

Assignment 3 – Coding Report

| | |
|--------------------|----------------------------------|
| Course: | IE502014 Artificial Intelligence |
| Date: | 16.04.2018 |
| Name: | Magnus Gribbestad |
| Studno: | 279372 |
| Assignment: | Assignment 3 – Report |

Table of contents

| | |
|---|----|
| Introduction | 1 |
| Challenge 1: Binary or Continuous GA..... | 1 |
| Challenge 1B: String Learning – Non-Binary | 6 |
| Challenge 2: Permutation GA for TSP | 8 |
| Appendix A – Crossover and mutation methods | 19 |
| Appendix B – How to use TSP GA package..... | 23 |
| Bibliografi | 25 |

Introduction

The report below describes my implementations for the coding challenges from the 3rd assignment. Please see Assignment 3 on blackboard, to read the full problem description.

Challenge 1: Binary or Continuous GA

Challenge 1 is to use either a binary or a continuous GA to solve a problem of own choice. The challenge allows the use of libraries and toolboxes.

I chose to solve the string learning problem and solve it from scratch, not using any libraries or toolboxes. In addition, I solved it in two different ways, one where I used binary GA and another approach.

Problem description

The problem I have chosen is referred to as string learning. This is a toy problem where a GA tries to find a predefined target string. The GA cannot use the target string to anything other than getting the fitness of the solutions.

Modelling Binary

I have modelled the problem by defining which symbols that are allowed to use. The allowed symbols are a-z, A-Z, 0-9, space and some other symbols (.,- !?*/%#&<>;_'"~|\$@£\$¤{}()[]=). Since the objective is to solve the problem using a binary GA I encode the symbols into bit strings. I have defined 95

different signals, with a binary representation it is important to fill the all possibilities within a bit range. In this case I need to use 7 bits, which means I need 128 allowed symbols. I use the method suggested by Schaathun (<http://www.hg.schaathun.net/FPIA/week12se2.html>) where the remaining bits are filled with another, already used symbol. As opposed to Schaathun, I have used different symbols (not just one) which will give more variety then having 33 (128-95) 'a' to fill the remaining places.

After having the 128 symbols, I encode the signals into bit strings. I create a lookup table consisting of an integer, bit string and string that represents the symbols. The figures below show some extractions from the lookup table.

| binary | decimal | symbol | binary | decimal | symbol |
|---------|---------|--------|---------|---------|--------|
| 0000000 | 0 | a | 0010110 | 22 | w |
| 0000001 | 1 | b | 0010111 | 23 | x |
| 0000010 | 2 | c | 0011000 | 24 | y |
| 0000011 | 3 | d | 0011001 | 25 | z |
| 0000100 | 4 | e | 0011010 | 26 | A |
| 0000101 | 5 | f | 0011011 | 27 | B |
| 0000110 | 6 | g | 0011100 | 28 | C |
| | | | 0011101 | 29 | D |

With this representation a gene is represented as 7 bits, which means it can be somewhere between 0000000 and 1111111. Every bit string represents a symbol.

Objective function

For this problem I have chosen to use a fitness function (higher number is better) to measure the quality of a chromosome. The fitness functions count how many genes that are correct, the count is normalised in such a way that if all genes are correct the fitness values is 1.0.

The function compares each gene (7 bits per gene) with the corresponding gene (7 bits) in the target string. If the gene's match, the fitness is increase by 1. After checking all the genes in a chromosome, the fitness is normalised, as explained above.

Categorisation

Dimensions

The string learning problem is a multi-dimensional problem. The dimension of the problem is determined by the number of symbols in the target word. The experiment part will show results from different dimensions.

Static

The problem is static, since the string will not change during the search.

Discrete

The problem is discrete since it contains a fixed number of possible solutions/symbols.

Constrained

The problem is constrained to a certain number of symbols. The problem could in theory be even more constrained by allowing only small letters and numbers.

ImplementationSelection

I have chosen to use the roulette wheel selection method in my implementation. This method is a strong method for selection that I know from experience is a good fit to several problems. The main idea of the roulette wheel selection method is that the better a chromosome is, the higher the chances of it being selected.

Crossover

Crossover has the responsibility of generating offspring from parent chromosomes. The parent chromosomes used for mating are selected using the selection method described above. When two parents are selected they have a probability (decided in initialisation face) of generating new kids. If they are not mating, the selected parents are simply passed on to the new generation.

I have implemented the single-point crossover. This is a method that randomly chooses a crossover point. Two offspring are generated from the parents by mixing their genes. The first offspring is made by taking all the genes from parent 1 up to the crossover point, the rest of the genes are taken from parent 2, after the crossover point. The second offspring has the opposite composition (first part from parent 2, second part from parent 1). Illustration of the single-point crossover is showed below.

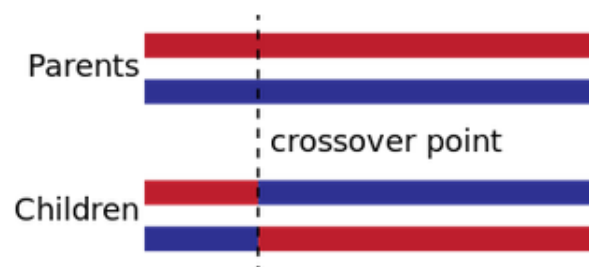


Figure 1 - Single-point crossover (Crossover (GA), 2018).

Mutation

Mutation has the responsibility to provide some randomness into the population, this can help to avoid getting stuck in a local optimum.

I have chosen to implement a bit-flip mutation technique. The idea here is that each bit in a chromosome has a small probability of being flipped. If a gene is chosen to be flipped, a 0 will be changed to a 1, and a 1 will be changed to a 0.

Elitism

I have implemented elitism by copying the best chromosomes from a previous generation into the new one. The elitism rate decides how many chromosomes should be copied. The elitism rate is decided during initialisation.

Stopping Criteria

A stopping criteria is as the name indicates a criteria that checks if the GA should stop. In general, it is a great idea to implement some kind of stopping criteria that measures if the GA has converged (stopped making progress). Normally when you are working with optimisation problems, you don't know how good solutions you will get, so convergence and maximum number of generations is two normal criteria. For my problem (string learning) I know that the best possible score I can get is 1 (all symbols match the target string). Therefore, I have two stopping criteria, one for when the fitness is 1, the other for maximum number of iterations.

