06.09.2018

Navn: Torstein gombos

Username: Tagombos

Travelling Salesman Problem

Biologically Inspired Computing

Innhold

[Introduction 2](#_Toc525115191)

[Tools 2](#_Toc525115192)

[Code 2](#_Toc525115193)

[Features 2](#_Toc525115194)

[How to run 2](#_Toc525115195)

[Code Structure 3](#_Toc525115196)

[Exhaustive search 3](#_Toc525115197)

[What is the shortest route and what is the distance? 3](#_Toc525115198)

[How long did it take the program to find it? 3](#_Toc525115199)

[How long would you expect it take with all 24 cities? 3](#_Toc525115200)

[Hill Climbing 4](#_Toc525115201)

[How well does hill climber perform on the same first 10 cities? 4](#_Toc525115202)

[Performing 20 hill climbs on random sequence of 10 cities: 4](#_Toc525115203)

[Performing 20 hill climbs on random sequence of 24 cities: 4](#_Toc525115204)

[Genetic Algorithm 5](#_Toc525115205)

[Performing genetic algorithm with three different population sizes: 5](#_Toc525115206)

[Conclude which is better in terms of tour length and number of generations of evolution: 5](#_Toc525115207)

[How well does genetic algorithm perform compared to exhaustive search on time 6](#_Toc525115208)

[Hybrid algorithm 7](#_Toc525115209)

[Results of baldwinian 8](#_Toc525115210)

[Results of Lamarckian 9](#_Toc525115211)

[Appendix 10](#_Toc525115212)

[Code 10](#_Toc525115213)

[Documentation 10](#_Toc525115214)

[Figures 10](#_Toc525115215)

# Introduction

The travelling salesman problem is an optimization problem about finding the shortest route between cities around the world. I will in this report implement various optimization methods and test performance on time and result. The questions in the assignment are answered in each chapter.

# Tools

I program the methods using python 3.6. The data used comes from “European\_citites.CSV”.

# Code

This is an explanation on how to run the script and how the script is written. For documentation about the various modules, methods and functions, see the documentation files in the same folder. An explanation on how to run the script exists also in the readme file.

## Features

When running the code without any arguments, the script will execute an exhaustive search on the 10 first cities in the csv file and yield the shortest route, its distance and how long it took to find it.

If Hill Climber is chosen, the script will execute a hill climber search on a number of cities provided by the user and yield the shortest route found and its distance. It then proceeds to do 20 runs and prints the best, the mean and standard deviation of the runs

If Genetic Algorithm is chosen, the script will create three different population sizes and execute 20 runs of a evolutionary algorithm for each population size and return the same results as for hill climber. Population size and number of generations to use can be modified in the beginning of the main function in TSP.py

If Hybrid is chosen, the script will execute 20 runs in the same fashion as with Genetic Algorithm, but will add a local search with Hill Climber for each generation. The user can provide either Lamarckian or Baldwinian as learning model. Lamarckian is default if nothing is chosen.

### How to run

Everything is run from the main script TSP.py. This reads the data from the CSV file and runs the different methods and algorithms according to the arguments given from the user. It then returns the results and prints and plots in a manner that can answer the questions in the assignment.

The user can input the following arguments in the command line terminal when running the script:

*What algorithm to use:*

1. *– m ex Exhaustive search*
2. *– m hc Hill climber*
3. *– m ga Genetic Algorithm*
4. *– m hybrid Hybrid Algorithm*

*Route length:*

*– r <number of cities> max 24, if exhaustive search, 11 cities is max.*

*Learning model (Only for hybrid mode):*

1. *– l lamarckian Uses the Lamarckian learning model (Default if none is chosen)*
2. *– l baldwinian Uses the baldwinian learning model.*

Example to run genetic algorithm for 24 cities: *python TSP.py -m ga -r 24*

## Code Structure

Code is divided into three scripts

1. *TSP.py*

Everything is run from here

1. *Routes.py*

Everything regarding setting up routes or calculating distances is done from here

1. *Search\_algorithms.py*

All the algorithms used for the assignment is implemented here

# Exhaustive search

Exhaustive search will use the function *create\_permutation of routes* to create all the combinations of routes for the first *n* cities. Then loop through all combinations and find the best one

## What is the shortest route and what is the distance?

Implementing exhaustive search for 10 cities yielded following route:

