Wholly owned by UTAR Education Foundation
(Co. No. 578227-M)
DU012(A)

UECS2153 Artificial Intelligence

May 2023

Lab 2 - Genetic Algorithm

Practical 4 Group 9

Student Name	Student ID	Practical Group	Year/Trimester	Programme
Koh Jun Dong	1801968	P4	Y4S3	SE
Goh Jie Sheng	2102074	P4	Y3S1	SE
Wong Wen Xuan	2101604	P4	Y3S1	SE
Yoon Wei Kai	2003219	P4	Y3S1	SE

Table of Contents

#TODO1 Population Initialisation

```
df = pd.read_csv(filename, delim_whitespace=True, header=None, usecols=[1, 2], names=['x', 'y'])

for _, row in df.iterrows():
    city = City(float(row['x']), float(row['y']))
    cityList.append(city)
```

In population initialisation, we generate a city list by reading the filename csv with pandas built-in function.

For each of the row in the dataframe, we append the x and y coordinate to the cityList.

#TODO2 Parent Selection – Tournament Selection

```
def tournamentParentSelection(population, poolSize=None, remove_duplicate=False, tournament_size=3):
    if poolSize == None:
        poolSize = len(population)

    matingPool = []
    population = population.copy()

    # Save the fitness of each individual in the population in a list
    fitnessList = [Fitness(ind).routeFitness() for ind in population]

    for i in range(0, poolSize):
        tournament_indice = random.sample(range(len(population)), tournament_size)
        tournament_fitness = [fitnessList[i] for i in tournament_indice]
        tournament_population = [population[i] for i in tournament_indice]

        best_fitness = max(tournament_fitness)
        best_index = tournament_fitness.index(best_fitness)
        best_individual = tournament_population[best_index]

        if remove_duplicate:
            population.remove(best_individual)
            fitnessList.remove(best_fitness)

        matingPool.append(best_individual)

    return matingPool
```

Tournament parent selection has been written to take up to four arguments: population, pool size, parent duplication and tournament size.

We first make a copy of population as we may be changing the list. We made the discovery that repeatedly calling `Fitness().routeFitness()` is very slow. As such, we cache the calculation for everyone in the population once, and conduct all arithmetic and selection based on that cache.

We choose a random set of indices based on the tournament size and use that to pull the fitness and individual into temporary lists. We pull the fittest individual from the tournament, and depending on whether parent duplication is allowed, we remove them from the tournament selection in the next round of tournament.

#TODO3 Parent Selection – Proportional Selection

```
def proportionalParentSelection(population, poolSize=None, remove_duplicate=False):
    if poolSize == None:
        poolSize = len(population)

    matingPool = []
    population = population.copy()

    # Save the fitness of each individual in the population in a list
    fitnessList = [Fitness(ind).routeFitness() for ind in population]

    # Calculate the total fitness of the population
    total_fitness = np.sum(fitnessList)

    for i in range(0, poolSize):
        # Generate a random number between 0 and the total fitness
        rand = random.uniform(0, total_fitness)
        current_sum = 0

        # Iterate through individuals and accumulate fitness values
        for index, individual in enumerate(population):
            current_sum += fitnessList[index]
            if current_sum > rand:
                if remove_duplicate:
                    population.pop(index)
                    fitnessList.pop(index)
                    total_fitness = np.sum(fitnessList)
                matingPool.append(individual)
                break

    return matingPool
```

We used a similar method aforementioned to reduce the amount of `Fitness().routeFitness()` call. Here, we calculate the sum of all the fitness value from the cached fitness list. We then choose a random value to be the selector point.

In the enumerate for-loop, we add each of the fitness sequentially, until the sum exceeds the chosen random value. The fitness value that caused the sum to exceed the chosen random is our proportionally selected parent.

As before, we also determine whether the user want to allow for duplicate parents and remove them from the selection pool and adjust the total fitness of the pool accordingly.

#TODO4 Survival Selection – Merge, Sort, Truncate

```
# Sort population by fitness
elites = population.sort(key=lambda x: Fitness(x).routeFitness(), reverse=True)

# Truncate population to eliteSize
elites = population[:eliteSize]
```

Our survival selection is very simple. Population is first sorted by fitness, greatest first and put into the elites list. The list is then truncated to `eliteSize`.

The merging is done in `oneGeneration()` already. The elites are selected first through this function, the parent selection process is carried out, and then the children are merged with the elites to form the new generation.

