Universiteit Utrecht

FACULTY OF SCIENCE

Laboratory Class Scientific Computing

2nd Report

Genetic Algorithms

Handed in	15/11/2011
Teacher	Bas Fagginger Auer
Student	Benjamin F. Maier (F110163)

Contents

1 Genetic Algorithms **5**
1.1 Introduction 5
1.2 Bit Strings as Simple Genes 5

2 Finding the Minimum of an Arbitrary Function **9**
2.1 General Setting for Genetic Algorithms 9
2.2 Specification of GA Procedures 9
2.3 Tests and Numerical Results 10
2.3.1 Local Search 11
2.3.2 Mutation and Probability 11
2.3.3 Testing another function 12

3 The Traveling Salesman Problem **17**
3.1 Specification of the Problem 17
3.2 Specification of GA Procedures 17
3.2.1 Mutation and Crossover 17
3.2.2 Local Search 18
3.3 Tests and Numerical Results 18
3.3.1 Local Search Influence 19
3.3.2 Crossover and Mutation Probability 20
3.3.3 Further Tests 20

4 Summary **23**

Chapter 1

Genetic Algorithms

1.1 Introduction

The subject of artificial intelligence has become a highly important research field during the last decades. Not only is it important for science and mathematics, but also for the solution of problems which are related to everyday life, for example finding the shortest route between two points, the recognition of faces for security devices or design [1, 2]. Over the years it became clear that adapting nature in its way of solving problems is a good approach to help "machines learn" [3]. A class of algorithms which provides a natural method of solving problems is the one of "Genetic Algorithms" (GAs).

A GA is designed to find a population which fits best to its environment. It achieves that goal by adapting evolutionary mechanisms. Since this sounds rather abstract, let us define the approach more clearly. For simplicity, we can reduce a living being on its genes, which can then be simplified to a string containing information. The domain of all possible information strings is set to be D. The probability of a string $x \in D$ surviving in its environment depends on how the information connects to this environment. We can simplify this to a fitness function f, which maps a gene string x to a value in the so-called "fitness landscape" [4].

Now, the evolutionary principles can be applied. Let us consider a population of gene strings consisting of members of D. Basic principles are selection, reproduction and mutation. A selection procedure chooses genes from the population for the next generation, which means that it chooses members after their fitness and/or let them reproduce with other members of high fitness to choose their children for the next generation. Since we are not able to reproduce the complex procedures of natural selection, we will have to stick with random numbers to determine this step. Another important drive for evolution is mutation. In mutation the strings are randomly slightly modified.

Applying these procedures iteratively, producing generations of higher fitness than their ancestors, we will obtain an adaption of genes to a fitness landscape. One can apply this approach to extremal problems, which are basically problems of a domain member being adapted to a fitness function, as well.

In the following, we will explain the basic principles of genetic algorithms with the help of "bit strings" as a simple example for genes within a GA.

1.2 Bit Strings as Simple Genes

One of the easiest gene strings to imagine is a bit string. A bit string of length $l \in \mathbb{N}$ consists of a combination of l zeros and ones. Therefore the domain consists of 2^l values. An easy interpretation of bit strings can be the decimal representation of the strings, which would be the numbers $[0, 2^l - 1] \subset \mathbb{N}$. As a decimal number, there are plenty of ways to map the bit strings indirectly to another domain, e.g. to a list of domain members which is accessed by the decimal value. However, there are other interpretations, such as the route through a binary tree.

In all cases, one needs a function to connect the domain to the fitness landscape. We call this the "objective value function" f.

Mathematical Problem Formulation

With these bit strings representing possible members of a domain D and the objective value function f, we want to solve the problem $\max_{x \in D} f(x)$ or $\min_{x \in D} f(x) = \max_{x \in D}(-f(x))$. We will do this with a population of n members, always living in the current generation $\mathcal{G}$.

Evaluating the Fitness in a Generation

The objective value function f yields a fitness value which makes it possible to compare members within the whole domain. However, it is useful to judge members only according to their fitness within the current generation. Therefore we will map the objective values of the i-th member of a generation to their generation fitness $\mathcal{F}_i$. For this procedure we have to map into a fitness interval $[m_-, m_+] \subset \mathbb{R}$, where we have to know the generation's maximal objective value f_+, as well as the minimal objective value f_-. With this information, the current objective value of a member $f(x_i)$ is mapped to its generation fitness according to

$$\mathcal{F}_i = m_- + (m_+ - m_-)\,\frac{f(x_i) - f_-}{f_+ - f_-}, \tag{1.1}$$

where we will work with $m_- = 1$ and $m_+ = 90$ throughout the whole paper.

Selection and Reproduction

The selection probability of a population's member should be proportional to its generation fitness. To achieve that, we start with adding up all current fitnesses to the total fitness $\mathcal{F}$. Afterwards we draw a random number S in $[0, \mathcal{F}]$ and start adding all fitnesses. Once we end up exceeding S with the sum, we count the fitnesses we added up and take that count as the number of the selected member. Since this procedure has parallels to its representation in the carnival, it is called "rolling wheel selection".

Then, with a probability of p_c, the member is chosen to be a parent for reproduction, which is also called "crossover". Once we have two chosen parents, we perform the crossover as described in the following. Given two parents p_1 and p_2, select a bit position. Then swap the bits of p_1 which lay right to that position with the bits of p_2 which lay right to that position. Save the new created bit strings as new members of the next generation.

This procedure is expected to do the optimization. As in nature, we expect two "fit" genes to produce children with a high fitness, as well. Since the swap of bits is restricted to the "right" part of the bit string, a crossover changes the decimal representation of a bit string rather locally. The more "right" the bits lay, the less is their effect on their decimal representation.

Mutation

After selecting and reproducing members for the next generation, the bit strings will mutate. The mutation procedure goes through every bit of a bit string and draws a random number $R \in [0, 1)$. If R is smaller than a set mutation probability p_m, the bit will flip (from zero to one or vice versa). Mutation is said to be the drive of diversity within a population and also responsible for a convergence of the procedure [5]. Furthermore, mutation can lead a population out of a local extremum to another extremum [4].

Convergence

Speaking of convergence - there are different criteria on how to decide if the current population will not drastically change anymore within the next generations. One is called the "hamming distance" for which one computes the difference between the generation's chromosomes. As soon as the distance is lower than a certain value, the algorithm is said to be converged. Another one is the behavior of the generation's fittest member. One may define a convergence threshold. As soon as a the fittest member has stayed the same for the convergence threshold number of generations, the algorithm is declared to be converged. In this paper, we will stick with this definition.

Elitism

Since there is the possibility of temporarily fit members being destroyed by mutation, we can think of a way to protect them. Once you have members with large fitness, you can save them as elite members. Elite members will only mutate if their mutation leads to a higher objective value than their former one. Furthermore, elite members are always chosen to be parents in the selection procedure.

Gray Codes

A flip of a bit during mutation may lead to a drastical change within the decimal representation of the bit string, because depending on where the bit is located, it may be responsible for a large part of its decimal value. Thus, one can apply the use of gray codes. With gray codes, one changes the bit representation to another decimal value, s.t. a change in one single bit does not yield a large change in the decimal value [4, 6]

Local Search

A very sophisticated method to improve the convergence of the genetic algorithm is to search locally for an improvement. That means, take member x_i, and search for members in the neighborhood of x_i which have a better objective value. However, this search highly depends on the problem – for the definition of "neighborhood" as well as for the search itself. Usually, the local search is only applied on elite members to keep the order of the algorithm fixed.

Summary

With these procedures we can set up a genetic algorithm as the following. Randomly select n members of the domain D (if they are bit strings, choose the gray code approach). Evaluate their objective values, their fitnesses and set n_e of them to be elite, according to their fitnesses. Select the members of the next generation and perform crossovers (use the elite members to be parents). Afterwards, mutate the new generation. Then perform a local search within the elite members. In the end, evaluate the objective values, the fitnesses and the elite members again. Repeat as long as the member with the highest objective value does not change within the generation number convergence threshold.

