Genetic Algorithms for Controller Generation of Understandable Bomberman Controllers

TIM WIJMA, University of Twente, The Netherlands

This thesis explores the use of genetic algorithms (GAs) to automatically generate understandable agent controllers for the game Bomberman using guarded command programs (GCPs). This approach evolves agents represented by a fixed-size set of human-readable, condition-action rules within the Pommerman environment. Experiments analyzing various mutation and crossover strategies confirmed that the GCP structure is effective at producing interpretable agent logic. The results demonstrate that the GA successfully evolved agents capable of basic survival tactics, primarily by learning to avoid self-destructive behaviors. However, the key finding of this research is that agent performance was fundamentally constrained by the fitness function, which failed to adequately reward complex strategic and tactical maneuvers. This work concludes that while GAs combined with GCPs provide a viable framework for generating understandable AI, achieving high-level strategic competence is critically dependent on the design of a more sophisticated fitness function that can guide evolution beyond simple behaviors.

Additional Key Words and Phrases: Genetic Algorithms, Guarded Command Programs, Bomberman, DEAP, Pommerman

1 INTRODUCTION

Bomberman is a game in which players navigate a grid-based world and strategically place bombs to eliminate opponents while avoiding explosions. Developing intelligent agents for dynamic environments such as Bomberman requires algorithms that balance offensive strategies, such as placing bombs and collecting power-ups, with defensive tactics, like avoiding bombs and efficiently navigating the environment. Agents are generally made with hand-crafted rule-based systems, finite state machines (FSM), and more recently, complex machine learning models. Although hand-crafted rule-based systems and FSMs are easy to interpret, they lack scalability for complex and dynamic environments. On the other hand, machine learning models scale well but behave as "black-boxes". Their decision-making is hard to understand, and debugging undesired behavior can be complicated [3, 8]. Genetic algorithms, combined with guarded commands, offer a middle ground by enabling automatic rule generation while keeping the resulting behavior easy to understand.

Guarded command programs are a set of simple condition-action rules which define the agents' behavior. Each rule consists of a guard, a boolean expression consisting of a number of logical conditions, and a command to execute if the conditions are true. In the context of Bomberman, a simple example of this could be `enemyInRange` $\wedge$ `hasBomb` $\rightarrow$ `dropBomb`. As long as the number of conditions is limited, these rules are inherently easy to understand, making them well suited for designing interpretable agent behavior in complex dynamic environments like Bomberman.

This thesis investigates how genetic algorithms can be used to automatically generate guarded command programs that control agents in the Bomberman environment. The main objective is to create agent behavior that is both effective in the game and interpretable. To guide this, the following research question is posed:

How can genetic algorithms be used to automatically generate understandable guarded command programs for agents in a Bomberman environment?

This question is broken down into the following sub-research questions:

- **SRQ1**: How do different mutation and crossover strategies affect the evolution of rule complexity and game-playing effectiveness in guarded command Bomberman controllers?
- **SRQ2**: What gameplay metrics (e.g., survival time, opponents eliminated) should be used to evaluate the performance of a controller in the Bomberman environment?
- **SRQ3**: What structure and constraints on guarded command programs balance interpretability and gameplay performance?

The complete source code for the genetic algorithm implementation, including the Bomberman environment, agent controllers, and data analysis and visualization scripts, is publicly available on Github. [1]

2 RELATED WORK

2.1 Genetic Algorithms

Genetic Algorithms (GA) are optimization algorithms inspired by the process of natural selection and biological evolution. They are widely used for solving complex optimization and search problems across various domains.

The core components and steps involved in a typical genetic algorithm can be conceptualized as an iterative loop, as illustrated in Figure 1, and include:

- **Population Initialization:** A starting population of candidate solutions is randomly generated. Each individual represents a potential solution to the problem, encoded in a specific genetic representation.
- **Fitness Evaluation:** Each individual in the population is evaluated based on a predefined fitness function. This function quantifies the quality or performance of a given solution, guiding the evolutionary process towards better solutions.
- **Selection:** Individuals are selected from the current population to form a 'mating pool' for the next generation. Selection methods (e.g., roulette wheel selection, tournament selection, rank selection) are designed to give preference to individuals with higher fitness, simulating 'survival of the fittest'.

TScIT 43, July 4, 2025, Enschede, The Netherlands

© 2025 University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

[1] https://github.com/TimWijma/genetic-bomberman

- **Crossover (Recombination):** Selected individuals (parents) are combined to produce new offspring. Crossover operators (e.g., single-point, two-point, uniform) exchange genetic material between parents, allowing for the exploration of new areas in the search space by recombining beneficial traits.
- **Mutation:** Small, random changes are introduced into the offspring's genetic material. Mutation operators (e.g., bit-flip, inversion) maintain genetic diversity within the population, helping the algorithm escape local optima and explore entirely new solutions.
- **Elitism (Optional):** To ensure that the best-performing individuals found so far are not lost due to the stochastic nature of crossover and mutation, a small number of the fittest individuals from the current generation may be directly copied into the next generation.

