values only make sense if the parameters, that they subordinate to, are assigned to certain values. This can be expressed as a scalar value constraint (e.g $a < b$) or as a dependency on concrete values of some categorical or ordinal parameter. In addition, one can force constraints for several different parameters combined (even the ones of different type).

Figure 2.2 shows the configuration software and software to configure. Configuration script has parameter metadata at its disposal. Based on them, the configuration software runs the software to configure with candidate configurations according to some algorithm. After the configuration process finishes, the configuration software renders the best configurations obtained. In the most general software case there are two measures of the performance - solution quality (to maximize) and computation time (to

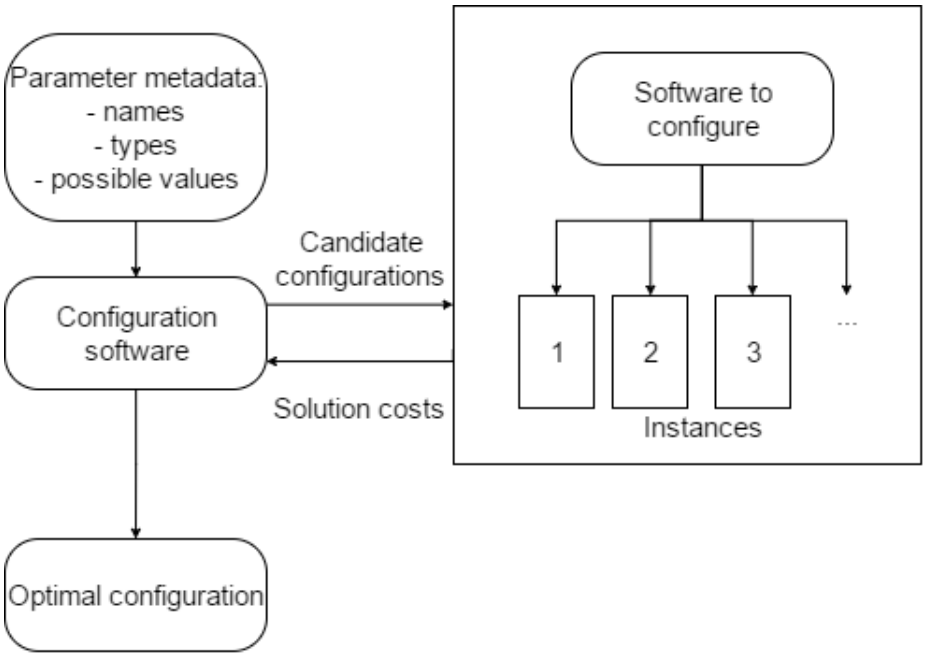


Figure 2.2: Automatic configuration scheme

minimize). This general approach is called an offline tuning. It introduces a learning stage on training instances before learning on the real-world instance set.

A widely used configuration algorithmic family is racing algorithms. The simplified algorithm is shown in 2.5 and an illustration is in 2.3.

Listing 2.5: General racing pseudo-code

```
1 procedure racing
2 start with an initial candidate set Theta
3 repeat iterations I
4   process an instance stream
5   evaluate the candidates sequentially
6   remove inferior candidates
7 until winner is selected or exit condition fulfilled
8 end
```

I-RACE (iterated race) configuration implementation was developed and described in [**?**]. It was implemented in R with taking into account the parallel programming techniques and an initial candidate set-up. It is implemented as an offline tuning. That means, that the tuning is a separate preparatory process from the actual real-world problem application. The feature of the I-RACE is based on iterated generation of new configurations and removing of solutions with lower quality for further evaluating on the problem instances.

In order to tune the configurations that are sampled, I-RACE algorithms uses independent sampling distributions for each of the parameters. For numeric values those are normal distributions and discretely-defined distributions for the rest. The configuration biasing procedure is based on modifying the sampling distributions.

