KAIST [CS454] AI-Based Software Engineering

20160810 Jae Sung Park

October 5th, 2020

[Assignment 2] Solving the Traveling Salesman Problem Using Genetic Algorithm

Table of Contents:

1. Introduction

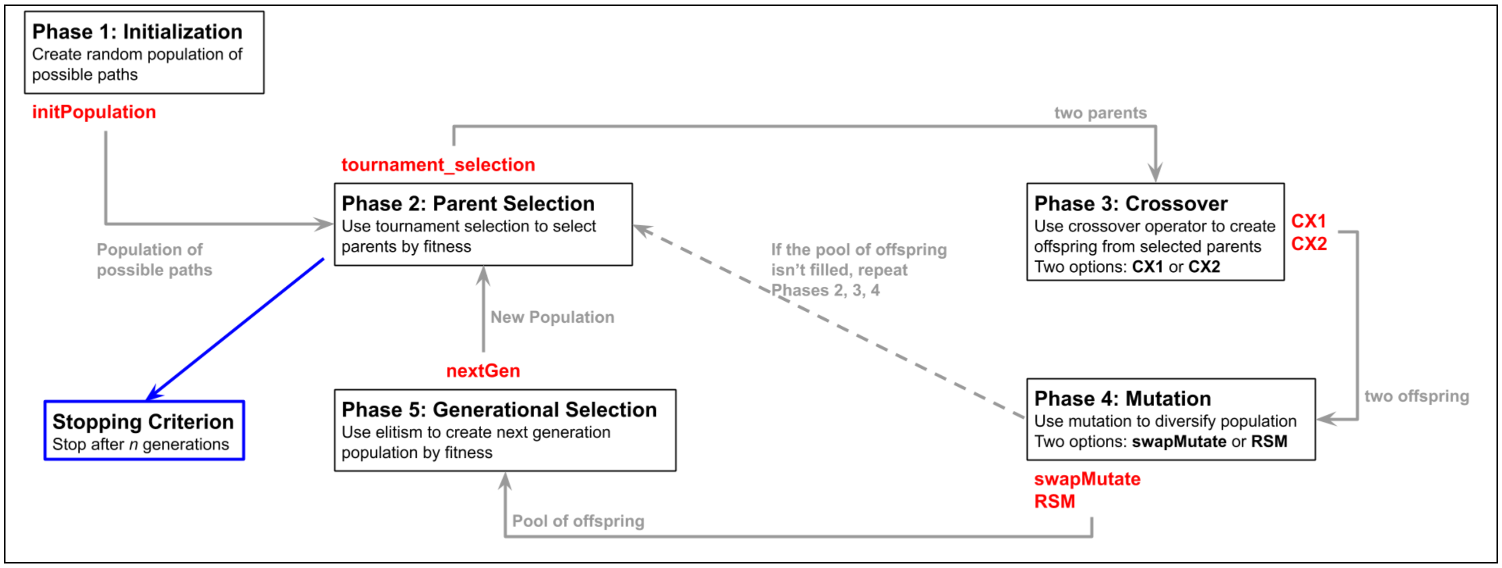
The task is to solve the TSP (specifically the rl11849.tsp instance) using a stochastic optimization method. This report outlines the genetic algorithm (GA) I've implemented to attempt to solve the TSP.

1.1 Genetic Algorithm

The idea of a GA is to randomly generate an initial population, produce offspring that inherit the "genes" of selected parents through crossover operations, cause random mutations in the offspring, and repeat. The diversity of the population in each generation allows a wide variety of solutions to be explored and the best is then selected. In the case of TSP, the objective is to select solutions with shorter paths. The GA is organized into five phases:

Phase 1: Initialization
Phase 2: Parent Selection
Phase 3: Crossover Operation
Phase 4: Mutation
Phase 5: Generational Selection

Figure 1 depicts an organization of the GA by phases. The GA goes through Phase 1 only once and the rest are repeated, until a stopping criterion is met. The specifics about each phase and the associated functions are detailed in Section 2.



Phase 1: Initialization
Create random population of possible paths
initPopulation
Population of possible paths
tournament_selection
two parents
Phase 2: Parent Selection
Use tournament selection to select parents by fitness
Phase 3: Crossover
Use crossover operator to create offspring from selected parents
Two options: CX1 or CX2
CX1
CX2
If the pool of offspring isn't filled, repeat Phases 2, 3, 4
New Population
nextGen
two offspring
Stopping Criterion
Stop after n generations
Phase 5: Generational Selection
Use elitism to create next generation population by fitness
Phase 4: Mutation
Use mutation to diversify population
Two options: swapMutate or RSM
swapMutate
RSM
Pool of offspring

Figure 1: Phases of the Genetic Algorithm and their Functions

For the TSP, the objective is to minimize the length of the path taken. Thus, in the GA, the fitness function applied is simply $f(x) = min(path)$. Thus, in order to improve the fitness, the value of the path length must be minimized.