Chapter 2

Finding the Minimum of an Arbitrary Function

2.1 General Setting for Genetic Algorithms

An easy problem to investigate the behavior of GAs is to find a global minimum of a function f on a domain $D \subset \mathbb{R}$, since the objective value function is given by f itself. Because D is one-dimensional in this case, we can easily use bit strings as members of a population and map their decimal representation to the interval $[a,b]$, where $a = \min D$ and $b = \max D$. Let d be the decimal representation of a bit string of length l. Its value in D is then

$$x = a + (b-a)\,\frac{d}{2^l - 1}. \tag{2.1}$$

We cannot map to all values in D, therefore it is useful to decide for a relative precision p of the values and adjust the bit length l to achieve a satisfying result.
Using a bit string of length l, the difference between two neighbors is

$$\Delta = \frac{b-a}{2^l - 2}. \tag{2.2}$$

To achieve a relative precision of p, one can consider a "general" relative distance between the neighbors $\Delta/(b-a)$ and set it equal to p, yielding

$$p = \frac{1}{2^l - 2} \tag{2.3}$$

$$\Rightarrow l = \left\lceil \frac{\ln(p^{-1}+2)}{\ln 2} \right\rceil, \tag{2.4}$$

where the ceil function $\lceil\,\rceil$ was used to assure the demanded precision.
To assign a value $x \in \mathbb{R}$ to a member of our population, we first have to check if it is a member of D. If not, the assignment is $x = a$ for $x < a$, respectively $x = b$ for $x > b$. Then, we can map x to a decimal value in $[0, 2^l - 1]$ according to

$$d = \text{round}\left[\frac{x-a}{b-a}\left(2^l - 1\right)\right] \tag{2.5}$$

and compute the corresponding bit string.

2.2 Specification of GA Procedures

Because of the use of bit strings representing the domain, we can take advantage of the procedures of selection, mutation and crossover introduced in the last chapter. Furthermore we will use elitism and gray codes, which should assure a faster convergence respectively a mutation in the neighborhood of a bit string. However, we have to think of a sophisticated method to perform a local search. As proposed in [4], there are two well-known solutions for a local optimization of a member x of the population.
The first is to search iteratively in a neighborhood of x for a new value $\tilde{x}$ with $f(x) < f(\tilde{x})$. One has to think of how to limit the members to be investigated in the neighborhood to guarantee a fast search.

The second is called “the method of steepest descent” and needs the existence of the first derivative of f on D, called f'. Starting from x, one computes $f'(x)$. If $f'(x) > 0$, one decrements x by a chosen δ. If $f'(x) < 0$, one increments x by the chosen δ. Now one repeats this with the new x for a certain amount of iterations. To obtain a fast local search, we would have to determine an optimal δ for the problem and the optimal number of iterations.

Since these two methods are of constant order to search for a local minimum, we propose a new method. The method of steepest descent equals a search for zeros of f', which can easily be done with Newton’s method. This method uses the second derivative f'', which has to exist on D without singularities. Given a member $x_0 = x$ of D, one can find an extremum by computing

$$x_{j+1} = x_j - \frac{f'(x_j)}{f''(x_j)}, \qquad j \in \mathbb{N} \cup \{0\} \tag{2.6}$$

iteratively, until a certain relative precision r is reached by the relative deviation $\|x_{n+1} - x_n\|/(b-a)$ of two iterations. This precision r should not be smaller than the precision p, which is limited by the bit string length. Furthermore it should not be smaller than 10^{-12} to guarantee a numerically stable evaluation and prevent errors from the machine epsilon and be limited by a maximum number $j_{\max}$ of iterations.

However, Newton’s method does not guarantee to find an extremum which is a minimum, but may also find a maximum. It is therefore appropriate to perform a hybrid local search consisting of the method of steepest descent and Newton’s method.

First, one checks the second derivative at x. If $f''(x) > 0$, perform Newton’s method, since it is more likely to converge to a local minimum of f. If $f''(x) < 0$, Newton’s method would rather converge to a local maximum, therefore we apply the method of steepest descent according to the value of the first derivative until the second derivative is greater than zero or a certain amount of iterations is reached. If $f''(x) = 0$, check $x \pm \delta$ and decide for the side with $f''(x) > 0$ to perform Newton’s method starting from there. This method is shown in Alg. 1.

Algorithm 1 Local improvement of a population member $x \in D$

```
Require: Let $f$ be the function to be minimized, $f'$ its derivative and $f''$ its second derivative
Require: Let $p$ be the smallest relative precision given by the bit string length
  $k \leftarrow 0$
  $s = -\mathrm{sign}(f'(x))$
  while $(f''(x) \le 0) \wedge (k < k_{\max}) \wedge (a < x < b)$ do
    $x \leftarrow x + s\delta$
    $k \leftarrow k + 1$
  end while
  if $f''(x) > 0$ then
    $j \leftarrow 0$
    $x_0 \leftarrow x$
    repeat
      $j \leftarrow j + 1$
      $x_j \leftarrow x_{j-1} - \frac{f'(x_{j-1})}{f''(x_{j-1})}$
    until $\left(\frac{\|x_j - x_{j-1}\|}{b-a} < p\right) \vee \left(\frac{\|x_j - x_{j-1}\|}{b-a} < 10^{-12}\right) \vee (j > j_{\max}) \vee (x_j \le a) \vee (x_j \ge b)$
    $x \leftarrow x_j$
  end if
```

Of course, this algorithm can only be used for functions with existing first and second derivative. For functions where only the first derivative is known, one can use the steepest descent method, while for functions with unknown derivative, one would have to use the naive iterative local search comparing the values of f.

2.3 Tests and Numerical Results

The method described above was implemented and tested. One population consisted of 11 bit strings, the number of elite members was chosen to be 2, the demanded precision was set to $p = 10^{-6}$. A sequence of generations was chosen to be converged when the maximum objective value did not change in 200 generations within the demanded precision. The function to minimize was chosen to be

$$f(x) = x^2 + 200\sin(x), \tag{2.7}$$

on the interval $D = [-20, 30] \subset \mathbb{R}$. Its global minimum is found at $x \approx -1.55524$.

2.3.1 Local Search

The first interest of investigation is the test of the different local searches. We set the population size to 11 and the number of elite members to two. Mutation probability was set to 0.45, crossover probability was set to 0.3. The parameter δ of the steepest descent was set to 0.5% of the interval length. The algorithm was said to be converged when the population's fittest member did not change in 200 iterations. For every local search, we run the algorithm 2000 times and sorted the generation number n^G where the probable optimum was found in a histogram of $N_k = 15$ classes with corresponding means n_i^G and frequencies f_i. Mean and standard deviation were computed according to

$$\bar{n}^G = \frac{1}{N}\sqrt{\sum_{i=1}^{N_k} n_i^G f_i} \tag{2.8}$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N_k}\left(\bar{n}^G - n_i^G\right)^2 f_i}{N-1}}. \tag{2.9}$$

Furthermore we computed the failure percentage, which represents the percentage of results which did not equal the real optimum, to make a fair comparison possible.

The result can be seen in Fig. 2.1. As one can see, Newton's method converges drastically faster than the method of the steepest descent. However, there is not really a difference between the hybrid Newton and the conservative Newton. But as described in the previous section, we would recommend the hybrid method, since for a worse setting of mutation/crossover probabilities, the hybrid method is expected to be more effective.