Experiment

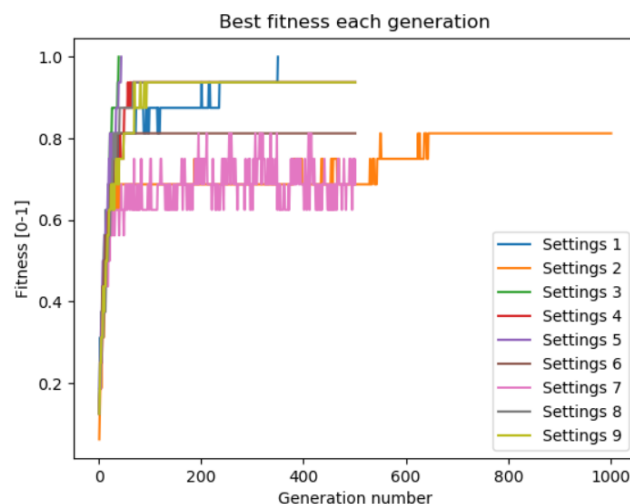
In a GA there are several settings that can be adjusted to achieve better results. Some of these settings are population size, maximum number of generations, crossover rate and mutation rate. In addition, it is possible to choose different selection, crossover and mutation methods, which also can scientifically improve solutions. I have tested the string learning problem with different settings for populations size, generations and rates. I have used the string: "This is a test!!" to test the different settings of the GA.

| Settings Tested | Population Size | Maximum generations | Crossover rate | Mutation Rate | Elitism Rate |
|------------------------|------------------------|----------------------------|-----------------------|----------------------|---------------------|
| Settings 1 | 500 | 500 | 0.7 | 0.001 | 0.2 |
| Settings 2 | 1000 | 200 | 0.7 | 0.001 | 0.2 |
| Settings 3 | 1000 | 1000 | 0.7 | 0.001 | 0.2 |
| Settings 4 | 500 | 500 | 0.5 | 0.001 | 0.2 |
| Settings 5 | 500 | 500 | 0.9 | 0.001 | 0.2 |
| Settings 6 | 500 | 500 | 0.7 | 0.0001 | 0.2 |
| Settings 7 | 500 | 500 | 0.7 | 0.01 | 0.2 |
| Settings 8 | 500 | 500 | 0.7 | 0.001 | 0.0 |
| Settings 9 | 500 | 500 | 0.7 | 0.001 | 0.5 |

Test Case – Target string: “This is a test!!”

| Settings | Best result | Number of generations |
|----------|--------------------|-----------------------|
| 1 | “This is a test!!” | 316 |
| 2 | “Yhis is a Reht!}” | 1000 |
| 3 | “This is a test!!” | 59 |
| 4 | “This is a test!!” | 303 |
| 5 | “This is a test!!” | 283 |
| 6 | “Thia is a teKt![“ | 500 |
| 7 | “Thiq isdaWt4stB!” | 500 |
| 8 | “This is a test!!” | 45 |
| 9 | “This Es a test!!” | 500 |

The figure below shows the best chromosome in each generation for the different settings.



Discuss

From the experiments above I can clearly see that the choice of GA settings clearly affects the result. Some of the settings are not able to find the optimum solution within the generation limit. All of the settings are quickly increasing their fitness within the first 50 generations. Then some of them are struggling to improve. I believe that the reason for this, is the nature of the problem and the fitness function I have chosen. My fitness function is only measuring correct or non-correct symbols. For instance, when 1 symbol is wrong, the GA will have no indication of one letter being better than another. It would have been interesting to try to use more advanced fitness functions that for instance uses a vocabulary to score a string. This assumes that the string should contain some legal words.

The string learning problem is a toy problem that is most useful to learn about GA. Since the assignment asked to implement a binary GA I solved it using binary GA. I have also solved it in an alternative way, which I believe will perform better than the binary GA. One of the main reasons is that you can specify the exact number of symbols you need. With a binary GA you need to fill a number of bits

for it to work properly. This means that if you have 65 symbols you want to include, you need 7 bits (128 combinations) to be able to include all. The remaining 63 combinations needs to be filled with duplicates. The next section briefly explains how I have implemented the string learning in a non-binary way.

Challenge 1B: String Learning – Non-Binary

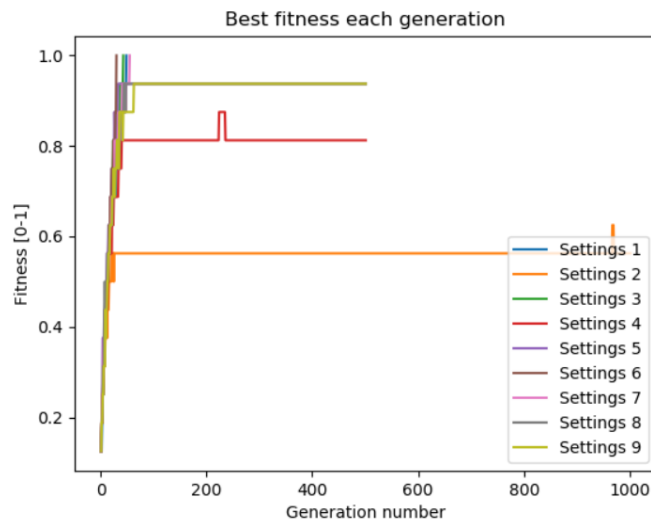
My implementation of the string learning problem, when not using binary GA uses the same ideas as the binary one. The representation of the problem and state space is a little bit different. Instead of representing one chromosome by a series of bits, I represent one chromosome with a series of symbols.

This way makes it possible to use the same selection and crossover technique as before. The mutation method needs to be changed. Instead of flipping a bit, I can now randomly switch from one symbol to another. Since there no longer is need to fill 128 combinations, it is more likely to hit the symbol you want during mutation. If you know what you are looking for in the string, you can limit the symbols to the allowed symbols and further improve the results. I chose to test the new implementation with the same combinations of settings as for the binary GA. The table below show the results:

Test Case 1 – Target string: "This is a test!!"

| Settings | Best result | Number of generations |
|----------|--------------------|-----------------------|
| 1 | "This is a test!!" | 48 |
| 2 | "zhis i!ha4tS#t1!" | 1000 |
| 3 | "This is a test!!" | 42 |
| 4 | "ThCs s ajtest!!" | 500 |
| 5 | "ThDs is a test!!" | 500 |
| 6 | "This is a test!!" | 29 |
| 7 | "This is a test!!" | 54 |
| 8 | "Tris is a test!!" | 500 |
| 9 | "This is a tkst!!" | 500 |

The figure below shows the best chromosome in each generation for the different settings.



It seems like the new implementation performs better than the binary one. This is simply because you are allowed to use fewer symbols, which gives a smaller search space.

