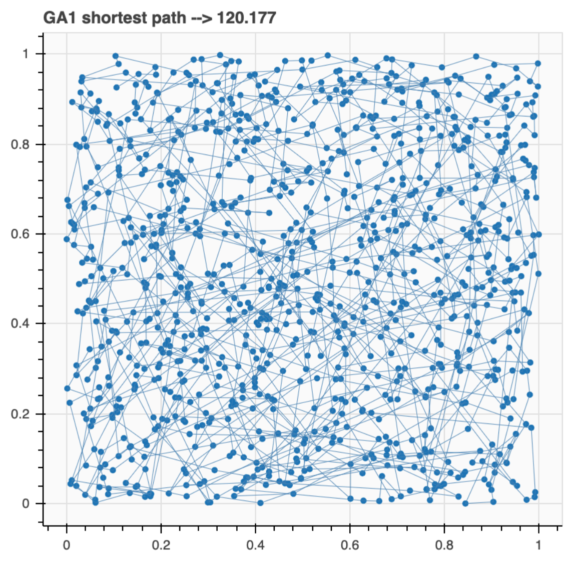
Assignment 1

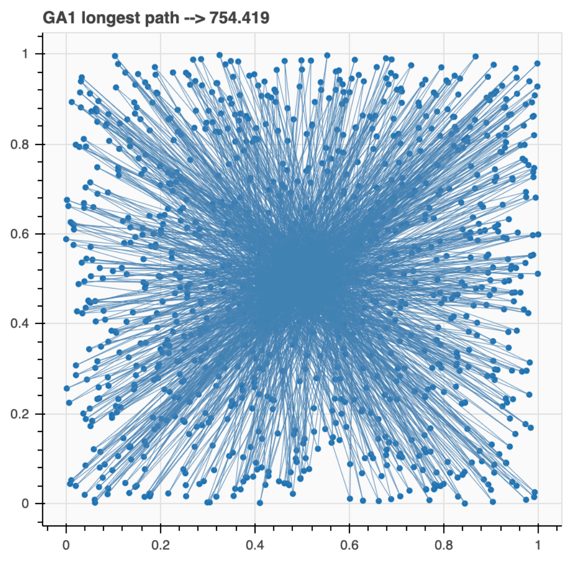
Name: Chiqu Li
UNI: cl3895
Course: MECS E4510 EVOLUTIONARY COMPUTATION&DESIGN AUTOMATION
Professor: Hod Lipson
Date submitted: 9/29/2019
Grace hours used: 0
Grace hours remaining: 116h

