The Genetic Algorithm for the Traveling Salesman Problem

By Moro Bamber, Ben Brown, and Josh Stapera
For CSCE A405 Artificial Intelligence
Dr. Masoumeh Heidari Kapourchali

Introduction

The traveling salesman problem (TSP) is a classic NP-complete problem with many attempted solutions. Existing solutions include the nearest neighbor algorithm and the Christofides algorithm, both of which are non-optimal. The goal of this project is to attempt a solution approach using genetic algorithms. Currently, there is limited literature attempting to solve this problem using genetic algorithms, as a result, some parts of this project are novel in their implementation. The intended result of this goal is a genetic algorithm that is able to produce a consistent close-to-optimal solution for any given TSP.

Literature Review

The current literature was reviewed in order to gain a more complete understanding of what approaches to genetic algorithms currently exist and how they are implemented. Additionally, it provided a source of inspiration for the creation of our own operators that were intended to outperform other operators in certain areas. In this review, the basis of genetic algorithms were researched and their use cases. It was determined that there are no true use cases for genetic algorithms and that a majority of uses are experimental at best. Furthermore, a number of selection, crossover, and mutation operators were investigated and filtered based on feasibility within the time constraints of the project. These operators were then further researched until we were able to develop our own pseudocode or we found an article providing a pseudocode implemented. This did result in some issues for some operators as the pseudocode was unclear and required referring to non-scholarly articles for more clear and effective pseudocode. This review also included a search for any literature discussing solving the traveling salesman problem using genetic algorithms. Scholarly results were limited and sometimes confusing to follow, as a result, these articles were not referenced in this report.

C++ Implementation

C++ was chosen as the programming language for this implementation because of its speed at runtime and our prior familiarity with its syntax. While it is a more low level language than most of the solutions for this problem we found, it provided us more insight into the underlying process behind different GA operations. We downloaded a library of TSP problems from TSPLIB, which served as our testing problems. On a Linux system, a user is able to make the

project using a makefile, then run the problem given parameters specified when executing the outfile. There are 40 unique permutations of GA operator combinations available to use, and combined with the infinite number of choices for maximum generations allowed, and population size, it makes for a very large search space to find optimal parameters for a given problem.

Objects and Encoding Scheme

Evolutionary algorithms can be encoded in a variety of ways. One of the most common methods is the binary encoding, where each gene is a series of 1's and 0's. This makes the gene easy to work with, and operations generally run quite fast on them. The TSP cannot be encoded in this way due to the size and complexity of encoding the paths into 1's and 0's. Instead, we must use a permutation encoding, or an ordering scheme, where we have unique elements we must put in a particular order for a desirable result [1]. To accomplish the task of encoding the TSP in C++, we used two objects: City and Trip. City keeps track of a node's coordinates in euclidean space as well as its unique ID, and Trip contains a vector of Cities. Trip has getter methods for the calculation of path length and retrieval of other attributes, a singular trip can be thought of as a gene.

Selection Operators

Selection is a crucial aspect of the TSP. We want to avoid gene convergence early on, while also trying to breed the better genes, this is a delicate balance that is hard to accomplish. We experimented with a few different ways to achieve this.

Roulette Wheel Selection (RWS)

This selection technique is based on the premise that fitter genes have a higher likelihood of being chosen for breeding. Essentially, the algorithm takes the inverse path length, and uses the sum of those inverse path's to create ranges between 0 and 1. Shorter paths, then, have larger ranges. We then choose a random number between 0 and 1, and select the gene that falls in that range. This is done until we have the desired number of genes to breed. The code for this algorithm is based on the python version found in a medium article [5].

