Gumoko: Best AI Player

Link to the project code

Or Cohen
Hebrew University of Jerusalem
Or.cohen7@mail.huji.ac.il

Yoni Ankri
Hebrew University of Jerusalem
Yoni.Ankri@mail.huji.ac.il

Eran Ella
Hebrew University of Jerusalem
Eran.Ella@mail.huji.ac.il

I. Introduction

Let's talk about a game that's simple, yet full of endless possibilities - Gomoku. Also known as Five in a Row, it's a classic board game where two players take turns placing black or white stones on a grid, with the goal of getting five stones in a row - horizontally, vertically, or diagonally. At first glance, it seems like a straightforward game, right? Just line up five stones. But here's the twist: beneath that simplicity, Gomoku hides an immense world of strategy, traps, and clever tricks. And that's where our AI agent comes into play.

Now, why would we want to create an AI agent for Gomoku? Well, imagine this: Gomoku is not only fun to play, but it's also a perfect playground for an AI to stretch its thinking muscles. The game demands careful planning, pattern recognition, and making decisions based on the entire board's state. It's like chess, but faster and leaner. And that's why it's an excellent challenge for an AI agent - there are countless possible moves to consider at any point in the game, making it a beautiful puzzle for an agent to solve.

But here's the catch: it's not enough for our AI agent to be a master strategist that can eventually win the game. We want it to be quick - like, really quick. Nobody wants to wait minutes for a single move, right? Sure, we could build an agent that exhaustively checks every possible move like a methodical detective using a technique like minimax. But that would take forever. Instead, we're aiming to build an AI that makes smart decisions very fast. Our goal isn't just victory - it's speed and victory.

The fun challenge? Finding the right balance. We need an agent that's not only clever enough to outwit an opponent but also fast enough to make snap decisions without overthinking. Can we create an AI that balances both power and speed? Well, that's exactly what this project is all about.

To make our process smoother, we also used Large Language Models (LLMs) for some parts of the project—like ensuring we didn't miss any grammar issues in this report and for small coding tasks, such as writing scripts for plots. LLMs helped us streamline these aspects, allowing us to focus on developing and refining our AI agent.

II. Previous Work

As we set out to build an AI for Gomoku, we found that no previous projects had been tackled in our course. Naturally, we turned to the internet to explore how others approached this classic game. Our goal was to learn from their experiences, uncover common strategies, and see if we could implement them efficiently. One well-known example is an AI developed by ByteDance, which claimed near-perfect victories when it started first [1]. This underscored the importance of turn order in Gomoku, something we had to factor into our own AI design. A key concept we found discussed on Stack Overflow was the use of a score function to evaluate the board based on continuous stones of the same color [2]. This idea influenced our own scoring system, allowing the AI to assess moves quickly. Vinicius Petratti shared his approach to designing a Gomoku AI, using a heuristic-based score function that balanced offensive and defensive strategies [3]. His work helped refine how we evaluated board states, ensuring our AI could adapt to different situations. Luke Salamone detailed the use of Minimax with Alpha-Beta Pruning to optimize decision-making [4]. Like him, we incorporated Alpha-Beta Pruning to reduce unnecessary calculations and speed up our AI. In addition, a GitHub project by MichalisPer implemented pattern-based AI [5], which inspired our heuristic design. The focus on specific patterns, like preventing critical moves, gave our AI an edge in recognizing threats. For a more advanced approach, we examined a deep learning model by slcz [6]. Although deep learning requires significant computational power, it introduced us to the potential of reinforcement learning in complex games like Gomoku.

Regarding benchmarks, there isn't much standardized data for Gomoku AI performance. In our case, we measured success by tracking which agent won the game and how many moves it took to complete. As for common assumptions, we noticed a shared belief in the community that the most relevant moves are those placed near existing stones. It's rare for an agent to place a stone far from the center of action, as most meaningful plays cluster around established formations. By pulling from these resources, we crafted an AI strategy that combines the strengths of score functions, heuristics, and search optimizations to build a smart, efficient agent.

III. Methodology

Now, let's dive into the heart of the project, where we experimented with several different brains for our Gomoku-playing agent. Our team didn't just stop at one approach—we tried quite a few, each with its own unique flair. The algorithms we explored were Q-Learning, Monte Carlo Tree Search

(MCTS), Expectimax, and Alpha-Beta Pruning. Each of these algorithms came with high hopes, but in practice, well, they didn't all live up to the hype. Let's break it down.

