AI Project - WooduKonquer

מגישים:

שון צ׳רני 213716608
מיכאל מרזין 323913277
נועם הקיני 213228802

1. Introduction

1.1. Problem Description

The problem we wanted to tackle is the mobile game Woodoku: a wood block puzzle game meets sudoku grid. Woodoku is a puzzle game. In the game, the player puts different shapes made of blocks of various shapes that the player has to place on the board. The goal is to fill rows, columns, and 3x3 squares, much like in Sudoku. When you manage to do this, the row, column, or square you filled is deleted and you get a score.
It is also worth noting that performing these "combos" simultaneously will result in a higher score than performing it sequentially. The game is over when the agent/player has no more ways to put blocks on the board legally.

```
https://www.youtube.com/watch?v=Yeop5J7ewdc
```

1.2. Why This Is An Interesting problem

The primary reason we chose this game is its enjoyment. Playing it manually is challenging and requires the player to carefully consider most, if not all, of their moves. Given the complexity, we were curious to see how AI agents would tackle the game's decision-making process. For instance, given a specific game state, should a riskier, more rewarding move be chosen? Would we, as human players, make the same decision? Can the AI outperform us in score, can we outperform it?
The second motivation for us to choose this problem domain is that at first glance, we didn't have a specific strategy we thought the agent should follow, nor did we have a specified solution we learned that we thought would outperform others. This "problem" meant we had a lot of room for creativity with the various techniques we learned, and we could explore how each technique's decision-making process is reflected in this specific domain.

1.3. Modeling The Problem

To model the problem, we chose to implement the game using the Python programming language. Our first step was to get the base game functioning. To achieve this, we utilized an existing reinforcement learning environment called *Gymnasium* (refer to the Gymnasium Documentation), which was adapted to this specific game by the gym-woodoku library. This library handles the rendering and the internal logic for performing actions within the environment. Later, we encapsulated this logic within a *GameState* class to provide additional functionality. With the game running, we developed an *Agent* class that receives a *GameState* and returns an *Action*. All of our agents will need to implement this function in

their own unique way.

1.4. Solving The Problem

To solve the problem we decided to explore 2 paths we learned in class:

- Adversarial Search - We chose to implement methods that proved useful in class, such as $ReflexAgent$, $AlphaBetaAgent$, and $ExpectimaxAgent$. We aimed to examine how different evaluation functions used by these methods affect the final result.
- Reinforcement Learning - Given the environment designed for it, we wanted to apply model-free methods we've learned, like $Q-learning$. This way we see how effectively the agent can learn optimal strategies through trial and error.

2. Previous Works

Initially, we searched for existing works that could inspire potential solutions for this game. From the start of our research, we realized that there are multiple ways to approach the problem of achieving optimal play in Woodoku. We discovered several different solutions, each offering a unique perspective and new ideas for tackling the problem. Some approaches focused on search problems and heuristics, others explored learning and model training, while some treated it as an adversarial search.

2.1. First Solution - Heuristic

```
https://github.com/Zeltq/WoodokuSolver/tree/main
```

This solution can be classified as a rule-based heuristic agent with an evaluation function, it is designed to make decisions based on a predefined set of rules and heuristics. The heuristic prioritizes choosing moves that are likely to give high scores based on the evaluation function, this evaluation function tries to capture strategic elements of the game, allowing the agent to make more informed decisions. Using this heuristic the agent gets a deterministic decision-making process which is based only on the current state of the game. Every 3 steps the agent evaluates all possible 3 block sequences of action, meaning that every 3 steps it chooses the next 3 actions. The evaluation function's key points are:

1. The evaluation function penalizes certain configurations, such as isolated 1 values surrounded by 0.
2. The function rewards placing blocks on the board's edges.
3. The function looks for specific patterns, such as an L shape of 1 value, and modifies the score accordingly.

Its important to note that the implementation of WoodokuSolver was done with a different score system then our own. We made small modifications in the WoodokuSolver code in order to align their score system with ours to compare results in the relevant sections. (The linked git page contains the original code of the creator).

2.2. Second solution - DFS

```
https://github.com/CosmicSubspace/WoodokuAI/tree/master
```

This solution uses DFS (Depth-first search) to explore outcomes of the different actions that can be taken. The agent starts from the current game state and tries all possible placements of the current piece. For each valid placement, it generates a new game state and recursively explores further moves, continuing this process to a set depth (targetDepth). Each time the agent finds that a path is not worth further exploring, it backtracks and tries other branches. To evaluate a game state the program uses two functions that favors empty blocks and separated spaces.

2.3. Third solution - RL

```
https://github.com/helpingstar/gym-woodoku/tree/main
```

This solution was done on the gym-woodoku environment on which we based our game's logic and rendering. While there wasn't sufficient documentation about the project's implementation or methods, we found a video summarizing the project results. This solution probably used reinforcement learning to train an agent.