Results summary

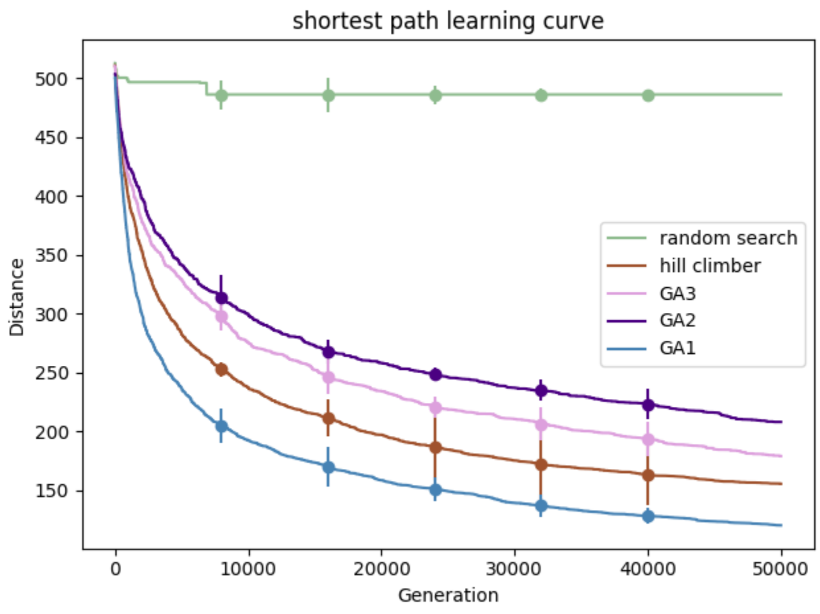
Table 1: Results summary

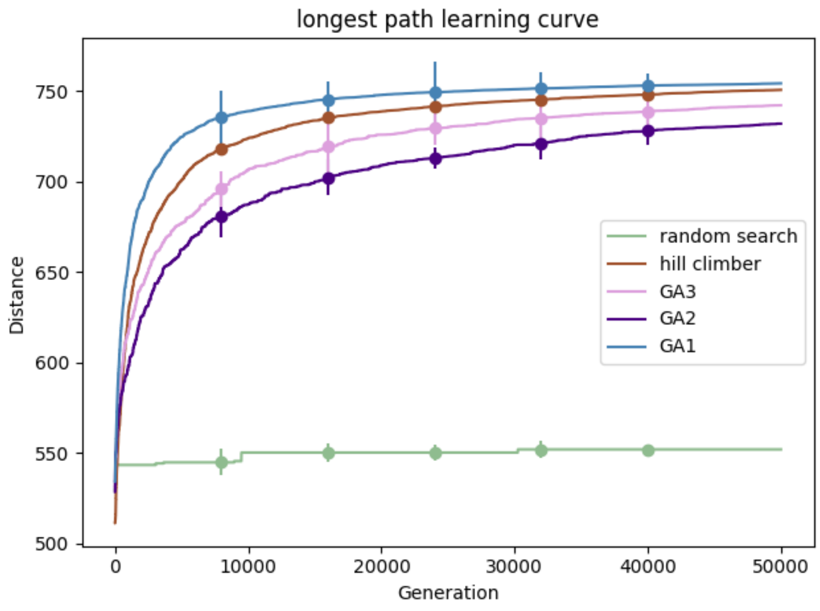
Best solution		Evaluations	Length
GA1	The shortest path	50000	120.177
	The longest path	50000	754.419

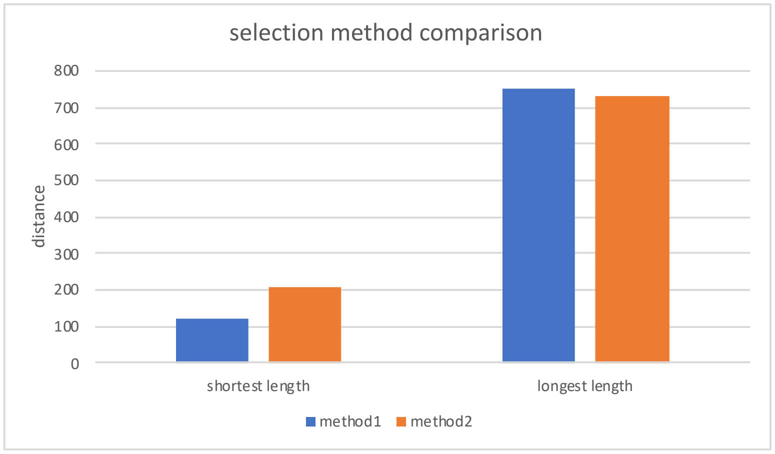




GA1: GA with Roulette algorithm
GA2: GA with random selection
GA3: A variation of GA1 with lower crossover rate


shortest path learning curve
Distance
Generation
random search
hill climber
GA3
GA2
GA1


longest path learning curve
Distance
Generation
random search
hill climber
GA3
GA2
GA1


selection method comparison
distance
shortest length
longest length
method1
method2

Methods

Representation: Path representation
I used the path representation, which is a basic way to represent the route. That means a gene randomly contain a list number from 0 to 999, which representing the order of 1000 cities. The order of numbers is the gene.

EA variation operators: Crossover rate
I set the crossover rate and mutation rate, which means in a group of genes, only a certain number of genes can crossover with others, so as mutation. I changed the crossover rate from 0.6 to 0.4, at the meantime, mutation rate maintained 0.1, and then I compared the fitness of two methods. It turned out that after 50000 generations, the crossover rate of 0.6 performed better.

Selection process: Two selection methods. Method 1 is to select parents randomly from the group. Method 2 is to select parents by the Roulette Algorithm. When the fitness is too low (lower than a fixed value 15), then I dropped this gene and select another one until the fitness is acceptable.

Description of random search
Random search is to mess the order of cities randomly, and then compare the total distance to the best distance. If the new distance is better, then we set the new distance as the best distance.

Description of hill climber
Hill climber is to choose a better solution based on the last try. I randomly switch two cities, if the distance gets better, then I kept the new array of cities, or else I switched back. It turned out that Hill climber method is a very efficient way to solve the TSP problem.

Analysis of performance
The selection and variation of the GA impact the consequence significantly. As we can see from the path, after 50000 generations, GA1 is the best solution with a shortest distance of 120. The selection method of GA1 is method 2, which gave the next generation a better gene. This is why we GA1 is better than GA2. However, when I decreased the crossover rate from 0.6 to 0.4(GA1 to GA3), the fitness decreased as well. It is mainly because the speed of evolution is slow down because of less crossover.

In terms of random search and hill climber, we can find that the hill climber is a very practical method, which gets very close to GA1. And the speed of hill climber is very fast because of its simplicity. However, the random search is inefficient.

