# Genetic Algorithm Implementation for Traveling Salesman Problem

Han Xiao 316101136

December 24, 2018

Project Two of Data Analysis and Algorithm Design

ISEE college, Zhejiang university

# Contents



# List of Figures

# 1 Introduction

In computer science and operations research, genetic algorithm (GA) is a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms (EA). Genetic algorithms are commonly used to generate high-quality solutions to optimization and search problems by relying on bio-inspired operators such as mutation, crossover and selection. John Holland introduced Genetic Algorithm (GA) in 1960 based on the concept of Darwin's theory of evolution; afterwards, his student Goldberg extended GA in 1989.

The Travelling Salesman Problem (often called TSP) is a classic algorithmic problem in the field of computer science and operations research. It is focused on optimization. In this context, better solution often means a solution that is cheaper, shorter, or faster. TSP is a mathematical problem. It is most easily expressed as a graph describing the locations of a set of nodes.

## 1.1 Problem Statement

Given that TSP is a classic algorithm puzzle in the past, there is no need to overstate it here.The focus of this paper is not to solve TSP, but to understand GA correctly and optimize the actual objective function. TSP is only regarded as the background of a practical problem.

The TSP problem involved in this time is relatively simple. The background is how a person can visit the following 31 famous Chinese cities by the shortest path. The starting point and ending point are Nanjing. In order to simplify the problem, only the two-dimensional coordinate distance between two cities is taken as the measurement standard of distance. When the model is really applied, the actual length of roads along the road should be fully considered.

| **City** | **Longitude** | **Latitude** |
|---|---|---|
| Chongqing | 106.54 | 29.59 |
| Lhasa | 91.11 | 29.97 |
| Urumqi | 87.68 | 43.77 |
| Yinchuan | 106.27 | 38.47 |
| Hohhot | 111.65 | 40.82 |
| Nanning | 108.33 | 22.84 |
| Harbin | 126.63 | 45.75 |
| Changchun | 125.35 | 43.88 |
| Shenyang | 123.38 | 41.8 |
| Shijiazhuang | 114.48 | 38.03 |
| Taiyuan | 112.53 | 37.87 |
| Xining | 101.74 | 36.56 |
| Jinan | 117 | 36.65 |
| Zhengzhou | 113.6 | 34.76 |
| Nanjing | 118.78 | 32.04 |
| Hefei | 117.27 | 31.86 |
| Hangzhou | 120.19 | 30.26 |
| Fuzhou | 119.3 | 26.08 |
| Nanchang | 115.89 | 28.68 |
| Changsha | 113 | 28.21 |

| | | |
|---|---|---|
| Wuhan | 114.31 | 30.52 |
| Guangzhou | 113.23 | 23.16 |
| Taipei | 121.5 | 25.05 |
| Haikou | 110.35 | 20.02 |
| Lanzhou | 103.73 | 36.03 |
| Xi'an | 108.95 | 34.27 |
| Chengdu | 104.06 | 30.67 |
| Guiyang | 106.71 | 26.57 |
| Kunming | 102.73 | 25.04 |
| Hong Kong | 114.1 | 22.2 |
| Macau | 113.33 | 22.13 |

## 1.2 The Work I did

In this report, I mainly completed and mentioned the following contents:

1. Introduction, application and research of the GA & TSP;
2. Apply GA to TSP and write Python code to solve it;
3. Use the Basemap tool to draw the given coordinates on a map provided by the national geographic database;
4. Analysis and discussion of the results.

# 2 A brief introduction to the model and algorithm flow of GA

## 2.1 Basic theory

1. Genetic algorithms start with a population representing a potential solution set to a problem, and a population consists of a certain number of individuals encoded by genes.

2. Each individual is actually an entity with a characteristic chromosome.

3. Chromosome, as the main carrier of genetic material, is the collection of multiple genes. Its internal expression (i.e., genotype) is a certain combination of genes, which determines the external expression of an individual's shape. For example, the characteristics of black hair are determined by a certain combination of genes in chromosomes that control this feature.

