CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF BACHELOR’S THESIS

Title:	Vehicle routing problem with time windows and its solving methods
Student:	Martin Macho
Supervisor:	RNDr. Tomáš Valla, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2016/17

Instructions

The topic of this thesis is a "vehicle routing problem (VRP)": Given a graph G with a set of requests on vertices, a set of vehicles is dispatched to satisfy these requests. Several variants of this problem have been considered in the past, most of them leading to NP-complete problems. Therefore, we have to use different approaches in order to track the time complexity of the problem, namely heuristic-based and approximation-based. In particular, we will mostly concentrate on the version of the problem called "vehicle routing problem with time windows (VRPTW)".
The goal of the thesis is to:
1) search and study related papers, with both heuristic and exact approaches,
2) implement some of these methods and perform benchmarks on a testing dataset,
3) try to develop new solving methods for VRPTW or adapt existing methods from other versions of VRP to VRPTW.

References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D.	prof. Ing. Pavel Tvrdík, CSc.
Head of Department	Dean

Prague January 6, 2016

Czech Technical University in Prague

Faculty of Information Technology

Department of Theoretical Computer Science

Bachelor's thesis

Vehicle routing problem with time windows and its solving methods

Martin Macho

Supervisor: RNDr. Tomáš Valla, Ph.D.

30th June 2017

Acknowledgements

I would like to thank the friends for their support and my thesis supervisor, Tomáš Valla, for his patience, his time and for his advice.

Declaration

In Prague on 30th June 2017 . . . .

Citation of this thesis

Macho, Martin. *Vehicle routing problem with time windows and its solving methods*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

Tato práce studuje Vehicle Routing Problem (VRP) a jeho varianty. Pro verzi s časovými okénky implementujeme varianty používaných algoritmů a navhrneme a implementujeme modifikaci genetického algoritmu pro řešení problémů lokálního prohledávání. Na dvou sadách instancí provedeme měření a porovnání těchto algoritmů s nejlepšími známými výsledky.

Klíčová slova optimalizace rozvozových tras, lokální prohledávání, optimalizace, VRPTW, VRP, TSP

Abstract

This thesis studies Vehicle Routing Problem (VRP) and its variants. We implement local search based algorithms for time windows variant of the problem (VRPTW) and we propose and implement a modification of genetic algorithm for solving local search based problems. We test our implementation on two sets of benchmarks and we compare our results to best-known solutions.

Keywords vehicle routing problem, local search, optimization, VRPTW, VRP, TSP

Contents

List of Figures

Chapter 1

Introduction

The vehicle routing problem (VRP) is a class of combinatorial optimization and integer programming problems. It was first published by Dantzig and Ramser [2] in 1959, and since then it was studied very extensively, because the problem is easy to define and is very hard to solve, and it may have huge impact on the costs of transportation, logistics and distribution. The problem's goal is to minimize solution cost and to serve a number of customers by a fleet of vehicles. The cost can be traveled distance or time needed for serving the customers. New variants of the problem can be made by adding various constraints.

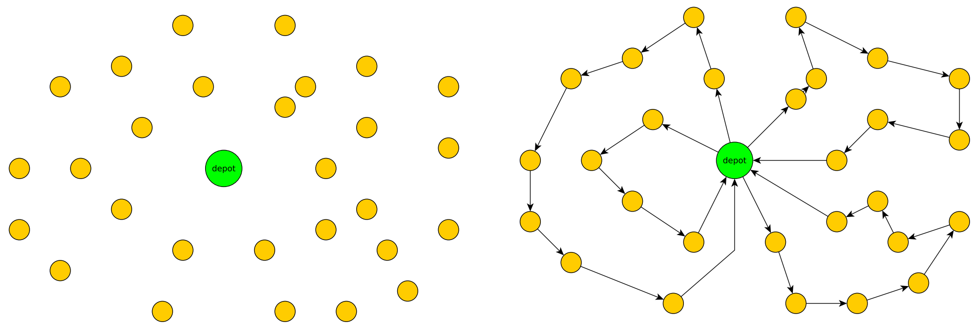

Figure 1.1: an example of VRP problem and possible solution

1.1 Vehicle routing problem definition

Let G be oriented graph.

- $V = \{v_0, v_1, \ldots, v_n\}$ is a set of vertices of graph G.
- v_0 is a depot.

- $V_1 = \{v_1, \dots, v_n\} \subset V$ is a set of vertices representing customers.
- $E = \{(v_i, v_j) | v_i, v_j \in V, i \neq j\}$ is an arc set.
- C is a matrix of non-negative costs c_{ij} between customers v_i and v_j.
- m is the number of vehicles (all identical). One route is assigned to each vehicle.
- Route $\mathcal{R}_i$ is a circuit which represents a route for vehicle i with the length $r_i = |R_i|$.
- $\mathcal{R} = \mathcal{R}_1, \dots, \mathcal{R}_m$ is a set of routes.
- Solution $\mathcal{S}$ is the set of routes.

Feasible solution is a solution, which has all vertices except v_0 assigned to exactly one route.

Cost P of the route $r = \mathcal{R}_i$ is sum of the arcs between every adjoining vertices.

$$P(r) = P_r = (\sum_{k=1}^{|r|-1} c_{(r_k)(r_{k+1})}) + c_{(r_{|r|})(r_1)}$$

Price of the Solution S is sum of every route in S.

$$P(\mathcal{S}) = P_{\mathcal{S}} = \sum_{i=1}^{m} P(\mathcal{R}_i)$$

Goal of the problem is to find $\mathcal{S}$ as $P_{\mathcal{S}}$ is minimal and $\mathcal{S}$ is feasible.

1.2 Problem variants

In this section, we will describe the best-known VRP variants. New variant is usually created by adding some constraint, which is taken from practical application (e. g. vehicle capacity, customers time windows or multiple depots), or by combining existing variants.

1.2.1 Capacitated VRP

Capacitated vehicle routing problem (CVRP) is VRP extended by capacity constraint. Every vehicle must have uniform capacity, and each customer have assigned demand, which we need to fulfill.

Let $Q = (q_0, q_1, \dots, q_n)$ be a vector of the customer demand and let F be a capacity of vehicles which are all identical.

Demand D **of the route** $r = \mathcal{R}_i$ is a sum of the customers demands in the route r.

$$D(r) = \sum_{k=1}^{|r|} d_k$$

For CVRP, we need to extend definition of the **feasible solution** to check capacity constraint: **Feasible solution (CVRP)** is a solution, which has all vertices except v_0 assigned to exactly one route and demand of all routes D_i is smaller or equal to car capacity F.

$$\forall r \in \mathcal{S} : D(r) \leq F$$

1.2.2 Multiple Depot VRP

Multiple depot vehicle routing problem (MDVRP) is VRP variant with more than one depot. Vehicles must start and end in the same depot, or they can start in one depot and end in another one (depends on the problem variant). This variant can be simplified to classic VRP variant by clustering customers and then solved for each depot separately.

1.2.3 Split Delivery VRP

Split delivery vehicle routing problem (SDVRP) is CVRP variant with relaxed delivery constraint – customers can be served by more than one vehicle (but still must be served completely). This variant can be useful when customers have large demands which are close to or over the vehicle capacity.

1.2.4 VRP with pick-up and delivering

Vehicle routing problem with pick-up and delivery (VRPPD) is CVRP variant in which customers may also return some commodities, not only accept them. Any picked-up commodities are returned to the depot, this variant does not permit deliveries between customers. In this variant it is necessary to take into account, that customers demands have to fit into free capacity of the vehicle.

1.2.5 VRP with time windows

Vehicle routing problem with windows (VRPTW) is CVRP variant extended by time constraints. The depot and each customer have assigned interval [a, b] that we call a time window and service time. There are two subvariants of the VRPTW – variant with soft time windows and variant with hard time windows. In the variant with hard time windows, vehicles must depart from the depot after the opening time, they must service customers in their time windows (the vehicle can arrive earlier and then wait for opening time, but the customer must be served before its closing time) and return to the depot before the depot's closing time. In the variant with soft time windows, the vehicles are permitted to arrive slightly later, but for penalty. In this thesis, we will focus on VRPTW with hard time windows.

1.3 Vehicle routing problem with time windows definition

Our formulation is based, as Kang[3] definition is, upon the model defined by Solomon[4]. Every customer $i \in \{1, 2, \ldots, N\}$ has assigned demand q_i and a service interval $[e_i, l_i]$ – vehicle must arrive before l_i and if the vehicle arrives before e_i, it must wait for w_i.

$$\min \sum_{i=0}^{N} \sum_{j=0}^{N} \sum_{k=1}^{m} c_{ij} \chi_{ijk} \tag{1}$$

$$\sum_{j=1}^{m} \chi_{ijk} = \sum_{j=1}^{N} \chi_{jik} = 1 \quad i = 0, k \in \{1, 2, \ldots, m\} \tag{2}$$

$$\sum_{k=1}^{m} \sum_{j=0}^{N} \chi_{ijk} = 1 \quad i \in \{1, \ldots, N\} \tag{3}$$

$$\sum_{i=0}^{N} q_i \sum_{j=0}^{N} \chi_{ijk} \leq F \quad k \in \{1, 2, \ldots, m\} \tag{4}$$

$$\sum_{k=1}^{m} \sum_{i=0}^{N} \chi_{ijk}(b_i + s_i + t_{ij} + w_j) = b_j, j \in \{1, 2, \ldots, N\} \tag{5}$$

$$e_i \leq (a_i + w_i) \leq l_i \quad i \in N \tag{6}$$

Where c_{ij} is a travelled distance between nodes i and j, t_{ij} is a travel time between nodes i and j, a_i is arrival time to node i, b_i is time to begin serving customer i and s_i is time needed to serve customer i. If vehicle k travels directly from node i to node j, $\chi_{ijk} = 1$. Otherwise $\chi_{ijk} = 0$.

Our goal is to find a solution $\mathcal{S}$ as $P_{\mathcal{S}}$ is minimal (1) and $\mathcal{S}$ is feasible. Each route must start and end at the depot (2) and vehicle capacity cannot be exceeded by sum of the route demands (4). Each customer must be served once and only once (3) within its service interval (5) (6).