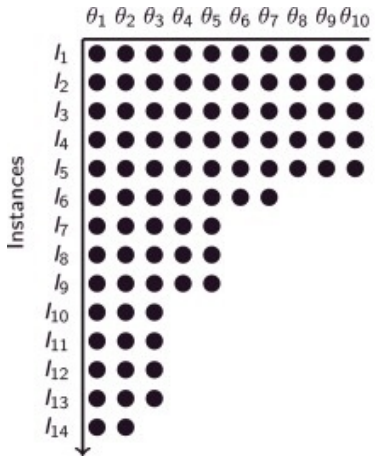


Figure 2.3: I-RACE execution illustration

Iterated racing is an automatic configuration implementation that consists of three steps:

- Sampling new configurations according to a particular distribution.
- Selecting the best configurations from the newly sampled ones by means of racing.
- Updating the sampling distribution in order to bias the sampling towards the best configurations.

After new configurations are sampled comes the selection stage. At the configuration selection stage, I-RACE runs each of the configurations on a single problem instance from the predefined instance set. After each racing iteration the worst configurations are discarded. Then the rest of the configurations update parameter sampling distributions. The racing stops when the number of the survived configurations becomes small enough (defined by termination criteria).

Some I-RACE extensions can be applied. **Initial configurations** can be set before the run of the I-RACE. **Soft-restart** is used for preventing premature convergence. Such convergence may suppresses configuration diversity and, therefore some good configurations may be lost. This restart is triggered if the value of an ad hoc function of configuration distance is lower than a certain margin. Then reinitialization is applied to the elite configurations.

Chapter 3

Vehicle Routing Problem

3.1 Problem definition

VRP is a generalization of Traveling Salesman Problem. TSP is such a particular case of VRP, where a single vehicle is disposed instead of a whole fleet, and all capacity and length constraints are always satisfied. Both of these problem require return to the initial node in general variation. Both problems have an objective defined as the total distance passed by a vehicle/vehicles.

As a basis for VRP formulation variation we use the Classical Vehicle Routing Problem [**?**]. However minor generalizations are actually implemented in the framework above the given formulation.

VRP is defined on a complete undirected graph $G = (V, E)$. A set of nodes $V = \{0, 1, ..., n\}$ is a union of all customer nodes $V \setminus \{0\}$ and a depot node 0. Distance (a.k.a. travel cost or length or time) c_e or c_{ij} is assigned to each edge $e \in E = \{(i, j) : i, j \in V, i < j\}$. Vehicle fleet is represented by m identical vehicles, having a capacity Q and total distance limit L.

Constraints are defined as follows:

1. Each customer is only visited by exactly one vehicle.
2. Each route starts and ends at the depot.
3. Total demand, that any vehicle provides does not exceed the capacity Q.
4. Total distance, that any vehicle passes on route does not exceed the limit L.

A proposed problem formulation is given below.

The decision variables x_e are presented by an integer variable for each edge $e \in E$.

Let $r(S)$ be the minimum number of vehicles needed to serve the customers $S \subseteq V \setminus \{0\}, S \neq \emptyset$.

Let $\delta(S) = \{(i, j) : i \in S, j \notin S \text{ or } i \notin S, j \in S\}$

Then we have a following integer linear programming problem:

$$minimize \sum_{e \in E} c_e x_e \tag{3.1}$$

$$\sum_{e \in \delta(i)} x_e = 2, i \in V \setminus \{0\} \tag{3.2}$$

$$\sum_{e \in \delta(0)} x_e = 2m, \tag{3.3}$$

$$\sum_{e \in \delta(s)} x_e \geq 2r(S), \tag{3.4}$$

$$x_e \in \{0, 1\}, e \notin \delta(0) \tag{3.5}$$

$$x_e \in \{0, 1, 2\}, e \in \delta(0) \tag{3.6}$$

However in addition to this definition one can apply several soothing generalizations:

1. **Non-symmetric matrix** can be considered, i.e. moving an edge one-way may have a different cost from moving backwards.
2. **Non-homogeneous vehicle fleet**. Every vehicle has its own maximum capacity and maximum distance.
3. A particular **vehicle may lack of maximum distance restriction**. In other words, maximum distance can be set to infinity.
4. **Open/closed problem variations can be also generalized**. Open VRP is a VRP variation, where a vehicle does not have to return to the depot. Closed is a variation, where it must return to the depot in any case. Generalization can be done at matrix definition step by setting all edges leaving from a customer to the depot to 0 in case of opened. On contrary, in closed VRP they are set to their actual travel cost.

