Visualization and Comparison of Randomized Search Heuristics for Dynamic Traveling Salesperson Problems

Daniel-Cristian Loghin

Kongens Lyngby 2013
IMM-M.Sc.-2013-2013-99

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-M.Sc.-2013-2013-99

Summary (English)

The present thesis deals with implementing, analyzing and comparing randomized search heuristics for the traveling salesperson problem. Together with it, a software application was built, that implements five choices of randomized search heuristics (Genetic Algorithm, (1+1) Evolutionary Algorithm, Randomized Local Search, Simulated Annealing and Min-Max Ant System) and the various implementation choices are discussed throughout the report.

The main goal of this Master Project is implementing and analyzing a system that allows the five algorithms to run in a dynamic environment. Four types of dynamic changes are implemented: interchanging the positions of two cities on the input map, having congested roads that get unusable from time to time, deleting and adding of cities. All these changes occur without restarting the algorithms, and the application's results on various tests are discussed in the latter stages of the report.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring an M.Sc. in Informatics.

The thesis deals with building a tool for visualizing and testing of randomized search heurstics on the dynamic traveling salesperson problem. Moreover, it deals with analyzing and comparing these heuristics on different traveling salesperson sample problems.

The thesis consists of a report and a software application.

Lyngby, 26-August-2013

[illegible]

Daniel-Cristian Loghin

Contents

CHAPTER 1

Introduction

A classical problem by now, the “traveling salesperson” (or “traveling salesman”) problem has raised the interest of many mathematicians and computer scientists throughout time. Given a number of cities, and knowing the distances between every two cities, the task is to find the shortest route that visits every city exactly once, and returns to the city of origin. Variations of this problem are known to date as back as the 1800s, but it was first properly formulated in its general form in the 1930s by mathematicians such as Karl Menger [1]. Traveling salesperson has become widely known in the 1960s, when the American company Procter & Gamble[2] launched a contest worth $10.000 for the person who would find the shortest possible route through 33 USA cities, starting and ending in Chicago, Illinois. A few people, applying various methods and algorithms, were tied for the first place, but no efficient algorithm for solving the generalized problem was devised.

Later on, in 1972, the NP-Completeness of the traveling salesperson problem was shown. A brute-force approach, meant to try every possible arrangement of cities in a route, has a complexity of O(n!). Other algorithms were devised, such as dynamic programming, but none yielding a decent result in terms of computation time.

Another way of solving the traveling salesperson problem (TSP) is using randomized search heuristics, designing algorithms that approximate the optimal solution, but offer no guarantee of finding the shortest path. The advantages of this method are speed, having acceptable running times in some implementations, and the high likeliness for reaching an optimal solution for small instances of the TSP. The biggest trade-offs are completeness and optimality, these algorithms offering no certainty that the best solution was found, or that all possible solutions were found along the way. Throughout this thesis a few nature-inspired algorithms from this class have been selected to be implemented and compared, as well as tested to see how well they adapt when dynamic changes occur to the original map: Evolutionary Algorithms (Genetic Algorithm and (1+1)EA), Simulated Annealing and Ant Colony Optimization. Randomized local search was also selected due to its similarities to the other algorithms stated.

1.1 Algorithms studied

Evolutionary algorithms use techniques inspired from biological evolution, such as reproduction, recombination, selection and mutation, techniques that in nature provide an evolution of a species, gradually improving its chances of survival. When similar behavior is applied to a computer problem, a graduate progression towards an optimal solution is shown. Solving TSP with a simple EA, means instantiating a population with random tours around the map, and, at each step, generating a new population by selecting the best fit individuals, reproducing, recombining and mutating them according to the selected heuristics, in order to obtain better tours. Two variations of these are studied in this thesis: (1+1) EA and a standard Genetic Algorithm.

(1+1) EA is perhaps one of the simplest, and most widely used variations of evolutionary algorithms, as it uses a population size of one – one individual that generates one offspring through mutation only. At each step a selection is made between the parent and the offspring, the individual with the best fitness being chosen to propagate the population. Special techniques are deployed either to the selection or to the mutation mechanism, in order to help the system escape from local maximum states.

Genetic algorithm (GA), on the other hand, is more complex, using a large population, usually dependent on the size of the problem, and special selection and reproduction operators. Usually not all the individuals get to generate offspring; the partners that mate are usually selected through specialized mechanisms, such as Tournament Selection or Roulette Wheel Selection. The offspring's solution to the problem is generated from recombining the solutions of its parents, and is also subject to eventual mutations.

Simulated annealing is one of the algorithms that are known to give good results when applied to the traveling salesperson problem. It was inspired from annealing metal – heating a material above its critical temperature, and then letting it cool, in order to alter and improve its physical properties. The algorithm replicates this method by keeping a temperature variable. Initially the temperature is set as high, and it is gradually decreased according to a selected cooling schedule. At higher temperatures we allow the program to accept solutions that are worse than the previous one, with a higher frequency, thus offering a good likelihood that the algorithm escapes from local optimums. Once the temperature drops, the frequency of accepting bad solutions also decreases, forcing the solution to be searched around the current optimum and changes to occur at a lower frequency and scale. During the first stages of the algorithm the general area around the best solution is searched for, and afterwards, once one such area is selected, the best solution is approximated with small shifts inside the selected area.

Ant Colony Optimization is another probabilistic algorithm, searching for the optimal solution of a problem by simulating the way ants find their path from their colony and the source of food. In nature ants rely on pheromones to find their way around obstacles, and to remember the location of their colony. Each ant moves at random, and while walking it releases an amount of pheromone to remember its path, and also help other ants follow on the same path. More pheromones on a trail, the higher the probability that trail will be followed by any specific ant. The algorithm uses a similar process: a number of ants are “let free” too choose their path at random, releasing pheromones on their way. At each node reached the ant decides on the next segment of the path based on the amount of pheromone on each segment. After each run of the algorithm, pheromones dissipate in a percentage, and the process is repeated until the method converges to one solution, as some fragments of the path get chosen more often. Selection mechanism can be added, updating the ant’s solution with the new path, only if this path offers better fitness than the old one. For the purpose of this thesis a variation of ACO was chosen, namely Min-Max Ant System (MMAS), its major difference being having maximum and minimum values for the pheromones around the map, enabling better results by giving other paths more chances to get chosen by the ant. This can help avoiding states where the system is stuck in a local optimum.

A final algorithm that is studied in this thesis is Randomized Local Search, with many similarities to the classical hill-climbing approach. It takes a random solution, and tries to improve it by applying small changes to it, and testing to see if it has reached a better fitness. If that is the case, the best-so-far solution is updated, and the process continues with it. This algorithm has a high affinity for local optimums, but has the advantages of simplicity and speed, giving acceptable results in most of the cases.

1.2 Motivation

The classical traveling salesperson problem has been intensively studied through time, and various solutions and optimizations have been designed. The algorithms discussed above have been proven to give good results when applied to this problem, for static instances of it, but the aspect that hasn't been researched very thoroughly is the way they adapt to a dynamic environment, where changes of the problem occur whilst they process the data. Various commercial and non-commercial applications can benefit from a study regarding the way these algorithms adapt to changes of the environment, and which adapt better and faster. For example, a routing mechanism would need to adapt its behavior if a server disappears from the network, or it would need a way to detect and compensate a high traffic on a line that could lead to congestion.

Four dynamic alterations have been chosen to be studies throughout this thesis: interchanging of the location of two cities, intermittent congestion of a road between two cities, and deleting, respectively inserting a city whilst the algorithm is still running. The selection criteria were based on the interestingness of the problem, as well as the level of study that was previously done and its applicability in real-life situations.

Another interesting aspect of this would be studying the way the moment in the algorithm's lifetime when the changes occur, influences the final outcome of the algorithm. Some algorithms might benefit from a random congested road in the beginning of their run, on some input instances, forcing them to find detours around that road, which can prove to be a better solution than the one previously found. Deleting or adding a city to the problem might be advisable to occur after a very close to optimum solution was found, requiring only small changes to return to a high fitness state.

1.3 Structure

The following chapter deals with theory notions providing deeper understanding of some of the main concepts used throughout this thesis, such as the 2-OPT operator used as a mutation operator in the implementation of the algorithms, or the Poisson Distribution, used to determine the scale of mutations that occur at one stage of the EA.

The theory chapter will provide the background information necessary for the following chapter, where each algorithm used is described and analyzed. Throughout its five sections, the general form of the algorithms is displayed, alongside with various theoretical notions that are used in their implementation.

The next chapter deals with the implementation of the project's application, the libraries and data structures used the implementation of the algorithms and the choices made to increase their efficiency to a dynamic environment, as well as the motivation behind these choices. This chapter also explains the implementation of the dynamic changes on the original input map.

The penultimate chapter portrays and analyzes the results of testing the algorithms on several types of input, both with and without dynamic changes of the environment. Their adaptability to problem changes is compared, thus providing a ranking of how fit these algorithms are to every specific alteration of the environment.

The final chapter will provide conclusions of the study and suggestions for further research.

Chapter 2

Theory notions

2.1 TSP and NP-Completeness

The classical description of the traveling salesperson problem refers to a salesman who has a set of cities and is required to visit every one of these cities at least once, starting from one set city, and returning to it. The requirement is for the salesman to cover the shortest distance possible.

Mathematically, this can be described as having a complete weighted graph G(V,E), and finding the Hamilton cycle with the minimum weight. A Hamilton cycle is a graph cycle passing through all the vertices once.

As previously said, TSP is a NP-Complete problem, meaning that solving it is done in a nondeterministic polynomial time. In other words, no algorithm is known that provides a solution in a fast, constant time, and that the computation time increases exponentially with small increases of the input size. This high complexity is the main factor why approximation algorithms are devised for these kind of problems, such as those described in this thesis. Proving that TSP is NP-Complete is a two-step phase: proving that the problem belongs to the NP class, and then proving that it is NP-hard.

Since NP refers to decision problems, we can reduce the classical TSP to a

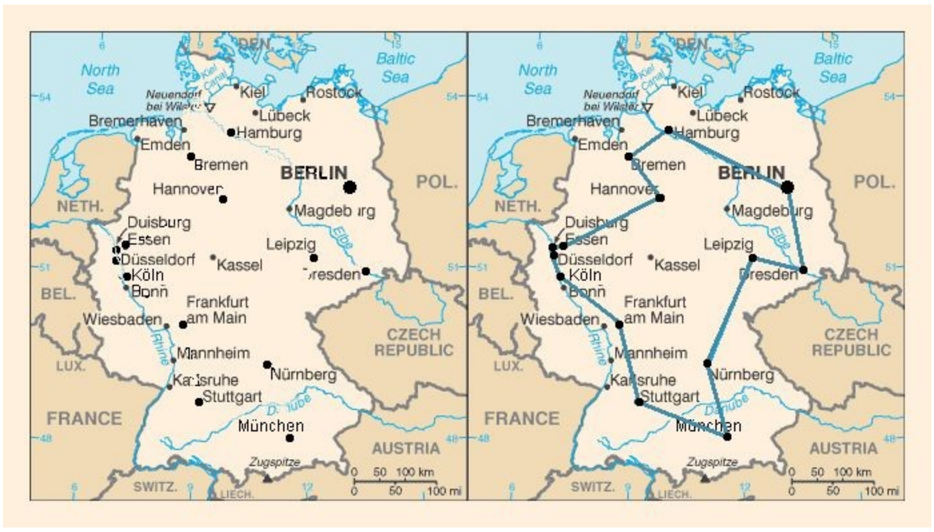

Figure 2.1: Better image to follow

decision problem: given a graph G, a maximum weight μ, and a Hamilton path through this graph, decide if the path is a Hamilton cycle with a total weight lower than μ. To prove that the TSP belongs to NP, it is sufficient to prove that this decision version can be solved in a polynomial time. This can easily be done by done by summing the weights of all the edges along the path, and then test if the total weight is lower than μ, and that the path goes through all the vertices, returning to the first.

In order to prove NP-hardness, we need to reduce another problem, known to be NP-hard, to our traveling salesperson problem. A suitable problem for this is the Directed Hamilton Cycle Problem. Given a directed finite graph G, this problem asks whether G has a directed cycle going through all vertices exactly once. Reducing TSP to the Directed Hamilton Problem has a straight-forward process: given a graph G(V,E), with a total number of vertices ν, for each pair of vertices x and y, if there exists an edge between x and y, then the distance between the two is 1, else the distance is 2. Then we need to prove that G contains a Hamilton Cycle if and only if there is a route of distance at most ν:

"$\Rightarrow$" Suppose the graph has a Hamilton Cycle (this means that there is a cycle going through each vertex exactly once), since there are ν vertices, the total weight of this cycle is exactly ν.

"$\Leftarrow$" Suppose there is a route going through all the cities, and its total weight is ν, we need to prove that the graph contains a Hamilton Cycle. If the length of the route is ν, and it goes through ν cities, it means that every edge has a weight of 1. By definition this concludes that these edges belong to the edge set of the graph E. If all edges of the route belong to G, and all vertices are visited, this means that the graph contains at least one Hamilton Cycle, in the form of the input route.

2.2 2-OPT move

First suggested by Flood in 1956, the 2-OPT move works by removing two edges from a path previously constructed, and then reconnecting the two halves of a path obtained. When disconnecting the tour by removing two edges, there is only one option of reconnecting the nodes that creates a valid path.

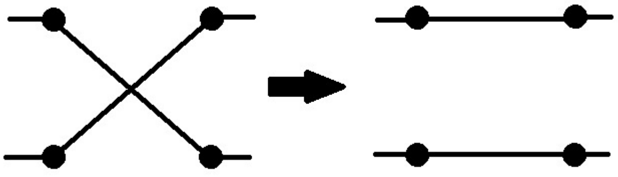

Figure 2.2: Fig 1. 2-OPT move example

This kind of move inspired Croes in 1958 [3] to devise an algorithm called 2-OPT (2-Optimization), algorithm that relies on repeatedly applying the 2-OPT move, if by doing so the total length of the path is decreased. When no more 2-OPT moves are left, that can improve the solution, the algorithm stops, and the tour obtained is called 2-optimal.

Similarly, a 3-OPT move exists, disconnecting three edges, and reconnecting the path, and a 3-OPT algorithm based on it. Recently there have been many studies focusing on k-OPT moves and algorithms, in order to decide whether increasing the size of the operator (the number of edges disconnected) will increase the success rate of the algorithms; combining multiple types of OPT moves in the same algorithm is another focal point for researchers studying the traveling salesperson problem.

For the purpose of this thesis, only the 2-OPT move is take into consideration. It is used as a mutation operator for the (1+1) EA and GA algorithms, as well as the path alteration method for the Randomized Local Search and Simulated Annealing.

No. of events	μ=1
0	0.368
1	0.368
2	0.184
3	0.061
4	0.015
5	0.003
6 or more	0.001

Table 2.1: Table 1. Poisson Table

2.3 Poisson Distribution

The Poisson Distribution, named after the French mathematician Simeon Denis Poisson, is a probabilistic distribution describing the number of events that could occur within a time interval [4]. Having an environment where a number of identical events happen within a time frame, the events being independent from each other, and the average frequency of the event is known, the Poisson distribution models the probability that N events will happen within the next time frame. The variable N is a natural number [0 : ∞].

$$P[N;\mu] = \frac{e^{-\mu}\mu^{N}}{N!}$$

Figure 2.3: Fig 2. Poisson formula

Computing the Poisson Distribution is done using the Poisson Formula shown above. Knowing the average number of occurrences, we can interrogate the likelihood that N events will happen in the next time frame. The symbol N represents the number of occurrences we would like to determine the probability and μ is the average number of event occurrences. Repeating the process for multiple values of N, we construct the Poisson Distribution for the selected values. This can be displayed in a Poisson Table:

Such a table is constructed within this project's implementation of the (1+1) EA, for determining how many 2-OPT mutations will occur to the individual at each step of the algorithm. The average number of mutations at every step is considered to be 1, and using this value for μ, a table of values similar to the one above is constructed (stored in an array). A random number between 0 and 1 is generated at every step, and the number of mutations applied corresponds to the first probability higher than the generated number. For example, if the random number is 0.16, the first probability higher is 0.184, corresponding to two occurrences of the event.

In order to ensure that at least one change is applied in every step (since there is a high probability that the random number will be higher than the Poisson Probability that one event occurs – 0.368), one extra mutation is added to the previously computer number.

Chapter 3

Algorithms

3.1 Randomized Local Search

One of the simpler randomized search heuristic, randomized local search (RLS), and based on the classical local search algorithms, works by trying to improve the current solution during each iteration, and hoping that going through better and better solutions, it will reach the best possible solution. Starting from an initial random solution, it generates new solutions at each step, one per step, by applying a small change to the previous one. If the newly found solution provides a better fitness, being closer to the global optimum, it is kept as the "best-so-far" solution to the problem, replacing the old. If the fitness of the new result is equal or worse than the current one, it is ignored. Being mostly used to generate bit-string solutions, RLS mutation for each step means flipping a randomly chosen bit; a more complex mutating operator is used in this thesis, the 2-OPT operator discussed above.

Using a random initial solution and having a random mutation operator to improve the solutions found at each step, is what differentiates RLS from the classical local search algorithm. Another aspect that could differentiate it is the stopping criterion. Based on the problem itself, if the fitness of the best solution is known from the start, and only the path to obtaining it is required, a stopping point could be reaching the solution with this fitness. When looking for the optimum solution only, this algorithm provides no guarantee for ever stopping. If an approximation of the optimum is required, other stopping criteria could be allowing the algorithm to only run for a set number of steps, in cases where speed is a lot more important than accuracy, or stopping the algorithm when no new improvements to the solution are found after a set time limit.

Algorithm 1 Randomized Local Search

```
choose solution $\in \{0,1\}^n$ randomly
while (stopping point not reached) do
    newSolution = solution
    choose K $\in$ [1,n] randomly
    flip bit K in newSolution
    if (fitness(newSolution) > fitness(solution)) then
        solution = newSolution
    end if
end while
return solution
```

The main advantage of RLS is speed and the relative small use of computer resources, at each step only holding two data structures (usually arrays) in memory, to store the current and the new solution. The main disadvantage of the algorithm is its high affinity for remaining stuck in states of local maximum, or a plateau, due to its search for new solutions in a very small neighborhood of the current best. In some instances of the input problem, with particular state space landscapes, one mutation per turn might not be sufficient to escape these states. Since reaching a local optimum state is mostly dependent of the initial random solution, some implementations of the algorithms use the technique of restarting the RLS with a new initial solution, for a number of times, and then taking the maximum over all the runs. Other algorithms (such as (1+1) EA and Simulated Annealing, that will be discussed in later chapters) extend the RLS functionality by allowing some worse fit solutions to be accepted, or doing more than one mutation at each step.

3.2 (1+1) EA

Being one of the simplest and more robust versions of evolutionary algorithms, (1+1) EA can also be seen as an expansion of RLS, due to its similar hill-climbing behavior. Evolutionary Algorithms, in general, represent the solutions to the given problem in the form of chromosomes of specific individuals of a population. Each such individual holds a possible solution to the input problem in its “genetic material”, and new individuals with new solutions are generated at each step of the algorithm, as children of the old population. The population evolves gradually, mimicking natural evolution and providing improved solutions as the population is regenerated and enriched using genetic-inspired operators, such as cross-over, mutations and selection.

(1+1) EA implements the same functionality, but for a population size of one: at each step only one individual with one solution exists, in the form of the “best-so-far” answer to the problem. A new individual is generated at every step, as an offspring of the current optimal individual, through mutation of its chromosomes. The fitness of the offspring is then compared to the parent’s, and, similarly to RLS, a selection is made between the two, the individual with the better result being selected for further improvements.

Stopping criteria of (1+1) EA are similar to those in RLS: either when a solution is found satisfying a fitness criteria, or the actual fitness of the optimum is reached, if it is known from beginning, or a set maximum number of generations have been generated. Another stopping criterion, which is also investigated in this thesis, is stopping the algorithm when no improvements to the current best solutions are found in a set number of tries.

Algorithm 2 (1+1) EA – General Scheme

```
choose solution $\in \{0,1\}^n$ randomly
set p(n) $\in (0, \frac{1}{n}]$.
while (stopping point not reached) do
    newSolution = solution
    Flip each bit in newSolution independently with probability p(n)
    if (fitness(newSolution) > fitness(solution)) then
        solution = newSolution
    end if
end while
return solution
```

The main difference between (1+1) Evolutionary Algorithm and Randomized Local Search, besides the obvious population-based approach, is the way it tries to deal with local optimums and plateaus in the state space landscape. While RLS does nothing to prevent getting stuck in these states, (1+1) EA allows a bigger scale of their mutations, or even multiple mutations during the same algorithm phase, with a given probability, thus giving the application a chance to escape such states. For example, in the classical bit-string representation of the solution, a classical mutation for the (1+1) EA is flipping each bit independently, with a probability of P(n) = 1/n as suggested by [5]. Having this probability, the algorithm provides an average of 1 bit flip per mutation, similarly to RLS, but allows the opportunity for multiple bit flips to occur, which can draw the result closer to the global optimum. Because of this, 1/n is the most recommended mutation probability for (1+1) EA, throughout many studies, such as [5].

The advantage of using this type of mutation behavior is, as stated above, the ability of the algorithm to escape some local optimal states, but the trade-off is an overhead to the expected running time: Garnier, Kallel and Schoenauer [7] have shown that the expected running time of (1+1) EA with the probability P(n)=$\frac{1}{n} is \frac{2^n}{1-e^{-1}} \approx 1.582 \bullet 2^n$. This is larger than the expected running time of random search, by a constant factor.

The current project uses a different type of data structure for storing a solution to the Traveling Salesperson Problem (a list integer values, representing the indexes of cities, in the order they are visited), and a different mutation operator, the 2-OPT move. Due to these differences, a different mutation mechanism and probability: the Poisson Distribution is used for determining the amplitude of the mutations – a random number in the interval [0,1] is generated, and it is used in conjunction with the Poisson Distribution to determine how many 2-OPT mutations are applied to the offspring. Having an expected number of event occurrences, μ, in the Poisson Formula, an average of one 2-OPT mutations is insured for each step, similar to the $\frac{1}{n}$ probability in the original (1+1) EA implementation.

3.3 Genetic Algorithm

Genetic Algorithms are devised to mimic the natural evolution process first described by Charles Darwin in his "Theory of evolution". In nature species evolve to better fitness through natural selection, sexual reproduction and mutation. At each point of natural evolution the individuals from a species that are better fit to survive the environment, get to live and produce offspring carrying their genetic material, while the ones with less adaptation skills perish. When two highly fit individuals mate, their genetic material get carried on to their offspring, usually leading to children with a high fitness, sometimes even higher than their parents', through combining the good genes from both. From time to time an individual suffers a random mutation that boosts the survival skills of that individual. These mutations also get to be passed to the individual's children, becoming gradually blended in the species' overall genes.

In Computer Science, Genetic Algorithms, first devised in the 1970s by John Holland, replicate this natural evolution process: they model the possible solutions to the problem as genetic material (chromosomes), and store them in the attributes of a virtual species, a population of possible solutions. Based on the fitness of the solutions, some individuals of this population get selected to mate, and generate new possible solutions, in the form of offspring. When two individuals mate, their chromosomes get combined, in the attempt to gather the better genes from both individuals' chromosome, and to provide better solution; the techniques of doing this recombination vary from algorithm to algorithm, but carry the generic name of "cross-over".

When new offspring are generated, they might suffer mutations, with a small probability, mutations that allow the algorithm to escape local optimum states, and improve its solutions. Repeating these operations, and regenerating the population at each step, usually leads to a convergence towards one solution over the entire population, the solution that approximates the general optimum best.

