1. Genetic algorithms are computer procedures modelled after the concept of natural evolution in nature. They can be used for many applications including artificial creativity and data fitting, but their ideal use case is for optimization problems.

   To use a genetic algorithm to solve a problem, you must first find a way to represent a solution as a set of ”genes” or genomes. Every possible solution will be assigned a score according to some fitness function. With each generation, some solutions will ”die” while the fittest solutions will survive onto the next generation. As time increases, the genomes should converge to an optimal solution.

   Following is a general description of a genetic algorithm.

Genetic Algorithm Terms

(a) **Population** - A set of possible solutions.

(b) **Chromosome** - A set of genes that represents a possible solution. Also referred to as an individual.

(c) **Gene** - Data belonging to a chromosome. Generally represented as one bit (1 or 0).

(d) **Mutation** - A function that slightly mutates a chromosome. Mutation functions will differ based on algorithms but a general mutation function would be to flip a gene bit.

(e) **Mating** - A function that produces a new chromosome based on two input individuals.

(f) **Fitness** - A rating for a solution given by a defined fitness function.

Example 1 - Traveling Salesman

The Traveling Salesman is a classic optimization problem in computer science. Given a list of cities, the goal is to find the shortest path that visits every city before returning to the starting city. The problem is generally modelled using a complete graph, with edge weights equal to the distance between connected cities. Because this is an optimization problems, a genetic algorithm is a good choice to find a solution.

Modeling A Solution

The solution to the salesman problem will be some ordered list of the graph vertices. Let us say we have a set of cities V={1, 2, 3, 4, 5} with a corresponding adjacency matrix:

$$M=\begin{pmatrix} 0 & b & c & d & e \\ b & 0 & f & g & h \\ c & f & 0 & i & j \\ d & g & i & 0 & k \\ e & h & j & k & 0 \end{pmatrix}$$

A potential solution to this problem could be written as: $S_1=\begin{bmatrix}3 & 1 & 4 & 5 & 2 & 3\end{bmatrix}$
We can calculate the fitness score of S_1 using fitness function f and our adjacency matrix.

$$f(S) = M_{S_n 0} + \sum_{n=0}^{n-1} M_{S_n, S_n+1}$$

Example

$$f(S_1) = c + d + k + h + f$$

Genetic Algorithm Steps

1. Initialize an initial population of size m made up of randomly generated paths.
2. Determine the two most 'fit' individuals from the population. The selected will survive into the next generation.
3. Put the selected individuals through the mating function to produce the remainder of the next generation.
4. Repeat steps 2 and 3 until a stop criteria is met. For the purpose of my program, the algorithm will terminate after a given number of generations has been produced.

Mating Function The mating function is essential for producing the next generation for the genetic algorithm. The function accepts two genomes as input and outputs a new genome that shares DNA from the two 'parent' genomes. Here is how the mating function works given: *mate*(A, B) = C

1. Select a random gene subset of A from index i to j
2. Copy this subset of A into C into the same positions as they appear in A.
3. While C still has blank genes, fill them in order from the genes of B, without adding duplicates.
4. Mutate C. Select two random indices and swap their positions.

Mating Example

Let parents A = $\begin{bmatrix}3 & 1 & 4 & 5 & 2 & 3\end{bmatrix}$ and B = $\begin{bmatrix}4 & 1 & 2 & 3 & 5 & 4\end{bmatrix}$

Let $i = 2$ and $j = 4$. So the random subset of A would be $\begin{bmatrix}4 & 5 & 2\end{bmatrix}$. Copy this subset into C.

At this point C = $\begin{bmatrix} c & c & 4 & 5 & 2 & c \end{bmatrix}$. c is meant to denote a null entry.

Fill in the remaining needed genes from B. The first gene taken from B will be 1 because C already has a 4. 1 will also be the last item of C so that the path forms a loop. The second gene taken will be a 3 because C already has a 2.

At this point C = $\begin{bmatrix} 1 & 3 & 4 & 5 & 2 & 1 \end{bmatrix}$

Now mutate C by swapping two random elements. (Not including the first and last elements.

A final C could be $\begin{bmatrix} 1 & 3 & 5 & 4 & 2 & 1 \end{bmatrix}$.

