20190102 Mintae Kim – IE260 HW4 Report

1. Initial population (Related function: createInitialPopulation(), performEvolution())

In the original version of code, we create randomly permutated genotype in the given number of times (numPopulation) and regard it as our initial population. However, during research, I found that k-means clustering performs better because it provides population with genotypes have less total distance. Then, the idea is simple, which is providing better genotypes than randomly selected out of N! (N: number of cities) cases. Hence, I utilized the fitness function to rank each genotype which is randomly created. Create 10 times numPopulation number of genotypes and rank it using the fitness function. Then, return the top numPopulation genotypes as an initial population. This method is simple but strong because we can prevent inferior genotypes which are way too far from the optimal solution.

2. Mutation (Related functions: calculateMutationFactor(), mutation(), performEvolution())

In the original version of code, we use a fixed mutation factor, 0.2. However, we need more mutation if the output distance is too large. On the other hand, we need less mutation if the output is sufficiently small. Therefore, I used calculateMutationFactor() function in order to make mutation factor more flexible. First get the total distance(standardTotalDist) of the best solution is prior iteration of evolution. If the best solution of current iteration(totalDist) is larger than 0.9 times standardTotalDist, which means that there was not enough improvement during one iteration, we mutate more. Otherwise, if the improvement is satisfactory, we loosen mutation with less mutation factor. The function that calculates mutation factor is as follows.

I found the function in the last line from this paper: Chunhua Fu, Lijun Zhang, Xiaojing Wang, and Liying Qiao, Solving TSP problem with improved genetic algorithm AIP Conference Proceedings 1967, 040057 (2018); https://doi.org/10.1063/1.5039131, Published Online: 23 May 2018

```
def calculateMutationFactor(self, population, offspring,
totalDist, standardTotalDist):
    if totalDist > standardTotalDist * 0.9:
        maxMutationFactor = 0.3
        minMutationFactor = 0.2
    elif totalDist <= standardTotalDist * 0.9 and totalDist >
standardTotalDist * 0.7:
        maxMutationFactor = 0.2
        minMutationFactor = 0.15
    else:
        maxMutationFactor = 0.15
        minMutationFactor = 0.08
```

```
    avgFitness = 0

    for genoType in population:
        avgFitness += self.fitness(genoType)
    avgFitness = avgFitness / float(len(population))
    if self.fitness(offspring) >= avgFitness:
        mutationFactor = maxMutationFactor * (maxMutationFactor -
minMutationFactor) * (self.fitness(offspring) - avgFitness) /
(maxMutationFactor - avgFitness)
    else:
        mutationFactor = maxMutationFactor
    return mutationFactor
```

I have also tried another version of mutation factor calculation as follows, but it did not work well.

```
# if totalDist > 2800:
#     maxMutationFactor = 0.3
#     minMutationFactor = 0.2
# elif totalDist <= 2800 and totalDist > standardTotalDist * 2500:
#     maxMutationFactor = 0.2
#     minMutationFactor = 0.15
# else:
#     maxMutationFactor = 0.15
#     minMutationFactor = 0.08
```

Another modification I tried in mutation procedure is to mutate cities with the longest distance. During the evolution, I found that the cities with the longest distance from each other are one of the main causes of inferior genotype. Therefore, I inserted another mutation procedure designed to mutate cities with the longest distance. In the iteration, we mutate factor number of times and before the original swapping, I inserted new swapping for the max distance cities. The actual code is as follows.

```
# mutate cities with the longest distance
currentCity = 0
maxDistance = 0.0
maxCities = {"MaxIdx1": 0, "MaxIdx2": 1}
for itr in range(len(genotype) - 1):
    nextCity = genotype[currentCity]
    current = self.dicLocations[currentCity]
    next = self.dicLocations[nextCity]
    if maxDistance < self.calculateDistance(current, next):
        maxDistance = self.calculateDistance(current, next)
        maxCities["MaxIdx1"] = currentCity
        maxCities["MaxIdx2"] = nextCity
    currentCity = nextCity
idxSwapMax1, idxSwapMax2 = maxCities.values()
idxSwap = random.randint(0, len(genotype))
genotype[idxSwapMax1], genotype[idxSwap] = genotype[idxSwap],
genotype[idxSwapMax1]
```