Challenge 2: Permutation GA for TSP

Problem description

The second coding challenge in assignment 3 is to implement (from scratch) a GA for solving TSP problems. Two datasets are given:

- 29 cities in Western Sahara
- 38 cities in Djibouti

The assignment text states that the GA should be implemented by using an alternative chromosome encoding that is proposed by Ücoluk (Ücoluk, 2002). The goal is to find the shortest possible tour through all the cities.

Introduction

In this assignment I chose to implement the GA as suggested in the problem description. In addition, I have explored different crossover and mutation methods for permutation problem to see which performs the best. I will try to compare and analyse the results with using both the alternative chromosome encoding and normal methods for permutation problems. I got motivated to test the different methods to try to see if any outperformed others.

I have also included an extra dataset to see how my solution works when increasing the number of cities. The dataset I have included represents 194 cities in Qatar (math.uwaterloo.ca, 2018).

I have used Python for my implementation. My focus has been on implementing the different methods, not on making the most beautiful and flexible code. The next sections will explain my implementation, later I will present and analyse the results.

Implementation

Modelling

For a TSP problem a chromosome is represented by an array with cities (index of cities: 0-N) that are to be visited. It is important that each city is visited only once. In python I have used a list to represent a chromosome.

Read Data

I have made a function which simply reads the data from file and saves it as tuples in a list. The three mentioned data files are included (wi29, dj38 and qa194).

```
map = []
for line in file.readlines():
    ind, y, x = line[0:-1].split(" ")
    map.append((float(x), float(y)))
return map, optimal_route
```

Initialise population

The initial population is generated by using a function which randomly shuffles a list of integers. The list of integers represents the cities (0-n). Each chromosome

is added to a list which holds all the original chromosomes, which represent the original population.

```
# Generate initial population
org_pop = []
for i in range(0, pop_size):
    org_pop.append(list(np.random.permutation(cities)))
```

Fitness

When working with a TSP, the goal is to find an optimal route, which means minimising distance of the tour. The distance of a tour can be calculated by finding the distance (Euclidian distance) between each consecutive city in a tour. It is also important to include the tour from the last city in the tour, and back to the first.

When minimising in GA you are talking about a cost function. I chose to use a fitness function to select (selection method explained next) chromosomes for crossover. This means I needed to convert from distance (cost) to fitness. The fitness function is implemented by taking 1 divided by the distance, times the number of chromosomes in the population. This means that this value will represent the fitness, and the higher number, the better solution.

Selection

The fitness (explained above) is used to select chromosomes for mating. I have chosen to use the roulette wheel selection method. This method uses the cumulative probability to choose chromosomes, which means that better chromosomes is more likely to be chosen. The illustration below shows an example where five chromosomes can be chosen with the roulette wheel function. Chromosome 3 is much better, than chromosome 4, and has therefore a higher probability for being chosen.

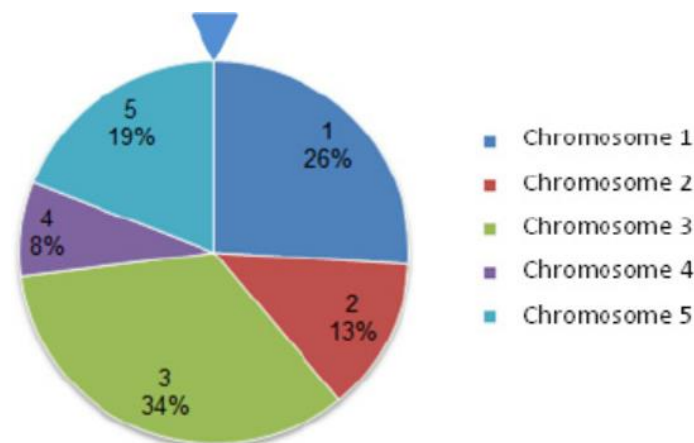


Figure 2 - Illustration of roulette wheel selection (Munther Hameed Abed, 2013)

Decoding/Encoding

As explained in the problem description, Ücoluk has proposed a method for alternative chromosome encoding, which allows the GA to be solved without using permutation crossover methods. This method is according to Ücoluk supposed to perform slightly worse than other methods, but many times faster (Ücoluk, 2002).

The idea behind the methods is that a permutation can be encoded. Crossover is then performed on the encoded chromosomes. The encoded chromosomes are not limited to having non-duplicating cities. This means that it is possible to use normal crossover methods such as the 2-point crossover. After executing the crossover, the offsprings are decoded into permutations which represent a tour through all cities, without any duplicating cities.

I have used the pseudo code represented by Ücoluk, to implement this in Python. My implementation of the encode and decode function is shown in the figures below:

```
def encode(tour):
    inv = [None] * len(tour)
    for i in range(1, len(tour) + 1):
        inv[i - 1] = 0
        m = 1
        while tour[m - 1] != i:
            if tour[m - 1] > i:
                inv[i - 1] += 1
            m += 1
    return inv
```

```
def decode(inv):
    tour = [None] * len(inv)
    pos = [0] * len(tour) * 2
    for i in reversed(range(1, len(tour) + 1)):
        for j in range(1, len(tour) + 1):
            m = i + j
            if pos[m-1] >= (inv[i-1]+1):
                pos[m-1] += 1
            pos[i-1] = inv[i-1] + 1
    tour = pos[0:len(tour)]
    return tour
```

Crossover

I have implemented two different types of crossover methods; one type for permutation problems, and one for normal crossover when using the alternative encoding for chromosomes.

Appendix A will explain how the different crossover methods I have used work, while in this section I will only refer to them by name.

Crossover methods I have used for permutation problems:

- PMX Crossover
- Order 1 crossover (OX)

Crossover methods I have used when using the alternative encoding:

- Single-point crossover
- Two-point crossover
- Uniform crossover

Mutation

Since the chromosomes are decoded after crossover, also the alternative encoding method needs to use mutation methods for permutation problems.

Appendix A will explain how the different mutation methods I have used work, while in this section I will only refer to them by name.