3.3.1 Chromosomes

In biology, a chromosome is an organized structure of DNA, protein and RNA, uniquely describing every individual. These chromosomes hold the genes, every gene corresponding to a various biological peculiarity of the individual: from eye color to number of limbs, blood type, etc.

Chromosome A	0111001011
Chromosome B	1100000101

Table 3.1: Bit-string chromosome samples

In Computer Science chromosomes (sometimes called genomes) is modeled as a set of genes, every such gene representing an independent variable of a legal solution to the proposed problem. A chromosome as a whole represents a possible solution to the input problem, and also an individual of a specific generation of the genetic algorithm. The encoding of a chromosome depends on the problem itself. Binary encoding is most frequent, mainly because the first studies of GA were using it, and most GA literature refers to it, but other encodings can be used: permutation encoding (as strings of numbers), value encoding (as a list of numbers, strings or complex objects, answering the given question together), or even tree encoding. [http://www.obitko.com/tutorials/genetic-algorithms/encoding.php]

3.3.2 Selection

The main purpose of selection is giving higher changes to better fit individuals to mate and to pass their genes to the next generations, but also preventing non-adapted individuals to propagate their genes in the detriment of the entire population. In nature, individuals with good genes are found more attractive by partners, and get higher chances to select the mates they find better fit, and multiple chances to reproduce. The bad genes naturally perish, first because the individuals carrying them lack the necessary survival skills, but also because they get limited or null chances to mate.

When devising a genetic algorithm, there are many options of selection operators to choose from [I-JEST11-03-05-190.pdf]. These usually operate at the level of chromosomes, evaluating the chances of each individual to reproduce based on the fitness of their genes. The selection method controls the diversity of the population, and thus the convergence speed of the algorithm towards the global optimum, as well as having a big influence on the ability of the algorithm to escape local maximum states. The two best known selection operators are "Roulette Wheel Selection" and "Tournament Selection".

Algorithm 3 Roulette Wheel Selection

```
Set FS = $\Sigma$ fitness(i) for all i $\in$ population
R = random number $\in$ [0,FS]
sum = 0
for all (individual i $\in$ population) do
    sum = sum + fitness(i)
    if (sum $\geq$ R) then
        return i
    end if
end for
```

In the Roulette Wheel Selection, each individual gets a chance at becoming a parent, proportional to its fitness. The way it works can be seen as having a roulette wheel with all the individuals in the population, each slot representing one of them, and each slot having the size proportional to the fitness of the individual it represents; the higher this fitness, the bigger the slot. This method allows for seemingly bad solutions to be selected, with a low probability, but also allows individuals with bigger slots to get chosen to mate repeatedly.

Algorithm 4 Tournament selection

```
Tournament[] = select K individuals from the population, at random
maxFit = -$\infty$
parent = null
for all (individual i in Tournament[]) do
    if (fitness(i) > maxFit) then
        maxFit = fitness(i)
        parent = i
    end if
end for
return parent
```

The Tournament Selection method offers a higher weight to the fitness of individuals than Roulette Wheel. It works by randomly selecting a K number of individuals (usually two), and comparing their fitness. The individual with the highest fitness is then selected to reproduce. Compared to the Roulette Selection, this method has lower probabilities that a seemingly bad solution gets selected (for example the individual with the worst fitness in the set could never get selected). Based on the size of the tournament, and the way the solutions get selected to be compared, some individuals can get chosen repeatedly, but the method usually ensures a high diversity of mating options.

3.3.3 Crossover

Crossover is a genetic operator that allows generating new chromosomes, and new individuals (children), by combining the chromosomes of multiple parents. In nature, two parents generate one or more children that share a set of characteristics (genes) from both parents. In Computer Science the operator is used to create new chromosomes from the parents', and new individuals; the main advantage of this method is the fact that the crossover may select the best genes from each parent, generating a better solution in the offspring.

Usually, for the genetic algorithm, only two parents are mating at a time, the crossover generating two new children. Some of the more frequent operators are the "one point crossover", "two point crossover" and uniform crossover.

Parent 1	0111‖001011
Parent 2	1100‖000101
Offsrping 1	0111‖000101
Offspring 2	1100‖001011

Table 3.2: One point crossover

k The one point crossover generates a random split point and splits the two parent's chromosome into two sections. The first offspring shares the first section of the first parent, and the second chromosome section of the second parent, while child number two has the first section of the second parent, and the second section of the first parent.

Parent 1	011‖1001‖011
Parent 2	110‖0000‖101
Offsrping 1	011‖0000‖011
Offspring 2	110‖1001‖101

Table 3.3: Two point crossover

The two point crossover randomly selects two points within the chromosome, and interchanges the section between the two points in the parents' chromosomes, to generate two new children.

The uniform crossover decides at the level of each gene (bit) of the child's chromosome, whether it will come from the first or the second parent. So this method uses a for-loop to go through all the genes from left to right, and decides with a probability of 50% if the gene of the first offspring comes from the first or the second parent; the remaining bit, from the unselected gene is added to the second child.

Selecting the right crossover operator depends on the problem itself, and on the chosen encoding for the chromosome. The operators presented above work well for bit-string representation of the solution, but many more options are available. Crossover is used at every step of the GA - the process of selecting parents, and applying crossover to generate offspring is repeated until a new generation of the same population size is obtained.

3.3.4 General Form

Algorithm 5 Genetic Algorithm

```
population[] = N randomly generated individuals, with independent solutions
while (stopping point not reached) do
    offspring[] = empty population set
    for (i = 1:n 2) do
        parent1 = individual ∈ population[] according to selection method
        parent2 = individual ∈ population[] according to selection method
        child1, child2 = Crossover(parent1,parent2)
        Mutate(child1) with a set probability
        Mutate(child2) with a set probability
        Add(child1, offspring[])
        Add(child2, offspring[])
    end for
    population[]=offspring[]
end while
return best fit solution from population[]
```

The genetic algorithm starts off with a randomly generated set of solutions, in the form of a population of individuals. At each step of the algorithm a new generation of the population is created, by repeatedly selecting two parents, according to the selection method, and applying the crossover operator to them, in order to obtain two new individuals; these children can then suffer mutations, according to a set probability (usually 1/N probability, so that only one mutation appears per generation). After this, the two new individuals are added to the new population, and the process is repeated until the new generation has the same size as the initial population.

The main advantage of GA is its ability to search different neighborhoods of the solutions space, at the same time, and to merge multiple such solutions, from different areas, into new solutions, closer to the optimum. This can dramatically decrease the chance of the algorithm to get stuck in local maximum states, and also increase the probability of reaching the area of the global optimum. From this point of view, crossovers can be seen as informed moves in the state space, while mutations can be seen as explorations of unknown regions. Another advantage of GA is the relative easiness to be converted into a parallel algorithm, where operations such as selection, crossover and mutation can be done simultaneously, in order to improve its running time.

The main disadvantage of GA is its high complexity, both temporal and spatial, using many computer resources, and using a large amount of time to apply the various operations, and to deal with the large number of individuals.

3.4 Simulated Annealing

As previously discussed, the main problem of RLS is getting stuck in local optimal states, sometimes never reaching the maximal solution due to the small size of its mutation. (1+1) EA is one of the algorithms devised to prevent, when possible, this problem, by implementing a greater scale of its mutations. Another algorithm extends randomized local search, by providing a nature-inspired technique for escaping local maximums – Simulated Annealing.

In metal works annealing is the process of improving a metal's structural properties, by repeatedly heating and cooling it. When metal reaches temperature higher than its critical point, its internal structure get altered, and when let to cool, these alterations get fixed as permanent characteristics. A similar heating-cooling off mechanism is used by the algorithm, by instantiating a variable with a high temperature, and then gradually decreasing it at each run. When the temperature is high, greater variations of the solution are allowed, by allowing seemingly worse solutions to be accepted; as temperature decreases this behavior is hindered, only better fitness being chosen. In the solutions space this technique can be seen as exploring the space for the general region where the optimum is located, by allowing bad solutions to get selected, at greater temperatures. Once the system cools off, as we have found a proper region, the variation rate decreases, smaller steps are taken in order to look for the maximum in that specific area. The algorithm start with an initial temperature,

Algorithm 6 SA – General Scheme

```
solution = initial solution, randomly chosen
t0 = initial temperature as a high number
alpha(t) = temperature cooling function
while (stopping point not reached) do
    newSolution = mutate(solution)
    diff = fitness(newSolution) – fitness(solution)
    if (diff > 0)) then
        solution = newSolution
    else
        choose random R in (0,1)
        if R then < $exp(\frac{-diff}{t0})$
            solution = newSolution
        end if
    end ift0 = alpha(t0)
end while
return solution
```

t0, that is gradually decreased according to a function alpha, called "cooling schedule"; multiple forms of cooling schedules have been proposed, from constant cooling factors, to methods that take into consideration the size of the problem or the number of steps computed so far.

$$alpha(t) = \begin{cases} a * t, \; with \; a \; = \; (0.8, 0.99) \\ \frac{t}{(1+b*t)}, \; with \; b \; small \; \approx \; 0 \\ \frac{c}{log(1+k)}, \; where \; c \; constant \; and \; k \; the \; number \; of \; steps \end{cases}$$

A new solution is generated at each step, according to the selected mutation operator. On bit string encoding, this operator could be the classical RLS flipping of one random bit. The fitness of the new solution is compared to the previous, and if it shows an improvement, the new solution is kept. If that is not the case, a random number R is generated between 0 and 1, and is compared to the exponential of the fitness difference over the current temperature ($R < exp(\frac{-diff}{t0})$). This formula was suggested by one of the inventors of simulated annealing, Scott Kirkpatrick, corresponding to the Metropolis-Hastings algorithm, and it is used to determine whether the current temperature allows a bad move.

This process is repeated, and as the temperature slowly drops, Simulated Annealing transforms into a classical randomized local search algorithm. Stopping criteria are similar to those in RLS or (1+1) EA.

3.5 Ant Colony Optimization

Invented in 1992 by Marco Dorigo et al, Ant Colony Optimization (ACO), is another nature inspired randomized search heuristic performing well for path finding problems, such as TSP; in fact, the authors first implemented it for the traveling salesperson problem, due to the natural similarities to the way ants find sources of food. In nature ants, when leaving their colony to find food, first walk randomly in search for food. Once the goal was reached, they return home, but leave a pheromone trace on their way back. The ants that follow will then sense this pheromone trace, and knowing it leads to food, are more prone to follow it.

The process repeats, and the more ants that use the same road, the more intense the level of pheromones gets, on that specific road. Another important aspect of this behavior is evaporation: in time the pheromones evaporate, and the tendency of ants following that trail drops. This is very important for escaping local optimum roads, as the length of the road influences directly the pheromone intensity on it – the longer it takes an ant to complete the road, the more evaporation occurs on it. Experiments have shown that given the same source of food, and multiple paths to it, the ants' behaviors converge towards the shortest road. In one particular experiment, two roads were built between the colony and the food source, one shorter than the other; the ants going towards the goal on each road were counted, and the results showed an astounding number of 80

The ACO algorithm implemented multiple techniques similar to the ants' behavior. It is designed to work well on problems where the solutions can be divided into multiple sub-goals that the ants can discover and solve incrementally, such as path and route finding problems. A pheromone map is generated for all the possible segments of solution, initially having equal values. Then, a population of ants is generated, usually having different starting points, and then let to "roam free" in order to discover the solution asked for. At every intersection reached (every sub-goal met), the ant randomly decides the next segment it will take towards reaching the solution. When an algorithm step was finalized, and all ants have found their solution, the pheromone map is updated, increasing the pheromone levels on the segments chosen by the ants, and evaporating it on the rest. The ant population is then reset, and the process repeats itself, with the addition that when an ant reaches an intersection, it will have a higher probability to select the segments with the higher values of pheromones.

Algorithm 7 Ant Colony Optimization

```
ant[] = ant population
$\tau_{i,j}$ = initialize pheromone map
**while** (stopping point not reached) **do**
    **for all** (ant A in ant[]) **do**
        set k current location of the ant
        set u current walk of the ant – empty list
        **while** (solution u is not complete AND there are still feasible continuations of u) **do**
            select next node l with probability $p_k l(n)$
            $p_k l(n) = \begin{cases} 0, if\ (k,l)\ is\ not\ feasible \\ \frac{g(\tau_k l(n), \eta_k l(u))}{\sum_{(k,r)} g(\tau_k r(n), \eta_k r(u))},\ over\ all\ possible\ continuations\ r \end{cases}$
            k = l
            append l to u
        **end while**
    **end for**
    Update Pheromones
**end while**
**return** solution
```

In the algorithm above, $n_i j(u)$ is called heuristic information value, and is a problem specific heuristic, dependent on the partial road u selected (as discussed in [6]) . It is also possible to construct an algorithm that doesn't use this kind of heuristic information values. The function g combines the pheromone trail and the heuristic information values, and usually represents a multiplication between the pheromone trail and the heuristic information, each with different exponents.

As the algorithm shows, at each step, it loops through each ant, every ant constructing its own possible solution. After all solutions for a specific step were found, the pheromone map is updated as described above, based on the generated roads. In time the algorithm converges towards on single solution that, based on the heuristics and pheromone scheme used, can be a good approximation of the global optimum.

Generally the mechanisms of selecting the next solution segment in ACO allow a degree of randomness. This, together with the pheromone evaporation rate, is a very important means of avoiding reinforcements of local optimums, allowing the algorithm to explore different regions of the solutions landscape, sometimes avoiding getting frozen plateaus and local maximum states. Termination of ACO is similar to the algorithms described above, either looking for a set fitness of the solution, or a set number of steps, or stopping it when no improvements get generated.

3.5.1 MMAS

Different variations of ACO were proposed throughout time, and this thesis implements one of these variations, named Min-Max Ant System (MMAS). This extension was proposed by Stytzle and Hoos, and it brings two main improvements to ACO.

While ACO allows all ants to deposit pheromones on their roads, MMAS allows only particularly good solutions to be reinforced. Usually this is implemented by only reinforcing the best solution/solutions found during one step, improving the convergence rate of the algorithm. Other implementations only reinforce the best found solution, throughout all the steps taken.

Another change from the classical algorithm is forcing a more balanced pheromone trail, between good and bad solutions; for this, MMAS proposes a minimum and maximum value that the pheromones can take.

This thesis uses $\frac{1}{n^2}$ as the lower bound, and $1 - \frac{1}{1-n^2}$ as the upper bound for pheromones. And, for simplicity, as the requirements of TSP allow, it implements a population of one ant starting from the same city at every step. Having one ant, this will be the ant providing the best solution of every step.

Chapter 4

Implementation

The following chapter will provide details about the choices made throughout the implementation of the projects' software application. The purpose of this application is providing a graphical interface to visualize the way the five algorithms selected (randomized local search, (1+1) EA, genetic algorithm, simulated annealing and min-max ant system) find their solutions to the traveling salesperson problem. The interface allows the user to choose the input problem and one of the five algorithms to solve it, and to visualize, step by step, the construction of the solution. The user is also given the option to choose one of the four implemented dynamic changes to be applied to the input TSP map.

The application was developed under Windows 7 Ultimate x86, using Microsoft Visual Studio 2010. The selected programming language is C#, using a .NET framework to construct a Windows Forms Application. The main motivation behind this choice is the graphics libraries built for this programming language that ease the process of building user interfaces and two-dimensional drawings.

4.1 User Interface

When opening the application executable, the user is shown the main (and only) window of the visualization of randomized search heuristics for dynamic TSP. This window has two points of focus: a vertical strip on the left side, allowing the user to select and input the running parameters of the application, and starting the run of the chosen algorithm; the other point of focus is the right side, where a two-dimensional graphic display of the solutions found by the algorithm is shown, in real time.

The first input that the user needs to provide is the traveling salesperson map instance that the algorithms need to study. The application has an in-built input map generator, called circle generator, which creates a new map by setting a given number of cities on a circular map. Another option provided to the user is the input of a file containing the map to be studied, with the number of cities and their coordinates.

Secondly, the user needs to select the algorithm to be run and whose process of solutions discovery will be displayed. For this, the application provides five radio buttons, marked with the names of the algorithms (GA, RLS, SA, (1+1) EA and MMAS), only one of them being selectable at each time.

Thirdly, the user can select one of four dynamic changes to be applied to the initial input problem: interchanging of two cities, forcing a road between two cities to suffer from congestion, deleting a city, or adding a new city. Only one of the four options is selectable at each time, but choosing a dynamic change is optional – if no option is selected, the algorithm will run on the static version of the input TSP map.

Finally, the user can press the “Start” button to initiate the selected algorithms, and to view the algorithms at work. Every accepted solution for every step of the algorithm is displayed, allowing the user to see the gradual improvements and the convergence towards the global optimum solution. Underneath the start button the user is shown the fitness of the solutions found throughout the exploration of the state space landscape.

When the run of an algorithm is started, the frame of the right side of the application displays a two dimensional representation of the inputted map, scaling the coordinates of the entire map to the size of the frame, and showing the cities as red squares on it. Every tour through the cities found and accepted as possible solution to the input problem is displayed in the same frame, every segment of the road being shown as a blue line between two cities. Additional visual cues are displayed when dynamic changes occur, but this will be discussed at a later stage.

4.2 Classes and methods

The application developed for visualizing and testing randomized search heuristics applied to the traveling salesperson problem uses several classes and methods that will be briefly discussed in the current section.

4.2.1 Form

This class deals with the user interface of the project, controlling the graphical layout, setting up the main window of the application, and ensuring a bidirectional communication with the users. It is also in charge with instantiating other classes, and starting the user selected algorithm as well as launching the dynamic changes and displaying the solutions found on the two dimensional map.

Methods:

- *pictureBox_Paint*– this function deals with updating the two-dimensional representation of the TSP map according to the new solutions found
- *button1_Click*- launched when the user clicks the "Start" button, this function initializes the variables necessary and parses the user input, in order to launch one of the five algorithms implemented. It holds the five algorithms themselves, using the help of other classes to construct the solutions. Also launches the dynamic changes.
- *resetAll* – resets the all the objects and variables that were previously used by one algorithm, after it has finished its run.
- *applyDynamicChange* – deals with applying a dynamic changes; has three inputs that help control the type of change to be applied and the moment of its initialization: a random number generator, the step count of the algorithm, and the step at which the dynamic change is applied. This function reads the selection of the radio buttons, to decide which type of change to enforce.
- *getpoisson* – generates the Poisson Distribution for 1 to 10 events, returning it as an array of doubles.

- *applyInterchange* – randomly interchanges the location of two cities.
- *copyCity*– creates a copy of the given City object.
- *applyCongestedRoad* – applies the congested road dynamic change of the problem, flipping the state of the road between the two cities in gets as parameters, from usable to unusable.
- *removeCity* – randomly deletes a city from the map, making it unusable by the algorithms; it does so by removing the city from the solutions found by the algorithms.
- *addCityBack* – to simulate the apparition of a new city, the application removes that city before an algorithm is launched, adding it back to the cities list, and algorithm's solutions when the dynamic change is enforced.

4.2.2 City

This class holds the information regarding one city of the TSP input map. Every city is assigned an ID, which uniquely identifies the city. The position on the map is represented with two coordinates, X and Y; since these coordinates need to be scaled to the size of the pictureBox where the city will be printed as a square, the class also holds this information. Finally, a variable of the type Brush is stored, indicating which color with which the city will be drawn on the map.

Methods:

- *City* – class constructor, creating a new city, and setting its ID and coordinates, according to the given parameters.
- *setScaledXY* – sets the values for the scaled coordinates to the parameters given.
- *getScaledX* – returns the scaled X coordinate.
- *getScaledY* – returns the scaled Y coordinate.
- *getId* – returns the ID of the city.
- *getX* – returns the value of the X coordinate.
- *getY* – returns the value of the Y coordinate.
- *getColor* – returns the color with which the city will be drawn.

- *setColor* – sets the color with which the city will be drawn.
- *cityUpdate* – updates all the class variables to the new values.

4.2.3 CircleGenerator

This class deals with generating a number of cities on a circular map. Given an integer number N, it generates N points on the circumference on a circle with radius 1 and the center having the coordinates (2,2); it does so by slicing the 360 degrees space into N regions, according to the formula:

$$\begin{cases} x = R * cos(t) + Xc; \\ y = R * sin(t) + Yc; \end{cases}$$

Where R represents the circle radius, (Xc, Yc) the coordinates of the circle center, and t is the angle at which the new city is located in reference to the center.

The class generates the cities one by one, by splitting the 360 degrees space to the number of cities required. It then stores these cities in a list of cities (List<City>), and also creates a two dimensional matrix holding the distances between each cities. Finally, it stores the maximum and minimum values for each both X and Y coordinates, in order to scale the entire map to the size of the printing area.

Methods:

- *CircleGenerator* – class constructor, that generates the cities, the distance matrix and also scales the coordinates, according to the given parameters.
- *getCities* – returns the list of cities (objects of type City);
- *getDistanceMat* – returns a matrix with the distances between all the cities generated

4.2.4 ReadFile

This class deals with reading a TSP instance from the user given file, constructing a list of objects City, and a distance matrix, similarly to the circle generator class discussed above. The input files should be formatted according to the TSPLIB [8] library; more precisely the formats that provide the coordinates of each city (types GEO or ATT), and not those that provide the distance matrix. This library was chosen since it has a huge amount of TSP instances that can be downloaded and tested, some of these instances being used throughout the development of the current project.

Methods:

- *readFile* – given a filename, and the size of the drawing area, reads the TSP instance file, creating the list of cities, and the distance matrix, and also scales the cities coordinates according to the drawing area. Returns the obtained list of cities.
- *setCities* – updates the list of cities to the new list given as parameter.
- *scaledCities* – built for testing purposes, it returns a list of cities as the scaled version of the list the class already holds.
- *getDistanceMat* – returns the distance matrix.
- *getCities* - returns the list of cities.

4.2.5 Tour

This class holds a possible solution to the given TSP instance, as a list of integers, each integer representing the ID of a city. The order in which the IDs appear in the list, gives the order in which the cities are to be visited in the solution, going through each city only once, and returning to the initial city.

Methods:

- *Tour* – class constructor; when run with no parameter, an empty list is instantiated, the other option being having a list of integers as parameter, to override the current list of city IDs.
- *getTour* – returns the tour held within the object.
- *setTourFromCities* – constructs the list of city IDs from a given array of integers.
- *shuffleTour* – randomly shuffles the list of city IDs.
- *normalize* – rotates the list of integers so that the ID given as parameter is the first integer in the list.

4.2.6 Individual

The class represents a population individual, as instructed by the evolutionary algorithms described in the previous chapter. Being a solution to the problem studied, in our case a route through all the cities of the map, it holds an object of type Tour representing one such solution. This class is also responsible with the main operators of the evolutionary algorithms - crossover and mutation – updating the Tour with the result of these operations.

Methods:

- *Individual* – constructor; creates a new object of type Individual, with the tour given as parameter; it can also be run without a parameter, constructing an empty tour.
- *getTour* – returns the solution as an object of type Tour.
- *getTour* – returns the fitness of the tour.
- *getcost* – returns the total length of the tour.
- *crossover* – returns two objects of type Tour, obtained by breeding (crossover) of the current individual with the individual given as parameter. This process will be discussed at a later stage.
- *mutate* – returns an array of integers (city IDs) by applying the 2-OPT mutation operator to another array of integer given as input.