2.4. Previous Works Summary

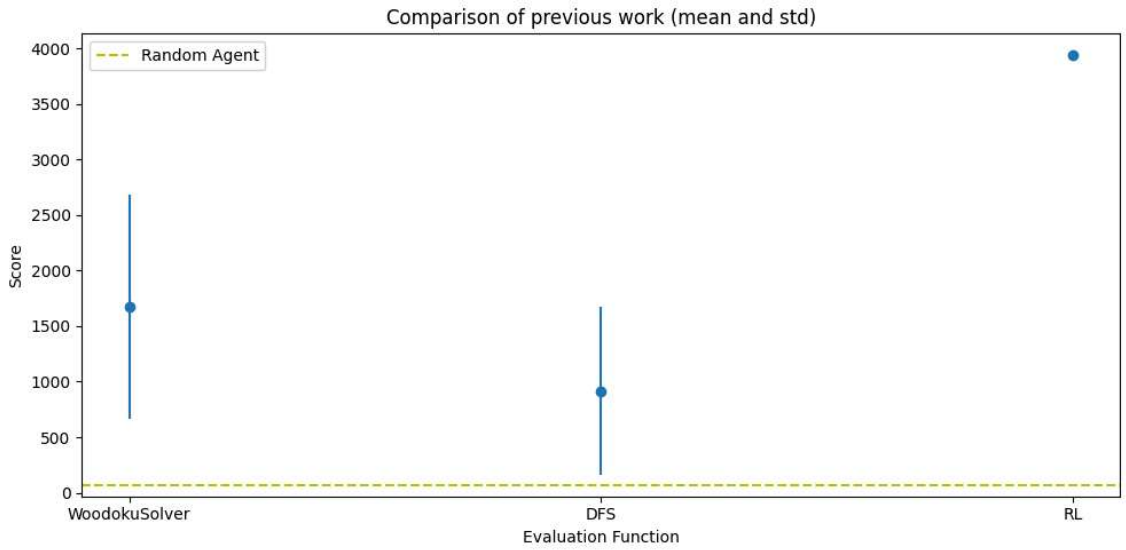

Figure 1: Previous Work Result

3. Methodology

3.1. Defining the Woodoku Game

3.1.1. Woodoku Sequence (trajectory)

We start by clearly defining the problem of Woodoku. A game of Woodoku is a continuous sequence of Woodoku Game States (which we will note as S), Woodoku Actions (which we will note as A), and Woodoku Rewards (which we will note as R). Meaning its a sequence of the format:

$$\tau = (s_i, a_i, r_i)_{i=1}^{N};\ s.t:\ \forall i \in N\ \rightarrow s_i \in S \wedge a_i \in A \wedge r_i \in R$$

3.1.2. Woodoku Game State

The valid state s in the Woodoku Game States S is compose of 2 things:

1. The board B which is a $9x9$ matrix of values of $\{0, 1\}$. Meaning: $B \in M_{9x9}(\{0, 1\})$.
2. A triplet (b_1, b_2, b_3) of the currently available blocks that we can position. They are $5x5$ matrices of values of $\{0, 1\}$. Meaning: $\forall i \in [3] : b_i \in M_{5x5}(\{0, 1\})$. In practice they are a finite amount of shapes (47 of them).

3.1.3. Woodoku Action

The action a in the Woodoku Action A is a number between $[0, 242]$ which represnt the following:

1. for $a \in [0, 80]$ we place b_1 at $[(a - 0)\ //\ 9,\ (a - 0)\ \%\ 9]$.
2. for $a \in [81, 161]$ we place b_2 at $[(a - 81)\ //\ 9,\ (a - 81)\ \%\ 9]$.
3. for $a \in [162, 242]$ we place b_3 at $[(a - 162)\ //\ 9,\ (a - 162)\ \%\ 9]$.

For example: given the action $a = 212$. We place th center of b_3 at position $[(212 - 162)\ //\ 9,\ (212 - 162)\ \%\ 9] = [5, 5]$.

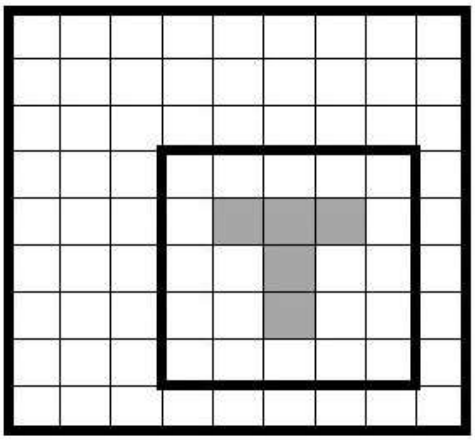

Figure 2: Action Example

3.1.4. Valid Woodoku Game

An action a will be valid if the block represented can be fitted entirely in the board B without overlapping any other taken cell in the board B. The taken cells are all the cells with 1, and we only look at the values of 1 inside the block we are checking for overlap. The state s_{i+1} will be valid if it is a direct result of applying action a_i on s_i, while applying action a_i on state s_i we add to B ones in the corresponding part of the added block, and remove 1 if we complete a row, column, or one of the designated $3x3$ squares. In addition, we remove the corresponding block from the suggested blocks of the state. When s_i get to a situation when there are no blocks left to place, the game generates 3 new random block to s_{i+1}.
In the case where there is not a single valid move from all 243 moves, in other words, we can't fit any block of s_i into B, the game stops and we lose.
A sequence τ will be valid if every state and action of it is valid.

3.1.5. Woodoku Reward

Before defining the reward we define "combo", a "combo" is the total number of rows, cols and $3x3$ squares which are fully occupied after block placement phase and before the crashing blocks phase (as the value 1).
We also define the term "straight", a "straight" is the number of consecutive steps in which blocks were crushed untill reaching the current state. For instance, if in the previous step I crushed some blocks on the board and in this step i do an action that also crushes some blocks on the board, my straight is currently at value 2. On the next step, if my action wont cause any blocks to crush, my straight value will reset to 0.
We will also define "n_cells" as the number of bricks in the block that was put on the board.
Given those definitions the reward r_i is defined by: $28 \cdot \ combo + 10 \cdot straight + n\_cells - 20$.

