Railroads
Genetic Algorithm

Marija Ćetković

e-mail marijacetkovic03@gmail.com

31 August, 2024

Abstract

This paper serves to demonstrate the application of a genetic algorithm to a railroad network problem, demonstrating its implementation in the sequential, parallelized (multithreaded) and distributed environment.

1. Introduction

The inspiration for Genetic Algorithms, used for various optimization techniques, is taken from the natural principles of evolution and genetics. They mimic the process of natural selection through generations (algorithm iterations) with the goal of finding an optimal solution within an often vast solution space. GA's apply mechanisms to the population of individuals that represent possible solutions to achieve convergence to a near-optimal or optimal solution. The mechanisms used are known as genetic operators: selection (of the parents), crossover (of the parents' genetic material) and mutation (of the offspring gene), which are all analogous to the behaviour observed in nature itself.

Algorithm 1 Genetic Algorithm

1: **Initialize** population P
2: **Evaluate** fitness of individuals in P
3: **repeat**
4: **Select** elite individuals E
5: **Select** parents from P
6: **Apply** GA operators to produce offspring
7: **Evaluate** fitness of offspring
8: **Create** new population P from offspring and E
9: **until** termination condition is met
10: **Return** best solution found

Key terms in understanding GA's architecture are the genes, individuals and the population. To be able to apply the GA one must model the terms in accordance with the problem one is trying to solve. The core aspect of any GA lies in the way the individuals are evaluated. Their fitness score is calculated by a fitness function, which is the heart of the convergence success of the algorithm, alongside parameters like elitism number, mutation and crossover rate and the implementation of the genetic operators.

2. Problem Definition: Railroads

Railroads is a 2D grid-based game where the objective is to create a train network that allows trains to travel from their starting tiles to their destination tiles. The grid consists of various tile types: straight junctions, 2- and 3-turn junctions (with rotations) and crossroads.

Players can configure each tile to optimize the network, aiming to enable all trains to reach their destinations efficiently. The challenge is to achieve this with the fewest tile changes and minimal penalty based on tile types.

The assignment involves implementing an evaluator using a Depth-First Search (DFS) algorithm to score the proposed solutions by determining if each train can reach its destination and applying penalties if not. A genetic algorithm is used to evolve solutions from a random initial state, continuously improving the network to ensure all trains reach their destinations while minimizing the cost.

3. Genetic Algorithm Application

To apply the GA, we must first obey the GA architecture we mentioned. In our case, the gene is represented by the different types of tiles (represented dually with 3x3 binary matrices modelling the exact shape of the tile and an accompanying integer key), the individual is an NxN tile matrix and the population is a list of the individuals. We defined a TileDictionary object to hold the keys

and 3x3 representations of the tiles, which is referenced for the conversion between the integer and the binary matrix representation.

```
public class Population {
    List<Railroad> solutions;
    ...
}
public class Railroad implements
    Serializable, Comparable<Railroad> {
    int[][] world;
    List<int[]> trains;
    ...
}
```

Listing 1: `Population` and `Railroad` classes

```
public class TileDictionary {
    HashMap<Integer, int[][]> tileMap;

    public TileDictionary() {
        tileMap = new HashMap<>();
        ...
        int[][] tile5 = {
            {0, 0, 0},
            {1, 1, 0},
            {0, 1, 0}
        };
        ...
    }

    public int[][] getTile(int tileNumber)
     { ... }
    public int getKey(int[][] tile) { ...
    }
    public int[][] transform(int[][] m) {
    ... }
    public int getPrice(int tileNumber) {
    ... }
}
```

Listing 2: `TileDictionary` class

The mentioned can be visualized with the GUI implemented for this algorithm, which can be set to run together with the algorithm execution in sequential, parallel or distributed mode.