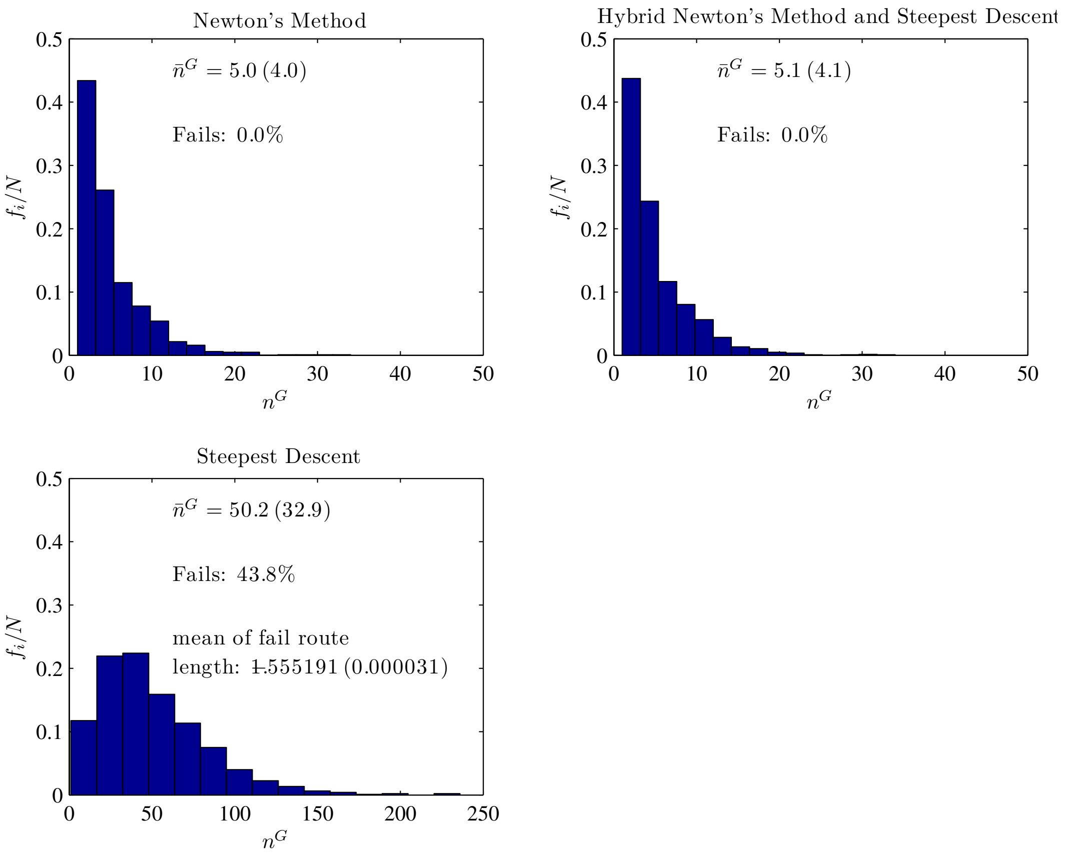

Figure 2.1: Convergence behavior of the algorithm with the three different local optimization methods. The x-axis shows the number of the generation in which the probable optimum was found. The y-axis shows the relative frequency. Furthermore one can see the generation mean and in brackets the standard deviation, computed with Eq. (2.8-2.9)

2.3.2 Mutation and Probability

The second aspect to be investigated is the behavior of the algorithm in dependence of the probabilities for mutation and crossover. We chose 10 different probabilities for each mutation and crossover and performed $N = 200$ runs for every combination. For every combination, the threshold 200 was subtracted from the numbers of generations n^G to obtain the generation numbers which firstly included the probable optimum. The results have been classified in a histogram with $N_k = 20$ classes, computing mean and standard deviation afterwards, as well as the computation of a failure percentage for a better comparison.

The results are shown in Fig. 2.2-2.3 respectively in Tab. 2.1-2.2. As one can see in the figures, the convergence behavior is highly connected to the mutation probability. For small crossover probabilities, one can see that the main drive for the evolution is the mutation, as it provides the diversity in the population [5]. For small mutation probabilities, the chance of escaping from local minima is rather low, therefore the failure percentages and the convergence generation n^G increase. As the mutation probability grows, the failure percentages decrease and the steps to reach convergence decrease, as well. However, with growing crossover probability, the effects of both, crossover and mutation, seem to lead to an increasing convergence step number as well as higher failure percentages. A possible explanation would be that improvements made by selection are destroyed by crossover and too aggressive mutation, as the elitism only prevents members from mutation which were elite before. The best area of probability combination seems to be in the neighborhood of $p = 0.3$ for mutation and $p = 0.6$ for crossover.

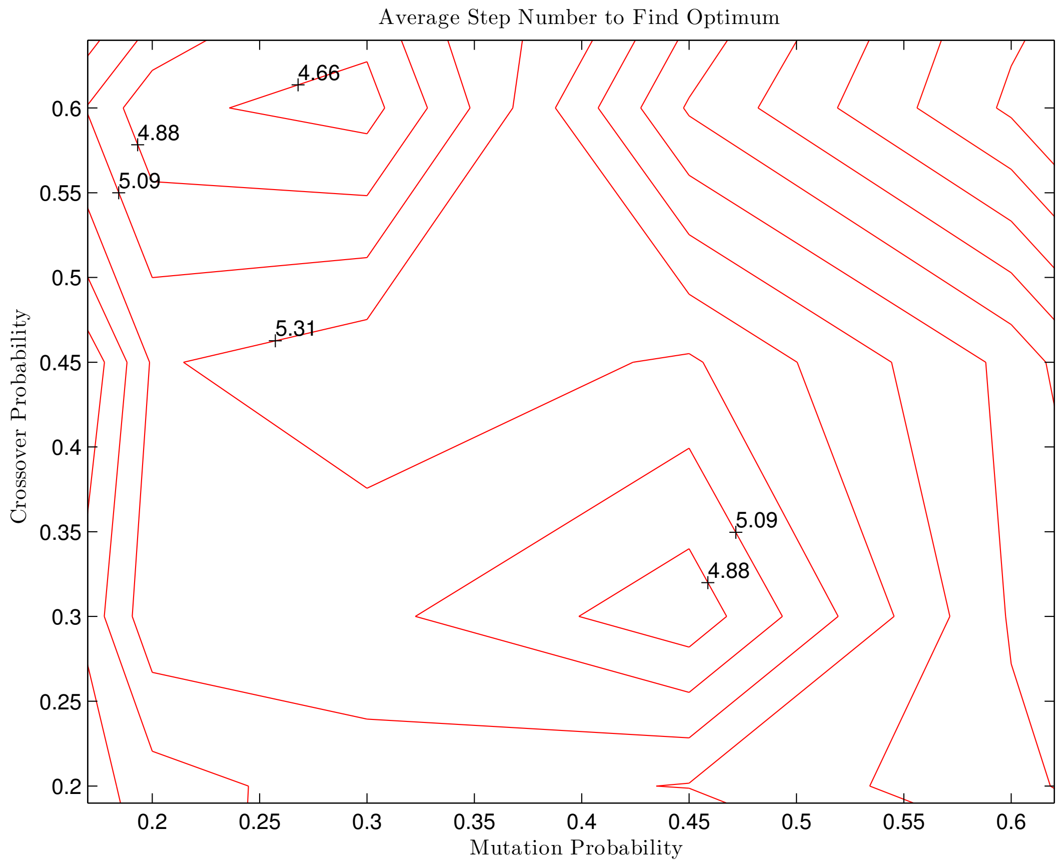

Figure 2.2: Contour plot of the average number of steps to reach the optimum depending on the probability of a mutation and the probability of a crossover

2.3.3 Testing another function

A second function to test is

$$f(x) = \frac{\sin(x)}{x} \tag{2.10}$$

on the interval $D = [0.1, 30] \subset \mathbb{R}$. The global minimum can be found to $x \approx 4.49341$. The procedure to test is the same as for the function above. The only change in the setting is that for this function, the algorithm has been run 500 times. The results are shown in Fig. 2.4-2.5 respectively in Tab. 2.3-2.4.

As one can see, the convergence behavior does not depend on crossover, but only on mutation in this case. Furthermore, the failure percentage does not really depend on the crossover probability, as well.