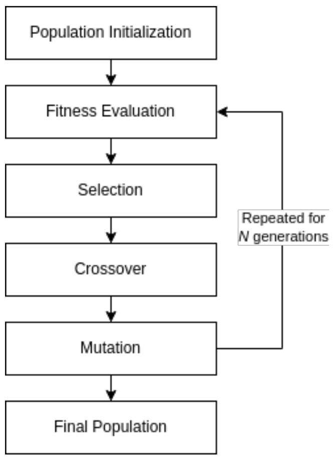

Fig. 1. General Flow of a Genetic Algorithm

These steps (evaluation, selection, crossover, mutation, and optionally elitism) are repeated for a set number of generations or until a termination condition is met, at which point the best solution(s) from the final population can be extracted.

One application of this is *EvolvingBehavior*, a tool developed by Partlan et al. [4] that uses a genetic algorithm to evolve behavior trees for non-player characters in a game environment. The developer initializes a simple decision tree and sets the evolution parameters, and the genetic algorithm then evolves the behavior into a complex decision tree. These evolved decision trees were tested in a 3D survival game, and the evolved trees were found to perform similarly to manually crafted trees made by researchers with experience in game AI. While the evolved decision trees were not perfectly efficient, they all used a manageable number of nodes, making their behavior easy to understand. This demonstrates the potential of genetic algorithms to evolve agents with complex but understandable behavior.

One of the strengths of GAs is their flexibility in encoding decision logic, allowing them to optimize not just for performance, but also for properties like simplicity and interpretability. Wang et al. [7] demonstrate this with a system where multiple agents worked together to create simple, human-readable rules from data. This system was tested on the Iris dataset, where the algorithm resulted in simple rules such as "If x_4 is small, then class 1;" (where x_4 is a feature in the dataset).

This simple, human-readable format of the rules generated by the genetic algorithm in Wang et al.'s work is very similar to the structure of guarded command programs. Both consist of a condition and a corresponding action or classification. This suggests that genetic algorithms could be well-suited for evolving guarded command programs to control agent behavior.

2.2 Guarded Command Programs

As mentioned above, guarded command programs are programs which are simply a list of rules. Each rule consists of a **guard** and a **command**. A guard is a boolean expression, which contains a number of conditions in combination with logical operators (such as AND, OR). In the context of Bomberman, this would result in rules such as `enemyInRange` $\wedge$ `hasBomb`, or `bombUp` $\wedge$ (`leftEmpty` $\vee$ `rightEmpty`). A command is an action that the agent should take if the guard is true. Examples of these actions in Bomberman would be `dropBomb`, or `moveLeft`. At every step during the game, the agent evaluates the state of the game and decides which guards are satisfied. In case there are multiple satisfied guards, the agent can choose any rule that is applicable. This clear, rule-based structure makes the agent's decision-making process easier for humans to understand compared to more complex AI models.

A related approach by Kim and Kim (2013) uses genetic algorithms to optimize rule-based systems in the game *Geometry Friends*, a simple 2D platformer with 2 agents. [2]. Their system relied on manually defined rules, such as logic for stopping movement, with the genetic algorithm only tuning numeric parameters within these rules. Their system had better performance than a purely man-made rule-based approach, but did not evolve the structure or logic of the rules themselves. In contrast, this thesis explores evolving both the logical conditions (guards) and actions (commands), which enables the generation of new rules and more adaptive agent behavior.

2.3 Bomberman

Bomberman is a maze-based action game where players navigate an arena, strategically placing bombs to destroy blocks and defeat enemies. The core objective involves using timed explosions to clear a path, uncover power-ups, and eliminate all opponents before the time runs out.

Previous work on agents for Bomberman environments explored a variety of rule-based and planning approaches. A popular tool for running these agents is Pommerman, an implementation of the Bomberman game which allows the user to run tournaments of AI agents against each other to compare their performance [5, 6]. Zhou et al. [9] implemented two agents in the Pommerman environment: one based on a Finite State Machine (FSM) and another using Monte Carlo Tree Search (MCTS).

The FSM agent was built with a set of hand-crafted states and transitions, which allowed it to make quick and understandable decisions. It performed significantly better than Pommerman's provided *SimpleAgent*, and always met the 100ms time limit per step.

The MCTS agent used a more complicated approach by simulating future states to plan ahead. Although this allows for better reasoning, it was often too slow to be effective with the time constraints, making it less practical than simpler approaches.

These two examples show that there is a trade-off to be made between performance and interpretability. Planning methods like MCTS can theoretically perform really well, but they struggle with a strict time limit. Although conceptually simple, planning methods can quickly become difficult to understand as the depth of their lookahead increases. Hand-crafted rule-based approaches like FSMs offer fast execution and are simple to understand, but their maintenance becomes tiresome in more dynamic environments.

This performance-interpretability trade-off in game agents motivates the use of genetic algorithms to automatically generate understandable guarded command programs. This approach aims for effective gameplay performance with more transparent decision-making than planning, and easier maintenance than a manual rule-based approach.

3 METHODOLOGY

This research explores the implementation of genetic algorithms to automatically create intelligent but interpretable agents for the Bomberman game. The approach models agents as a set of guarded command programs, which consist of a number of condition-action rules designed for human readability. The following sections detail the specific design of the genetic algorithm, individual representation, and the game environment.

3.1 Genetic Algorithm Design

To enable agents to incrementally improve their performance over generations, a genetic algorithm was implemented. This section details the specific implementation of each step in the genetic algorithm as detailed in section 2.1.