Two general feasibility constraints can be verified in advance before solving a problem instance. Firstly, one can check whether total demand of all customers is not larger than total supply of all vehicles in case of limited fleet. Secondly, one can check that maximum demand of a single customer does not exceed maximum supply of a vehicle fleet, since this customer will never be satisfied, due to the fact, that only one vehicle can supply a customer.

The problem size parameter, that is required for MMAS, is equal to the number of nodes, i.e. all customers plus one depot.

In case of limited non-homogeneous vehicle fleet, the results strongly depend on order of vehicle departure. We propose to emit vehicles in order of their decreasing capacities. By this we try to maximize number of customers visited by a single vehicle and remove redundant long-distance travels over zones, where customers were already satisfied.

3.2 Candidate list

Candidate list can severely boost performance. Similarly to TSP, **cl nearest neighbors** candidate lists can be used. For sake of optimization, these lists are computed once in the beginning of metaheuristcs execution and exploited during whole the run. Such a candidate list must be determined separately for every node.

During the preselection phase, the algorithm will examine the candidate list for the current node in the first place. In case if none of those will satisfy constraints (capacity, distance, already visited constraints), only then the nodes outside of the candidate list will be considered.

3.3 Heuristic values

In VRP and TSP, one typically uses the inverted value of the distance between two nodes as the heuristics value, see 3.7. Thus, the selection phase will bias the tour edge towards shorter ones.

$$\eta_{edge} = 1/c_{edge} \tag{3.7}$$

3.4 Local search

Two local search variations are proposed - 2.5-opt move [**?**] and Iterated Local Search, that uses underlying 2.5-opt move.

2.5-opt move illustrtions are shown in 3.1, 3.2, 3.3, 3.4. In 2.5-opt move, a replacable edge $(y1; y2)$ is chosen, and swapped with another sequence of edges, consisting from one up to three customers, denoted by $(x2)$, $(x2, x3)$ or $(x2, x3, x4)$ on the illustrations. Therefore, size of solution neighborhood is a function depending on tour size of complexity $O(n^2)$.

The whole local search procedure consists of sequential attempts of such local moves by trying all possible position combinations of the replacable edge and the replacing ones. The length of the replacing sequence is defined by a random number generator. The local move is applied if it is computed that it will benefit to the solution objective function. However, if local move was applied, one does not reset the position combinations of the loop, but continue with the current progress. In case if solution consists of several tours, then this procedure is applied to every one of them.

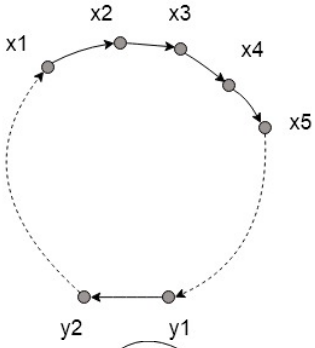
In order to accelerate computation of the objective function value one may simply distract the distance of the edges, that are removed and add distance of the edges, that are added. Thus, we avoid full tour recomputation. If during the full attempt cycle one applied the local move at least one time, then the whole attempt cycle is repeated. Otherwise, it means that we have converged to a local optimum and we keep the current solution.

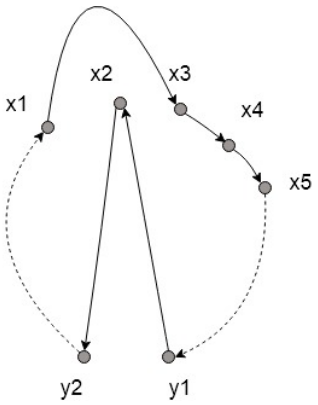
[Add pseudo-code???]

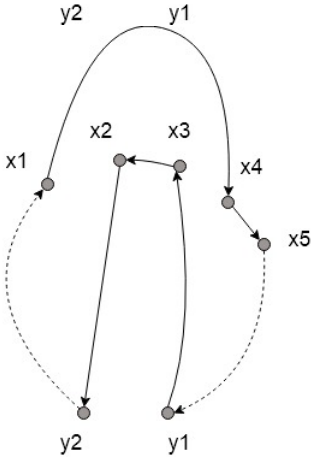
For ILS in case of VRP problems, one can implement perturbation as double-bridge move, see 3.5 and 3.6.