Graphic Example

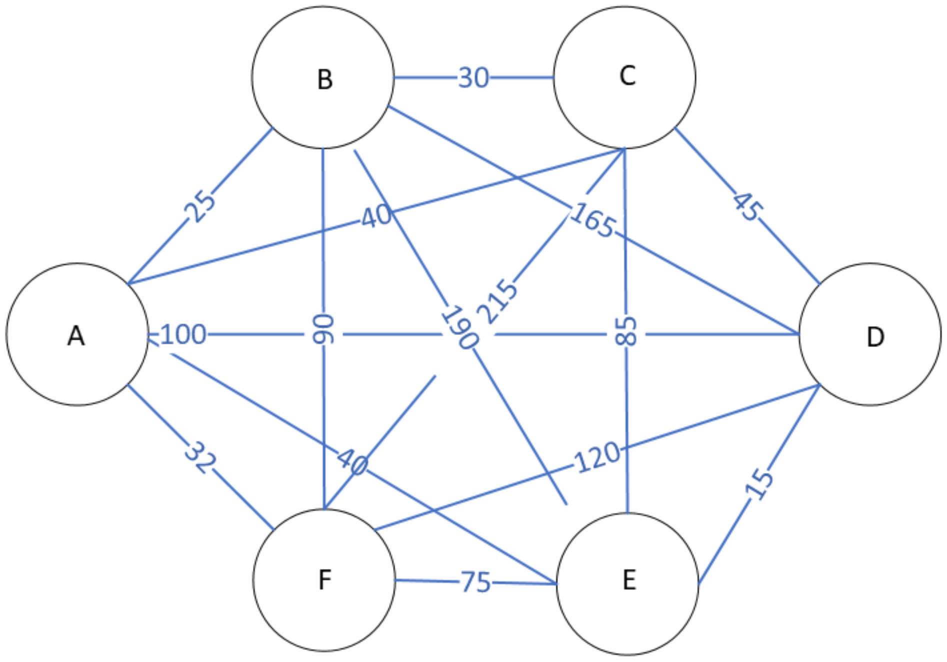


Figure 1: City Graph

Let's use the algorithm to try and find the optimal path through Figure 1. For this simple example, we will set the population size to 4 and let the algorithm run for 4 generations. Figure 2 was produced based on output from my software. The two most fit individuals from each population are filled in blue. The arrows point to the individuals that were born into the next generation.

Rust Implementation

Figure 3 shows how the data structures for the algorithm are represented in the Rust programming language. *Vec⟨T⟩* represents an arbitrary vector of type T. *i64* is a 64 bit integer. *CityMap* represents the adjacency matrix of the cities. *Path* represents a possible path solution. Population is where current generations are stored and where we keep track of how many generations have occurred. Log is used to save generation data to a text file. My full Rust implementation can be found here: GitHub.

