Genetic algorithms applied to Knapsack and TSP

Alvaro Menendez, Joaquín Monedero

February 2024

Introduction

This document shows our approach to (approximately) solving the 0-1 Knapsack and Travelling Salesman Problem (TSP) using a Genetic Algorithm (GA). We will justify the decisions made during the process, and display some results. We presume the reader has some prior knowledge, so if you're unfamiliar with these topics, consider consulting Wikipedia or other resources for more information.

0-1 Knapsack

Our 0-1 knapsack problem is defined as following:

"Given a maximum weight and a Dictionary of weights and values where each entry corresponds to an item, pick the items that maximize the total value without exceding the maximum weight"

A knapsack problem of 10 items might look like this:

```
{0: (1, 10), 1: (2, 4), 2: (3, 7), 3: (4, 8), 4: (5, 6), 5: (6, 5), 6: (7, 4), 7: (8,
    5), 8: (9, 10), 9: (10, 4)}
```

For convenience, the usefulness (key of the dictionary) of each item is set to 1,2,3...n and the weights (values of the dictionary) are randomly set between 1 and 0. Our individual's **chromosomes** are bitstrings of size n (where n is the number of items in the problem) indicating whether that item has been picked or not.

Fitness function

The first idea that we implemented was adding up the value of the items in each individual and setting the fitness to 0 if the weight exceeded the maximum available. This idea works good for trivial problems but as the size of the problem increases, its effectiveness decreases. The two main problems of this approach are the following:

1. **It does not take into account how full the bakcpack is.**
2. **If an individual is off by just a unit of weight, the whole solution is discarded.**

To handle these issues, we added a **penalty** depending on how much they exceeded the weight, and a **reward**, depending on how much weight there was left. [1]

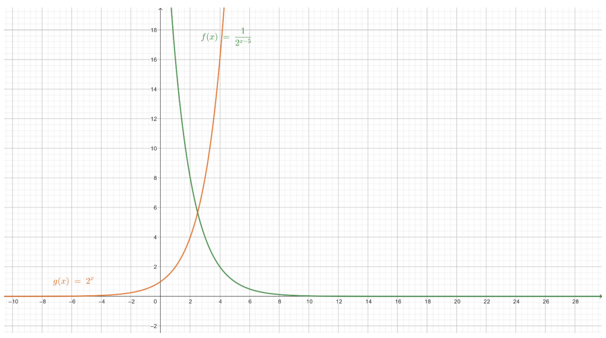

Figure 1: Reward(orange) and penalty(green) functions

[1] Actually, the reward is not needed as we are maximizing the value and automatically the weight will increase. But we realized this too late

Design choices & Implementation

For this problem, we decided to use a **elitist selection** of the 20% individuals with the highest fitness value, which will become the parents of the next generation. The new generation is created by randomly picking two parents and employing a **single point crossover** method to generate a new individual. The stopping condition of the algorithm is not finding an improvement in the maximum fitness within 50 generations.

Results

Our genetic algorithm correctly improves in every generation and eventually gets to a near optimal solution. For a knapsack problem with **100 elements**, **maximum weight of 30** and **mutation rate of 0.3** our GA gets a value of **1319**. Here is a graph showing the performance of the algorithm over generations. The average fitness of every generation is shown as well as the maximum

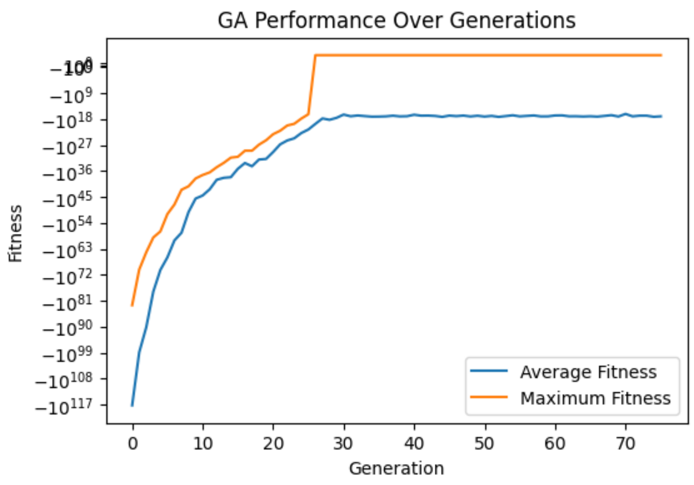

Figure 2: GA performance over generations

With the same initial conditions, but changing the mutation rate from 0.01 until 1, we get the following:

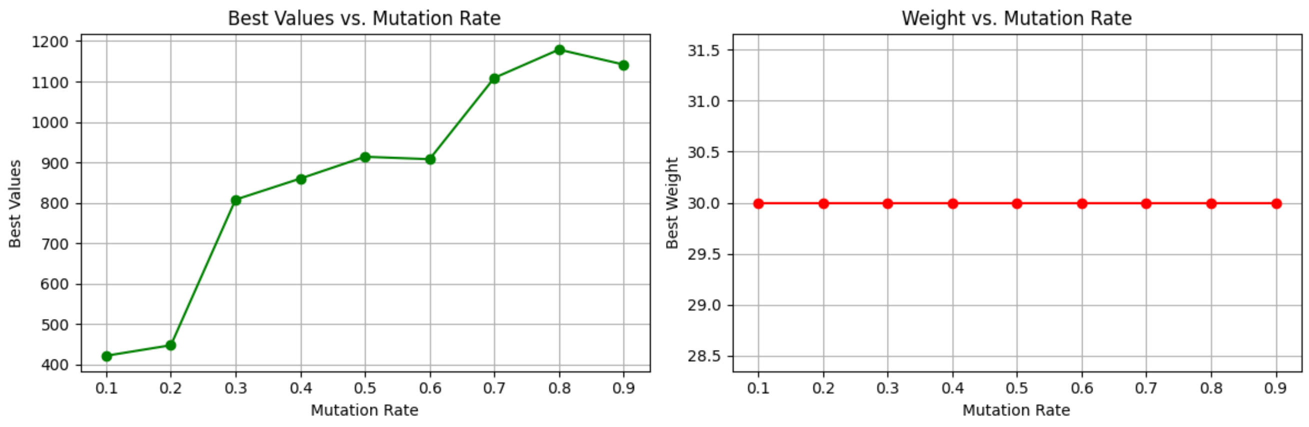

Figure 3: Best values and Weights for different mutation rates

As we can see, as the mutation rate increases, the best value found becomes better. This is because in this particular problem, a source of randomness is important, otherwise the GA can get stuck in suboptimal points. We can also see that the weight of the backpack remains the same for every case

1 Travelling Salesman Problem (TSP)

The travelling salesman problem (TSP) asks the following question:
"Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?".

In our case, the input of the GA is a 2D symmetric matrix representing the distances between cities, that was calculated given a list of points in the X-Y plane. An input example of 5 cities where all the cities lie on circunference points looks like the following:

$$\begin{bmatrix} 0.00 & 1.18 & 1.90 & 1.90 & 1.18 \\ 1.18 & 0.00 & 1.18 & 1.90 & 1.90 \\ 1.90 & 1.18 & 0.00 & 1.18 & 1.90 \\ 1.90 & 1.90 & 1.18 & 0.00 & 1.18 \\ 1.18 & 1.90 & 1.90 & 1.18 & 0.00 \end{bmatrix}$$

1.1 Design choices & implementation

An individual's (or also called tour) **chromosome** is just a list that contains the order in which the cities are visited. The **fitness function** of a tour is simpler than in the previous problem, and it is the *negative* total distance covered, because we are maximizing this value in the GA. The **mutations** in a tour are computed by randomly swapping the visiting order of two cities. An important thing we noted, is that compared to the previous knapsack problem, a lower mutation rate helps the algorithm converge quicker. Again, we implemented an **elitist selection** of the 50% individuals with the highest fitness.

1.2 Crossover

The crossover was the hardest to come up with, as the solution was not intuitive. Single point crossover methods do not work well in this problem as they usually produce invalid chromosomes. We then implemented an **Ordered crossover** heavily inspired by [1], which combines parts of two parent tours while maintaining the relative order of cities from each parent. An image displaying how the crossover works can be seen below.

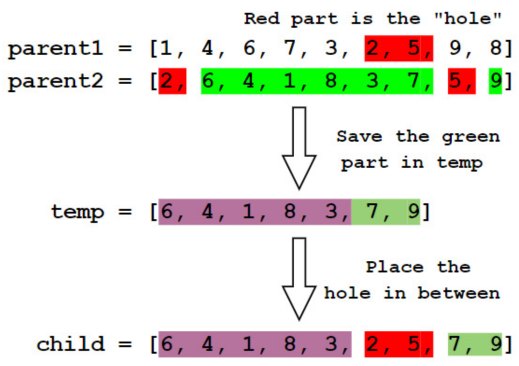
Red part is the "hole"
parent1 = [1, 4, 6, 7, 3, 2, 5, 9, 8]
parent2 = [2, 6, 4, 1, 8, 3, 7, 5, 9]
Save the green part in temp
temp = [6, 4, 1, 8, 3, 7, 9]
Place the hole in between
child = [6, 4, 1, 8, 3, 2, 5, 7, 9]

Figure 4: Ordered crossover implementation

See the Appendix for the implementation.