3.5 Solution destruction

Solution destruction is a procedure, that is necessary to implement for iterated greedy extensions. A whole range of diverse algorithms can be proposed here only being limited by fantasy. We implemented it as the following stochastic procedure: every tour of a certain solution with certain probability is a subject to slicing. Slicing means that a random number of final solution components are removed. At the same moment, the corresponding quantity of length-capacity and supply-capacity is compensated to the vehicle of the sliced tour, so refinishing of the tour can be done. Objective function is also modified to the actual progress. Still, the initial "parent" objective function value has to be backed-up, in order to be able to compare partially destroyed solutions in External memory during tournament selection process.







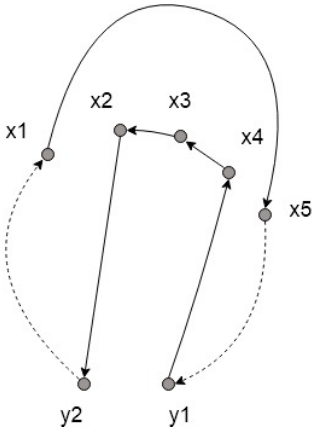
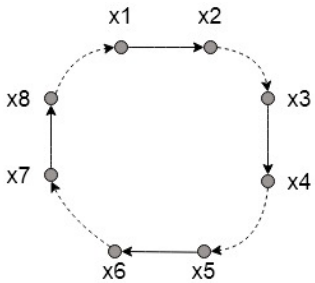


Figure 3.1: Initial tour

Figure 3.2: Result of swapping with 1 node

Figure 3.3: Result of swapping with 2 nodes

Figure 3.4: Result of swapping with 3 nodes



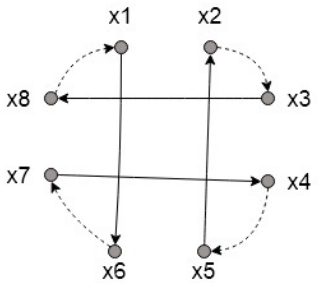


Figure 3.5: Tour before perturbation

Figure 3.6: Tour after applying double-bridge perturbation

Chapter 4

Implementation

4.1 Software design

The software framework is implemented in Java 1.8.0 in correspondence with object-oriented programming paradigm. Besides VRP, many other combinatorial optimization problems are also anticipated to be solved by means of this framework. This framework remains highly flexible and adaptive in terms of problem variations. This is attained by design patterns and deep abstraction of the framework components. Large application of abstractions ensures minor code-repetition, being ready for some algorithm modifications. Dependency injection concept ensures, that high-level abstraction do not determine the low-level details themselves. Instead, they only receive these dependencies as arguments. This also allows to resort to white-box unit testing of some software components.

All classes are strictly grouped in two parts. The first part is **problem independent** classes. Those are:

- **Solver** - abstract class, that encapsulates high-level metaheuristics, that aim to solve abstract combinatorial optimization problems. It is the central class in the framework. It also contains all shared lower-level algorithmic components, such as problem instance itself, solution component selector, termination criteria, local search, pheromone initializer, local and global update strategies. Basic abstract operations are already implemented in this class such as constructing one abstract solution of an abstract problem.
- **SolverStandard** - metaheuristcs without any iterated greedy or other extensions.
- **SolverIteratedAnts** - metaheuristics with iterated ants as iterated greedy extension. Also holds an IteratedCriteria.
- **IteratedCriteria** - an algorithm that gives preference to either one solution or another. Is used in SolverIteratedAnts. Can be either deterministic or probabilistic.
- **SolverExternalMemory** - metaheuristics with iterated ants as iterated greedy extension. Also contains TournamentSelector.