First up, we have **Q-Learning**. This one is like the adventurers of the AI world, always exploring, trying to learn from every little move they make. Normally, this algorithm is a rock star in environments where the AI can gradually learn from its mistakes and adjust its actions over time. But in Gomoku? It's a different beast. The board is structured, the rewards only come at the very end of the game, and the search space is enormous. Imagine trying to learn the best path through a maze where you only get feedback when you exit, and nothing in between. Our Q-learning agent couldn't generalize well enough to figure out smart strategies. They also required way more computational power and training time than we could realistically provide. Without massive resources, this algorithm is simply too slow to converge on effective strategies. To make things worse, coming up with a reward function that captured the complexity of Gomoku felt like trying to catch lightning in a bottle. It just didn't work out.

Next, we turned to **Monte Carlo Tree Search (MCTS)**. MCTS is all about possibilities. It explores tons of future scenarios, randomly trying things out and then figuring out which path seems the most promising. To give MCTS a better shot at handling Gomoku, we used a heuristic function to guide it towards the most relevant moves, just like we did with Alpha-Beta Pruning. This way, MCTS didn't have to search the entire game space but could zoom in on likely candidates. After each simulation, it used a score function to evaluate the board and decide which move looked the best.

One of the biggest challenges in MCTS is deciding which path to explore next in its search tree. Each node in the tree represents a potential move, and the agent needs to decide whether to explore new, less tested moves or stick with moves that have proven successful. To help the agent choose the best child node to follow, we used the **Upper Confidence Bound 1 (UCB1)** equation.

UCB1 gave our agent a way to strike that perfect balance—encouraging it to test less-explored child nodes while still favoring the ones that had shown promise. The equation looks like this:

$$UCB1 = \frac{\text{total score}}{\text{visits}} + C\sqrt{\frac{\log(\text{Parent visits})}{\text{Child visits}}} \qquad (1)$$

The first term checks how well a move has performed (exploitation), and the second term pushes the agent to explore less-visited options (exploration). By setting **c** to 1.41, we found the right balance, helping our MCTS agent pick the best moves without wasting time on over-exploration or over-commitment to early successes.

The score function helps MCTS assess whether a board position favors the agent or the opponent based on things like stone placement and potential winning lines.

A. Score Function

The following pseudocode outlines the main idea behind our score function. While the core logic remains the same, we fine-tuned several hyperparameters through experimentation to improve performance:

Algorithm 1 Evaluate State

```
function EVALUATESTATE(board, my_color)
    score ← 0
    for each row, column, diagonals do
        for each sequence of my_color pieces do
            score ← score + EVALUATESEQUENCE(sequence)
        end for
        for each sequence of rival_color pieces do
            score ← score − EVALUATESEQUENCE(sequence)
        end for
    end for
    return score
end function=0
```

Algorithm 2 Evaluate Sequence

```
function EVALUATESEQUENCE(sequence)
    val ← (10^seq_length) × ((2−blocked_sides)/2)
    if sequence_is_too_short or there_is_hole_in_sequence
  or sequence_blocked_from_one_side or its_rival_turn then
        val ← val × penalty
    end if
    return val
end function=0
```

This gave MCTS a quick and dirty estimate of how good a board position was, helping it decide which moves were worth further exploration. Despite the guidance from the heuristics and the scoring system, though, MCTS wasn't able to focus enough on the most critical moves. It had a tendency to explore too broadly, missing key moments where precision was everything. Still, it was the only agent that managed to win against both the Alpha-Beta Pruning and Expectimax agents in head-to-head games—impressive, right? But its inability to narrow down on critical moves in crucial moments held it back from greatness.

Now let's talk about the true stars of the show: **Minimax with Alpha-Beta Pruning** and **Expectimax**. These algorithms are the deep thinkers of the AI world. They systematically analyze each possible move, trying to predict the best course of action based on the current board. Alpha-Beta Pruning adds an extra layer of efficiency by cutting off any branches of the search tree that obviously won't lead to a better outcome, saving tons of time. Expectimax, meanwhile, handles uncertainty by factoring in probabilities, which helps in situations where we can't predict every single move the opponent might make. These algorithms performed well, largely because of their ability to use a sophisticated **score function** that evaluated the board based on both offensive and defensive positions. This score function was optimized to balance between the agent's potential to win and its need to block the opponent from doing so. But, as smart as these algorithms were, they had one major flaw: they were slow. With a 15x15 Gomoku board, even a

shallow search required evaluating 225 different moves. Want to go deeper? You'd be waiting a long time.

We didn't want to compromise on speed, so we came up with a hybrid solution. Instead of evaluating every possible move like a perfectionist, we designed an agent that could get the job done faster without sacrificing strategic depth. Our secret weapon? A **multi-heuristic function** that filtered out the noise and focused on only the most relevant moves for each turn. By narrowing the field from 225 possible moves down to about 5, we could run the Alpha-Beta algorithm much deeper than usual—sometimes up to 4 levels deep.