Chapter 2

State-of-the-art

2.1 Solving methods

There are several approaches how vehicle routing problems can be solved.

2.1.1 Exact methods

Exact algorithms explore a whole state space and try to find the best solution. If the number of customers is smaller, i. e. dozens, exact methods can be used to optimally solve the problem, but because the problem complexity is very high for bigger instances, exact approach can not solve them. Due to this fact, we only describe representative of the category, but we will not implement it.

2.1.1.1 Branch and bound

Branch and bound algorithm is similar to brute force algorithm, but unlike brute force algorithm, which is designed to explore each solution, branch and bound algorithm explores only subset of solutions. Let the set of solutions form a tree. The algorithm explores every path from the root to leaf, but whenever the algorithm should branch, estimated lower and upper bounds are checked and if current solution does not fit, the solution is discarded and the algorithm explores next path.

2.1.2 Heuristics

Heuristics are problem-dependent techniques. They are usually adapted for the problem and they often take advantage of problem specifics, but as they are often too greedy, they usually get trapped in local optimum. Heuristics usually do not find optimal solution, but are faster than exact methods and found solution has usually moderate quality. We can categorize heuristics into two groups - constructive methods and improvement methods.

2.1.2.1 Savings algorithm

The savings algorithm is constructive method and was introduced in 1964[5] by Clarke and Wright and it is one of the best-known heuristics for solving VRP. Algorithm was built on assumption, that customers who are nearby is advantageous to handle by one vehicle. Firstly, the algorithm calculates savings between customers and assigns each customer to his own route. Then customers are chosen by the biggest savings and if it is possible, they are assigned to one common route. The algorithm has low computation complexity $\mathcal{O}(n^2)$ and is usually used to construct an initial solution, which is improved by other techniques, such as hill climbing. We describe the savings algorithm in detail in Chapter 3.

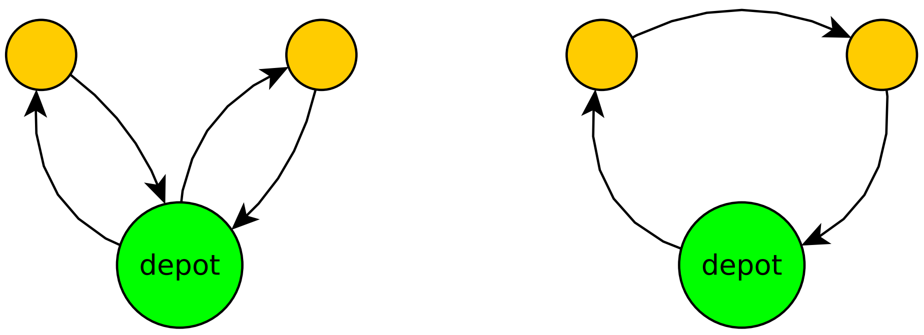

Figure 2.1: Before and after the savings merge

2.1.2.2 Route first, cluster second

Route first, cluster second algorithm is constructive method and is made of two phases. In the first phase all customers are on one route, which creates big travelling salesman problem (TSP) instance. After solving TSP, there is the second phase, where the route is split and customers are clustered into feasible routes.

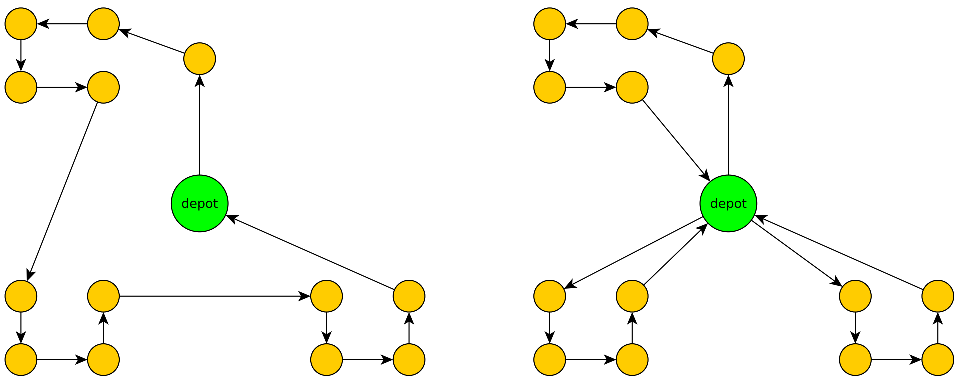

Figure 2.2: Route construction and clusterization

2.1.3 Metaheuristics

Metaheuristics, unlike heuristics, are problem independent. They do not take advantage of problem specifics and therefore they are more versatile and can be used as black-boxes. As they are not so greedy (they do not always take the best solution, they can even take worse solution than found optimum), they can get out of local optimum and find a better solution.

To properly understand how metaheuristics can work, we need to define *exploration* and *exploitation* and difference between them.

2.1.3.1 Exploitation and Exploration

Exploitation is exploring a limited region of the search space. We hope in improvement of a promising solution $\mathcal{S}$ (in metaheuristics we call it candidate solution, because it is candidating for the best solution) that we already have. In *exploitation*, our primary focus is searching in the vicinity of the candidate solution $\mathcal{S}$.

The opposite of *exploitation* is *exploration*. *Exploration* is exploring a much larger portion of the search space. We hope in finding another promising solution $\mathcal{S}$ that we can try to improve.

When solving an optimization problem with metaheuristics methods, we are trying to find optimal ratio between *exploitation* and *exploration*. If we had large *exploitation* and small *exploration*, our algorithm would probably get stuck in local optimum. In other hand if we would have large *exploration* and small *exploitation*, our algorithm will probably never reach neither local nor global optimum, because it will be still exploring the whole state space.

2.1.3.2 Hill climbing

Hill climbing is the simplest metaheuristic method for local search. It begins with (usually randomly generated) initial solution S_0. The method explores a neighbourhood of the solution and picks the best found solution as the next point to explore. Advantage of this method is its simplicity, but *hill climbing* often gets stuck in a local optimum which can be far away from global optimum. There are several techniques how to deal with getting stuck in local optimum, for example random restart whenever the method gets stuck, or use of the advanced methods (some of them are listed below).

Algorithm 1 Hill climbing algorithm pseudocode

```
1: create an initial solution $bestSolution$ (usually random)
2: while termination condition not met do
3:     $temporarySolution$ := getFeasibleNeighbour($bestSolution$)
4:     if getPrice($temporarySolution$) $\leq$ getPrice($bestSolution$) then
5:         $bestSolution$ := $temporarySolution$
6: return $bestSolution$
```

2.1.3.3 Late acceptance hill climbing

Late acceptance hill climbing (LAHC) is relatively new improvement of *hill climbing* method. It was invented and presented by Yuri Bykov in 2008[6]. LAHC is an iterative search algorithm, which can accept worse candidate solution when found candidate solution is equal or better than candidate solution found N iterations before, where N is an input parameter for the algorithm and can be based on problem specifications. One of the major advantages of LAHC approach is the absence of a cooling schedule. This makes it significantly more robust than cooling-schedule based techniques (see Simulated annealing for more details). Also, LAHC is easy to implement and yet it is an effective searching algorithm. We will take a deeper look at it in Chapter 3).

Algorithm 2 Late acceptance hill climbing algorithm pseudocode

```
1: create a candidate solution $bestSolution$ (usually random)
2: $lastCosts[] =$ getPrice( $bestSolution$ )
3: $i := 0$
4: while termination condition not met do
5:     $tempSolution :=$ getFeasibleNeighbour($bestSolution$)
6:     if getPrice($tempSolution$) $\leq lastCosts[i \mod pL] \lor$ getPrice($tempSolution$) $\leq$ getPrice($bestSolution$) then
7:         $bestSolution := tempSolution$
8:     $lastCosts[i \mod pL] :=$ getPrice( $tempSolution$ )
9:     $i := i + 1$
10: return $bestSolution$
```

2.1.3.4 Step Counting Hill Climbing

Step Counting Hill Climbing (SCHC) is a new local search heuristic, invented and presented by Y. Bykov and S. Petrovic in 2013[7]. It is similar to LAHC – it is also improvement of Hill climbing method with ability to escape a local optimum. The cost of the current solution sets an upper acceptance bound for next pL steps, where pL is a single input parameter which is set by user, or it can be determined from the problem instance. There are also many ways how to count the steps which can lead into many variant of this method. Advantages of this method are that SCHC is even simpler than LAHC and yet may be even more effective.

Algorithm 3 Step Counting hill climbing algorithm pseudocode

```
1: create a candidate solution $bestSolution$ (usually random)
2: $lastCost =$ getPrice( $bestSolution$ )
3: $i := 0$
4: while termination condition not met do
5:     $tempSolution :=$ getFeasibleNeighbour($bestSolution$)
6:     if getPrice($tempSolution$) $\leq$ getPrice($bestSolution$) $\lor$ getPrice($tempSolution$) $\leq$ lastCost then
7:         $bestSolution := tempSolution$
8:         $i := i + 1$
9:     if $i \geq pL$ then
10:        $lastCost :=$ getPrice($bestSolution$)
11:        $i := 0$
12: return $bestSolution$
```

2.1.3.5 Tabu search

Tabu search is another meta heuristic for solving optimization problems. It was introduced by Fred Glover in 1986[8] and it is superimprosed on another heuristic (we will show this method on Hill climbing). *Tabu search* deals with getting stuck at local optimum by introducing a forbidding strategy. We can divide the forbidding strategies into three main categories – short term, intermediate term and long term[9].

Short term forbidding strategy is usually a memory (also called tabu-list), where we store recent operations, which can not be used until their expiration period passes. For example, when we swap nodes i and j in one route, we add record to tabu-list and it prevents the algorithm from using i and j in any other operation for a certain time. Intermediate term forbidding strategy is a set of rules which bias the search towards promising areas. Long term forbidding strategy is a set of rules which provide diversity in search, for example random restarts when the algorithm gets stuck.