- **TournamentSelector** - a procedure used in SolverExternalMemory, that selects certain number of solutions from a given set according to tournament selection rule.
- **SolverCunningAnts** - metaheuristics with cunning ants as iterated greedy extension. Is is also obliged to use MMAS as global update strategy.
- **Problem** - is an encapsulation of an abstract problem. It contains general problem components as CandidateDeterminer, ComponentSructure and a flag whether it is maximizing or minimizing problem.
- **Component** - encapsulated an abstract problem solution component. Allows computation of its selection weight. Performs caching of selection weight, so it gets rid of redundant recomputations.
- **ComponentStructure** - contains the solution components in an unspecified data structure (matrix in case of VRP).
- **CandidateDeterminer** - generates abstract problem-specific CandidateList.
- **CandidateList** - abstract problem-specific candidate list. Is computed once before the run.
- **Solution** - encapsulates an abstract problem-independent solution of an abstract problem. It must have an objective function value, that is update during the construction run according to the problem formulation. It also must have a procedure of obtaining a solution component set, that are available at the current construction step. Also deep copying method must be implemented, since it is widely used by solvers and local search strategies.
- **GlobalUpdate** - abstract global pheromone update strategy. Includes standard evaporation method, since it is shared by most of the global updates.
- **AntSystem**, **AntColonySystem**, **MinMaxAntSystem**, **RankBasedAntSystem**, **ElitistAntSystem**, **BestWorthAntSystem** - correspond to the respective global update strategies. MMAS in addition contains PheromoneTrailSmoothing.
- **PheromoneTrailSmoothing** - is used bu MMAS. It is an optional step.
- **LocalUpdate** - optional argument in general case and mandatory in case of choosing AntColonyOptimization as a global update strategy.
- **LocalSearch** - can be either a direct local search implementation or an IteratedLocalSearch, that uses an abstract Move dependency.
- **IteratedLocalSearch** - performs an iterated local search. Depends on Perturbation and Move strategy. High level class, that is not dependent on a problem type.
- **Move** - full convergence local move, that is used by ILS.
- **Selector** - an abstract stochastic algorithm that selects a solution component out of a set of components according to some stochastic rule, by taking into consideration pheromone trail values and heuristic values of these components.

- **TerminationCriteria** - an abstract solver termination criteria. Implemented as time criteria and number of iterations criteria for unit testing.
- **PheromoneInitializer** - an abstract strategy that initialized/reinitializes pheromone trail values for a whole ComponentStructure. Normally implemented as a stochastic operation.

The second part is **problem dependent** classes. These classes may be easily substituted by the others, thus switching the framework application. It is only necessary to determine certain low-level methods for them, such as determining of possible solution components, problem-specific local searches and solution destroyers, determining solution component candidate lists , etc. Those are:

- **Problem2d** - extends Problem class. Encapsulates a problem, that has a two-dimensional matrix organization of its solution components.
- **ProblemVRP** - encapsulates a VRP problem instance.
- **ComponentStucture2d** - a ComponentStructure implementation for Problem2d. Implemented as dense matrix of solution components and sparse matrix of solution components by means of hash table. The latter allows to have some memory consumption gain, however it cannot be used specifically for VRP problem, due its dense definition of solution components.
- **ParserVRP** - VRP problem instance parsing. May have many implementations, since one problem type may have many formats of representation.
- **Vehicle** - encapsulation of a vehicle. Contains information related to constraints of capacity and distance.
- **Fleet** - abstract class, that contains a sequence of vehicles. Generates a vehicle iterator for their sequential emission during solution process. The order of emission depends on a particular implementation of this class.
- **Tour** - is an inherent part of VRP problem solution. Set of tours defines a solution. One tour is associated with one predefined vehicle. Contains current consumption of capacity and distance, resting capacity and distance, list of visited nodes, associated vehicle pointer.
- **SolutionVRP** - is a solution of a VRP problem instance. Contains all selected solution components, array of flags of visited customers with, vehicle iterator (generated by Fleet class), all solution tours, and current solution construction progress information.

Design patterns are widely used during development process. One of the most used patterns here are "Strategy", "Command" and "Template method", due to having a various range of algorithms on different levels, that adhere to some common structure. All concrete implementation in the program entry point are inferred from the program arguments and are created by means of many "Factory methods" in the **Main** class. "Prototype" pattern is used for **Solution** class and its sub-classes in order to be able to clone solutions one from another as it is required by some algorithms. "Iterator" is used for vehicle fleet for obtaining a sequence of vehicles, that build a solution in order of their emission.

4.2 Preselection stage

Generally in ACO optimization heuristics there are two ways of maintaining solution feasibility. First is two ensure solution **feasibility at every moment** of the run. Second is to **allow processing unfeasible solutions**, however trying to bias towards the components, that keep or likely to keep the solution feasible. Second is good for solving problems with complicated constraints that require a lot of computations in order to keep them feasible. In VRP however, we have decided to use the first approach. This will allow to avoid unnecessary construction of solutions, that are expected to be unfeasible beforehand.