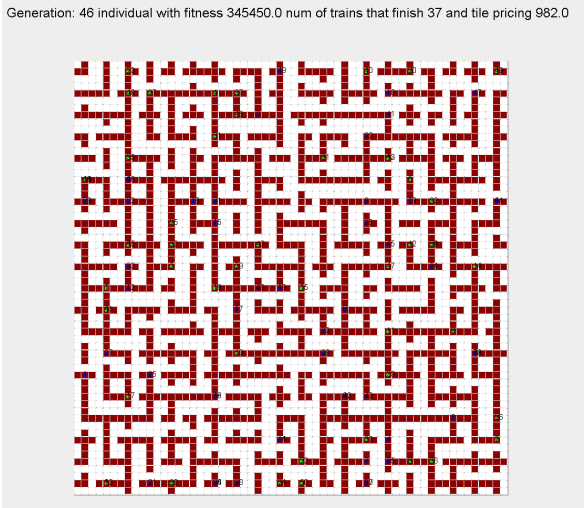

Figure 1: Example Railroad

4. Fitness Evaluation Functions

The fitness of an individual in the genetic algorithm is assessed using two distinct functions: `rateFitness` and `rateFitnessWithPricing`.

The `rateFitness` function evaluates the fitness of a Railroad individual based on the connectivity of train paths within the grid. It computes the total path length of all trains in the transformed world representation. The fitness score is directly proportional to this total path length, reflecting the effectiveness of the train connections. The function is designed to assess how well the trains are connected, without considering the cost of the tiles used. The pseudocode for this function is:

```
function rateFitness() {
    for each train in trains {
        fitness += findPath(world, start,
    end)
    }
    return fitness
}
```

Listing 3: Pseudocode for `rateFitness`

In contrast, the `rateFitnessWithPricing` function provides a more comprehensive evaluation by incorporating both the efficiency of the train paths and the associated tile costs. It calculates the total number of train paths, scales this number, and then subtracts the cost of the tiles used. This results in a fitness score that accounts for both connectivity and cost-efficiency. The pseudocode for this function is:

```
function rateFitnessWithPricing() {
    for each train in trains {
        numTrains += findPath(world, start
    , end)
    }
    tilePricing = getSum()
    scaledTilePricing = tilePricing *
    TILE_PRICING_SF
    numTrainsScaled = numTrains *
    NUM_TRAINS_SF
    fitness = numTrainsScaled -
    scaledTilePricing
    return fitness
}
```

Listing 4: Pseudocode for `rateFitnessWithPricing`

4.1. Future Work

Alternatively, we will aim to explore the implementation of a fitness function that targets the minimal number of tile swaps.

5. Convergence Improvements

5.1. Improving the Initial Population

The performance of the genetic algorithm was observed to be satisfactory for smaller world sizes. However, as the world size increased, the algorithm's performance deteriorated significantly. This decline is attributed to the initial population being too random, which often results in solutions far from any feasible configuration. Specifically, as the network size increases, connectedness becomes more challenging to achieve due to the increased complexity.

To address this issue, we considered the trivial solution: railroad filled with crossroads, which represents a fully connected graph in which every train can reach its destination.

We could can calculate the Hamming distance between randomly generated solutions and this reference solution. We then retain those solutions with minimized Hamming distance, thus ensuring better initial connectedness.

Using Crossroad Tiles Distribution: Another approach, computationally light, that achieves the same idea is assigning *crossroad_distribution* percentage of the railroad area to the tile that guarantees improved connectedness, at random.

To better grasp and visualize the importance of having a near-feasible solution in the initial population for the convergence of the algorithm, we tested a 1000-population of world size 50 on 250 trains. The regular one had a crossroad tile distribution of 0.25, what brought it to be stagnating at 60 trains found for the first 50 generations, without any progress. Spiking the crossroad tile distribution by only 0.02 made a significant difference in the enhanced run of the algorithm.

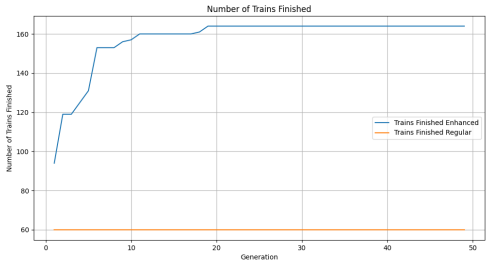

Figure 2: Number of trains

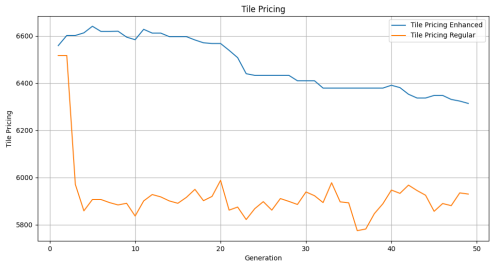

Figure 3: Tile pricing

The regular one still converges in terms of tile pricing, but it takes drastically more time to discover new train roads due to a highly disconnected initial solutions.

