Railroads: An Evolutionary Approach to Problem Solving

Viktor Taseski
University of Primorska FAMNIT
Koper, Slovenia
vtaseski24@gmail.com
89231157@student.upr.si

1 INTRODUCTION

The problem of constructing optimal train paths is complex due to the constraints on tile placement and path connectivity. This project employs a genetic algorithm to iteratively improve solutions. Different execution modes (sequential, parallel, distributed) are available for comparison in computation speed.

2 PROBLEM DEFINITION

Given a grid-based world with trains assigned start and end locations, the objective is to create a valid track system while minimizing the map cost. Each tile type has a specific cost, and solutions are evaluated based on fitness, which considers connectivity, efficiency, and penalties. Depth-First-Search (DFS) algorithm should be used to find the path from the start to the end location. The solution fitness is determined by how many trains have reached the end location and the total map cost.

3 METHODOLOGY

I employ a genetic algorithm to evolve optimal railway network configurations. The fitness function evaluates the connectivity of train paths and penalizes solutions that fail to complete routes, ensuring robust performance. My implementation leverages both multi-threaded execution and distributed computing to harness multiple CPU cores and MPI communication for accelerated evaluations.

3.1 Genetic Algorithm

The algorithm consists of:

- Initialization: Generating random grids. `initPopulation()`
- Selection: Favoring high-fitness solutions. `selectParent()`
- Crossover: Combining solutions to generate new ones.
- Mutation: Introducing small changes to maintain diversity.
- Evaluation: Evaluating fitness of the new generation.

3.2 Parallelization and Distribution

Parallel computation is used to evaluate multiple solutions simultaneously. An executor service with a fixed thread pool of size equal to the population size is used to achieve parallelization, therefore each grid is evaluated at the same time. Distributed processing with MPI extends this further by utilizing multiple machines. The population is divided into partitions across all machines equally. `Scatter()` method by MPI is used. Then each machine evaluates its partition using the parallel implementation. After all machines have completed evaluation, the partitions are gathered (`using gather() method`) by the root machine and the next generation is generated.

4 IMPLEMENTATION

The Railroads project is implemented in Java and utilizes a hybrid approach combining genetic algorithms with both multi-threaded and distributed computing to optimize railway map layouts. The key components of the implementation are as follows:

- **Genetic Algorithm**
  - **Population Initialization:** The algorithm begins by generating an initial population of candidate map layouts. This population includes the original layout and a set of randomized variants created using helper functions. Each candidate is encapsulated in a `MapSolution` object.
  - **Fitness Evaluation:** A custom fitness function assesses each candidate based on criteria such as path existence and cost. Both sequential and parallel evaluation strategies are implemented to ensure accurate and efficient fitness computation.
  - **Selection, Crossover, and Mutation:** Parents are selected using a weighted random (ranking-based) method. A diagonal crossover operator is employed to exchange sub-regions between two parent maps, and mutation is performed by randomly altering a tile's type (while preserving essential elements such as trains and stations). These operators ensure diversity and guide the search towards higher-quality solutions.
- **Parallel Computation:**
  - **Thread-safe Evaluation:** To speed up the fitness evaluation phase, the project utilizes Java's `ExecutorService` to execute fitness computations concurrently across available CPU cores. This parallelism is integrated seamlessly within the genetic algorithm loop.
- **Distributed Processing via MPI:**
  - **Population Distribution:** The project employs MPJ Express (a Java MPI implementation) to distribute the population among multiple processing nodes. MPI's `Scatter` operation partitions the population so that each node evaluates a subset of candidates.
  - **Synchronization and Communication:** After local evaluations, the nodes use `Gather` to send their results back to the root process. Synchronization is maintained using MPI barriers and simple signal messages, ensuring that all nodes progress through the evolutionary iterations in unison.
- **Visualization and Logging:** Real-time console outputs and a custom logging utility provide insight into the evolution process. These tools display intermediate population states and performance metrics, aiding in debugging and performance tuning.