3.1.6. Woodoku Opponent Agent

To apply adversarial search in Woodoku, we need to accurately model opponent agents.

Although Woodoku is a single-player game, it includes a randomization element. Every 3 turns, a set of blocks to place is chosen at random. We can define our adversarial agent as the one responsible for selecting these blocks. Formally, the agent operates as follows:

- If there is at least one block left to place, it takes no action.
- Otherwise, choose a set of 3 new blocks for the player to place. Those blocks are put in the available blocks slots and the slots become placable again.

The opponent selects the next 3 blocks in the game, leading to 47^3 possible combinations. To reduce the large search space, the focus is shifted to combinations of "worst-case" blocks. Each "worst-case" block ensures that if it can be placed, so can the corresponding "regular" block. This approach narrows the possibilities to 13^3 combinations.

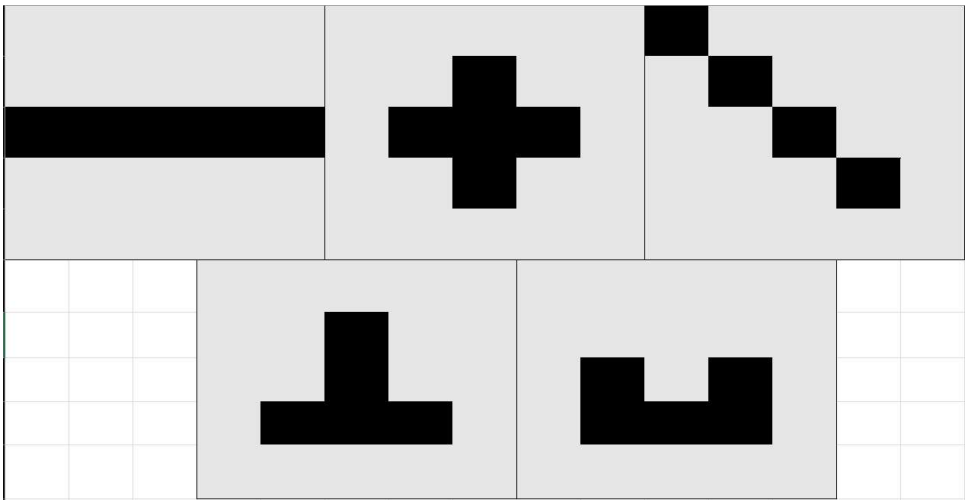

Figure 3: "Worst Case" blocks without rotations and shifts

3.2. Defining The Agent

Every agent will receive the current game state s_i and will return a valid action a_i. The assumptions of the agent will be:

1. The agent knows the current game state, what valid actions are for every state, the given reward for them, and all the possible shapes of blocks (we are not in the $POMDP$ scenario). Justification: The player is presented with the actual game state at every step.
2. The agent can simulate games, or part of them, before choosing an action, or do so before the game starts ("training"). Justification: The player can think before deciding to take a move.
3. The agent can learn from one iteration to the next, meaning that choosing one action can influence the decision of the subsequent action. In practice, this is represented by making the agent a class with persistent variables between iteration, rather than just a single function. Justification: The player can use previous run knowledge in the current run.

4. The agent does not have knowledge of the future. Every three steps in the game, a random event occurs where three new blocks are generated, and the agent doesn't know what those blocks will be. Justification: The player does not know the next 3 blocks he will get.

The agent has a general definition for a reward function, meaning that swapping out the reward function is fairly easy. On the other hand, swapping the game state and action definitions will prove to be quite difficult and will require change of the agent's strategy.

3.3. Success Criteria

The success criteria for all the agents is to achieve the highest mean score, with reasonable running times (of about 1-2 minutes or less per step) and consistency between results.

4. Results

4.1. General Timeline

Building on previous work with heuristic functions, search using DFS, and $Reinforcement\ Learning$, we've decided to tackle the Woodoku problem by exploring $Adversarial\ Search$ and applying the concepts of RL learned in class.
We began with a trivial agent, the $Random$ agent, to establish a simple baseline for the score. With this foundation, we moved on to developing $Adversarial\ Search$ agents.
The first agent we developed was the $Reflex$ agent, which served as a helpful testing ground for developing and refining various evaluation functions that would later be utilized by the adversarial agents. After identifying useful evaluation functions, we designed the $MiniMax$ adversarial agent, followed by its time-optimized counterpart, the $Alpha-Beta\ Pruning$ agent. Given the game's inherent randomness, we also wanted to consider the probabilistic outcomes of moves, so we explored the $Expectimax$ agent.
After gaining insights into the scores with adversarial agents, we shifted our focus to implementing RL agents as covered in class. Entrusting learning to trial and error, we chose to implement the model-free $Q-learning$ agent. After several test runs, it became apparent that the abundance of states and action spaces prevented Q-learning from successfully learning the environment. To address the aforementioned issue, we utilized a feature extraction technique using the $Approximate\ Q-learning$ agent.

4.2. Adversarial Search

4.2.1. Random agent

The $Random$ agent lacks significant logic and will act as a minimal baseline for evaluating other agents' performance. Its score ranges from 40 to 170, with an average of around 71.