***The shortest route using exhaustive search:***

*Barcelona Belgrade Istanbul Bucharest Budapest Berlin Copenhagen Hamburg Brussels Dublin Barcelona*

*The total distance is 7486.31km*

*Code execution: 3.715876340866089s*

## How long did it take the program to find it?

The code used about 3.7s when finding optimal route for 10 cities

## How long would you expect it take with all 24 cities?

Since it does not matter what the starting point is as long as the sequence of cities is the same. One can therefore do permutations

|  |  |  |  |
| --- | --- | --- | --- |
| Number of cities | Distance(km) | Time(s) | Permutations |
| 6 | 5018.81 | 0.00203 | 120 |
| 7 | 5487.89 | 0.00697 | 720 |
| 8 | 6667.49 | 0.04188 | 5040 |
| 9 | 6678.55 | 0.36299 | 40320 |
| 10 | 7486.31 | 3.54444 | 362880 |
| 11 | 8339.36 | 39.1216 | 3628800 |

(converted answer from seconds to years)

# Hill Climbing

## How well does hill climber perform on the same first 10 cities?

Implementing hill climbing for the first 10 cities yielded:

***The shortest route:***

*Hamburg -> Copenhagen -> Berlin -> Budapest -> Bucharest -> Istanbul -> Belgrade -> Barcelona -> Dublin -> Brussels -> Hamburg ->*

*The total distance is 7486.31km*

*Code execution: 0.013934135437011719s*

However, it does not always reach global minimum,.

## Performing 20 hill climbs on random sequence of 10 cities:

*The shortest route was 7486.31km:*

*Istanbul -> Belgrade -> Barcelona -> Dublin -> Brussels -> Hamburg -> Copenhagen -> Berlin -> Budapest -> Bucharest -> Istanbul ->*

*The longest route was 9410.61km:*

*Belgrade -> Brussels -> Dublin -> Barcelona -> Istanbul -> Bucharest -> Copenhagen -> Hamburg -> Berlin -> Budapest -> Belgrade ->*

*The mean was: 7998.818500000001*

*The standard deviation was: 549.2403150215758*

*Code execution: 0.31919145584106445s*

## Performing 20 hill climbs on random sequence of 24 cities:

*The shortest route was 13483.66km:*

*Barcelona -> Madrid -> Dublin -> London -> Brussels -> Paris -> Milan -> Munich -> Prague -> Vienna -> Budapest -> Belgrade -> Sofia -> Istanbul -> Bucharest -> Warsaw -> Berlin -> Hamburg -> Copenhagen -> Stockholm -> Saint Petersburg -> Moscow -> Kiev -> Rome -> Barcelona ->*

*The longest route was 17955.05km:*

*Saint Petersburg -> Barcelona -> Madrid -> Paris -> Brussels -> Copenhagen -> Stockholm -> Moscow -> Kiev -> Hamburg -> Dublin -> London -> Berlin -> Prague -> Munich -> Milan -> Rome -> Vienna -> Belgrade -> Sofia -> Istanbul -> Bucharest -> Budapest -> Warsaw -> Saint Petersburg ->*

*The mean was: 15190.461*

*The standard deviation was: 1045.0609307640743*

*Code execution: 1.2655680179595947s*

# Genetic Algorithm

The genetic algorithm I have written follows simple GA structure:

|  |  |
| --- | --- |
| Initialize Population | An *x* amount of random generated routes |
| Evaluate Population (Fitness) | Total distance for each route |
| Select Parents | Based on a fitness proportionate selection  Uses windowing to scale probabilities |
| Create Offspring | Uses partially mapped crossover between selected parents |
| Mutate Offspring | Random swap on a small selection of offsprings to keep some diversity |
| Replace population | Normally replaces entire population with offspring unless hybrid mode is selected. |

## Performing genetic algorithm with three different population sizes:

### Conclude which is better in terms of tour length and number of generations of evolution:

**Tour length: All 24 cities, Best of: 20 runs,**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Population** | **Evaluations** | **Best fitness(km)** | **Worst fitness(km)** | **Mean(km)** | **Standard deviation(km)** |
| 200 | 80 000 | 12511 | 17410 | 14869 | 1031 |
| 400 | 160 000 | 13049 | 16117 | 14778 | 873 |
| 500 | 200 000 | 12960 | 16001 | 14307 | 815 |

