**INTRODUCTION**

The **travelling salesman problem**(**TSP**) consist on "finding a complete tour of n given cities of minimal length" [1]. This project is about analysing a previously provided Genetic algorithm code to solve a TSP problem and implementing different representation and methods to improvise the solution. On the tableN, we present the list of default methods, plus the new implemented methods used to modify the genetic algorithm for getting a better solution.

The effect of 2-opt local optimization heuristic was tested on the final solution. ??

Table 1

|  |  |  |
| --- | --- | --- |
|  | **DEFAULT** | **Our implementation** |
| **REPRESENTATION** | Adjacency | Path |
| **CROSSOVER** | Alternate edges | Order Crossover |
| **MUTATION** | Swapping | Insertion |
| **PARENT SELECTION** | Ranking with SUS | Fitness proportionate |
| **SURVIVAL SELECTION** | Elitism | Replace Worst (GENITOR) |
| **STOPPING CRITERIA** | The amount of individuals of equal fitness/cost reaches a pre-defined limit | 1. Maximal improvement of the solution over last N generations is lower than limit. 2. Diversity in phenotype space is lower than limit. 3. Efficiency drops below a limit. |
| **LOCAL OPTIMIZATION HEURISTIC** | 2opt method/ local loop detection | - |

**EXISTING GENETIC ALGORITHM**

The existing genetic algorithm code for solving the TSP problem include a GUI file with the following variables initialised on it.

* Number of cities.
* Data file with the list of cities and their location (on x-y axes).
* Number of individuals or solutions in each generation.
* Number of generations.
* Probability of crossover and mutation.
* Elitism percentage.
* Percentage of equal fitness individuals for termination condition.
* Boolean value for applying local optimization heuristic.

The GUI function loads the data from the file for a particular TSP problem with a fixed number of cities and calls for the genetic algorithm function for initialising a matrix of distances between each city in the file to every other city which is then used to get a list of TSP solutions by using the inbuilt random permutation function in MATLAB. Below are the various functions used to modify the population to get a better improved solution for the TSP problem.

1. ~~Path-to-Adjacency: This function uses the randomly generated population and converts it to adjacency representation. It starts from the initial node and assign next node value to the node represented by the current node value, until the last node is reached. This newly generated values are used as chromosomes for adjacency representation of the problem.~~
2. ~~Calculation of TSP cost function: The distance matrix is calculated by using the data from the file containing the X-Y coordinates of the cities for a particular TSP problem. This distance matrix has the distance between every two cities in the list. This matrix is used to calculate the distance travelled by each individual solution in a particular generation. The cost function is the total distance covered by that particular solution.~~
3. ~~Parent Selection: Ranking algorithm is used here as a parent selection method to rank the individuals as per their fitness values. The ranks are allocated from 0 to 2 such that the best solution with minimum cost has highest fitness equal to 2 and vice versa.~~
4. ~~Recombination: The default crossover operator used here is alternating edge crossover. The crossover probability decides whether or not the parents undergoes recombination.   
   <<EXPLAIN the METHOD>>~~
5. ~~Mutation: The default mutation operator used here is Swap operator. The probability of mutation decides whether or not the parent undergoes mutation. In this process, we select 2 random node values between 1 and number of cities and swap the path between selected node values.~~
6. ~~Survival selection: It is done using Elite percentage which tells that the selected percentage of population do not go any recombination or mutation and are the elite population who survive to the next generation. If the percentage is 100, still this function selects 2 parents stochastically according to the fitness function to generate 2 offsprings which replace the weakest chromosomes from the parents with fitness rank 0.~~
7. ~~Local heuristic optimization: It uses a Boolean variable to decide whether or not the local heuristic optimization is implemented. 2-opt method used here checks by reordering the path of length 3 and check if there is a reduction in the distance covered (cost value), in which case it replaces the solution with the reordered solution~~

~~After all the above functions were implemented, the cost function of the newly generated population is recalculated, and the steps are reiterated until we reach the maximum number of generations.~~

**EXPERIMENTS PERFORMED ON EXISTING PROGRAM**

We implemented a function that automatically varies the parameters of the existing genetic algorithm. We take as input a range of values for each parameter, listed on TableN, and get as output a graph of minimum distance travelled across all generations (keeping the max. number of generation constant) NOTE: comment stopping criteria to test this vs. each parameter value.

**average number of evaluations to a solution** (AES) – for N runs

We varied one parameter at the time, by setting all the rest as a constant (default value in tableN), we also disabeled stopping criterions, so we can analyse all experiments under the same conditions.

Table

Parameter – default value – range of variation

In order to measure the efficiency of the algorithm, we also creatd a graph of the time that it took to the algorithm to reach the minim traveling distance vs the parameter value. We consider plotting the number of the best generation instead of the time, but as we are changing parameters the execution time for each generation might change as in we change the parameters (for instance, for more number of individuals, the execution time for single generation is longer). So, we decided to use time in order to have a standard measure among parameters.

1. Variation of the number of Individuals.

Graph

1. Optimal number of generations.
2. Optimal values for other parameters (Crossover, Mutation, Elite percentage).

The above optimal results were tested on the different number of cities. This optimal setting is used to proceed with the rest of the project henceforth.

**IMPLEMENTED ALGORITHM**

1. Stopping Criteria: Implementation of Stopping criteria to the existing algorithm to avoid useless iterations. Below, the stopping criterions that were implemented (Code in Appendix A):

**Case1**: If the maximal improvement of the solution over last N generations is lower than a predefined limit. The algorithm proceeds until the best solution during the evolution process doesn't change to a better value for a predefined value of generations. This predefined value can be 20% or 30% of the generation number which the best solution has found so far. i.e. the algorithm reaches to a value of 200 at generation 50, then this value doesn't change for 15 generations (30% of 50), so the algorithm stops.

**Case2**: If Diversity in the phenotype space is lower than a particular limit. The limit is set by manual experimentation by observing the variation in the fitness values which leads to the diversity in the population phenotype.

**Case3**: If Efficiency drops below a certain limit which is also tuned as per experimentation.

1. Representation: We implemented the Path representation as it is easier to interpret. This representation shows the path of the cities travelled starting from an initial position to the next city and ends at the city which is in turn connected to the initial city to complete the tour.   
   (Code in Appendix B)
2. Crossover. (Code in Appendix C)
3. Mutation. (Code in Appendix D)
4. Parent Selection. (Code in Appendix E)
5. Survival Selection. (Code in Appendix F)
6. Testing optimal combination of parameters on benchmark problems.

**APPENDIX A**

**APPENDIX B**

**APPENDIX C**

**APPENDIX D**

**APPENDIX E**

**APPENDIX F**

# Bibliografía

|  |  |
| --- | --- |
| [1] | J. S. A.E. Eiben, Introduction to Evolutionary Computing, Springer, 2003. |