4.2.2. Reflex agent

The $Reflex$ agent operates with very simple logic, selecting actions based solely on the current state to determine the next state, without considering any previous history. It chooses the next state that maximizes a given utility function. Formally, given state s_i and utility function O that agent will pick the action:

$$a_i = max_{a \in legal\_action(s_i)}\{O(s_i, a)\};$$

Only maximizing the next state lacks the complete consideration of the overall strategy or long-term consequences, which may lead to suboptimal decisions in the environment. However, this property of the $Reflex$ agents can be very useful for testing different utility functions, as it allows for a clear examination, with a GUI interface, of how the function influences the agent's actions when it solely focuses on maximizing it.

4.2.3. Evaluation functions

We have (number) basic evaluation functions with defined purposes:

1. check_score - This function returns the score that is displayed in the game after applying the given action on the given state. Formally, given a state s_i and an action a this function returns the following score:

$$Score = \begin{cases} n\_cells & combo = 0 \\ 28 \cdot combo + 10 \cdot straight + n\_cells - 20 & else \end{cases}$$

2. avoid_jagged_edges - This function returns the number of jagged edges on the board after applying an action. A jagged edge is an empty brick that is surrounded from every direction by occupied bricks or the edges of the board. The formal definition of a jagged edge is as follows:

$$(x, y) \text{ is a jagged edge on board } B \iff$$

$$\iff N \ni x, y \wedge \left(\sum_{i = min(1,x-1)}^{max(9,x+1)} \sum_{j = min(1,y-1)}^{max(9,y+1)} B_{i,j} \right) - B_{x,y} = 8$$

where B is a 9x9 matrix representing a legal Woodoku board.
Formally, given a state s_i and an action a the function returns the following score:

$$Score = \sum_{(x,y) \in J_B} 1$$

$$where\ J_B = \{(x, y) | \ (x, y) \text{ is a jagged edge on board } B\}$$

3. square_contribution - This function returns the maximum number of occupied bricks within a crashable square on the board. The logic behind this function is that the agent should try to maximize the number of bricks within a crashable square so that more opportunities to crash the square will appear in the future. Formally, given a s that has board B the function returns:

$$Score = max_{1 \le x \le 3,\ 1 \le y \le 3} \left(\sum_{i = min(1,x-1)}^{max(9,x+1)} \sum_{j = min(1,y-1)}^{max(9,y+1)} B_{i,j} \right)$$

4. connected_components - This function returns a penalty based on the connected components currently on board. The function takes into consideration both the number of connected components on board and the convexity of the components. In addition, we will penalize even more from none convex components, if the component is convex we will penalize a little bit less, a component is convex if for each pair of points in the component, the line segment connecting them is fully inside the component.