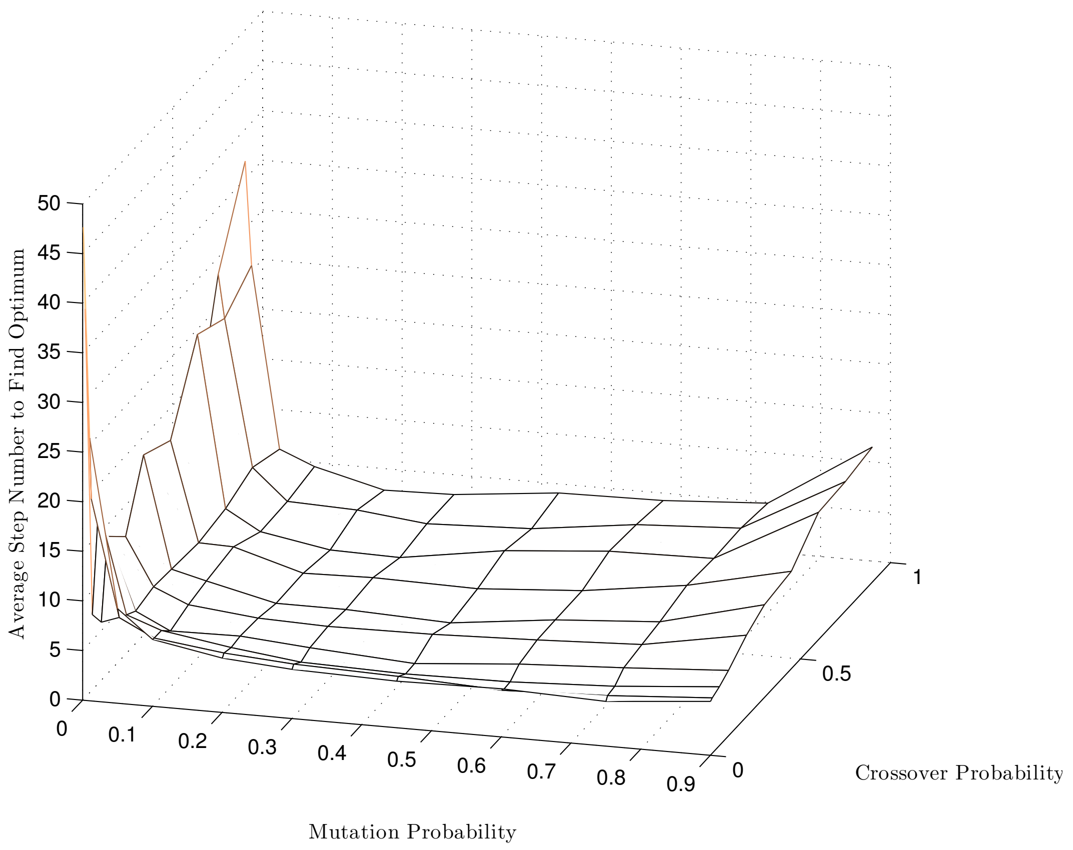

Figure 2.3: Surface plot of the average number of steps to reach the optimum depending on the probability of a mutation and the probability of a crossover

	Steps until	Crossover Probability									
	optimum	0.001	0.010	0.050	0.100	0.200	0.300	0.450	0.600	0.750	0.900
Mutation Probability	0.001	42 (64)	35 (50)	7 (9)	5 (5)	4 (4)	5 (8)	6 (10)	10 (20)	22 (41)	30 (50)
	0.010	27 (37)	20 (25)	18 (23)	15 (23)	12 (19)	17 (27)	17 (23)	23 (34)	23 (39)	24 (36)
	0.050	9 (11)	8 (9)	8 (8)	7 (7)	8 (8)	8 (9)	7 (8)	8 (9)	9 (11)	8 (9)
	0.100	7 (7)	7 (7)	7 (6)	6 (5)	6 (6)	7 (6)	7 (6)	6 (6)	6 (5)	7 (5)
	0.200	5 (5)	6 (4)	6 (4)	6 (5)	6 (4)	5 (4)	5 (5)	5 (4)	6 (5)	5 (4)
	0.300	5 (4)	5 (4)	5 (4)	5 (4)	5 (4)	5 (4)	5 (4)	5 (4)	5 (4)	5 (4)
	0.450	5 (3)	5 (4)	4 (4)	5 (4)	6 (5)	5 (4)	5 (4)	6 (5)	5 (5)	6 (5)
	0.600	5 (4)	5 (4)	5 (3)	5 (5)	6 (4)	6 (5)	6 (6)	7 (6)	7 (6)	6 (7)
	0.750	4 (4)	5 (3)	5 (4)	6 (5)	6 (5)	7 (6)	7 (7)	7 (7)	7 (8)	7 (8)
	0.900	5 (4)	6 (6)	6 (5)	7 (6)	8 (8)	9 (10)	10 (10)	13 (14)	13 (14)	14 (14)

Table 2.1: Average number of steps to reach the probable optimum depending on the probability of a mutation and the probability of a crossover. For each combination of probabilities, the algorithm has been run 200 times until it converged (with a convergence threshold of 200). The number of steps has been decremented by the threshold. Average and standard deviation σ have been calculated with sorting the values in a histogram of $N = 15$ classes.

	Fails in	Crossover Probability									
	%	0.001	0.010	0.050	0.100	0.200	0.300	0.450	0.600	0.750	0.900
Mutation Probability	0.001	26.00	19.00	36.50	32.50	44.50	47.00	56.50	50.50	33.50	35.50
	0.010	0.50	0.00	2.50	4.50	14.00	14.50	9.50	9.50	7.50	12.00
	0.050	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.100	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.200	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.300	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.450	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.600	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.750	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.900	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 2.2: Percentage of failures in finding the real optimum depending on the probability of a mutation and the probability of a crossover (convergence threshold was set to 200)

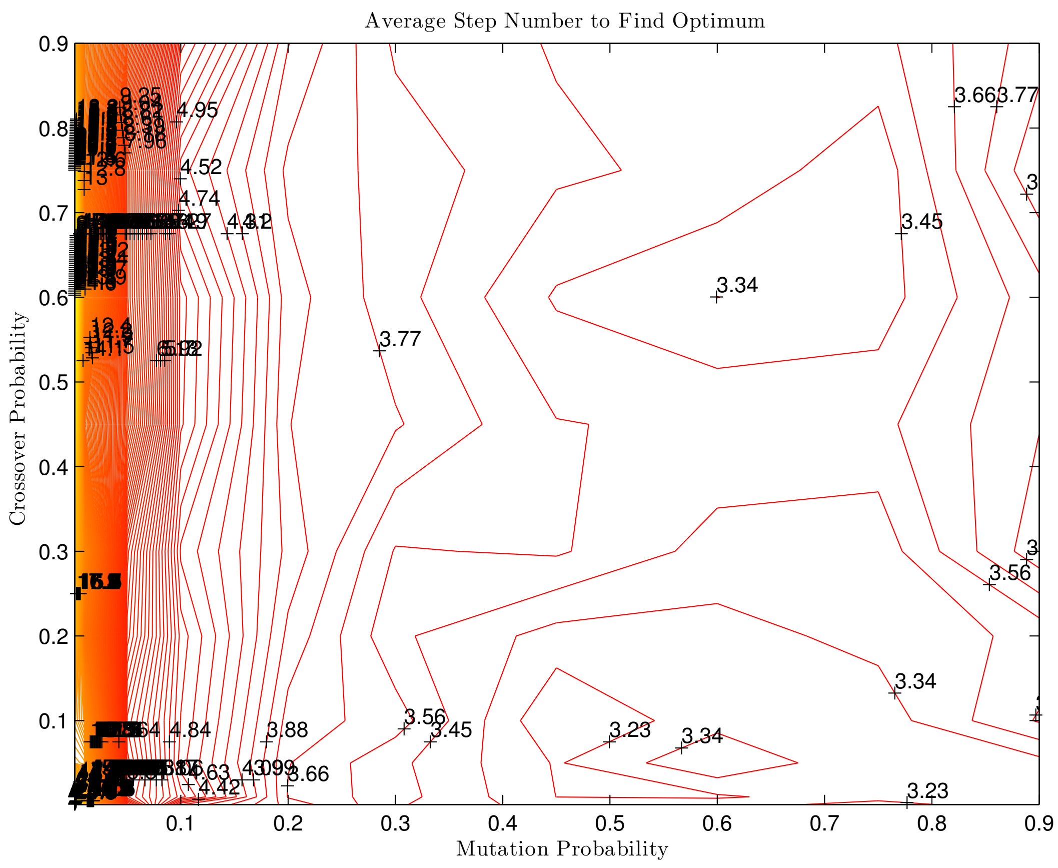

Figure 2.4: Contour plot of the average number of steps to reach the optimum depending on the probability of a mutation and the probability of a crossover