The genetic algorithm is implemented using **DEAP (Distributed Evolutionary Algorithms in Python)** [1], a Python library that provides helper functions for building and running genetic algorithms.

3.1.1 Individual representation. Each agent (or individual) is represented as a fixed-size, ordered list of ten guarded command programs. These programs, or rules, form the basis of an agent's behavior in the Pommerman environment, with each rule being an instance of the `Rule` class, containing the following components:

- **Conditions**: Each rule contains one to three conditions from the `ConditionType` enumeration, which are boolean checks within the game (e.g., `CAN_MOVE_LEFT`, `HAS_BOMB`).
- **Operators**: When a rule contains more than one condition, these conditions are chained together using `AND` and `OR` from the `OperatorType` enumeration.
- **Actions**: Each rule specifies a single action from the `ActionType` enumeration (e.g., `MOVE_UP`, `PLACE_BOMB`), which will be executed if the rule's conditions are satisfied.

The order of these rules is critical to the agents' performance, as they are evaluated from top to bottom, executing the first satisfied rule. Therefore, the evolutionary process must not only discover effective rules but also optimize their ordering, as rules at the top of the list are most frequently executed.

3.1.2 Population Initialization. The initial population for the genetic algorithm is created by generating a set of agents, each initialized with a list of ten rules. For each rule, the number of conditions, N, is randomly chosen from 1, 2, or 3, with weights of 1, 3, and 1 respectively. These weights prioritize conditions with 2 conditions, as these conditions were found to be expressive enough for intelligent behavior, while remaining interpretable. Next, $N - 1$ operators (either `AND` or `OR`) are randomly selected to chain these conditions together. Finally, an action is randomly chosen from the `ActionType` enumeration. This random initialization of conditions, operators, and actions ensures that the initial population is diverse and unbiased to certain conditions or actions.

These rules are initialized with prioritization for two conditions. Three conditions were found to be too complicated and executed too infrequently to be beneficial to an agent's performance, especially when initialized with random conditions and operators. In contrast, rules with only a single condition do not provide enough information and guards for a rule to be useful. These rules would get executed too frequently and prevented better rules from being executed.

To give agents a starting point and prevent them from dying immediately, a number of predetermined 'seed' rules are added during initialization. These rules consist mainly of bomb evasion behavior (e.g., "If `IS_BOMB_ON_PLAYER` and `CAN_MOVE_UP` then `MOVE_UP`", or "If `IS_BOMB_DOWN` and `CAN_MOVE_LEFT` then `MOVE_LEFT`"). For each agent, between one and six of these rules are randomly selected and inserted into their initial ruleset. These rules are fully subject to mutation during the evolutionary process, allowing the algorithm to refine or adapt them to find more optimized strategies.

3.1.3 Fitness Evaluation. The fitness function is one of the most crucial components of the genetic algorithm, as it directly determines which agent behaviors are rewarded. The agents aim to maximize and exploit this function, thereby shaping their behavior.

In *Bomberman*, the agent should learn to strategically place bombs, trapping and killing its opponents. To stimulate this behavior, agents receive points for actions such as placing bombs, breaking wood, and eliminating opponents. To prevent undesired behavior such as passiveness and self-kills, these behaviors are heavily penalized. Table 1 details the specific metrics and their associated rewards and penalties used in the fitness calculation.

Per generation, agents are randomly selected to participate in five rounds of games. Each round consists of 10 episodes, with four agents starting in each corner of the game environment. The same four agents play together throughout all episodes within a single game round, ensuring consistent comparison among them. During these episodes, agents internally track various metrics, such as number of bombs placed, kills, and unique tiles visited. At the end of each episode, agents receive points according to Table 1. These accumulated points are then averaged by the number of episodes played, which determines the final fitness of an agent.

Although the metrics outlined in Table 1 appear straightforward, designing an effective fitness function proved to be one of the most challenging aspects of this research. For instance, early versions of

Table 1. Reward and Penalty Values for Fitness Metrics

Metric	Reward
Steps	0.1 (per game step)
Unique tiles visited	15 (per tile)
Bombs placed	75 (per bomb)
Wood destroyed	150 (per wood)
Kills	750 (per kill)
Winning with kills	1000
Winning without kills	200
Dying	-500
Self-kill	-1000
Alive at end but not winning	-150
Less than 10 tiles visited	-50 (per tile away from 10)
Less than 4 bombs placed	-100 (per bomb away from 4)
Tiles visited more than 3 times	-5 (per tile per count above 3)

this fitness function heavily rewarded bomb placement but lightly penalized self-kills. This resulted in agents evolving to place bombs haphazardly, earning them *some* points, but almost always ended in self-kills. The logical solution for this would have been to apply a harsher penalty for self-kills; however, this led agents to learn not to place *any* bombs, as it yielded a higher score than risking self-kills.

3.1.4 Selection. For the creation of the next generation's offspring, parents were selected using DEAP's `selTournament` method. This method operates by creating tournaments of seven randomly chosen individuals; the individual with the highest fitness from each tournament is then selected as a parent. This process is repeated until a sufficient number of parents are chosen to generate the new population. Elitism, as described in Section 2.1, is implemented for the top 5% of individuals to prevent the loss of optimal solutions across generations.