1.2 Parameters

Values of parameters such as population size and number of generations can have impacts that result in improved or worse solutions. Because there is no theoretical basis to derive the optimal value for population size or number of generations, these parameters have been assigned arbitrary default values. The default values of these parameters can be adjusted in the command line through the use of flags. Table 1 outlines the available flags and the functionality of each flag.

Parameter	Flag	Default Value	Range of Use	Description
Population Size	-p	100	int	Number of candidate solutions generated in each generation
Number of Generations	-f	50	int	Stopping criterion: number of generations the GA will go through before stopping
Mutation Operator	-m	swapMutate	swapMutate, RSM	Operator used in mutation. Two operators are available: a simple swap mutation and reverse sequence mutation.
Crossover Operator	-x	CX1	CX1, CX2	Operator used in crossover. Two operators are available: cycle crossover, and a modified cycle crossover.
Selection Pressure	-s	0.5	float $(0 \leq s \leq 1)$	The proportion of population sampled during tournament selection.
Elitism Proportion	-e	0.1	float $(0 \leq e \leq 1)$	The proportion of parent generation maintained.
Mutation Rate	-mr	0.5	float $(0 \leq mr \leq 1)$	Probability that an offspring will mutate.

Table 1: Parameters and Use of Flags

1.3 How to Run the TSP Solver

The only file necessary to run the TSP Solver is the *tsp_solver.py* file. No external frameworks are necessary to run the code. In addition, the desired .tsp file must be placed in the same directory as the *tsp_solver.py* file.

To run the program with the default parameter values, the user can simply run the Python file with the name of the .tsp file after it.

```
python tsp_solver.py rl11849.tsp
```

Parameters can be manipulated using flags. For example, if the user wants to set population size to 200, number of generations to 100, mutation operator to RSM, crossover operator to CX2, selection pressure to 0.7, elitism proportion to 0.05, and mutation rate to 0.7, the following command can be run.

```
python tsp_solver.py rl11849.tsp -p 200 -f 100 -m ‘RSM’ -x ‘CX2’ -s 0.7 -e 0.05 -mr 0.7
```

After running the code, the best path value will be outputted and a *solution.csv* file containing a single column of the obtained order of locations will be created.

2. GA Implementation

The following sections will outline the algorithms used in each phase and present the code used in implementation. I have implemented my interpretations and versions of algorithms proposed by research papers (linked in Section 4) for some of the phases.

2.1 (Phase 0) Data Pre-Processing

Before implementing the GA, the .tsp file must be processed and translated into appropriate format. The function `preProcess(fileName)` takes in the .tsp file and returns a dictionary, in which each key is the node number and the values contain a tuple of the *x* and *y* coordinates of the node.

```python
# Location is represented by dictionary: key = int index, value = tuple(float x, float y)
def preProcess(fileName):
    with open(fileName, 'r') as f:
        coordinates = f.readlines()

        locations = list(map(
            lambda x: [l for l in x.strip().split()],
            coordinates
        ))

        start = locations.index(['NODE_COORD_SECTION'])
        end = locations.index(['EOF'])

        locations = list(map(
            lambda x: (int(x[0]), (float(x[1]), float(x[2]))),
            locations[start+1:end]
        ))

        locations_dict = dict(zip(
            [x[0] for x in locations],
            [y[1] for y in locations]
        ))

        return locations_dict
```

2.2 (Phase 1) Initialization

Each path (hereafter referred to as an individual) is represented using a list. A population is represented by a list of these individuals (i.e. a list of lists). The length of the population is defined by the parameter inputted by the user.

To begin the GA, an initial population is required. This generation 1 population is created by the `initPopulation(locations)` function by simply randomly shuffling the original individual and appending it until the population is filled.

```
# Create Random Population
def initPopulation(locations):
    inOrder = list(locations.keys())
    POPULATION = []

    for i in range(POPULATION_SIZE):
        random.shuffle(inOrder)
        POPULATION.append(inOrder[:])

    return POPULATION
```

Phase 1 is done only once in the GA.

2.3 (Phase 2) Parent Selection

Next, certain individuals in the current population must be selected as parent chromosomes to “breed” offspring from. In this assignment, the tournament selection method was used to select parents. The tournament selection method first selects *n* random individuals from the population (*n* is determined by the selection pressure parameter) and returns the single individual with the best fitness (i.e. the shortest path).