Steps until optimum		Crossover Probability									
		0.001	0.010	0.050	0.100	0.200	0.300	0.450	0.600	0.750	0.900
Mutation Probability	0.001	25 (55)	23 (47)	16 (36)	15 (35)	18 (38)	18 (39)	22 (43)	25 (50)	22 (45)	17 (39)
	0.010	19 (33)	18 (31)	16 (28)	15 (31)	13 (26)	14 (32)	11 (26)	14 (31)	13 (30)	13 (29)
	0.050	6 (9)	6 (6)	6 (7)	6 (6)	6 (8)	6 (10)	8 (16)	7 (12)	7 (14)	9 (17)
	0.100	4 (4)	5 (4)	5 (4)	5 (4)	5 (4)	4 (4)	5 (6)	5 (5)	4 (5)	5 (5)
	0.200	4 (3)	4 (2)	4 (3)	4 (3)	4 (3)	4 (3)	4 (3)	4 (3)	4 (3)	4 (3)
	0.300	3 (2)	3 (2)	3 (2)	4 (3)	3 (2)	4 (2)	4 (3)	4 (3)	4 (2)	4 (2)
	0.450	3 (2)	3 (2)	3 (2)	3 (2)	3 (2)	4 (2)	4 (2)	3 (2)	4 (2)	4 (2)
	0.600	3 (2)	3 (2)	3 (2)	3 (2)	3 (2)	3 (2)	4 (2)	3 (2)	4 (2)	3 (2)
	0.750	3 (2)	3 (2)	3 (2)	3 (2)	3 (2)	3 (2)	4 (2)	3 (2)	3 (2)	4 (2)
	0.900	3 (2)	3 (2)	3 (2)	4 (2)	3 (2)	4 (3)	4 (3)	4 (2)	4 (3)	4 (3)

Table 2.3: Average number of steps to reach the probable optimum depending on the probability of a mutation and the probability of a crossover. For each combination of probabilities, the algorithm has been run 500 times until it converged (with a convergence threshold of 200). The number of steps has been decremented by the threshold. Average and standard deviation σ have been calculated with sorting the values in a histogram of $N = 15$ classes.

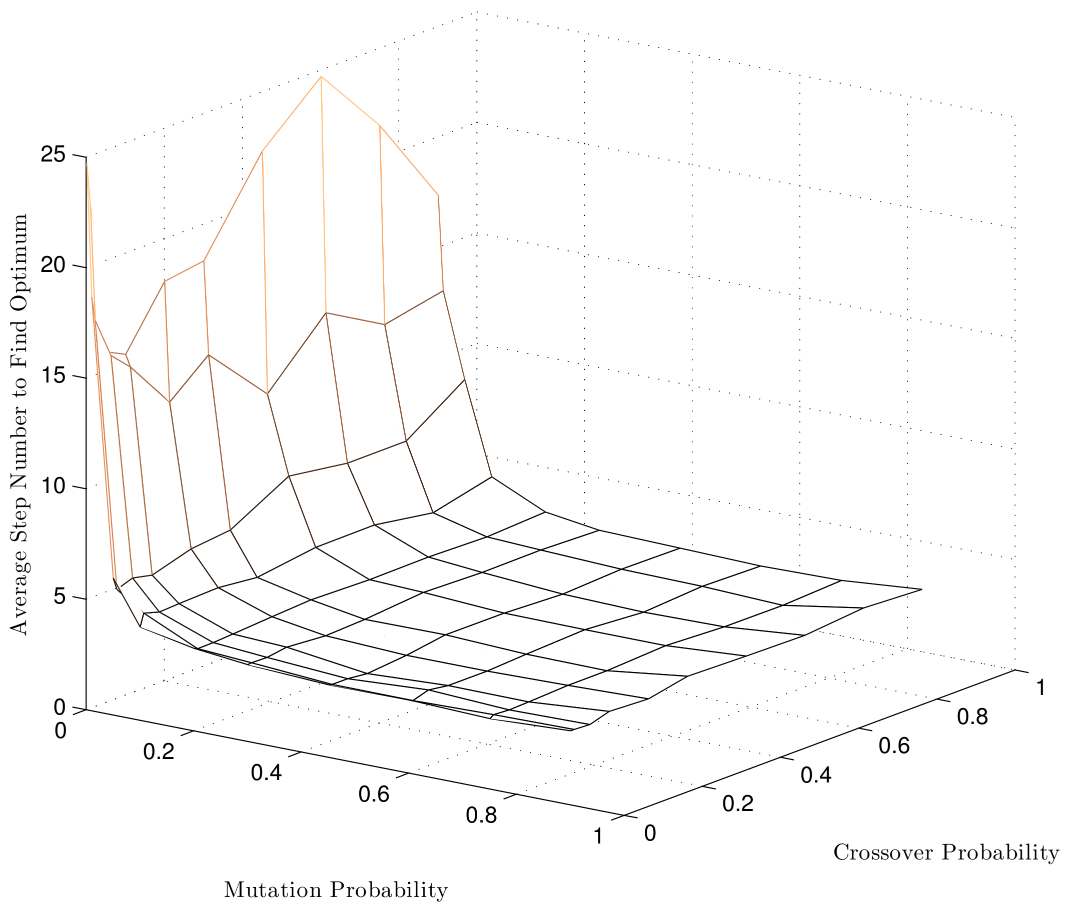

Figure 2.5: Surface plot of the average number of steps to reach the optimum depending on the probability of a mutation and the probability of a crossover

Fails in %		Crossover Probability									
		0.001	0.010	0.050	0.100	0.200	0.300	0.450	0.600	0.750	0.900
Mutation Probability	0.001	19.60	12.80	15.00	19.20	15.40	23.20	24.00	25.40	20.80	26.60
	0.010	0.40	0.00	0.80	2.40	5.80	9.40	9.60	13.00	12.00	13.80
	0.050	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.100	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.200	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.300	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.450	0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.600	0.00	0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.750	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.900	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 2.4: Percentage of failures in finding the real optimum depending on the probability of a mutation and the probability of a crossover (convergence threshold was set to 200)

Chapter 3

The Traveling Salesman Problem

3.1 Specification of the Problem

The “Traveling Salesman Problem” (TSP) was first introduced as a mathematical problem in the 1930s and has become an important problem in mathematics and computer science over time . The problem is formulated such that one has to find the shortest closed route connecting all cities of a certain set. An index set $I = \{0,...,n-1\}$, with $n \in \mathbb{N}$ labels all cities a traveling salesman has to visit. The cities are located at the coordinates $\mathbf{z}_0,...,\mathbf{z}_{n-1}$ with $\mathbf{z}_i = (x_i, y_i) \in \mathbb{R}^2$. He visits every city only once and returns to his starting point at the end of the journey.

The domain of the TSP is the set of all possible permutations of the index set $D = \{\sigma : \sigma \in S_n\}$, where S_n is the symmetric group of order n. On this domain, we want to minimize the route length function

$$f(\sigma) = d(\mathbf{z}_{\sigma(0)}, \mathbf{z}_{\sigma(n-1)}) + \sum_{i=1}^{n-1} d(\mathbf{z}_{\sigma(i-1)}, \mathbf{z}_{\sigma(i)}), \tag{3.1}$$

where $\sigma(i)$ means the index on the i-th position of the permutation σ and the distance is measured as

$$d(\mathbf{z}_i, \mathbf{z}_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \tag{3.2}$$

and can be saved in a symmetric matrix D with $D_{ij} = D_{ji} = d(\mathbf{z}_i, \mathbf{z}_j)$.

3.2 Specification of GA Procedures

Bit strings do not seem to be very handy as representation for chromosomes for this problem. The decimal representation of a bit string could be mapped to a list of permutations of the cities. However, we would need such a list, which is a problem of $O(n!)$ and therefore not affordable. Furthermore, a local search with bit strings would be rather complicated.