Stochastic Universal Sampling (SUS)

Originally, the code for this was written based on pseudocode in this paper [2] but after noticing a significant difference in the number of mutations occurring between this operator and other selection techniques, we discovered that it was not selecting an appropriate number of parents. We rewrote the algorithm, this time based on the wikipedia article for Stochastic Universal Sampling [4]. The steps are as follows:

1. N = half the size of the gene pool, so if it's 32, then N is 16, it will be the number of parent genes we select.
2. F = Sum of all paths - total sum. So if Path 1 is 5, Path 2 is 10, and path 3 is 7, then F would be 5 + 10 + 7 = 22
3. P = F / N - should be some type of interval that makes sense.

4. Start = select a random number between 0 and P.
5. For [i=0 - I = N-1]: value = i * P, add value to list.
6. Do a Roulette wheel selection on predefined intervals determined by i*P. So if i is 1, then we select the last gene less than P. The next gene we select is 2*P, which means the last gene less than 2*P is selected.

A good way to think about this algorithm is to picture a wheel, and there are arrows pointing to slices at a random interval, but an interval that allows a specific number of genes to be selected.

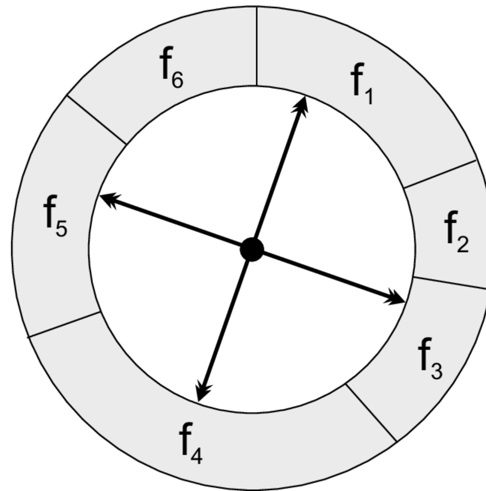


Figure 1: Visualization of Stochastic Universal Sampling

Linear Rank Selection (LRS)

This technique works very similarly to the Roulette wheel selection, however, instead of a fitness score determining range, a gene's rank among other genes determines its probability of selection. The first step is to rank all path's by length, shortest being ranked higher. Then, we compute the probability of each gene with a formula

$$p(i) = (1/n) * (sp - (2*sp -2)*(i-1)/(n-1))$$

Where *sp* is the selection pressure - between 1 and 2, where 2 is high pressure, and 1 is no pressure. *n* is the number of genes in the gene pool, and *i* is the rank of the gene. We then loop through all genes and use the above formula to find its probability, while also creating a cumulative sum of all probabilities to create ranges, as we did with roulette wheel. We then use the same technique of roulette wheel and select a random number between 0 and 1, and select the gene with its range intersecting this number.

Mean Deviation Selection (MDEV)

This technique was developed in-house as an approach to creating genetic diversity at any cost. The idea was originally to use the standard deviation as some type of selection pressure gauge, but finding the deviation of the mean was much easier. It works by calculating a cumulative sum of path lengths in a gene pool, then dividing it by the number of genes to get the mean or

average. The average is then subtracted from each gene's path length and takes its absolute value to get a score of deviation from the average. All of the genes are then ranked based on this score and select the amount of genes we want in ascending order of mean deviation.

Crossover Operators

Crossover operations act as the methods we use to 'breed' two parent genes to produce 1 or 2 children. Ideally we are combining sequences from two fit genes to produce an even fitter child gene. In the TSP, crossover operations are made especially difficult based on the way the problem must be encoded.

Single Point Crossover (SPX)

Single Point Crossover is likely the most common and most intuitive crossover method for the genetic algorithm. The idea is to splice two genes at the same point, and swap sections. Where gene 1 is receiving the second section of gene 2, and gene 2 is receiving the first section of gene 1. This process is very simple for binary encoded problems, but our permutation encoding required some more legwork. The algorithm works as followed:

1. Generate a random number in the range of the number of cities.
2. Splice genes at that point, storing all 4 sections.
3. For path 1, add all cities in the second section of path 2 not already in the first section of path 1.
4. For path 2, add all cities in the first section of path 1 not already in the second section of path 2.
5. Add new combined genes to the children gene pool.

