Leveraging Genetic Algorithms to Provide Solutions to the Knapsack and Traveling Salesman Problems

Leo J. Paggen, Panagiotis Stekouleas

March 22, 2024

Abstract

The Knapsack and Traveling Salesman problems are classic np-hard problems within the field of computer science. This report presents viable solutions to both problems using Genetic Algorithms (GAs). A list of methods are used to construct the GAs to solve both problems, namely crossover, mutation, and elitism. The methods are explained throughout the report, and near optimal solutions are presented at the end of each section.

1 Introduction

Optimization problems such as the knapsack and the traveling salesman problems are np-hard problems, requiring intricate methods to find near-optimal solutions to them.

This report aims to provide optimal solutions to both problems using genetic algorithms, with crossover, mutation, and elitism features.

2 The Knapsack Problem

2.1 Representation of the problem

The version of the problem discussed in this report is as follows: 10 items $i : w_i \in [1, 10]$, $v_i \in [1, 4]$, where w_i and v_i are the weight and value of item i, are to be put in a knapsack with a weight limit $w_s = 30$. A randomly generated list of items is used throughout this report:

item	0	1	2	3	4	5	6	7	8	9
value	2	3	4	3	2	4	2	3	4	2
weight	3	1	5	10	8	7	7	8	9	5

2.2 Specifics of the algorithm

2.2.1 Generating an initial population

Each *individual* s_i in the algorithm is a potential solution to the problem. *Individuals* are represented as bit strings of length 10, where each 0 corresponds to an item not being taken, and each 1 represents an item being taken in the solution. An example of a randomly *initial population* is given below:

Solution 1:	1010010110
Solution 2:	0111010101
Solution 3:	1100101110
Solution 4:	1001100101
...	...
Solution n_{-1}:	1011001101
Solution n:	0100111101

2.2.2 Fitness of individual solutions

Each *individual* is subject to the weight limit of the knapsack w_s, therefore a fitness function is introduced:

$$f(x) = \begin{cases} \sum_{i=1}^{n} v_i & \text{if } \sum_{i=1}^{n} w_i x_i \leq w_s \\ 0 & \text{otherwise} \end{cases}$$

where x_i is an *individual* i. The fitness of a solution is equal to the sum of the values of its items, and is 0 if the weight of the items exceeds w_s. Each solution in the *initial population* is evaluated using the above function.

A new *population* is constructed by assigning "importance" to each solution; the higher the score, the more likely a solution is to be selected in the new *population*. First, the total score of the *population* is computed as follows:

$$S_{pop} = \sum_{i=1}^{N} s_i$$

The probability of a solution being selected for the next *population* is computed as follows:

$$P(x_i) = \frac{s_i}{S_{pop}}, \quad \text{for } i = 1, 2, \ldots, \text{N}$$

The new population is then generated based on this probability.

2.2.3 Crossover function

The GA includes a single-point *crossover function*. Individual solutions are taken "2 by 2", representing *parents*. a random index is determined as the crossover point, and two *children* are generated as follows: The first children retains the genes of the first parent *genes* up until the crossover point, and the genes of the second parent from the crossover point onwards are added to it. The same is done for the second child. Below is an example of the crossover:

Parent 1: 1010010110
Parent 2: 0111010101

We perform a single-point crossover at index 4. The crossover operation results in:

Child 1: 1010|010101
Child 2: 0111|010110

In these crossover results, the vertical bar (|) represents the crossover point. Every pair of solutions undergoes crossover with a rate of 40%, and the resulting children represent the *individuals* of the new population.

2.2.4 Mutation function

We perform a mutation operation to introduce random changes in the bit string. With a 10% mutation rate, each bit will be changed from a 1 to a 0, and vice versa. Let's say we randomly select the fifth *gene* for mutation:

Original solution: 1010010110

The mutated bit string becomes:

Mutated solution: 1010|1|10110

In this mutated bit string, the vertical bar (|) indicates the location of the mutation. Mutation is applied to each solution, and the resulting solutions compose the new population.