3.1.5 Crossover. After parents for the new generation have been selected, their genetic material is combined to create offspring. The selected parent population is iterated over in pairs. Each pair has a crossover rate, defined by `CROSSOVER_RATE` (0.75), meaning there is a 75% probability that they will undergo recombination. When a pair is selected for crossover, the *two-point crossover* method is applied: two indices are randomly chosen between 1 and the maximum number of rules an agent can have. The rules between these two indices are swapped between the two agents, resulting in two new rulesets (offspring).

3.1.6 Mutation. To prevent offspring behavior from becoming stagnant and to introduce new genetic material, mutation is applied to the population. Each individual in the offspring population has a `MUTATION_RATE` (0.4) chance to undergo a mutation operation.

For each rule within an individual, there is a `MUTATION_RATE_RULE` (0.15) probability that it will be altered. First, a rule has a `MUTATION_RATE_REPLACE` (0.075) probability of being replaced in its entirety by a newly generated random rule to introduce entirely new strategic behaviors. If a rule is not replaced, its components can undergo various mutations:

- **Number of Conditions**: If a rule has fewer than three conditions, there is a `MUTATION_RATE_ADD_COND` (0.05) chance of adding a randomly selected condition and a corresponding random logical operator (`AND` or `OR`) to it. Alternatively, if a rule has more than one condition, there is a `MUTATION_RATE_REMOVE_COND` (0.05) chance that one of them will be removed.
- **Condition Replacement**: For each condition within a rule, there is a `MUTATION_RATE_REPLACE_COND` (0.3) chance of being replaced by a different random condition.
- **Operator Replacement**: For each operator in a rule, there is a `MUTATION_RATE_REPLACE_OPERATOR` (0.15) chance of it being swapped to a different operator (e.g., `AND` becomes `OR`).
- **Action Replacement**: The action associated with a rule has a `MUTATION_RATE_REPLACE_ACTION` (0.15) chance of being replaced by a different random action.

After the rule mutations, the entire list of rules within an individual has a `MUTATION_RATE_SHUFFLE` (0.10) chance of having a random subset of its rules reordered. This operation is particularly important given that the order of rules is critical to an agent's behavior, since they are evaluated from top to bottom. Therefore, rule shuffling helps an agent explore the optimal ordering of their rules.

3.2 Environment

For a genetic agent to effectively execute its rules, it requires real-time access to the game state and the ability to quickly evaluate this information to determine its next action. This section details how the Pommerman environment works and how these genetic agents observe and evaluate it.

3.2.1 Pommerman. The Pommerman environment provides a game class that initializes the game for the agents to play with a number of customizable parameters. For all the games that were played in this study, the following parameters were used:

- **Board size**: The default board size of 11x11 was used, small enough for agents to find each other, but not too big that the games would take too long.
- **Game type**: `PommeFFACompetitionFast-v0`, a fast-paced free-for-all game.
- **Powerups**: Powerups have been disabled for the games, as they complicated the behavior too much and made an agents' performance too noisy.
- **Max steps**: To minimize training time, the number of game steps has been limited to 600.

At each game step, the environment provides the current game state to each agent in the form of a dictionary containing components such as the board layout, bomb positions, and agent positions.

3.2.2 Decision making. To determine the next action an agent should take for the current game state, the agent's list of 10 rules is evaluated sequentially from top to bottom. For each rule, the agent checks if its conditions are met. Each condition corresponds to a helper function that evaluates against the current game state (e.g., the condition `CAN_MOVE_UP` checks if the tile above is safe). When a rule comprises multiple conditions, these are combined using logical operators (`AND` or `OR`), with `AND` taking precedence. When a rule is satisfied, the associated action is immediately returned and executed.

3.3 Experimental Protocol

All experiments used a population size of 200 individuals over 150 generations, with 5% elitism. Genetic operators were applied with parameters as detailed in sections above. Each experimental run was repeated 10 times to ensure that the observed trends were not due to random fluctuations and to provide statistically significant data.

3.3.1 Experimental Variations. To explore the impact of genetic algorithm parameters and the agent configurations on the resulting agents, a number of experiments were conducted.

Baseline Configuration. The baseline for agent performance was established using an overall `MUTATION_RATE` of 0.4 and two-point crossover as the crossover method. All other parameters for the baseline run were as specified in their respective sections.

Mutation Rate Experiments. To observe the effect of the overall mutation rate on agent performance, two extreme `MUTATION_RATE`s were tested: 0.1 and 0.75. All other parameters for these experiments were kept at their baseline values.

Crossover Method Experiments. To assess how different recombination methods influence performance and convergence, the following crossover strategies were evaluated:

- Single-point Crossover: Similar to two-point crossover, but the list of rules gets split on a single point, preserving larger chunks of consecutive rules.
- Uniform Crossover: Each rule within a parent agent has a 50% chance of being swapped with the other parent.

For these experiments, the overall `MUTATION_RATE` was set to 0.4 (baseline).

Agent Configuration Experiments. To assess how playing against fixed-strategy agents influences the genetic agents' behavior, the baseline experiment was repeated with two of the agents being replaced by Pommerman's `SimpleAgents`

4 RESULTS

4.1 Baseline Performance and Initial Observations

Figure 2 shows the progression of fitness metrics across generations in the baseline setup. The blue, orange, and green lines correspond to the maximum, mean, and minimum fitness values in each generation, respectively. The shaded regions around each line indicate the standard deviation across independent runs.