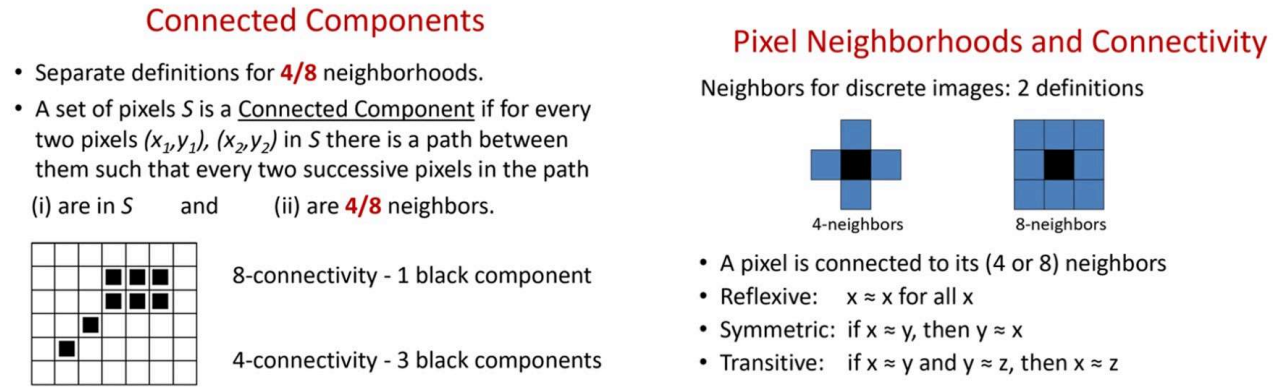

Figure 4: Definion for connected components and neighborhoods

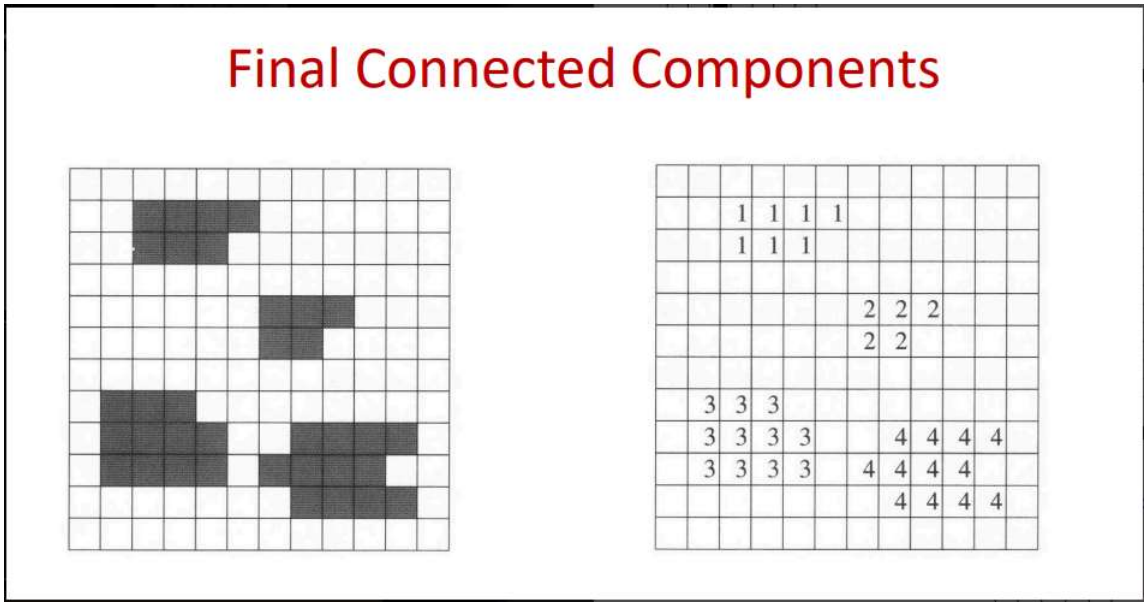

Figure 5: Examples of connected components

5. num_empty_squares - This function returns the number of empty $3x3$ squares currently on the board.
6. avoid_jagged_edges_diag - This function returns the number of sharp edges currently on the board. given a board B, a sharp edge is defined as follows:
   $(x,\ y)\ is\ a\ sharp\ edge\ \Leftrightarrow\ x,\ y \in N \wedge\ B_{x,y} = B_{x+1,y+1}\ \wedge\ B_{x+1,y} = B_{x,y+1} \wedge\ B_{x,y} \neq B_{x+1,y}$
   Therefore, this function returns:

$$Score\ = \sum_{(x,y)\ is\ a\ sharp\ edge\ of\ B} 1$$

8. remaining_possible_moves - This function returns the number of legal actions available at the current state as previously defined.
9. best_evaluation - This function returns a score representing a weighted combination

of the several basic evaluation functions we've came up with. The aforementioned combination achieves the best results for $Reflex$ agent and great results for most other agents.

$$Score = 15 \cdot remaining\_possible\_moves + connected\_components + \\ + square\_contribution^2 - 3 \cdot avoid\_jagged\_edges + 2 \cdot num\_empty\_squares - \\ avoid\_jagged\_edges\_diag$$

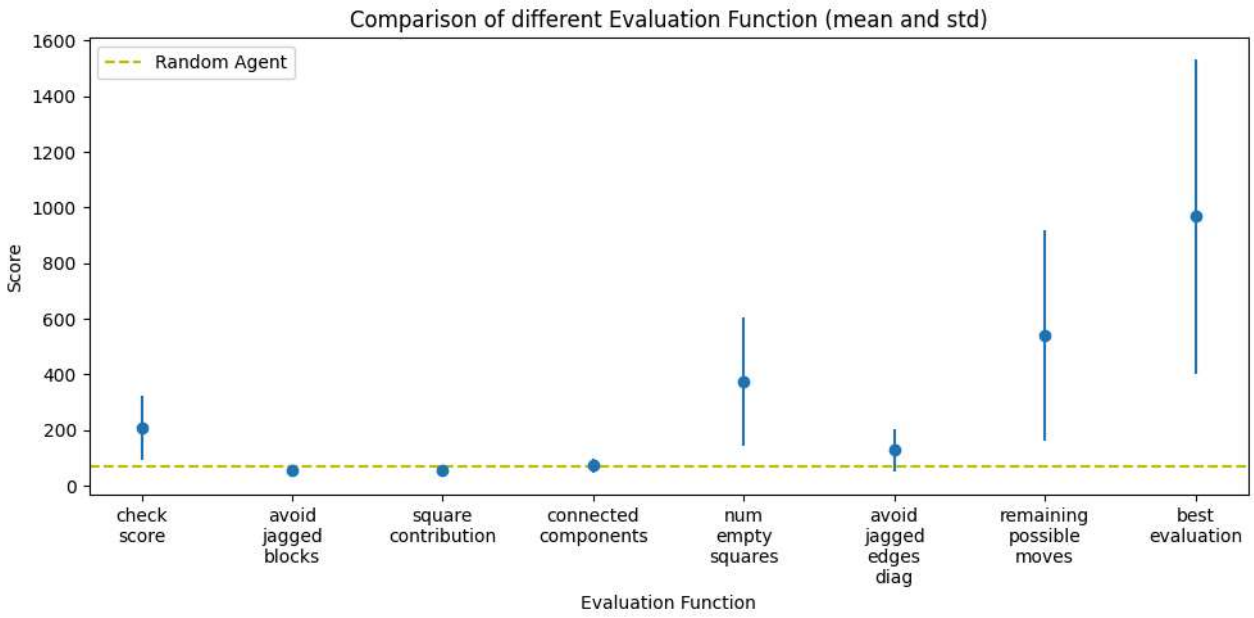

Figure 6: Evaluation Function Summary

As we can see, most evaluation functions weren't enough on their own to make the cut and suppress random choice, but when the evaluations were combined in a certain way it made a substantial impact on the results. It makes sense since the few functions that cleared the cut detect opportunities to crash blocks and the other functions that didn't cut help improve the situation on the board so that more opportunities to crash blocks arise, thus making their combination very powerful.

