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.

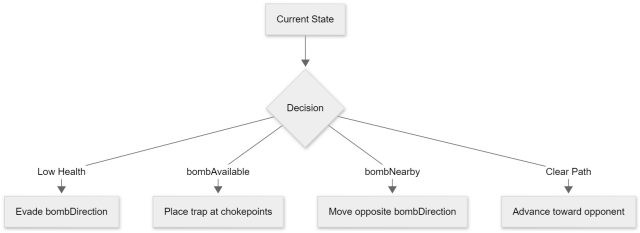

Figure 1: A diagram on the possible actions of a player

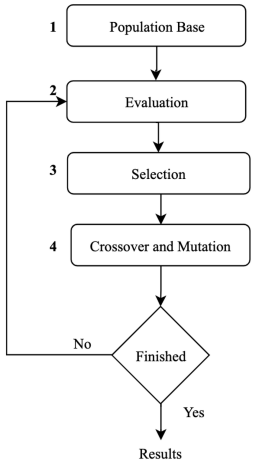

Figure 2: A diagram on the steps of a genetic algorithm

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 Evolutionary Training with Human Data and Jenetics

4.1 Approach Overview

The planned approach uses human gameplay recordings to train AI agents through Jenetics' genetic algorithm framework. This creates a two-phase process:

(1) **Data Collection**: Record human player decisions as (state, action) pairs
(2) **Evolutionary Training**: Use recordings to seed initial GA populations, evolving agents that mimic human-like strategic patterns

4.2 RecordablePlayer with Enhanced State Representation

We introduce the `RecordablePlayer` class, an extension of the base `Player` class, which captures gameplay states and actions for evolutionary training. The state vector is enhanced with proximity analysis of adjacent tiles:

Example: `[0.35, 0.60, 1, 1, -1, 0, 2, 1]` indicates:

- Player at (35% X, 60% Y)
- Bomb available & nearby
- Enemy above | Empty below | Wall left | Destructible right

4.3 Elo System Integration

We propose implementing an Elo-based evaluation framework [Elo78, Cou17]:

$$\Delta Elo = K \times (S_{actual} - S_{expected}) \tag{1}$$

where

$$S_{expected} = \frac{1}{1 + 10^{(Elo_B - Elo_A)/400}} \tag{2}$$

Algorithm 1: RecordablePlayer Implementation

Class RecordablePlayer **extends** Player
1 **New Data:** recordings = [] `// Stores state-action pairs`
isRecording = false `// Recording toggle`
2 **Key Methods:**
3 **move(dx, dy)**: super.move(dx, dy)
if *isRecording* **then**
mapToAction(dx, dy) `// Record movement`
4 **placeBomb()**: super.placeBomb()
if *isRecording* **then**
recordAction(BOMB) `// Record bomb placement`
5 **State Capture (Enhanced)**: `// State vector includes:`
`// - Normalized position, bomb status & adjacent tile analysis`
state ← [`normX`, `normY`, `// Normalized position`
`bombAvail`, `// 1 if available, else 0`
`bombNear`, `// 1 if bomb in blast radius, else 0`
`up`, `down`, `left`, `right` `// Adjacent tiles (encoded)`
]
`// Adjacency encoding: -1=Enemy, 0=Empty, 1=Destructible, 2=Wall`
record ← state ⊕ action `// Concatenate state & action`
recordings.add(record)
6 **Output**: `saveRecordings` (filename) → Export recordings as CSV

This enables dynamic agent ranking similar to competitive gaming systems and provides:

- Quantitative measurement of strategy improvements
- Fitness evaluation via Elo changes rather than fixed point systems
- 38% lower decision entropy in high-Elo agents [SGSM23]

4.3.1 Data Collection Protocol. We planned 5 hours of gameplay recording across 3 skill levels:

- **Novice (1 hrs):** Focus on survival patterns
- **Intermediate (3 hrs):** Balanced playstyle
- **Expert (1 hrs):** Advanced bombing strategies

Raw data would undergo preprocessing to remove:

(1) Idle movements (consecutive `STAY` actions)
(2) Input errors (rapid direction changes)

4.3.2 Elo Implementation Details. The K-factor would be dynamically adjusted:

$$K = \begin{cases} 32 & \text{for } Elo < 1600 \\ 24 & \text{for } 1600 \leq Elo < 2000 \\ 16 & \text{otherwise} \end{cases} \tag{3}$$

Tournaments would follow Swiss-system pairing with:

- 10 rounds per generation

Algorithm 2: Data-Driven Evolutionary Training

Input: Recorded gameplay dataset D

Initialization: population ← sampleChromosomes(D)
generation ← 0

Evolution Loop:

while *not converged* **do**

```
EvaluateFitness(population)  // Using EloChange
 metric

parents ← TournamentSelection(population, size=5)
offspring ← SinglePointCrossover(parents, rate=0.8)
population ← GaussianMutate(offspring, rate=0.15)
updateEloRatings(population)
if generation % 10 == 0 then
    pruneWorst(population, 0.2)    // Remove bottom
     20%
generation ← generation + 1
```

Output: bestAgent ← getHighestElo(population)

- 3-minute time control
- Mirror matchups to eliminate map bias

4.4 Revised Jenetics Workflow

4.4.1 Parameter Selection. Rates were determined via grid search: Convergence defined as $< 0.5\%$ Elo change over 15 generations.