Algorithm 4 Tabu search algorithm pseudocode

```
1: $tabuList$ := initTabuList()
2: create an initial solution $bestSolution$ (usually random)
3: while termination condition not met do
4:     $temporarySolution$ := getFeasibleNeighbour($bestSolution$)
5:     if getPrice($temporarySolution$) $\leq$ getPrice($bestSolution$) $\wedge$ !in($tabuList$, $temporarySolution$) then
6:         $bestSolution$ := $temporarySolution$
7:     addToTabuList($tabuList$, $temporarySolution$)
8:     deleteExpired($tabuList$)
9: return $bestSolution$
```

2.1.3.6 Simulated annealing

Simulated annealing is inspired by physical annealing process in metals. The principle of annealing process is heating metal to high temperature and then slowly cooling the metal according to predefined schedule. At the high temperature, atoms in the metal leave crystalline lattice and they are randomly distributed over the metal and within the cooling process the atoms place themselves in a pattern that corresponds to the global energy minimum of a perfect crystal.

Similar principle is used in *simulated annealing* method. It was invented and presented in 1983[10] by S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi. At the beginning, the method starts with high temperature (large *exploration*) to find the most promising solution in the whole state space and as time goes, temperature slowly goes lower and lower (large *exploitation*) to locally improve solution that was found. The way how temperature gets cooler is called cooling schedule or strategy. Choosing a cooling schedule is a hard part of the algorithm because there is no universal schedule that performs well on all problems. Disadvantage of the simulated annealing (and cooling

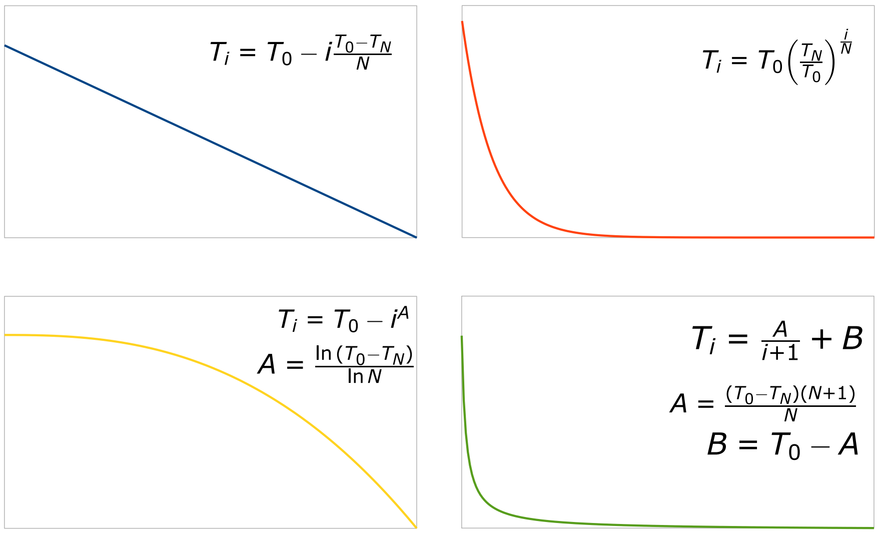

Figure 2.3: Example of cooling schedules [1]. T_i is the temperature for cycle i, where i increases from 1 to N. The initial and final temperatures, T_0 and T_N respectively, are determined by the user, as is N.

schedules in general) is that we need to set the initial parameters for proper temperature calculating - initial temperature, final temperature and estimate of total iterations, which we usually do not know. Example of some cooling schedules is on Figure 2.3.

2.1.3.7 Genetic algorithm

Genetic algorithm (GA) is a population based meta-heuristic approach for solving problems that do not have usable exact algorithm. Genetic algorithm was introduced by J. H. Holland in 1975[11] and is inspired, as name suggests, by genetic evolution. The algorithm keeps a set of solutions which is called population. Solution from the population is called candidate, as it is trying to be the best solution. In order to use genetic algorithm, we need function which returns information about solution quality. In GA terminology, the function is called a fitness function and e. g. for vehicle routing problems, it can be total travel distance. The algorithm was designed to solve problems where solution can be encoded as binary vector, thus we need to adapt it for vehicle routing problem (details can be found in Section 3.4.1).

General GA has four phases - initialization, selection, crossover and mutation, but there are also variants of GA where either mutation or selection is missing. Ilustration of how the algorithm works can be found in Figure 2.5.

Initialization is process, where starting population is created from candidates. The candidates can be generated randomly or with some heuristic.

Selection is process, where candidates are selected from current population for next operations. Selected candidates are usually chosen by their fitness function, in order to keep population strong. Example of the selection is tournament selection - it randomly selects x candidates and from the selected candidates chooses the best one.

Crossover is process, where candidates are combined to generate another candidate. This process usually helps to keep well solved parts of a solution in the population. Example of the crossover is one-point crossover - it takes two candidates $a, b \in \{0, 1\}$ (binary vectors), splits them by randomly generated point i into four binary vectors $(a_1, a_2, \ldots, a_{i-1}), (a_i, a_{i+1}, \ldots, a_n), (b_1, b_2, \ldots, b_{i-1})$ and $(b_i, b_{i+1}, \ldots, b_n)$ and then combines splitted parts together into solutions $(a_1, a_2, \ldots, a_{i-1}, b_i, b_{i+1}, \ldots, b_n), (b_1, b_2, \ldots, b_{i-1}, a_i, a_{i+1}, \ldots, a_n)$.

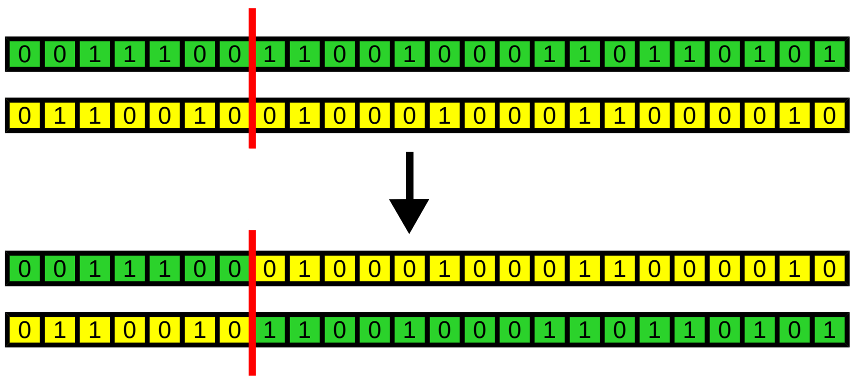

Figure 2.4: Example of crossover operation

Mutation is process, where one candidate is, usually slightly, modified. Example of the mutation can be a bit-flip - it takes one candidate $a \in \{0, 1\}$ (binary vector). For each bit in a, it chooses randomly (with probability p) whether the bit should be negated or not.

Algorithm 5 Bit-flip operation for binary vector

```
1: function bitFlip(solution, p)
2:     $a := \text{Copy}(solution)$
3:     for all $i$ in $a$ do
4:         if Random($p$) then
5:             Flip($a[i]$)
6:     return $a$
```

2.2 Benchmarks

We will test our algorithms on two groups of benchmarks. The first group of benchmarks is Solomon's benchmark[12], which is focused on various VRPTW subproblems. There are randomly generated samples (R1, R2), samples where customers are clustered (C1, C2) and combination of randomly generated and clustered (RC1, RC2) instances. We have chosen Solomon's benchmark, because it is well studied and optimal solutions are known, moreover the instances of the benchmark vary a lot (there are instances with very tight time windows, but also with time windows which are hardly constrainting). For the second group of benchmarks we have chosen Gehring and Homberger benchmark[13], which have instances with 200, 400, 600, 800 and 1000 customers. As Solomon's benchmark, Gehring and Homberger benchmark is well studied and are known best results (but not the optimal ones). We have chosen it for its size, we will benchmark 1000 customers variant for testing computational performance of our implementation.

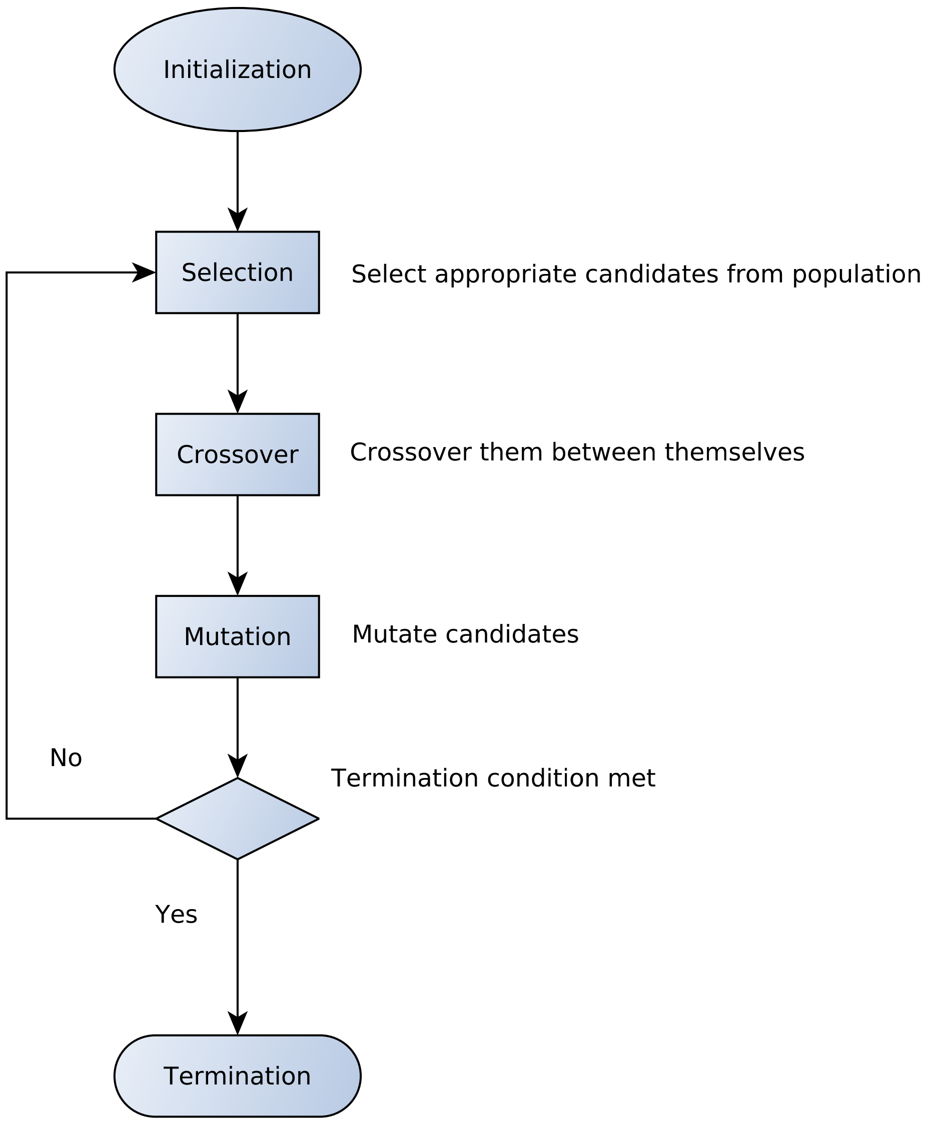
Initialization
Selection
Select appropriate candidates from population
Crossover
Crossover them between themselves
Mutation
Mutate candidates
No
Termination condition met
Yes
Termination