Therefore, we decide to introduce a new class of chromosomes. One chromosome is a permutation of the city indices. Using this setting we have to adapt some of the GA procedures. The objective value function is $-f$ and has to be maximized. For elitism and selection we use the standard procedures, however, we have to adapt crossover, mutation and local search.

3.2.1 Mutation and Crossover

Elite members will mutate but only be changed when the mutation actually improved the objective value (to keep current optima in the population). A mutation of a non-elite chromosome is performed when a random number in $[0,1)$ is lower than the mutation probability p_m. Here, mutating a chromosome means to switch two random cities in the route. An effective way to choose two random cities without choosing a certain city twice is described in the following. Select a random city $c_1 \in [0, n-1] \subset]\mathbb{N}$. Then select a random city $c_2 \in [c_1 + 1, c_1 + n - 1]$ and perform $c_2 \leftarrow c_2 \mod n$. With this method, we do not have to check the second city to be identical with the first city.

An effective crossover procedure is described in detail in [4] and roughly works as described in the following. Given two parent chromosomes p_1 and p_2, choose a random city c, add it to the child and look for its position in the parent permutations. Then save the city N_1, respectively N_2 which is following c in the permutations of the parents. If both N_1 and N_2 are already members of the child, add a non-selected city to the child. If N_2 is already a member of the child but N_1 is not, select N_1 as the next member in the child's permutation. If N_1 is already a member of the child but N_2 is not, select

N_2 as the next member in the child's permutation. If both cities are not a member of the child's permutation, compare the distances between c and N_1 and c and N_2, select the N which is nearer to c. Repeat this until the child's permutation is filled.

With this method, we obtain a child route which takes the locally shorter routes of two parent's routes. Because we get only one child from two parents, we have to perform the crossover twice.

3.2.2 Local Search

There exist different approaches for performing a local search in a permutation chromosome.

Permute one City

The first one is to select a random city in the permutation and shift it through all possible positions. Then compare the objective values of all permutations and select the one with the smallest route length.

Permute a Sublist

The second one refers to find a local minimum in a sublist of length l in the route permutation. One evaluates a list of permutations for l using Alg. 2. Afterwards, choose a random city in the route and permute it with the $l-1$ following cities in the route. Compare the objective values of all the permutations and choose the one with minimum length. Note that this method is $O(l!)$. Therefore we should use a small l, e.g. $l = 3$ or $l = 4$. An efficient implementation of this method only computes the route difference between the permuted cities and their neighbors in the route, rather than computing the objective value of the whole routes.

Algorithm 2 function getPermutation(I)

```
Require: $I$ is a set containing the elements to be permutated
Require: $p$ is a list containing lists of permutations of $I$
Require: $a$ is a list containing lists of permutations of $I \setminus I(i)$
  if $|I| = 1$ then
    return $I$
  else
    for $i = 1 \to |I|$ do
      $a \leftarrow$ getPermutation($I \setminus I(i)$)
      for $j = 1 \to |I| - 1$ do
        append $I(i)$ at the end of $a(j)$
        append $a(j)$ at the end of $p$
      end for
    end for
    return $p$
  end if
```

Permute within Nearest Neighbors

A third method refers to a local search by actually creating a sublist of nearest neighbors for all cities and compare the objective values of permutations of this sublist. Again, we can use Alg. 2 for creating a list of permutations for $l < n$ cities. We then need an efficient procedure to find the nearest neighbors of every city. That can be done in the beginning of the program, saving the index of cities' neighbors in an $n \times n$-matrix E, where every column j stands for one city and every i-th row for its i-th nearest neighbor. To fill the matrix, create a list e_j for every city c_j, which contains all cities and their distance to c. Sort that list after the distances and set the corresponding city list as the j-th column in E.

As in the case with the sublist, an efficient implementation of this method only computes the route difference between the permuted cities and their neighbors in the route.

3.3 Tests and Numerical Results

For testing the algorithms, we used the problems from the TSPLIB, specified in [7], especially the problem of 29 cities in Bavaria, Germany (bays29). The optimal route of this problem has a length of 2020.

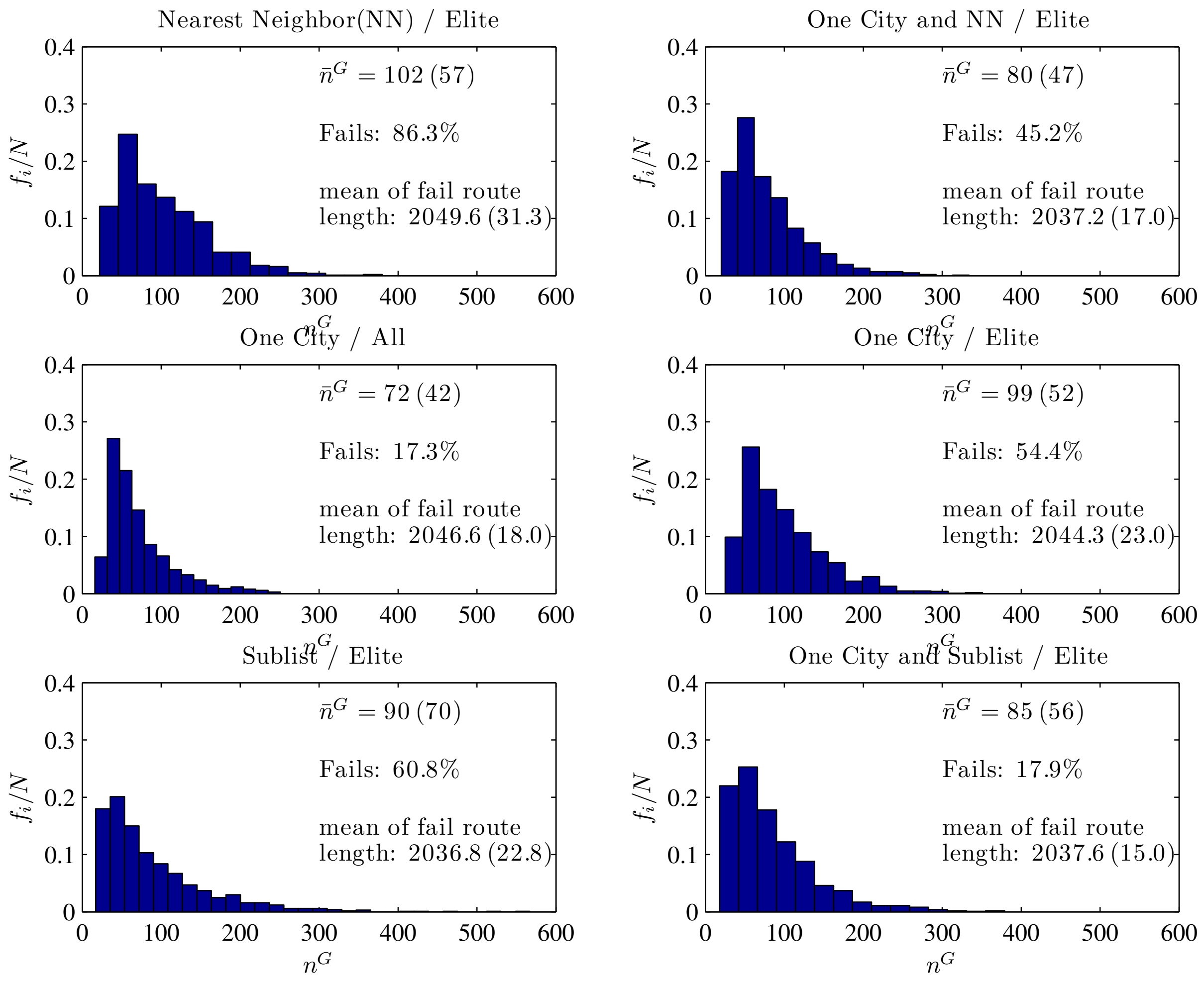

Figure 3.1: Histograms of the convergence behavior of the algorithm for the different approaches of local optimization. The x-axis corresponds to the number n^G of the generation, in which the probable optimum was found. The y-axis shows the corresponding relative frequency. Furthermore the mean was computed according to Eq. (2.8) and displayed with the corresponding standard deviation in brackets, computed using Eq. (2.9). Additionally, we computed the failure percentage (percentage of results not being the real optimum) and the mean route length with its standard deviation in brackets.