Results

Our GA, with **20 cities**, a **mutation rate** of 0.1, a **generation size** of 100 correctly improves every generation and eventually gets to an optimal solution. This can be easily checked in the two examples below:

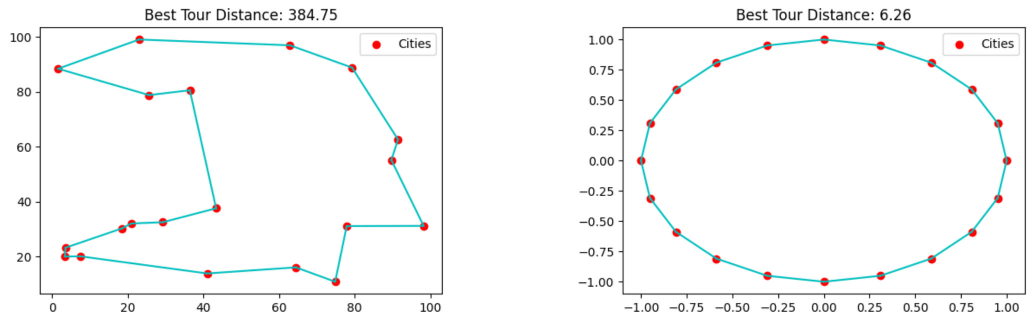

Figure 5: Two examples of a TSP problem solved by our GA

In the following two graphs we can see how the GA correctly improves in every generation, as well as how the mutation rate conditions the GA to converge into an optimal solution or not. Note how, as opposed to the previous knapsack problem, as the mutation rate increases, the solution becomes worse (We are minimising the distance). This might be because, since TSP solutions have a strong dependency on the order of cities, a small change can significantly increase the total distance, making the mutation worsen the solution.

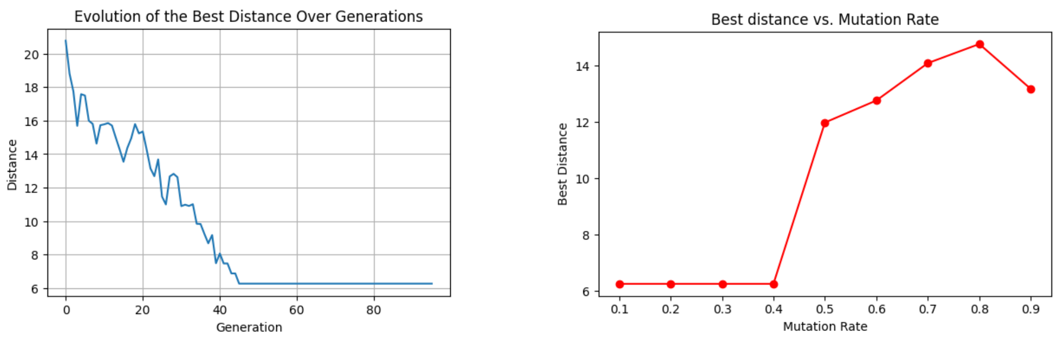

Figure 6: Improvement of our GA and Mutation rate change impact

Finally, we plotted for both the Knapsack and the TSP GA's, the number of generations needed against the input sizes:

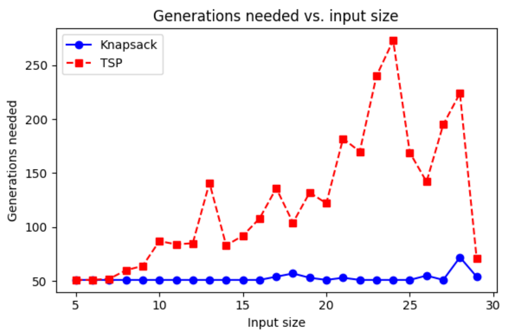

Figure 7: Generations needed against input size

There are many takeaways we can get from this graph: The first one is that there is no input size where the generations needed are less than 50. This is because of the stopping condition we use for both GA is that there has been no improvement over the last 50 generations, meaning in many of the cases, convergence has been reached earlier. We did not overcomplicate as it will arrive to the solution anyway, nevertheless this can be improved in future work. We can also see that the knapsack problem needs much less generations overall, this is because the problem itself is much simpler. Finally, we can see that the knapsack has a smooth function and TSP has many spikes. This is because TSP has much less mutations, so there are some times it arrives to the solution by pure luck while some other times it gets stuck. This implies that some a better mutation method could be made.

Appendix

```python
def ordered_crossover(parent1, parent2):
    size = len(parent1)
    a, b = sorted(random.sample(range(size), 2))
    hole = parent1[a:b]
    child = [city for city in parent2 if city not in hole]
    return child[:a] + hole + child[a:]
```

References

[1] Pedro Larranaga et al. “Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators”. In: *Artificial Intelligence Review* 13 (Jan. 1999), pp. 129–170. DOI: `10.1023/A:1006529012972`.