2.2.5 Elitism, or selection of the fittest

After each operation (crossover, mutation, ...), the pre-operation population is compared with the post-operation population, and in case the best solutions of new population are worse than the best solutions of the old population, the worst solutions of the new population are replaced by the best solutions of the old population. Formally:

If best(S_{new}) < best(S_{old}), where S_{new} and S_{old} represent the populations after and before the operation respectively, then replace the worst solutions of S_{new} with the best solutions of S_{old}.

2.2.6 Generalizing the operations

We group all the operations into one algorithm, which we can run any number of times. Per iteration, the algorithm updates the best solution whenever one is found. The total score and weight of the optimal solution are then reported:

$$v_{\max} = \sum_{i=1}^{n} v_{best}, \quad w_{\max} = \sum_{i=1}^{n} w_{best},$$

where v_{best} represents the value of item i in the best solution, w_{best} represents the weight of item i in the best solution, and n is the total number of items.

2.3 Evaluating the results of the algorithm

Running the algorithm with the parameters discussed throughout the report produces a fast optimal solution to the problem. The solution converges to an optimum in around 100 iterations. The optimal solution looks as follows:

item	0	1	2	X	X	5	X	X	8	X
value	2	3	4	X	X	4	X	X	4	X
weight	3	1	5	X	X	7	X	X	9	X

$v_{max} = 17$, and $w_{max} = 25$.

3 The Traveling Salesman Problem

3.1 Representation of the problem

The version of the problem discussed throughout this report is as follows: 10 European cities i with x and y coordinates corresponding to the real world coordinates of the cities are selected. Additionally, for testing purposes, the coordinates of each city on a unit circle are given. The goal of the problem is to find the ordering of the 10 cities such that the distance traveled along that path is minimized. A randomly selected set of 10 cities is used throughout the report:

city	Paris	Berlin	London	Madrid	Rome	Amst.	Lisbon	Prague	Vienna	Stock.
lat.	48.85	52.52	51.51	40.42	41.90	52.37	38.72	50.08	48.21	59.33
long.	2.35	13.41	-0.13	-3.70	12.50	4.90	-9.13	14.44	16.37	18.07
unit x	1	cos(36°)	cos(72°)	cos(108°)	cos(144°)	-1	cos(216°)	cos(252°)	cos(288°)	cos(324°)
unit y	0	sin(36°)	sin(72°)	sin(108°)	sin(144°)	0	sin(216°)	sin(252°)	sin(288°)	sin(324°)

3.2 Specifics of the algorithm

3.2.1 Generating an initial population

Each *individual* in the algorithm is a set of 10 cities along with their coordinates, in a randomized order. We made the choice not to represent this as a *bitstring* due to the complexity of the problem. An example of an initial solution is given below:

Solution	**Example of a random initial population**
Sol 1	Pa, Be, Vi, Ma, Ro, Am, Li, Pr, Lo, St
Sol 2	Be, Lo, St, Ma, Ro, Li, Am, Pr, Vi, Pa
Sol 3	St, Pr, Pa, Lo, Ro, Am, Vi, Be, Li, Ma
Sol 4	Am, Pa, Ro, Ma, Vi, Be, Li, Pr, St, Lo
...	...
Sol n_{-1}	Pr, Be, Ro, Lo, Ma, Am, Vi, St, Pa, Li
Sol n	Ro, Pr, Am, Vi, Li, Ma, Lo, Be, Pa, St

Table 1: Initial Population (City Names)

3.2.2 Fitness of individual solutions

Each *individual* is a randomly ordered combination of 10 cities with a fixed total distance. We calculate the total *euclidian distance* of each solution as follows:

$$D_i = sum_{i=1}^{n} \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2} + \sqrt{(x_n - x_1)^2 + (y_n - y_1)^2}$$

where n is the number of cities in the path, x_i is the x-coordinate of city i, and y_i is the y-coordinate of city i. The equation has two parts: the first part calculates the total distance between the first city and the last city, and the second part calculates the distance between the last and the first city, ensuring the path forms a closed loop. The score of a solution s_i is inversely proportional with the length of a solution, and is computed as follows:

$$s_i = \frac{1}{1 + D_i}$$

The probability of a solution being selected for the next *population* is computed as follows:

$$P(x_i) = \frac{s_i}{S}, \quad \text{for } i = 1, 2, \ldots, N$$

where N is the size of the population, s_i is the score of a solution, S is the total score of the population. A new population is constructed based on these probabilities.