I have implemented and tested these mutation methods:

- Twors Mutation
- Centre inverse mutation (CIM)
- Reverse sequence mutation (RSM)
- Partial Shuffle Mutation (PSM)

Elitism

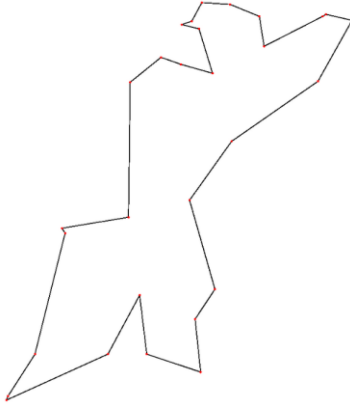
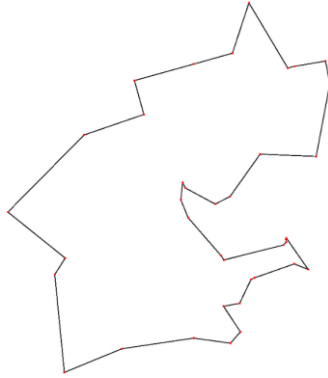
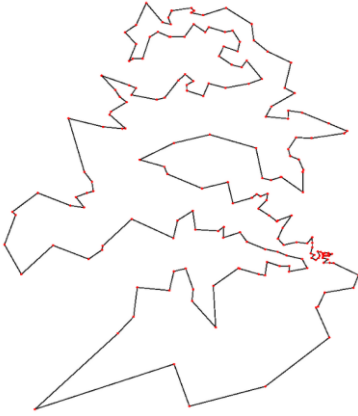
I have implemented elitism by simply selecting the best chromosomes from the previous population to be copied into the new population. The number of chromosomes to be copied are determined by the elitism rate (0.2 means that 20% best chromosomes are copied). Since each crossover produces two offspring, I have made sure that the elitism always choses an even number of chromosomes.

Termination Criteria

I have implemented two termination criteria, if one of them is met the GA will stop. The first criteria are just a maximum number of iterations (generation), while the second criteria checks if the best solution has improved during the last x generations. If the best solution has not improved the GA will be terminated.

Analysis

For this problem the optimal route and the distance of it is stated from the source (math.uwaterloo.ca, 2018). This means that it is easy to compare and see how good solutions the GA provides. The optimal route for the three different datasets are given in the table below:

| Map | Distance | Route |
|----------------|----------|--|
| Western Sahara | 27603 |  |
| Djibouti | 6656 |  |
| Qatar | 9352 |  |

In this analysis I will use the smallest dataset (Western Sahara) to test which combination of crossover and mutation methods that perform the best. I will try to find the best combination both for alternative encoding and permutation.

In order to test the best combination of methods I used each crossover method (PMX, OX) with each mutation method (TWORS, CIM, PSM, RSM). I run each combination 10 times, the table below shows the distance and time consumption of the best route and the average distance and time consumption of the 10 runs. Settings of the GA (population size, crossover probability, mutation probability, elitism ratio etc.) can have great influence on the results. Different methods, needs different settings to perform well, especially when it comes to crossover

and mutation probability. For simplicity I have used the same settings for all methods which is the following:

- Max generations: 1000
- Stall generation limit: 200
- Population size: 300
- Crossover probability: 0.7
- Mutation probability: 0.05

| Crossover | Mutation | Best route | Difference from optimal [%] | Time best route [sec] | Avg. route | Avg. time [sec] |
|-----------|----------|------------|-----------------------------|-----------------------|------------|-----------------|
| PMX | TWORS | 28864.92 | 4.37% | 32 | 34367.20 | 24 |
| PMX | CIM | 27601.17 | -0.0066%* | 36 | 28397.89 | 30 |
| PMX | PSM | 27601.17 | -0.0066%* | 28 | 32774.89 | 32 |
| PMX | RSM | 27601.17 | -0.0066%* | 34 | 28110.13 | 28 |
| OX | TWORS | 27748.71 | 0.52% | 29 | 31197.21 | 23 |
| OX | CIM | 27601.17 | -0.0066%* | 22 | 27864.71 | 24 |
| OX | PSM | 27601.17 | -0.0066%* | 20 | 30020.01 | 30 |
| OX | RSM | 27748.71 | 0.52% | 25 | 28269.03 | 19 |

* Obviously it should not be possible to get a better solution than the optimal. This route is the optimal one, I have discussed this in the discussion section.

The above table shows that the results are quite similar, almost all combination could find optimal solutions. I will refer to the 27601.17 as the optimal route. Among those combination which found the optimal route PMX with RSM and OX with CIM had the lowest averages over the 10 runs. Since all combination performed so well on the smallest dataset (29 cities), I will test the combination with the same settings, but for the medium dataset (38 cities). The table below shows the result.

| Crossover | Mutation | Best route | Difference from optimal [%] | Time best route [sec] | Avg. difference from optimal [%] | Avg. time [sec] |
|-----------|----------|------------|-----------------------------|-----------------------|----------------------------------|-----------------|
| PMX | TWORS | 8236.96 | 19.19% | 40 | 28.7% | 41 |
| PMX | CIM | 6758.28 | 1.51% | 49 | 5.31% | 60 |
| PMX | PSM | 7605.59 | 12.48% | 71 | 25.42% | 60 |
| PMX | RSM | 6659.43 | 0.0515%* | 48 | 6.92% | 46 |
| OX | TWORS | 7290.80 | 8.71% | 54 | 20.06% | 37 |
| OX | CIM | 6659.43 | 0.0515%* | 54 | 1.80% | 42 |
| OX | PSM | 7532.99 | 11.64 | 46 | 23.02% | 49 |
| OX | RSM | 6659.43 | 0.0515%* | 34 | 3.26% | 36 |

* This is in fact the optimal route, look in the discussion section.

The results in the table above shows much bigger differences than the results from the small dataset. This is probably since the dataset has more cities, which means it is a more complex problem. The results might also differ so much from each other because I use the same settings for each combination. It might be that tuning the crossover and mutation probability for some of the combination could improve results. Anyway, the table above shows that PMX with RSM and OX with CIM stills perform very good. OX with RSM also performed good, both when it comes to average distance and time. The rest of the combinations are performing worse. OX with CIM is performing best, with the lowest average difference from the optimal route.

Next, I will see if the inversion method (alternative chromosomes encoding) can outperform any of the previous tests. For this problem I use the medium dataset (Djibouti) directly.

| Crossover | Mutation | Best route | Difference from optimal [%] | Time best route [sec] | Avg. difference from optimal [%] | Avg. time [sec] |
|-----------|----------|------------|-----------------------------|-----------------------|----------------------------------|-----------------|
| 1p | TWORS | 7415.21 | 10.24% | 109 | 25.97% | 112 |
| 1p | CIM | 9011.86 | 26.14% | 163 | 34.06% | 132 |
| 1p | PSM | 8134.72 | 18.18% | 166 | 24.31% | 144 |
| 1p | RSM | 6659.43 | 0.0515% | 156 | 5.96% | 132 |
| 2p | TWORS | 7415.61 | 10.24% | 149 | 23.13% | 119 |
| 2p | CIM | 7594.54 | 12.36% | 143 | 24.46% | 150 |
| 2p | PSM | 7363.26 | 9.61% | 139 | 21.28% | 146 |
| 2p | RSM | 6659.43 | 0.0515% | 119 | 3.01% | 140 |
| Uniform | TWORS | 7688.32 | 13.43% | 154 | 21.87% | 119 |
| Uniform | CIM | 7421.81 | 10.32% | 165 | 21.42% | 141 |
| Uniform | PSM | 7313.06 | 8.98% | 166 | 21.70% | 161 |
| Uniform | RSM | 6659.43 | 0.0515% | 161 | 3.89% | 149 |

The table above shows that two of the combinations are able to find the optimal route. These combinations are 2-point crossover with RSM and uniform crossover with RSM. The next section will discuss the results.

