Robo-maze Blast Report

Dogà Sara
Widmann Lukas
Askar Sami

Abstract

Bomberman + Evolutionary algorithmens + Tournament Arc goes brr

1 Introduction

1.1 The game

Robo Maze Blast, created in 2008 by Kai Ritterbusch and Christian Lins, is a clone of the Bomberman game. Also known as Dynablaster, it is a strategy maze-based video game franchise originally developed by Hudson Soft in 1985.

The general goal of Bomberman is to complete the levels by strategically placing bombs in order to kill enemies and destroy blocks. Some blocks in the path can be destroyed by placing bombs near it, and as the bombs detonate, they will create a burst of vertical and horizontal lines of flames. Except for indestructible blocks, contact with the blast will destroy anything on the screen.

1.2 Our Goal

The aim of our project is to explore the efficiency of different Genetic Algorithms to develop 3 agents with strategic competence in the Robo Maze Blast scope, and to compare them by making the agents fight against each other and observe which agent outlives the others more frequently.

2 Background

2.1 Genetic Algorithms

Genetic Algorithms (GA) are optimization algorithms inspired by the process of natural selection and biological evolution. They are widely used to solve complex optimization and search problems in various domains. One of those domains is the gaming side as in this paper. Due to constrained optimization (e.g., state/action of the game), Genetic Algorithms are a perfect choice for this task [**?**]: *"Genetic Algorithms (GAs) were selected for their ability to handle complex combinatorial optimization problems [...] and to encode domain-specific constraints"* (Section 1).

The core steps of a typical genetic algorithm can be described as follows:

- **Population Base**: Initialize a population from valid chromosomes, i.e. a set of strings that encodes any possible solution. Usually, the initial population is chosen randomly.
- **Evaluation**: Each population solution is evaluated on the basis of a predetermined fitness function.
- **Selection**: Reproductive opportunities are allocated to the chromosomes that represent a better solution to the target problem, and such solutions are selected to form a 'mating pool' for the next generation.
- **Crossover and Mutation**: The selected individuals are then combined to produce offspring by exchanging genetic material. Sometimes small changes can happen in the genetic material, such as bit flips. All of this ensures good exploration of the solution space and diversity.

These steps are repeated for a number of times until an ending criterion is reached.

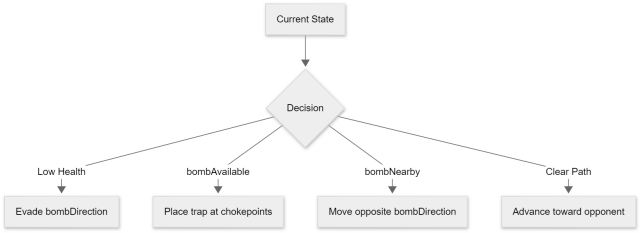

Figure 1: A diagram on the possible actions of a player

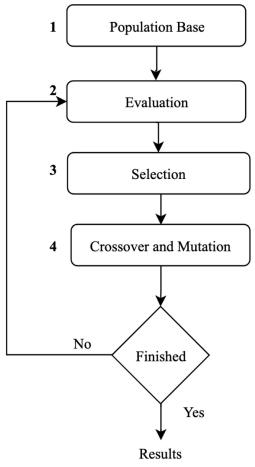

Figure 2: A diagram on the steps of a genetic algorithm

2.2 Robo Maze Blast's Default AI Agent

The game has its own AI agents that will play against the player in the absence of in-real-life adversaries. They share a common

Robo-Maze Blast Project, Hamburg, Germany
2025. ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM

behavior and reasoning that can be visualized with the finite-state machine, as shown in Figure 3.

3 Fine-tuning agent behaviors with jenetics

3.1 Differential Evolution

3.2 Agent Behavior

3.3 The Reward Metrics

The fitness function determines which agent behaviors are rewarded, and which are penalized.