3. Crossover (Related functions: evaluateInheritance(), performEvolution())

Since we are using position-based encoding, there are not many things we can modify. Even we use k-ary encoding and so on, it would require a lot of computation time so it hinders evolution. When we use position-based encoding, the offspring defers from its parents too much after a crossover. Therefore, I created a function which evaluates whether the offspring is well inherited or not. In performEvolution, we check the inheritance rate using evaluateInheritance function below. The actual code is as follows.

```
for itr in range(numOffsprings):
    gp1, gp2 = self.selectParents(population)
    p1 = self.crossoverParents(gp1, gp2)
    p2 = self.crossoverParents(gp1, gp2)
    wellInherited = False
    while wellInherited == False:
        offspring = self.crossoverParents(p1, p2)
        wellInherited = self.evaluateInheritance(p1.getGenotype(),
p2.getGenotype(), offspring.getGenotype())
    offsprings[itr] = offspring
    mutationFactor = self.calculateMutationFactor(population,
offsprings[itr], self.calculateTotalDistance(offsprings[itr]),
standardTotalDist)
    factor = int(mutationFactor * len(self.dicLocations.keys()))
    self.mutation(offsprings[itr], factor)
```

```
def evaluateInheritance(self, genotypeP1, genotypeP2,
genotypeOff):
    match1 = 0
    match2 = 0
    for itr in range(len(genotypeP1)):
        if genotypeP1[itr] == genotypeOff[itr]:
            match1 += 1
        if genotypeP2[itr] == genotypeOff[itr]:
            match2 += 1
    if float(match1)/float(len(genotypeP1)) > 0.1 and
float(match2)/float(len(genotypeP1)) > 0.1:
        return True
    else:
        return False
```

We check whether the ratio of same neighbor between a parent and an offspring is larger than 0.1 (e.g. If the length of a genotype is 15, we check whether offspring has more than 2 same neighbors as its parents). If this evaluation we can avoid totally irrelevant offspring. Surprisingly, the acceptance rate of evaluateInheritance() is approximately 5%. This implies there were so many irrelevant offspring after crossover.

Reference

Chunhua Fu, Lijun Zhang, Xiaojing Wang, and Liying Qiao, Solving TSP problem with improved genetic algorithm AIP Conference Proceedings 1967, 040057 (2018); https://doi.org/10.1063/1.5039131, Published Online: 23 May 2018