3.2.3 Ordered crossover function

The GA uses ordered crossover, due to the nature of the problem. *Parents* P_i are selected two-by-two, and a random segment of *genes* is selected from P_1. A *child* C_0 is generated, with the segment selected in P_0. The missing *genes* are then transferred from P_1 in the order in which they appear in P_1. The same procedure is applied to obtain C_1. An example is given below:

Parent 1: Pa, Be, Lo, Ma, Ro, Am, Li, Pr, Vi
Parent 2: Be, Lo, Pa, Ma, Ro, Li, Am, Pr, Vi

Random segments are taken from $P_{0,1}$:

Parent 1: ?, ? | Lo, Ma, Ro, Am | ?, ?, ?
Parent 2: ?, ? | Pa, Ma, Ro | ? , ?, ?, ?

The *children* inherit the genes from the other *parent* in order:

Child 1:Pa, Be | Ma, Ro, Li, Am | Pr, Vi
Child 2:Be, Lo | Pa, Ro, Am, Li | Pr, Vi

3.2.4 Mutation function

With a set rate of 20%, every city in a *solution* is "swapped" with another city from the *solution*. An example is given below:

S: Paris, Berlin | London | Madrid, Rome, Amsterdam | Lisbon | Prague, Vienna, Stockholm

London and Lisbon are selected for mutation. The mutation operation results in:

S': Paris, Berlin | Lisbon | Madrid, Rome, Amsterdam | London | Prague, Vienna, Stockholm

The bars represent the cities selected for *mutation*.

3.2.5 Elitism

In case a full iteration of all the previously defined operations does not produce a better solution, the worst solution of the new population is replaced by the best solution of the old population, this ensures convergence. Formally:

If $\text{best}(S_{\text{new}}) < \text{best}(S_{\text{old}})$, where S_{new} and S_{old} represent the populations after and before the operation respectively, then replace the worst solutions of S_{new} with the best solutions of S_{old}.

3.3 Evaluating the results of the algorithm

Running the algorithm with the parameters defined above produces an optimal solution to the TSP quickly. The algorithm takes around 1000 iterations for the solutions to converge at the optimum. An overview of the optimal solutions is also shown below on a unit circle.

city	Stockholm	Berlin	Prague	Vienna	Rome	Madrid	Lisbon	London	Paris	Amst.
x coord	59.33	52.52	50.08	48.21	41.90	40.42	38.72	51.51	48.86	52.37
y coord	18.07	13.41	14.44	16.38	12.50	-3.70	-9.13	-0.13	2.35	4.90

$$D_{\max} = 81.465$$

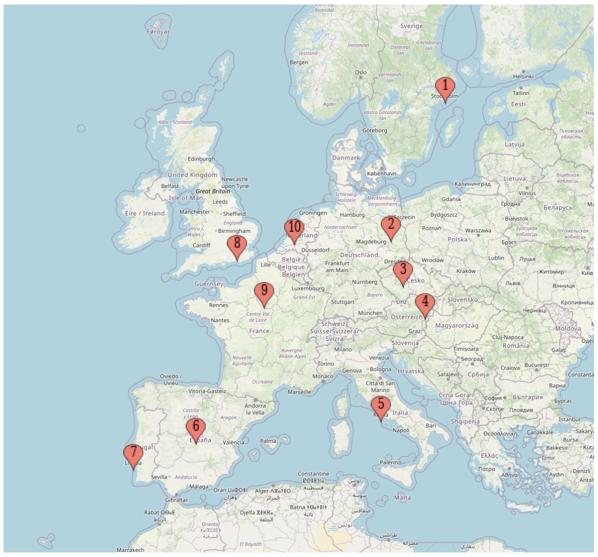
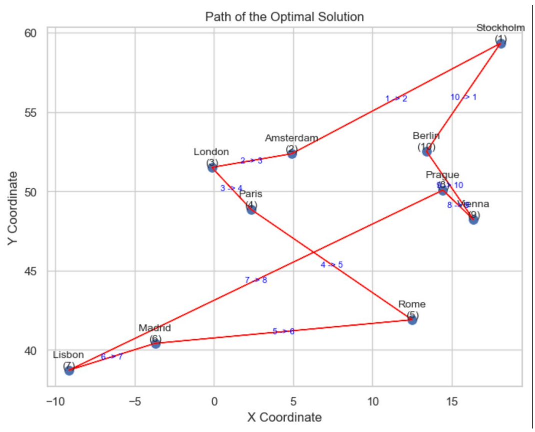