4.2.7 PheromoneMap

The implementation of the Min-Max Ant System algorithm requires that all the roads between the cities of the TSP instance maintain a level of pheromones, which should be updated according to the solution generated by the ant/ants. This class holds a matrix of doubles, every value representing the pheromone level on the road between two cities. The coordinates of the value in the matrix are the IDs of the cities themselves.

The class is in charge of initializing the pheromone matrix (every road holding the same initial level of pheromones) and updating it with the new values.

Methods:

- *setPheromone* – initializes the pheromone matrix, every road between two cities having the same value.
- *getPheromone* – returns the entire pheromone matrix.
- *getOnePher* – given the IDs of two cities, it returns the pheromone level on the road between them.
- *updatePher* – receiving a list of tours found by the ants at a specific time of the algorithm, it updates the pheromone levels of the entire map, decreasing it on the roads not passed by the ants, and increasing it on the selected edges.

4.2.8 Ant

This class represents a solution found by the MMAS algorithm to the traveling salesperson problem, so an individual of the ant population. For this it holds a list of integers, each integer representing the ID of one city, and the list as a whole the order at which the cities are visited in the solution. This could have similarly been done using an object of type Tour, but it would have added unnecessary computational overhead to the solution construction process that the class is responsible for.

For easiness of constructing new solutions, the class also maintains a list of integers that holds the last feasible and complete solution found by the ant. Most of the time the two lists are identical, the only time they are different is when one class instantiation is in the process of constructing a new solution.

Methods:

- *setTour* - initializes the two lists of integers to the list given as parameter
- *constructTour* – constructs a new tour through all the cities, returning to the initial city, according to the ACO algorithm, using the PheromoneMap given as input, and returns it as a list of city IDs; will be discussed at a later stage.
- *getLastTour* – returns the last solution found by the ant.
- *getFitness* – returns the fitness of the latest solution found.

4.3 Data representation

Various features and data structures were used in the implementation of the project's software application. These choices are briefly discussed in the current section.

Solutions for the input traveling salesperson problem are, as stated above, represented as a list of integers, generically named "Tour", each integer being the ID of one city. The order at which these cities appear in the tour is the order in which the cities are to be visited, according to the proposed solution, and returning to the initial city. This data type was used because, due to its implementation in C#, allows it to be traversed both as an array, with indexes of the value's position in the array, and also as a linked list, going through each element from first to last.

Similarly to routes, the **population** of individuals, respectively ants, are stored as a list of objects of type Individual or Ant, for easiness of access.

Cities hold their coordinates in the two-dimensional space, and are uniquely identified by an ID. Another possible option could have been a graph-like implementation, only storing the distances between the cities, but this option was not chosen due to the difficulties it presents when it comes to the graphical display of the map.

The **distances between cities** are computed based on the cities coordinates, using the Euclidian distance formula, and stored in a matrix of doubles; the value at position [1][2], for example, represents the distance between the city with ID "1" and the city with ID "2". The same data structure is used for storing the values that represent pheromone levels on each road of the map, an array of double values in the (0,1) interval.

Fitness, being a representation of how good a provided solution is, needs to have higher values for better solutions. Since, in the case of TSP, the total distance of the route through all the cities represents the quality of a solution, and the shorter the length, the better the solution, the choice was made that fitness is implemented as the negative value of the total distance. The closer the fitness value is to zero, the better the solution.

Generating **random initial solutions** for the evolutionary algorithms, and also for SA and RLS, is done by creating a list with all the cities in the system, and then shuffling it for each random solution needed. Shuffling is done by randomly selecting a position from the list, removing the city at that position, and adding it to the resulted list, and repeating this process until the initial list is empty.

4.4 Algorithms

4.4.1 Genetic Algorithm

All algorithms are implemented in the onClick function of the “Start” button of the application, and they are run according to the selection of the radio buttons of the user interface. All the algorithms have a similar initialization phase, and share some of the data structures, in order to maintain consistency across them, and to reduce the usage of computer resources.

The prerequisites of the Genetic Algorithm are constructing the population of randomly generated solutions. A number of 100 such individuals were chosen to represent the population, each being an object of type Individual, and having a random initial solution to the problem at hand, in the form of a Tour object. Constructing such a tour is done by taking the list of cities that was given as input of the problem, and building a new list of integer values, each value representing the ID of a unique city. After this list is constructed, it is used for instantiating a new Tour object. The tour is then shuffled, according to the method described in the previous section, and is added to the new individual. The process is repeated for each of the 100 individuals of the population, thus having 100 randomly generated solutions to feed the algorithm itself.

At every step of the algorithm an entire new population is created, with possible new solutions, is created with the aid of the selection, crossover and mutation operators. **Selection** of the individuals that get to mate and produce offspring was implemented using the Tournament Selection technique. Two such Tournament Selection processes are ran, for choosing the two individuals that reproduce. The size of a tournament is 10, and the best fitted solution out of the 10 solutions is chosen to become a parent. A starting position in the population list is selected, and the tournament selection is applied to the first 10 individuals after that starting point, using a for-loop that goes through the 10 individuals, finds the fitness of each, and selects the best one. Two parents are selected with this method, and then they are bread, generating two new offspring, by the use of the crossover operator. The process is repeated N/2 times, where N is the population size, generating an entire new population, with the same size as their parents’.

Crossover tries to use as much of the parent’s genes as possible, by using the permutation crossover [10]. It is implemented similarly to the classical Two Point Crossover, by randomly selecting a section from one parent’s solution. Two random integers are generated, representing the start and end points in the firts parent’s solution, so that the selected section length is at least a third of the initial list's length. This section is then copied in the first child's chromosome, and then the empty spaces left in the chromosome (to the left and right of the section) are filled with cities from the second parent (in the order they appear in its chromosome), cities that haven't yet been included in the solution. The second child is created using the same process, copying the section from the second parent, and then filling with the genes of the first parent.

Each newly constructed child can suffer a mutation, with a probability of 1%. This probability was selected because, since there are 100 new individuals generated, it respects the algorithm's directions that an average of one mutation occurs at each step. The **2-OPT Mutation** described in the first part of this thesis removes two edges from a route, and the reconnects the two obtained tour halves in the only feasible way remaining. On a representation of routes as lists of cities, this mutation is implemented as follows: two distinct cities are randomly selected, and then it is decided which city comes first and which comes last in the input tour. The section of tour contained between the two selected cities (the road between first and last city) is inverted in the final solution, to represent the road back from the last city towards the first city. The remaining sections (the ones before and after the inverted section) are copied as they appear in the initial tour.

Once an entire new population is created, it will replace the population from the previous step. All the solutions generated are then analyzed in terms of fitness, the best fitness being printed out to the user, so that he can notice the convergence of the method. Moreover, this best fitness is used in the **stopping criterion**: the algorithm stops when M^2 consecutive algorithm steps were finalized without providing an improvement to the solution, where M represents the total number of cities in the input problem. After the algorithm stops, the best fitted solution is selected and printed to the user.

4.4.2 (1+1) EA

(1+1) EA is a version of Genetic Algorithms that uses a population size of one, one individual that is repeatedly mutated to create new solutions. Due to this population size, the algorithm doesn't provide any selection or crossover operators, relying solely on mutations. Even though it also uses the 2-OPT mutation operator, (1+1) EA has a different behavior than GA: it allows for multiple mutations to occur in the same algorithm step, using Poisson Distribution to determine the number of their number.

Data initialization is similar to GA; a population of random initial solutions is generated. This time, a single array with all the cities is created, then randomly shuffled, and used to create an object of type Tour. Then, one individual is instantiated with this Tour. The Poisson Distribution for 10 events, with one average event per turn, is also computed and stored into an array at this phase. The Poisson Distribution array is composed of values in the [0,1] interval, more precisely the first value representing the probability that one event occurs at a set time – 0.36 – and the following values decreasing abruptly.

Once these prerequisites are met, the algorithm is then run. At each step a new individual is created as a copy of the individual from the previous step. A random number in the [0,1] interval is created, and then used to determine how many mutations should be applied consecutively. This is done by determining under which interval the random value falls, in the [0,0.36] space split into smaller intervals by the Poisson Distribution values. For example, if the random number is 0.30, this falls into the interval (0.18, 0.36] represented by the probabilities that one, respectively two events will occur; in this case, one mutation will be applied. If the value falls between 0.06 and 0.18, then two mutations are applied, and so one.

Since the random number generator can return values higher than 0.36, and we want to ensure that at least one mutation is enforced at each run, one extra mutation is applied at each step, regardless of the random number.

Once the new solution is created, its fitness is determined. If the new solution has a better fitness than the previous, if the newly created offspring has a shorter route through all the cities than its parent, than the parent individual is replaced by the child, the algorithm continuing its run with the new solution. If that is not the case, and the child is worse fit than its parent, the parent's the algorithm continues with the parent's tour, ignoring the newly created one. Stopping criterion is the same as GA's.

4.4.3 Randomized local search

The simplest algorithm implemented for this project, for comparison purposes, randomized local search, can be seen as a simpler version of (1+1) EA, a random initial solution is created, and new solutions are generated through mutations only; RLS has one important difference: only one mutation is applied at each step.

Due to these obvious similarities, the implementation of RLS uses the same data structures and objects as (1+1) EA and GA. A random initial solution is created by storing all the IDs of all the cities in the input problem in an array, then shuffling this array, creating an object of type Tour from it, and assigning this

Tour to an Individual object. This represents the prerequisite of the algorithm.

At each step of the “while” loop of the algorithm, a new Tour is created as a copy of the solution from the previous steps. One 2-OPT mutation is applied to the new Tour, using the same method as the two algorithms above. This solution is then compared to the previous solution, in terms of fitness. If the new route through the cities has a better fitness (a shorter length), it will replace the previous solution, and it will be used for the following steps. If not, then this route is ignored. As we can see, RLS has a classical hill-climbing behavior, having no means of escaping states of local optimum.

The algorithm stops when no better fit solutions are found during M^2 consecutive steps, where M represents the total number of cities in the given input.

4.4.4 Simulated Annealing

Simulated annealing is also very similar to (1+1) EA, as it works with only one solution at each time, and tries to improve on through 2-OPT mutations only. But SA is more focused on escaping local maximum states, and does it by allowing seemingly bad solutions to be accepted. It does so by replicating the annealing process of metal works, more precisely by storing a variable called temperature, which starts off with high values and it is then gradually increased; when the temperature is still high, solutions that are worse fit than the previous ones are accepted with a higher probability. Once the temperature cools off, taking bad solutions becomes almost impossible.

Before actually running the algorithm, an initial solution is selected randomly, using the same process as (1+1) EA or RLS. A variable called “temperature” is also created, having a very high initial value - M^2 where M represents the number of cities.

At each step the algorithm creates a new solution by applying a 2-OPT mutation to the previous tour. If the new tour is shorter than the old, it will overwrite it, and the algorithm continues to the next step. If the fitness is worse, then the fitness difference between the old and new solutions is computed. A random number R, in the [0,1] interval is generated, and if R is lower than the exponential function applied to the division of the fitness difference over the temperature (if $R < exp(\frac{difference}{temperature})$), then the old solution is overwritten with the new one. If this also fails, the algorithm continues with the old solution.

After each new solution was generated, and the comparisons from above were applied, the temperature is decreased: $temperature = temperature * (1 - \frac{1}{M})^{step}$, where “step” is the step count of the algorithm (the number of generations created so far), and M is still the cities count.

4.4.5 Min-Max Ant System

Ant Colony Optimization (and its extension MMAS) generate solutions by simulating the way ants find their way towards food sources and back to their colony. In the initial phase random walks are made, constructing a solution step by step. Once a solution was found, the “road” towards that solution is marked with pheromones. The next walks get guided by these pheromone trails, making it more probable that each ant reuses segments of the solutions with high concentration of pheromones. In time the algorithm converges towards one solution that closely approximates the global optimum.

In the beginning the ant population is initialized. In the case of MMAS, only one ant is used, and it is instantiated with a randomly selected solution, in a similar way as the evolutionary algorithms; this will represent the “last tour” – each ant stores the previously found tour, to know the IDs of the cities that it needs to pass through. A pheromone map is also created at this phase, as an array of double values, each value representing the pheromone level on the direct road between two cities; the initial value of pheromones, on all the direct roads between cities, is $\frac{1}{M}$, M being the number of cities.

The algorithm creates a new ant at each step, and calls for it to find a new walk through all the cities of the input problem. This new solution is generated as follows: a starting city is selected, and then removed from the list of cities to be visited. For the remaining cities the pheromones of the roads between the current city and them are summed. Using this sum and the weight each pheromone has in constructing this pheromones sum, the [0,1] interval is split into multiple segments, each segment having its size proportional to the pheromone level of the road it represents. A random number R is generated in the [0,1] interval, and then the algorithm looks for the segment R falls into on the [0,1] split interval. Once the correct interval is found, the correspondent road is added to the solution, the newly visited city being removed from the list of cities to be visited, and becomes the current location. The process then repeats itself, from the new location, until the solution contains all the cities of the problem, and the list of cities to be visited is empty.

When the new ant has finished constructing its solution, the fitness of this route is computed and compared to the best solution found so far, replacing it if it represents a shorter tour through the cities. Regardless of the result of this comparison, the pheromone map is updated as follows: the pheromone level of each segment contained in the newly found solution is increased by a constant value ρ, and the pheromones of the rest of the segments are decreased by multiplying the previous value with $(1-\rho)$. In this project, through testing, the best value for ρ was found to be 0.05.

As previously said, MMAS has a minimum and maximum value for the pheromone concentration on each direct road between two cities. Because of that, when updating the pheromones, its values are limited to the minimum and maximum allowed levels: $\frac{1}{M^2}$ as the lower boundary, and $(1-\frac{1}{M^2})$ as the upper boundary; again, M represents the number of cities.

$$Pher_{i,j} = \begin{cases} Max(((1-ro)*Pher_{i,j}), \frac{1}{M^2}), \; segment\ not\ in\ solution \\ Min((Pher_{i,j}+ro), (1-\frac{1}{M^2})), segment\ in\ solution \end{cases}$$

The algorithm continues with the new pheromone map until the stopping criterion is reached - when no better fit solutions are found.

4.5 Dynamic changes

4.5.1 Interchange cities

This mutation swaps two cities, the first city taking the location of the second city, and the other way around, and what interests us is analyzing the way the algorithms adapt to this change, and how quick they can adapt to this new situation, and find the optimum, when starting from a solution to the unchanged problem.

The implementation of this mutation is quite simple: the coordinates of the two cities are interchanged, with the use of an auxiliary object of type City. Then the matrix containing the distances between each of the two cities, and the rest of the cities are also updated to the new locations, by interchanging the line and column representing the first city, with the line and column of the second city.

4.5.2 Congested road

The "congested road" tries to simulate a real-life situation, where a road gets so much traffic during some time intervals, that it can no longer be used, and has normal traffic during other intervals. What interests is if the algorithms decide to ignore the set road, due to its lack of dependency, or they choose to select it because it is contained in the shortest route, and is sometimes usable. Another point of interest is how the percentage of time that the road is unusable influences if the algorithm chooses it for the solution or not.

The way this dynamic change is implemented is that it selects two cities, c1 and c2, and a frequency of congestion appearance. For example, the road becomes congested at every 10 steps of the algorithm, 90% of the time being usable. Computing the remainder of the division between the step count and 10, if the remainder is 0, for example, the road gets congested. What that means is that the direct road between cities c1 and c2, in the distance matrix, is set to a very high value (the maximum value of a double number, in our case), when the road is congested, and to its normal values, when it is not.

4.5.3 Deleting a city

One of the dynamic changes that are particularly useful for computer related problems is deleting a city while the algorithm is still running. A common problem in network routing is the failure of one network node (a server, for example). The algorithm should be able to adapt to this, by finding different ways of getting the information through the network, efficiently.

Removing a city from the cities list is done as follows: a random number is generated, between 1 and M, and the city with the ID corresponding to that specific number is removed from the list of cities of the initial problem. To prevent any failures of the algorithm that is currently running, the ID of the city is also removed from any current solutions of the algorithms, removing it from the chromosomes of the GA population, for example, or from the last tour of the ant.

4.5.4 Adding a city

Another dynamic change that is useful in computer related problems is the apparition of a new city, like, for example, adding a new server to the network. In order to have a city to be added, the software application of this thesis first removes a random city from the list, before the algorithm is launched, then launches the algorithm, and inserts the city back, at the appropriate time.

The initial removal of the city is done using the same technique as the “Deleting a city" dynamic change from above, with the addition that the removed city is stored in a variable, to be added back. Actually adding the city while the algorithm is running, involves first putting the city back in the initial list of cities, but also adding it to the solutions found by the algorithm, so that it can continue improving on them.

As suggested by authors [9], the algorithms can react better to this dynamic change if the new city is added to their solutions in the right place – meaning in the position where the resulting tour has the highest fitness. To replicate this, every possible position where the city can be added to the tours are tested, one by one, and their fitness is computed, storing along the way only the position providing the maximum fitness. After all possibilities were tested, the new city is added to the position providing the highest fitness.

Chapter 5

Test results

The present chapter deals with the testing part of the project, discussing the various tests applied to the software application, the implementation choices made, and the results found. The chapter is split into two sections, dealing with the two different testing phases of the application: testing the way the algorithms work in a static environment, with no dynamic changes, and then testing the results they generate for the four dynamic changes.

For speed and convenience, some changes were applied to the software application, enabling it to do batch testing, running the same test, on the same input problem, multiple times. For each test configuration from those that will be discussed in this chapter, 100 runs were made to the same input problem, for each algorithm. The changes made to the application enabled it to store the results in a file, one file for each algorithm, listing the solutions found at each run, the length of each solution, and also the number of steps it took the algorithm to reach it.

For testing in a dynamic environment, the same changes were applied, with the addition that the output file contains the tour, length and step count for the best solution found before applying the change, and also the tour, length and step count of the best solution found after the change occurred.

This chapter briefly discussed the results of each of the tests the application was subjected to, but all the results found are displayed in a graphical nature in the appendices of the present report.

5.1 Static environment

The first phase of testing is responsible with finding out how good solutions the five algorithms offer, when run in on a static problem, with no dynamic changes occurring. This gives a good basis for comparison of the accuracy of the solutions the algorithms provide in the present implementation, allowing them to be ranked on how fit they are to the traveling salesperson problem. This phase can also give good indications of how well the algorithms will adapt to the dynamic environment.

Four tests were run, using the circle generator tool: 100 test runs for each algorithm, on circles with 10, 25, 50 and 100 cities created along the circumference of a circle. The circle generator was preferred for this phase, because the optimum tour is known beforehand, thus allowing us to compute how close the solutions found are to this optimum.

The type of results that were searched for were the success rate of the algorithms to find the optimum solution, the average distance from this optimum, and also the number of steps it took the algorithm to find the best solution, at each step.

The overview these tests provide is that RLS and SA provide better solution in the longer run, and for bigger maps, remaining with a high success rate to hitting the optimum as the problem size increases. Genetic Algorithm proves to be very good on smaller instances, but its performance drops importantly as the number of cities increases. Min Max Ant System, even though it provides decent approximations for smaller problems, proves to be in the last place when it comes to performance, at least in this implementation.

5.1.1 Circular map with 10 cities

The first and simplest of the tests run in the static environment analyzes the solutions found by the algorithms on a map containing 10 cities positioned on the circumference of a circle. The optimal solution is known, and has a total length of the rout of 6.19345368405876.

Appendix I shows the success rate of each algorithm to reach the optimum solution. In this case Genetic Algorithm offers the best results, finding the optimum 89% of the time. Simulated Annealing follows in the top, with 85% of successful hits. Surprisingly Randomized Local Search outranks (1+1) Evolutionary Algorithm, by a big margin: it find the optimum solution 84% of the time, whilst (1+1) EA finds it in only 76 runs out of 100.

The implementation of MMAS shows its limitations, only hitting the correct answer 42% of the time.

When we look on the average length of the best solutions found by the algorithm, shown in *Appendix II* we see that the differences are not so high: except for MMAS, which offers a somewhat bad average approximation of the optimum, the results of the other four algorithms only differ by approximately 0.1, 0.2 in their average tour lengths. Moreover, this graph shows the same ranking as the average hit rate.

Looking at the average distance from the optimum solution graph (*Appendix III*), we again see that Genetic Algorithm is the algorithm having the closest solutions to the optimum. Even though Simulated Annealing has a bigger success rate than Randomized Local Search, on average its solutions are farther away from the optimum then RLS.

The average number of steps to find a solution, presented in *Appendix IV* shows GA to be the fastest algorithm, from a step count point of view, needing only an average of 61.02 steps to find its final solution. RLS follows in the top, with 132.25 steps, and the other three algorithms average around 140 steps.

Appendix V displays a graphical representation of the variation of lengths in the solutions provided by the algorithms, over the first 50 runs. The most often, and biggest spikes in the graph (showing a high length of the tour) is corresponding to GA. Simulated Annealing shows bigger spikes (worse fitness) than (1+1) EA, but the second algorithm has more frequent, smaller, spikes.

5.1.2 Circular map with 25 cities

The second test was made on another circular map, containing 25 cities. The best tour length is 6.26556527863881.

The first surprise of this second test is increase in success rate for Randomized Local Search, while the success rate of all the other algorithms drop(*Appendix VI*). RLS manages to gain the first rank, with a staggering success rate of 90%, followed by Genetic Algorithm, who found the optimum only 84% of the times.

Simulated Annealing and (1+1) Evolutionary Algorithm follow, only finding the optimum 81, respectively 73 times out of 100. The biggest drop is present for MMAS, which only found the optimum 17 times during this test.

The average tour length (*Appendix VII*) shows a big increase in the length of the solutions found by MMAS, and a difference of lengths similar to the 10 cities test, for the rest of the algorithms.

The average distance from optimum (*Appendix VIII*) shows RLS to having only an average distance of 0.06 from the optimum, followed by Genetic Algorithm with 0.177. MMAS offers solutions with an average length almost double than the optimum.

Genetic Algorithm has again found its solutions in the lowest number of steps (*Appendix IX*), 924.7 on average. RLS and SA closely follow, with around 1140 steps. (1+1) EA and MMAS took a lot longer, 1475.88 for the first, and 1865.93, double than GA, for Min-Max Ant System.

The tour length variation over the first 50 steps (*Appendix X*) shows a leveling of results for RLS, (1+1) EA, and SA, with few spikes from GA. MMAS displays a high variation, and a bigger difference from the optimum, than the previous steps.

Overall, we see a drop in performance from all algorithms, except RLS, which manages to offer better solutions for this bigger map. The performance drop of MMAS is the most worrying, its solutions being almost double in length than the optimum.

5.1.3 Circular map with 50 cities

The next tests are run on a map consisting of 50 cities generated along the circumference of a circle, and having an optimum length of 6.2787345387654.

This time it was SA's turn to increase its success rate (*Appedix XI*), reaching the optimum solution 84% of the time. Tying with SA on the first place, RLS again provides good solutions, showing a success rate of 84% as well. GA and (1+1) EA follow, in this order, dropping their rate to 79, respectively 67 hits of the optimum. MMAS again shows the biggest performance drop, only reaching the optimum 8% of the time.

Except MMAS, who shows and average tour length (*Appendix XII*) of almost triple to the size of the optimum, the other algorithms level up on their solutions, their lengths differing by less than 0.1.

RLS provides the closest solutions to the optimum, as *Appendix XIII* shows, its average tour length being higher than the optimum with only 0.082821. Simulated Annealing is also very close, with 0.112337, followed by Genetic Algorithm and (1+1) EA.