First, a function to evaluate the fitness of each individual is defined in `calcDistance(locations, order)`. Given the (*x*,*y*) coordinate of each node, the distance from one node to the next can be calculated. These calculations can be done in order of the given path to find the total length of the path.

```
# Calculate Total Distance of Path
def calcDistance(locations, order):
    totalDist = 0
    for i in range(len(order)-1):
        locationA = locations[order[i]]
        locationB = locations[order[i+1]]

        dist = math.sqrt((locationA[0] - locationB[0])**2 + (locationA[1] - loca-
tionB[1])**2)
        totalDist += dist
    return totalDist
```

Then, tournament selection systematically selects a chromosome to be a parent. `tournament_selection(locations, POPULATION)` implements tournament selection.

```
# Tournament Selection: Sample individuals according to selection probabilities
def tournament_selection(locations, POPULATION):
    mating_pool = random.sample(POPULATION, int(SELECTION_PRESSURE*POPULATION_SIZE))
    mating_pool = sorted(
        mating_pool,
        key = lambda x: calcDistance(locations, x)
    )
    return mating_pool[0]
```

2.4 (Phase 3) Crossover

After having selected parent individuals, a crossover of the parents must be done to generate offspring. In this assignment, two crossover operators were implemented. I had found a comparison of the two operators presented by Yousaf Shad Muhammad et al. in the paper *Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator* [1] and wanted to compare the two operators for myself.

Cycle Crossover Operator (CX1)

First proposed by Oliver et al.[2], CX1 identifies "cycles" of bits between the two parent chromosomes. Then, two offspring chromosomes are produced by copying certain cycles into each chromosome. Examples and specifics about the algorithm can be found in the paper referenced in Section 4. In `CX1(parent1, parent2)` I've implemented my interpretation of the algorithm.

```
# Crossover: Cycle Crossover Algorithm (CX1)
def CX1(parent1, parent2):
    offspring1 = [x for x in parent2]
    offspring2 = [x for x in parent1]

    i = 0
    nextIndex = -1

    offspring1[i] = parent1[i]
    offspring2[i] = parent2[i]

    while(nextIndex != 0):
        nextIndex = parent1.index(parent2[i])
        offspring1[nextIndex] = parent1[nextIndex]
        offspring2[nextIndex] = parent2[nextIndex]
        i = nextIndex

    return offspring1, offspring2
```

Modified Cycle Crossover Operator (CX2)

The modified version of CX1 is proposed by Muhammad et al. in the aforementioned paper. Similarly to CX1, CX2 identifies cycles within the parent chromosomes. The two operators differ in their definition of a cycle, and approach the cycle in distinct ways. Examples and specifics of CX2 can be found in the paper referenced in Section 4. In `CX2(parent1, parent2)` I've implemented my interpretation of the algorithm.

```
# Crossover: Modified Cycle Crossover Algorithm (CX2)
def CX2(parent1, parent2):
    offspring1 = []
    offspring2 = []

    O1_index = 0
    O2_index = 0

    while(True):
        offspring1.append(parent2[O1_index])
        O2_index = parent1.index(
            parent2[
                parent1.index(parent2[O1_index])
            ]
        )
        offspring2.append(parent2[O2_index])
        O1_index = parent1.index(parent2[O2_index])

        if(parent1[0] in offspring2):
            break

    if(len(offspring1) != len(parent1)):
        sub_parent1 = list(filter(
            lambda x: (x not in offspring1),
            parent1
        ))
        sub_parent2 = list(filter(
            lambda x: (x not in offspring2),
            parent2
        ))

        if(set(sub_parent1) == set(sub_parent2)):
            nextSection1, nextSection2 = CX2(sub_parent1, sub_parent2)
        else:
            nextSection1, nextSection2 = sub_parent1, sub_parent2

        offspring1 += nextSection1
        offspring2 += nextSection2

    return offspring1, offspring2
```

The authors that proposed CX2 have documented that the modified version produces better results specifically in the TSP. In this assignment, I will test whether the modified version produced the better result on the rl11849.tsp instance.\

2.5 (Phase 4) Mutation

Relying on only crossover operators for diversity in the offspring population might not provide enough exploration of the entire solution space. The only way to explore such excluded solutions is by adding random variations to the chromosomes. Therefore, a mutation rate parameter is added so that each offspring chromosome has a certain probability of undergoing mutation. I've attempted to implement two types of mutation operators.

Swap Mutation (swapMutate)

In swap mutation, a certain point in the chromosome is randomly selected. The chromosome is then separated into two chromosomes. The order of these two chromosomes are swapped and reattached.

```
# Mutation: Swap
def swapMutate(path):
    num = len(path) - 1
    if(random.random() < MUTATION_RATE):
        a = random.randint(0, num)
        b = random.randint(0, num)
        while(a == b):
            b = random.choice(path)

        path[a], path[b] = path[b], path[a]
    return path
```

Reverse Sequence Mutation (RSM)

In RSM, two points in the chromosome are randomly selected. The chromosome is then separated into three sections. The middle section of the chromosome is reversed and the three sections are reattached in the same order.