5.2. Dynamically Adjusting Mutation Rate

As the algorithm converges, a problem that many optimization algorithms face in the search within the vast solution space is being stuck in a local optimum. The algorithm has exhausted the patterns in individuals and exploited them to the maximum, but at that point of the convergence process it is prevented from ever exploring some different patterns lying in the solution space, that might potentially improve the optimality of the algorithm. Hence, we can keep information about the stagnation of our generational fitness, and as soon as it exceeds a certain value, we can adjust the mutation rate, in hopes of taking care of the stagnated exploration factor of the GA.

6. Genetic Operators Used

While it is valuable to evaluate the efficiency of the algorithm with respect to different genetic operators, several variations were explored during the initial development phase. The selected operators, as described below, were chosen based on their performance and contribution to the genetic algorithm (GA).

6.1. Selection

Three variations of the selection operator were tested:

- **Truncation Selection:** This method involves selecting individuals based on a sorted order of fitness, giving preference to the fittest individuals. Although theoretically sound, this approach did not perform well in practice. It was assumed that truncation selection limited the choice to only the fittest individuals, thereby reducing the diversity of the population and diminishing the exploration factor of the GA.

- **Random Selection:** This approach involves randomly selecting individuals that are within a certain threshold of fitness from the fittest individual. However, this method was ultimately not chosen.
- **Roulette Wheel Selection:** The selected method involves calculating the total fitness of the population and determining the selection probability for each individual. This probability is proportional to a segment on a 'roulette wheel'. A random spin of the wheel then determines which individuals are chosen. This method favors individuals with higher fitness but also allows less fit individuals to be included, striking a balance between exploitation and exploration. The probability of selecting individual i is given by:

$$P_i = \frac{F_i}{\sum_{j=1}^{N} F_j} \tag{1}$$

where F_i is the fitness of individual i and F_j is the fitness of the j-th individual in the population.

6.2. Crossover

The crossover variation employed is a single random vertical point in the matrix representation of the Railroad. Taking the point to be fixed (e.g. middle) did not significantly change the initial efficiency. We thought taking more points would not be beneficial for this gene representation (tile matrix), but this variation was not tested.

6.3. Mutation

The chosen mutation operator is Insertion Mutation. This method involves randomly selecting matrix tiles of the individual Railroad for mutation based on the mutation rate. This approach was selected due to its effectiveness in introducing diversity into the population.

6.4. Elite Solutions

Preserving a percentage of the best-evaluated solutions of a generation without applying mutation or crossover significantly contributed to the convergence of the algorithm. It is also a parameter that can be tweaked (spiked up for really large population sizes), but without it, the best solutions are risked to be changed by the GA operators, potentially harming the found pathways and preventing convergence.

6.5. Future Work

Future steps include investigating other variations of genetic operators, thoroughly testing them, and documenting their effects on algorithm efficiency. This could provide further insights into optimizing the genetic algorithm's performance.

7. Sequential Mode

This implementation of the genetic algorithm operates sequentially to optimize the railroad network problem. Initially, the algorithm sets up the Population class, which manages the collection of potential solutions. The process begins by initializing the population with random solutions (with enhancement for larger worlds with crossroad distribution) and then proceeds through a loop for a predefined number of generations.

In each generation, the algorithm performs the genetic operations. First, it evaluates the fitness of each individual using the fitness function with tile pricing. Following the evaluation, the mutation rate is dynamically adjusted using a strategy to prevent stagnation, where the mutation rate is increased if there is no significant improvement and restored to default when progress is observed.

The elite (variable number of best solutions) is chosen to preserve, and crossover and mutation are applied on the rest of the population. After evaluation, the best individual is updated based on the latest generation's results and recorded for further analysis.

Finally, the program tracks the execution time and outputs the performance metrics once the algorithm has completed its run.

8. Parallel Mode

Unlike the sequential approach, which executes tasks one after another, the parallel mode distributes computational work across multiple threads, thereby speeding up the execution of the algorithm.