Discussion

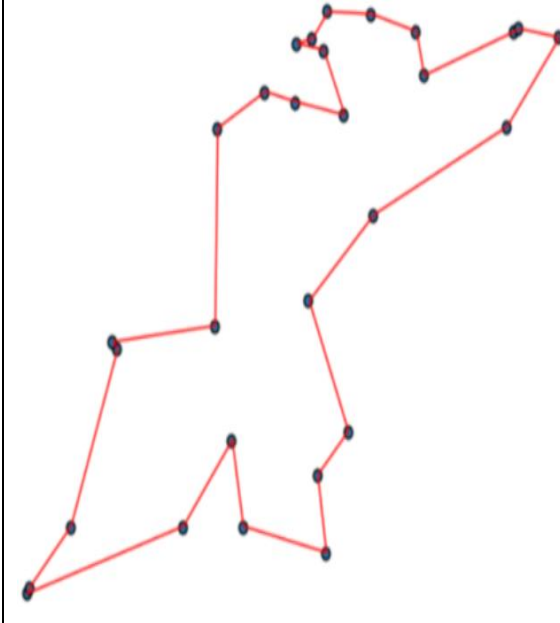
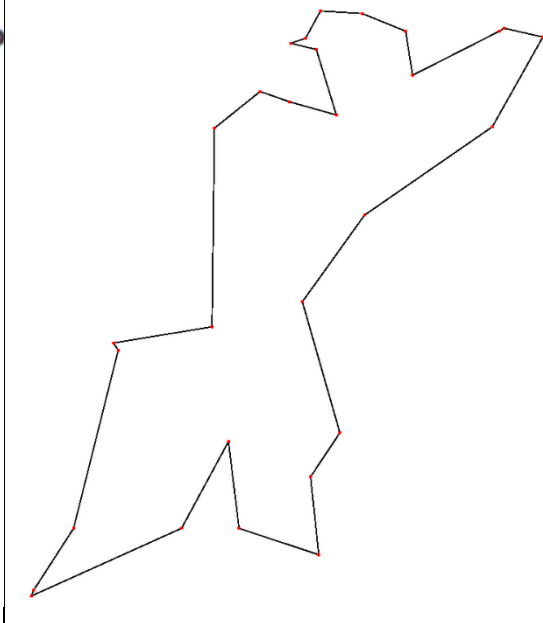
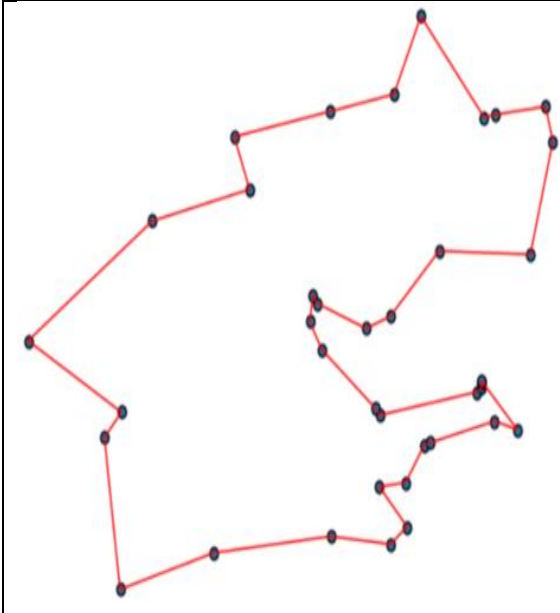
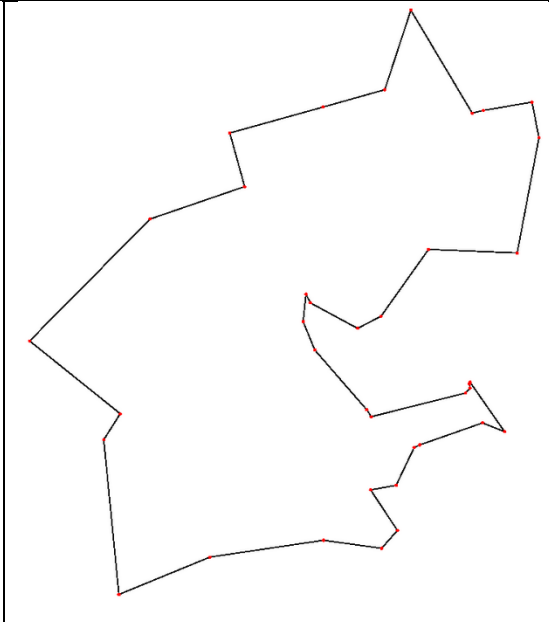
I should probably have tested and documented the different combination of methods more accurately. In addition, if the comparison should be perfect, another loop should be added. This loop should test different GA settings for each combination. It can be that the results are as presented because of the selected values. I have tried to run a couple of random tests where I try different settings to see if any crossover-mutation combinations perform better, I was not able to make any significant improvements. For this assignment I felt it was enough to run the tests that I did.

The results from the GA look pretty good, I even got some routes that outperforms the optimal one. Is that even possible?

Better than optimal?

One of the first questions I asked myself after running the comparisons above was how is it possible that I got a better route than the optimal. I have some theories on that. It seems like the source (math.uwaterloo.ca, 2018) uses integers to represent the shortest route. I use much higher precision than integers, but the error are above 1.0 which means that it cannot be just a simple rounding error. They might have used some other rounding which lead to precision fault. Another option is that I have implemented something wrong when calculating

the distance. To check if the routes I find are as good as the numbers indicates, I will compare the routes manually. I have chosen to compare the best route I have found for both datasets, with the optimal route on the source webpage.

| My best route (Western Sahara) | Optimal route (Western Sahara) |
|---|--|
|  |  |
| My best route (Djibouti) | Optimal route (Djibouti) |
|  |  |

The figures above show that both for the Western Sahara TSP and the Djibouti the best route is the exact same as the optimal route, even though the distances are unequal. As mentioned, there can be several reasons why, but both the optimal routes were found, which is a great result.