The average number of steps graph (*Appendix XIV*) again shows GA to be the fastest from this point of view (even though its actual running time is a lot higher than the other algorithms), finding its solutions in around 4643.99 steps. Randomized Local Search closes its distance from GA, with 4950 steps. Min-Max Ant System shows a bad performance, having an average step count almost triple than GA.

5.1.4 Circular map with 100 cities

The last test is also the most time consuming and largest in the number of cities: a map consisting of 100 cities positioned in a circle. The length of the optimum solutions is 6.28206967118347.

The success rate of the algorithms (*Appendix XV*) drops again, but shows a constant trend, similar to the previous tests: Simulated Annealing remains the algorithm with the most optimum solutions found, 83 out of 100, followed by Randomized Local Search with 78. (1+1) EA manages to outrank GA this time, having a success rate of 65%, whilst the Genetic Algorithm only 62%. This time MMAS doesn't manage to find the optimum at all, never reaching the tour with the shortest length.

The average tour length displayed in *Appendix XVI* shows another decrease of performance for MMAS, this time not as abruptly as the previous steps: 21.443484 average lengths. The other four algorithms show average lengths of around 6.4, GA providing the smallest lengths, followed by SA and RLS. *Appendix XVII*, the average distance from optimum, backs up this data, having the same ranking of the algorithms.

Finally, the average number of steps to find the solution (*Appendix XVIII*) shows RLS to find its solutions the quickest (around 24888.89 steps), followed by SA and GA. Min Max Ant System is very close to them, this time not having such a big increase in the number of steps. (1+1) EA provides the worst performance, taking around 37668 steps to decide on one solution.

5.2 Dynamic Environment

The second phase of the tests consisted in determining how well the algorithms adapt to the four dynamic changes implemented, when these occur when the algorithm is still running.

A set number of 100 tests were run for each algorithm, for each dynamic change. Two sets of 100 tests were run for the congested road and the interchanges of cities dynamic changes, and only one set for adding and deleting a city.

The time of applying a dynamic change was set to be the time when the algorithm has reached a final solution according to the static implementation (N*N steps passed without an improvement to the solution, where N is the number of cities). One such dynamic change is applied at that moment, with the exception of congested road, which has a different behavior: after the algorithms has reached N*N steps without improving on the solution, one road begins to suffer from congestion every 10th step.

Different algorithms offer different behavior, as the following results will show.

5.2.1 Interchange cities on a circle map of 10 points

The first dynamic change that was tested is the interchanging of locations of two cities. One such change is applied per run, and what the tests try to find is whether the algorithms manage to adapt to this change that increases the length of its tour, if the tour is close to optimum.

The test consists in a map of 10 cities positioned around a circle, with a optimum tour length of 6.19345368405876.

Appendix XIX shows the success rate of the algorithm to reach the optimal solution, after two cities have swapped places. The results show Genetic Algorithm to be the highest adaptable, with 91% success rate, followed by Simulated Annealing with 88% and (1+1) EA, respectively RLS, which are tied for the third place, with 85 optimum solutions found. MMAS performs badly, only hitting the optimum 15% of the time.

Appendix XX and *Appendix XXI* show the average distance from the optimum before the change was applied, respectively the average distance after the change was applied. Simulated Annealing and Randomized Local Search show a large increase of their distance from the optimum, after the two cities were interchanged. Genetic Algorithm also provides worse solutions after the change, but at a lower increase level. (1+1) EA and Min Max Ant System provide a surprisingly decrease of their average length after the change was applied, which could suggest that the change itself improved on their solutions some of the time.

The difference between the best solutions found before and after the dynamic change was applied can be seen more accurately in *Appendix XXII.*

5.2.2 Interchange cities on a circle map of 25 points

Another test for adaptability of this dynamic change was made, using a map of 25 cities in a circle, with the shortest tour having a length of 6.2655652786388.

Appendix XXIII shows the average success rate for hitting the optimum solution after the cities swapping change occurred. Simulated Annealing is the leader of this ranking, proving to be highly adaptable, and finding the optimum 92% of the times, even though two cities were interchanged. GA and RLS also show good adaptability, with 81% success rate both, being closely followed by (1+1) EA with 76 hits. MMAS again performs badly, finding the optimum only 5 times. The average distances from the optimum, before and after the change was applied (*Appendix XXIV* and *Appendix XXV*) show distances from the optimum comparable to the tests on the same map, in a static environment, all of the showing high adaptability. Simulated Annealing, as expected from the previous graph, has the closest distance from the optimum, around 0.11, showing an improvement from the first part of the run, before the change. RLS and GA also provide close solutions, both before and after the dynamic change was applied. *Appendix XXVI* shows the average difference in tour lengths before and after the interchange was applied. The graph shows and increase in length for (1+1)EA, GA and RLS, a minor decrease in length for SA. More importantly, it again shows a very high decrease in length for the solutions provided by MMAS.

5.2.3 Congested road on a circle map of 10 points

Testing the congested road dynamic change tries to determine whether the algorithms can adapt to having a road being congested one every K runs (K was set to 10 in this testing phase), and if it finds decent detours, or it gets stuck with that road, even though it gets unusable from time to time. Also, it interests the number of steps it takes each algorithm to decide to select or not that specific road.

The road to be congested was closely selected to ensure it influences the outcome: a segment of the optimum tour was selected to be congested for these tests.

Appendix XXVII shows the percentage of time an algorithm chose to include the congested road in their optimal solution. (1+1) EA and MMAS showed to be more inclined to choose detours than the congested road, only selecting it 6/

The average step to find a solution graph (*Appendix XXVIII*) shows Genetic Algorithm and Simulated Annealing, the two algorithms, which decided to include the congested road most often, also managed to make this decision the fastest: 130, respectively 131 steps. (1+1) EA, on the other hand, took the longest number of steps to find its solutions, an average of 754.72 steps. RLS took around 429 steps, and MMAS 226 steps.

5.2.4 Congested road on a circle map of 25 points

The second test for the congested road dynamic change was run on the 25 cities circular map.

Appendix XXIX shows the number of times an algorithm included the congested road in their solution. Remarkably, (1+1) EA and MMAS continued on their detour-preferring behavior, and none of their solutions included that road. The proportions remain the same as the previous step; GA leads with 32% of the time including the road, SA with 24% and RLS with 5%.

The average step count (*Appendix XXX*) shows the same proportions as the 10 cities test: GA and SA were the quickest to find their solutions, in approximate 803 steps the first and 717 the second. The longest number of steps was used by (1+1) EA (6176.66 on average), followed by RLS and MMAS (3448, respectively 1732 steps on average).

5.2.5 Deleting a city

The next test deals with determining the adaptability of the algorithms to the case where one city disappears from the map, while the algorithm is still running. Implementing this dynamic change on a circular map would have given no useful results, as applying it to an already optimal solution, leaves the system with another optimal solution, and the algorithms have no adapting to do.

In order to avoid this situation, a different input problem was used, one that

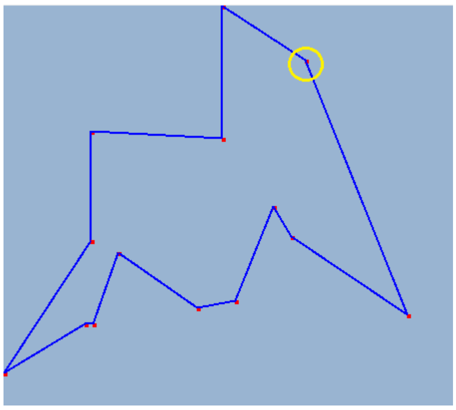

Figure 5.1: Optimum tour with the city to be deleted

provides a different optimal tour when a city is deleted. When deleting the city from the image, if the algorithm has previously found the optimal solution, it needs to change it in order to find the new optimal solution, that doesn't include that specific city. The same thing applies to the case where this specific city is added to the map while the algorithm is still running.

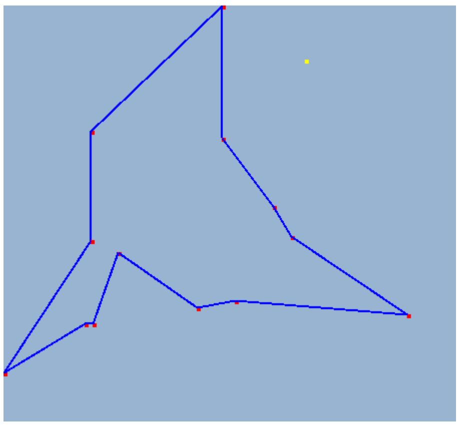

Figure 5.2: Optimum tour without the city

Appendix XXXI shows the average success rate of finding the optimal solution after the city was deleted. This graph shows Genetic Algorithm to be the highest adaptable algorithm, having a success rate of 43%. Simulated Annealing, (1+1) EA, and Randomized Local Search follow, finding the optimum around 30 times out of 100. MMAS performs extremely bad, having a success rate of 2%. This is also expressed in the graph showing the difference from the optimum solution, after the city was deleted (*Appendix XXXII*): MMAS offers solutions far away from the optimum, whilst the closest solutions are offered by GA and SA.

The average length decrease between the solutions found before and after the deletion of a city is expressed in *Appendix XXXIII*. The highest decrease is present for MMAS, approx. 4.05, the decrease of the other algorithms varying from 2.319, for GA, to 2.64 for (1+1) EA.

The step count graphic in *Appendix XXXIV* shows algorithms to have similar steps to finding a new solution, after the dynamic change was applied. SA, GA and RLS all have an average step count, from the moment the dynamic change was applied, to the moment a solution was selected, of around 277. At the bottom of the scale, (1+1) EA needs 303 steps.

5.2.6 Adding a city

Testing the addition of a city was made using the same map, and the same city as the deletion of a city dynamic change. The city is initially deleted, before the algorithm starts, and then is added back in when the algorithm decides on one solution.

All the algorithms showed a very low success rate of reaching the optimal solution after the dynamic change was applied (below 4%), thus a graph for it was not needed. Only (1+1) EA and GA managed to find the optimal, the first algorithm finding it 2 times out of 100, the second finding it 4 times.

The length difference from the optimum (*Appendix XXXV*) , after the change was applied, shows that the algorithms have performed in a similar fashion, except MMAS. Min Max Ant System showed a difference of 3.68158, while the other algorithms having longer tours than the optimum, with only around 1.1.

The length increase between the solutions found before, and the ones found after adding a new city (*Appendix XXXVI*) shows MMAS to have the smallest tour length increase 0.979. Genetic Algorithm's solutions were the ones that suffered the highest length increase, on average 2.933633.

Appendix XXXVII shows the number of steps it took an algorithm to decide on a new solution, after the dynamic change was applied. The lowest number of steps were used by MMAS (around 253), and the highest by (1+1) EA, 285 steps.

Chapter 6

Conclusions

The software application implemented throughout this project allows users to test the five implemented algorithms (Genetic Algorithm, (1+1) Evolutionary Algorithm, Simulated Annealing, Randomized Local Search and Min-Max Ant System), with user-selected input problems, or with generated circular maps. The application allows the user to select whether the algorithm will run in a static or a dynamic environment, and it offers both a graphical display of the tours found, and also the length of each solutions found along the way.

The present report also deals with analyzing and comparing the five algorithms. On a static environment, Genetic Algorithm offers good solutions for small scale problems, and in a relatively low number of steps. As the scale of the problems increases, the performance of each algorithm drops, but two algorithms maintain a decent success rate of finding the optimum solution, and offer a decent approximation of the solution at every run: Simulated Annealing and Randomized Local Search. RLS seems actually to perform better on larger-scale problems then smaller scale.

In a dynamic environment, when interchanging of cities occurs, Simulated Annealing and RLS also have a good adaptability rate, being able to react to the change and find the good approximations once more. They, together with GA, are the best options for such an environment.

When congestion of roads occur, RLS and (1+1) EA have the highest rate of finding a decent detour around that congested road, as they seem to take consider consistency more important than tour length. Even though its solutions don't approximate the optimum so well, MMAS also has a good rate of finding such detours.

In an environment where cities might get deleted, Genetic Algorithm seems to be the best solution, as it has a very good rate of finding the optimum, or at least a good approximation of it, even after such a change occurs. Except MMAS, the other algorithms also show decent adaptability to this change.

Adding a new city to the problem seems to be a problem too complex for the five algorithms. Even though the algorithms are helped by inserting the city in the place that seems the best option, they only manage to find decent approximations of the optimum after a new city is added, but no the actual optimum solution. Genetic Algorithm and Simulated Annealing are again very adaptable, as they offer the best approximation.