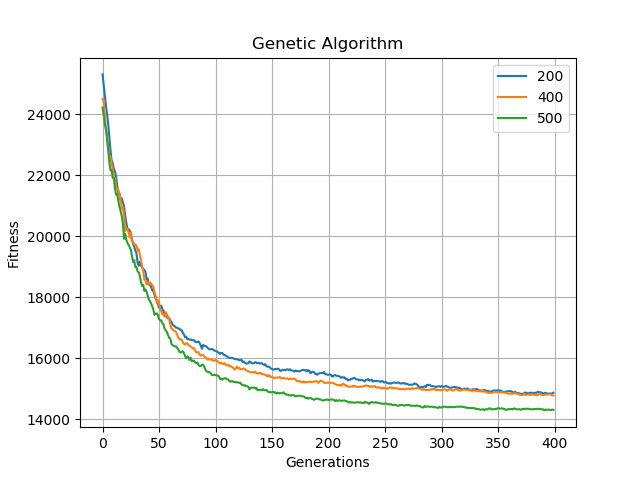


Figure 1 Shows average fitness from 20 runs. Legend is population size.

Even though the smallest population had the over all best individual, the largest population of 300 more individuals performs a lot better on an average of 20 runs.

## How well does genetic algorithm perform compared to exhaustive search on time

Comparisons are performed with one run of GA for both 10 cities and 24 cities to check time. The number of evaluated cities for each population should be:

For 10 cities, 300 generations were used:

**Route length: 10 cities  
Generations: 100**

|  |  |  |  |
| --- | --- | --- | --- |
| **Population size:** | **Evaluations** | **Time(s)** | **Best distance(km)** |
| 200 | 20 000 | 0.38 | 7870 |
| 500 | 50 000 | 1.17 | 7486 |
| 800 | 80 000 | 2.26 | 7846 |

We see that genetic algorithm outperforms exhaustive search in time, even though the algorithm uses more evaluations than necessary. The two graphs below show that for 10 cities, it quickly converges to the global maximum for 10 cities. Time spent is also quite less than for exhaustive search, which uses 3.7s for 10 cities

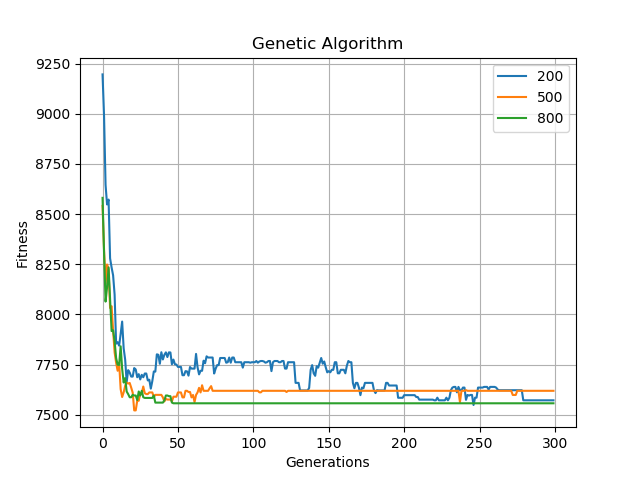
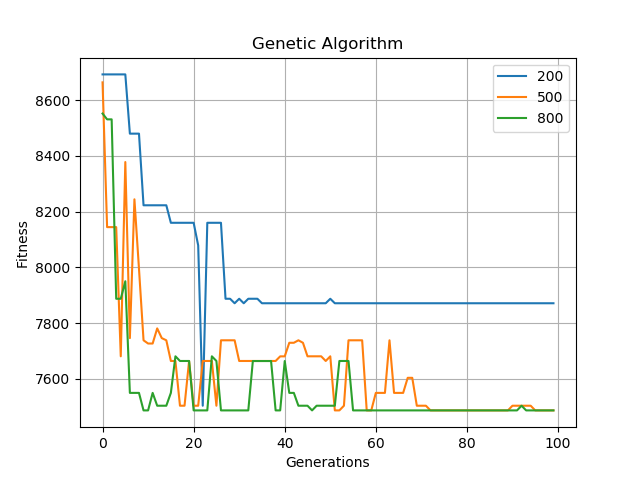