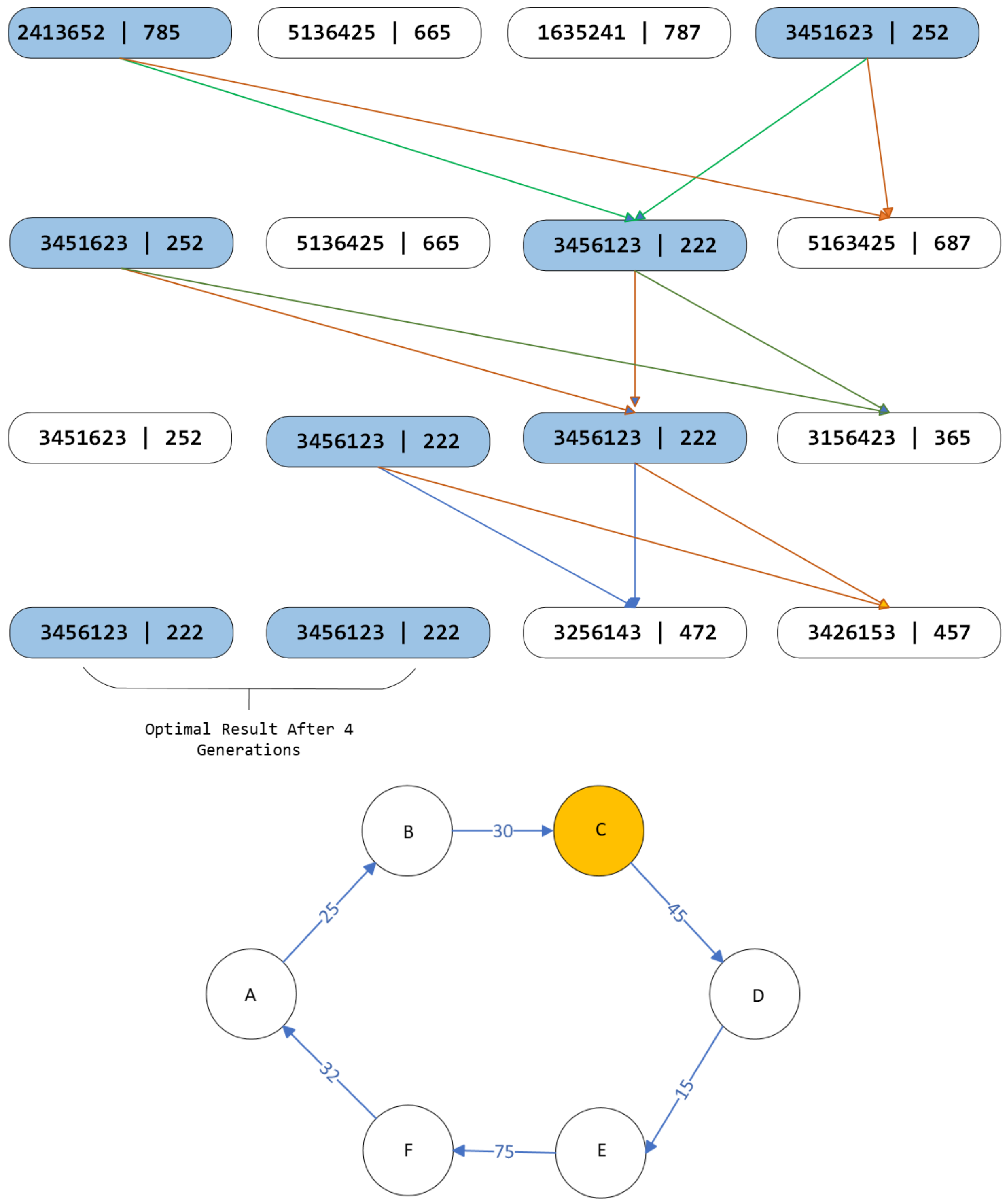
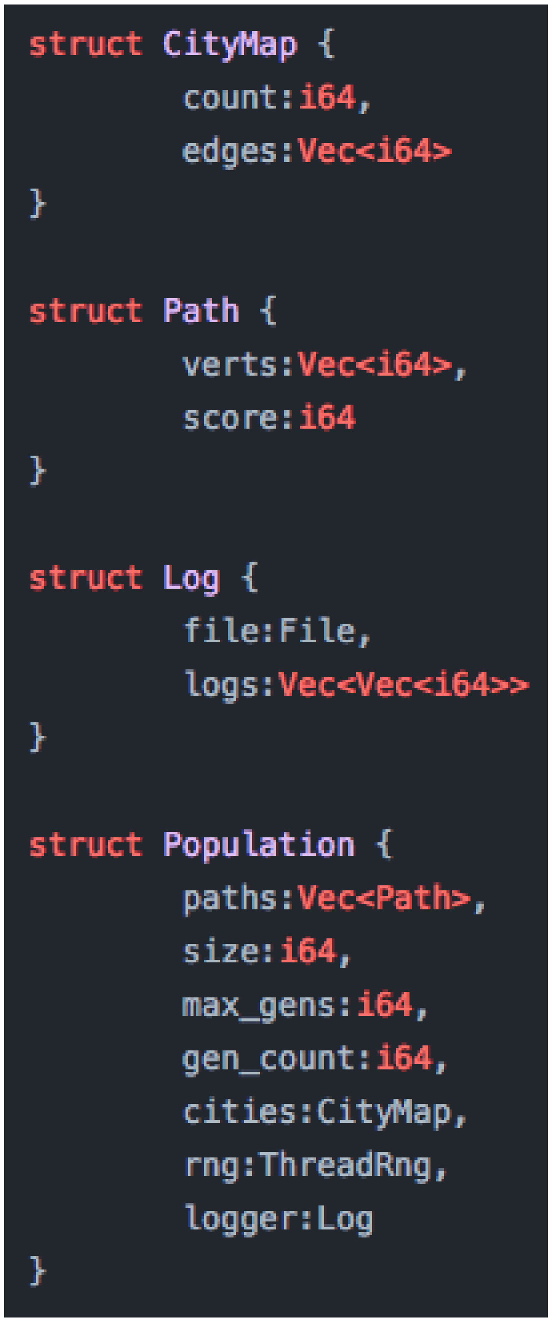


Figure 2: Traveling Salesman Run



```
struct CityMap {
        count:i64,
        edges:Vec<i64>
}

struct Path {
        verts:Vec<i64>,
        score:i64
}

struct Log {
        file:File,
        logs:Vec<Vec<i64>>
}

struct Population {
        paths:Vec<Path>,
        size:i64,
        max_gens:i64,
        gen_count:i64,
        cities:CityMap,
        rng:ThreadRng,
        logger:Log
}
```

Figure 3: TSP Data Structures

Conclusion

As we can see, a decent solution to the 6 city traveling salesman problem was found in just four generations. As the number of cities increases, it becomes more difficult to know when to stop the genetic algorithm. The program may converge to one solution for many generations, but we cannot always prove that there is no better solution. Gene mutation generally helps prevent the program from converging on a solution too quickly, but it is not always fool proof. It may be worthwhile to run the algorithm multiple times with different starting generations to become more confident in a solution.