Appendix I

Appendix I

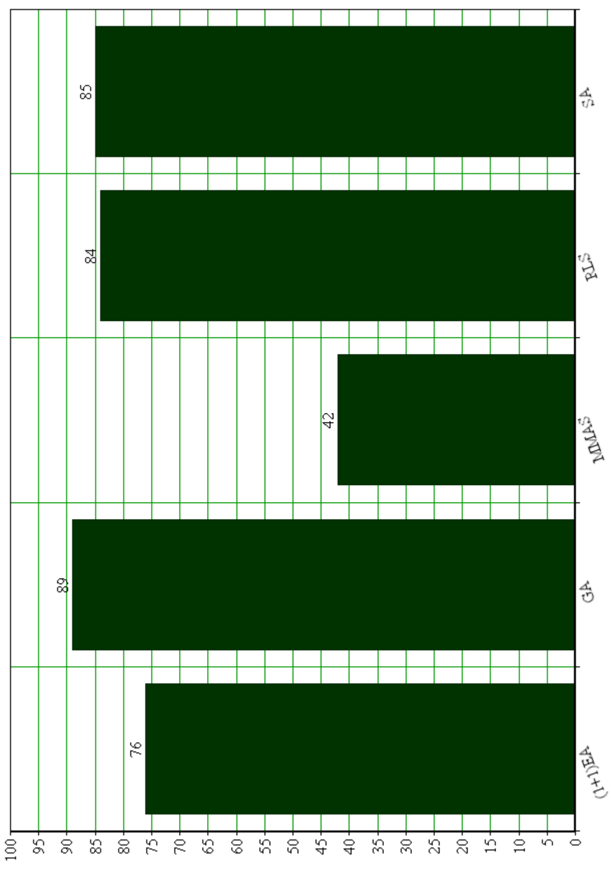

Figure I.1: Static environment - Circle 10 - success rate

Appendix II

Appendix II

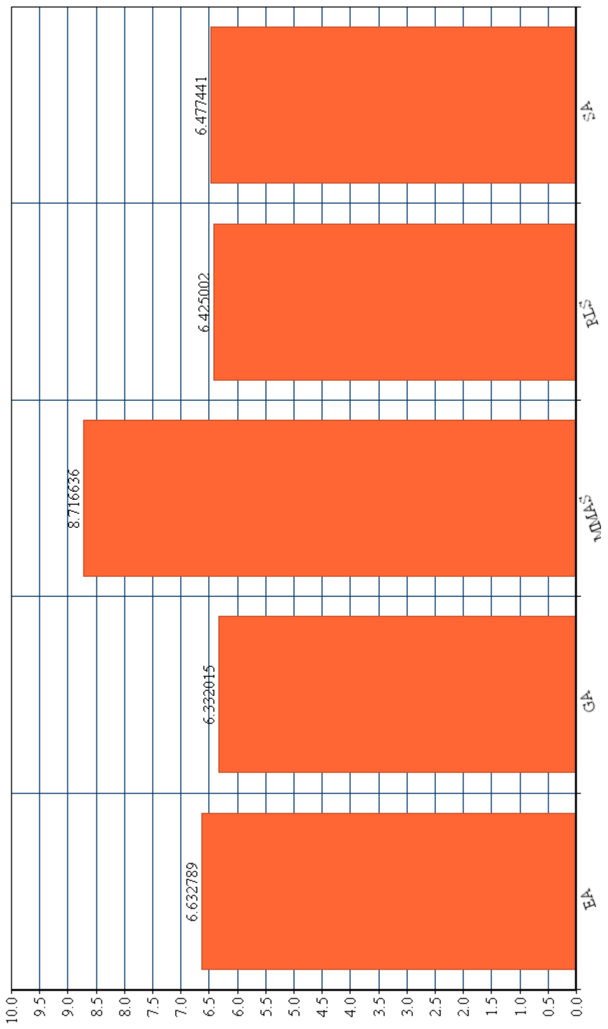

Figure II.1: Static environment - Circle 10 - average tour length

Appendix III

Appendix III

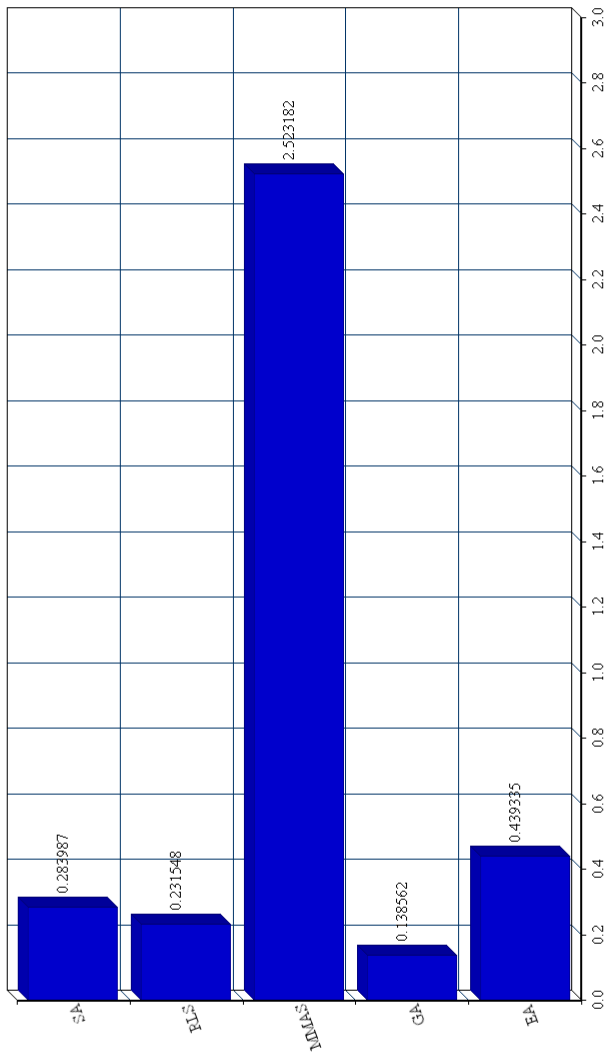

Figure III.1: Static environment - Circle 10 - average distance from optimum

Appendix IV

Appendix IV

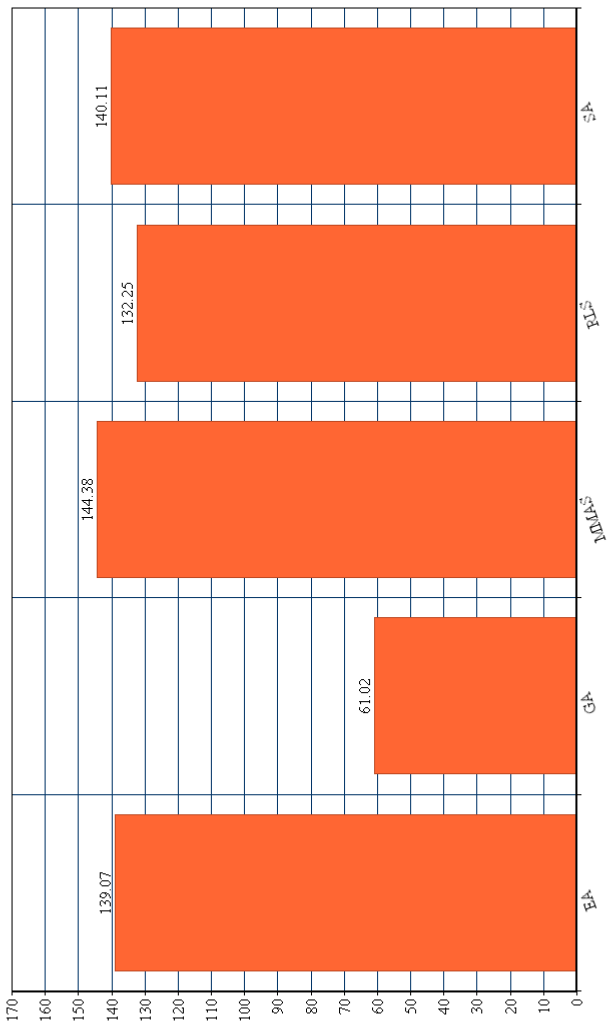

Figure IV.1: Static environment - Circle 10 - average number of steps

Appendix V

Appendix V

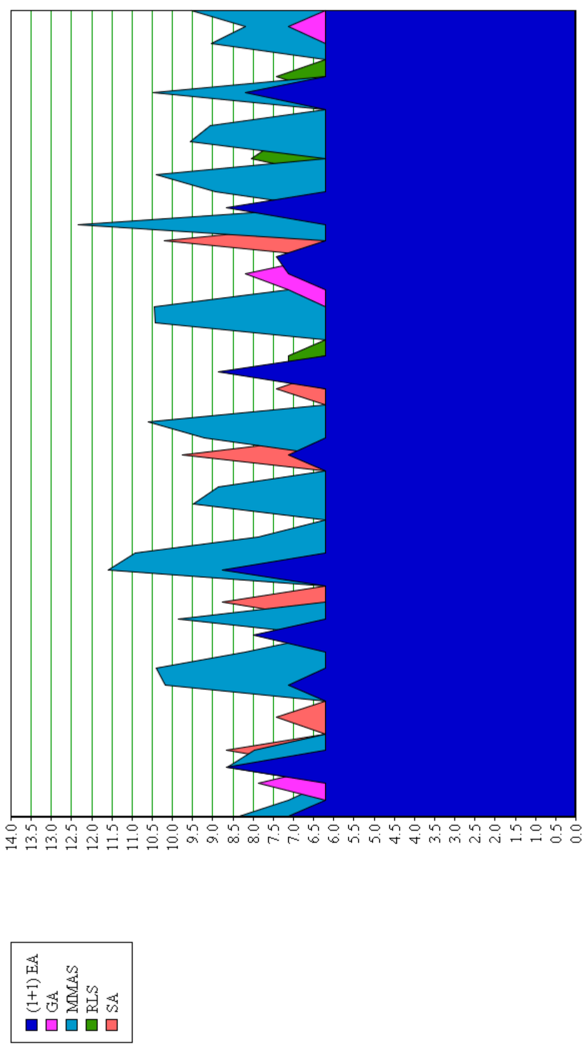

Figure V.1: Static environment - Circle 10 - length variation over 50 steps

Appendix VI

Appendix VI

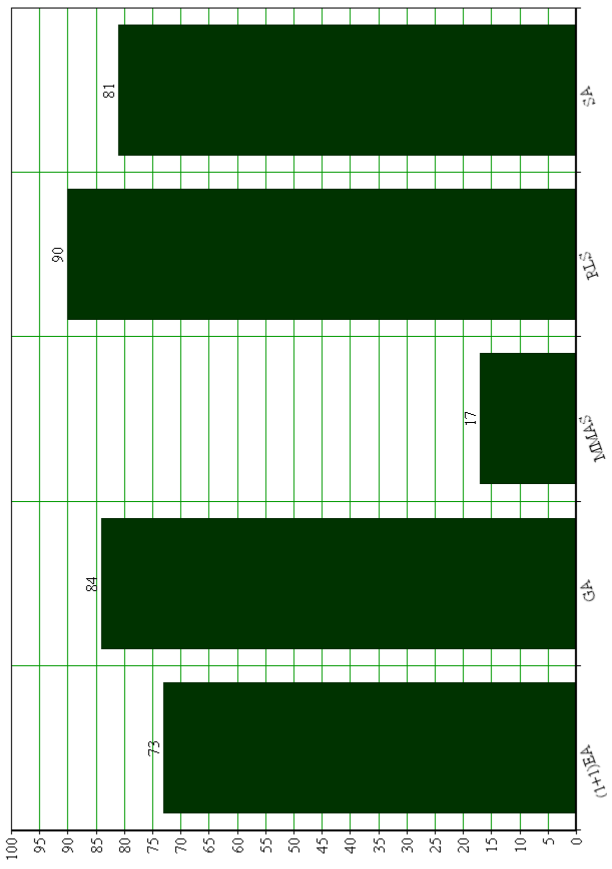

Figure VI.1: Static environment - Circle 25 - success rate

Appendix VII

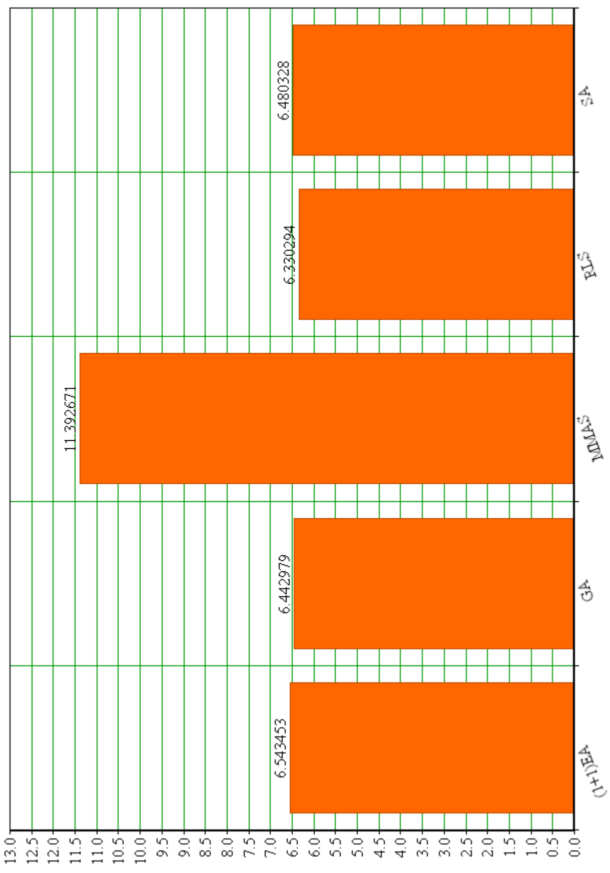

Figure VII.1: Static environment - Circle 25 - average tour length

Appendix VIII

Appendix VIII

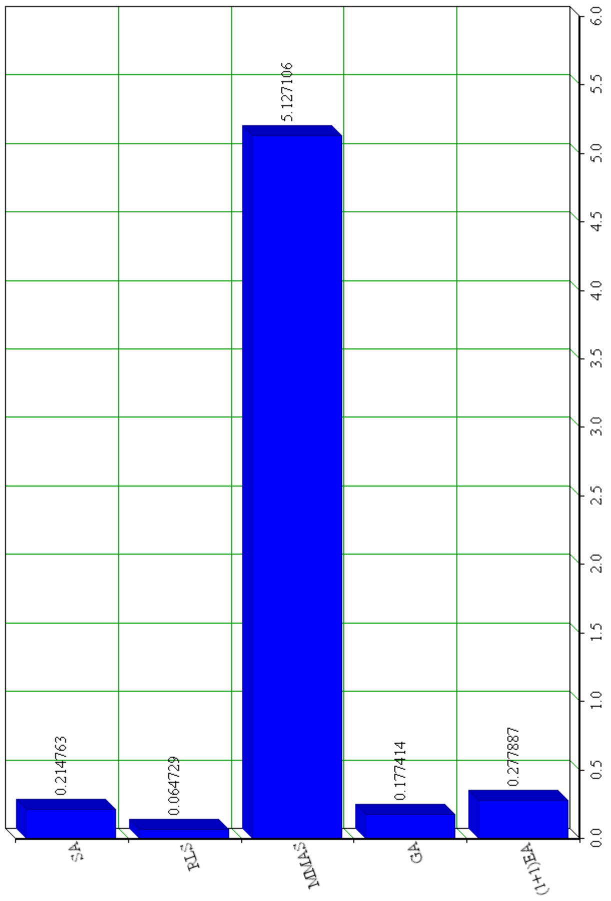

Figure VIII.1: Static environment - Circle 25 - average distance from optimum

Appendix IX

Appendix IX

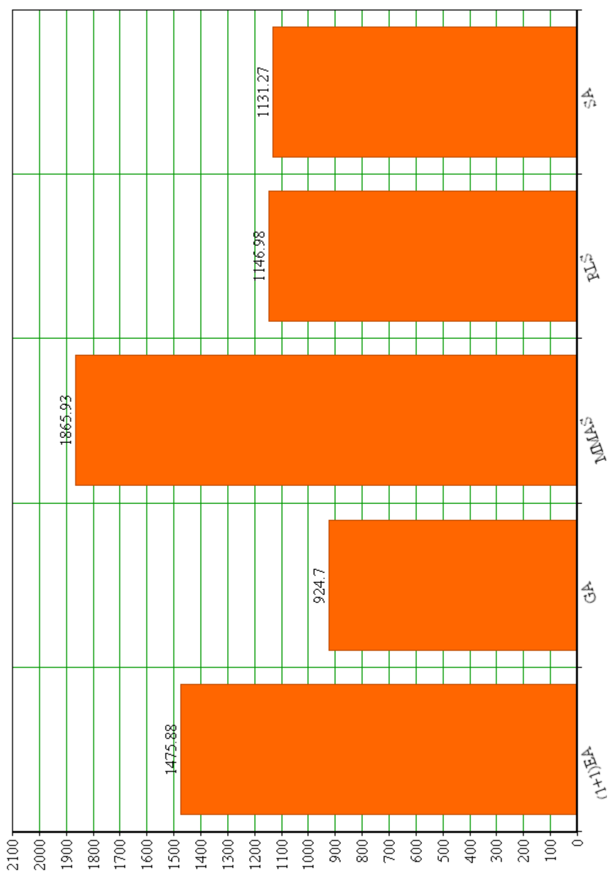

Figure IX.1: Static environment - Circle 25 - average step count

Appendix X

Appendix X

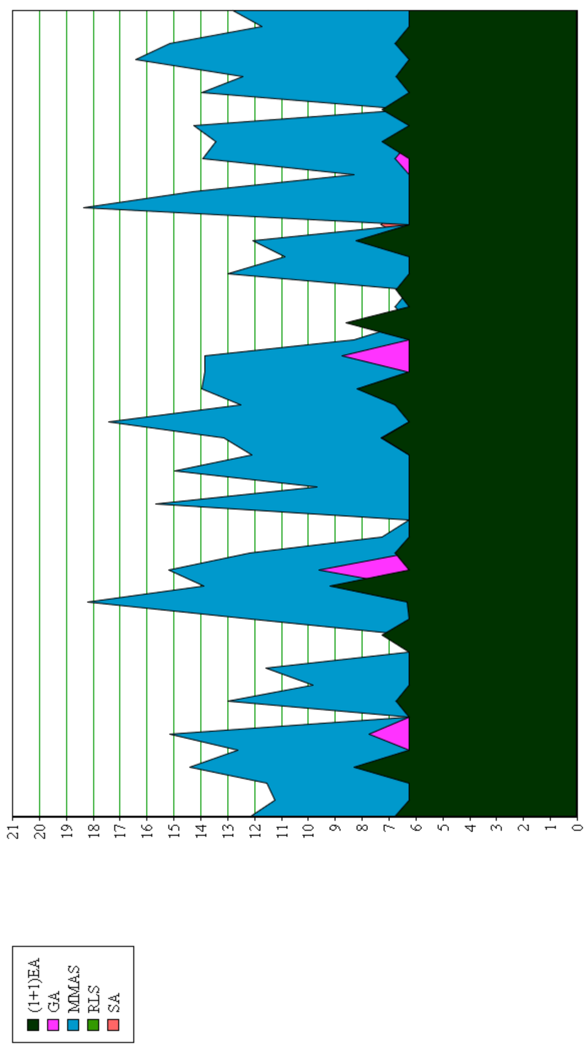

Figure X.1: Static environment - Circle 25 - tour length variation over 50 steps

Appendix XI

Appendix XI

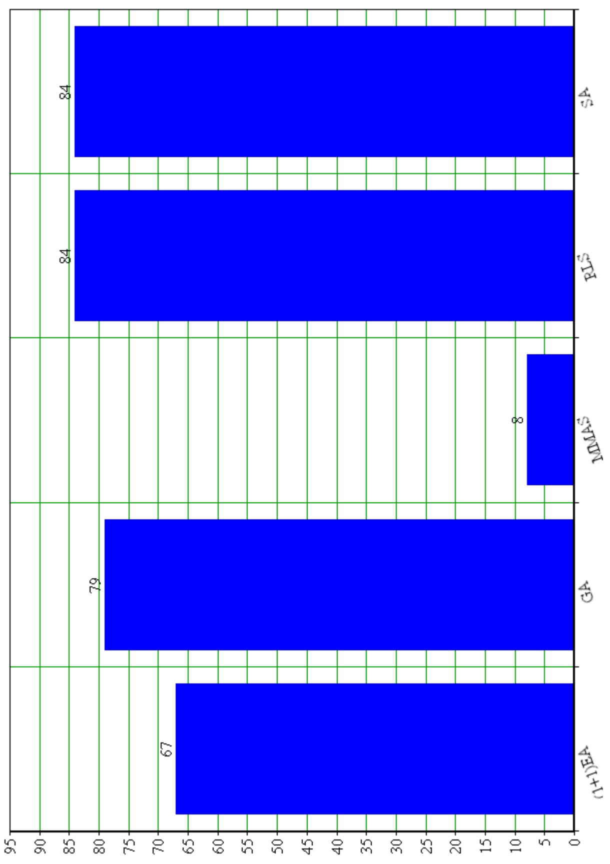

Figure XI.1: Static environment - Circle 50 - sucess rate

Appendix XII

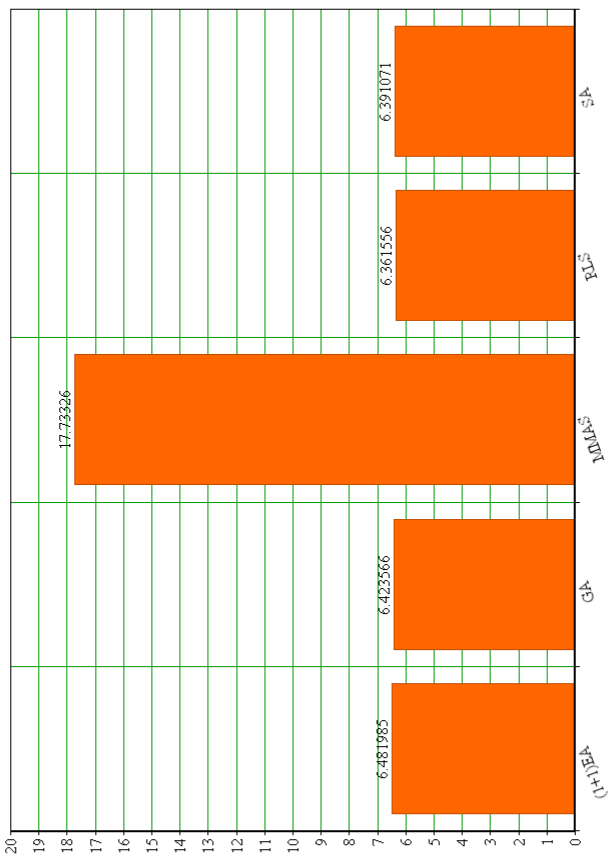

Figure XII.1: Static environment - Circle 50 - average tour length

Appendix XIII

Appendix XIII

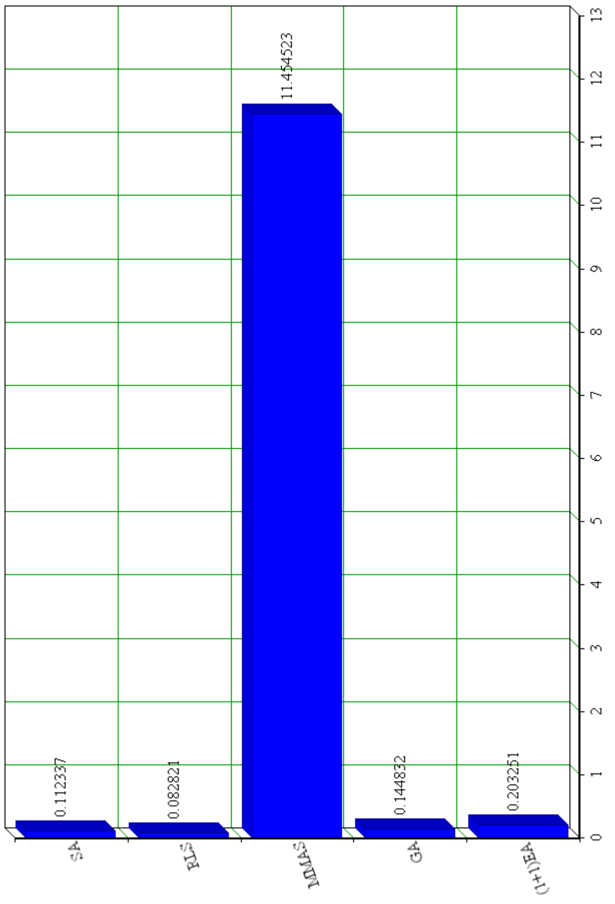

Figure XIII.1: Static environment - Circle 50 - average distance from optimum