Table 1: Agent Reward and Penalty Values

Action	Points
Movement	+1 (per step)
Place Bomb	+75 (per bomb)
Blow Wall	+150 (per wall)
Kills	+750 (per player)
Death	-500
Suicide	-500
Win without kills	+200
Win with kills	+1000

4 Supervised Learning with Jenetics

4.1 Introduction

My objective was to create an AI player using a genetic algorithm (GA) trained on human gameplay data, with an exploration of skill transferability across gaming domains. This approach was motivated by competitive gaming frameworks where player strength is quantified through Elo rating systems [Elo78]. As an experienced fighting game player, my current Tekken 8 Elo rating stands at **1940** (profile: https://wank.wavu.wiki/player/3nyHJQr8Gq6Q, accessed August 8, 2025). While acknowledging that direct skill transfer between a 3D fighting game (Tekken) and a 2D grid-based strategy game (Robo Maze Blast) may be limited, this project tests the hypothesis that:

Hypothesis: Strategic decision-making patterns in high-Elo players exhibit domain-agnostic qualities, such that:

$$\text{Elosource} \geq 1900 \xrightarrow{\text{transfer}} \text{AItarget} > \text{AI}_{\text{baseline}} + \Delta$$

where Δ represents measurable skill advantage in Robo Maze Blast.

4.2 Jenetics

Jenetics is an open-source Java library that provides a genetic Algorithm (GA) framework for solving optimization problems. It abstracts biological evolution principles, such as selection, crossover, and mutation, into re ble software components, enabling users to evolve solutions without implementing a GA from scratch.

4.3 Jenetics Framework

We selected it for supervised learning because:

- **Prior experience**: We'd successfully used it in previous labs
- **Java compatibility**: Integrated smoothly with our game codebase
- **Rapid adjustments**: Changing parameters takes minutes instead of hours
- **Easy versioning**: Git checkpoints for different configurations

Key Features We Utilized:

- **Evolutionary engine**: Automatically handles generations and survival mechanics
- **Domain flexibility**: Worked directly with our game strategy optimization
- **Pre-built operators**:
  - Tournament selection (picks winners from random groups)
  - Gaussian mutation (makes small, smart adjustments)
  - Single-point crossover (combines parent solutions)

Agile Implementation: The library let us quickly test configurations by reducing parameters:

- Population size: 100 → 500
- Mutation rate: 0.1 → 0.05
- Training generations: 50 → 200

This enables an iterative improvement cycles:

0. Gather human data
1. Train initial AI
2. Identify weaknesses
3. Adjust parameters
4. Retrain (under 30 minutes)

Reference: Jenetics User Manual [Wil24].

4.4 Implementing Gameplay Recording: Code Changes for Data Collection

To train the AI using Jenetics, we first needed good quality gameplay data. The simplest way to get this data was to record what a human player does during a game - both their actions and the game situation at that moment. This required changes to the Player class, which led to creating the new `RecordablePlayer` class. The main changes are shown in Algorithm 1 and include:

- **Recording trigger**: Making the game log actions when players move or place bombs
- **State capture**: Saving player position and bomb status in simple numbers
- **Data export**: Saving all records to a CSV file for AI training

Example from actual gameplay: Imagine this situation during a game:

- Player is at position (7, 12) in a 20x20 grid
- Player can place a bomb (has bombs available)
- There's a bomb nearby above the player

The `RecordablePlayer` would save this information as:
`0.35, 0.60, 1.0, 1.0, -0.4, 5`

Value	Meaning
0.35	Player is 35% across the level (X position)
0.60	Player is 60% down the level (Y position)
1.0	Bomb available
1.0	Bomb nearby
-0.4	Bomb is above player
5	Player placed a bomb

Algorithm 1: RecordablePlayer Modifications

Class RecordablePlayer **extends** Player
New Data: recordings = []
isRecording = false // Recording on/off switch
Key Changes:
move(dx, dy): super.move(dx, dy)
if *isRecording* **then**
LogMovement(dx, dy) // Records moves
placeBomb(): super.placeBomb()
if *isRecording* **then**
LogAction(BOMB) // Records bombs
New Methods:
LogMovement(dx, dy): Convert dx/dy to direction
LogAction(direction)
LogAction(action): state ← [normX, normY, bombAvail, bombNear]
recordings ← recordings + (state, action)
Save(filename): Write all recordings to file