3.3.1 Local Search Influence

First, we checked the influence of the different local search algorithms on the convergence behavior. We chose six different kinds of local search algorithms, containing of the three methods described above and variations/combinations of them. The mutation and crossover probability was set to 0.3. The population consisted of 100 members, where the number of elite members was set to be $n_e = 10$. Convergence was defined as having no change in the best member for 100 generations. The length of the sublist for the nearest neighbor and sublist approach was chosen to be 4. For every local search approach, we run the algorithm $N = 1000$ times and sorted the number n^G of the generation in which the probable optimum was found in a histogram of $N_c = 15$ classes ($N_c = 30$ classes for the sublist approach) with class means n_i^G and class frequencies f_i. Afterwards we computed the mean and the standard deviation σ according to Eq. (2.8-2.9). Furthermore we computed the percentage of routes which are not the optimum and mean length with standard deviation of these routes.

The results of this test are shown in Fig. 3.1.

The names of the approaches are chosen after the order in which they have been used and in which chromosomes we looked for a local optimum. As one can see, the fastest convergence in generations is achieved by permuting one city and by looking in all chromosomes. However, looking in all chromosomes is a problem of $O(n)$, while the other algorithms only scale with $O(n_e)$, where n_e is the number of elite members. Therefore, the evaluation of this method grows with an increasing dimension of the problem, while the order of the other approaches is constant.

The next best algorithm is the combination of permuting one city and the search in a sublist of nearest neighbors, both only for the elite members. However, the failure percentage of finding the real optimum is rather high with nearly 50%.

Therefore, the best algorithm for a local optimization seems to be the combination of permuting one city and the search in a sublist, both only for the elite members. Furthermore, the failure percentage seems to be one of the smallest compared with the other approaches. Additionally, the deviation of the route length seems to be small and stable as well.

The histograms seem to follow the prediction of an exponential decrease of convergence probability with increasing step number n^G, as described in [5]. However, this is just an impression and would be needed to confirmed in a deeper investigation.

3.3.2 Crossover and Mutation Probability

The second important test of the algorithm is to find the optimum for the crossover and mutation probabilities. The influence of these have been tested with the local search of permuting one city, afterwards permuting a sublist of length $l = 4$. The population size was 100 with 10 elite members. The convergence threshold was set to 100. The algorithm has been run 200 times for each probability combination. For every run we sorted the number n^G of the generation in which the probable optimum was found in a histogram of $N_c = 12$ classes with class means n_i^G and class frequencies f_i. Afterwards we computed the mean and the standard deviation σ according to Eq. (2.8-2.9). Furthermore we computed the percentage of routes which are not the optimum and mean length with standard deviation of these routes. The results are shown in Fig. 3.2-3.3, respectively Tab. 3.1-3.2.

As one can see in both figures, the influence of the crossover increased. A small crossover probability seems to lead to an increased failure percentage, as well as it increases the convergence step number. That may be explained by the hypothesis that a TSP crossover is more responsible for an optimization than a bit string representing a decimal value.

Mutation increases the time for the algorithm to converge, but delivers a more stable result of the optimum. It possibly provides a good way to lead the current members out of a local optimum.

If one wants to choose a certain crossover/mutation probability, one has to decide if it is satisfying to get a fast result rather than a stable optimum and may decide after the corresponding values in the shown figures or tables.

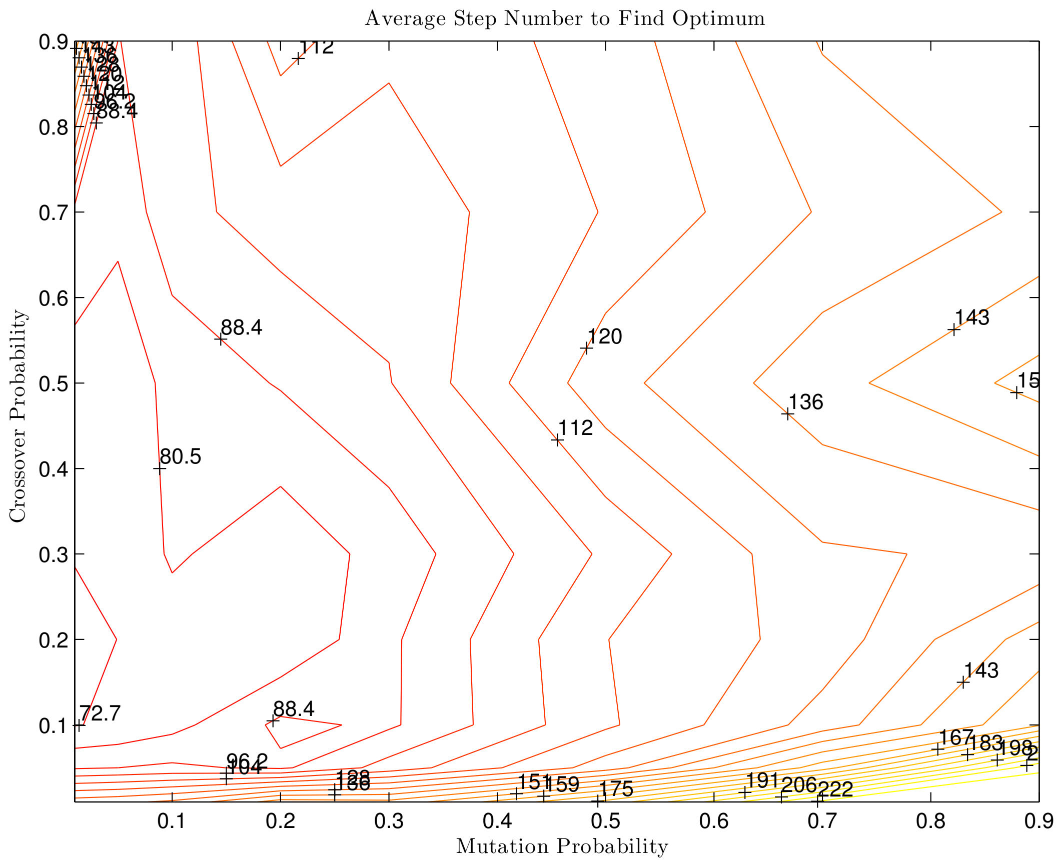

Figure 3.2: Contour plot of the average number of steps to reach the probable optimum depending on the probability of a mutation and the probability of a crossover

3.3.3 Further Tests

Further tests have been done with the problems “berlin52” and “ch130”. While the algorithm converges to the optimum for berlin52, it does not find the optimum for the latter. The optimum would be 6110, but the algorithm mostly stops at