4. Therefore, at the very beginning, the mapping from phenotype to genotype is required, namely, coding.Due to simulate the work of gene encoding is very complex, we tend to simplify, such as binary coding, original population is generated, according to the principle of survival of the fittest and the evolution, the evolution of generational produce better approximate solution.

5. In each generation, according to the individual problem domain size to choose the fitness of individuals, and by means of natural genetics, genetic operators of crossover and mutation, produced on behalf of the new solution set of the population. This process will lead to the epigenetic population, like the natural evolution, being more adaptable to the environment than the previous generation.

6. The optimal individuals in the last generation can be decoded as the approximate optimal solution of the problem.

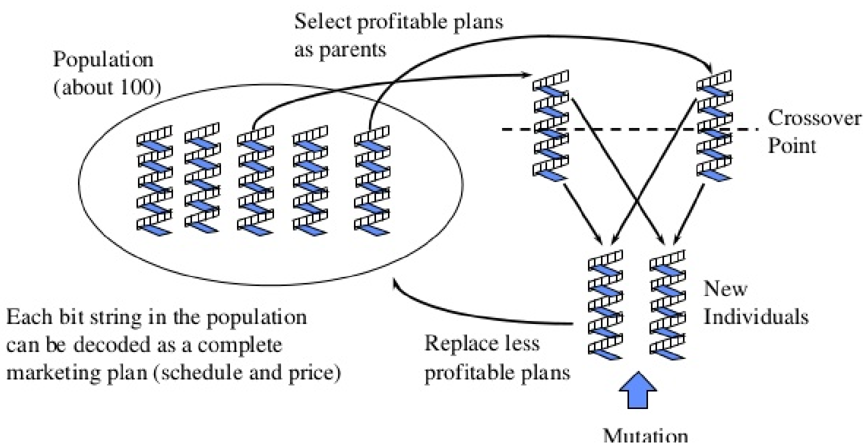


Figure 1: Model of GA

## 2.2 Algorithm flow

1. Initialization: set the evolutionary algebra counter t=0, set the maximum evolutionary algebra t, and randomly generate M individuals as the initial group P(0)

2. Individual evaluation: calculate the fitness of each individual in the population P(t).

3. Selection operation: apply the selection operator to the group.The purpose of selection is to pass the optimized individual directly to the next generation or to pass on the new individual to the next generation through mating crossover.Selection is based on fitness assessment of individuals in a population.

4. Cossover operation: apply crossover operator to the group.The crossover operator plays a key role in genetic algorithm.

5. Mutation operation: apply mutation operator to the population.It is to change the gene value on some loci of the individual string in the population.Population P(t) was selected, crossed and mutated to obtain the next generation population P(t+1).

6. Judgment of termination condition: if t= t, the individual with the maximum fitness obtained in the evolutionary process is taken as the output of the optimal solution, and the calculation is terminated.