#TODO5 Crossover – Partially Mapped Crossover Approach

```
# Define random crossover points
crossover_points = sorted(random.sample(range(len(parent1)), 2))
start_point, end_point = crossover_points

# Initialize child chromosomes
child1 = [-1] * len(parent1)
child2 = [-1] * len(parent2)

# Copy the crossover segment from parents to children
child1[start_point:end_point+1] = parent1[start_point:end_point+1]
child2[start_point:end_point+1] = parent2[start_point:end_point+1]

# Map genes from the second parent to the first child
for i in range(start_point, end_point+1):
    if parent2[i] not in child1:
        index = parent2.index(parent1[i])
        while child1[index] != -1:
            index = parent2.index(parent1[index])
        child1[index] = parent2[i]

# Map genes from the first parent to the second child
for i in range(start_point, end_point+1):
    if parent1[i] not in child2:
        index = parent1.index(parent2[i])
        while child2[index] != -1:
            index = parent1.index(parent2[index])
        child2[index] = parent1[i]

# Fill in the remaining genes using the remaining genes of the parents
for i in range(len(parent1)):
    if child1[i] == -1:
        child1[i] = parent2[i]
    if child2[i] == -1:
        child2[i] = parent1[i]
```

In our PMX implementation, we first choose two random indices in the parent to act as the crossover points.

Each of the child is initialised with the element `-1`, then we copy the crossover segment from the parents to their respective children. Next, we iterate and check if the element in the opposing parent exists within the children.

If the number does not exist, we do the following:

1. We take note of the non-existent number, `nen`. We also have our current index, `i`, which points to `nen`.
2. We use `i` as the index to find a value on the **main parent**, known as `parent1[i]`.
3. We find the index of the value `parent1[i]` on the **opposing parent**, known as `index`.
4. We check whether the child has a value at `index`.
5. If no, we insert `nen` at `index`.
6. Otherwise, we loop back to 2, setting `i` to our current location.

We do this for both children. Once we are done, we fill in the rest of the blanks sequentially from the **opposing parent**. Note that this function assumes both parents are of the same length.

#TODO6 Mutation – Insertion Mutation Approach

```
# Select a random position to insert the gene
insert_position = random.randint(0, len(route) - 1)

# Get the gene to be moved
gene_to_move = mutated_route[i]

# Remove the gene from its original position
mutated_route.remove(gene_to_move)

# Insert the gene at the new position
mutated_route.insert(insert_position, gene_to_move)
```

For our insertion mutation, we loop through each of the gene and run a mutation chance. This is in the original code and is not shown here.

If the gene mutates, we choose a random insertion location. We take the current gene, remove it from its current location and move it to the selected insertion location.

Optimisations & Other Code Changes

np.sqrt() to math.hypot()

```
from math import hypot
…
distance = hypot(xDis, yDis)
```

During our profiling test, we discovered that the distance calculation function in the City class is taking up a significant amount of time. We have swapped it with math.hypot() instead. According to the profiler, this resulted in a 50% reduction in distance runtime.

Check `extras/profiler_math_hypot.txt` and `extras/profiler_np_sqrt.txt` for reference.

Changes to random parent selection

```
def randomParentSelection(population, poolSize=None, remove_duplicate=False):
…
matingPool = []
population = population.copy()

for i in range(0, poolSize):
    candidate = random.choice(population)
    if remove_duplicate:
        population.remove(candidate)
    matingPool.append(candidate)
…
```

To allow the user to choose whether a parent can be chosen twice, we added a toggle to random parent selection to remove the candidate from the original pool if they have been selected.

Other changes

These changes are too extensive to be directly included in the PDF. Check the Jupyter file for more information.

1. `oneGeneration()` has been modified to accept parent selection method as a parameter. Furthermore, parent duplication and tournament size has been added as parameters.
2. The entire #TODO7 box has gone through extensive modifications.
    a. The entire codeblock is now a function, to make it easier to integrate with automated testing and parameterisation.
    b. We keep track of each iterations' best route and route distance. This includes the initial population.
    c. The function also takes `write_file` as parameter. If `write_file` is true, we write each iteration's best route and route distance into a csv. The CSV name is defined by `run_id`.
3. We setup a bunch of default parameters, to make it easier for our hyperparameter testing.

4. We setup a bunch of loops based on a certain set of values to run our hyperparameter testing.

Hyperparameter Optimisation

Unless otherwise specified, these are the hyperparameter used during each of the test runs:

Variable	Value
Iteration Limit	250
Parent Selection	Tournament
Parent Duplicate in Selection	Disallow
Population Size	50
Elite Size	30% or 15
Mutation Probability	0.00015
Tournament Size	10% or 5

Mutation chance

We use the mutation probabilities of 0.00001, 0.00005, 0.0001, 0.0002, 0.0003, 0.0004, 0.0008.

Each hyperparameter is ran 10 times for more representative data.