The `RParallel` class initializes the parallel execution environment with a specified number of threads and sets up necessary components including the `WorkSplitter` class for dividing tasks and a `ExecutorService` thread pool for managing parallel execution. The core of the parallel approach involves distributing the evaluation and building of new populations across threads.

```
1 public class WorkSplitter {
2     int capacity;
3     int size;
4
5     public int getStart(int rank) {
6         int chunk = calculateChunkSize();
7         return rank * chunk;
```

```
    }

    public int getEnd(int rank) {
        int chunk = calculateChunkSize();
        return Math.min(capacity - 1, (
    rank + 1) * chunk);
    }

    private int calculateChunkSize() {
        return (int) Math.ceil((double)
    capacity / size);
    }
}
```

Listing 5: The `WorkSplitter` Class

In each generation, the algorithm first evaluates individuals in parallel. The `evaluateInParallel` method assigns evaluation tasks to multiple threads using the `PEvaluatorWorker` class, each handling a segment of the population. A shared `CyclicBarrier` is used to synchronize the work between the threads, ensuring that they complete their tasks before proceeding to the next phase.

```
public class PEvaluatorWorker implements
    Runnable {
    Population p;
    int start;
    int end;

    @Override
    public void run() {
        p.performEvaluation(start, end);
        RParallel.awaitBarrier();
    }
}
```

Listing 6: The `PEvaluatorWorker` Class

The population is then updated in parallel through the `buildInParallel` method. This method coordinates the generation of new individuals using multiple threads via the `PBuilderWorker class`. Results from these threads are collected into a concurrent queue to ensure thread safety, and the `collectResults` method aggregates them into the new population.

```
public class PBuilderWorker implements
    Runnable {
    Population p;
    int start;
    int end;
    List<Railroad> workerP;
    ConcurrentLinkedQueue<List<Railroad>>
    results;

    @Override
    public void run() {
        p.buildPopulation(start, end,
    workerP);
        results.add(workerP);
        RParallel.awaitBarrier();
    }
}
```

Listing 7: The `PBuilderWorker` Class

9. Distributed Mode (Message Passing Interface)

In the MPI-based implementation of the genetic algorithm, the computation is distributed across multiple processes, with one process acting as the master and the others as workers.

The master process (rank 0) handles the initialization and coordination. Initially, the master sets up the population and broadcasts it to all worker processes using `MPI.COMM_WORLD.Bcast()`. This ensures that each process starts with the same initial population. Master also chooses the elite for the generation. The work is split among the threads (like in parallel version), which perform evaluation on their chunks of population. The list of Railroad objects (population's solutions) is passed through MPI functions as `MPI.OBJECT` variable. Upon all threads reaching the barrier `MPI.COMM_WORLD.Barrier()`, master collects evaluated solutions from the workers and combines them.

This is achieved using `MPI.COMM_WORLD.Send()` and `MPI.COMM_WORLD.Recv()` for sending and receiving partial solutions.

Upon gathering of the results, master broadcasts the population back to all processes. A similar process is performed for building the population (applying GA operators).

The master process also manages the performance metrics output, tracking execution time and the best individual for the generation.

10. GUI Synchronization

In this implementation, the `BlockingQueue<Railroad>` is used to coordinate between the genetic algorithm and the GUI. The genetic algorithm, running in one thread (main in sequential and parallel, master in distributed), adds the current best solution to the queue with `q.add(bestIndividual)`. The GUI, running in another thread, retrieves this solution using `q.take()`, which blocks until a new solution is available. This setup ensures that the algorithm can continue processing while the GUI updates asynchronously, allowing efficient and synchronized interaction between the two components.

11. Testing

11.1. Performance across implementations

Testing of the algorithm execution times for the three implementation methods was performed on different world sizes, for a fixed number of generations (100), number of trains (100) and population

size (1000). For the larger world sizes (50,60,70) adjustments in the distribution of crossroad tiles was made to help the convergence.

Size	Sequential	Parallel	Distributive
2	3825	5563	4850
5	10941	6739	9742
10	33912	15792	21225
20	97127	39697	88652
30	257789	97097	144205
40	461001	162553	238517
50	524316	184743	297699
60	1002489	341591	488409
70	1849573	786805	—

Table 1: Execution Times for Sequential, Parallel and Distributive Implementation

The table shows execution times for the algorithm measured in milliseconds. The sequential method is the slowest, as expected, with times exceeding 30 minutes for the largest input size. The Distributive method, effective up to size 60 (due to the lack of technical resources used during testing), shows improved performance in comparison to the sequential one, but the parallel implementation is conveniently the highest performing one.