Best combination

The results showed that the combination of crossover and mutation method was not that important for the smaller dataset. Several of the combinations were able to find the optimal route. Increasing the problem made it much easier to see the differences between the combinations. The crossover method OX with the CIM mutation method gave best results over the 10 runs, with an average of 1.8% difference from the optimal route. PMX with RSM and OX with RSM performed quite good as well, they both found the optimal route, but had slightly worse averages (6.92% and 3.26%). Interestingly, using the alternative chromosome encoding showed that three of the combinations were able to find the optimal route, and all of them used the RSM mutation method. This tells me that the RSM method seems like a strong method for TSP.

Alternative chromosome encoding vs. permutation methods

Ücoluk stated in his paper that the alternative chromosome encoding performs slightly worse than other methods, but in return it is much faster (Ücoluk, 2002). I have not been able to replicate these findings; my implementation uses a lot more time. My implementation can probably be improved and optimised, but so far the alternative chromosome encoding did not impress. Three of the combinations were able to find the optimal route, but none of them had a better average performance than the OX with CIM.

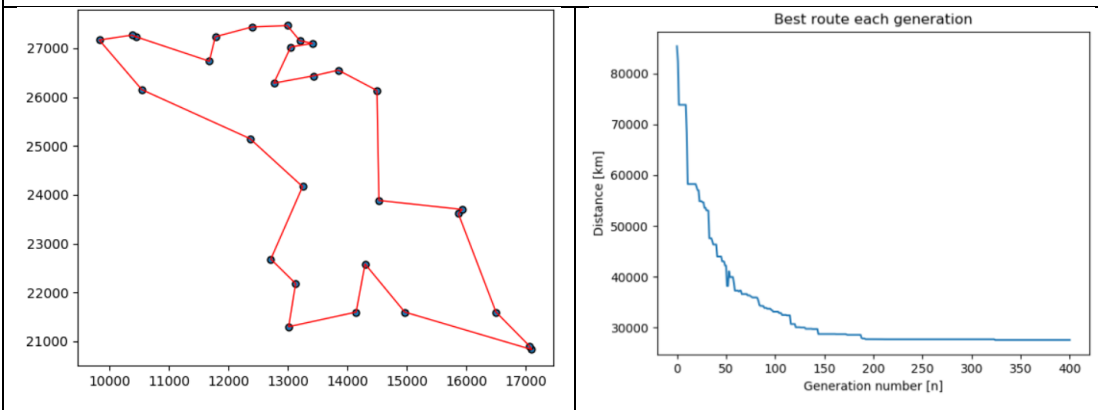
Best results

The assignment states that we are to find the shortest TSP distances for each problem, the table below shows the best routes.

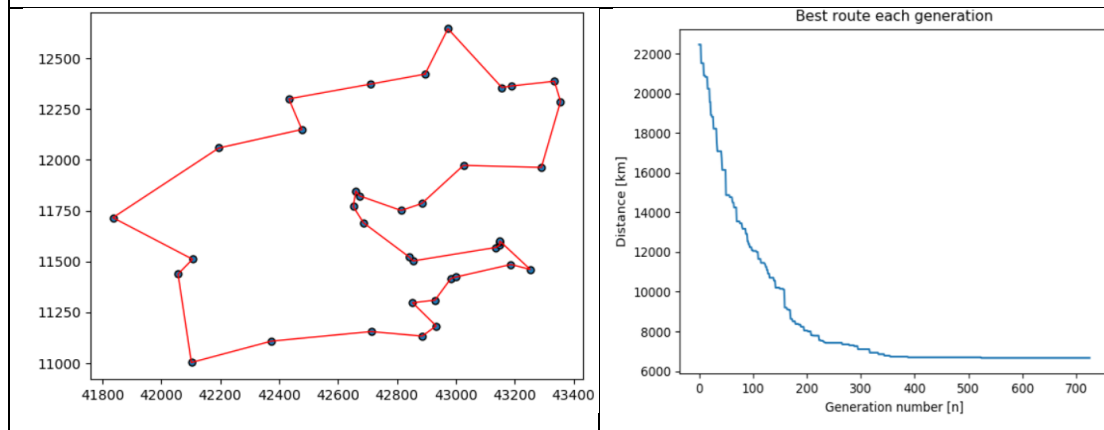
| Map | Cities | Best route | % Difference | Difference | Optimal |
|----------------|---------------|-------------------|---------------------|-------------------|----------------|
| Western Sahara | 29 | 27601.17 | -0.0066% | -1.82 | Yes! |
| Djibouti | 38 | 6659.43 | 0.0515% | 3.43 | Yes! |
| Qatar | 194 | 9897.82 | 5.15% | 545.82 | No! |

The figures below show the best route and the shortest distance for each generation.

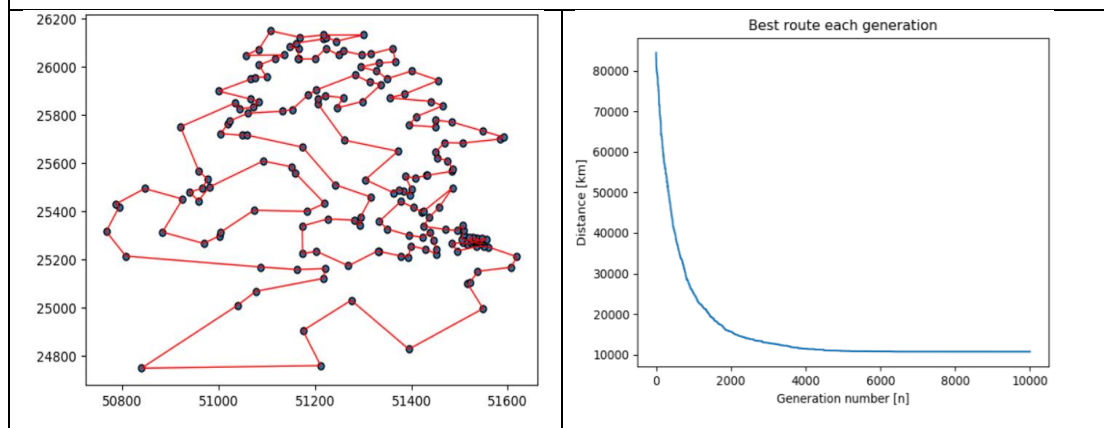
Western Sahara



Djibouti



Qatar



Conclusion

The main objective of the assignment was to find the optimal route between the cities in the provided datasets. The optimal route was found in both datasets, which is a great result.