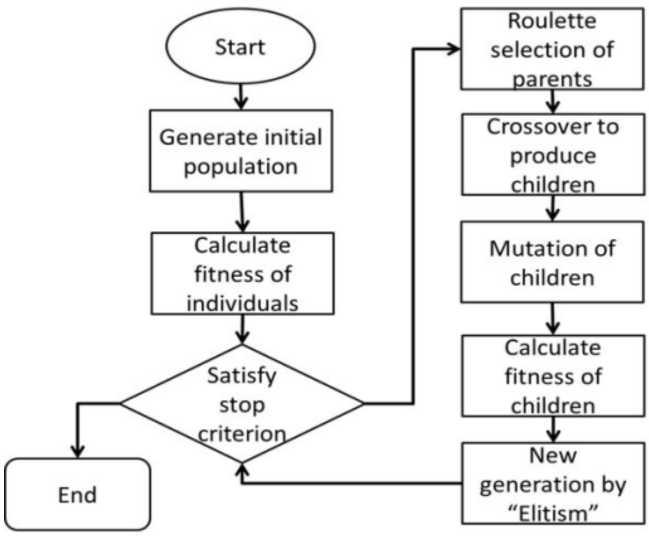


Figure 2: Algorithm flow chart of GA

# 3 Solution to TSP

## 3.1 Algorithm application and optimization objectives

- Unlike the traditional TSP solution, it is not necessary to consider the geometric relationship between points (i.e., 31 cities) in a point set;
- Just consider the cost function for traversing all points in one way (that is, the optimization objective in GA), where the cost can be measured by the sum of the distances between adjacent points in the traversal order.

## 3.2 Analysis of calculation results

Through the program operation, the optimization objective is decreasing and the convergence speed is fast. As can be seen from the figure below, when the number of iterations is about 300, the optimization value gradually tends to be stable, and the subsequent changes are not obvious.

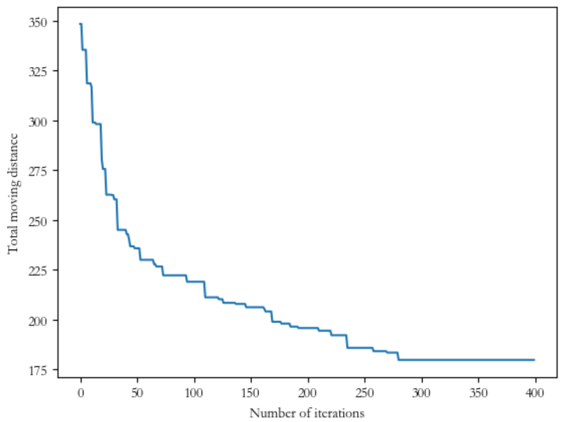


Figure 3: The change curve of the total moving distance vs the number of iterations

Finally, when the number of iterations is about 3000, the travel route of the travelling salesman is shown in the figure below. It can be seen intuitively from the figure that GA solution is correct.It is necessary to mention again that the application of GA here is relatively simple, and the optimization objective is not complicated. Therefore, this example only has universality in thought, not in method.

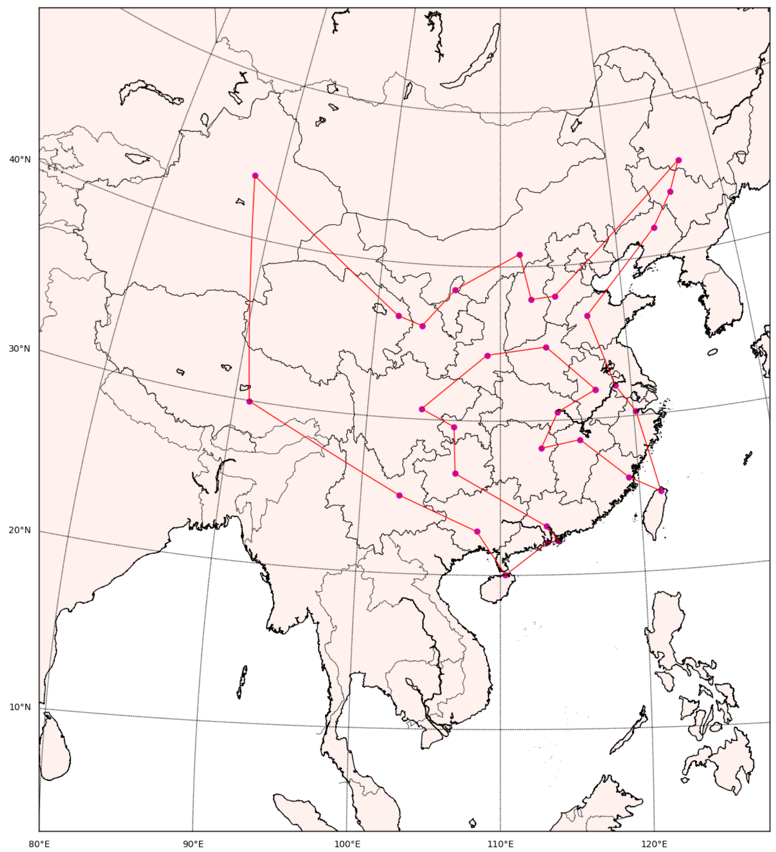


Figure 4: The shortest path drawn on a map

# A Appendix

I provide all my source code in the appendix.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib
import math
import random

matplotlib.rcParams['font.family'] = 'STSong'