4.2.4. Multi-Agents

Given the randomness inherent in the game, it's apparent that the reflex agent would struggle in such states. This is because the agent cannot accurately evaluate the next state due to its uncertainty about the blocks it will be given in the future. To address this issue, an adversarial search agent can be employed, using the model of the opponent agent.

4.2.5. Mini-Max and Alpha-Beta Pruning agent

Starting with the MiniMax agent, the logic it follows is: "Find the action a that leads to a state where, in the worst-case scenario, the utility is maximized". To improve runtime, we can incorporate

$Alpha - Beta\ Pruning$, which allows us to cut off branches early and avoid evaluating certain states. In the Woodoku game, the agent will choose the action that maximizes utility for the worst-case set of three blocks picked by the opponent agent.
Its score ranges from 286 to 1889, with an average of around 832.

4.2.6. Expectimax agent

Given that Woodoku is a 1-player game with a significant element of luck, we decided to implement a more suitable agent to handle the randomness. An $Expectimax$ agent was used, following the logic: "Find the action a that leads to a state where, in the average-case scenario, the utility is maximized." In the Woodoku game, the agent will choose the action that maximizes utility for the average-case set of three blocks picked by the opponent agent.
Its score ranges from 203 to 2321, with an average of around 730.

4.3. Reinforcement Learning

4.3.1. Q-learning

Shifting our focus to Reinforcement learning, we perceived to implement the model-free approach with $Q - learning$. in this approach, the agent's actions at each state are determined by a function we'll refer to as the agent's policy, denoted by π, where $\pi : S \rightarrow A$.
The goal of $Q - learning$ is to estimate the function $Q^*(s, a)$ which define to be:

$$\textit{The expected reward we get if we follow the policy optimal policy } \pi^* \\ \textit{from the state s and doing the action a}$$

Once we have this estimate, we can derive the policy by choosing the action that maximizes the Q-value: $\pi(s) = argmax_a Q(s, a)$. The agent learns by following iteration process:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha\Big[\underbrace{r_t + \gamma \cdot max_a\{Q(s_{t+1}, a)\}}_{Current\ Approximation} - \underbrace{Q(s_t, a_t)}_{Previous\ Approximation}\Big]$$

Where γ is the discount factor that models the importance of future rewards, and α is the learning rate that determines how much new information overrides the old estimate. Both are hyperparameters of the model. Additionally, it is important to note that s_t and a_t are chosen with $\epsilon - greedy$ strategy. This strategy balances exploration and exploitation by selecting random actions with probability ϵ (exploration) and choosing the action that maximizes the Q-value with probability$1 - \epsilon$ (exploitation).

4.3.2. Hyperparameters tuning

As explained, we have 3 hyperparameters:

- learning rate α - The learning rate determines to what extent the newly acquired information overrides the old estimate.
- exploration rate ϵ - This parameter that controls the balance between exploration and exploitation in the $\epsilon - greedy$ strategy. If ϵ is high, it encourages the agent to explore a

broader range of actions and states,

- discount factor γ - High discount factor takes more consideration on future rewards, this fact encourages the agent to take safer paths because the future has more value to it.

Its score ranges from 83 to 325, with an average of around 167.

4.3.3. Approximate Q-learning

Using a Q-table has one main limitations:
It requires that we visit every reachable state many times and apply every action many time to get a good estimation of $Q^*(s,a)$, so, if we never visit a state s we have no estimation of $Q(s,a)$ even if we have visited states that are very similar to s, in our problem it is even more problematic becuase we have a huge $state\ space$ $\left(2^{81} \cdot \binom{Number\ of\ blocks}{3}\right)$ and many possible actions.
In approximate Q-learning we store features and weights, not states, what we need to learn is how important each feature (learn it's weight) for each action.
The feature vector $f(s,a)$ consists of N features. Each feature extractor $f_i : SxA \rightarrow \mathbb{R}$ extracts the $i-th$ feature from the (s,a) pair. $f(s,a) = (f_i(s,a))_{i=1}^{N}$. And the weight vector w of size N: one weight for each feature-action pair. w_i defines the weight of a feature i for action a.
Given a feature vector f and a weight vector w, the Q-value of a state and action is a linear combination of feature and weights.:

$$Q(s,a) = \sum_{i=1}^{n} f_i(s,a) w_i$$

At each step of the learning we will update the weights as follows:

$$w_i = w_i + \alpha[r + \gamma \cdot max_{a'} Q(s',a') - Q(s,a)] \cdot f_i(s,a)$$

4.3.4. Feature Extraction

In standard Q-learning, the agent maintains a Q-table that maps every possible state-action pair to a corresponding Q-value. However, when the state and action spaces are enormous such as in Woodoku, storing and updating Q-values for each pair becomes computationally infeasible. Furthermore, the learning process is damaged since the agent rarely visits the same state and action pair twice.
In order to encapsulate all the information or most information about states in a compressed way we tried several $FeatureExtractors$:

- The $IdentityExtractor$ serves as a simple baseline by assigning a value of 1.0 to each state-action pair, treating every pair as an individual feature.
- In contrast, the $SmartExtractor$ captures specific aspects of the Woodoku game, such

as the combo count, game score, the number of straight lines completed, the number of stuck cells (cells surrounded by others), and the change in empty cells (cell diff). These features provide a more nuanced understanding of how actions impact the game state.