Figure 2.5: Genetic algorithm diagram

Chapter 3

Our approach

In this chapter we describe our approach, mainly we focus on state space search and neighbour generation. We also describe our proposal of Genetic Algorithm for VRPTW and our modification of Savings algorithm. But firstly, we describe our observation about problem solvability.

We have observed that every solvable instance of the VRPTW problem can be solved by assigning each customer into his own route (we are not limited by the number of vehicles). We can say that infeasible solution must break either capacity constraint or time window constraint. In case we have a solution where each customer is served by his own vehicle and the solution is infeasible then problem is insolvable, because either we can not serve a customer due to his high demand excessing a vehicle capacity, or we can not serve customer and then return back to depot in a given time windows. We can utilize this observation also for correcting infeasible solution. We can convert infeasible solution of solvable instance to feasible one by splitting route which breaks the constraints. If demand of the route exceeds vehicle capacity, we split the route into smaller routes which can fit the vehicle capacity. Similarly we can split routes for fulfilling time windows.

3.1 State space

We have chosen to move accross the feasible solutions only and that is achieved by a repairing function. The repairing function is based on the observartion, that every infeasible solution of the solvable instance of VRPTW can be converted to a feasible solution by splitting the routes into the multiple ones. Disadvantage of this aproach is an amount of routes created by repairing function, thus we need some mechanism to reduce the amount of routes. We have done this within neighbour generation.

Algorithm 6 Pseudocode of the repairing function

```
1: function repairSolution($solution$)
2:     $feasibleRoutes$ = []
3:     for all $route$ in $solution$ do
4:         if not RouteFeasible($route$) then
5:             continue
6:         $feasibleRoutes$ = merge(splitRouteToFeasible($route$), $feasibleRoutes$)
       return createSolutionFromRoutes($feasibleRoutes$)
```

3.2 Creating initial solution

We have implemented three algorithms for creating initial solution. The first one is randomly generated without any heuristic, the second one is savings algorithm as described in the Section 2.1.2.1, which is deterministic and uses heuristic. As the last algorithm we use randomized savings algorithm, which is the savings algorithm extended by randomness.

3.2.1 Random feasible solution

Our implementation of random initialization is similar to route first, cluster second algorithm. Firstly, we assign all customers into one route, then the customers are shuffled and route is split into many feasible routes by repair function (clusterization). This initialization method produces a candidate solution with higher cost, but with high portion of randomness, which other methods can utilize.

Algorithm 7 Creating random feasible solution

```
1: $solution$ := createSolution()
2: $route$ := createRoute()
3: $route$ := assignCustomersToRoute($route$)
4: $route$ := shuffleRoute($route$)
5: addRoutesToSolution(splitRouteToFeasible($route$))
6:  return $solution$
```

3.2.2 Savings algorithm

Another initialization method we use is a savings algorithm. We have chosen this algorithm because it is a deterministic and it produces a moderate starting solution (compared to random initialization). Also, computational complexity is relatively low – $\mathcal{O}(N^2)$ is needed for calculating saves and $\mathcal{O}(N \log N)$ is needed for sorting (it depends on used sorting algorithm, $\mathcal{O}(N \log N)$ is computational complexity of average quick sort run), which give us $\mathcal{O}(N^2)$ in total. Disadvantage of this algorithm as initialization method may be lack of randomness, which is important for exploring large portion of the state space, especially in a population based methods as Genetic algorithm is. We try to deal with this disadvantage in a randomized savings algorithm.

Algorithm 8 Savings algorithm

```
1: Initialize each customer in his own route
2: for all $i$ in $customers$ do
3:     for all $j$ in $customers$ do
4:         $s[ij].save := c_{i0} + c_{0j} - c_{ij}$
5:         $s[ij].from := i$
6:         $s[ij].to := j$
7: Sort $s$ in descending order
8: for all $x$ in $s$ do
9:     if $x.from.next = depot$ & $x.to.prev = depot$ then
10:        connectRoutes($x.from, x.to$)
11: return distinctRoutes($x$)
```

3.2.3 Randomized savings algorithm

We have developed a randomized savings algorithm in faith that we could combine advantages of both random initialization and savings algorithm. We have added a variance parameter, which we use to add randomness to saving calculation.

Algorithm 9 Randomized savings algorithm

```
1: function RandomizedSavings(variance)
2:     Initialize each customer in his own route
3:     for all $i$ in $customers$ do
4:         for all $j$ in $customers$ do
5:             $s[ij].save := (c_{i0} + c_{0j} - c_{ij})$ * (1 + Rand($-variance, variance$)
6:             $s[ij].from := i$
7:             $s[ij].to := j$
8:     Sort $s$ in descending order
9:     for all $x$ in $s$ do
10:        if $x.from.next = depot$ & $x.to.prev = depot$ then
11:            connectRoutes($x.from, x.to$)
12:    return distinctRoutes($x$)
```

3.3 Neighbours generation

In order to do a state space search, we need to be able to generate neighbour of the candidate solution. There are many ways how we can generate a neighbour from the candidate solution, all of them are based on moving one or more customers from one route to another one. We describe some of them below, but before we can describe operations used for neighbour generation, we need to define a predecessor and a successor of a customer.

Predecessor i_p is a customer from route r which is served right before the customer i is served and before applying a neighbour operation. Similarly, *successor* i_s is a customer from route r which is server right after the customer i is served and before applying a neighbour operation.

We have used three methods for neighbour generation – two of them, *relocate* and *exchange*, are primarily used to exploit the state space because they can only move one or two customers at once. The last method, *2-opt-swap*, can move many customers at once, thus we use it to explore the state space.

3.3.1 Relocate

Relocate procedure takes two different customers a, $b \in \{1, ..., N\}$ and moves the first customer after the second customer as is listed on the Figures 3.1 and 3.2. Having a sub-route (a_p, a, a_s) and a sub-route (b_p, b, b_s), after the procedure relocate we will get sub-routes (a_p, a_s) and (b_p, b, a, b_s). We have implemented this procedure in both intra-route and inter-route variant.

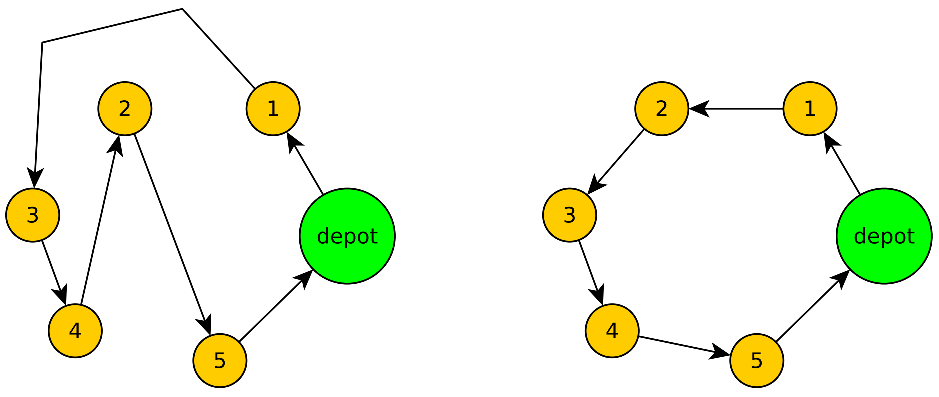

Figure 3.1: Before and after the procedure relocate (relocated the customer 2 after the customer 1).

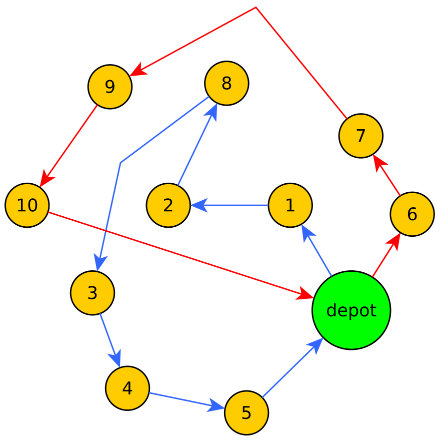

Figure 3.2: Before and after the procedure relocate (relocated the customer 8 after the customer 7).

3.3.2 Exchange

Exchange procedure takes two different customers $a, b \in \{1, ..., N\}$ and exchanges them as listed on Figures 3.3 as 3.4. Having a sub-routes (a_p, a, a_s) and (b_p, b, b_s), after the procedure relocate we will get subroutes (a_p, b, a_s) and (b_p, a, b_s). We have implemented this procedure in both intra-route and inter-route variant.