# Load the data
city_name = []
city_condition = []
with open('data.txt', 'r') as f:
    lines = f.readlines()
    for line in lines:
        line = line.split('\n')[0]
        line = line.split(',')
        city_name.append(line[0])
        city_condition.append([float(line[1]), float(line[2])])
city_condition = np.array(city_condition)

# Distance matrix
city_count = len(city_name)
Distance = np.zeros([city_count, city_count])
for i in range(city_count):
    for j in range(city_count):
        Distance[i][j] = math.sqrt(
            (city_condition[i][0] - city_condition[j][0]) ** 2 + (
               city_condition[i][1] - city_condition[j][1]) ** 2)

# Population
count = 300
# Number of improvement
improve_count = 10000
# Number of evolution
itter_time = 3000

# Set the definition probability of the strong, that is, the first 30% of the
   population is the strong
retain_rate = 0.3

# Set the survival probability of the weak
random_select_rate = 0.5

# The mutation rate
mutation_rate = 0.1

```

```python
# Set the starting point
origin = 15
index = [i for i in range(city_count)]
index.remove(15)


# The total distance
def get_total_distance(x):
    distance = 0
    distance += Distance[origin][x[0]]
    for i in range(len(x)):
        if i == len(x) - 1:
            distance += Distance[origin][x[i]]
        else:
            distance += Distance[x[i]][x[i + 1]]
    return distance


# Improved
def improve(x):
    i = 0
    distance = get_total_distance(x)
    while i < improve_count:
        # randint [a,b]
        u = random.randint(0, len(x) - 1)
        v = random.randint(0, len(x) - 1)
        if u != v:
            new_x = x.copy()
            t = new_x[u]
            new_x[u] = new_x[v]
            new_x[v] = t
            new_distance = get_total_distance(new_x)
            if new_distance < distance:
                distance = new_distance
                x = new_x.copy()
        else:
            continue
        i += 1


# Natural selection
def selection(population):

    # Sort the total distance from the smallest to the largest
    graded = [[get_total_distance(x), x] for x in population]
    graded = [x[1] for x in sorted(graded)]
    # Pick out the chromosomes that are resilient
    retain_length = int(len(graded) * retain_rate)
    parents = graded[:retain_length]
    # Pick out the chromosomes that are less adaptable, but that survive
    for chromosome in graded[retain_length:]:
```

```python
        if random.random() < random_select_rate:
            parents.append(chromosome)
    return parents


# Cross breeding
def crossover(parents):
    # The number of progeny generated to ensure population stability
    target_count = count - len(parents)
    # The children list
    children = []
    while len(children) < target_count:
        male_index = random.randint(0, len(parents) - 1)
        female_index = random.randint(0, len(parents) - 1)
        if male_index != female_index:
            male = parents[male_index]
            female = parents[female_index]

            left = random.randint(0, len(male) - 2)
            right = random.randint(left + 1, len(male) - 1)