- **Serialization and Data Integrity:** All candidate solutions (instances of `MapSolution`) implement the `Serializable` interface to support MPI communication. Special care is taken to serialize complex objects (e.g., 2D arrays representing map layouts) correctly, ensuring that all necessary data is preserved across distributed nodes.

In summary, the Railroads project leverages a robust genetic algorithm framework enhanced by multi-threaded and distributed computing techniques. This design not only accelerates the computation through parallelism but also scales effectively across multiple nodes, thereby providing a powerful tool for optimizing railway map configurations.

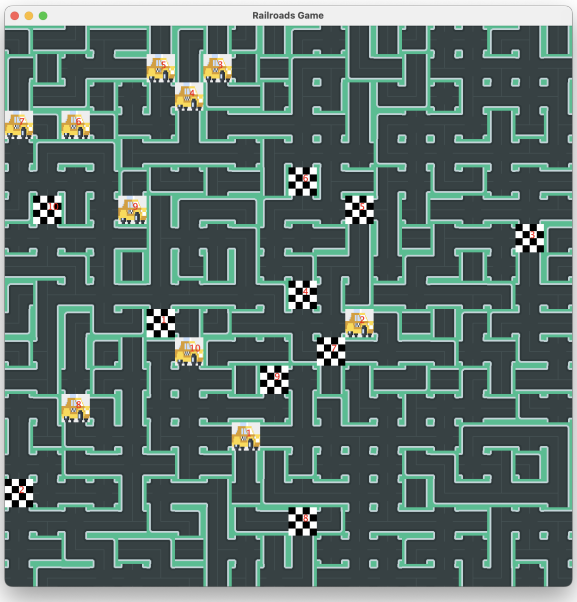
5 USER GUIDE

5.1 Installation

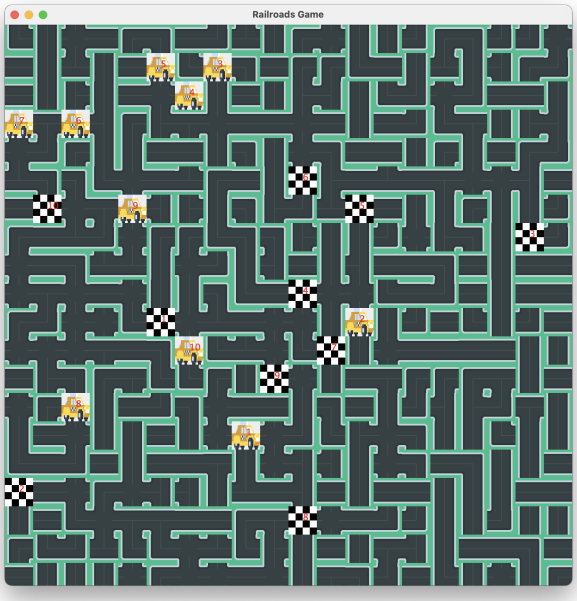
(1) Install Java and MPI.
(2) Clone the repository:
`git clone https://github.com/viktortaseski/Railroads`
(3) Compile the project: `javac Main.java`
(4) Open Project Structure > Libraries and add MPJ Express
(5) Open Run Configurations > DistributedMain and add VM Options
(6) Inside VM enter the args:
`-jar /Users/your/path/mpj-v044/lib/starter.jar -np 5`
(7) Inside Environment Variables enter:
`MPJ_HOME=/Users/your/path/mpj-v0_44`

5.2 Running the Program

- Sequential mode:
(1) Run Main.java
(2) Open terminal
(3) Enter mode = 1
(4) Enter the size of the grid
(5) Enter the number of trains
(6) Enter the number of iterations
(7) Click on the window and press 's'
- Parallel mode:
(1) Run Main.java
(2) Open terminal
(3) Enter mode = 2
(4) Enter the size of the grid
(5) Enter the number of trains
(6) Enter the number of iterations
(7) Click on the window and press 's'
- Distributed mode:
(1) Make sure steps (4) to (7) are completed from 5.1
(2) Game settings have to be changed statically
(3) Default size = 20, numberOfTrains = 5, iterations = 1000.
(4) Run DistributedMain.java
- Testing mode:
(1) Run Main.java
(2) Open terminal
(3) Enter mode = 3
(4) Wait for all tests to finish
(5) Open `results.txt`



(a) Initial random grid



(b) Optimized solution

Figure 1: Example of the genetic algorithm improving a railway network.