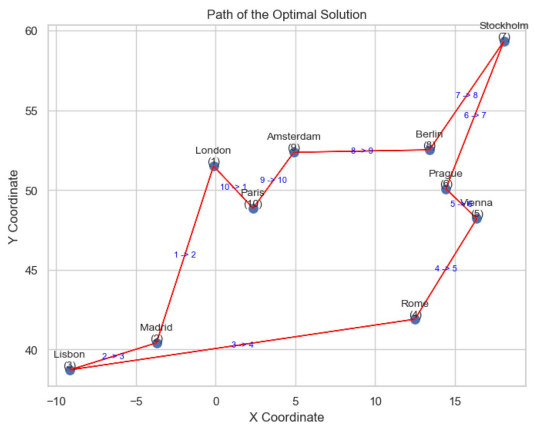
Figure 1: Optimal Path

$$D_{\max} = 81.465$$

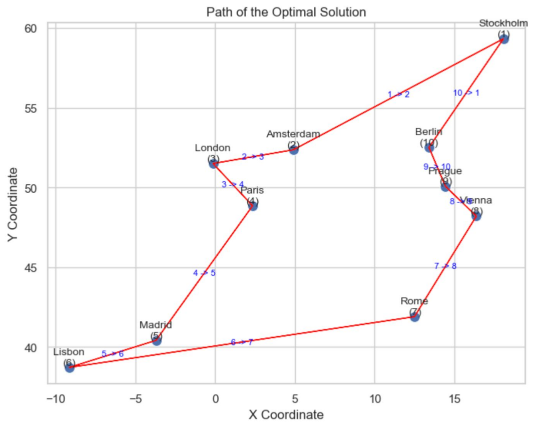
4 Appendix



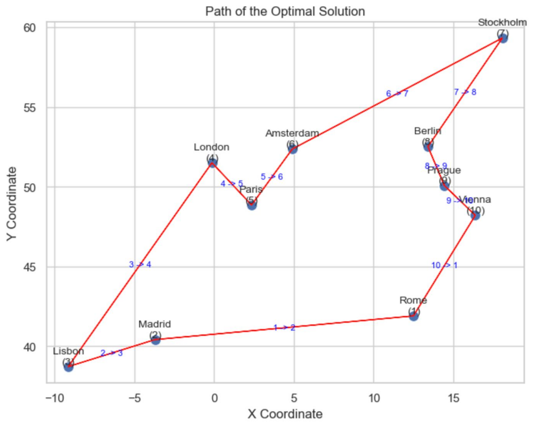
(a) 1 iteration, $D = 100.24$



(b) 100 iterations, $D = 83.97$

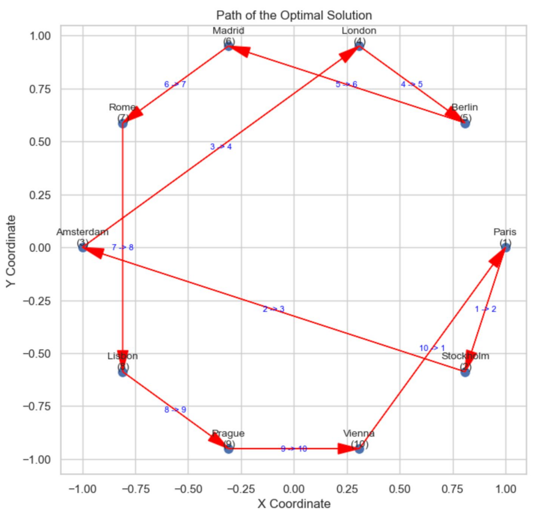


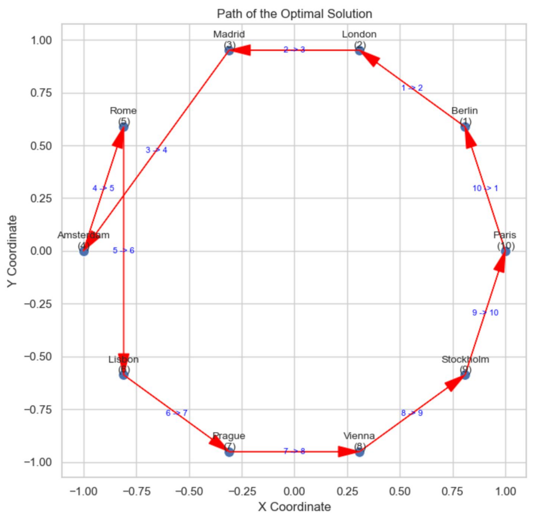
(c) 500 iterations, $D = 82.57$

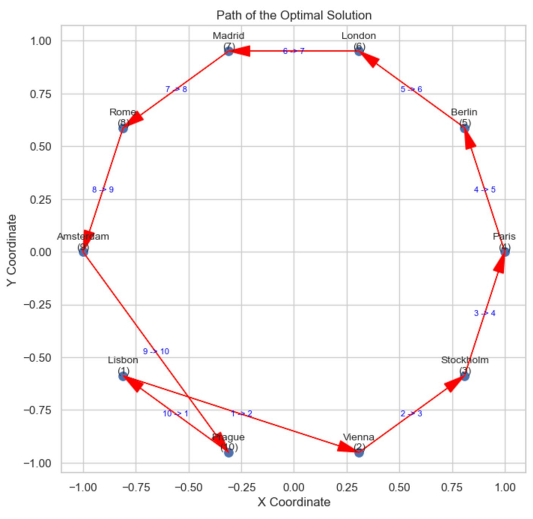


(d) 1000 iterations, $D = 81.46$

Figure 2: Total Distance per Number of Iteration


(a) 1 iteration, $D = 10.14$


(b) 100 iterations, $D = 7.29$


(c) 500 iterations, $D = 7.29$

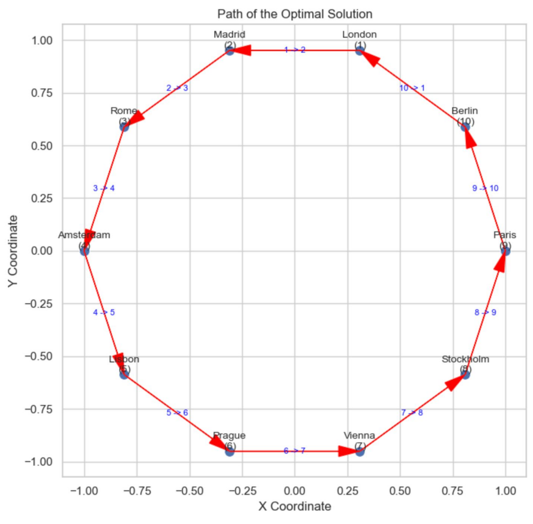

(d) 1000 iterations, $D = 6.18$

Figure 3: Total Distance per Number of Iteration (Unit Circle)

References

[Bhand] Arpit Bhayani. Genetically solving the age old knapsack problem, nd. [Accessed on March 3, 2024].

[Dav] Davis. Order crossover. pdf.

[PC98] J.E. Beasly P.C. Chu. A genetic algorithm for the multidimensional knapsack problem. *Journal of Heurstics*, 4:63–86, 1998.

[She23] Ramez Shendy. Traveling salesman problem (tsp) using genetic algorithm (python), 2023. [Accessed on March 5, 2024].

[Tiw19] Satvik Tiwari. Genetic algorithm: Part 3 — knapsack problem, 2019. [Accessed on March 4, 2024].

[Bhand] [PC98] [Dav] [Tiw19] [She23]