Uniform Crossover (UX)

This operator works by giving every index for a path the same likelihood of being swapped with a city at the same index in another path. In our implementation we are giving each index a 20% chance to swap. This crossover is very simple, but we also must take into account any swap leads to a path to having a duplicate city. To address this, every time a swap takes place, we search both parent genes for the city it just added, and replace it with the city it just gave to the other gene. We also must check if we are at the same index as the swap, or else the operation would do nothing.

Edge Recombination (ERX)

The edge recombination operator creates a path similar to the set of its parents by looking at the edges instead of the vertices. It is based on an adjacency matrix which lists the neighbors of each node in any parent. We wrote the code for this algorithm based on the wikipedia article for it [4]. Of all the operator implementations that are functional, this was easily the most challenging code we wrote, and likely also the hardest to read.

Mutation Operators

Scramble Mutate (SM)

We start by picking a random index, and selecting the cities at that index through to that index plus the mutation length, for example, if we choose 4, and the mutation length is 4, then we are choosing cities at indices 4,5,6,7. We then randomize the order that this subset of cities are in, so if we had 4,5,6,7, we scramble them to 6,4,7,5. Then we place this subset back where they came from starting at index 4. So if before we had [1,2,3,4,5,6,7,8,9], now we have [1,2,3,6,4,7,5,8,9].

Swap Mutate (S)

This is the simplest mutation function we have. All it is doing is picking two random individuals in the path, and swapping their places.

Moro Mutate (M)

Moro Mutate — named after Moro who wrote it — was developed with the idea that for smaller paths, trying to swap adjacent cities could yield good results. It works by doing several different swaps on a path. The amount of swaps is proportional to the length of the problem - specifically ⅓ * problem length. For the amount of swaps, we go through, pick a random index, and swap that city with the next city. For example, if we have 2 swaps, the first one could be: choose index 5, swap city at index 5 with index 6. The next swap could be chosen at index 7, swap city at index 7 and index 8. We have not seen this type of mutation in literature, if we do, we will change the name.

Challenges

The choice to implement the GA in C++ - a lower level language - added difficulty. We thought the language's speed would make testing on larger problems more viable. However, doing anything slightly complicated requires much more forethought and careful programming than if we were using a higher level language like python. It proved to be a challenge to write this implementation. While code exists for this algorithm in C++, we had a hard time finding any that was relevant to what we were trying to do with writing all the various operators. In the process of writing such specific code without any references, we learned how operators work in great detail.

The biggest challenge we faced implementing the partially mapped crossover. The idea of it is simple:

1. Pick a section of two genes to swap with each other, but in the process, map the cities to each other for replacement purposes.
2. Use the mappings to replace any cities that were swapped so that every city is unique.