- The $EstimationExtractor$ and $EstimationDifExtractor$ focus on the placement of blocks within 3x3 squares, with the former valuing the current and future board states regarding to the action and the latter emphasizing the difference in board configuration after the action. Both extractors assess the number of filled cells and the impact on future moves. a feature is a tuple of compressed board state (3x3 board where each brick represents a 3x3 square on the real board and its value represents how much bricks are occupied in that square), the coordinate of the 3x3 square in which the action dictates the block should be placed in and a block type.

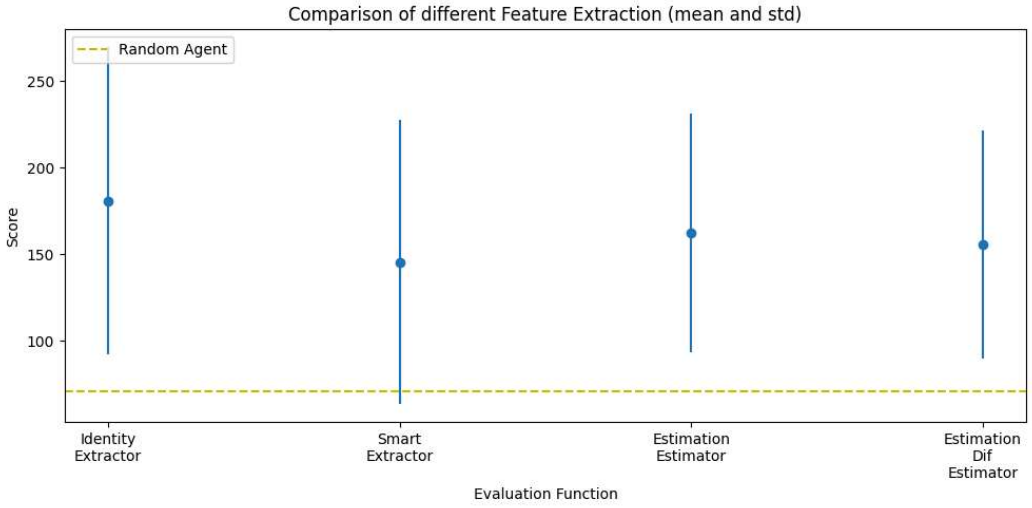

Figure 7: Feature Extraction

The different feature extraction techniques didn't make a major difference and their results were worse compared to previous agents.

4.4. Comparing The Results

Running the agents $10x$ times results in:

Method	Avg Score	Min Score	Max Score
$Reflex$	781	242	1692
$AlphaBeta$	832	197	1458
$Expectimax$	730	915	2321
$Q-Learning$	167	83	325

Approximate $Q-learning$	180	43	290

Table 1: Results Summary (mean and min/max)

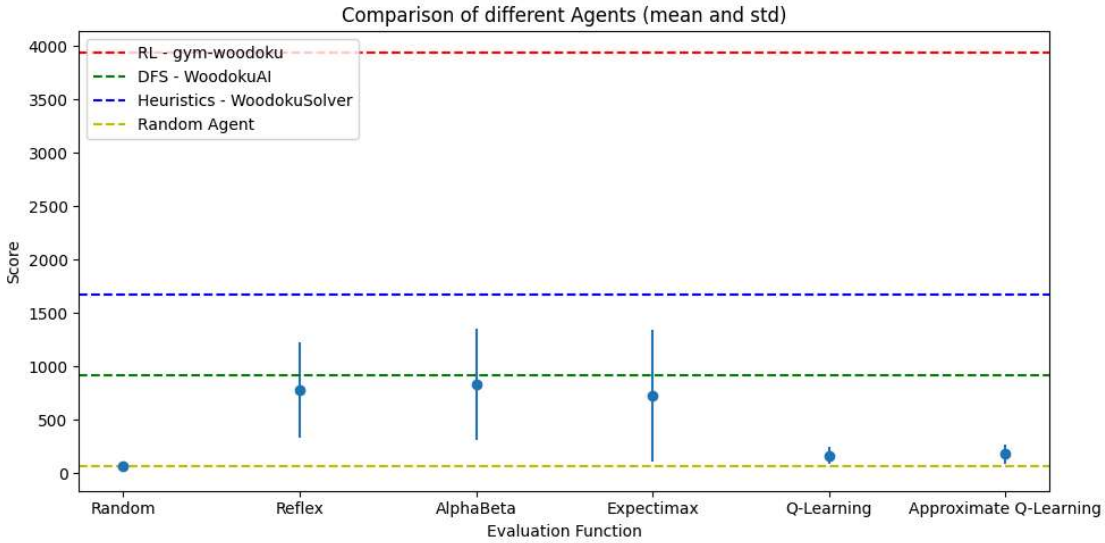

Figure 8: Results Summary (mean and std)

The results clearly indicate that the adversarial agents outperformed the random agent, delivering encouraging outcomes. Among the adversarial agents, performance levels were quite similar, with $Alpha-Beta$ pruning slightly surpassing the $Reflex\ agent$, and the $Reflex\ agent$ marginally outperforming $Expectimax$. Notably, the variance in performance for the $Reflex$ and $alpha-beta\ pruning$ agents was smaller compared to $Expectimax$, which showed higher results in some runs but, in other cases, performed on par with the random agent. $Q-Learning$ and $Approximate\ Q-Learning$, meanwhile, both yielded results akin to the random agent, with Approximate $Q-Learning$ showing a slight edge over standard $Q-Learning$. A key takeaway is that the adversarial agents were the closest to achieving results comparable to WoodokuAI. Although the results from WoodokuSolver are still significantly better then those of the adversarial agents, it's important to note that WoodokuSolver's algorithms are highly inefficient in terms of runtime complexity, whereas the adversarial agents make relatively swift decisions while still achieving impressive results. Based on our experience as players, it's challenging to maintain an average score near 1000.