- Customizing map:
(1) Do steps 1 to 6 from sequential or parallel
(2) Click on the window
(3) Enter position for X then press Enter

(4) Enter position for Y then press Enter
(5) Enter [0,1,2,3] for Rotation
(6) Enter [0,1,2,3] for tile type
(7) You can repeat to modify other tiles.
(8) Rotation: index*90 = rotation
(9) Tile type: (0,straight), (1,turn), (2,threeway), (3,cross)

6 RESULTS AND ANALYSIS

Performance benchmarks indicate that parallel and distributed modes significantly reduce execution time.

Before talking about the performance, I want to mention that increasing the number of iterations significantly improves the solution, but increases the computation time as expected. Therefore, better solutions require more time. After inspection we can say that on average the parallel implementation is about 4-5 times faster than sequential implementation. In theory distributed implementation should yield the fastest computation, however since all machines are running on one machine we get slower computation in comparison to parallel. Comparing sequential with distributed, we can say that computing smaller problems (small maps with few trains) the sequential is faster. That is a consequence of MPI establishing a connection and running all nodes. As we scale the problem size we begin to notice improvement in distributed implementation. We can examine for size = 40 and trains = 100, the distributed implementation has 50% faster computation, in comparison to sequential implementation. Given the same problem, but with iterations = 1000 we can say the following: Sequential is the slowest with 503 seconds, then we have distributed with 167 seconds and parallel with 117 seconds with the fastest computation time of all implementations.

Table 1: Performance comparison tests of different implementations

Test Type	Size	Trains	Time (ms)	Map Cost	Iterations
Sequential Test	2	1	29	200	100
Sequential Test	5	2	39	203	100
Sequential Test	10	5	153	232	100
Sequential Test	20	20	2841	589	100
Sequential Test	30	50	14156	1298	100
Sequential Test	40	100	51988	2297	100
Parallel Test	2	1	15	200	100
Parallel Test	5	2	16	203	100
Parallel Test	10	5	46	232	100
Parallel Test	20	20	617	589	100
Parallel Test	30	50	2941	1298	100
Parallel Test	40	100	10631	2297	100
Distributed Test	30	50	12823	1289	100
Distributed Test	40	100	24555	1289	100
Sequential Test	2	1	18	200	1000
Sequential Test	5	2	69	202	1000
Sequential Test	10	5	835	219	1000
Sequential Test	20	20	22581	349	1000
Sequential Test	30	50	113232	585	1000
Sequential Test	40	100	503211	1217	1000
Parallel Test	2	1	45	200	1000
Parallel Test	5	2	67	202	1000
Parallel Test	10	5	331	219	1000
Parallel Test	20	20	4967	349	1000
Parallel Test	30	50	23954	585	1000
Parallel Test	40	100	117484	1217	1000
Distributed Test	30	50	62511	585	1000
Distributed Test	40	100	167659	1217	1000

7 CONCLUSION

The project successfully demonstrates the effectiveness of genetic algorithms for optimizing railway networks. As we scale in problem size, it is crucial to optimize the computation speed. In this project most of the computation is done to find a path for each train which is part of the evaluation, hence that is why we parallelize the evaluation. With evaluation phase optimized using multi-threading increase in computation speed is achieved. Further distributing the population evaluation to multiple machines achieves even better efficiency. Small optimizations can contribute to extremely faster computations. Therefore eliminating unnecessary evaluations, introducing concurrency as well as dividing the work among multiple machines can significantly increase computational speed without increasing computational power.