In addition, some comparisons between methods were presented. TSP can be complex problems, and it is not straight forward to compare different methods and settings. The results show that the RSM method for mutation gave the best results, in general. The best combinations of crossover and mutation where OX with CIM. The alternative chromosome encoding did not perform as good as expected; it was slower than the other methods.

The comparisons were performed with similar settings for all methods, which of course could have great influence on the results. Look at this as a preliminary step to gain knowledge, future steps would need to include a more thorough analysis were each set of combinations is optimised.

Appendix A – Crossover and mutation methods

This appendix will explain the crossover and mutation methods used in the assignment.

Crossover methods:

Crossover methods I have used for permutation problems:

PMX Crossover

The PMX crossover method is used for crossover in permutation problems. It is a bit complex to explain in words, but with the included example below it should be understandable. A single-point is randomly selected. The first child is copied from the first parent. Then the value of the first gene in the second parent should replace the first gene in the child. In order to do this, the first gene in the child must be swapped with the gene (in the child) containing the value from the parent.

Next, the value of the second gene in the second parent, should replace the value in the second gene in the child. To do this, the second gene in the child must be swapped with the gene containing the value of the second gene from the parent. This is done for each gene until the single-point is reached.

To illustrate the PMX crossover I have taken an example that Ücoluk used in his paper (Ücoluk, 2002). The example is showed below:

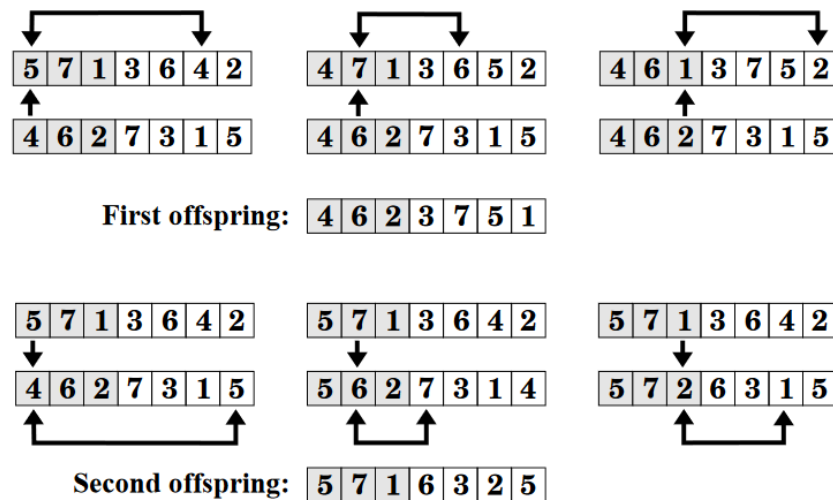


Figure 3 - Example of PMX crossover (Ücoluk, 2002)..

From the figure above, you can see that the value of the first gene in the second parent (4) should replace the first gene in the child (5). In order to do this, the first gene in the child (5) must be swapped with the gene containing the value from the parent (4). Next 7 and 6 are swapped, then 1 and 2.

Order 1 crossover (OX)

Order 1 crossover (often referred to as OX or order crossover) is a crossover method for permutation problems. The method is based on randomly selecting a section within the parents. This can for instance be the 4 middle genes. Child 1 will then directly inherit these 4 middle genes from parent 1 (into the same position in the child), while child 2 will inherit from parent 2. The remaining genes are then filled with values from the other parent.

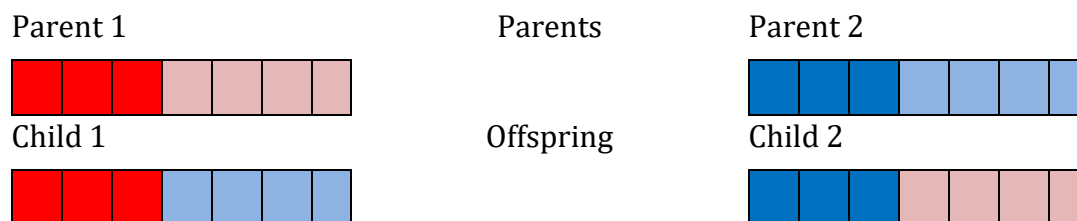
Since this is a permutation crossover, it is important that there are no duplicate values in the child. Therefore, the child starts with looking at the index of the first non-assigned gene, in the other parent. If this gene/value does not exist in the child already, it is copied into the child. If the value already exists in the child, it must continue to check the next gene of the other parent. It is easier to show this with an example. I will show how this crossover works for generating one child, the process is the same to generate the second child, only opposite.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Parent 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Parent 2 | 9 | 7 | 6 | 1 | 3 | 5 | 8 | 2 | 4 |
| Selected section copied to child 1 | | | | | | | | | |
| Child 1 | | | | 4 | 5 | 6 | 7 | | |
| Starts with first gene of parent 2, which is 9. Since 9 does not exist in child 1 yet, it can be copied. | | | | | | | | | |
| Child 1 | 9 | | | 4 | 5 | 6 | 7 | | |
| The next gene in parent 2 is 7, this already exist in child 1. The next gene is checked, which is 6. It also exists in child 1. The next gene is 1, this is not in child 1 yet and therefore inherited in the next free position. | | | | | | | | | |
| Child 1 | 9 | 1 | | 4 | 5 | 6 | 7 | | |
| The next gene is 3, which can be inherited, and so on. Finally, you get the child below. | | | | | | | | | |
| Child 1 | 9 | 1 | 3 | 4 | 5 | 6 | 7 | 8 | 2 |

Crossover methods I have used when using the alternative encoding:

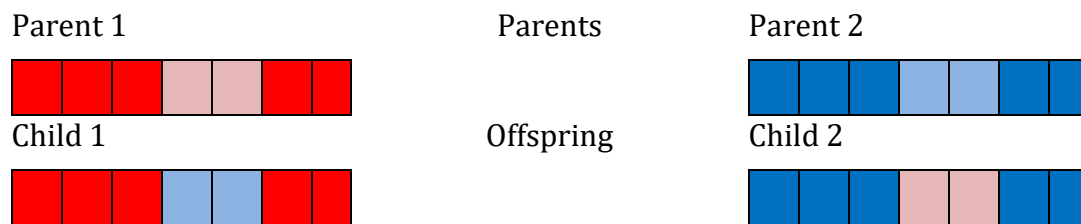
Single-point crossover

Single-point crossover is a method that can be used for none permutation problems. Two chromosomes (parents) are needed for the crossover method. A random point in the chromosomes are chosen, such that both parents are split into two sections. These parents generate two offspring. Offspring 1 takes the first section from parent 1 and the second section from parent 2. Offspring 2 takes the first section from parent 2 and the second section from parent 1 (Wikipedia, 2018). Here I show an example of such a crossover:



