Artificial Intelligence

Assignment 1

Student Name: Tomasz Gruca

Student ID: 21733075

IMPLEMENTATION DETAILS

LANGUAGE

To implement a genetic algorithm to create a solution for the travelling salesman problem, I choose python as my preferred language because I am more familiar with the language so understanding syntax would be easier and with many libraries for implementation, python was my choice of language for this problem. Additionally there was a lot of material on implementation online which would help with any issues I would have myself.[1]

LOAD_TSP

This function opens the file and reads the lines, it creates a list of nodes that all the coordinates of the cities will be stored, if a line of the read in line has NODE_COORD_SECTION, it still signal the node coordinates, setting it to true, if its true the lines are split of any whitespace and then appends the nodes and returns the array list back, it extracts the x and y coordinates

COMPUTE_DISTANCE_MATRIX

Computes the distance matrix for the nodes, it takes the nodes x and y and calculates the Euclidean distance using the NumPy's library, and also normalizes the distance between them.

FITNESS

This is used to evaluate the genetic algorithm to see the quality of the tour by getting its full distance, it gets the input of the specific run and the distance matrixes of all the distances between each pair of city. It calculates the distances between the consecutive cities and once the sum is computed, additionally calculates the distance from the last city to the first city to complete the TSP problem. We return a negative value as the genetic algorithm will seek to maximize fitness, this negative value represents a shorter tour for the TSP, which is better for the TSP problem.

INITIALISE POPULATION

Creates the first population for the algorithm to use, but generating a set of random tours, it gets the size of the runs and the number of cities in the TSP problem, initializes an empty array and then for the range of the number of runs, generates a random tour of the cities and shuffles them, then the it inserts city at beginning and end of array to make sure that these cities are first so that the run starts and ends at the same city to complete the TSP route, it returns the population afterwards.

SELECTION

I implemented the core genetic components required for solving the TSP, for the selection of next generation, I used tournament selection because it helps with keeping the best solutions that are known to us but also adding new diversity which could help us reach a global maxima instead of just a local maxima, tournament selection works by getting a small subset of individual and seeing which of those subset individuals are the best when compared to each other in a tournament style, the best individual from the tournament is selected to be a parent for the next generation, this is repeated until we get the amount of parent genes necessary for the next generation.

By adjusting the tournament size, we get more increase favorability for bigger individuals and smaller tournaments help us maintain diversity and stopping us from going to a local maxima by going towards a convergence to a local maxima. Additionally, tournament selections chooses the stronger individuals on a regular basis rather than a probability selection like roulette wheel selection, also tournament selection remains stable when changing other parameters in the genetic algorithm.

This is done in the code by getting the population, and selecting a random k number of individuals, then it compares their fitness score and returns the one with the highest score of x[1][0] using the zip function we can pair individuals and the random.sample gets a random sample from the subset.

CROSSOVER

For crossover, I implemented order crossover and partially mapped crossover, I choose those two crossover methods as they would work with the travelling salesperson where they keep existing node paths that may be optimal are kept and the order is readjusted to see the best order of those nodes. Order crossover selects a subset of genes from both parents, keeps the order and places them in the same position in the child, then place the remaining genes filled from the second parent into the first child in the same order they appeared in the second parent and ensuring that no genes are duplicated, this preserves the sequence of a certain values of cities that the salesperson has gone to but in the same order, this helps explore other paths whilst also keeping integrity that works from other generations. [2]

In code: the order_crossover function creates two childrens from the parent parameters, it creates a random subsequence child_middle from the start and end of the parents, it then creates the children and adds their genes from the opposite parent and then select the subsequent other values to fit and inserting missing genes in order, it then returns the children

The other crossover method I used was partially mapped crossover, another good solution for TSP, it works by swapping segments of the parent then ensuring that the offsprings don’t have repeated genes by mapping them, this makes sure that each city is only travelled to once, except the first and last, it keeps the structure of the parent’s like the OX crossover but also adds new combinations of numbers which can improve the best path[3], it preserves validity from previous generations whilst also increasing the chances of finding more optimal paths by changing the orders of how the salesman travels to these cities.