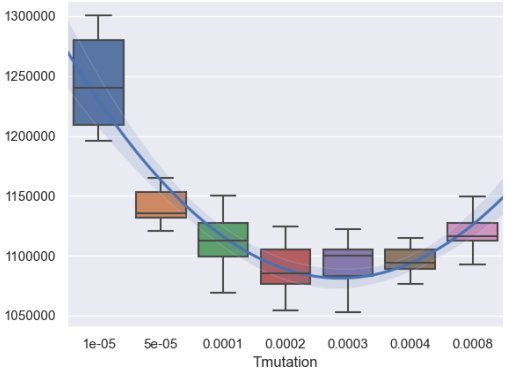

Figure 2: Tournament Selection

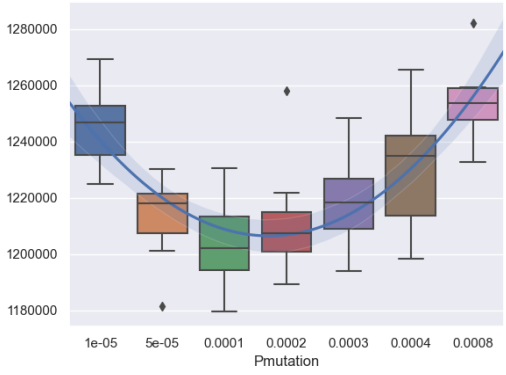

Figure 1: Proportional Selection

Both proportional and tournament selection have different preference for mutation probability. However, our search space is relatively limited, and we are limited to 250 iterations. It is very clear that mutation chance is an important factor in determining the convergence rate.

Elite Proportion

We use the elite proportion of 0.1, 0.2, 0.3 and 0.5. We run each parameter 20 times.

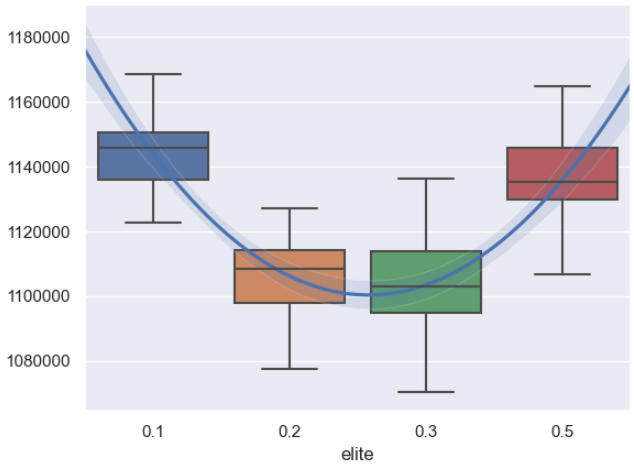

Figure 3: Elite proportion in Tournament Selection

It is suggested that the elite proportion between 0.2 and 0.3 to be most ideal in our search space. Larger elite proportion significantly reduces the number of offspring, thus affecting convergence.

Population Size

We use the population size of 30, 50, 80 and 100. We run each parameter 20 times.

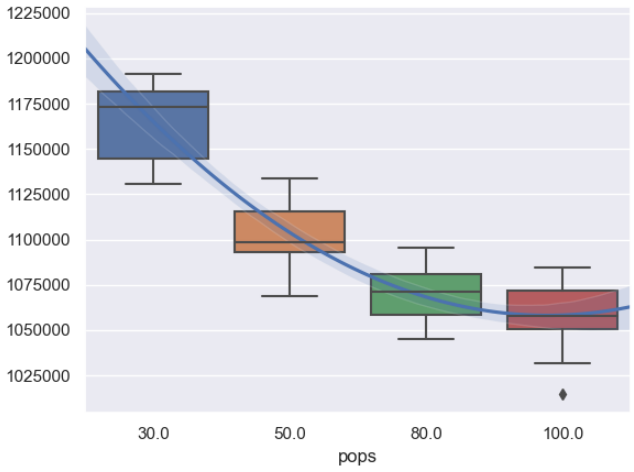

Figure 4: Population size in Tournament Selection

As the population size increase, the less iteration is required to converge on a better solution. However, increasing the population size causes a non-linear increase in execution time. Hence, in our final run we decided to keep using 50 population as our default.

Tournament Proportion

We use the tournament proportion of 0.05, 0.1, 0.2 and 0.3. We run each parameter 20 times.