Appendix XIV

Appendix XIV

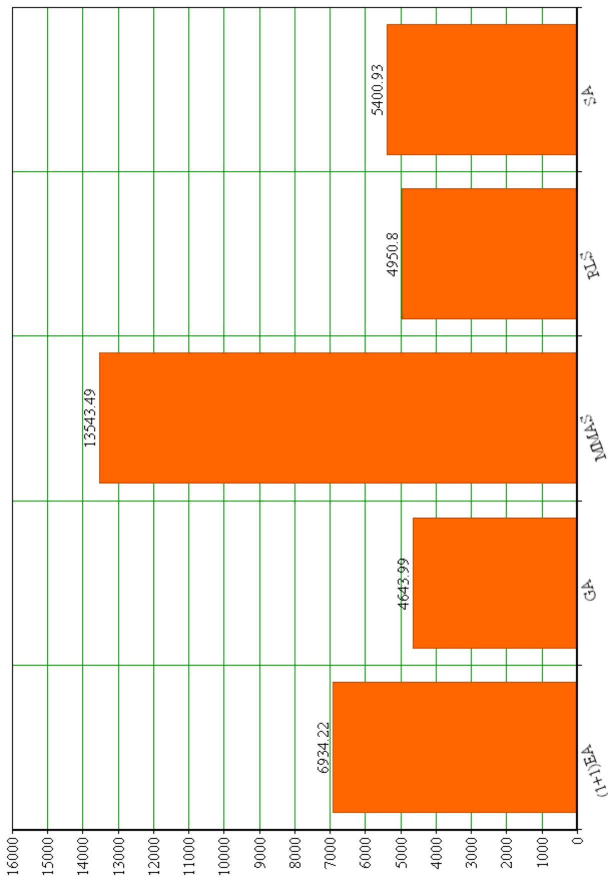

Figure XIV.1: Static environment - Circle 50 - average step count

Appendix XV

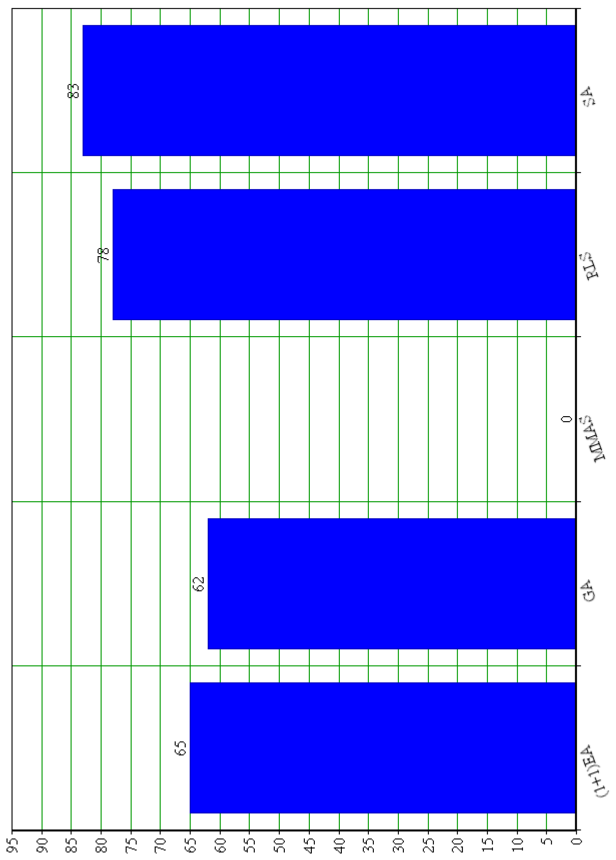

Figure XV.1: Static environment - Circle 100 - sucess rate

Appendix XVI

Appendix XVI

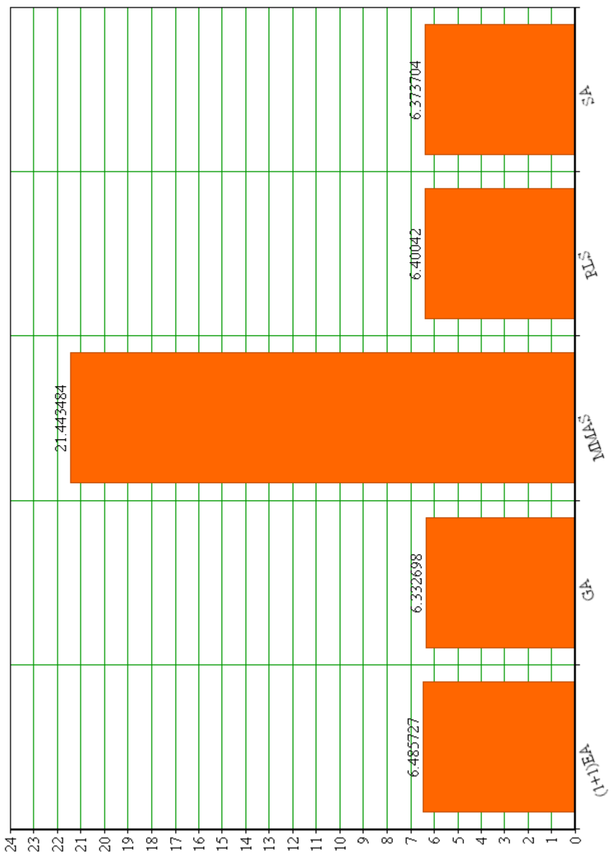

Figure XVI.1: Static environment - Circle 100 - average tour length

Appendix XVII

Appendix XVII

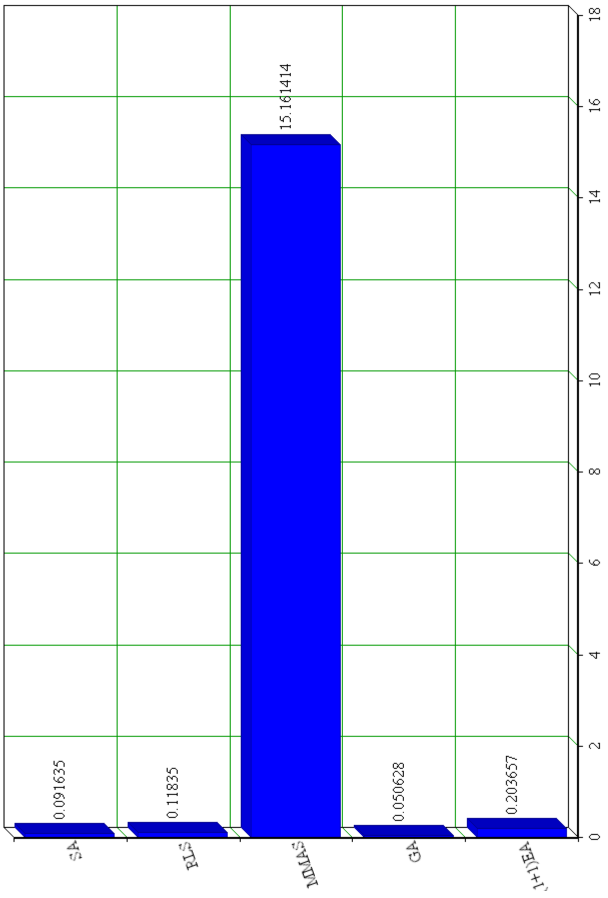

Figure XVII.1: Static environment - Circle 100 - average distance from optimum

Appendix XVIII

Appendix XVIII

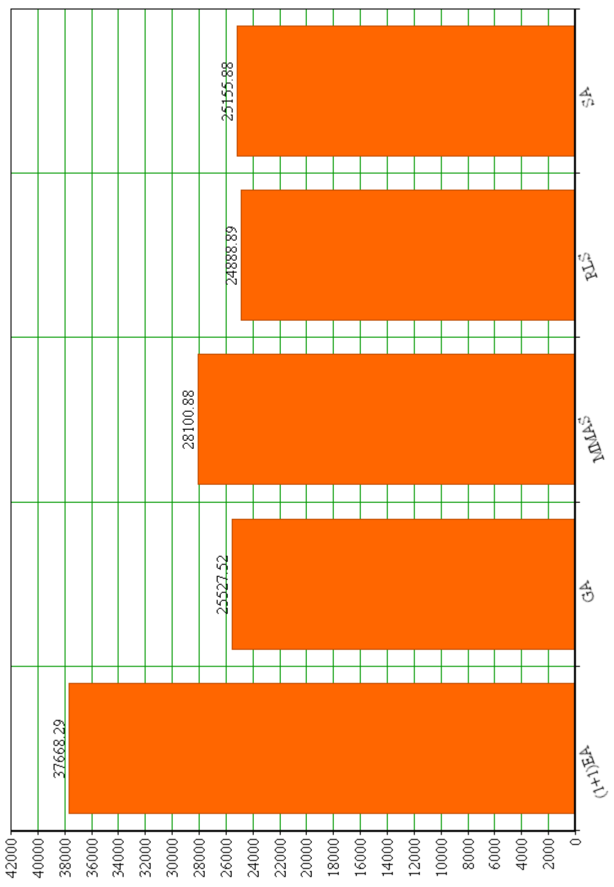

Figure XVIII.1: Static environment - Circle 100 - step count

Appendix XIX

Appendix XIX

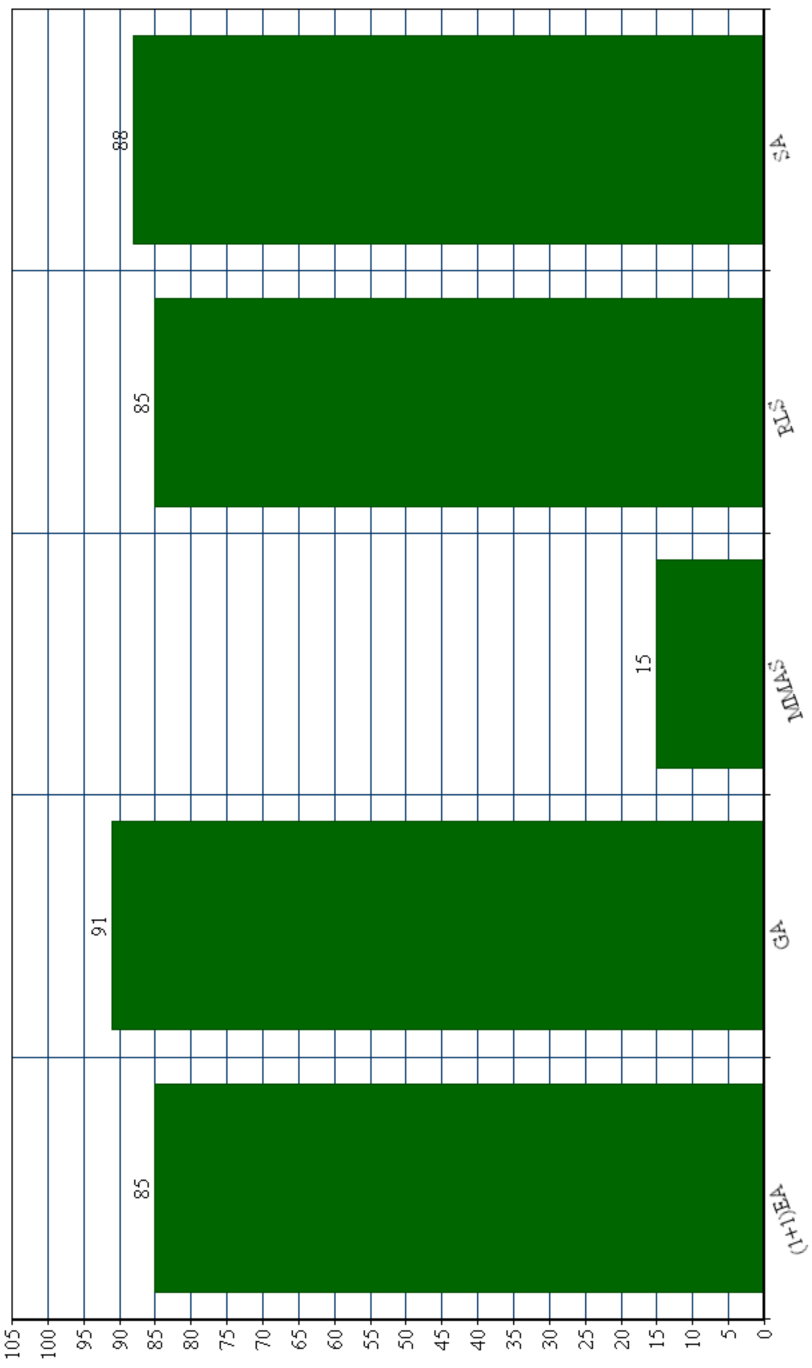

Figure XIX.1: Interchange Cities - Circle 10 - sucess rate

Appendix XX

Appendix XX

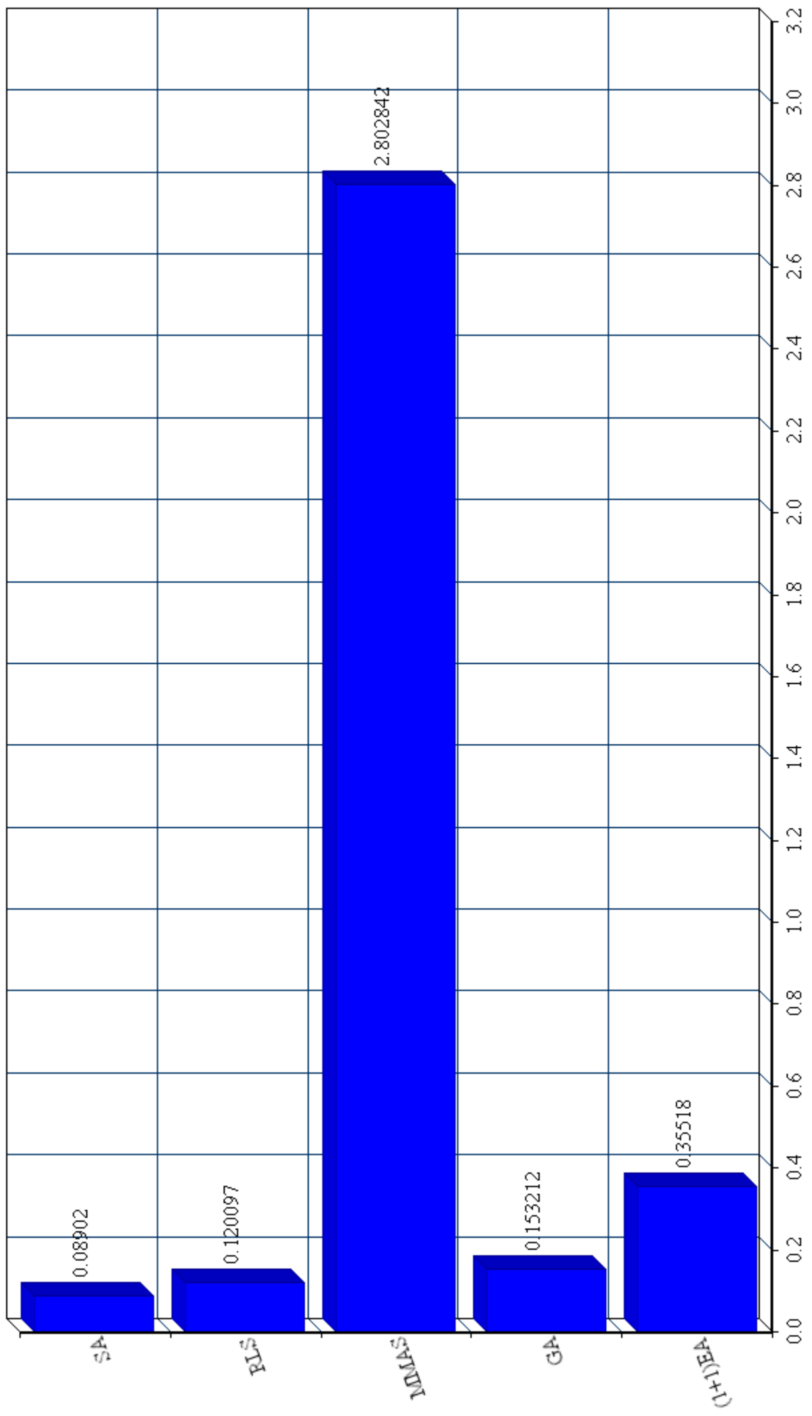

Figure XX.1: Interchange Cities - Circle 10 - average distance from optimum before change

Appendix XXI

Appendix XXI

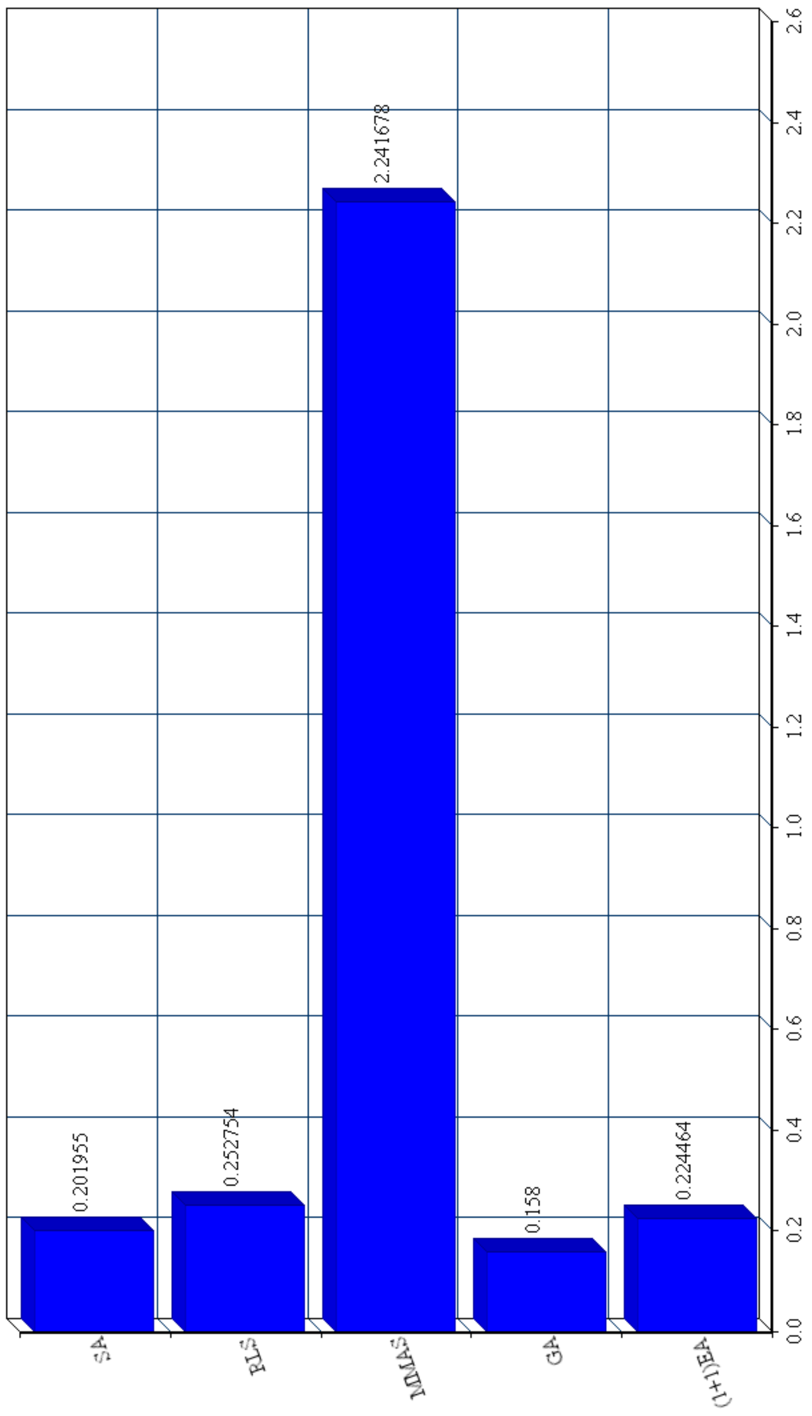

Figure XXI.1: Interchange Cities - Circle 10 - average distance from optimum after change

APPENDIX XXII

Appendix XXII

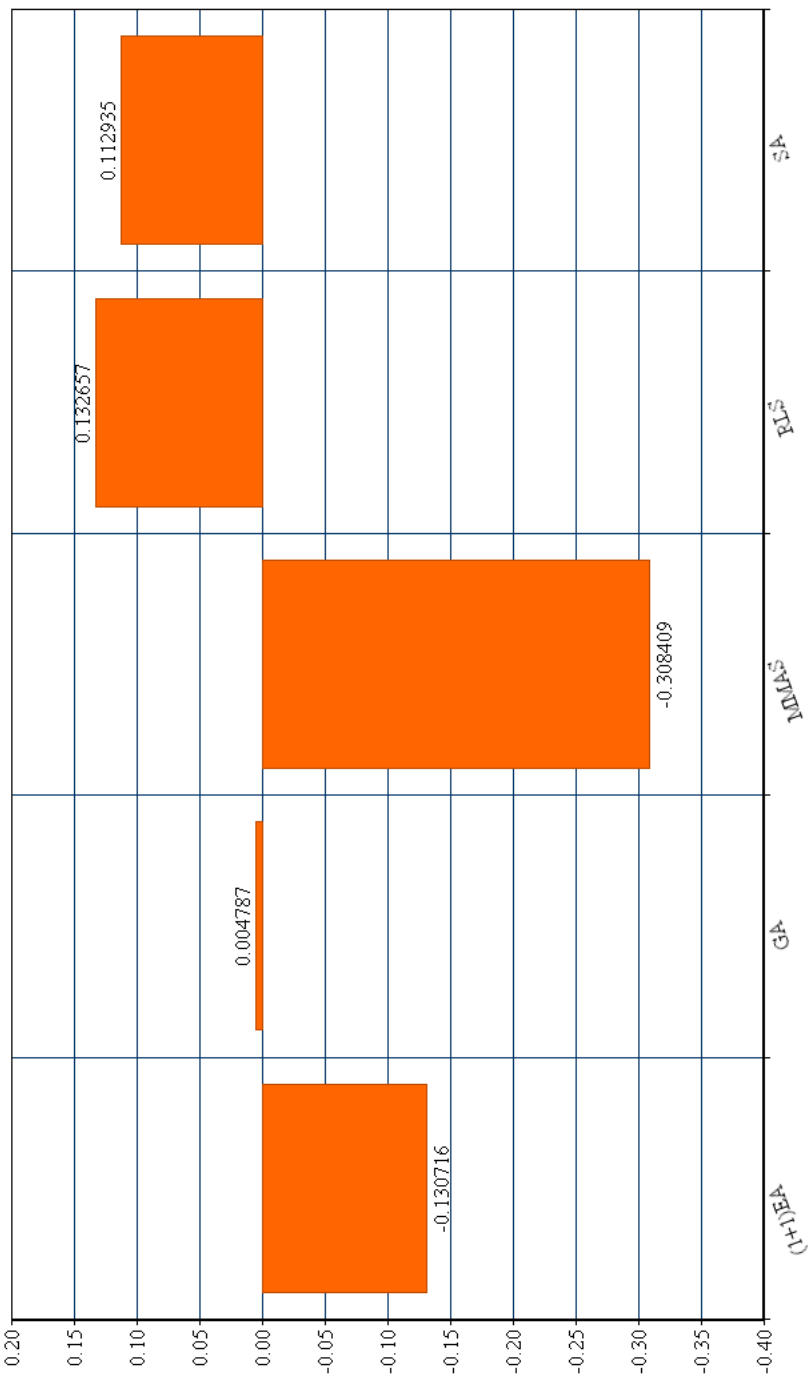

Figure XXII.1: Interchange Cities - Circle 10 - average distance between solutions before and after the change

Appendix XXIII

Appendix XXIII

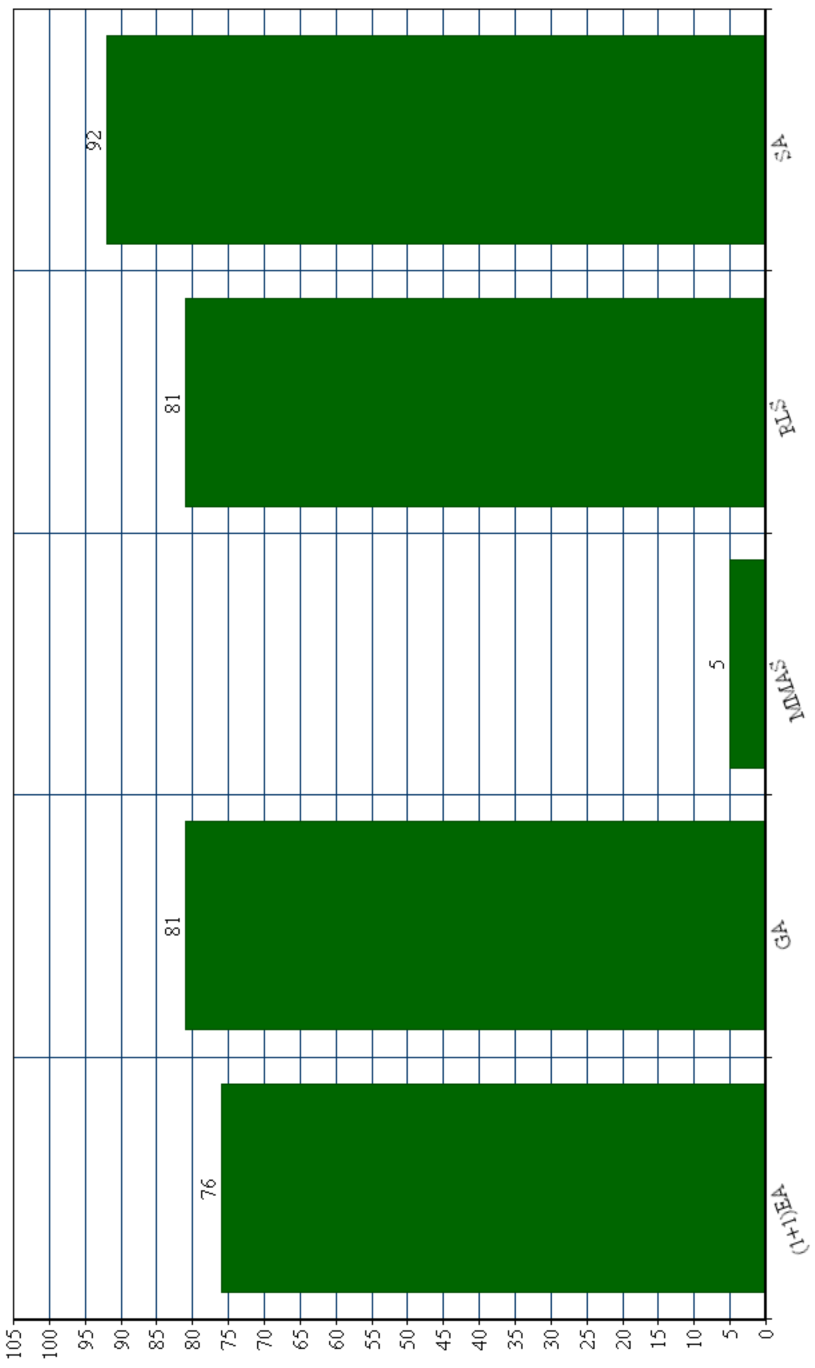

Figure XXIII.1: Interchange Cities - Circle 25 - sucess rate

Appendix XXIV

Appendix XXIV

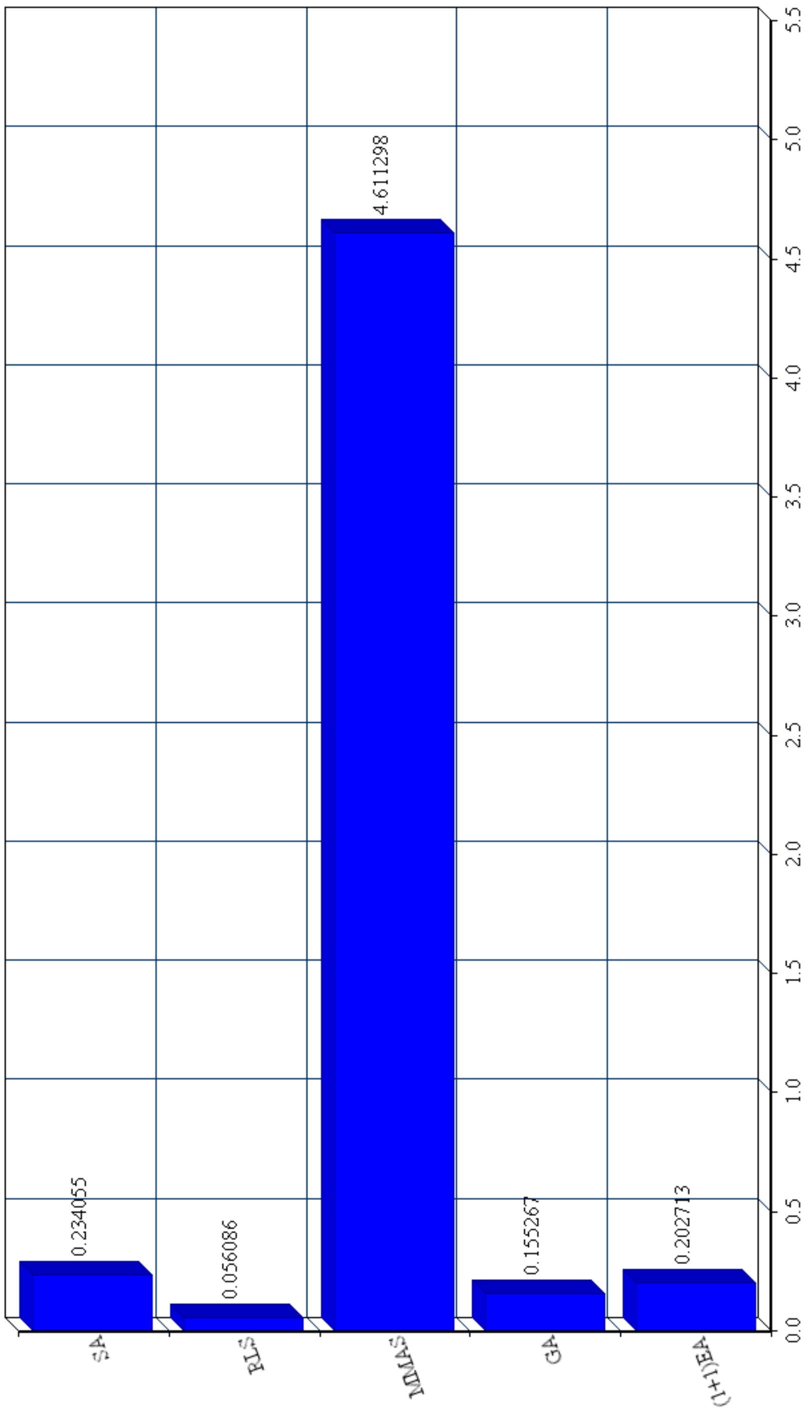

Figure XXIV.1: Interchange Cities - Circle 25 - average distance from optimum before change

Appendix XXV

Appendix XXV

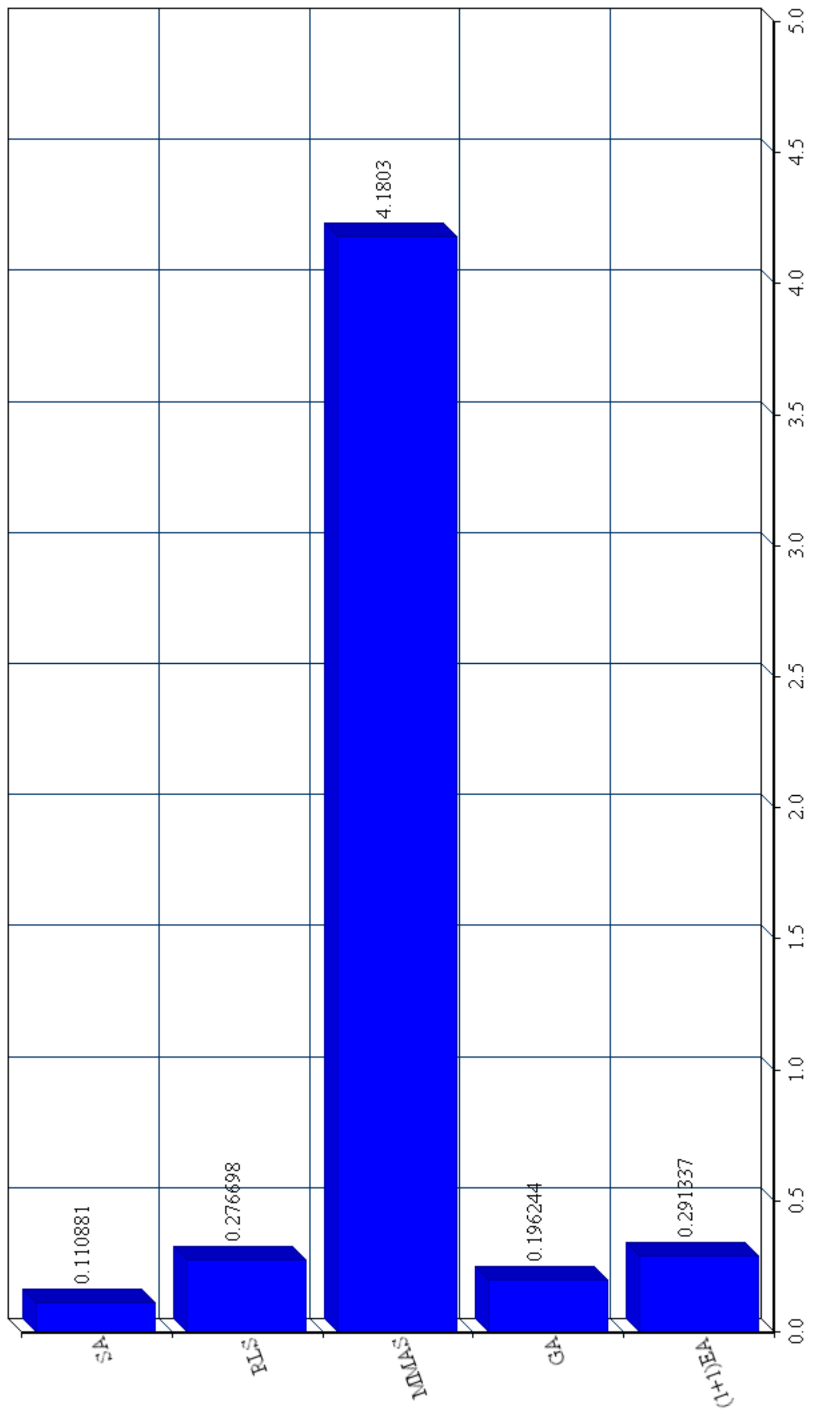

Figure XXV.1: Interchange Cities - Circle 25 - average distance from optimum after change