Figure 2. Single run Figure 3. Average of 5 runs

**Route length: 24 cities  
Generations: 200**

|  |  |  |  |
| --- | --- | --- | --- |
| **Population size:** | **Evaluations** | **Time(s)** | **Best distance(km)** |
| 200 | 50 000 | 3.06 | 15204 |
| 700 | 70 000 | 6.08 | 15204 |
| 1200 | 240 000 | 9.97 | 13797 |

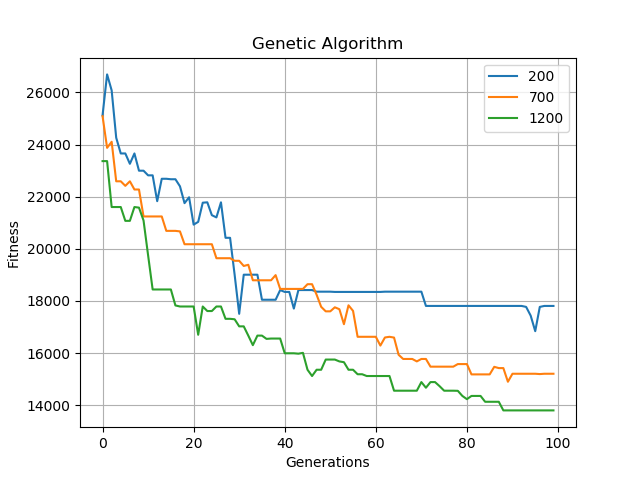


Figure 4 Result from one, timed run

From one run it manages to get decent result, though it is a stochastic method, and results varies. The best result achieved under 14000km in under 10 seconds, which compares to exhaustive is of course a lot less time.

# Hybrid algorithm

This method involves doing a local search on each individual to optimize the population before going to the next generation. For this method, Hill Climber is used as local search. After hill climber is executed, one can use either Lamarckian or Baldwinian learning methods. Lamarckian will keep both fitness and optimized offsprings over to the next generation and usually converges fast towards local or global minimum. Baldwinian will do a local search, but only keep the fitness for the next generation and select offspring as normal.

I have not counted the extra evaluations done for each local search, which perhaps should be a part of the evaluations. In that case, the evaluations would have to be multiplied by 20 for every individual in the population

An assumption has been made that the hill climber method will only change an individual if the individual was improved when a permutation operation is done. I have therefore not written code that checks if the overall fitness was improved after a local search, since hill climber is written so that it should be impossible for it to worsen.

### Results of baldwinian

**Baldwiniand:  
Route length: 24  
Generations: 70  
Local searches: max 20 for each individual**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Population** | **Evaluations** | **Best fitness(km)** | **Worst fitness(km)** | **Mean(km)** | **Standard deviation(km)** |
| 20 | 1000 | 17121 | 22125 | 20004 | 1454 |
| 30 | 1500 | 16821 | 20110 | 18545 | 905 |
| 50 | 2500 | 16306 | 20050 | 18024 | 1082 |

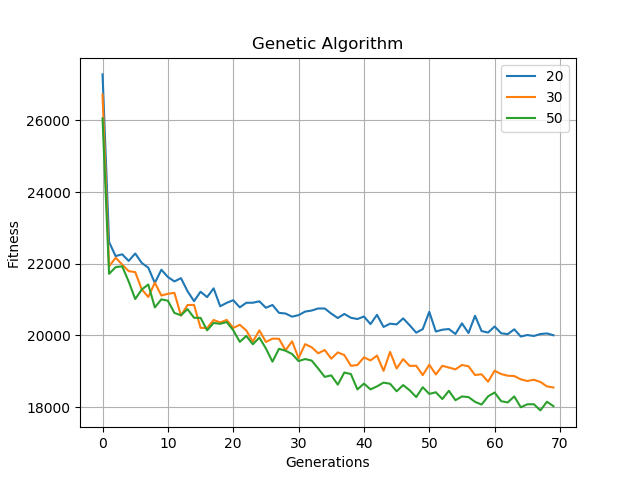


Figure 5 Average of 20 runs - Baldwinian

The first test (figure 6) of baldwinian was performed with very few evaluations and it looks like it performs poorly compared to pure GA. When increasing number of evaluations, see figure 7, performance increases slightly. Figure 8 shows a test with a population size that is the same as one of pure GA’s population sizes. At 100 generations, baldwinian slightly outperforms that of pure GA at the same amount of evaluations.

