A Genetic Algorithm for the 0-1 Knapsack and TSP

Carlos Soto Garcia Delgad i6216792 & JD O'Hea i6208128

I. Introduction

Genetic algorithms are a powerful tool we can use to help us find an approximate solution to NP-Hard problems. In this assignment we developed genetic algorithm solutions to the travelling sales man and knapsack problems.

First, we designed a genetic algorithm framework that can be implemented for any problem that can be modelled with a genetic algorithm. The common components are the individual, the phenotype, the crossover method, the selection methods, the population and a mutator. We created further objects to help with the implementation such as an evolution object, fitness calculator, an individual factory.

The phenotype, mutator, individual factory, crossover and fitness calculator have to be implemented specifically per problem but the remaining classes are abstract and applicable to any implementation.

II. Problems

We implemented the following two problems with our framework

A. Travelling Salesman problem (TSP)

The travelling salesman problem is a problem in which given an undirected, weighted, fully connected graph, the cycle with the lowest cost that visits every node exactly once has to be found.

B. Knapsack problem

Given a set of items, each with a weight and a value, it has to be decided what items have to be included in a collection, so that the total weight is less than or equal to a given limit and the total value is as large as possible.

In this study we consider n packages. The rewards associated to the packages are 1,2,3,4,..n respectively. Each package has a random weight between 1 and 10.

III. Solutions

A. Travelling Salesman Problem

1) Representation of an individual: We represent this cycle as a list of node IDs where the order of the list is the path, and then to form a cycle, the path is extended from the last node in the list to the first node in the list. This data structure is the phenotype of an individual in our implementation.

2) Population Initialisation: Initializing a population for the TSP starts with a list of all the city's identifiers, this list is then shuffled, changing the order of the route to a completely random order.

3) Fitness: The fitness calculation of an individual for the TSP implementation is straightforward; the cost of the cycle.

4) Mutator: For the mutation of the cycle, we take two random nodes in the list of nodes, and we swap their position in the list. Because the order of the list determines the route, this is changing the phenotype of the individual

5) Crossover: This is the most complex step in designing a genetic algorithm for the TSP. At this stage there are two different individuals as parents, and you have to cross them over in such a way that you maintain a cycle that visits each city exactly once in the resulting child. Thus it is not as simple as taking a portion of one parents genotype and a portion of the other

i) Order one (O1) crossover method:
   a) Take two points to cross at. Point one equal to a third of the length of the first parents phenotype and the second point to be two thirds the length of the first parents phenotype.
   b) The remaining genes are populated using the second parents genotype. We start at point two plus one and add the gene to the child if the gene has is not already contained within the child. If the gene is already contained within the child, we mark it out and move on to the next gene. 1
   c) Do ib until all genes (in this case cities) are in the child's phenotype.

ii) CX crossover:
   First introduced in [OSH87], CX attempts to create an offspring from the parents where every position is occupied by a corresponding element from one of the parents.
   a) Choose an element from one of the parents and add this element to the child.

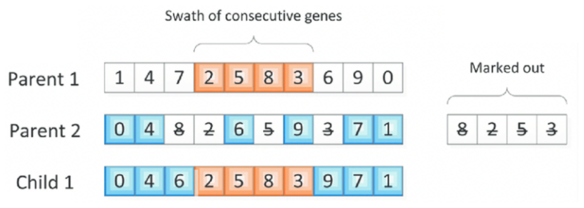

Figure 1. Worked example of O1 crossover, from [WZW+15]

b) Find the index of this element in the other parent and add the element in the index of the first parent to the child. This is necessary as adding this element from the second parent would make an illegal child.
c) Repeat iib until there are no more illegal items contained within the other parent, this forms the first cycle.
d) Repeat iia and iib with alternating parents until the child is complete, this also completes the second cycle.

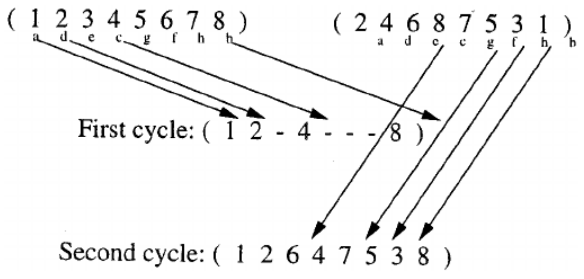
First cycle: (1 2 - 4 - - - 8)
Second cycle: (1 2 6 4 7 5 3 8)

Figure 2. Worked example of CX crossover [OSH87]

B. Knapsack Problem

1) Representation of an individual: Each individual in our population encodes a solution, i.e. a selection of packages. An individual is represented as an array of size n of 1s and 0s, where n is the number of available packages, let this array be denoted by ind_array.