In code: the pmx_crossover function creates two empty children, it randomly selects a segment of the parents and sorts them, the pmx_fill is used to fill remaining positions, checks if the gene is in the child, it traces its position, avoiding segment that was changed and places the number, it returns the children after applying pmx_fill to both the children of opposite parents.

MUTATION

I used displacement and inversion mutation for the TSP problem, these mutations allow to add variations into the populations which prevent the algorithm from climbing a local maxima early, allowing it to find more potential solutions.

Displacement mutation grabs a random segment of nodes in the original parent and inserts it at a new position in the sequence, this mutation gives more diversity by shuffling around different paths that the algorithm travels to, finding if those paths give a better route and finding potential shorter paths by having paths that were far away from one another to be closer now by mutation. This helps reduce the distance travelled creating the best solution.

In code: randomly selects a segment, chooses two points from the start and end, removes the selected segment from the list and then inserts that segment back into the list at a random position, returns the modified list with the changed list of cities.

Inversion mutation selects a sequence of nodes in the path and inverses the individual nodes in the path, in the tsp problem, we are inversing the order of cities, this allows us to get a different perspective of the path instead of keeping it in order all the time, which may find us the most optimal pieces of the path by constantly adjusting the sequence, it helps from getting stuck at the local maxima by changing the path.

In code: the inversion mutation gets randomly selects two positions from the list and sorts them to make sure the first index is smaller than the second, it then reverse the list between those points, returns the list with those points reversed.

GENETIC ALGORITHM

This loads the tsp file and computes the distance between the nodes, it uses the parameters to initialize the population and creates the fitness, this has a loop for each generation and calculates the fitness, new individuals get created and applied using the crossover and mutation depending on the crossover_operator and mutation_operator parameters based off the name, it also has a counter created with a max counter number of 1000 so if after 1000 generations it doesn't improve it will stop the run, it will then record the time taken, if the best generation fitness is better, it will update all the global variables, afterwards it will see the best solution and how long it took time wise and then return all the best fitnesses and solutions aswell as the parameters it was using to create that result.

GENERATE_NEW_POPULATION

The population is sorted and the top are saved in elite_count, new individuals are generated using tournament selection and applying crossovers and mutations, the crossover rate is the rate of applying crossovers to the parents, same for the mutation rate, the children are added to a new population until its filled and then it returns the new population with the same size

EVALUATE_PARAMETERS

Allows the user to hardcode parameters for the genetic algorithm to use, then for each of those in the that array parameters we run it if we find the best data, we print it to the console and also we create a txt file to store all the data correctly, then we read in the datasets and evaluate the best result with that txt file, using the best parameters that we generate, we plot a graph of results, and we print the best solutions to the console after the run.

DESIGN CHOICES

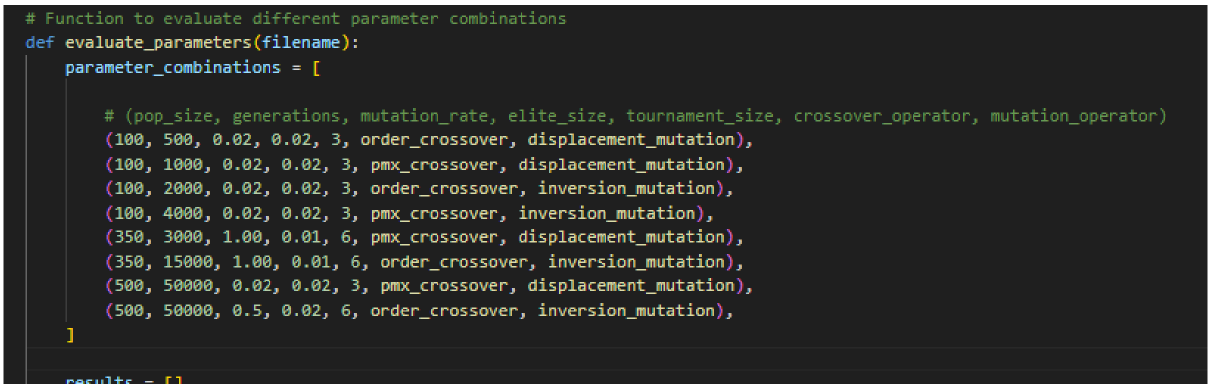
To select which mutation and crossover method is used, we allowed the user to pass them in as parameters, this allows for the user to fine tune the details of the parameters but as well as what mutation and crossover allowing the user to have different combinations of parameters, by providing this option the user can fine tune if he finds a specific set up which is getting closer to the known optima and finding the more optimal set up than relying on a random function to hopefully give you the combination you wanted.