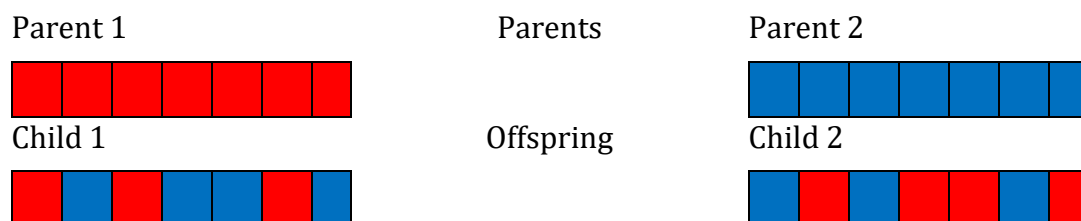
Two-point crossover

Two-point crossover is quite similar to the single-point crossover, but for this method two points are selected. This means that each parent is divided into three sections. The first child will then inherit the first and last section from parent 1, and the middle section from parent 2. The second child will inherit the first and last section from parent 2, and the middle section from parent 1 (Wikipedia, 2018). This is illustrated in the figure below.



Uniform crossover

Uniform crossover is different from the other methods so far. This method goes through every gene, and determines if it should be inherited from parent 1 or 2. If the probability is set to 0.5, each gene would have 50% chance of being from parent 1 (Wikipedia, 2018). The example below shows how crossover can be executed with a 0.5 probability.



Mutation methods:

Twos Mutation

Twos mutation is a mutation method that also can be referred to as swap. Two genes are randomly chosen, and their position are swapped (Otman Abdoun). Here is an example, where gene number 3 and 5 are randomly chosen.



Centre inverse mutation (CIM)

The centre inverse mutation method chooses one random point, which divides a chromosome into two sections. The two sections are flipped (Otman Abdoun). Here is an example, where the random point is selected between gene 3 and 4.



Reverse sequence mutation (RSM)

The reverse sequence mutation methods chooses two random points that selects a section of a chromosome. The gene sequence inside the selected section is flipped / reversed (Otman Abdoun). Here is an example where index 3 and 6 is randomly chosen.



Partial Shuffle Mutation (PSM)

The partial shuffle mutation method iterates through each gene in a chromosome. Each gene uses the mutation probability to determine if the gene should be swapped with another. If the gene is determined to be swapped, the gene it will be swapped with is randomly chosen (Otman Abdoun). Below is an example where gene number 2 and 6 was determined to be swapped with gene 4 and 5 respectively.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | → | 1 | 4 | 3 | 2 | 5 | 6 | 7 |
| 1 | 4 | 3 | 2 | 5 | 6 | 7 | → | 1 | 4 | 3 | 2 | 6 | 5 | 7 |

Appendix B – How to use TSP GA package

For the assignment I have implemented my own GA package for solving TSP. This appendix describes the dependencies and how to use it.

The TSP_GA package is developed in Python3 (used version Python 3.6.3), but should work for all Python3 versions.

Python3 packages that must be installed to use the TSP_GA package (some of them you might have installed already):

- numpy
- random (most likely installed as default)
- matplotlib
- time (most likely installed as default)
- math (most likely installed as default)
- copy (most likely installed as default)
- warnings (most likely installed as default)

After installing these packages, the GA_TSP package should be ready to use. The “Solve_TSP.py” shows an example implementation of how to use the package.

Here is an explanation on how to use it, and which parameters that are available. In order to use the package, you need to do the following steps:

1. Instantiate an object of the GA_TSP class.
2. To use the GA run the “runGA()”-method. This method takes a lot of parameters that decide on which data and settings to use. The next section describes these parameters.
3. When the GA has finished the run there are some function that can be called to retrieve the results:

| Function name | What it does? |
|-------------------------|--|
| get_best_route() | Returns the best route (city indexes), shortest distance and time consumption. |
| plot_best_route() | Returns a map that indicates the best route through the cities. |
| plot_best_generations() | Returns a graph showing the shortest distance for each generation. |
| plot_avg_generations() | Returns a graph showing the average distance for each generation. |

Parameters

| Name | Datatype | Optional | Default | Description |
|------------------|----------------|----------|-----------------|--|
| Map * | List of tuples | No | - | The map of the location of the cities |
| genLimit | Int | | - | Max number of generations to run |
| stallLimit | Int | | - | Max number of generations without improvement |
| popSize | Int | | - | Population size |
| nStops | Int | Yes | Elements in map | Number of stops, if not chosen, all cities from map are used |
| pCross | Float [0-1] | Yes | 0.7 | Probability of doing crossover |
| pMut | Float [0-1] | Yes | 0.05 | Probability of doing mutation |
| eliteRatio | Float [0-1] | Yes | 0.2 | Elite ratio: Ratio of best chromosomes from previous tour to keep. |
| optimalRoute | Float | Yes | None | If optimal route is known it can be added. |
| crossover_method | String | Yes | "pmx" | Which crossover method to use: |

| | | | | |
|-----------------|---------|-----|-------|---|
| | | | | pmx, ox, 1p, 2p or uniform. 1p, 2p and uniform uses chromosome encoding. |
| mutation_method | String | Yes | "rsm" | Which mutation method to use: rsm, psm, cim, twors |
| Plotting | Boolean | Yes | True | Decides if best route each generation should show. This turned on slows down the performance. |
| map_name | String | Yes | None | Name of cities/map. Used only in the title of the plot. |

* Package includes examples datasets that can be retrieved by using the `get_tsp_data` function

Bibliography

Crossover (GA). (2018, April). Hentet fra Wikipedia:

[https://en.wikipedia.org/wiki/Crossover_\(genetic_algorithm\)#Single-point](https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)#Single-point)

math.uwaterloo.ca. (2018). *TSP DATA - Qatar*. Hentet fra

www.math.uwaterloo.ca:

<http://www.math.uwaterloo.ca/tsp/world/qa194.tsp>

Munther Hameed Abed, A. Y. (2013). *Hybridizing Genetic Algorithm and Record-to-Record Travel Algorithm for Solving Uncapacitated Examination Timetabling Problem*.

Otman Abdoun, J. A. (u.d.). *Analyzing the Performance of Mutation Operators to Solve the Travelling Salesman Problem*.

Wikipedia. (2018). *Crossover (genetic algorithm)*. Hentet fra Wikipedia:

[https://en.wikipedia.org/wiki/Crossover_\(genetic_algorithm\)](https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm))

Ücoluk, G. (2002). *Genetic algorithm solution of the TSP avoiding special crossover and mutation*.