ARCADA

Solving a Traveling Salesman Problem with a Genetic Algorithm using the Partially Mapped Crossover Operator.

Maskininlärning och optimering

John Wickström & Mattias Kallman

TABLE OF CONTENTS

1 PROBLEM FORMULATION

1.1 Introduction

The traveling salesman problem (TSP) aims to find the shortest distance between a list of nodes with the constraint of visiting each node exactly once before returning to the initial node. TSP is a relevant problem for today as it is of interest to many different types of companies (Wikipedia*: Traveling salesman problem* 2020). Example applications of TSP include optimizing travel routes for logistics companies and map services such as Google Maps, that return optimal routes to its end users.

TSP is a non-linear exponential problem, which implies that the number of possible solutions grows exponentially with the number of nodes in the network (Wikipedia*: Traveling salesman problem* 2020). In fact, the possible solutions for TSP is a factorial of *n* where *n* is the number of nodes. To illustrate its implications, if we have factorial of 12 nodes, we get 479 001 600 possible solutions. Compare that to factorial of 13 nodes, we get 6 227 020 800 possible solutions. In our use case, that is present in section 1.3, we chose to use 15 nodes, making it a very complex problem with $\sim 1.3\ e^{12}$ possible solutions. For this reason, it is advantageous to use a heuristic method to find a good solution rather than brute forcing an optimal one (brute force refers to the process of calculating every possible solution).

TSP is defined as a NP-hard problem and also belongs to the problem class of NP-complete making it good problem solve with a genetic algorithm (Wikipedia*: Traveling salesman problem* 2020).

1.2 Our Problem Formulation

We chose to illustrate the traveling salesman problem by trying to find the optimal route between 15 cities in Finland. 15 cities were chosen because we wanted to make the problem difficult enough so that LP solve could not feasibly find a solution within a reasonable time. Our primary goal is to build a genetic algorithm with the partially mapped crossover method to find an optimal route.

1.3 Partially Mapped Crossover

PMX is a variation of the two-point crossover operator. The difference between the operators is that PMX guarantees that the list integrity is maintained by checking the middle segment that is swapped between parents. The mutation process acts according to a specific ruleset that guarantees that the resulting list does not include duplicate values. (Wikipedia: *Crossover* 2020)

1.4 Mathematical Formula of TSP

There are many ways to formulate TSP as an integer linear program. Some of the more notable formulations are the Miller–Tucker–Zemlin (MTZ) formulation and Dantzig–Fulkerson–Johnson (DFJ) formulation. Below is presented the DFJ equation. (Wikipedia*: Traveling salesman problem* 2020)

Cities are labeled with numbers 1, …, n and defined as:

$$x_{ij} = \begin{cases} 1 & \text{the path goes from city } i \text{ to city } j \\ 0 & \text{otherwise} \end{cases}$$

For $c_{ij} > 0$ is the distance from city *i* to city *j*. TSP can be written as the linear programming problem seen in Figure 1. The last constraint in the equation ensures that their no sub-tours.

$$\begin{aligned} &\min \sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} c_{ij} x_{ij}: \\ &\sum_{i=1, i \neq j}^{n} x_{ij} = 1 && j = 1, \ldots, n; \\ &\sum_{j=1, j \neq i}^{n} x_{ij} = 1 && i = 1, \ldots, n; \\ &\sum_{i \in Q} \sum_{j \neq i, j \in Q} x_{ij} \leq |Q| - 1 && \forall Q \subsetneq \{1, \ldots, n\}, |Q| \geq 2 \end{aligned}$$

Figure 1: Dantzig–Fulkerson–Johnson formulation

2 IMPLEMENTATION

Our implementation attempts to solve the traveling salesman problem with a genetic algorithm that uses the partially mapped crossover (PMX) operator. PMX is suitable for solving TSP as it maintains list integrity while it creates permutations of the original pair, i.e., it will not create solutions that break the constraints of prohibiting sub-routing and visiting the same node (city) more than once. Our algorithm was implemented with a Python program using the following pseudo code as seen in Figure 2.