As observed in Figure 2, there is a rapid increase in fitness within the first 20-30 generations, indicating a rapid adaptation to the fitness function and efficient identification of basic viable strategies. This initial spike in fitness likely reflects agents quickly learning to avoid self-kills, which are heavily penalized.

It is likely that during these initial generations, agents that received a low number of 'seeded' rules during their initialization had a lower starting fitness than agents who received the maximum (six) number of seeded rules. While this implies that agents might not

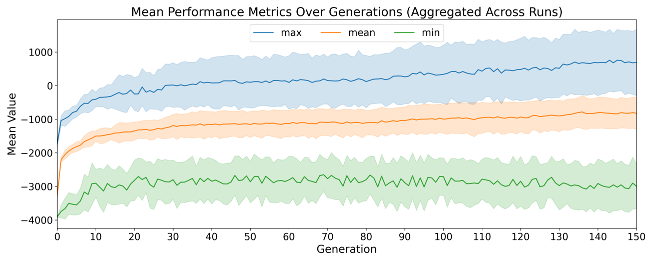

Fig. 2. Min, Max, and Mean Fitness over Generations for Baseline Experiment

be discovering complex strategies by themselves, but rather using the predetermined rules, this observation further supports the idea that the fitness rapidly improves due to agents 'learning' to avoid self-kills, and bombs in general.

Following this initial spike, the fitness shows a gradual convergence over the next 100 generations. This suggests that while agents are discovering incrementally improved strategies, they may be struggling to find novel and effective rules that lead to a significant further increase in fitness, potentially indicating convergence to a local optimum.

Furthermore, the variability across the 10 runs, as indicated by the shaded regions around the fitness curves, is notable. Specifically, the relatively large standard deviation for the maximum fitness shows that independent runs are heavily influenced by the stochasticity of genetic algorithms; some runs coincidentally find a highly efficient rule, while other runs struggle to improve.

4.2 Impact of Low Mutation Rate

Figure 3 shows the fitness curves of the experiment with a mutation rate of 0.1.

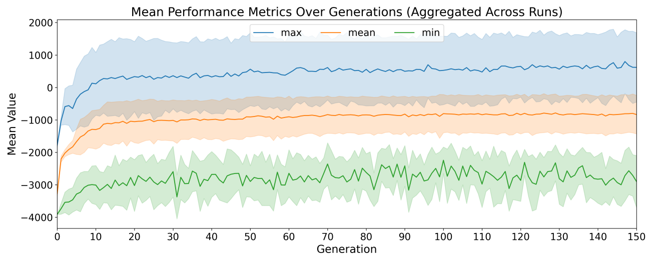

Fig. 3. Min, Max, and Mean Fitness over Generations for Low Mutation Rate Experiment

This experiment yielded curious results compared to the baseline experiment. While an initial spike in fitness was still observed, convergence occurred significantly earlier, around generation 10, in contrast to the baseline's more gradual convergence after generation 20-30. Furthermore, the variance of maximum fitness across runs was notably higher, at times even overlapping with the variance of mean fitness.

This behavior can be explained by the combination of reduced mutation rate, the rule shuffling mechanic, and the impact of the initial seeded rules. As detailed in Section 3.1.2, agents are initialized with one to six pre-determined 'seed' rules, primarily for bomb evasion. The top-to-bottom evaluation of an agent's rules means that the initial position of these seeded rules is critical for an agents immediate performance.

A lower overall mutation rate (0.1) reduces the probability that an agent will undergo any form of mutation, including the rule shuffling mutation. Consequently, agents with suboptimal initial rule ordering, where effective seeded rules are positioned lower in its ruleset, are less frequently mutated through shuffling. This allows agents with unfavorable initial rule placements to remain in the population, leading to lower average fitness. In contrast, agents that, purely by chance, receive well-positioned seeded rules perform well and keep this advantage due to the reduced rate of shuffling. This preservation of favorable and unfavorable rule orderings could contribute to the larger variance and earlier convergence observed in the maximum fitness.

4.3 Impact of High Mutation Rate

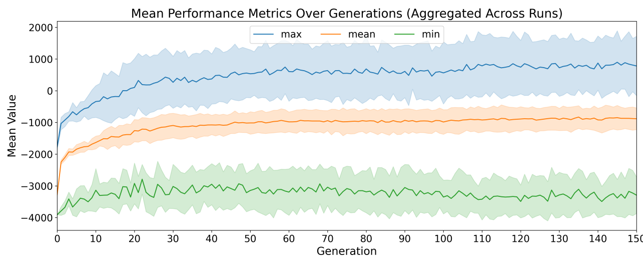

Fig. 4. Min, Max, and Mean Fitness over Generations for High Mutation Rate Experiment

The results of this high mutation rate (0.75) experiment can be found in Figure 4. The curves for minimum, maximum, and mean fitness closely resemble those observed in the baseline experiment (Figure 2). There is still an initial rapid increase in fitness, followed by a gradual convergence over the next generations.

This outcome suggests that within the tested range, increasing the mutation rate beyond the baseline does not significantly change the algorithm's ability to discover novel strategies and high-performing agents for this problem. This implies that the baseline mutation rate of 0.4 is sufficient to introduce enough genetic diversity for exploration without hindering convergence by "over-mutating" the population.