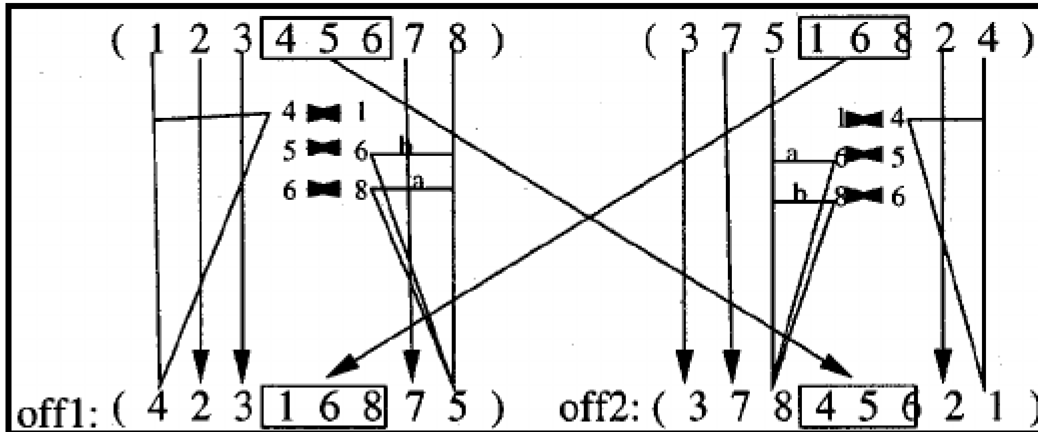


Figure 2: Sample Partially Mapped Crossover Operation

Example:
https://www.researchgate.net/publication/226665831_Genetic_Algorithms_for_the_Travelling_Salesman_Problem_A_Review_of_Representations_and_Operators

We believed we were close to implementing it, even writing it in python before we did it in C++. However, in testing, a mapping issue arose and opened a pandora's box of mapping issues.

In one of our problems, there are 10 cities from around the world. And we noticed that the algorithm did not work for a certain mapping (see appendix for graphical description). This mapping type was not considered when building the algorithm, which gave rise to the possibility of mappings on larger problems that had increasing complexity. This complexity was not something that we could figure out.

We also had a challenge getting the Stochastic Universal Sampling (SUS) method to work. As mentioned in the section on the SUS operation, the pseudocode from the Moroccan paper [2] was confusing and not working properly. We also found the way the technique was described to be inadequate. By using the pseudocode in the wikipedia article for SUS we were able to implement the operator properly [4].

Visualizations

In order to extract and present meaningful insights from the genetic algorithm's implementation on the TSP, we utilized two primary methods for outputting data and visualizing results.

Pygame Visualization

To illustrate the improvement of the best path over successive generations, we developed a Python program using Pygame to visualize the output data from the C++ implementation of the GA. This approach allowed us to showcase the algorithm's progress in an intuitive and accessible way.

We chose to use a separate Python program for visualization because Pygame greatly simplified the development process compared to the alternatives available in C++. Additionally, since the visualization process is far less computationally demanding than the GA itself, implementing it in Python introduced minimal performance drawbacks.

The data flow between the C++ GA and the Python program was facilitated by exporting intermediary CSV files. These files contained all necessary information for the visualization, including details about the TSP dataset, the paths between cities for each generation, the path lengths of every gene in each generation, and the configuration options used for the GA. The configuration options included parameters such as the selection method, crossover type, mutation rate, population size, and generation count.

The Python program used this data to first render the TSP map and display the GA configuration details. It then iterated through each generation, dynamically drawing the path corresponding to the best gene of that generation. This created a step-by-step visualization of the GA's progress over time, highlighting how the algorithm improved the solution.

The visualization clearly demonstrated the effectiveness of the GA. In the initial stages, crossover operations led to rapid improvements by combining the best traits from different individuals. Later in the process, as the algorithm approached convergence, mutations played a crucial role in escaping local optima and refining the solution. The result was a compelling depiction of how the GA evolved to find better paths over successive generations.