Appendix

#Random search code

```
import numpy as np

a = np.loadtxt('tsp.txt')
min_a = np.array([])

def cal1():
    dismin1 = np.array([])
    min_a = np.array([])
    for looptimes in range(times):
        np.random.shuffle(a) ##shuffle a randomly
        distance = cal_distance(a, looptimes)
        if looptimes == 0:
            dismin = distance
        if distance > dismin:
            dismin = distance
            min_a = a
        print('looptimes',looptimes)
        print('dismin', dismin)
        if looptimes % 1 == 0:
            dismin1 = np.append(dismin1, np.array([dismin]))
    return (dismin1, dismin, min_a)

def cal_distance(a):
    distance = 0
    for j in range(1, len(a)):
        distance += np.linalg.norm(a[j] - a[j - 1])
    return distance

dismin1, min1, min_a1 = cal1()
```

#Hill climber code

```
import numpy as np

a = np.loadtxt('tsp.txt')
times = 50000

idx1 = np.random.randint(sizedata, size=1)
idx2 = np.random.randint(sizedata, size=1)

min_a = np.array([])

def cal_distance(a):
    distance = 0
    for j in range(1, len(a)):
        distance += np.linalg.norm(a[j] - a[j - 1])
    distance += np.linalg.norm(a[len(a)] - a[0])
    return distance

distance_original = cal_distance(a)

def cal1(distance_original):
    dismin1 = np.array([])
    for looptimes in range(times):
        idx1 = np.random.randint(sizedata, size=1)
        idx2 = np.random.randint(sizedata, size=1)

        a[[idx1, idx2], :] = a[[idx2, idx1], :]
        distance1 = cal_distance(a)

        if distance1 < distance_original:
            a[[idx1, idx2], :] = a[[idx2, idx1], :]
        else:
            distance_original = distance1

        print(distance_original, looptimes)
        dismin1 = np.append(dismin1, np.array([distance_original]))
        if looptimes in range(300, 500):

    return distance_original, a, dismin1

distance1, a1, dismin1 = cal1(distance_original)
```

#GA Code

```
import numpy as np
a = np.loadtxt('tsp.txt')

geneLength = len(a)
lifeCount = 20 #a group of 30
lives = np.zeros(shape = (lifeCount, geneLength))
crossRate = 0.6 #crossRate change to 0.4 in GA2
mutationRate = 0.1
generation = 0

def initPopulation(geneLength, lives):
    gene = np.array([x for x in range(geneLength)])
    for i in range(lifeCount):
        np.random.shuffle(gene)
        lives[i] = gene
    return lives



def judge(lives):
    bounds = 0.0
    best = lives[0]
    bestscore = matchFun(a, best)

    for i in range(len(lives)):
        score = matchFun(a, lives[i])
        bounds = bounds + score
        if score > bestscore:
            best = lives[i]
            bestscore = score

    return best

def matchFun(a, gene):
    return (1.0 / distance(a, gene))

def distance(a, gene):
    distance = 0
    for j in range(-1, len(a)-1):
        index1, index2 = int(gene[j]), int(gene[j+1])
        city1, city2 = a[index1], a[index2]
        distance += np.linalg.norm(city1 - city2)
    return distance

def cross(parent1, parent2):
    crossCount = 0
    index1 = np.random.randint(0, geneLength - 1)
    index2 = np.random.randint(index1, geneLength - 1)
    tempGene = parent2[index1:index2]  # gene piece
    newGene = []
    p1len = 0
    for g in parent1:
        if p1len == index1:
            newGene = np.append(newGene, tempGene)  #insert gene piece
            p1len += 1
        if g not in tempGene:
            newGene = np.append(newGene, g)
            p1len += 1
    crossCount += 1
    return newGene

def mutation(gene):
    index1 = np.random.randint(0, geneLength - 1)
    index2 = np.random.randint(0, geneLength - 1)
    newGene = gene
    newGene[index1], newGene[index2] = newGene[index2], newGene[index1]
    return newGene

def select1(lives, bound):
    flag=np.random.rand()
    flag = flag * bound
    for life in lives:
        flag = flag - matchFun(life)
        if flag <= 25:
            return life
#select1 uses the Roulette Algorithm

def select2(lives):
    flag = np.random.rand()
    life = lives[int(flag * lifeCount)]
    return life
#select1 choose life randomly

def newChild(lives):
    parent1 = select1(lives)
    rate = np.random.rand(1)

    if rate < crossRate:
        # crossover #
        parent2 = select1(lives)
        genee = cross(parent1, parent2)
    else:
        genee = parent1

    rate = np.random.rand(1)
    if rate < mutationRate:
        genee = mutation(genee)

    return genee

def next(lives, generation):

    best = judge(lives)
    newLives = np.zeros(shape = (lifeCount, geneLength))
    newLives[0] = best

    for i in range(lifeCount ):
        newLives[i] =newChild(lives)

    bestgene = judge(newLives)

    return newLives, bestgene, generation

def run(n, lives1, generation1):
    distance_list = np.array([])
    generate = np.array([index for index in range(1, n + 1)])
    lives, bestgene, generation = next(lives1, generation1)
    while n>0:
        lives, bestgene, generation = next(lives, generation1)
        distance1 = distance(a, bestgene)
        distance_list = np.append(distance_list, distance1)

def main():
    lives = np.zeros(shape=(lifeCount, geneLength))
    lives = initPopulation(geneLength, lives)
    run(1000, lives, generation)
main()
```