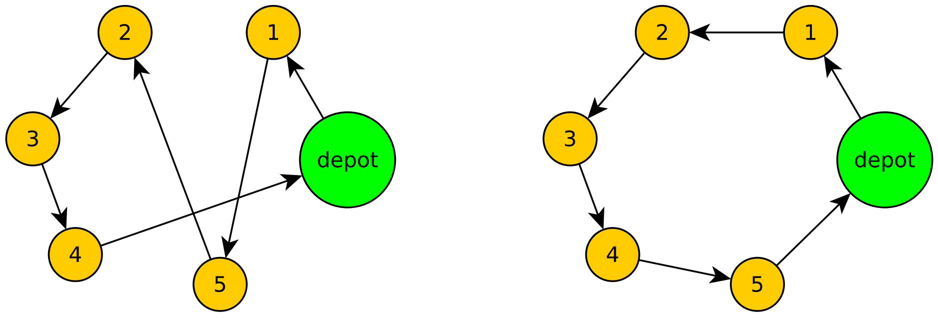

Figure 3.3: Before and after the procedure exchange (exchanged the customer 2 with the customer 5).

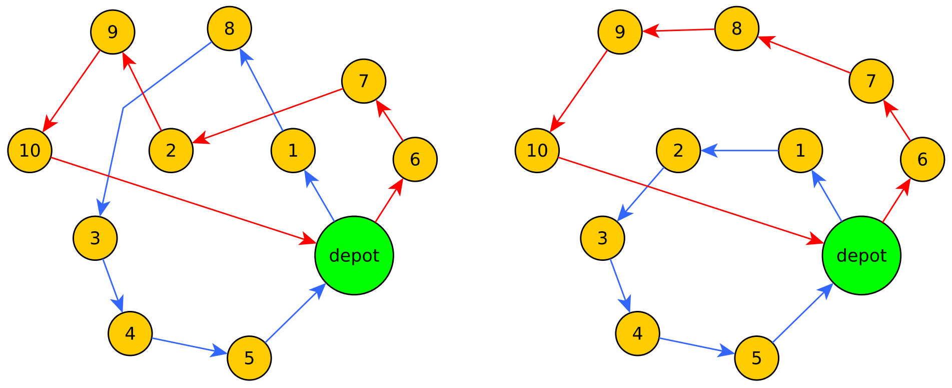

Figure 3.4: Before and after the procedure exchange (exchanged the customer 2 with the customer 8).

3.3.3 2-opt swap

Unlike previous two methods, which takes only 2 customers, 2-opt-swap can affect much more customers at once. Also, implementation of inter-route variant differs from implementation of intra-route variant.

Inter-route variant takes a subroute $(a_1, a_2, \cdots, a_k)$ from a route $(\cdots, a_{1_p}, a_1, \cdots, a_k, a_{k_s}, \cdots)$ and moves it to another route as listed on the Figure 3.5.

Intra-route variant takes a subroute $(a_1, a_2, \cdots, a_k)$ from route $(\cdots, a_{1_p}, a_1, \cdots, a_k, a_{k_s}, \cdots)$ and reverts it into a route $(\cdots, a_{1_p}, a_k, \cdots, a_1, a_{k_s}, \cdots)$ as listed on the Figure 3.6. Although we have implemented this procedure, we do not use it on small instances (instances with 100 and less customers), because we have got better results without using it.

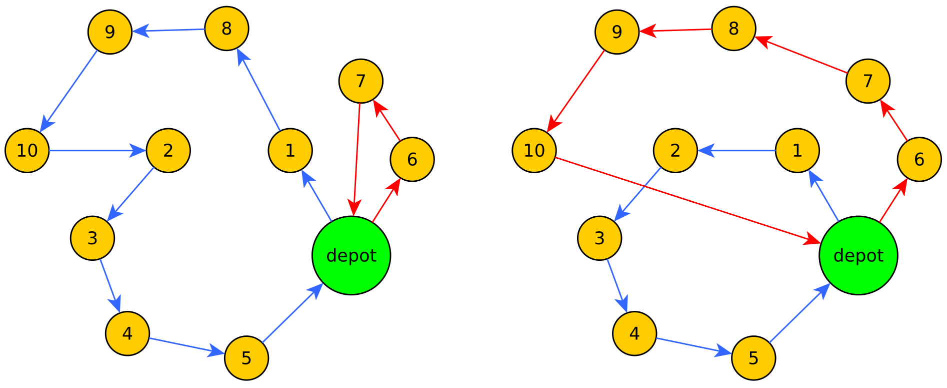

Figure 3.5: Before and after the inter-route variant of the procedure 2-opt swap, moved subroute $(8, 9, 10)$ into route $(6, 7, 8, 9, 10)$.

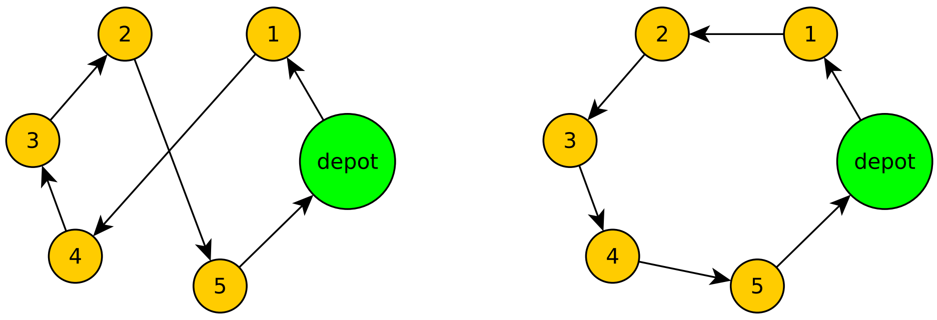

Figure 3.6: Before and after the intra-route variant of the procedure 2-opt swap, reverted subroute $(4, 3, 2)$.

3.4 Our approach

We have implemented five algorithms from the Chapter 2 – Savings, Hill Climbing, Step Counting Hill Climbing, Late Acceptance Hill Climbing and Genetic Algorithm.

3.4.1 Genetic algorithm

As we have mentioned in the Section 2.1.3.7, the Genetic algorithm was designed for solving problems where a solution can be encoded as a binary vector, but VRP solution cannot be effectively encoded as a binary vector because we need to preserve order in our routes. We propose a modification of *Genetic algorithm* that can solve every local search based problems. In our modification, we do not use *crossover* operation. We have also replaced *mutation* operation with neighbour generation, which is a very similar process.

Algorithm 10 Genetic algorithm

```
 1: function geneticAlgorithm($problem, populationSize$)
 2:     $population$ := initPopulation($populationSize$)
 3:     while terminationConditionNotMet() do
 4:         $newPopulation$ := createEmptyPopulation($populationSize$)
 5:         $bestCandidate$ := getBestCandidate($population$)
 6:         AddToPopulation($newPopulation$, bestCandidate)
 7:
 8:         for $i \in \{2, \ldots, populationSize\}$ do
 9:             $tmpSolution$ := Selection($population$)
10:             $tmpSolution$ := getFeasibleNeighbour($tmpSolution$)
11:             AddToPopulation($newPopulation$, $tmpSolution$)
12:         $population$ := $newPopulation$
13:
14:     return getBestCandidate($population$)
```

Chapter 4

Results

Algorithms implemented by us were tested on chosen instances from both above-mentioned benchmarks. Tests ran on a computer with Linux powered by Intel(R) Core(TM) i5-4590 (4 cores, 3.3 GHz) and 16 GB of RAM. From Solomon benchmark collection we chose instances R103, R211, C105, C205, CR105 and RC205 where every instance has 100 customers. Every instance was tested nine times for the period of ten seconds. LAHC and SCHC and their variants were tested progressively with 1, 10, 100, 400, 3000, 10000 and 100000 parameters. Genetic algorithm was tested for population sizes of 100, 400, 3000, 10000 and 100000 candidates. For each population size, we tested 1, 5, 10 and 50 rounds in the tournament selection.

From the second group of benchmarks, we chose C1, C2, R1, R2, RC1 a RC2 instances, where each has 1000 customers. Every instance was tested three times for one minute. LAHC and SCHC were with these benchmarks tested with the same parameters as in Solomon's benchmarks.

Results of tests were plotted into a bar chart, where each bar represents a ratio between our average result and the best-known result. For example, if our result is 20 % worse than the best-known result, the chart will show + 20 % and vice versa, if our result is 20 % better, the chart will show - 20 %. Algorithms with no prefix (LAHC, SCHC, hill-climbing, genetic-algorithm) are initialized randomly. Algorithms with the prefix s- are initialized with *Savings algorithm* and algorithms with the prefix rs- are initialized by *Randomized savings algorithm.*

4.1 Cost over time depending on parameters

In this section, we take a look on how algorithms behave with different parameters. We have chosen the RC211 instance of Solomon benchmark as the reference because we have the best results on it. As we can see below on the Figures 4.2 and 4.1, for the LAHC and SCHC the parameter affects convergence speed – the lower the parameter is, the faster the algorithm converges.

On the other hand, the algorithm can explore wider portion of state space with higher parameter value.

Unlike SCHC and LAHC, parameters in Genetic algorithm are more difficult to set as you can see on the Figure 4.3. Number of rounds in the tournament also affects convergence speed, but unlike LAHC and SCHC, higher value can lead to getting stuck at local optimum (as is shown on the Figure 4.3 for the parameter value 50).