Appendix XXVI

Appendix XXVI

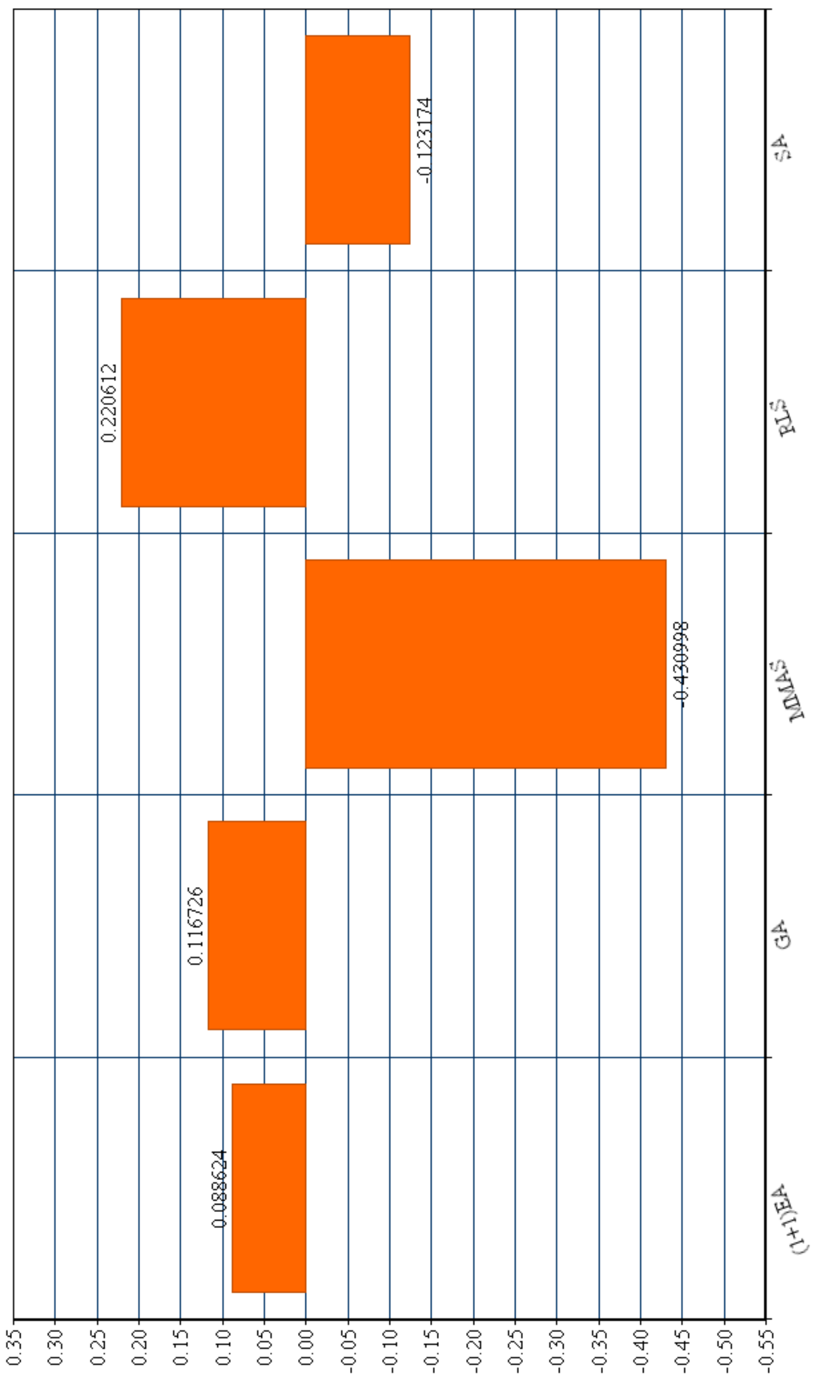

Figure XXVI.1: Interchange Cities - Circle 25 - average distance between solutions before and after the change

Appendix XXVII

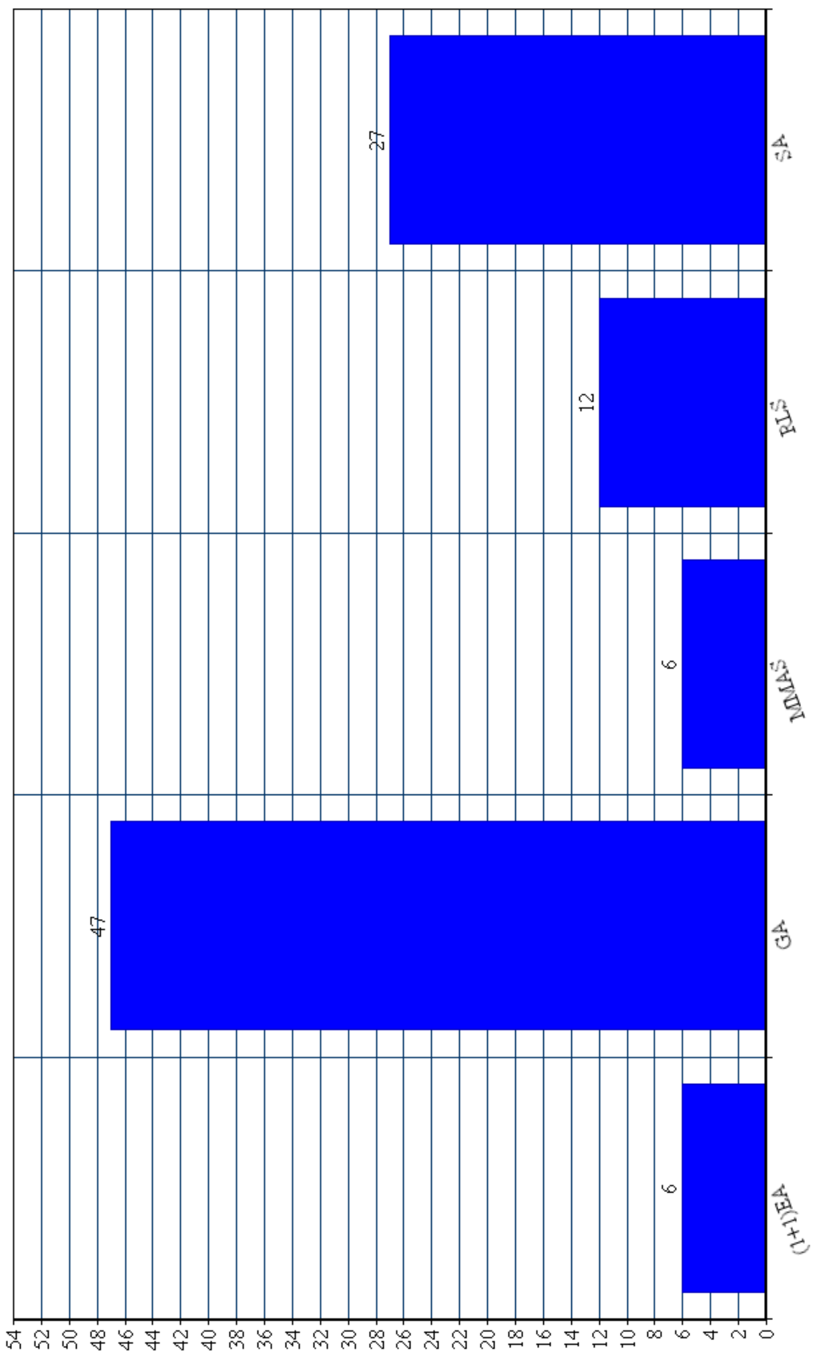

Figure XXVII.1: Congested Road - Circle 10 - number of times congested road was included in solution

Appendix XXVIII

Appendix XXVIII

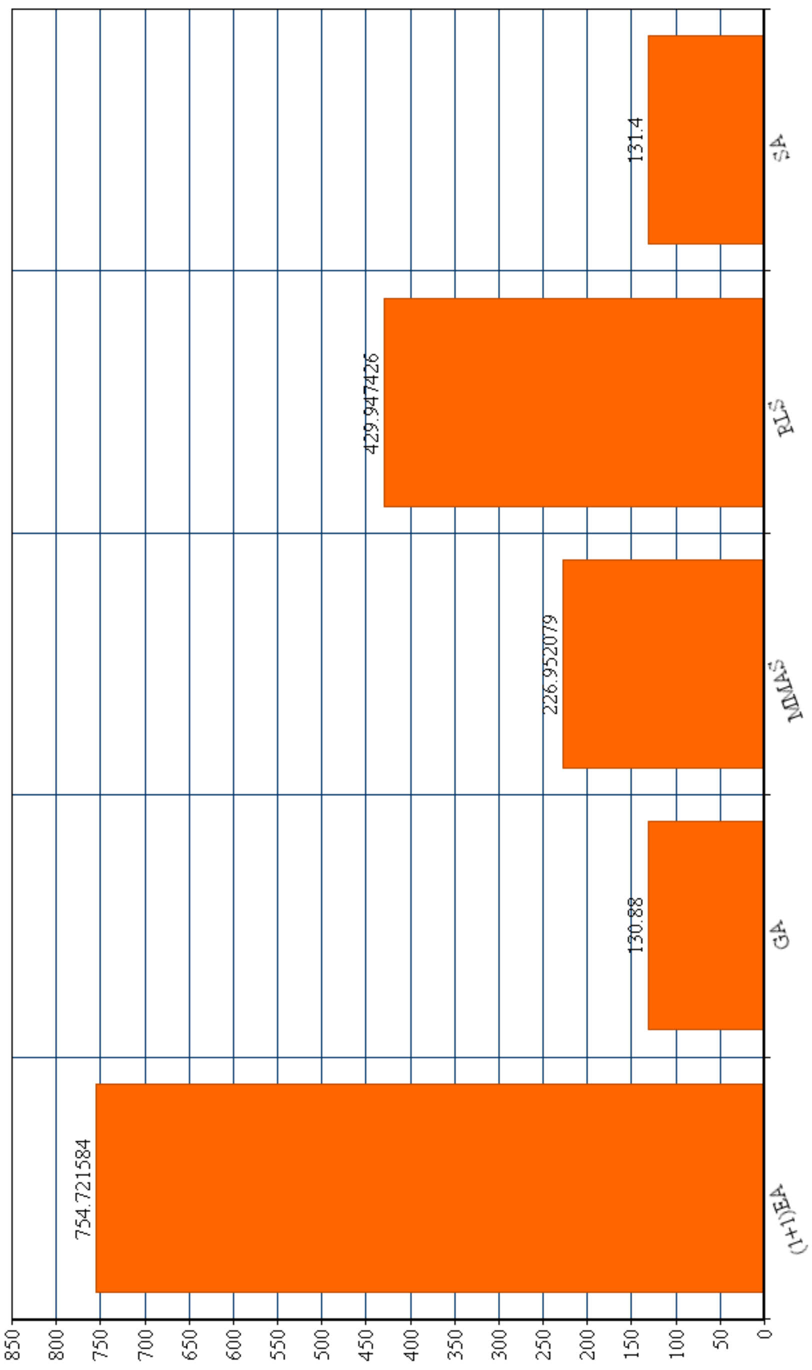

Figure XXVIII.1: Congested Road - Circle 10 - step count

Appendix XXIX

Appendix XXIX

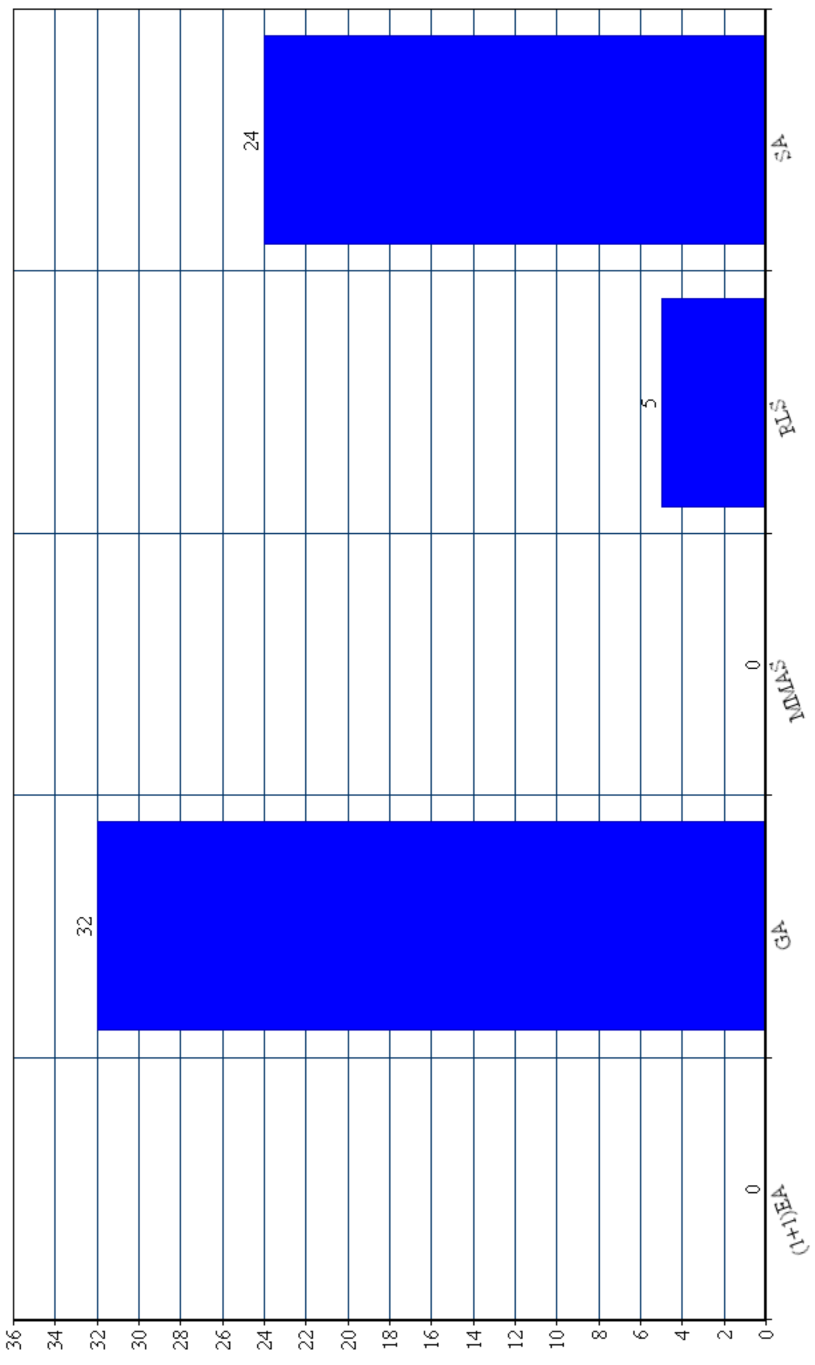

Figure XXIX.1: Congested Road - Circle 25 - number of times congested road was included in solution