5. Summary

5.1. Problem Description

The problem in Woodoku is to achieve the highest possible score by strategically crashing blocks on a 9x9 board. Blocks are crashed by filling an entire row, column, or 3x3 square. Success depends not only on maximizing score through block placements but also on ensuring survivability, which involves preventing the game from ending by keeping enough board space for future block placements. Challenges in improving survivability include managing random block sets, avoiding unreachable areas on the board, and finding consistent strategies to clear blocks and maintain open spaces.

5.2. Project General Timeline

In this project, we explored various approaches to solve the Woodoku problem, starting with simple agents and advancing to more sophisticated methods. We began with a baseline $Random$ agent and then developed a $Reflex$ agent to test and refine evaluation functions. Building on these foundations, we implemented adversarial search agents such as the $MiniMax$ agent and its time-optimized version, $Alpha-Beta$ Pruning. To address the game's randomness, we introduced the $Expectimax$ agent for handling probabilistic outcomes. Finally, we transitioned to reinforcement learning, developing both $Q-learning$ and $Approximate\ Q-learning$ agents, experimenting with feature extraction techniques to manage the game's large state and action spaces.

5.3. Adversarial Search

5.3.1. Random Agent

The $Random\ agent$, which serves as a baseline, selects actions without logic, and its scores typically range between 40 and 170, with an average of 71. This agent provided a low-performing benchmark to compare against more advanced agents.

5.3.2. Reflex Agent

The $Reflex\ agent$ is slightly more sophisticated, selecting actions based on immediate utility without considering long-term consequences. This agent allowed us to experiment with various evaluation functions, such as maximizing the game score or minimizing jagged edges. While this approach improved the agent’s decision-making, its lack of strategic foresight led to suboptimal results in complex game scenarios.

5.3.3. Evaluation Functions

Several evaluation functions were designed to guide the Reflex and adversarial agents:

$$check\_score$$

$$avoid\_jagged\_edges$$
$$square\_contribution$$
$$connected\_components$$
$$remaining\_possible\_moves$$

Combining these into a weighted evaluation function resulted in better decision-making for the $Reflex\ agent$ and also performed well with adversarial search agents.

5.3.4. MiniMax and Alpha-Beta Pruning Agents

The $MiniMax\ agent$ performs well by selecting actions that maximize utility in the worst-case scenario. Incorporating $Alpha - Beta\ Pruning$ improved its runtime by eliminating unnecessary branches. This optimization allowed the agent to make decisions more efficiently, but it still struggled with the randomness of future block placements.

5.3.5. Expectimax Agent

The $Expectimax\ agent$ accounts for randomness by selecting actions based on the average case scenario, rather than the worst case, as done by $MiniMax$. This approach yielded better results in Woodoku, where the set of blocks presented to the player is unpredictable, making the $Expectimax\ agent$ more suited to handle the game's probabilistic nature.

5.4. Reinforcement Learning

5.4.1. Q-learning

When implementing the model-free $Q - learning\ agent$, we encountered difficulties due to the enormous state and action spaces. The agent's learning process was slow, and it struggled to converge, mainly due to the need to explore a vast number of state-action pairs. Despite several attempts to adjust the learning rate, exploration rate, and discount factor, the traditional Q-table method proved inadequate for learning a reliable policy in this environment.

5.4.2. Approximate Q-learning

To try to overcome the limitations of the Q-table, we implemented $Approximate\ Q - learning$, which relies on feature extraction rather than visiting every possible state. We developed several feature extractors to simplify the state-action space.

5.5. Comparing Results

Our varied solutions to the Woodoku problem excelled in different aspects, such as handling randomness and optimizing speed while maintaining reasonable scores. A key assumption throughout the project was that the blocks available to the player were drawn from a limited pool of 47 known types, including rotated variations. This assumption simplified the problem, as more challenging configurations, such as blocks with widths or heights of 5 bricks, were

rarely included.
Additionally, the vast number of possible random block configurations was difficult to fully encapsulate due to the high computational complexity. To maintain feasible runtimes, our multi-agents were given only a small subset of all possible combinations of 3 blocks. We believe there may be a method to account for all potential blocks without iterating over every combination, which could significantly improve the performance of both the $Expectimax$ and $MiniMax$ agents without extending runtimes. However, we have not yet discovered an efficient way to achieve this.

5.6. Summary

In this project, we explored different AI approaches for solving the Woodoku game, moving from basic agents like Random to more advanced models such as $Expectimax$ and $Approximate\ Q-Learning$. Contrary to expectations, Adversarial search outperformed Q-Learning, despite the environment being designed for the latter. We experimented with evaluation functions and state-action representations, gaining valuable insights into game strategies. While we faced challenges, such as managing random generation and balancing performance with runtime, most were resolved. However, improvements in areas like block space representation and feature extraction remain for future work.