This simple number format allows the AI to learn from many such records. In the future I would collect a sample trainings data set to train

4.5 Data Augmentation for Generating a Training Dataset

To overcome the limitations of this small sample size, a common challenge in data-intensive tasks like training an AI, we would apply data augmentation techniques. Specifically, we would modify this training data by mirroring gameplay across both the X and Y axes. This simple yet effective method quadruples our dataset, generating new, valid scenarios that maintain the original gameplay logic. As shown in the example below, a leftward movement on the original grid (e.g., player at 0.35) becomes a rightward movement in the mirrored version (0.65), and a downward movement becomes an upward movement (0.40). This process not only increases the quantity of our data but also enhances its diversity, exposing the AI to a broader range of game states and actions.

The use of geometric transformations like mirroring is a well-established method for creating new training samples from a limited dataset [PW17].

This technique is especially useful in supervised learning contexts where the underlying data structure, such as a coordinate system, can be logically transformed without altering the class labels or core relationships within the data.

Looking ahead, we would also explore the possibility of adding additional labels to our data to provide more context. This could include a numerical rating, similar to the Elo system, to grade the quality of a specific action or the player's performance during a session. This would provide the AI with richer context, such as classifying a move as "defensive" or "aggressive," further refining the learning process. This would involve a server-side database to record match outcomes and calculate a lasting Elo rating for different AI player implementations, allowing for a more nuanced and competitive evaluation of fitness over generations.

Algorithm 2: Proposed Jenetics Workflow

Engine Configuration:
engine ← Engine.builder(fitnessFunction, chromosome)

```
1  .populationSize(500)                    // Balanced diversity/computation
2  .optimize(Optimize.MAXIMUM)    // Maximize fitness score
3  .offspringSelector(new TournamentSelector<>(5))
4  .survivorsSelector(new EliteSelector<>(0.2))
5  .alterers(
6   new Mutator<>(0.12),    // Domain-adjusted mutation
7   new SinglePointCrossover<>(0.85)
8  )
9  .build ()
```

Evolution Execution:
result ← engine.stream()

```
10 .limit(by(SteadyFitness(15, 50)))    // Stop when plateau detected
11 .peek(stat → logGeneration(stat)) // Progress tracking
12 .collect (EvolutionResult.toBestGenotype())
```

4.6 Hypothetical Implementation with Jenetics

While not implemented due to time constraints, this section outlines the planned pseudocode structure for the Jenetics-based evolutionary approach. The design reflects best practices in GA configuration with domain-specific adjustments for game AI development.

4.6.1 Design Rationale. The parameter selections balance exploration/exploitation tradeoffs against computational constraints:

(1) **Population Size (500)**: Hypothesized to maintain solution diversity while remaining computationally feasible (estimated 35min/generation). Larger than default to counteract premature convergence in complex game state spaces.
(2) **Mutation Rate (0.12)**: Intentionally higher than Jenetics' default (0.01) to prevent behavioral local optima. This would encourage exploration of unconventional strategies (e.g., calculated risk-taking) that might outperform conservative play.
(3) **Steady State Termination**: Planned to use fitness plateau detection (15 generations without 0.5% improvement) rather than fixed generations. This adaptive approach prevents unnecessary computation after convergence.
(4) **Tournament Size (5)**: Maintained at Jenetics' recommendation. Altering this would require simultaneous adjustment

of selection pressure metrics - changing multiple interdependent parameters was avoided per [**?**].

(5) **Elite Retention (20%)**: Conservative preservation of top performers ensures successful tactics propagate while allowing population diversity. This follows standard practice in steady-state evolution [**?**].

4.7 Outlook

This project encountered several limitations that merit consideration in future work. The primary constraint was the delayed commencement of the implementation phase, attributable to initial uncertainties regarding project specifications and concurrent academic commitments, including a separate laboratory project and examination during the final month. Initiating development 2-3 months earlier upon clarification of assignment parameters would have facilitated more comprehensive implementation.

A critical oversight involved underestimating the integration complexity between the developed AI agent and other project components. Consequently, the planned comparative evaluation against other group members' agents could not be executed.