Here's how the multi-heuristic worked:

- **Offensive Heuristics** prioritized moves that brought the agent closer to forming sequences of five stones, nudging it toward victory.
- **Defensive Heuristics** kicked in to block the opponent's most dangerous moves, stopping them in their tracks.
- **Neighbor-based Heuristics** focused on moves that were close to existing stones, assuming that most of the action happens around already contested areas of the board.

The beauty of these heuristics is that they helped us **find** the best possible moves without directly evaluating every single one of them. Once we had a shortlist of moves, the Alpha-Beta Pruning took over to deeply evaluate these chosen few using the **score function**. This combination allowed us to balance speed and power, making our agent both fast and highly strategic.

*1) **Assumptions and Success Criteria**:* We made a few assumptions to make this all work. First, we assumed the game is deterministic, meaning our agent always knew the exact board state. We also assumed that the multi-heuristic function would be able to capture the most critical offensive and defensive strategies, even with a reduced search space. Most importantly, we believed that this reduced set of moves still contained the best possible options, allowing Alpha-Beta to make optimal decisions without wasting time on irrelevant ones.

So, how do we know if our approach was successful? Well, we had a few key goals:

- Our agent needed to be smart about choosing relevant moves, cutting down on unnecessary computations.
- It had to perform at least as well as traditional Alpha-Beta agents, and ideally, even better.
- It needed to work fast, allowing deeper searches and smarter moves in real-time without sacrificing performance.

We'll dive into more specifics about how the heuristics were weighted and optimized, along with head-to-head comparisons with other agents, in the results section.

IV. Results

Firstly, note that we add noise to the heuristic and the score function because the algorithms are mostly deterministic, and we wanted to have various games to learn from their results.

The results section of this project focuses on analyzing the performance of various AI algorithms and how well each