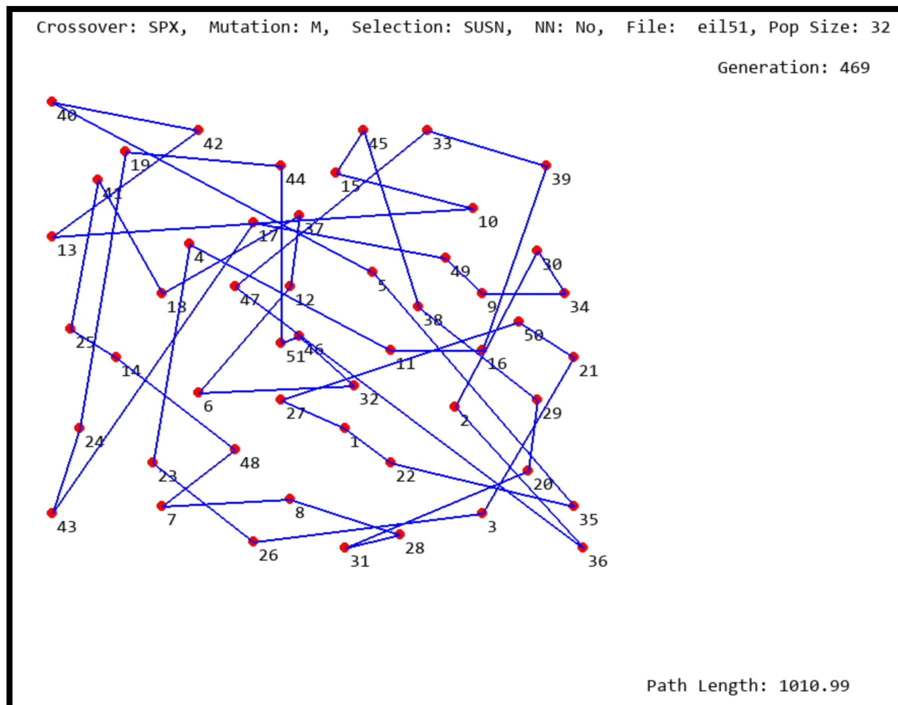


Figure 3: Screenshot of the Pygame visualization.

Excel Graphs

Excel was also used in order to visualize the results from different configurations of the GA. To do this we used the same exported data described above, to draw visual conclusions from the path lengths of all genes of each GA run.

Within these excel graphs we were able to visualize several different main components of the GA runs.

1. How different combinations of crossover, mutation, and selection strategies fared at solving TSP compared with each other.
2. How different likelihoods of mutation impacts the ability of different GA configurations to solve TSP.
3. How different configurations of GA lead to gene convergence over generations.

Below shows a graph visualizing the high convergence over time of the GA, which was a common trend across many different GA configurations we tested. This is an issue for the GA as high convergence limits the possibility of future improvement and avoids being caught in local minima.