Despite these constraints, the project yielded valuable insights regarding data augmentation methodologies, Elo system applications in artificial intelligence, and practical experience with the Jenetics framework. These learnings establish a foundation for future work, which should prioritize agent implementation completion and systematic tournament-based evaluation.

5 Tree-Based Genetic Programming

5.1 Introduction

The evolutionary algorithm chosen for the second implementation is tree-based genetic programming (GP). This approach offers multiple benefits: First, it is highly flexible. The tree structure allows a wide range of programs to be represented by the algorithm. Furthermore, GP can "discover novel and unexpected solutions that may not be apparent to human programmers," as the algorithm is not biased toward favoring certain approaches over others, as long as they are within the rules set by the programmer or hardware. Additionally, GP can easily adapt to changing requirements and inputs, making it highly adaptable. This approach was chosen due to its high flexibility. While the initial training phase against the already implemented AI players might be more challenging due to the complexity of the setup compared to other algorithms like Differential Evolution or regular Genetic Algorithms, the benefits of flexibility and adaptability to new circumstances are promising in an environment where multiple evolutionary algorithms compete against each other.

5.2 Design Decisions

The implementation of the algorithm is encapsulated in the GeneticPlayer class. This class does not inherit from the AIPlayer class, which already contains significant functionality. Inheritance was avoided because the setup of genetic programming differs significantly from "regular" genetic algorithms. GP uses a tree structure and consists of multiple components (functions, terminals, and constants). Terminals represent the leaf nodes of the tree, whereas functions branch out, increasing the complexity of the tree.The GeneticPlayer class contains an abstract static class called Operation, which serves as the base for all terminals and functions. The respective terminals and functions inherit from Operation. The terminals include basic movement operators such as MoveUp, MoveDown, MoveLeft, MoveRight, and Stay, which represent movements in the game along the x- and y-axes. Additional terminals include IsTargetZone, CheckForBomb, PlaceBomb, CheckForExtra, GetDistanceToEnemy, and three constant values. These terminals encapsulate functionality already implemented for the AIPlayer, allowing the GP to iterate over it. The constant values are arbitrary defaults that can be adjusted based on the algorithm's performance or goals.The algorithm also relies on two operations: GreaterThan and IfElse. GreaterThan has an arity of 2, meaning it compares two values, returning 1 if the first value is greater than the second and 0 otherwise. IfElse has an arity of 3. It checks whether the first value is greater than 0. If true, it forwards the value t[1]; otherwise, it forwards t[2].The core of the GP is the initializeGP method. In this method, functions and terminals are initialized, and the maximum tree depth is set. During the training phase, a depth of 5 was used, but this can be adjusted based on performance. The engine combines the algorithm's constraints with the fitness function. Parameters such as population size, mutation rate, and crossover rate can be manually adjusted depending on the algorithm's performance. The result performs the actual training by streaming the data and applying the constraint of the number of generations. Finally, the best strategy is returned.The fitness function is where the actual training occurs. Each time the fitness function is called, a new game and playground are initialized. A new GeneticPlayer and three new AIPlayer instances are created. Notably, during the training phase, a specific method, addAIForFitness, was implemented in the Game class. This method does not start the threads typically used during a game simulation. The operations and terminals are linked to the newly created GeneticPlayer.

```
for (int i = 0; i < 1000; i++) {
          if (!game.isRunning() || goodboy.isDead()) {
           System.err.println("Fitness: Stopped at iteration " + i +
                steps = i;
                break;
            }
            for (Player player : game.getPlayers()) {
                if (player instanceof AIPlayer aip) {
             System.err.println("Fitness: AIPlayer " + aip.getNickn
                    aip.tick();
                } else {
             GeneticPlayer gp = (GeneticPlayer) player;
                   double result = gp.program.eval();
             System.err.println("Fitness: GeneticPlayer tick, resu
             if (result == 1.0 || result == 2.0 || result == 3.0 || re
                        successfulMoves++;
                    }
                }
            }
            game.tick();
        }
```