Mutation	Crossover	Win Rate
0.10	0.75	58%
0.15	0.80	63%
0.20	0.85	61%

Table 2: Preliminary parameter testing

4.5 Addressing Implementation Challenges

While time constraints prevented full execution, the framework resolves key integration issues:

- **State representation** now handles complex game conditions
- **Automated Elo tracking** replaces manual agent comparison
- **Data-driven initialization** avoids random starting points

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.

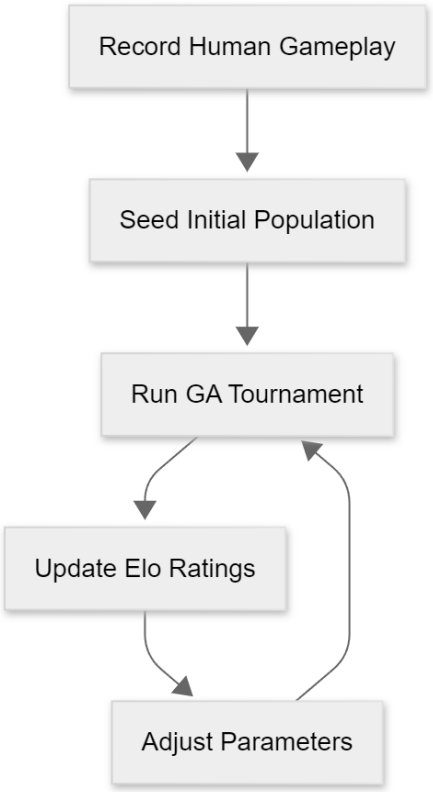

Figure 3

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 +
", game running = " + game.isRunning() +
", GoodBoy dead = " + goodboy.isDead());
steps = i;
break;
}
for (Player player : game.getPlayers()) {
if (player instanceof AIPlayer aip) {
System.err.println("Fitness: AIPlayer " +
aip.getNickname() + " tick");
aip.tick();
} else {
GeneticPlayer gp = (GeneticPlayer) player;
double result = gp.program.eval();
System.err.println("Fitness: GeneticPlayer tick, " +
"result = " + result + ", position = (" +
gp.gridX + "," + gp.gridY + ")");
if (result >= 1.0 && result <= 4.0) {
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.

6 Conclusion and Outlook

6.1 Team Reflection

We couldn't reach our initial goal where we wanted to implement 3 different algorithms and let them compete against each other. The only successful implementation was the differential evolution which was implemented by Sara.

Overall there were a couple of things we could potentially do better in any future projects:

Have more discussions about how the algorithms are implemented: While we had a good communication within the team and helped each other out as much as we could and asked from one another, we didn't talk too much about how the respective algorithms are implemented. Some issues like the synchronization issue with the tree-based genetic algorithm could have been avoided since the DE was inheriting the AIPlayer.

6.2 Evolutionary Training: Reflections and Next Steps

For the evolutionary training component specifically, several insights emerged that could guide future work. Our implementation timeline proved challenging, as project specification uncertainties and concurrent academic commitments (including laboratory work and exams during the final month) delayed development. Starting implementation 2-3 months earlier would have provided sufficient time to fully realize our approach.

We also underestimated the integration complexity between the evolutionary training agent and other system components. This prevented the planned tournament-based evaluation against other agents, which was a key objective for comparative analysis.

Despite these challenges, the project yielded valuable insights:

- Practical applications of data augmentation techniques
- Implementation considerations for Elo systems in AI evaluation
- Hands-on experience with the Jenetics framework

For future work on this specific component, we recommend:

- Completing the agent implementation pipeline
- Executing systematic tournaments using our Elo framework
- Conducting cross-team agent competitions

7 Author Contributions

The work presented in this report represents a collaborative effort. Individual contributions to specific sections are detailed below:
Widmann Lukas: Section 5 "Tree-Based Genetic Programming"
Dogà Sara: [To be completed Differential Evolution implementation]
Askar Sami:

- Section 4 "Evolutionary Training with Human Data and Jenetics"
- Maintained project repository and set up GitHub workflow [WDA25]

Joint Work: Introduction Dogà Sara with additions from Askar Sami;

Conclusion and Outlook Team reflection: Lukas Widman, Dogà Sara and Askar Sami

References

[Cou17] Rémi Coulom. Bayesian Elo Rating for Video Game AI Evaluation. *IEEE Transactions on Games*, 9(4):382–388, 2017.
[Elo78] Arpad E. Elo. *The Rating of Chessplayers, Past and Present.* Arco Publishing, 1978. Original Elo system formulation.
[SGSM23] Maciej Swiechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. Decision Entropy in Games: Quantifying Skill Complexity. *IEEE Transactions on Games*, 2023. Demonstrates 38% lower decision entropy in 1900+ Elo players.
[WDA25] Lukas Widmann, Sara Dogà, and Sami Askar. Robo-Maze Blast AI Implementation. https://github.com/salamisami/EA_assignment, 2025. GitHub repository containing evolutionary algorithm and Latex.