2. **Other Applications in Machine Learning**

Feature selection can be a difficult part of creating an accurate machine learning model. Genetic algorithms can be used to find optimal sets of feature weights. Furthermore, they could be used to determine an optimal sampling from input data. For example, if you have a large heterogeneous data set with over-sampled and under-sampled groups, a genetic algorithm may be a good choice for determining an optimal subset to be used as training data.

Statistical Analysis and Creativity

Genetic algorithms can be used to find a function to match a data set. This may be for finding a linear fit for a data set or for attempting to solve a differential equation. In general, a genetic algorithm is not the best choice for these types of problems, but they can still produce cool results. The imperfection of a GA can sometimes be desirable for fields like artificial creativity. You could design an algorithm that attempts to find a function based on Claude Debussy's compositions. The first generations might sound incoherent, but the individuals that get closer to a Debussy piece will live on. If you let the program run indefinitely, perhaps it would converge to "Claire de Lune". But if you set the program to terminate earlier, it may have produced a new piano composition that (maybe) sounds good.

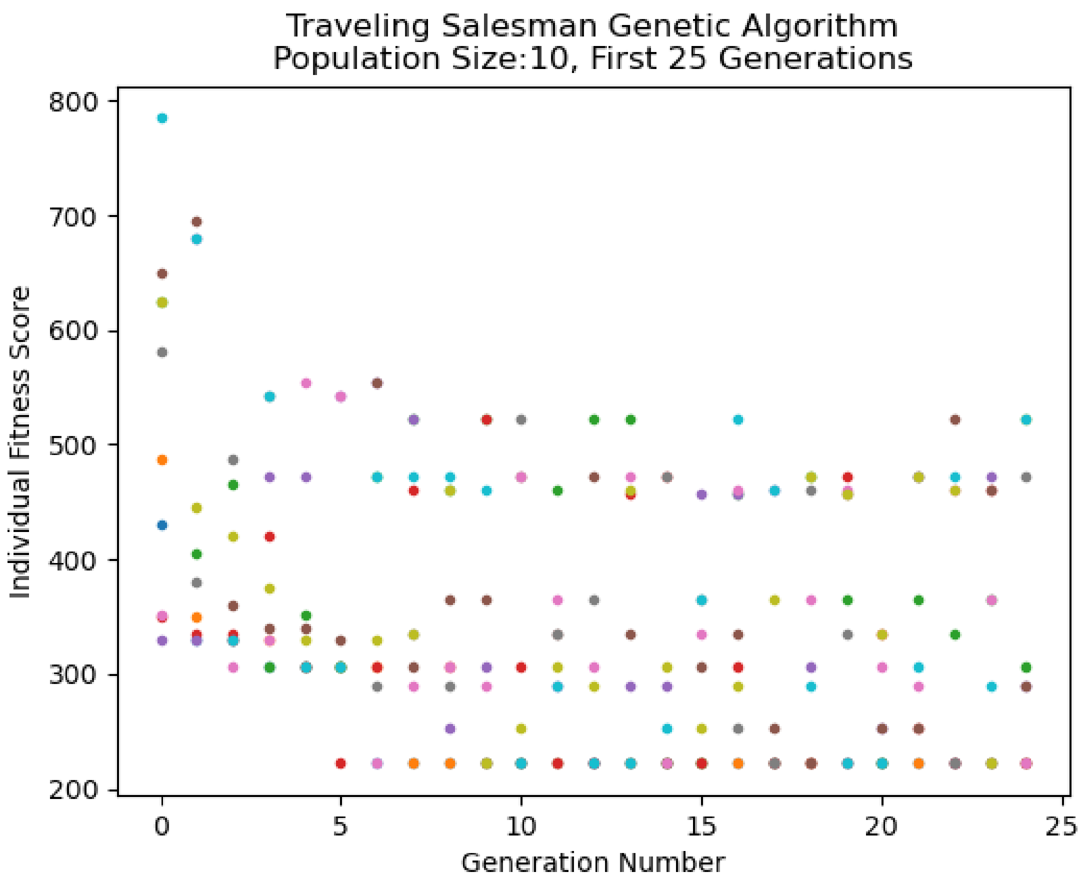


Figure 4: The First 25 Generations of the TSP

Resources

1. https://www.section.io/engineering-education/the-basics-of-genetic-algorithms-in-ml/
2. http://www2.econ.iastate.edu/tesfatsi/holland.GAIntro.htm
3. https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3
4. https://numpy.org/doc/stable/reference/
5. https://www.britannica.com/science/traveling-salesman-problem
6. https://machinelearningmastery.com/why-optimization-is-important-in-machine-learning/