Additionally to test all the parameters, I used a grid parameter combination which manually tested different combination of algorithms and saving the best score to a txt file allowing it to be saved to another file to have a list of all the scores from several tests to see which parameters performed the best and then use those list of parameters to further use to see whether it will improve more on those results or whether it was the local maxima.

A method was also created for the larger datasets that were taking a lot of generations with no improvement, a counter was introduced to keep track of how generations were produced with no improvements, if no improvements were made for a certain amount of generations, this run would be stopped as it might have hit a roadblock or local maxima and no longer advance so its no use to keep running it.

Experimental Results

For testing different size of datasets, I tested on the recommended datasets, this include berlin52, kroA100 and pr1002.



```python
# Function to evaluate different parameter combinations
def evaluate_parameters(filename):
    parameter_combinations = [

        # (pop_size, generations, mutation_rate, elite_size, tournament_size, crossover_operator, mutation_operator)
        (100, 500, 0.02, 0.02, 3, order_crossover, displacement_mutation),
        (100, 1000, 0.02, 0.02, 3, pmx_crossover, displacement_mutation),
        (100, 2000, 0.02, 0.02, 3, order_crossover, inversion_mutation),
        (100, 4000, 0.02, 0.02, 3, pmx_crossover, inversion_mutation),
        (350, 3000, 1.00, 0.01, 6, pmx_crossover, displacement_mutation),
        (350, 15000, 1.00, 0.01, 6, order_crossover, inversion_mutation),
        (500, 50000, 0.02, 0.02, 3, pmx_crossover, displacement_mutation),
        (500, 50000, 0.5, 0.02, 6, order_crossover, inversion_mutation),
    ]

    results = []
```

Parameters being used to test different combinations of values

These parameters were used to find a lot of information about different combinations of values. From having smaller population sizes that would effectively and quickly run, however we don't know whether we would find optimal solutions with this much generations being run or whether the genetic algorithm ran out of generations to find even more optimal solutions.

To help with that issue, we increased the number of generations whilst keeping the same amount of population size to see whether these longer runs would allow the algorithm and to see whether more generations optimized our result closer to the known optima, after changing the size of our generations to have a higher chance of running into an optimal solution, we increased the population size as the number of generations grew to allow for more different variations to be found by the algorithm children and also stop a chance of converging to a local maxima, this would give us a way of exploring more chances. Each run would also have a different run of using different crossover and mutation methods to see which combination produced better results, additionally elitism was brought into this, elitism is the process of preserving the best individuals from each generation and passing it onto the next generation without undergoing mutations or crossovers, this makes it so that the best solutions that we have found are not lost by changing a value in it and undermining it all. Experimenting with different sizes

Using different values for both crossover rates and mutation and tournament size all give different properties of how each run will go differently, this lets us see how different runs will occur and which parameters might be leading us to a better path and which stop further from the known optima which might not be useful to us or they might be if we change a further parameter that could lead to an even better result. So each run had a different set of parameters to see which would perform better straight off initially and if further changes will make it even more accurate to the known optima.

PERFORMANCE COMPARISON WITH KNOWN OPTIMAL SOLUTIONS

Using an online website that has a list of all the optimal solutions for the TSP problem[4] allowed me to analyze how close my AI model was to the known optimal or whether I would have to adjust my parameters further to see if another combination will create a better result.