Before applying selectors according to formulas 2.1, 2.2 or 2.3 one has to determine the possible solution components at the current construction step. Proposed preselection algorithm is shown in 4.1. It selects all constraint non-violating edges. Among such constraints there are non-revisiting, not going to itself, not going to the depot edges. Out of those several are filtered out, such as the ones, who violate capacity constraint and distance constraint. In particular, we ensure condition $leftDistance >= dist(current; N) + dist(N; depot)$, since it does not make to go by a certain edge, if it will not be possible even to return to the depot from it. Neither it will be possible to return to the depot by going any farther from that node, which is ensured by triangle inequality property, that works in all VRP instances, that are based on Euclidean space. In case if none such node is found, then the vehicle returns directly to the depot from the current node.

In case if a candidate list is used, then the initial set is populated from this list, and the constraints are applied after.

Listing 4.1: Solution component preselection pseudo-code

```
1  procedure preselection(currentNode)
2    foreach non-visited non-loop non-depot-leading node N
3      if leftCapacity >= demand(N)
4        if leftDistance >= dist(current;N) + dist(N;depot)
5          add component(current;N) to resultSet
6        end
7      end
8    end
9
10   if size(resultSet) == 0
11     add component (current;depot) to result set
12   end
13
14   return resultSet
15 end
```

[All blue classes decisions here]

4.3 Configuration process set-up

By aggregating all options, strategy choices and flags, one forms a parameter space according to table 1. This space formulation corresponds to the i-race representation. Every parameter is defined by its i-race name, publicly known alias (or new if such does was not invented), type, possible parameter values, condition of validity. Every parameter is can be categorical, integer or real. In case of integer and real lower and upper bounds of the parameter are specified. In case of categorical all possible values are listed. Condition is a predicate, that defines whether this parameter must be assigned a value, or this parameter must not present in the generated configuration.

In addition to the conditional restrictions, several additional constraints have to be ensured by means of forbidden configuration file.

- Ant Colony System only works with standard or Dorigo selectors.
- Ant Colony System must use local update.
- In external memory $topK$ has to be equal or less than the ant number m.
- Cunning ants must use MMAS as global update technique.

4.4 Testing

All parameter constraints are checked on two levels. Firstly, i-race scenario is set in such way, that it will only generate valid configurations by means of conditional parameters and specification of forbidden configurations. Secondly, configuration is wholly scanned in the framework itself, not only by checking valid parameter bounds, but also by verifying consistency between several parameters. In case of inconsistency, the framework returns a verbal exception response with error description.

Thorough unit and integrated testing is done in order to validate performance of the framework. Several auxiliary problem instances are used in order to simulate very specific artificial situations. Both white-box and black-box methodologies are used depending on a test case. For white-box mock objects are used to test specific method calls/returns by tested objects. This is done by using Mockito framework in combination with JUnit library. Random seed forcing is used to switch execution behavior from pseudo-random to deterministic, thus ensuring a concrete scenario.

Chapter 5

Experimental

[public samples description, some publicly know plots - YES, mention web-site]

[randomness of results even with same seed]

One must also take into account that, although deterministic behavior of the program is assured by forcing a random seed, one can not guarantee the same result of the program run even with the same configuration, since number of iterations, that will be effectively done within the specified time range, may vary.

[configuration of i-race]

[obtained result configuration for the framework]

[description of the configuration]

Chapter 6

Conclusion

[We planned ...]

[We implemeted ...]

[We analyzed ...]

[It will be very useful for such domains as ...]

Chapter 7

Literature

Appendix

The framework sources, automatic configuration setup, problem instances, analysis results and documentation are available on `https://github.com/ElderMayday/master`.