$\forall i \in \{1, 2, 3, .., n\}$ package i is included in the solution if and only if $ind\_array[i] = 1$

For instance, the following array [0,1,1,0,1] would mean that the packages selected are 2,3 and 5.

2) Fitness function: The fitness function is defined as:

$$fitness = \sum_{i \in \{1,2,3,..,n\}} reward_i - (capacity\_violation)$$

where $capacity\_violation$ is the number of units by which the capacity is violated. In case the capacity constraint is respected, then this number is 0. Hence, $capacity\_violation \geq 0$.

3) Obtaining a new population: In order to obtain a new population, two different aspects have to be considered: how to select individuals to be reproduced and how to reproduce them.

i) Selecting individuals
In this study a modified version of the roulette wheel selection method is implemented. With roulette selection method, the probability of selecting an individual i for reproduction is as follows:

$$prob(i) = \frac{fitness(i)}{\sum_{j \in Population} fitness(j)}$$

In our study it might be the case that an individual with a good fitness value does not encode a feasible solution. In order to avoid the population containing only non feasible solutions (which does not happen in general) the best feasible individual in the population is always kept for the next iteration.

ii) Crossover operation The crossover method works as follows: Two indivuals' arrays are received as input, array1 and array2. A splitting point is randomly chosen, which will divide both arrays into a left and a right part. Then, the left part of array1 is recombined with the right part of array2 and viceversa. For example, let array1 = [0,1,0,1,1] and array 2=[1,0,0,0,1] and the splitting point be 1.5. Then the two new descendants will be: [0,1,0,0,1] and [1,0,0,1,1]

4) Mutation operation: The mutation operation consists on randomly selecting a given point in the array and negating its value.

IV. Experiments

When experimenting on genetic algorithms you have the following parameters to adjust for:

i) Mutation rate
ii) Number of iterations (population reproduction)
iii) Population size
iv) Size of the individuals phenotype

We tested every combination of number of iteration, population size and size of individuals phenotype with the values 10, 100, & 200. We tested 10 mutation rates (from 0 to 0.9 with a step of 0.1)

We tested each combination of parameters 5 times and took the average of the 5 resulting fitness score as the score for that combination of parameters.

In our implementation the fitness score of the run, is the fitness score of the best individual from any iteration within the run.

Note that the fitness score for the TSP is $-1 * distanceOfRoute$. This is because our framework aims to maximize the fitness score and our aim for the TSP is to find the minimum distance.

We analyse the effect of mutation rate, number of iterations and population size on the quality of our result. We assume make the assumption that the size of the individuals phenotype is representative or the problem space for the purpose of this study.

V. Results

We can see the outcome of our experiments in the Appendix Section VII. As expected we see the higher population numbers and higher iteration numbers have on average an improving effect on the most fitted individual found throughout our search.

For each experiment we can compare the graphs of varying the population parameter versus varying the iterations parameter and we can see that we are getting more gains from increasing the population parameter versus increasing the number of iterations.

It is also clear from each graph for which mutation rate to chose for a given combination of parameters.

The O1 crossover method on average beat the CX crossover method.

VI. Discussion & Conclusion

We have developed a genetic algorithm framework that can easily be implemented for a problem that can be modelled as a genetic algorithm. It is highly customisable as we have demonstrated by implementing different crossover functions, and varying our parameters. It is also possible to vary the selection method, fitness score calculator, population initializer, and mutator.

We have demonstrated the ability to gain insights from the programs behaviour by creating a report of results and building visualisations of the programs development. Our framework makes it possible to implement a feedback mechanism on many levels of the program including throughout the evolution of a single run.

References

[OSH87] IM Oliver, DJd Smith, and John RC 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*. Hillsdale, NJ: L. Erlhaum Associates, 1987., 1987.

[WZW+15] Shuihua Wang, Lu Zeyuan, Ling Wei, Genlin ji, and Jiquan Yang. Fitness-scaling adaptive genetic algorithm with local search for solving the multiple depot vehicle routing problem. *SIMULATION*, 91, 09 2015.