Berlin52

Overall Best Fitness: 7928.98

Best Solution Found: 7928.98 in 4.82 sec

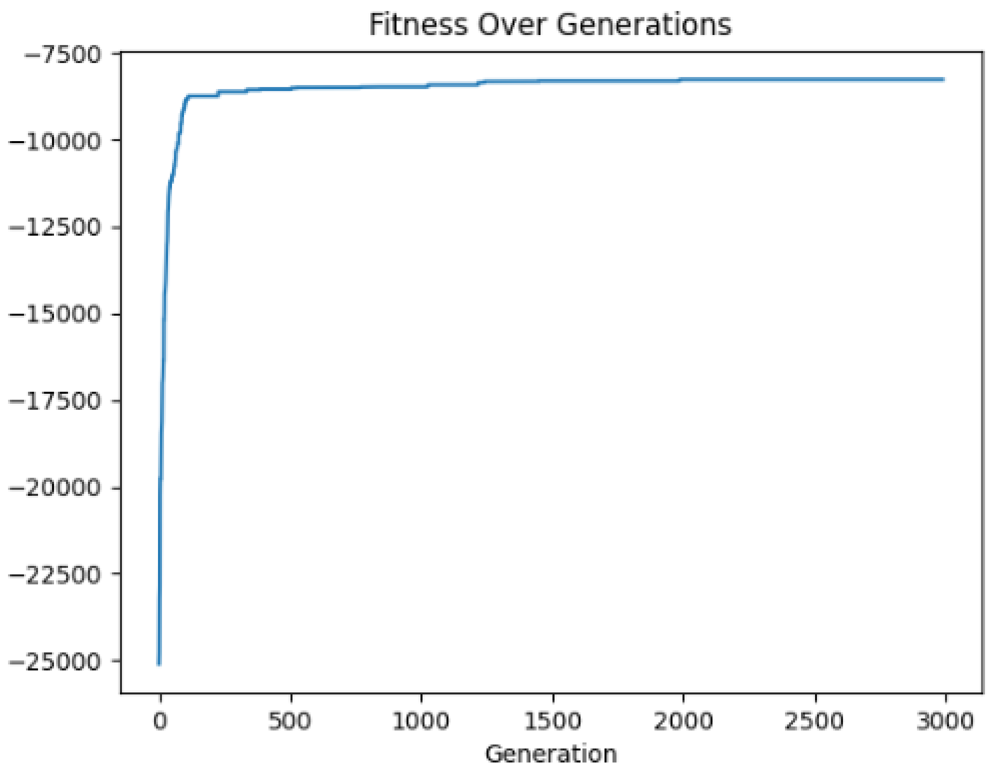
Overall Best Parameters: Population Size = 350, Generations = 3000, Crossover Rate = 1.0, Mutation Rate = 0.05, Elite Size = 0.1, Tournament Size = 6

Crossover Operator: pmx crossover

Mutation Operator: displacement mutation

Comparing my best result to the known optima which is berlin52 : 7542m , my closest model was only 386 off which I would think is a pretty close value to the known optimal, I would have to further adjust a value by a small amount by doing so might risk me getting a worse score for the next run.

For Berlin (7928 – 7542)/7542 x 100 = 5.12 % above the optimum



Graph of Berlin52, fitness over generations

KroA100

Overall Best Fitness: 23476.65

Overall Best Parameters: Population Size = 600, Generations = 2500, Crossover Rate = 1.0, Mutation Rate = 0.04, Elite Size = 0.05, Tournament Size = 5