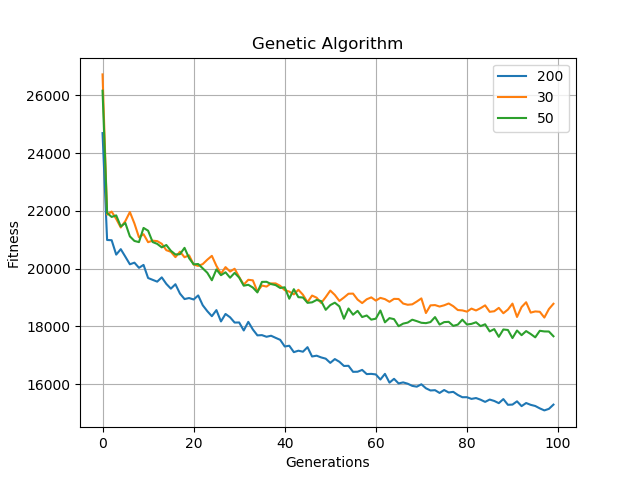
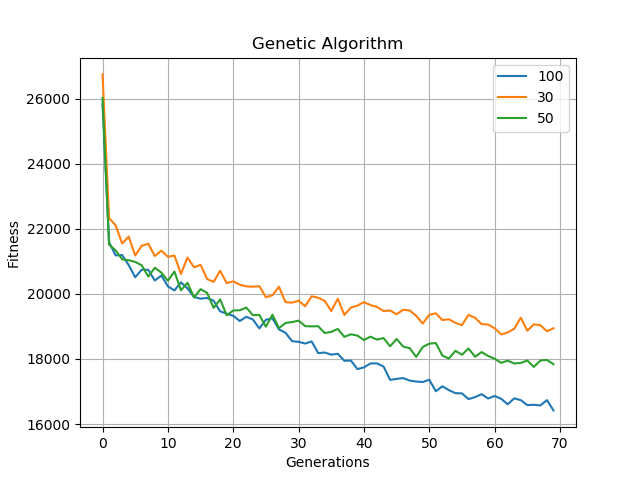
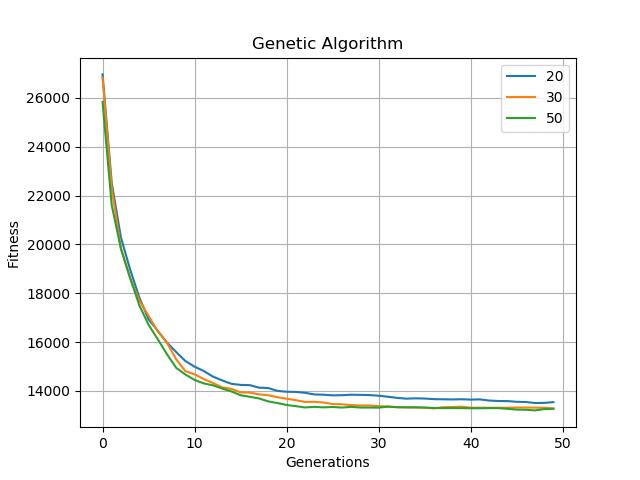


Figure 6 Figure 7

### Results of Lamarckian

**Lamarckian:  
Route length: 24  
Generations: 50  
Local searches: max 20 per individual**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Population** | **Evaluations** | **Best fitness(km)** | **Worst fitness(km)** | **Mean(km)** | **Standard deviation(km)** |
| 20 | 1000 | 12737 | 14259 | 13537 | 442 |
| 30 | 1500 | 12396 | 14315 | 13281 | 563 |
| 50 | 2500 | 12287 | 14221 | 13264 | 534 |



With only a few evaluations compared to GA. Lamarckian outperforms all pervious stochastic methods by a lot. Even the smallest number of evaluations, which was 1000, beats pure GA, whose largest number of evaluations was over 200 000 and it was still beat. Execution time was an issue though. It took over 800 seconds to do all the evaluations.

# Appendix

## Code

In the folder *scripts*:

1. TSP.py
2. Routes.py
3. Search\_algorithms.py
4. European\_cities.csv

## Documentation

In the folder *documentation:*

1. TSP.html
2. Routes.html
3. Search\_algorithms.html

## Figures

In the folder *Figures:*

Various figures from testing. All relevant figures are in the report.