I chose to implement RSM along with swap mutation after reading the paper *Analyzing the Performance of Mutation Operators to Solve the Traveling Salesman Problem* [3] written by Abdoun et al. The results of their analysis show RSM produced the best solutions for TSP, and I wanted to implement the mutation operator that would produce the best possible results. The function `RSM(path)` is my implementation of the algorithm.

```python
# Mutation: Reverse Sequence Mutation (RSM)
def RSM(path):
    num = len(path) - 1
    if(random.random() < MUTATION_RATE):
        a = random.randint(0, num)
        b = random.randint(0, num)

        while(a == b):
            b = random.choice(path)
        if (b < a):
            a, b = b, a

        reverse = list(reversed(path[a:b]))
        path = path[:a] + reverse + path[b:]
    return path
```

2.6 (Phase 5) Generational Selection

There are now two populations: the original population from phase 1 and the offspring population produced in phases 3 and 4. Given these populations, certain individuals must be picked to form the next generation used in the next cycle in the GA. The next generation is formed using elitism, in which the top *n* individuals from the original population are maintained while the rest of the population size is filled by the best offspring results. The elitism proportion parameter is introduced to determine how much of the original generation is maintained.

```python
# Picking the Next Generation: Elitism
def nextGen(POPULATION, OFFSPRING, locations):
    eliteNum = int(len(POPULATION)*ELITISM_PROPORTION)
    bestPopulation = sorted(
        POPULATION,
        key = lambda individual: calcDistance(locations, individual)
    )[:eliteNum]

    bestOffspring = sorted(
        OFFSPRING,
        key = lambda individual: calcDistance(locations,individual)
    )[:(POPULATION_SIZE - eliteNum)]

    newPOPULATION = sorted(
        bestPopulation + bestOffspring,
        key = lambda individual: calcDistance(locations,individual)
    )

    return newPOPULATION, bestPopulation[0], bestOffspring[0]
```

2.7 GA Implementation

Phases 0 to 5 can be combined to create a functioning GA. Figure 1 in Section 1.1 outlines the process of the GA implemented. The function `ga()` implements the steps outlined in Table 2.

Step Number	Description
Step 1 (Phase 0)	Data Pre-Processing
Step 2 (Phase 1)	Create the Generation 1 Population
Step 3 (Phase 2)	Select 2 parent chromosomes to create offspring using tournament selection
Step 4 (Phase 3)	Use crossover operators to breed 2 offspring chromosomes from parents selected in Step 3
Step 5 (Phase 4)	Use mutation operators to mutate the offspring and append the resulting 2 offspring into the offspring population
Step 6 (Repeat)	Repeat Step 3, Step 4, Step 5 until offspring population is filled.
Step 7 (Phase 5)	Given the original and offspring populations, use elitism to create next generation
Step 8 (Stopping Criterion)	when the stopping criterion is met, return the best path length and create *solution.csv*

Table 2: Steps taken in the Genetic Algorithm Implementation

3. Results

The rl11849.tsp instance of the TSP provided by [4]TSPLIB was used in this assignment. After multiple runs of the GA, the best path obtained had a length of 75,239,792.73902. The best known solution for this problem instance is 923,288 (as documented in the TSPLIB page). The result of my GA implementation yields a result approximately 75 times greater than the best known solution.

The parameters used to obtain the best path were the same as the default values except that the stopping criterion was increased to 100 generations and the CX2 crossover operator and RSM mutation operator was used.

4. References

[1] Hussain, Abid, et al. “Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator.” *Computational Intelligence and Neuroscience*, Hindawi, 25 Oct. 2017, www.hindawi.com/journals/cin/2017/7430125/.

[2] I. M. Oliver, D. J. d. Smith, and R. C. J. Holland, “Study of permutation crossover operators on the traveling salesman problem,” in Genetic algorithms and their applications: proceedings of the second International Conference on Genetic Algorithms: July 28-31, 1987 at the Massachusetts Institute of Technology, Cambridge, MA, USA, 1987.

[3] Abdoun, Otman, et al. “Analyzing the Performance of Mutation Operators to Solve the Travelling Salesman Problem .” *ArXiv.org e-Print Archive*, Cornell University, Mar. 2012, arxiv.org/ftp/arxiv/papers/1203/1203.3099.pdf.

[4] Skorobohatyj, Grg. “MP-TESTDATA Mirror of TSPLIB.” *TSPLIB*, 24 Nov. 1999, elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html.