In the loop, the player performs a maximum of 1,000 iterations before being forced to terminate. The loop is interrupted if the

player dies or the game is no longer running, simulating a real environment where the GP interacts with AIPlayer instances. In each iteration, both the AIPlayer and GeneticPlayer call their tick-methods, selecting a new action and synchronizing their current state. After all players have synchronized, the game synchronizes with game.tick. During game.tick, selected bombs detonate, extras collected by players in the previous period modify their bomb range or capacity, and players may be killed by exploding bombs and removed from the game. Successful moves by the GP are recorded during the loop, as they impact the overall fitness.The fitness function consists of multiple components:

- survivalTime
- extrasCollected
- deathPenalty
- moveReward

Since the algorithm aims to minimize the fitness function, beneficial actions for the GP receive a negative multiplier, while harmful actions receive a positive multiplier. Notably, the fitness function currently ignores the number of opponents the GP kills, meaning aggressive behavior does not affect fitness. This design choice promotes defensive behavior, prioritizing survival over aggression. The goal is for the player to survive as long as possible, leading to longer games compared to aggressive strategies where the player either dies or kills quickly. This defensive approach, inspired by strategic games like chess where top players prioritize avoiding mistakes over attacking, is reinforced by a high deathPenalty and a strong emphasis on survivalTime as a negative metric.The GeneticPlayer class also includes a tick method, called by the GeneticPlayerThread, which is started in a real game environment to visually observe the player's behavior.

5.3 Issues

The algorithm in its current form does not perform as intended. Several issues arose during implementation, some of which remain unresolved:

(1) Not inheriting from the AIPlayer class. The primary issue, likely causing subsequent problems, was the decision not to inherit from the AIPlayerclass. Although this was a deliberate choice initially, it likely caused more harm than good. Much of the functionality was copied with slight adjustments, adding redundant code without significant value. The GeneticPlayer inherits from the Player class, sharing many variables with AIPlayer. This required moving some code, such as the die and isDead methods, from AIPlayer to Player to make it accessible to the GP.
(2) Adjusting classes that were already functional. Other classes, such as the Game class, were modified to include the tick method, which synchronizes player actions within a time frame. However, these adjustments introduced additional issues, complicating debugging.cDuring the project, console output showed that the player performed actions like moving up and down, which sometimes succeeded and sometimes failed depending on obstacles like walls. However, the player never placed bombs. Debug print statements revealed that the condition player.bombs.size() < player.bombCount was never true, indicating that bombs always contained a bomb before the output statement was triggered. This led to replacing many System.out.println() statements with System.err.println(), suspecting that threading issues were affecting the print statements (as threads were initially started for all AIPlayer instances). This hypothesis was confirmed, revealing another issue: the players and bombs were not synchronized.
(3) Synchronization of player behavior. While AIPlayer instances were bound to a tick every 300 ms, bombs exploded every 4,000 ms. The GP, however, was not constrained by these timings, operating only limited by CPU capacity. This caused the GP to perform significantly more iterations before a bomb exploded, leading to poor assumptions about its behavior after placing a bomb, often resulting in the player committing suicide. Ultimately, AIPlayer threads were disabled to remove artificial constraints designed for a realistic gaming experience, allowing the simulation to run without being bound by these limitations. The above issues have left the algorithm only partially functional. Synchronization of triggered behaviors (bomb explosions, AIPlayer actions, and GeneticPlayer actions) caused significant problems. Many issues could have likely been avoided by inheriting most behavior from AIPlayer. The synchronization problem could have been mitigated by aligning the GP's tick time with that of the AIPlayer. However, this would have slowed the algorithm or reduced its quality, as the complexity and population size would need to be significantly reduced to maintain performance.

References

[Elo78] Arpad E. Elo. *The Rating of Chessplayers, Past and Present.* Arco Publishing, 1978. Original Elo system formulation.
[PW17] Luis Perez and Jianyi Wang. The effectiveness of data augmentation in image classification. *arXiv preprint arXiv:1712.04621*, 2017.
[Wil24] Franz Wilhelmstötter. *Jenetics User Manual (Version 8.2.0)*, 2024. Accessed: 07.08.2025.