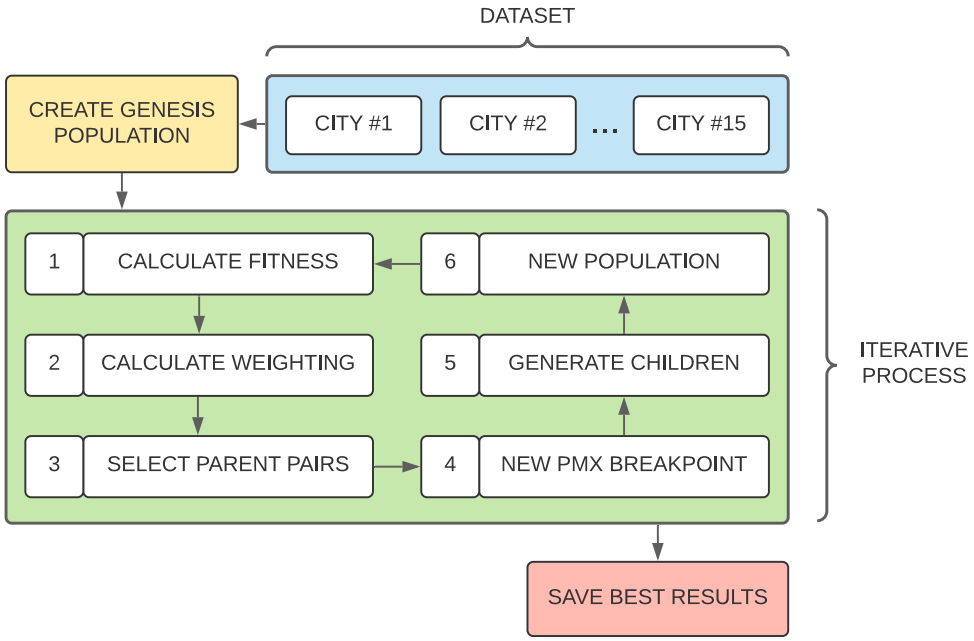

Figure 2: The implementation visualized.

2.1 Data Scraping

It is worth noting that we were unable to find a good data table for city distances in Finland containing 15 or more cities. We overcame this obstacle by building a data scraping script to fetch the distances and thus simplifying the data gathering process. The reason for this is because as the list of cities become longer the amount of data needed grows exponentially. The amount of data points needed can be expressed as:

$$n^2 - n, where\ n > 1\ and\ integer.$$

In our use case, n is equal to 15, which results in 210 different distances. The data in Figure 3 shows the relational distance between the cities. We used a service that gave us

the flight path (i.e., a straight line) between the cities. Getting the driving distance between cities was impractical as the service we used gave zero distance between cities that were too close to each other. Helsinki – Vantaa is one example of this. This has no practical implications on our implementation as the data points don't affect the theory, but only the results.

	helsinki	espoo	tampere	vantaa	oulu	turku	kotka	lahti	kuopio	pori	kouvola	joensuu	lappeenranta	mikkeli	vaasa
helsinki	0	16	160	15	539	150	114	99	336	225	124	373	203	211	370
espoo	16	0	151	24	536	134	128	103	339	211	134	381	215	217	359
tampere	160	151	0	150	400	142	204	116	254	106	171	335	240	186	210
vantaa	15	24	150	0	525	153	105	84	321	220	111	359	191	196	360
oulu	539	536	400	525	0	533	511	448	259	433	465	341	460	380	284
turku	150	134	142	153	533	0	255	194	394	118	246	463	328	302	296
kotka	114	128	204	105	511	255	0	89	273	299	46	281	95	137	404
lahti	99	103	116	84	448	194	89	0	237	215	58	281	136	116	316
kuopio	336	339	254	321	259	394	273	237	0	343	231	111	206	135	307
pori	225	211	106	220	433	118	299	215	343	0	272	434	345	291	180
kouvola	124	134	171	111	465	246	46	58	231	272	0	251	83	96	363
joensuu	373	381	335	359	341	463	281	281	111	434	251	0	190	164	417
lappeenranta	203	215	240	191	460	328	95	136	206	345	83	190	0	85	410
mikkeli	211	217	186	196	380	302	137	116	135	291	96	164	85	0	331
vaasa	370	359	210	360	284	296	404	316	307	180	363	417	410	331	0

Figure 3: Distances matrix of Finnish cities.

2.2 Observations

2.2.1 Breakpoints

PMX breakpoints determine where the segmentation of parents occurs. We discovered that using a static breakpoint would frequently halt the progress of a generation as seen in Figure 5.

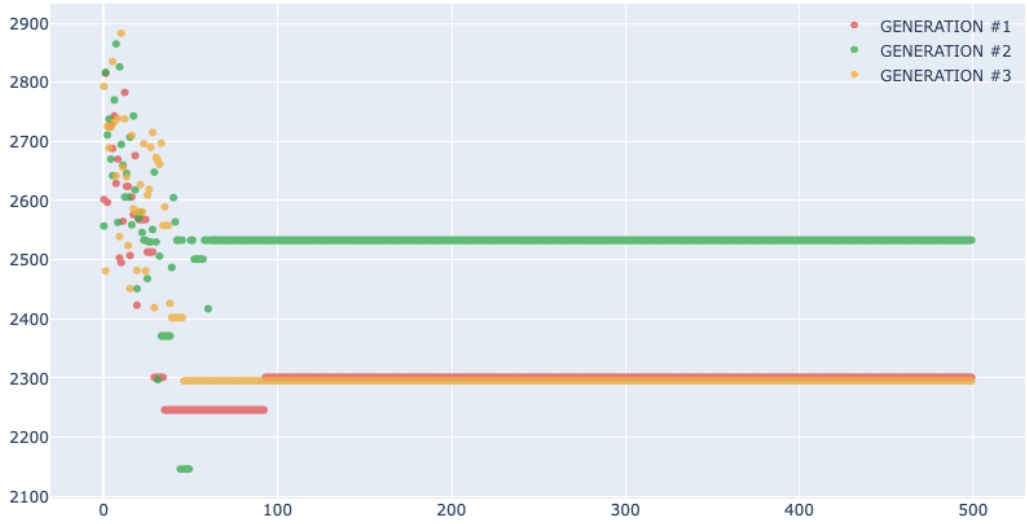

Figure 4:Using a static PMX breakpoint for each iteration.