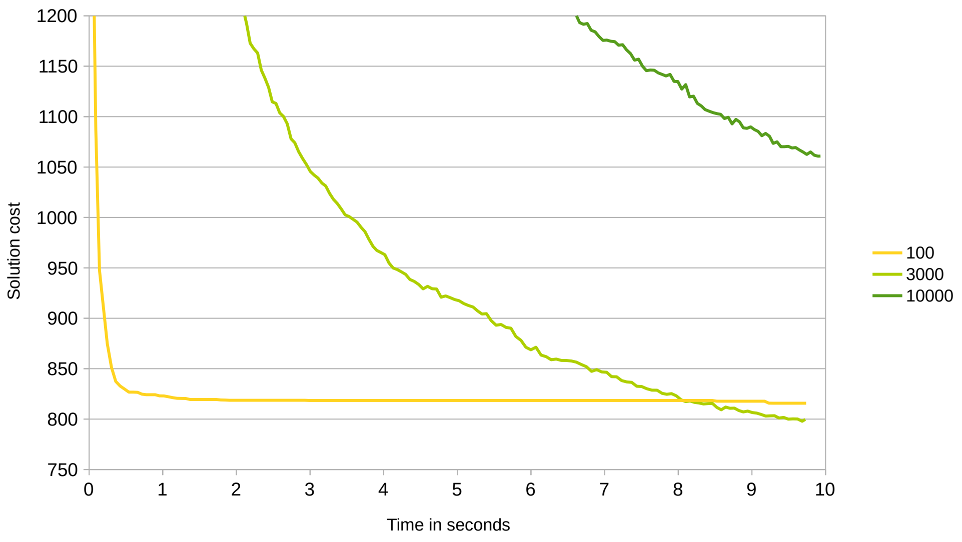

Figure 4.1: SCHC results over time depending on parameter

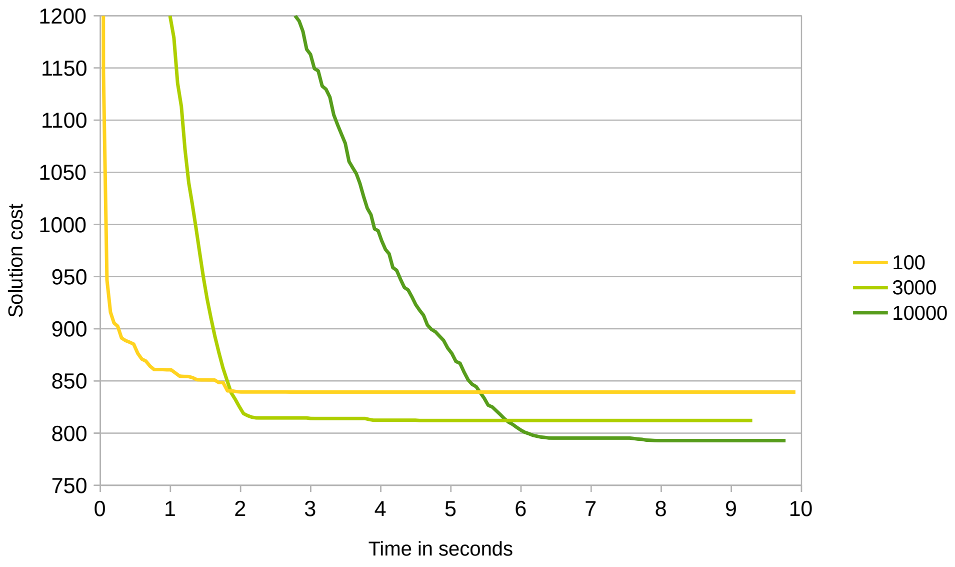

Figure 4.2: LAHC results over time depending on parameter

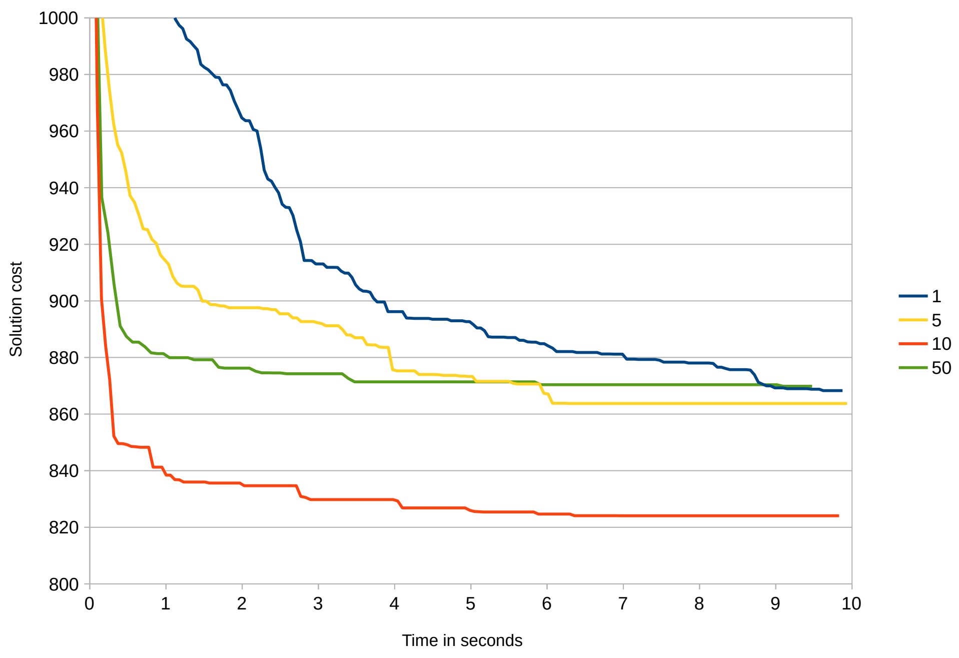

Figure 4.3: Genetic algorithm results over time depending on rounds in the tournament selection

4.2 Computional results

4.2.1 Solomon's benchmarks

On Solomon's benchmarks our algorithms brought satisfying results. LAHC, SCHC and Genetic algorithm had usually the same or better results than the best-known results on chosen instances. You can see result at Figures 4.4, 4.5, 4.6, 4.7, 4.8 and 4.9.