VII. Appendix

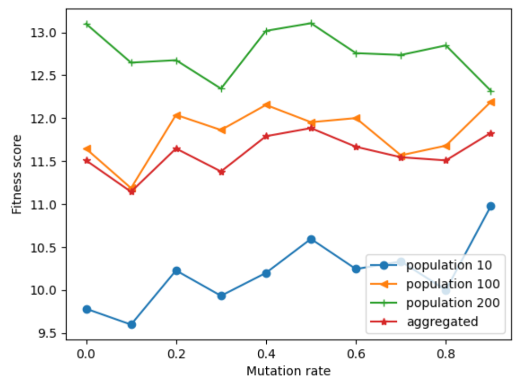

Figure 3. Plot of mutation rate against fitness score for the Knapsack problem's GA by population parameter.

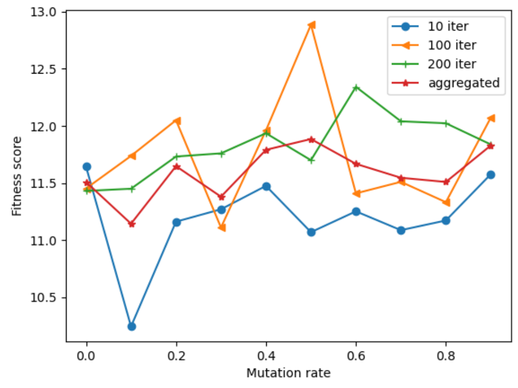

Figure 4. Plot of mutation rate against fitness score for the Knapsack problem's GA by iteration parameter.

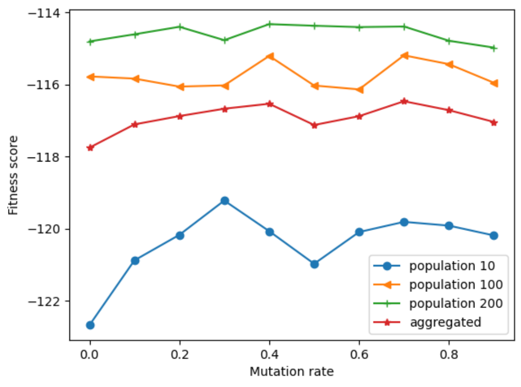

Figure 5. Plot of mutation rate against fitness score for the TSP GA by population number using the CX crossover method

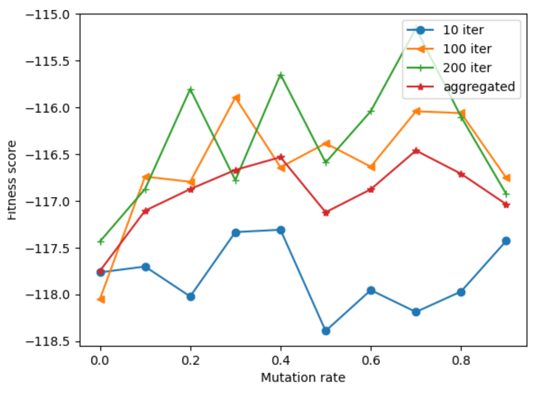

Figure 6. Plot of mutation rate against fitness score for the TSP GA by iteration number using the CX crossover method

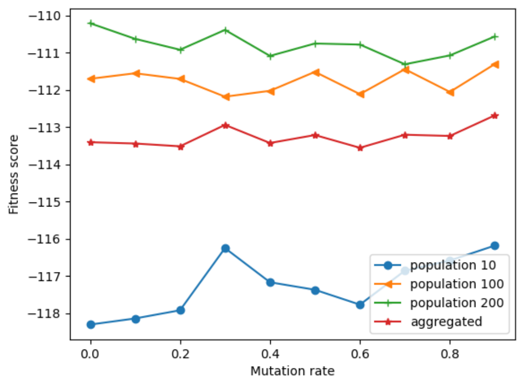

Figure 7. Plot of mutation rate against fitness score for the TSP GA by population number using the O1 crossover method

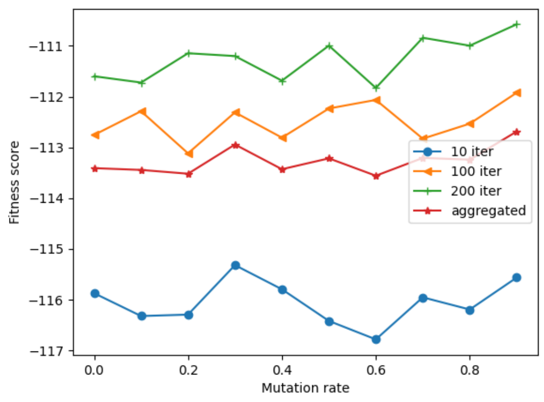

Figure 8. Plot of mutation rate against fitness score for the TSP GA by iteration number using the O1 crossover method