Appendix XXX

Appendix XXX

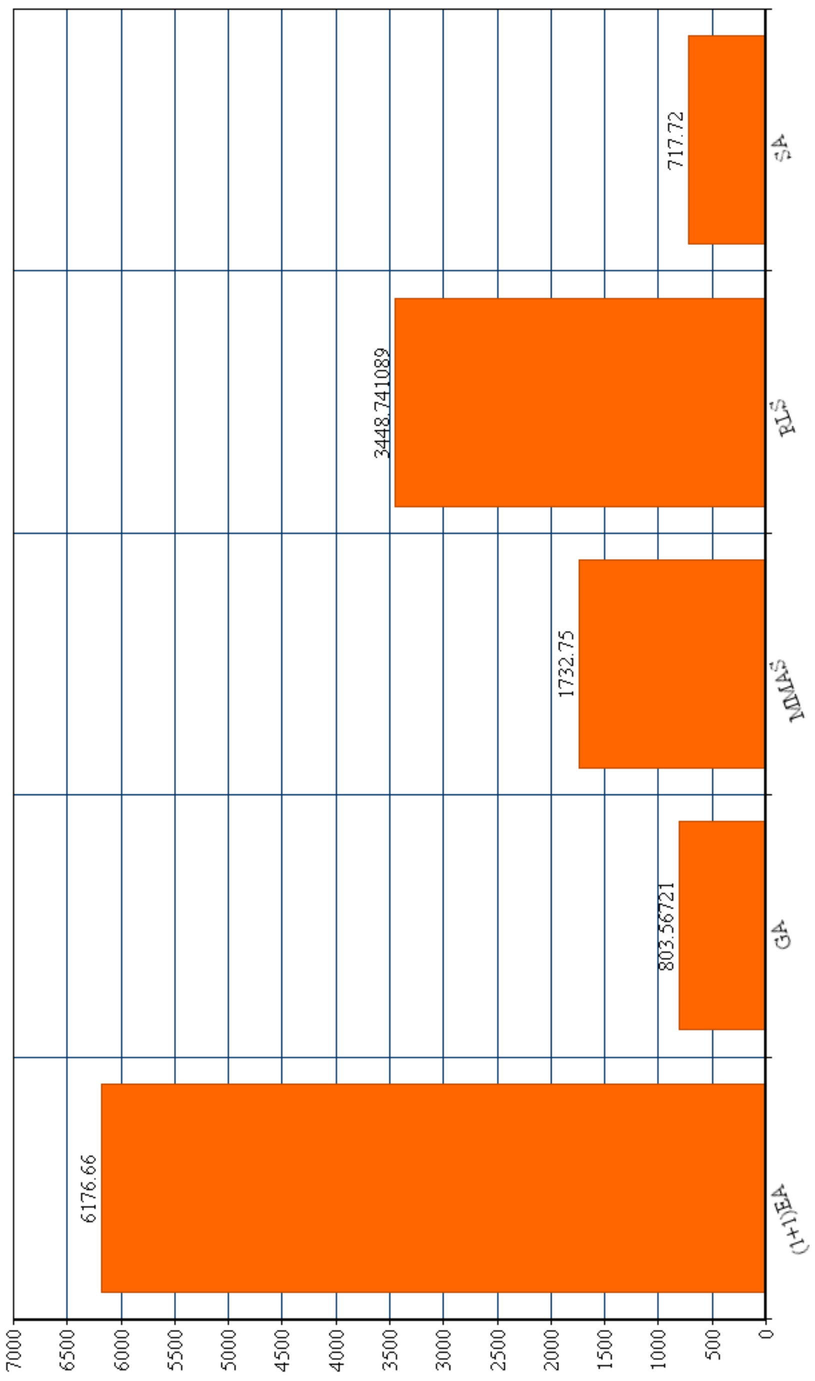

Figure XXX.1: Congested Road - Circle 25 - average step count

Appendix XXXI

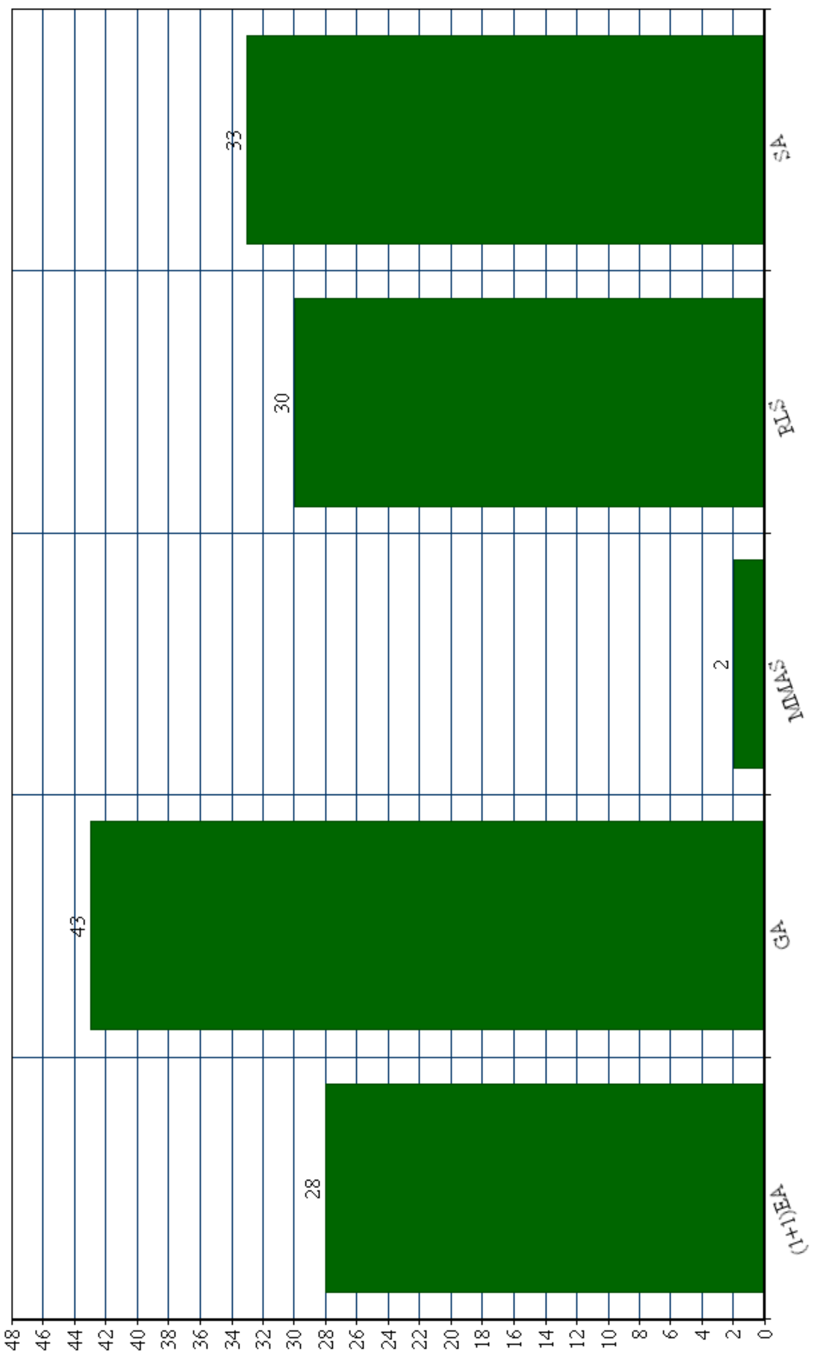

Figure XXXI.1: Delete city - success rate

Appendix XXXII

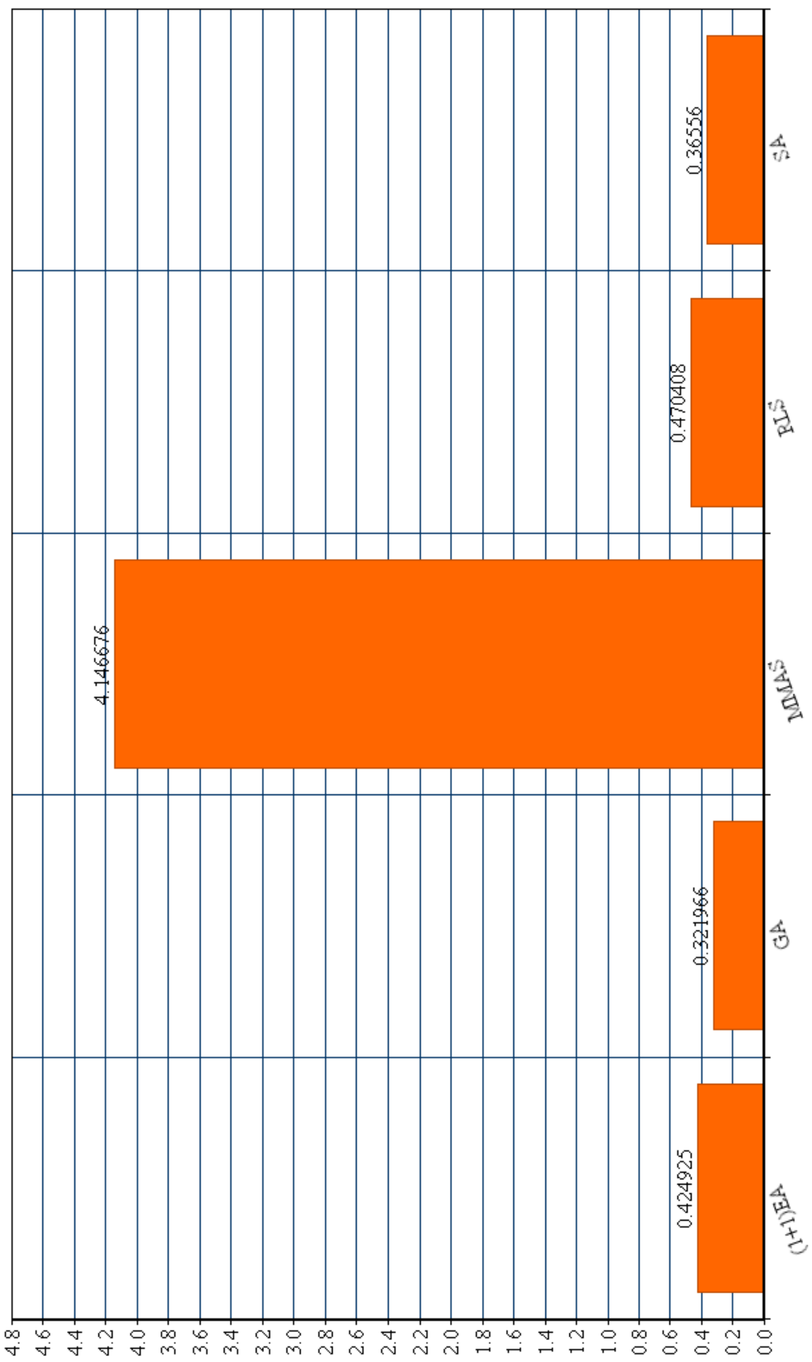

Figure XXXII.1: Delete city - average distance from optimum, after dynamic change

APPENDIX XXXIII

Appendix XXXIII

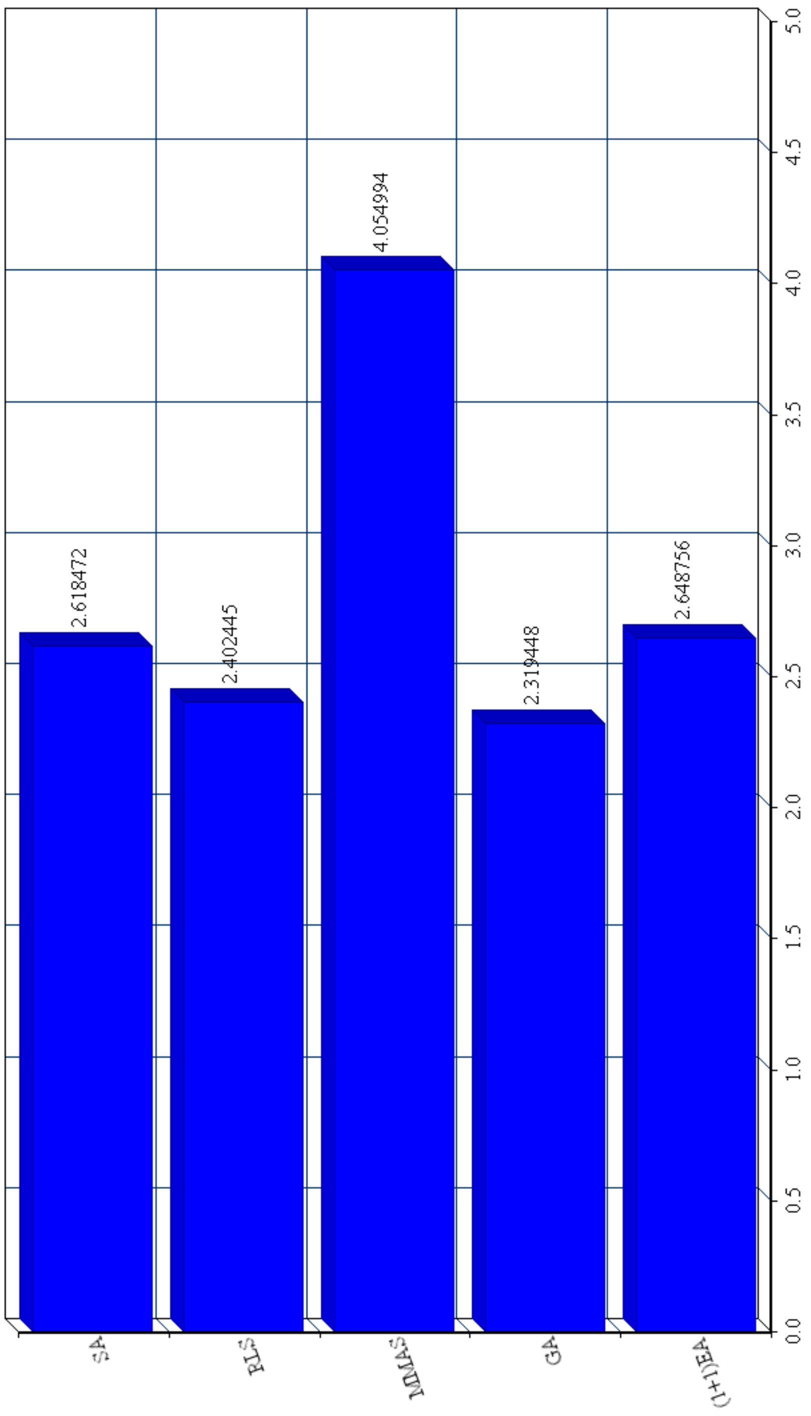

Figure XXXIII.1: Delete city - average length decrease between solutions found before and after deletion

Appendix XXXIV

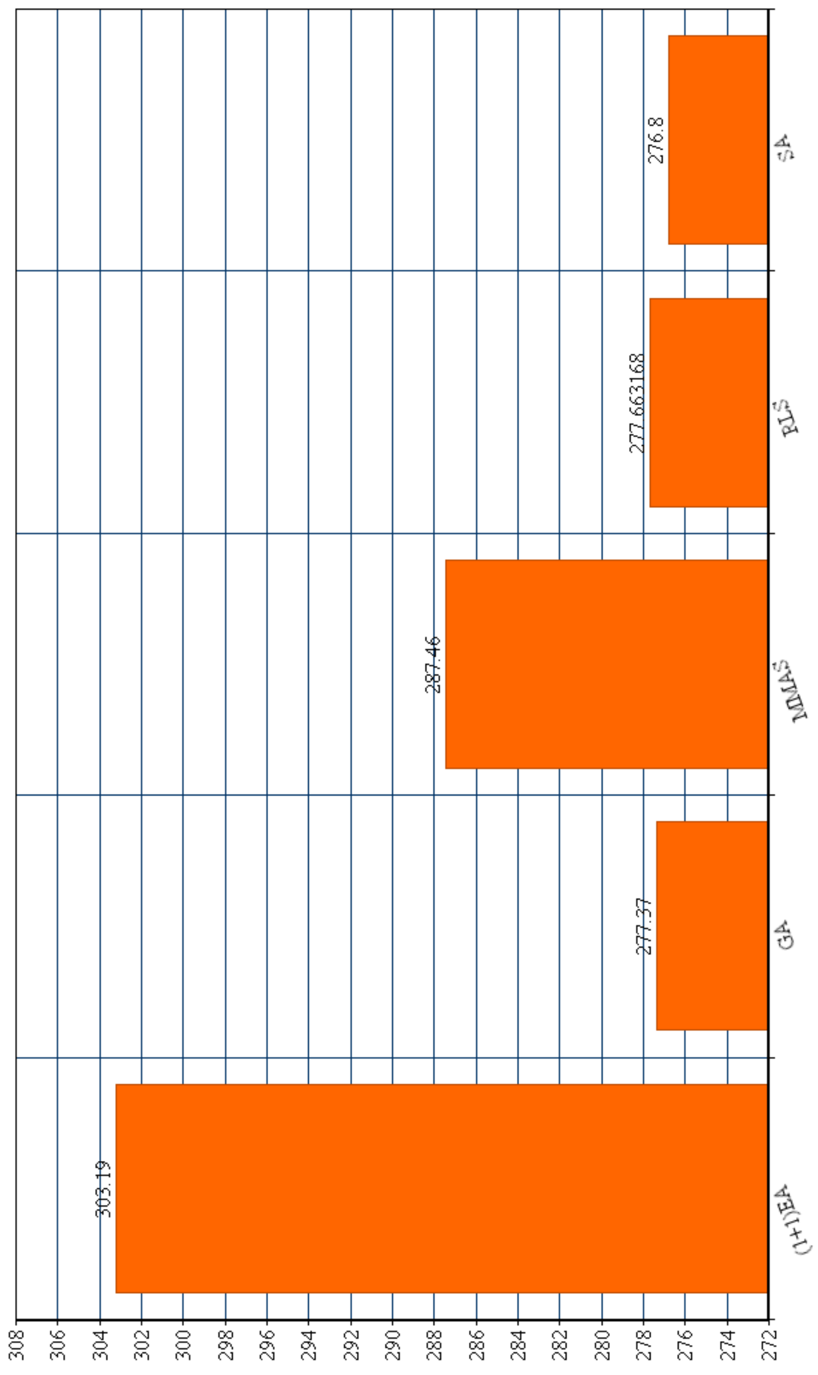

Figure XXXIV.1: Delete city - average step count

Appendix XXXV

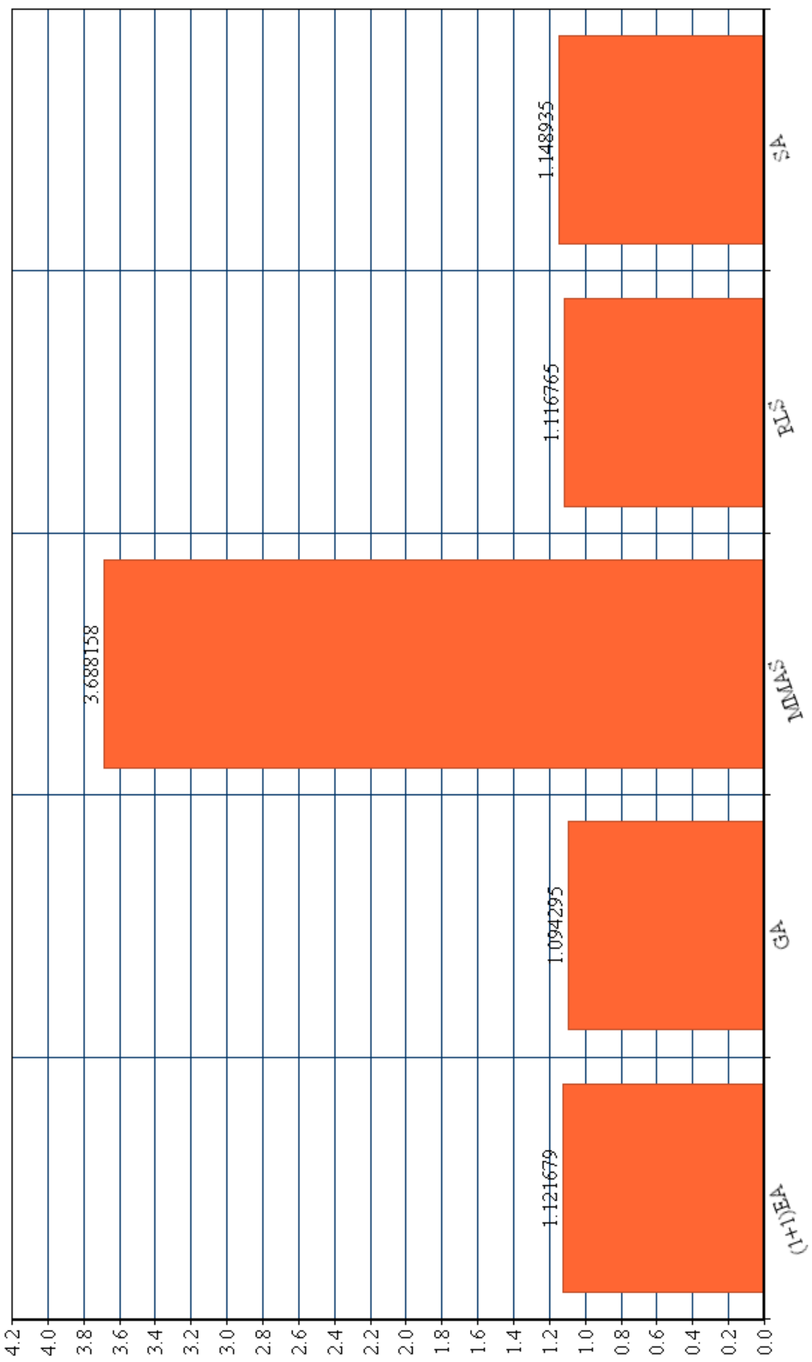

Figure XXXV.1: Add city - average distance from optimum, after dynamic change

Appendix XXXVI

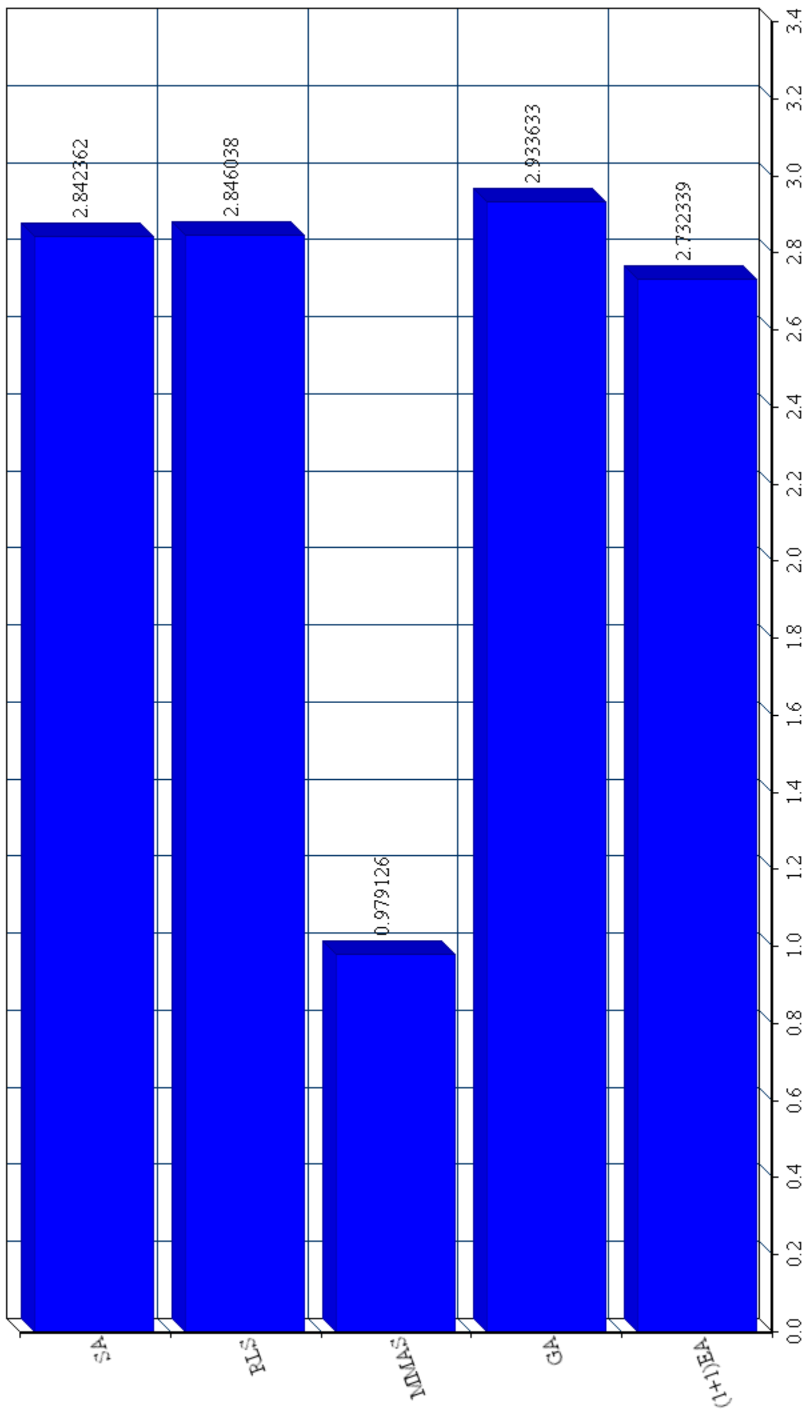

Figure XXXVI.1: Add city - average length decrease between solutions found before and after insertion

Appendix XXXVII

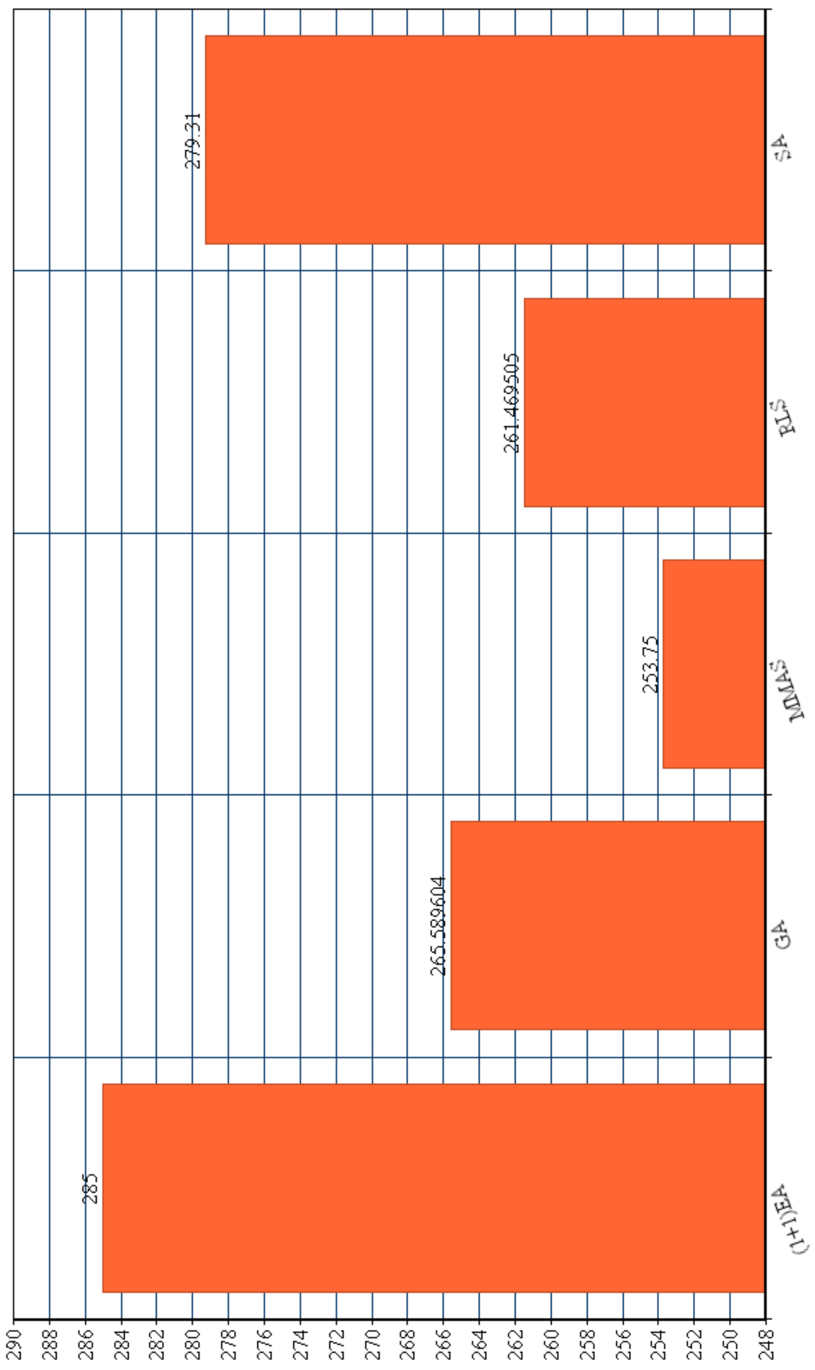

Figure XXXVII.1: Add city - average step count

[1] http://en.wikipedia.org/wiki/Travelling_salesman_problem

[2] In pursuit of the traveling salesman - William J. Cook - 2012 Princeton University Press

[3] Heuristics for the Traveling Salesman Problem - Christian Nilsson - Linkoping University - chrni794@student.liu.se

[4] http://www.umass.edu/wsp/statistics/lessons/poisson/

[5] On the Choice of the Mutation Probability for the (1+1) EA - Thomas Jansen and Ingo Wegener - FB 4, LS 2, Univ. Dortmund, 44221 Dortmund, Germany - jansen, wegener@ls2.cs.uni-dortmund.de

[6] Mathematical Runtime Analysis of ACO Algorithms: Survey on an Emerging Issue - Walter J. Gutjahr - Department of Statistics and Decision Support Systems - University of Vienna -

[7] Rigorous hitting times for binary mutations. Evolutionary Computation 7. - Garnier, J. Kallel, L., and Schoenauer, M. (1999).

[8] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

[9] An Ant Colony Optimization Approach to Dynamic TSP - Michael Guntsch, Martin Middendorf, Harmut Schmeck - Institute AIFB - University of Karlsruhe - D-76128 Karlsruhe Germany

[10] A study of permutation crossover operators on the traveling salesman problem - I. M. Oliver, D. J. Smith, and J. R. C. Holland. Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application, page 224–230. Mahwah, NJ, USA, L. E. Associates, Inc., (1987)