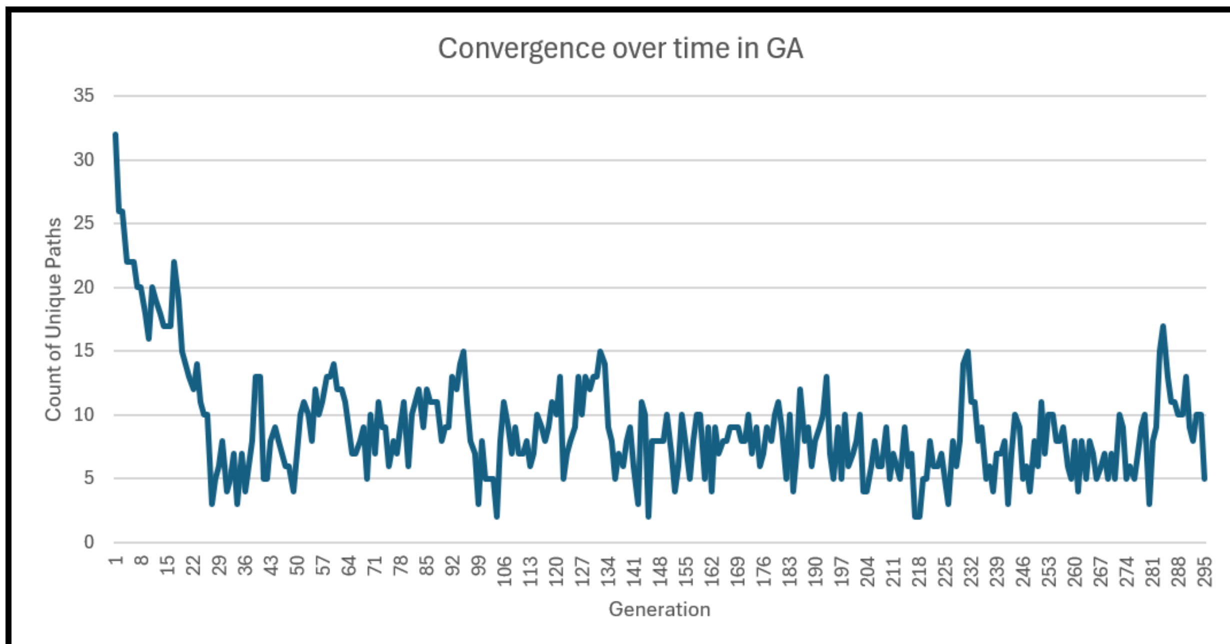


Figure 4: Graph showing the convergence of GA over time (Using Single Point Crossover, Moro Mutate, and Linear Rank Selection)

The results likely show our mutation methods, selection pressure, and population size may be limiting to the success of the GA in solving the TSP

Results

To determine the best ways to use different combinations of operators, we conducted a series of tests. The tests consisted of setting the max generation size and population size to be adequate for the problem length. We then would use different combinations of operators to see the resulting best path length, and time it took for the problem to complete. In larger problems we added a break method. This method stops the run after it has not found a better solution in ⅓ the number of maximum generations, so we also tracked this as well. We chose problems of two different sizes, 10 and 51.

For the problem of size 10 we used 500 generations and a population size of 32 per generation. We saw no large discrepancies between operator combinations (denoted as <[selection], [crossover], [mutation]>) on path length except when using the <RWS, SPX, S>, which yielded a much poorer path length on average. All other combinations we tested converged to near what seems like a global optimum. The metric that stood out in these tests was time taken. We observed that the fastest combination for 10 cities was in fact the worst performing combination of <RWS, SPX, SM>. Followed closely by <LRS, SPX, S>. Notably, <LRS, SPX, S> did converge to the global optimum. We observed that any combination using the Edge Recombination (ERX) operator was very slow compared to all other combinations of operators.

The problem with 51 cities yielded more interesting results. Here we saw more varied path lengths as well as times. We used 1500 generations and 64 as the population size. In these tests we noticed that the Swap mutate was outperforming Scramble and Moro mutate by a significant margin. We also noticed that Mean Deviation Selection was performing fine, but not as well as LRS, or RWS. When not using Nearest Neighbor (NN) before running, we found that <RWS, SPX, S> and <LRS, SPX, S> performed better than all others except <RWS, ERX, S> which at that point had found the best path length without nearest neighbor. As we saw in the problem with 10 cities, ERX was much slower than all other operators, despite performing well with path length. To continue testing, we selected the best combinations without NN and ran them with NN with 3000 generations and 128 population size to try and find the best path length we could find. We found that using NN drastically improved the solutions, and at the same time was triggering a break after not finding a solution for ⅓ the number of generations. In these tests we saw the power of ERX. It found on average 13.5% better solutions than its counterparts with the same selection and mutation operators with SPX. However, while non-ERX solutions ran in 15-20 seconds, ERX would take 5.5 minutes.

Contributions

Moro:

- Developed C++ Implementation of Genetic Algorithm
- Tested different combinations of operators for the algorithm on sample problems
- Actively participated in team planning, analysis, and discussions during weekly / biweekly meetings throughout the semester.

Ben:

- Developed the code to visualize the genetic algorithm using Python, ensuring full compatibility with the C++ implementation of the GA program.
- Conducted analysis of the GA outputs under various selection/crossover/mutation combinations to evaluate performance.
- Investigated the effects of mutation and crossover rate adjustments on the algorithm's efficiency and effectiveness.
- Actively participated in team planning, analysis, and discussions during weekly / biweekly meetings throughout the semester.

Josh:

- Reviewed literature and analyzed potential implementations of selection, crossover, and mutations algorithms along with collecting general information about genetic algorithms
- Wrote portions of the report and proofread the entire report for clarity and cohesion, making edits as required
- Actively participated in team planning, analysis, and discussions during weekly / biweekly meetings throughout the semester.