4.4 Impact of Crossover Methods

Figures 5 and 6 show the fitness graphs of single-point and uniform crossover, respectively.

When comparing the fitness curves of single-point crossover (Figure 5 and uniform crossover (Figure 6) to the baseline two-point crossover (Figure 2), distinct patterns in early generation variance emerge, highlighting their impact on the early exploration of the solution space.

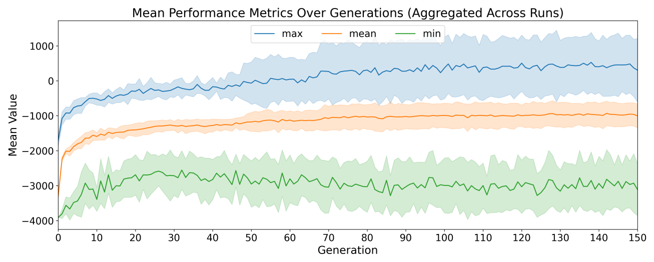

Fig. 5. Min, Max, and Mean Fitness over Generations for Single-Point Crossover Experiment

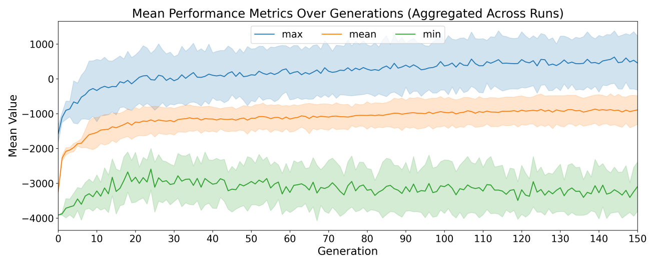

Fig. 6. Min, Max, and Mean Fitness over Generations for Uniform Crossover Experiment

Single-Point Crossover. This method shows a notable low variance in fitness across runs during the initial generations, roughly up to generation 40-50. This behavior can be attributed to single-point crossover's conservative recombination strategy, which preserves large chunks of rules and provided only nine possible crossover points for a 10-rule agent. Unlike the two-point crossover used in the baseline, this limited genetic mixing can cause agents to develop similar rule structures more quickly, reducing population diversity. The sudden increase in variance after generation 50 suggests that accumulated mutations finally created sufficient diversity for the evolution of more advanced strategies. This pattern demonstrates that while conservative crossover methods may lead to stable convergence, they can delay the discovery of novel and effective strategies.

Uniform Crossover. The uniform crossover experiment produces fitness curves that are quite similar to the baseline two-point crossover experiment, as evidenced by comparing Figures 6 and 2. Both experiments exhibit the rapid initial improvement within the first 20-30 generations, followed by gradual convergence. The uniform crossover experiment however has a larger variance in these early generations. This is likely because uniform crossover is a more aggressive crossover method, with each rule having a 50% chance of being swapped between parents. This causes a quicker initial exploration of the search space, leading to larger variance in early generations compared to two-point crossover.

4.5 Impact of Agent Configurations

Figure 7 shows the results of the agent configuration experiment.

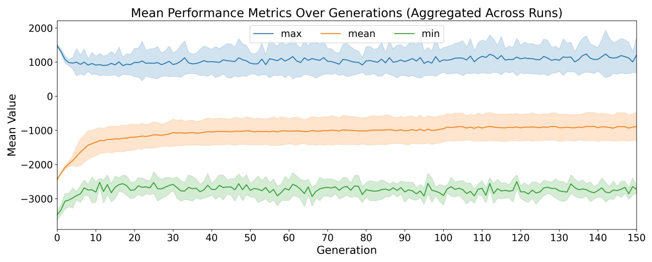

Fig. 7. Min, Max, and Mean Fitness over Generations for Two GeneticAgents and Two SimpleAgents Experiment

As shown in Figure 7, this experiment yielded significantly different and unusual results compared to the previous experiments. In particular, maximum fitness starts at a high value before decreasing and immediately converging within the first 10 generations. Simultaneously, the mean fitness still shows the initial spike in early generations, albeit slower than before, but still converges rather quickly after these initial generations.

These trends in fitness can likely be attributed to the vagueness of the path to success within the current fitness function when agents face more aggressive opponents such as the `SimpleAgents`.

When `GeneticAgents` play against other evolving `GeneticAgents`, the main path to success lies in survival, cautious bomb placement to avoid self-kills, and occasionally getting a kill. The "best" an agent can achieve is generally a balance of survival and minimal offensive action, as strategic kills are rare and difficult to consistently achieve against equally unsophisticated agents.

In contrast, `SimpleAgents` actively seek to kill opponents. While `GeneticAgents` receive points for eliminating opponents, the actual probability of a kill occurring due to strategic behavior is extremely low, especially for agents early in their evolution. When a kill *does* happen, it is often a matter of chance or circumstance rather than the direct result of a strategic, evolved play. This can result in lucky agents being selected for future generations, propagating its rules to its offspring, while not necessarily being a high-performing agent.

5 DISCUSSION

This section interprets the findings presented in Section 4, evaluating their implications in the context of the overarching research question: **How can genetic algorithms be used to automatically generate understandable guarded command programs for agents in a Bomberman environment?** The influence of the chosen program structure, mutation, and crossover strategies on the evolution of controller performance and interpretability will be discussed, drawing insights from the various experiments conducted.