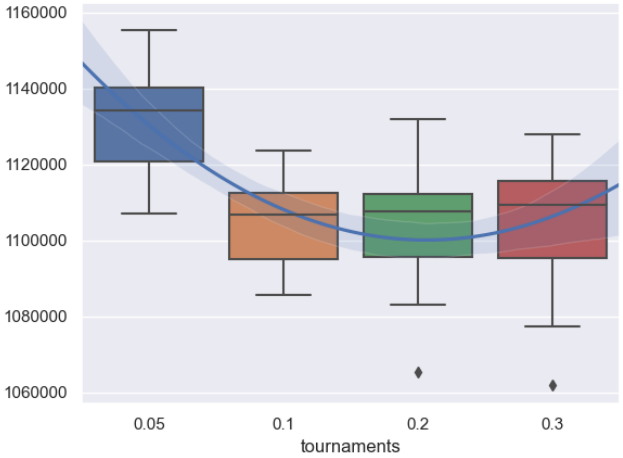

Figure 5: Tournament proportion in Tournament Selection

Tournament size needs to be sufficiently big for it to be effective. Tournament size that is too small just becomes random selection.

Final Answer, Distance & Validation

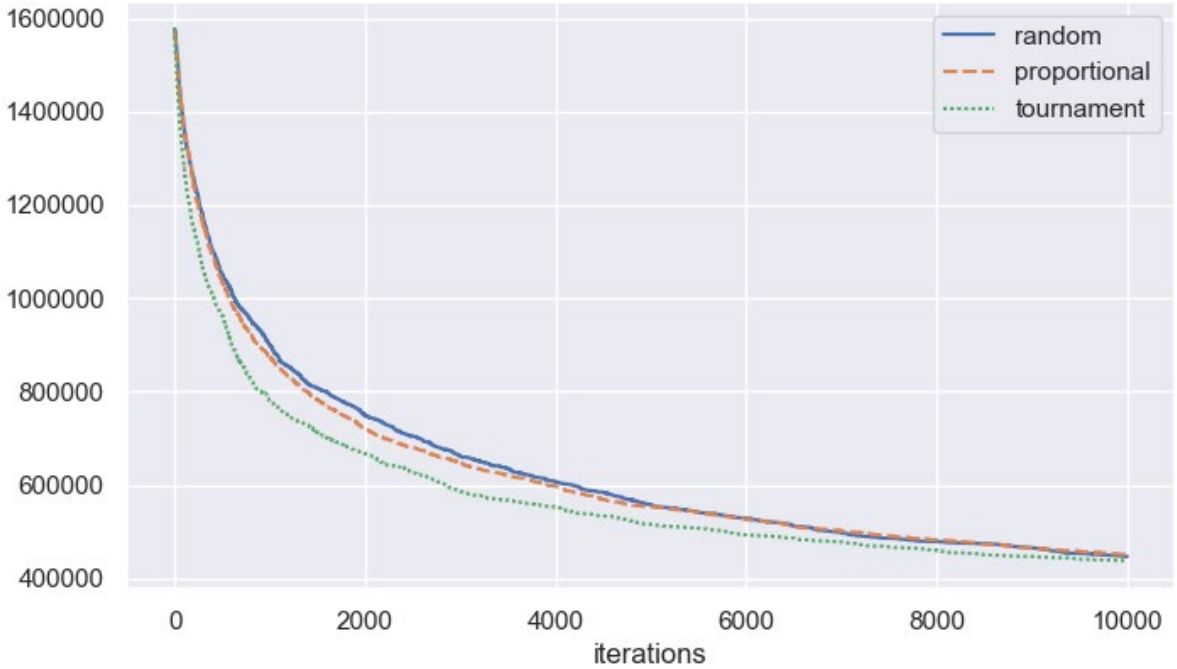

Figure 6: A plot of best distance over iteration count

Parent Selection	Time to 10000 iterations	Best distance (Lower is better)
Tournament	851.0 s	436365
Proportional	909.2 s	450989
Random	774.0 s	446920

	Iterations to 1 million	Best Distance Time to 1 million (est.)
Tournament	413	35.1 s
Proportional	571	51.9 s
Random	627	48.5 s

	Iterations to 600k	Best Distance Time to 600k (est.)
Tournament	2840	241.7 s
Proportional	3897	354.3 s
Random	4177	323.3 s

Tournament selection is a clear winner in almost all scenarios. On a per-iteration basis, random selection is faster than tournament selection. However, the convergence rate of tournament is significantly faster.

Tournament selection is on average faster than random selection by 30-38%.

One of the things we failed to consider is the amount of memory required by each of the method. Owing to our caching optimisation strategy, it is conceivable that with a large enough population, tournament selection will lose its edge.

Proportional selection faired very poorly in our tests. It did not converge as fast as random selection, and it was the slowest to run per iterations. This could be a result of our hyperparameter optimisation – after all, we mainly used tournament selection when trying to find an optimum parameter.

Final Answer

After 9994 iterations, tournament selection arrived at a solution with the **route distance of 436365.39.**

We verify that the answer is fulfils the requirement of the travelling salesman. Attached with this assignment is `extras/answer.csv`.