References

[1] Katoch, S., Chauhan, S.S. & Kumar, V. A review on genetic algorithms: past, present, and future. *Multimed Tools Appl* **80**, 8091–8126 (2021). https://doi.org/10.1007/s11042-020-10139-6

[2] Jebari, Khalid. (2013). Selection Methods for Genetic Algorithms. International Journal of Emerging Sciences. 3. 333-344.

Edge recombination
[3] Wikipedia. “Edge Recombination Operator.” *Wikipedia*, https://en.wikipedia.org/wiki/Edge_recombination_operator.

Stochastic Universal Sampling
[4] Wikipedia. “Stochastic Universal Sampling.” *Wikipedia*, https://en.wikipedia.org/wiki/Stochastic_universal_sampling .

[5] Shendy, Ramez. “Traveling Salesman Problem (TSP) using Genetic Algorithm (Python).” *Medium*, 5 August 2023, https://medium.com/aimonks/traveling-salesman-problem-tsp-using-genetic-algorithm-fea640713758.

Figure 1: Marcot, Pierre
https://www.researchgate.net/figure/Stochastic-universal-sampling_fig9_279488363

Figure 2: Larranaga, Pedro & Kuijpers, Cindy & Murga, R. & Inza, I. & Dizdarevic, S.. (1999). Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators. Artificial Intelligence Review. 13. 129-170. 10.1023/A:1006529012972.

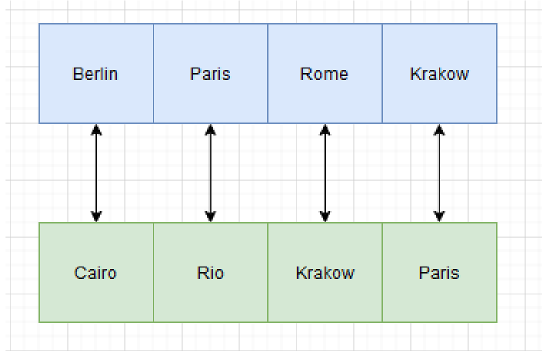
GitHub Repository for C++ Implementation

https://github.com/UnderYourSpell/TravelingSalesmanGA

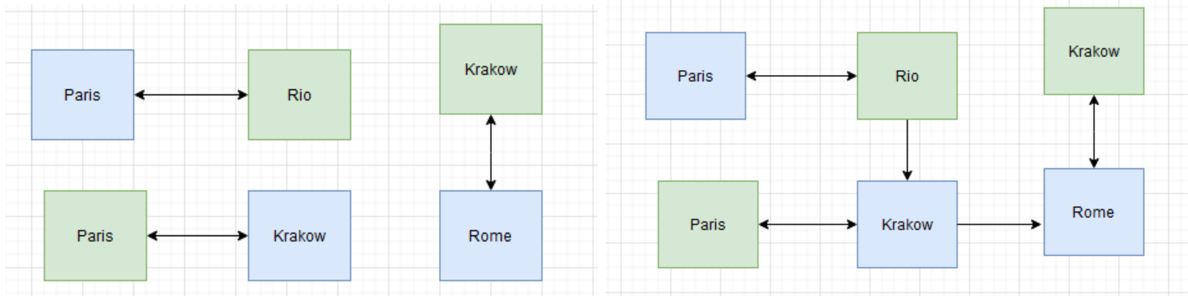
Appendix

Problem with Advanced Mapping

Consider this Partially Mapped Crossover



This is what the mapping looks like:



What should the proper mapping for Rio be?
Rome!

It would be Rome via krakow. While this seems like it can be solved, we were unable to develop a way to do these sophisticated mappings. And with a problem 1000+ cities long, a PMX could contain hundreds of cities, and mappings could get really messy.