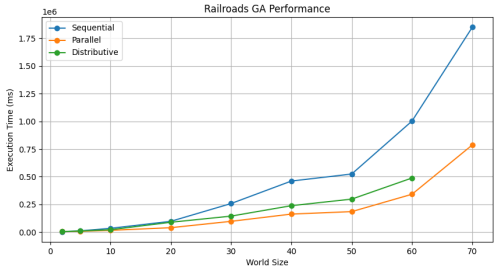

Figure 4: Testing results

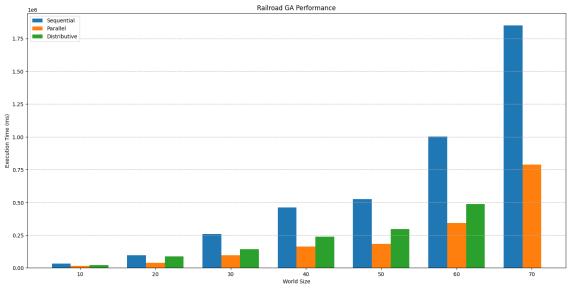

Figure 5: Testing results as Bar Chart

11.2. Varying train/fixed world size

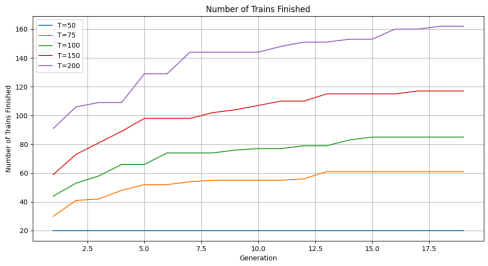

Figure 6: Varying train number on fixed world size

We fixed world size on $N = 30$ and tested with $T = 50, 75, 100, 150, 200$ number of trains. With more trains in a fixed world size, the available tiles are increasingly utilized, leading to more frequent path overlaps and a smaller search space. This makes it easier for the GA to find paths, as there are fewer configurations to explore. As a result we can see that the GA converges better for increased number of trains because viable paths are more readily discovered in the less random environment. Smaller number of trains allow for a much greater search space, where a more connected initial railroad solutions are needed to avoid stagnation.

11.3. Technical Remark

The program is executed through command-line arguments, where the user must specify the mode of execution (1 for sequential and 2 for parallel), the world size, the number of trains, and whether to display the algorithm simulation (GUI). The standard usage for the serial and parallel versions is as follows:

> java Main <mode><worldSize ><numTrains><displayGui>

For the distributed version, the program is executed using the MPJ Express library. It is essential that MPJ Express is installed on your system. To run the distributed mode set the following as the VM options in your IDE:

> -jar "MPJ_PATH/lib/starter.jar" RDistributed -np <numThreads >

And program arguments to:

> <worldSize><numTrains><displayGui>

12. Problems faced

12.1. Initial Convergence Issues

The primary challenge, as previously discussed in Section 5.1, was the randomness of the initial pop-

ulation. Through a process of trial and error and careful monitoring of the algorithm's behavior, this obstacle was identified and subsequently addressed with the implementation of the crossroad distribution technique.

12.2. Fitness Function with Tile Pricing

Adjusting scale factors for the number of trains that finish their journey relative to the scale factor for the total tile price posed a challenge as well. The performance of the fitness function is noticeable, but further research could potentially improve it.

12.3. Adapting to different implementations

The parallel implementation of the genetic algorithm required synchronization mechanisms to ensure consistency across multiple threads, but it was not hard to split the problem. Implementing the distributed version of the algorithm proved more complex. The primary issues involved managing the message transmission of lists of Railroad objects and regulating packet sizes using MPJ library functions, with lack of debugging options.

12.4. Testing Genetic Operators

Evaluating various configurations of genetic operators, such as selection methods, proved to be time-consuming and challenging. Comparing the convergence performance of different operator variations required extensive testing and analysis.

References

[1] Genetic Algorithm https://en.wikipedia.org/wiki/Genetic_algorithm

[2] Selection operator https://en.wikipedia.org/wiki/Selection_(genetic_algorithm)

[3] MPJ Express http://mpjexpress.org/

The source code for the genetic algorithm implementation presented in this paper can be accessed at the following GitHub repository: https://github.com/marijacetkovic/railroads.