We solved this by randomizing the breakpoint for each iteration of a generation. The algorithm randomly selects where the middle segment of the PMX operator starts and ends. This acts as a form of mutation within the algorithm which protects it from genetic stagnation, resulting in improved results.

Figure 5: Using a randomized breakpoint for each iteration.

2.2.2 Initial Starting Population Size

During development, we also noticed that the initial population size greatly affected how quickly the genes of children started to stagnate. Figure 5 presents three generations with the initial population size of 50 that quickly flatlined and were unable to improve.

Figure 6: The population size is set to 50.

To overcome this limitation, we increase the initial population size to 500. Results can be seen in Figure 6. It appears that population siz--e is important when applying a genetic algorithm, but the number has diminishing returns and must be determined heuristically.

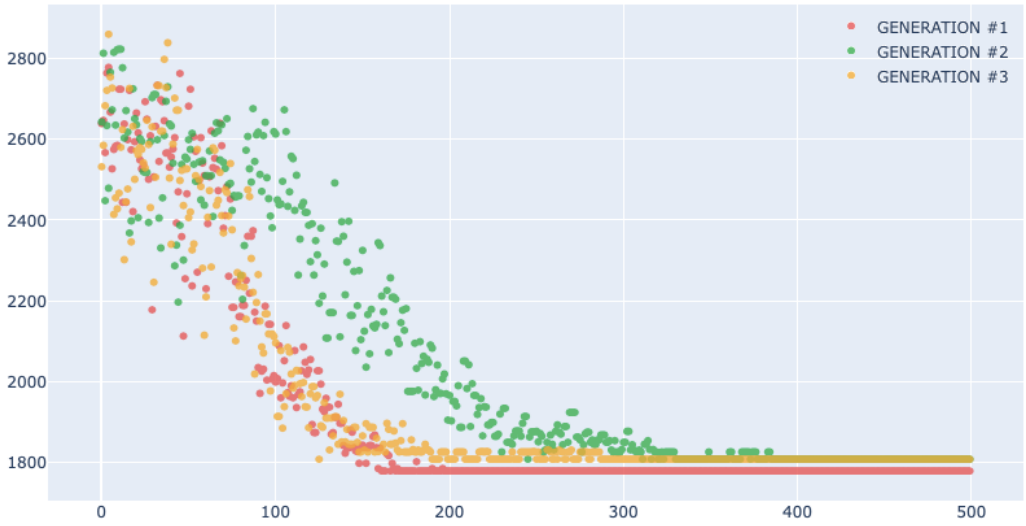

Figure 7: The population size is set to 500.

3 CONCLUSION

This work shows that a genetic algorithm that uses the partially mapped crossover operator can find the optimal route between 15 waypoints in a reasonable amount of time. Mutation is an effective way of preventing the stagnation of a generation's gene pool, but this type of problem does not support a mutation in the traditional sense. However, we managed to incorporate a similar effect by generating new PMX breakpoints for each iteration of a generation's evaluation. A generation's initial population size can also determine how quickly an optimal solution is found as demonstrated in section 2.3.2. While a larger population generally yielded better results, a more efficient method of achieving the same result was to instead maintain a reasonable population size but increase the number of generations.

The optimal distance for this specific problem is 1 779 km. Assuming our start location is Helsinki the optimal route is:

Helsinki – Espoo – Turku – Tampere – Pori – Vaasa – Oulu – Kuopio – Joensuu – Mikkeli – Lappeenranta – Kotka – Kouvola – Lahti – Vantaa – Helsinki

The solution for this problem is cyclical, which means that the number of optimal solutions will be equivalent to the number of nodes. All optimal solutions have the same collective distance and the only differing factor is which city comes first.

As this was a heuristic process, it does not guarantee an optimal solution. Further research and development could be aimed at improving the performance of the algorithm. We are unsure whether our algorithm can be applied on a grander scale without requiring significant modification. Since increasing the number of generations seems to be a more effective method of finding good solutions, parallel processing seems like a reasonable next step for the implementation.

REFERENCES

Wikipedia*: Traveling salesman problem*. (2020). [online] Available at: https://en.wikipedia.org/wiki/Travelling_salesman_problem [Accessed 8 Dec. 2020].

Wikipedia: *Crossover*. (2020). [online] Available at: https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm) [Accessed 8 Dec. 2020].