Table 1: Parameter space for the framework in context of VRP

Name	**Alias**	**Type**	**Possible values**	**Condition**
selector	-	c	(selector-standard, selector-dorigo, selector-maniezzo)	-
alpha	α	r	(0.1,5.0)	selector in c("selector-standard", "selector-dorigo")
beta	β	r	(0.1,5.0)	selector in c("selector-standard", "selector-dorigo")
alpha_maniezzo	α	r	(0.0,1.0)	selector == "selector-maniezzo"
dorigo_probability	q_0	r	(0.0,1.0)	selector == "selector-dorigo"
candidate_list	-	c	(candidate-yes, candidate-no)	-
candidate_ratio	r_{cand}	r	(0.0,1.0)	candidate_list == "candidate-yes"
global_update	-	c	(ant-s, ant-colony-s, min-max-s, rank-based-ant-s, elitist-ant-s, best-worst-ant-s)	-
local_update	-	c	(local-update-yes, local-update-no)	selector in c("selector-standard", "selector-dorigo")
iterated_greedy	-	c	(standard, external-memory, iterated-ants, cunning-ants)	-
ant_num	m	i	(1,50)	-
lupd_epsilon	ϵ	r	(0.0,1.0)	local_update == "local-update-yes"
lupd_tau0	τ_0	r	(0.0,1.0)	local_update == "local-update-yes"
local_search	-	c	(local-search-none, local-search-ils-twohalf, local-search-twohalf)	-
ils_iterations	$iter_{ils}$	i	(1,30)	local_search == "local-search-ils-twohalf"
evaporation_remains	$1-\rho$	r	(0.2,1.0)	-
ant_s_is_bounded	-	c	(ant-s-bounded-yes, ant-s-bounded-no)	global_update == "ant-s"

Name	Alias	Type	Possible values	Condition
ant_s_k	k	r	(1.0,10.0)	(global_update == "ant-s") & (ant_s_is_bounded == "ant-s-bounded-yes")
min_max_s_p_best	p_{best}	r	(0.01,1.0)	global_update == "min-max-s"
min_max_s_global_best	-	c	(min-max-s-global-best-yes, min-max-s-global-best-no)	global_update == "min-max-s"
min_max_s_global_iterations	$iter_{mg}$	i	(1,30)	(global_update == "min-max-s") & (min_max_s_global_best == "min-max-s-global-best-yes")
min_max_s_pts	-	c	(min-max-s-pts-yes, min-max-s-pts-no)	global_update == "min-max-s"
min_max_s_pts_iterations	$iter_{pts}$	i	(1,30)	(global_update == "min-max-s") & (min_max_s_pts == "min-max-s-pts-yes")
pts_lambda	λ	r	(0.0,1.0)	(global_update == "min-max-s") & (min_max_s_pts == "min-max-s-pts-yes")
pts_ratio	Ω'	r	(0.0,1.0)	(global_update == "min-max-s") & (min_max_s_pts == "min-max-s-pts-yes")
pts_delta	δ	r	(0.0,1.0)	(global_update == "min-max-s") & (min_max_s_pts == "min-max-s-pts-yes")
ras_w	w	i	(1,30)	global_update == "rank-based-ant-s"
ras_is_bounded	-	c	(ras-bounded-yes, ras-bounded-no)	global_update == "rank-based-ant-s"
eas_m_elite	m_{elite}	i	(1, 100)	global_update == "elitist-ant-s"
eas_is_bounded	-	c	(eas-bounded-yes, eas-bounded-no)	global_update == "elitist-ant-s"
bwas_mutation_probability	p_{mut}	r	(0.0,1.0)	global_update == "best-worst-ant-s"
reinitialization	-	c	(reinitialization-yes, reinitialization-no)	-
reinitialization_time	t_{reinit}	i	(0.01,0.2)	reinitialization == "reinitialization-yes"

Name	**Alias**	**Type**	**Possible values**	**Condition**
top_k	*topK*	i	(1,5)	iterated_greedy == "external-memory"
memory_size	*mem*	i	(10,30)	iterated_greedy == "external-memory"
tournament_selector_size	*tSize*	i	(2,10)	iterated_greedy == "external-memory"
criteria	-	c	(iterated-criteria-best, iterated-criteria-probabilistic)	iterated_greedy == "iterated-ants"
probabilistic_best	p_{iter}	r	(0.0,1.0)	(iterated_greedy == "iterated-ants") & (criteria == "iterated-criteria-probabilistic")
destruction_probability	p_{destr}	r	(0.0,1.0)	iterated_greedy in c("external-memory", "iterated-ants", "cunning-ants")