            # Cross section
            gene1 = male[left:right]
            gene2 = female[left:right]

            child1_c = male[right:] + male[:right]
            child2_c = female[right:] + female[:right]
            child1 = child1_c.copy()
            child2 = child2_c.copy()

            for o in gene2:
                child1_c.remove(o)

            for o in gene1:
                child2_c.remove(o)

            child1[left:right] = gene2
            child2[left:right] = gene1

            child1[right:] = child1_c[0:len(child1) - right]
            child1[:left] = child1_c[len(child1) - right:]

            child2[right:] = child2_c[0:len(child1) - right]
            child2[:left] = child2_c[len(child1) - right:]

            children.append(child1)
            children.append(child2)

    return children
```

```python
# Mutation
def mutation(children):
    for i in range(len(children)):
        if random.random() < mutation_rate:
            child = children[i]
            u = random.randint(1, len(child) - 4)
            v = random.randint(u + 1, len(child) - 3)
            w = random.randint(v + 1, len(child) - 2)
            child = children[i]
            child = child[0:u] + child[v:w] + child[u:v] + child[w:]


# Get the best pure output
def get_result(population):
    graded = [[get_total_distance(x), x] for x in population]
    graded = sorted(graded)
    return graded[0][0], graded[0][1]


# The population was initialized using an improved loop algorithm
population = []
for i in range(count):
    # Randomly generated individuals
    x = index.copy()
    random.shuffle(x)
    improve(x)
    population.append(x)

register = []
i = 0
distance, result_path = get_result(population)
while i < itter_time:
    # Select breeding groups of individuals
    parents = selection(population)
    # Cross breeding
    children = crossover(parents)
    # Mutation
    mutation(children)
    # Update the population
    population = parents + children

    distance, result_path = get_result(population)
    register.append(distance)
    i = i + 1

result_path = [origin] + result_path + [origin]
print(distance)
print(result_path)


X = []
```

```
Y = []
for index in result_path:
    X.append(city_condition[index, 0])
    Y.append(city_condition[index, 1])

plt.plot(X, Y, '-o')
plt.show()

plt.xlabel("Number of iterations")
plt.ylabel("Total moving distance")
plt.plot(list(range(len(register))), register)
plt.show()
```

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap

city_name = []
city_condition = []
with open('data.txt', 'r') as f:
    lines = f.readlines()
    for line in lines:
        line = line.split('\n')[0]
        line = line.split(',')
        city_name.append(line[0])
        city_condition.append([float(line[1]), float(line[2])])
city_condition = np.array(city_condition)

result_path = [15, 13, 25, 26, 0, 27, 21, 29, 30, 23, 5, 28, 1, 2, 11,
   24, 3, 4, 10, 9, 6, 7, 8, 12, 14, 16, 22, 17, 18, 19, 20, 15]


X = []
Y = []
for index in result_path:
    X.append(city_condition[index, 0])
    Y.append(city_condition[index, 1])

fig = plt.figure(figsize=(20,16))
ax1 = fig.add_axes([0.1,0.1,0.8,0.8])
map = Basemap(projection='poly',lat_0=35,lon_0=110,llcrnrlon=80,llcrnrlat
   =3.01,urcrnrlon=140,urcrnrlat=53.123,resolution='h',area_thresh=1000,
   rsphere=6371200.,ax = ax1)
map.readshapefile("./template/bou2_4p","china",drawbounds=True)
map.drawcoastlines()
map.drawcountries()
map.fillcontinents(color = 'coral',alpha = .1)
map.drawmapboundary()
map.drawparallels(np.arange(0.,90,10.),labels=[1,0,0,0],fontsize=10)
map.drawmeridians(np.arange(80.,140.,10.),labels=[0,0,0,1],fontsize=10)
```

```python


for i in list(range(0,31)):
    if i == 31:
        start_lon = X[31]
        start_lat = Y[31]
        end_lon = X[0]
        end_lat = Y[0]
    else:
        start_lon = X[i]
        start_lat = Y[i]
        end_lon = X[i+1]
        end_lat = Y[i+1]
    if abs(end_lat - start_lat) < 180 and abs(end_lon - start_lon) < 180:
        map.drawgreatcircle(start_lon, start_lat, end_lon, end_lat,
           linewidth=1, color = "red")

x, y = map(X, Y)
map.scatter(x, y, marker='o', color='m')
plt.savefig('./Result.png')
plt.show()
```