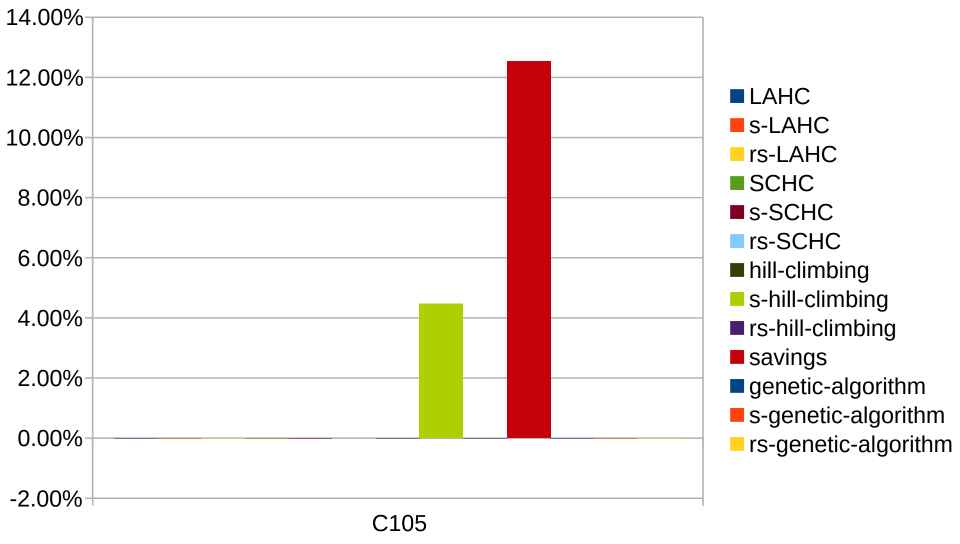

Figure 4.4: Results for instance C105

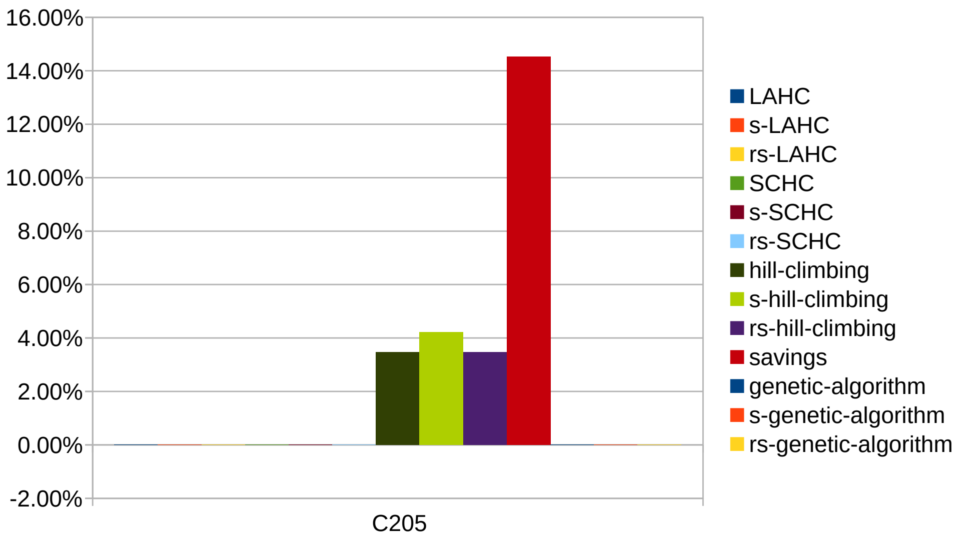

Figure 4.5: Results for instance C205

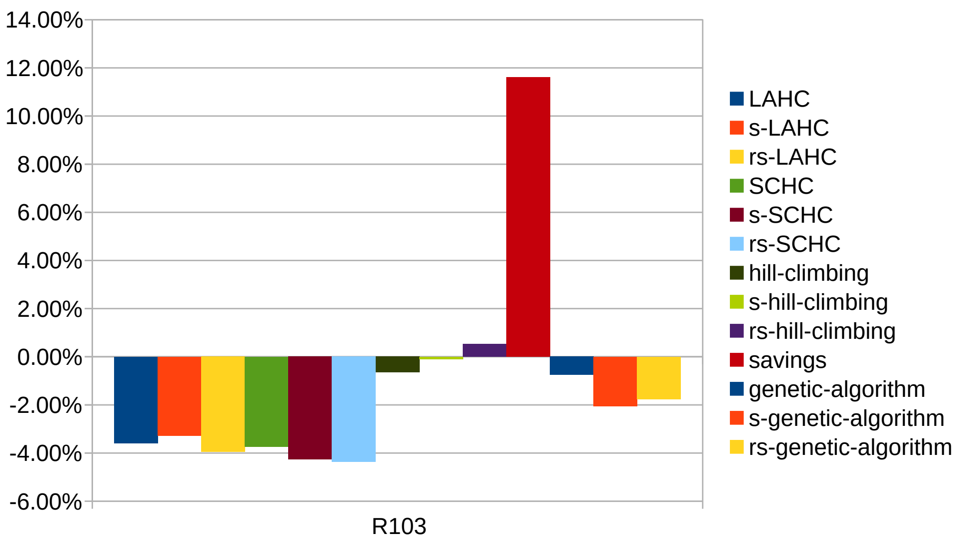

Figure 4.6: Results for instance R103

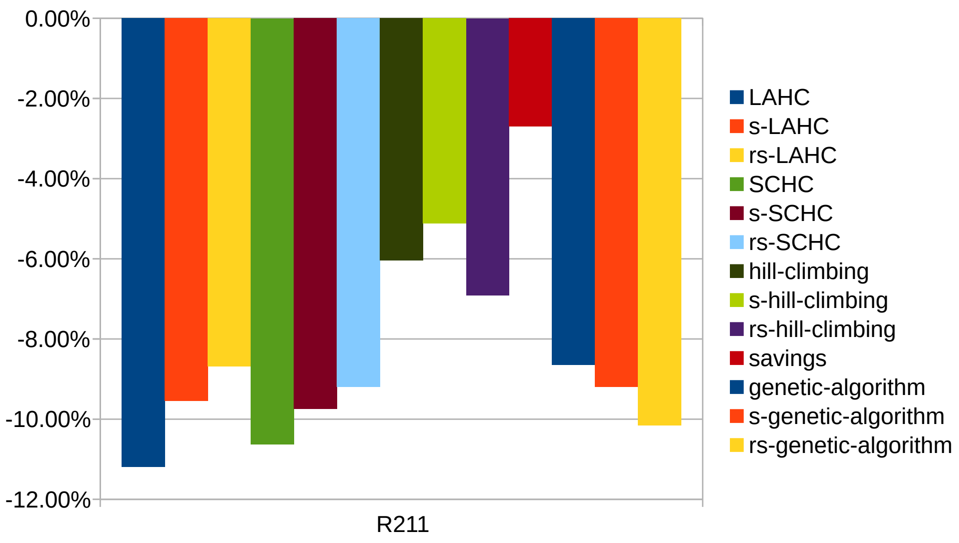

Figure 4.7: Results for instance R211

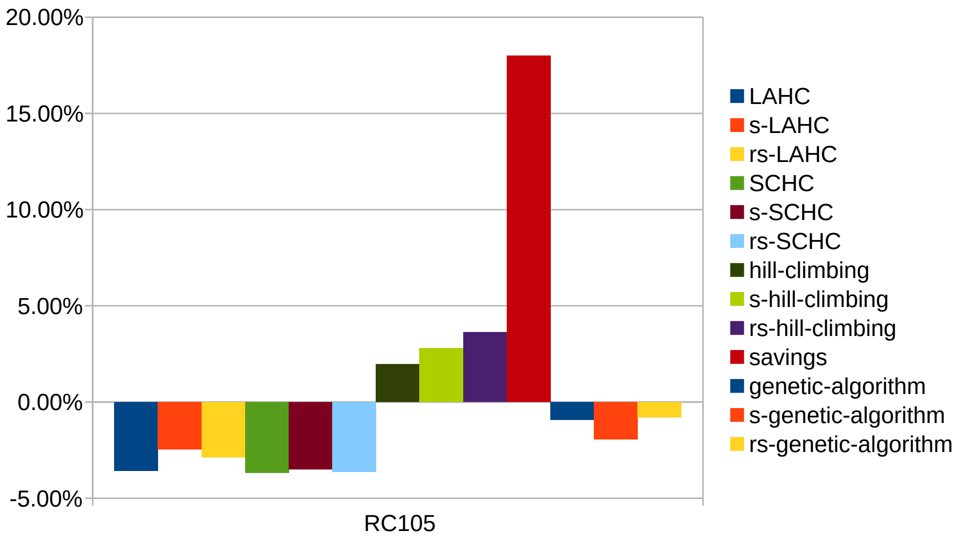

Figure 4.8: Results for instance RC105

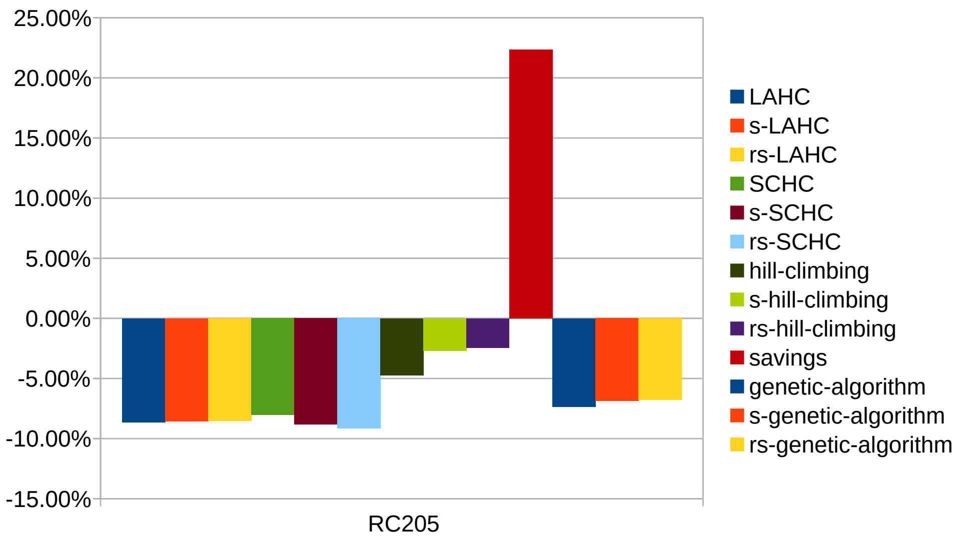

Figure 4.9: Results for instance RC205

4.2.2 Homberger's benchmarks

Our algorithms did not do so well on Homberger's benchmarks. It is obvious from the charts, that our modification of algorihms with initialization using Savings and Randomized Saving algorithm were in most cases better than their competitors that were using random initialization. You can see result at Figures 4.10, 4.11, 4.12, 4.13, 4.14 and 4.15.