Full code

```
from GeneticAlgorithmProblem import *
import random
import math
import time
import csv

class TravelingSalesmanProblem(GeneticAlgorithmProblem):

    genes = []
    dicLocations = {}
    gui = ''
    best = ''
    time = 0

    def __init__(self, data_mode, csvfile, numCities, height, width, time):
        self.time = time
        if data_mode == 'Random':
            for itr in range(numCities):
                x = random.uniform(0, width)
                y = random.uniform(0, height)
                coordinate = [x, y]
                self.dicLocations[itr] = coordinate
        elif data_mode == 'Load':
            with open(csvfile, 'r') as my_csv:
                contents = list(csv.reader(my_csv, delimiter=","))
                for itr in range(len(contents)):
                    x , y= contents[itr][0],contents[itr][1]
                    self.dicLocations[itr] = [float(x),float(y)]
    def registerGUI(self, gui):
        self.gui = gui

    def performEvolution(self, numIterations, numOffsprings, numPopulation,
mutationFactor):
        if self.gui != '':
            self.gui.start()

        startTime = time.time()
        population = self.createInitialPopulation(numPopulation,
len(self.dicLocations.keys()))
        standardTotalDist = 0

        while True:
            currentTime = time.time()
            if (currentTime - startTime) >= self.time:
                break
            offsprings = {}
            mostFittest = self.findBestSolution(population)
            fistTotalDistance = 0
            for itr in range(numOffsprings):
```

```
                p1, p2 = self.selectParents(population)
                wellInherited = False
                while wellInherited == False:
                    offspring = self.crossoverParents(p1, p2)
                    wellInherited =
self.evaluateInheritance(p1.getGenotype(), p2.getGenotype(),
offspring.getGenotype())
                offsprings[itr] = offspring
                mutationFactor = self.calculateMutationFactor(population,
offsprings[itr], self.calculateTotalDistance(offsprings[itr]),
standardTotalDist)
                factor = int(mutationFactor * len(self.dicLocations.keys()))
                self.mutation(offsprings[itr], factor)
            population = self.substitutePopulation(population, offsprings)
            mostFittest = self.findBestSolution(population)
            self.best = mostFittest

            print(self.calculateTotalDistance(self.best))
            standardTotalDistance = self.calculateTotalDistance(self.best)

            if self.gui != '':
                self.gui.update()

        endTime = time.time()
        return self.best.getGenotype(), self.fitness(self.best),
self.calculateTotalDistance(self.best), (endTime - startTime)

    def evaluateInheritance(self, genotypeP1, genotypeP2, genotypeOff):
        match1 = 0
        match2 = 0
        for itr in range(len(genotypeP1)):
            if genotypeP1[itr] == genotypeOff[itr]:
                match1 += 1
            if genotypeP2[itr] == genotypeOff[itr]:
                match2 += 1
        if float(match1)/float(len(genotypeP1)) > 0.1 and
float(match2)/float(len(genotypeP1)) > 0.1:
            return True
        else:
            return False

    def calculateMutationFactor(self, population, offspring, totalDist,
standardTotalDist):
        if totalDist > standardTotalDist * 0.9:
            maxMutationFactor = 0.3
            minMutationFactor = 0.2
        elif totalDist <= standardTotalDist * 0.9 and totalDist >
standardTotalDist * 0.7:
            maxMutationFactor = 0.2
            minMutationFactor = 0.15
        else:
            maxMutationFactor = 0.15
            minMutationFactor = 0.08

        avgFitness = 0

        for genoType in population:
            avgFitness += self.fitness(genoType)
        avgFitness = avgFitness / float(len(population))
        if self.fitness(offspring) >= avgFitness:
            mutationFactor = maxMutationFactor * (maxMutationFactor -
```

```
minMutationFactor) * (self.fitness(offspring) - avgFitness) /
(maxMutationFactor - avgFitness)
        else:
            mutationFactor = maxMutationFactor
        return mutationFactor

    def fitness(self, instance):
        genotype = instance.getGenotype()
        currentCity = 0
        distance = 0.0
        for itr in range(len(genotype)-1):
            nextCity = genotype[currentCity]
            distance = distance +
self.calculateDistance(self.dicLocations[currentCity],
self.dicLocations[nextCity])
            currentCity = nextCity
        utility = 10000.0 / distance
        return utility

    def calculateTotalDistance(self, instance):
        genotype = instance.getGenotype()
        currentCity = 0
        distance = 0.0
        for itr in range(len(genotype)-1):
            nextCity = genotype[currentCity]
            current = self.dicLocations[currentCity]
            next = self.dicLocations[nextCity]
            distance = distance + self.calculateDistance(current, next)
            currentCity = nextCity
        return distance

    def calculateDistance(self, coordinate1, coordinate2):
        distance = math.sqrt(math.pow(coordinate1[0]-coordinate2[0], 2) +
math.pow(coordinate1[1]-coordinate2[1], 2) )
        return distance

    def getPotentialGenes(self):
        return self.dicLocations.keys()

    def createInitialPopulation(self, numPopulation, numCities):
        population = []
        for itr in range(numPopulation * 10):
            genotype = list(range(numCities))
            while self.isInfeasible(genotype) == False:
                random.shuffle(genotype)
            instance = GeneticAlgorithmInstance()
            instance.setGenotype(genotype)
            population.append(instance)

        rankFitness = {}
        originalFitness = {}
        maxUtility = -999999
        minUtility = 999999
        for itr in range(len(population)):
            originalFitness[itr] = self.fitness(population[itr])
            if maxUtility < originalFitness[itr]:
                maxUtility = originalFitness[itr]
            if minUtility > originalFitness[itr]:
                minUtility = originalFitness[itr]

        for itr1 in range(len(population)):
```

```
            for itr2 in range(itr1+1,len(population)):
                if originalFitness[itr1] < originalFitness[itr2]:
                    originalFitness[itr1], originalFitness[itr2] =
originalFitness[itr2], originalFitness[itr1]
                    population[itr1], population[itr2] = population[itr2],
population[itr1]
        return population[:numPopulation]

    def isInfeasible(self, genotype):
        currentCity = 0