5.1 Influence of Genetic Operators on Evolution

The experiments on mutation rates and crossover methods directly address **SRQ1: How do different mutation and crossover strategies affect the evolution of rule complexity and game-playing effectiveness in guarded command Bomberman controllers?**

The baseline experiment, with an overall mutation rate of 0.4 and two-point crossover, showed a rapid initial increase in fitness, followed by gradual convergence. This initial spike is likely primarily caused by agents quickly learning self-preservation techniques, either by finding optimal rule ordering for the seeded rules or by learning to be cautious with their bomb placements. The subsequent gradual convergence suggests that while the algorithm continued to iteratively improve its strategies, it reached a local optimum, struggling to discover novel rules and strategies to significantly increase the agents' performance. The observed variance across runs shows the stochastic nature of genetic algorithms across runs, where chance plays a large role in the discovery of new rules.

The low mutation rate experiment (Section 4.2) yielded particularly insightful results. The earlier convergence and the higher variance in maximum fitness can be explained by the reduced rule-shuffling mutation. This mechanism allows an agent to find the optimal execution order of its rules and plays a significant role in its performance. With fewer mutations, and thus, fewer rules shuffled, agents that coincidentally began with suboptimal rule orderings (where effective seeded rules were positioned lower in the ruleset) had a lower probability of having its rules reordered to favor these seeded rules. When a large portion of the population is initialized with these suboptimal agents, the resulting maximum fitness will be significantly lower compared to populations with more favorable rule orderings. This suggests that for these complex rule-based agents, a sufficient mutation rate, especially one that mutates the ordering/prioritization of these rules, is critical for effective exploration.

In contrast, increasing the mutation rate to 0.75 did not significantly alter the fitness curves compared to the baseline. This indicates that the baseline mutation rate of 0.4 was already sufficient to introduce enough genetic diversity for exploration in this environment. Beyond this point, further increases in mutation might lead to "over-mutating" the population, potentially disrupting beneficial traits faster than they can be propagated.

Regarding crossover methods, single-point crossover showed a lower variance in early generations and delayed the discovery of more effective rules. This conservative crossover method preserves larger chunks of rules, which limits the rate at which the population diversity increases, and slows the initial exploration of the solution space. In contrast, uniform crossover, being a more aggressive crossover method, led to a larger variance in early generations, likely indicative of a quicker and more thorough initial exploration. Despite this, its overall fitness curves closely resembled the baseline experiment, suggesting that for this specific problem, two-point and uniform crossover provide sufficient and comparable levels of genetic mixing for convergence.

Collectively, these findings show that the choice of genetic operators significantly influences the evolutionary dynamics of these guarded command programs. A balanced mutation rate, especially one that allows for internal rule restructuring (like shuffling), is crucial for escaping local optima and exploring the search space. Similarly, more aggressive crossover methods (two-point or uniform) cause a quicker initial exploration of the search space.

5.2 Performance against Fixed-Strategy Agents

The experiment pitting `GeneticAgents` against Pommerman's `SimpleAgents` provided essential insights into the fitness function's effectiveness. Section 3.1.3 outlines the rewards and penalties given to agents within this study, but this experiment has shown the weakness of these metrics.

This leads to **SRQ2: What gameplay metrics (e.g., survival time, opponents eliminated) should be used to evaluate the performance of a controller in the Bomberman environment?** In this study, the performance of a controller is primarily evaluated through a fitness function that rewards survival time, avoiding self-kills, eliminating opponents, and destroying wood or items. However, observed unusual fitness curve: a high initial maximum fitness that quickly decreased and converged, alongside a quickly stagnating mean fitness, is a strong indicator of a fundamental mismatch between the fitness function's rewards, and the desired strategic behavior against predictable, but aggressive, opponents.

In the baseline experiment (`GeneticAgents` playing against each other), the path to success for the agents primarily focused on survival and avoiding self-kills. Agents are quickly considered the "best" agent, simply by not exploding themselves and surviving till the end of the game. Strategic kills were rare and difficult to achieve consistently against equally unsophisticated agents. However, when faced against agents with a strategy, the shortcomings become visible very quickly. Playing defensive is no longer a viable strategy as your opponents are actively seeking you out, trying to trap and eliminate you. The provided fitness function is not exhaustive enough for the `GeneticAgents` to learn from this. There are no direct rewards for escaping an opponent's bomb blast, placing a bomb *near* an opponent, trapping an opponent, etc. Without these rewards, an agent does not have the ability to evolve strategic plays.

To truly evaluate a controller's performance in a dynamic environment like Bomberman, gameplay metrics should extend beyond survival and include rewards for offensive actions and defensive maneuvers against opponents, such as direct engagement, evasive actions, and tactical use of bombs to influence opponent movement.

5.3 Interpretability of Guarded Command Programs

Addressing **SRQ3: What structure and constraints on guarded command programs balance interpretability and gameplay performance?** The main reason for using Guarded Command Programs (GCPs) as the agent representation in this study is their inherent interpretability. Each rule in a GCP is composed of a guard (a logical expression) and a corresponding action. This structure is similar to human reasoning: "if this, then that." Because of this structure, these rules are practical for understandability.