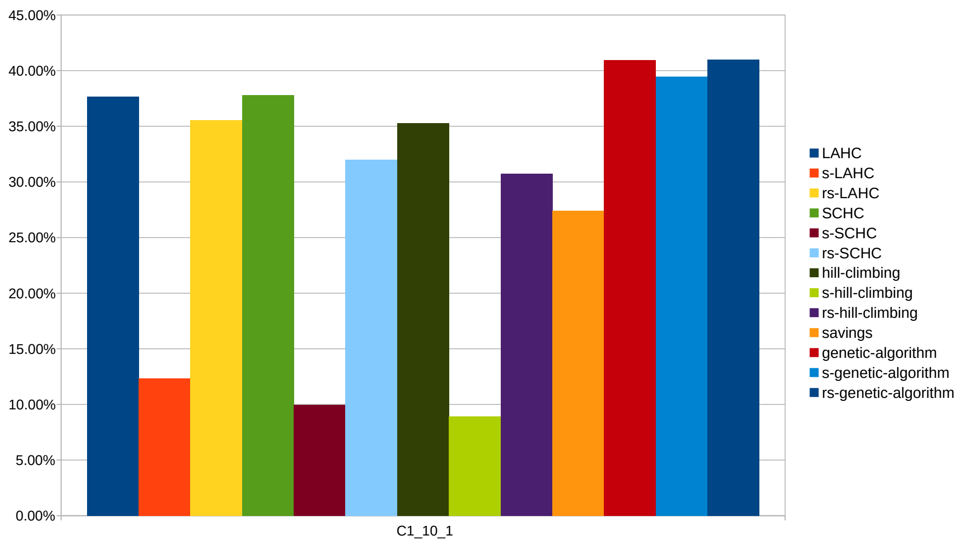

Figure 4.10: Results for instance C1_10_1

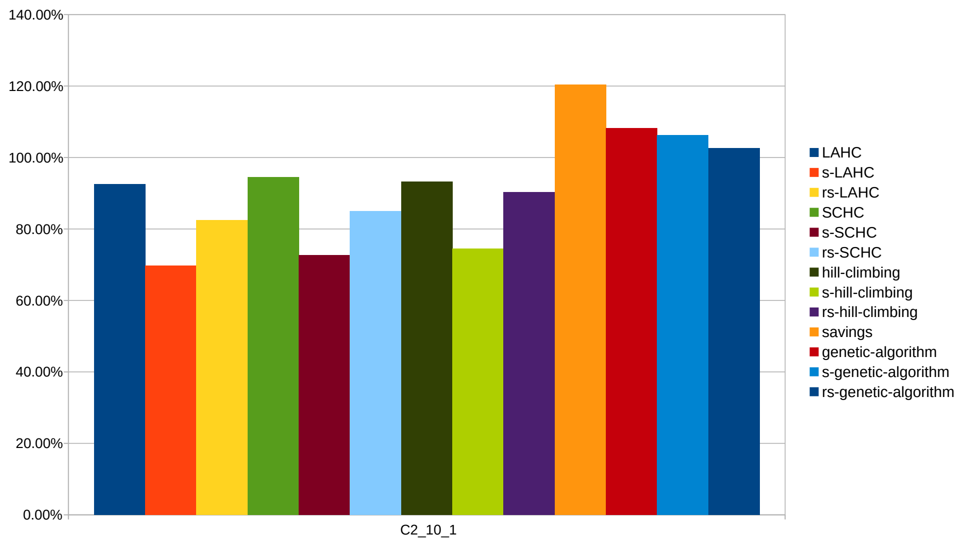

Figure 4.11: Results for instance C2_10_1

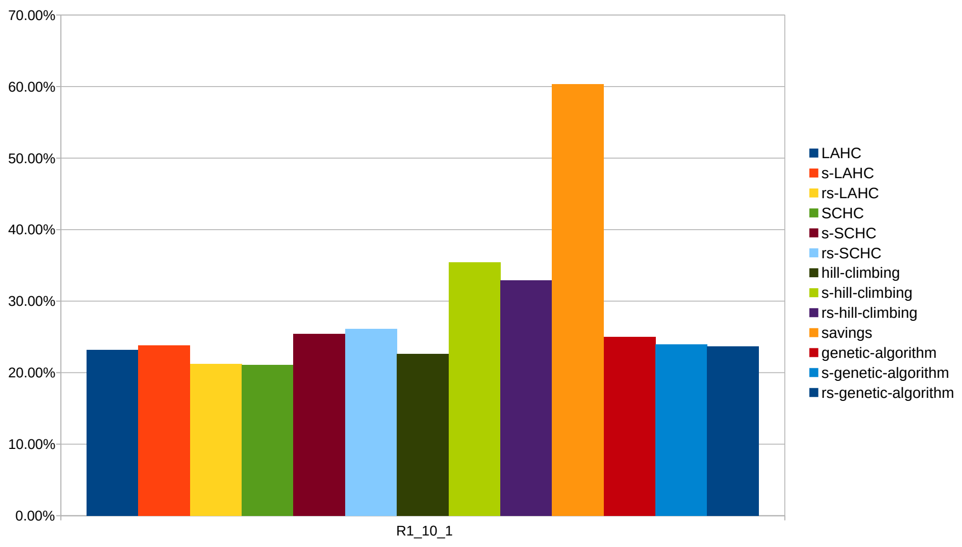

Figure 4.12: Results for instance R1_10_1

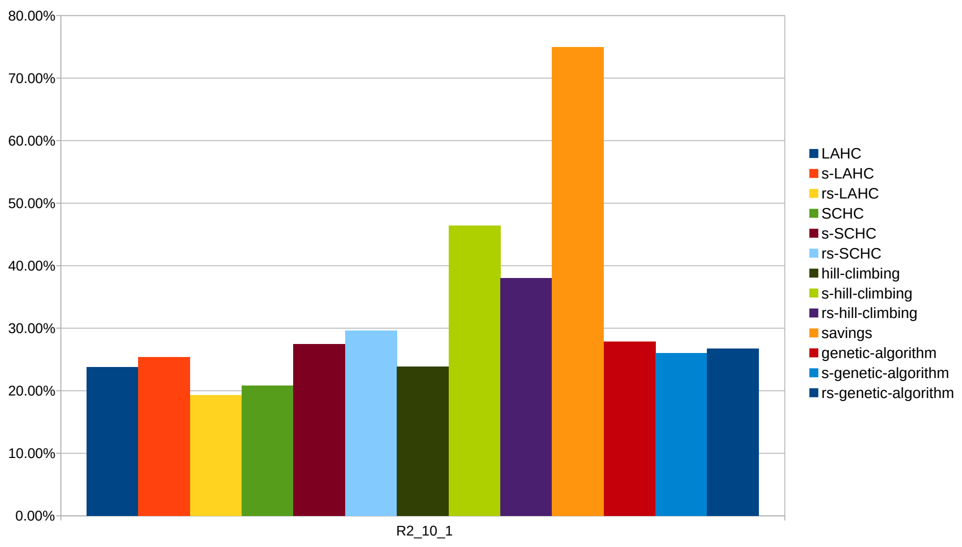

Figure 4.13: Results for instance R2_10_1

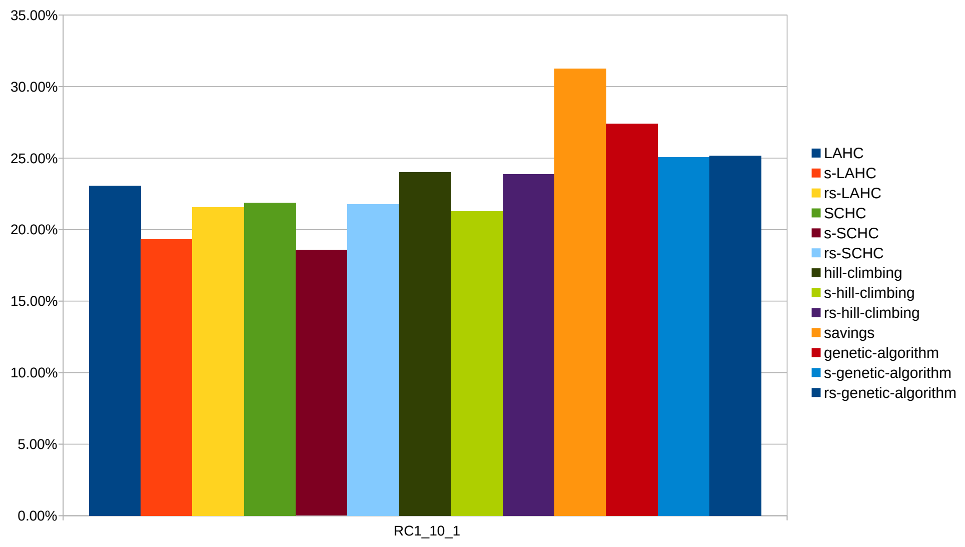

Figure 4.14: Results for instance RC1_10_1

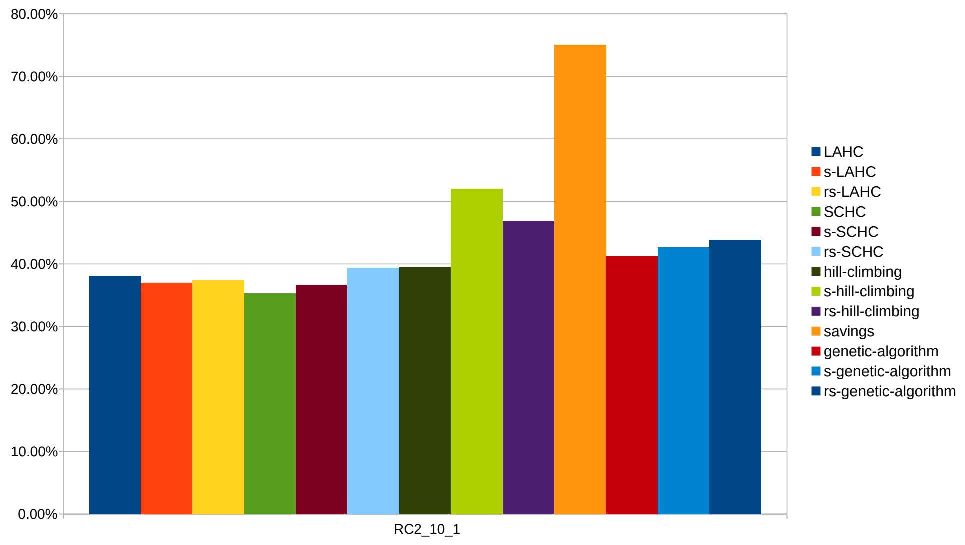

Figure 4.15: Results for instance RC2_10_1

4.3 Conclusion

We have tested SCHC, LAHC, Hill climbing ang Genetic algorithms on two sets of benchmarks. We have shown that SCHC and LAHC (which until now were never tested on vehicle routing problem with time windows) are promising methods for solving VRPTW problems. When combined with more advanced initialization technique (in our case we used Savings and Randomized Savings), we can reach far better results. Results of our modification of the Genetic algorithm are slightly worse than results of SCHC and LAHC. Our variant could be an alternative to already known algorithms, possibly it might serve as a reference for further research of local-search based problems.

Bibliography

[1] Simulated Annealing Cooling Schedules. `http://www.btluke.com/simanf1.html`, accessed 23. 4. 2017.

[2] DANTZIG G. B., R. J. H. The truck dispatching problem. *Management Science*, volume 6, 1959: p. 80–91.

[3] KANG L., X., Yan. *Advances in Computation and Intelligence: Third International Symposium on Intelligence Computation and Applications, ISICA 2008 Wuhan, China, December 19-21, 2008 Proceedings*. Springer, 2008, ISBN 9783540921363.

[4] SOLOMON, M. M. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Institute for Operations Research and the Management Sciences (INFORMS)*, volume 35, 1987: pp. 254 – 265.

[5] Qarke, G.; Wright, J. W. SCHEDULING OF VEHICLES FROM A CENTRAL DEPOT TO A NUMBER OF DELIVERY POINTS. *Operations Research*, volume 12 Issue 4, 1964: pp. 568 – 581.

[6] Burke, E. K.; Bykov, Y. A Late Acceptance Strategy in Hill-Climbing for Exam Timetabling Problems. In *7th International Conference on the Practice and Theory of Automated Timetabling(PATAT2008)*, 2008.

[7] Bykov, Y.; Petrovic, S. A Step Counting Hill Climbing Algorithm. *Nottingham University Business School Research Paper Series*, volume 10, 2013.

[8] Glover, F. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, volume 13, 1986: pp. 533 – 549.

[9] Glover, F. New approaches for heuristic search: A bilateral linkage with artificial intelligence. *European Journal of Operational Research*, volume 39, 1989: pp. 119 – 130.

[10] Kirkpatrick, S.; Gelatt, C. D.; Vecchi, M. P. Optimization by Simulated Annealing. *SCIENCE*, volume 220, 1983: pp. 671 – 680.

[11] H., H. J. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975, ISBN 0262581116.

[12] Solomon, M. M. Solomon's benchmark. `http://w.cba.neu.edu/~msolomon/problems.htm`, accessed 6. 1. 2017.

[13] Gehring and Homberger benchmark. `https://www.sintef.no/projectweb/top/vrptw/homberger-benchmark/`, accessed 6. 1. 2017.

Appendix A

Acronyms

TSP	Traveling salesman problem
VRP	Vehicle routing problem
VRPTW	Vehicle routing problem with time windows
LAHC	Late Acceptance Hill Climbing
SCHC	Step Counting Hill Climbing
GA	Genetic algorithm

Appendix B

Contents of enclosed CD

```
readme.txt ....................... the file with CD contents description
benchmarks ......................... the directory of benchmark results
src .......................................the directory of source codes
  app ........................................implementation sources
  thesis ..............the directory of LaTeX source codes of the thesis
text ...........................................the thesis text directory
  BP_Macho_Martin_2017.pdf ...........the thesis text in PDF format
```