        visitedCity = {}
        for itr in range(len(genotype)):
            visitedCity[currentCity] = 1
            currentCity = genotype[currentCity]

        if len(visitedCity.keys()) == len(genotype):
            return True
        else:
            return False

    def findBestSolution(self, population):
        idxMaximum = -1
        max = -99999
        for itr in range(len(population)):
            if max < self.fitness(population[itr]):
                max = self.fitness(population[itr])
                idxMaximum = itr
        return population[idxMaximum]

    def findWorstSolution(self, population):
        idxMinimum = -1
        min = 99999
        for itr in range(len(population)):
            if min > self.fitness(population[itr]):
                min = self.fitness(population[itr])
                idxMinimum = itr
        return population[idxMinimum]

    def selectParents(self, population):
        rankFitness = {}
        originalFitness = {}
        maxUtility = -999999
        minUtility = 999999
        for itr in range(len(population)):
            originalFitness[itr] = self.fitness(population[itr])
            if maxUtility < originalFitness[itr]:
                maxUtility = originalFitness[itr]
            if minUtility > originalFitness[itr]:
                minUtility = originalFitness[itr]
        for itr1 in range(len(population)):
            for itr2 in range(itr1+1,len(population)):
                if originalFitness[itr1] < originalFitness[itr2]:
                    originalFitness[itr1], originalFitness[itr2] =
originalFitness[itr2], originalFitness[itr1]
                    population[itr1], population[itr2] = population[itr2],
population[itr1]
        size = float(len(population))
        total = 0.0
        for itr in range(len(population)):
            rankFitness[itr] = ( maxUtility + (float(itr) - 1.0)*
(maxUtility - minUtility)) / ( size - 1 )
```

```
            total = total + rankFitness[itr]

        idx1 = -1
        idx2 = -1
        while idx1 == idx2:
            dart = random.uniform(0, total)
            sum = 0.0
            for itr in range(len(population)):
                sum = sum + rankFitness[itr]
                if dart <= sum:
                    idx1 = itr
                    break
            dart = random.uniform(0, total)
            sum = 0.0
            for itr in range(len(population)):
                sum = sum + rankFitness[itr]
                if dart <= sum:
                    idx2 = itr
                    break
        return population[idx1], population[idx2]

    def crossoverParents(self, instance1, instance2):

        genotype1 = instance1.getGenotype()
        genotype2 = instance2.getGenotype()
        newInstance = GeneticAlgorithmInstance()
        dicNeighbor = {}

        for itr in range(len(genotype1)):
            neighbor = {}
            neighbor1 = self.getNeighborCity(instance1, itr)
            neighbor2 = self.getNeighborCity(instance2, itr)
            neighbor[neighbor1[0]] = 1
            neighbor[neighbor1[1]] = 1
            neighbor[neighbor2[0]] = 1
            neighbor[neighbor2[1]] = 1
            dicNeighbor[itr] = neighbor.keys()

        currentCity = 0
        visitedCity = {}
        path = {}
        for itr in range(len(genotype1)):
            visitedCity[currentCity] = 1
            nextCity =
self.getMinimumNeighborNotVisitedCity(list(visitedCity.keys()), dicNeighbor)
            if nextCity == -1:
                nextCity = 0
            path[currentCity] = nextCity
            currentCity = nextCity

        newInstance.setGenotype(path)

        return newInstance

    def getMinimumNeighborNotVisitedCity(self, lstVisitedCity, dicNeighbor):
        cities = list(dicNeighbor.keys())
        for itr in range(len(lstVisitedCity)):
            cities.remove(lstVisitedCity[itr])
        minimum = 999
        candidates = []
        for itr in range(len(cities)):
```

```
            location = cities[itr]
            if len(dicNeighbor[location]) <= minimum:
                minimum = len(dicNeighbor[location])
                candidates.append(location)
        random.shuffle(candidates)
        if len(candidates) == 0:
            return -1
        return candidates[0]

    def getNeighborCity(self, instance, currentCity):

        genotype = instance.getGenotype()
        ret1 = -1
        ret2 = -1
        for itr in range(len(genotype)):
            if genotype[itr] == currentCity:
                ret1 = itr
                break
        ret2 = genotype[currentCity]
        neighbor = [ret1, ret2]
        return neighbor

    def mutation(self, instance, factor):

        genotype = instance.getGenotype()
        mutationDone = False
        while mutationDone == True:
            for itr in range(factor):
                # mutate cities with the longest distance
                currentCity = 0
                maxDistance = 0.0
                maxCities = {"MaxIdx1": 0, "MaxIdx2": 1}
                for itr in range(len(genotype) - 1):
                    nextCity = genotype[currentCity]
                    current = self.dicLocations[currentCity]
                    next = self.dicLocations[nextCity]
                    if maxDistance < self.calculateDistance(current, next):
                        maxDistance = self.calculateDistance(current, next)
                        maxCities["MaxIdx1"] = currentCity
                        maxCities["MaxIdx2"] = nextCity
                    currentCity = nextCity
                idxSwapMax1, idxSwapMax2 = maxCities.values()
                idxSwap = random.randint(0, len(genotype))
                genotype[idxSwapMax1], genotype[idxSwap] =
genotype[idxSwap], genotype[idxSwapMax1]

                # normal mutation
                idxSwap1 = random.randint(0, len(genotype))
                idxSwap2 = random.randint(0, len(genotype))
                genotype[idxSwap1], genotype[idxSwap2] = genotype[idxSwap2],
genotype[idxSwap1]
            if self.isInfeasible(genotype) == True:
                mutationDone = False
            else:
                mutationDone = True
        instance.setGenotype(genotype)

    def substitutePopulation(self, population, children):
        for itr1 in range(len(population)):
            for itr2 in range(itr1+1,len(population)):
                if self.fitness(population[itr1]) <
```

```
self.fitness(population[itr2]):
                    population[itr1], population[itr2] = population[itr2],
population[itr1]
        for itr in range(len(children)):
            population[len(population)-len(children)+itr] = children[itr]
        return population
```