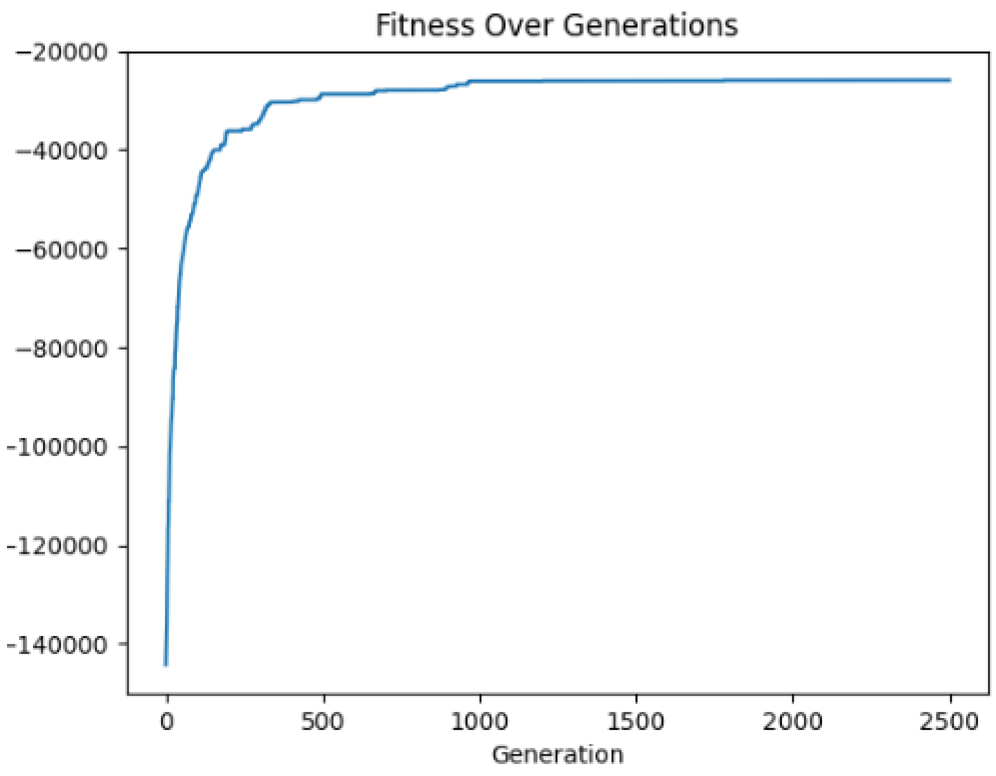
Crossover Operator: pmx crossover

Mutation Operator: inversion mutation

Comparing the best known optima from kroA which is 23476.65 gives a difference of 2194

For KroA100, (23476 – 21282)/21282 x 100 = 10.31% above known optima

For KroA I was finding it harder to find a closer optimal then this one.



Graph of kroA100, fitness over generations

Pr1002

Best Solution Found: 728538.82 in 11490.44 sec

Overall Best Fitness: 728538.82

Overall Best Parameters: Population Size = 100, Generations = 45000, Crossover Rate = 1.0, Mutation Rate = 0.02, Elite Size = 0.05, Tournament Size = 4

Crossover Operator: pmx crossover

Mutation Operator: inversion mutation

Testing the pr1002, the dataset was very large that my laptop was having difficulty run with the data, so I was only able to get one complete run of the dataset.

My PR1002 was of a percentage of 181.22% above the optimal known so there is very much room for improvement with this dataset, it is just that running it would take an extreme amount of time for my laptop to execute.

DISCUSSION OF POTENTIAL IMPROVEMENTS

Cross-Grid Search

Using a cross-grid search to create a list of each parameter and running accordingly instead of individually might have resulted in a better more optimize result instead of manually changing the parameters each time, additionally allow for this would make me run it in the background to get the best results overtime.

Python

Whilst using python proved to be good with syntax and understanding the problem, python proved to be very slow when executing the code, this made testing a lot harder as each problem was taking more time and waiting for an optimal result took longer and longer, using a language like C or Go might have increased the speeds of the runs which may have allowed more time for testing different parameters.

Big O Notation

Given more time, I would have refactored the code to look at the big O notation of each function and trying to reduce it to reduce O(n^2) or any higher big O calls, having everything be linear or log of would have increased the speed of each run allowing for more time to test.

With these potential improvements, I believe that I would get closer to the optimal in the bigger datasets and even closer in the smaller datasets.

REFERENCES

https://medium.com/aimonks/traveling-salesman-problem-tsp-using-genetic-algorithm-fea640713758

https://mat.uab.cat/~alseda/MasterOpt/GeneticOperations.pdf

https://www.researchgate.net/figure/Example-of-partially-mapped-crossover_fig1_312336654

http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/STSP.html