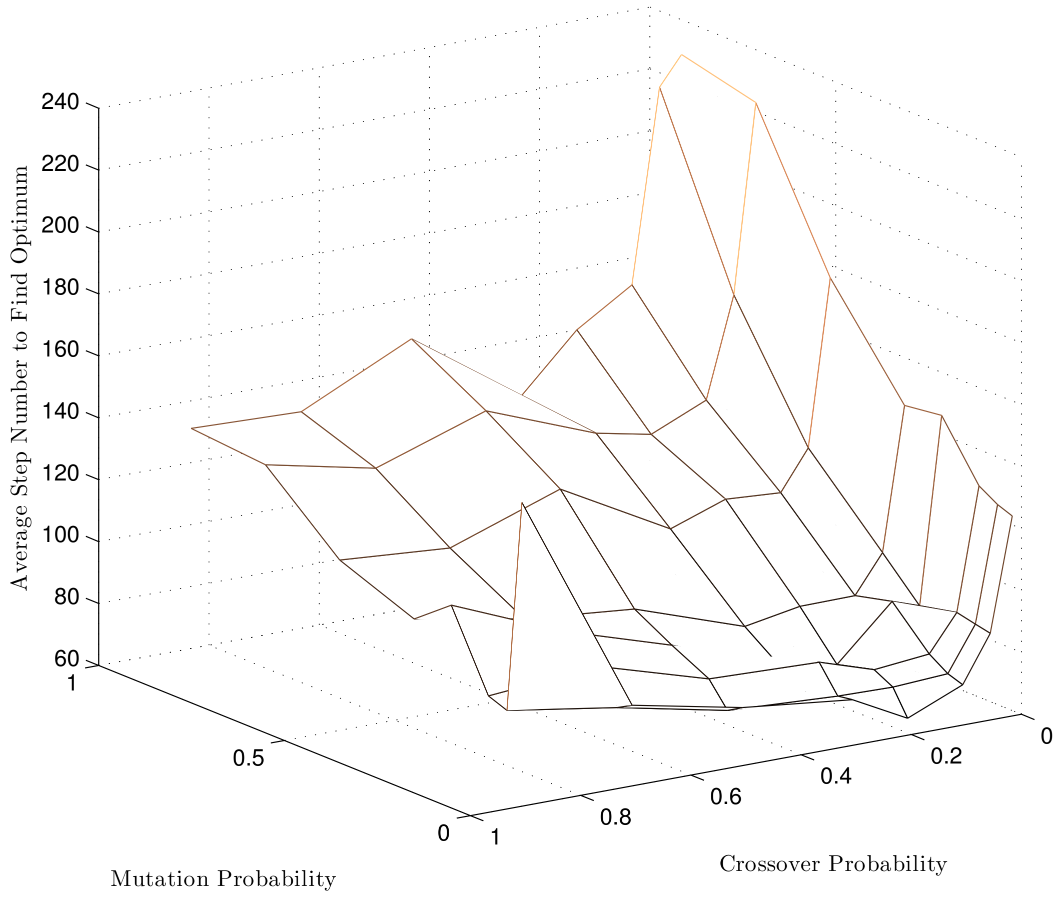

Figure 3.3: Surface plot of the average number of steps to reach the probable optimum depending on the probability of a mutation and the probability of a crossover

route length ≈ 6500. A possible problem could be premature convergence due to mutation [5] or local search.

Steps until optimum		Crossover Probability							
		0.010	0.050	0.100	0.200	0.300	0.500	0.700	0.900
Mutation Probability	0.010	124 (53)	87 (43)	72 (45)	65 (51)	74 (60)	78 (74)	85 (83)	157 (182)
	0.050	126 (61)	88 (47)	74 (45)	73 (51)	73 (50)	75 (45)	83 (61)	88 (62)
	0.100	129 (61)	90 (48)	78 (49)	76 (64)	82 (66)	83 (66)	94 (80)	91 (68)
	0.200	147 (78)	87 (44)	90 (57)	73 (44)	75 (49)	89 (62)	100 (71)	115 (91)
	0.300	146 (75)	99 (55)	87 (49)	87 (58)	84 (58)	96 (76)	99 (68)	106 (81)
	0.500	177 (99)	123 (67)	111 (66)	112 (79)	105 (67)	125 (84)	112 (85)	115 (74)
	0.700	224 (130)	163 (103)	131 (79)	123 (82)	127 (98)	140 (109)	128 (95)	136 (111)
	0.900	230 (142)	221 (146)	158 (93)	147 (103)	129 (84)	154 (115)	137 (93)	138 (94)

Table 3.1: Average number of steps and its standard deviation to reach the probable optimum depending on the probability of a mutation and the probability of a crossover. For each combination of probabilities, the algorithm has been run 200 times until it converged (with a convergence threshold of 100). The number of steps has been decremented by the threshold.

		Crossover Probability							
		0.010	0.050	0.100	0.200	0.300	0.500	0.700	0.900
Mutation Probability	0.010	22.50%	19.00%	23.50%	24.50%	19.50%	19.00%	18.50%	17.50%
		2033.5(14.9)	2043.8(16.1)	2048.4(20.2)	2045.1(16.0)	2043.1(15.1)	2042.0(20.8)	2035.4(12.1)	2040.3(16.5)
	0.050	26.50%	15.00%	18.00%	17.50%	20.50%	11.50%	18.00%	18.50%
		2034.2(11.7)	2043.9(16.8)	2043.8(21.8)	2049.4(24.7)	2040.1(18.4)	2035.3(12.6)	2038.9(14.3)	2039.2(14.1)
	0.100	23.50%	21.50%	20.50%	13.50%	17.50%	20.00%	14.00%	11.50%
		2038.0(18.2)	2040.7(16.1)	2041.2(17.1)	2040.2(16.0)	2041.7(16.3)	2037.5(15.8)	2040.0(17.9)	2039.0(15.7)
	0.200	32.00%	27.00%	18.50%	19.50%	16.50%	14.00%	10.00%	11.00%
		2037.2(14.9)	2042.5(17.3)	2039.2(15.4)	2040.6(16.1)	2038.3(12.8)	2040.4(14.6)	2035.5(14.7)	2048.9(24.9)
	0.300	38.50%	22.50%	13.00%	16.00%	14.50%	13.00%	10.50%	12.50%
		2038.9(15.2)	2045.0(15.0)	2040.1(17.6)	2035.5(14.3)	2042.1(20.0)	2039.5(17.3)	2041.0(13.8)	2044.3(14.3)
	0.500	47.00%	25.00%	18.50%	13.00%	15.00%	13.00%	13.00%	8.50%
		2038.1(14.8)	2045.1(15.8)	2039.5(15.3)	2043.1(15.1)	2039.1(16.8)	2037.7(13.2)	2043.2(23.1)	2035.0(11.5)
	0.700	38.00%	21.00%	15.00%	13.50%	19.50%	11.00%	13.00%	14.50%
		2038.8(18.9)	2040.0(16.4)	2043.3(15.7)	2048.7(16.1)	2044.7(18.3)	2040.7(12.1)	2040.0(11.2)	2044.9(18.9)
	0.900	44.00%	29.00%	20.00%	14.50%	18.00%	16.50%	10.00%	10.50%
		2039.7(19.0)	2039.0(16.4)	2044.0(12.3)	2042.3(11.9)	2043.3(19.6)	2043.2(16.7)	2049.7(10.6)	2041.5(14.6)

Table 3.2: Percentage of failures in finding the probable optimum, as well as mean and standard deviation of the length of failed routes, depending on the probability of a mutation and the probability of a crossover. The convergence threshold was set to 100.

Chapter 4

Summary

GAs seem to provide a promising approach for the heuristic solution of optimization problems.

We applied the bit string approach to search for a minimum of functions. Furthermore we present a GA for a heuristic solution of the TSP. For both, we concentrated on sophisticated methods for a local optimization, with the success to find satisfying procedures for both problems. Furthermore we studied the convergence behavior of the different methods depending on the local search, crossover and mutation probability.

While it was nearly always possible to find the optimum for minimizing a function, there were more problems with the solution of the TSP. The results of the latter were not as stable. Furthermore the algorithm fails in finding the optimum for larger problems. However, one could think of some other approaches of local search, which include shifting a sublist of cities through a permutation or organizing nearest neighbors in lists of being neighbors in the permutation, as well. For a deeper insight in this problem, these approaches could be implemented and studied.

Bibliography

[1] “Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence”, J. H. Holland, 1992, MIT Press/Bradford Books edition

[2] “GENETIC ALGORITHMS AS A COMPUTATIONAL TOOL FOR DESIGN”, Sushil J. Louis, 1997, http://www.cse.unr.edu/~sushil/pubs/thesis/thesishtml/thesishtml.html

[3] “Genetic Algorithms and Machine Learning”, D.E. Goldberg, J.H. Holland, Machine Learning 3: 95-99, 1988, Kluwer Academic Publishers

[4] Script “Laboratory Class Scientific Computing”, B.F. Auer, A.-J. Yzelman, A. van Dam, A. Swart, (2011)

[5] “Predicting Convergence Time for Genetic Algorithms”, Sushil J. Louis, Gregory J.E. Rawlins, ftp://www.cs.indiana.edu/pub/techreports/TR370.pdf, 2011

[6] Weisstein, Eric W. "Gray Code." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/GrayCode.html

[7] “TSPLIB”, G. Reinelt, 1995-2012, http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/index.html