The population initialization strategy discussed in Section 3.1.2 reflects an intentional design trade-off between rule expressiveness and interpretability. Rules were most frequently initialized with two conditions, based on empirical observations: rules with a single condition were too simplistic to capture meaningful decision logic, leading to overly general behaviors that executed too often and interfered with more specific rules. On the other hand, rules with three conditions were rarely useful when initialized randomly. They required a more precise match in the environment to be triggered, and their complexity made them less likely to evolve into effective behaviors early in the search process.

This bias toward two-condition rules aligns well with the goal of maintaining interpretability. Rules of this complexity are still easily comprehensible by humans, while being expressive enough to encode non-trivial behavior. However, due to the poor performance of the agents, these results are not conclusive. It is possible that more complex rules could be necessary for sophisticated decision-making, especially in more dynamic or complicated environments.

Despite this, the guarded command structure remains interpretable even as the number of conditions increases. Logical expressions with up to three conditions are generally still readable and understandable by humans, particularly if operators and condition names are semantically meaningful. Therefore, while increasing rule complexity may improve agent performance, it does not inherently compromise interpretability. This balance between expressiveness and human readability makes GCPs a suitable choice for applications where transparency and performance are required.

6 CONCLUSION

This thesis investigated using genetic algorithms (GAs) to automatically generate understandable guarded command programs (GCPs) for Bomberman agents. The research demonstrated the viability of this approach, showing that GAs can produce transparent, rule-based controllers. The findings also confirmed that genetic operators, particularly a balanced mutation rate with a rule-shuffling mechanism, are essential for exploration and optimizing rule order. The GCP structure itself, favoring simple "if-then" rules, successfully balanced gameplay expressiveness with human interpretability.

However, the most significant limitation and key finding of this study lies in the design of the fitness function. While the implemented metrics were sufficient to guide agents toward basic survival and avoiding self-destructive behavior, they failed to encourage the development of sophisticated, strategic play. When faced against fixed-strategy opponents, the genetically evolved agents were unable to adapt beyond passive tactics because the fitness function did not adequately reward complex offensive or defensive maneuvers. This highlights a fundamental challenge: for genetic algorithms to produce effective controllers, the gameplay metrics used for evaluation must be nuanced enough to recognize and reward tactical depth, such as trapping opponents or executing strategic evasions.

In conclusion, GAs can successfully generate understandable controllers for Bomberman agents, but their ability to produce strategically intelligent behavior is fundamentally limited by the sophistication of the evaluation metrics.

7 FUTURE WORK

Future work should mainly focus on improving the fitness function to reward more strategic behaviors beyond simple survival. This could involve incorporating metrics for tactical bomb placement, such as trapping opponents or controlling territory, and rewarding successful evasive maneuvers. Future research could investigate dynamic agent structures where the number of rules or their complexity can evolve, potentially allowing for more sophisticated strategies to emerge.

REFERENCES

[1] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research* 13 (jul 2012), 2171–2175.

[2] Hyun-Tae Kim and Kyung-Joong Kim. 2013. Hybrid of Rule-based Systems Using Genetic Algorithm to Improve Platform Game Performance. *Procedia Computer Science* 24 (2013), 114–120. https://doi.org/10.1016/j.procs.2013.10.033

[3] Ian Millington. 2019. *AI for Games.* CRC Press. https://doi.org/10.1201/9781351053303

[4] Nathan Partlan, Luis Soto, Jim Howe, Sarthak Shrivastava, Magy Seif El-Nasr, and Stacy Marsella. 2022. EvolvingBehavior: Towards Co-Creative Evolution of Behavior Trees for Game NPCs. arXiv:2209.01020 [cs.NE] https://arxiv.org/abs/2209.01020

[5] Cinjon Resnick, Wes Eldridge, David Ha, Denny Britz, Jakob Foerster, Julian Togelius, Kyunghyun Cho, and Joan Bruna. 2018. Pommerman: A Multi-Agent Playground. *CoRR* abs/1809.07124 (2018). arXiv:1809.07124 http://arxiv.org/abs/1809.07124

[6] Cinjon Resnick, Wes Eldridge, David Ha, Denny Britz, Jakob Foerster, Julian Togelius, Kyunghyun Cho, and Joan Bruna. 2022. Pommerman: A Multi-Agent Playground. arXiv:1809.07124 [cs.MA] https://arxiv.org/abs/1809.07124

[7] Hanli Wang, S. Kwong, Yaochu Jin, Wei Wei, and Kim-Fung Man. 2005. Agent-based evolutionary approach for interpretable rule-based knowledge extraction. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 35, 2 (2005), 143–155. https://doi.org/10.1109/TSMCC.2004.841910

[8] Georgios N. Yannakakis and Julian Togelius. 2018. *Artificial Intelligence and Games.* Springer International Publishing. https://doi.org/10.1007/978-3-319-63519-4

[9] Hongwei Zhou, Yichen Gong, Luvneesh Mugrai, Ahmed Khalifa, Andy Nealen, and Julian Togelius. 2018. A hybrid search agent in pommerman. In *Proceedings of the 13th International Conference on the Foundations of Digital Games* (Malmö, Sweden) *(FDG '18)*. Association for Computing Machinery, New York, NY, USA, Article 46, 4 pages. https://doi.org/10.1145/3235765.3235812