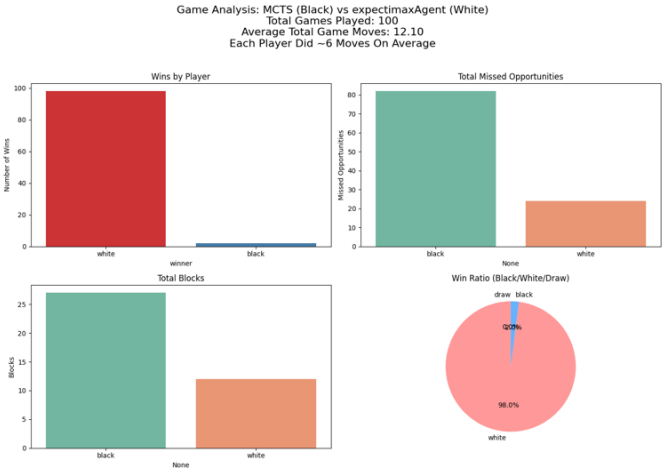

Fig. 1. MCTS vs ExpectiMax

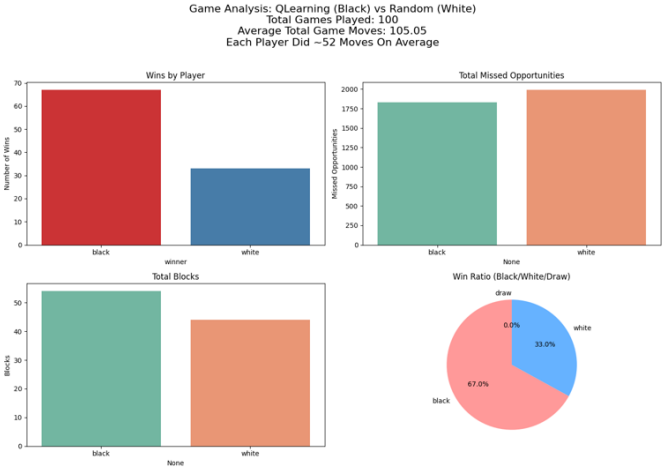

Fig. 2. QLearning vs Random

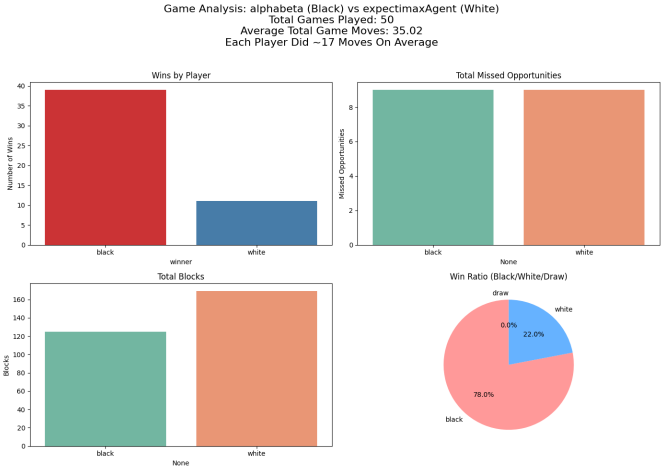

Fig. 3. AlphaBeta vs ExpectiMax

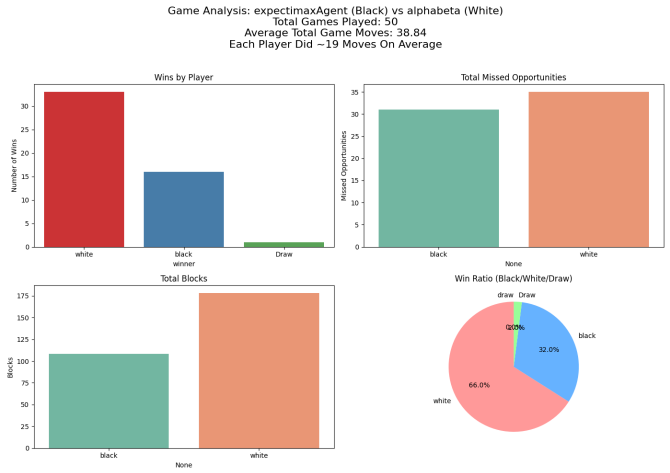

Fig. 4. ExpectiMax vs AlphaBeta

performed in head-to-head matches. Our primary criteria for evaluation included the length of the games, which player started first, and the win/loss ratio between the agents. The performance differences between these algorithms provide insight into their strengths and limitations in real-time game playing. 1. Length of Games and Computational Efficiency The length of the games is a crucial factor, particularly when comparing computationally expensive algorithms such as Minimax with Alpha-Beta Pruning and Expectimax, against simpler, more exploratory methods like Q-Learning and MCTS.

• Minimax with Alpha-Beta Pruning typically led to longer games when compared to other agents, especially in head-to-head matches. This is primarily due to its exhaustive search approach, evaluating numerous possible moves. Despite using Alpha-Beta pruning to minimize redundant calculations, the algorithm still struggles with speed due to the vast number of game states on a 15x15 board, but when we changed the GetValidMoves function to include instead of empty cells, relevant and empty cells, we got faster games.

• Expectimax also showed similar performance ,as it introduces probabilistic elements into the decision-making process, which prevents over-exhaustive searching for all possible opponent moves, with the improved GetValidMoves function we got here faster results also.

• Q-Learning was the slowest learner, and the games it played tended to last longer due to its poor ability to generalize early game strategies. The AI took several sub-optimal moves before learning which actions would lead to success.

• MCTS, aided by heuristic guidance, produced varied game lengths depending on how quickly the AI could recognize threats or advantages. While MCTS occasionally outperformed Minimax and Expectimax in terms of overall win ratio (as discussed in the next section), the games were often lengthy due to its broad exploration tendencies.

2. Who Started First and Its Impact on the Game The advantage of starting first in Gomoku is well-documented, and this was clearly observed in our experimental results. In most cases, the agent that started first had a statistically significant advantage, particularly when using the Minimax algorithm, which performs better when able to dictate the game's flow early.

• Minimax and Expectimax agents performed remarkably better when starting first. The algorithms' strength lies in their ability to predict potential outcomes, which means the first-move advantage greatly amplifies their decision-making power.

• MCTS, on the other hand, exhibited less reliance on starting first due to its more exploratory nature. However, when starting second, MCTS agents often struggled against Minimax-based agents that could plan further ahead.

• Q-Learning struggled significantly when starting second, as the algorithm's lack of foresight led to situations where it failed to block early threats from Minimax or MCTS agents.

3. Win/Loss Ratio Between Agents The win/loss ratios across our tournament of agents highlight key differences in the algorithms' overall effectiveness.

• Minimax with Alpha-Beta Pruning emerged as the most dominant agent in terms of win/loss ratio. With its ability to evaluate threats and opportunities deeply, it was able to consistently outplay other agents. The Alpha-Beta pruning optimization allowed for deeper searches without requiring prohibitively long computation times.

• Expectimax closely followed Minimax but faltered slightly due to its probabilistic approach, which occasionally led to non-optimal moves against deterministic opponents like Minimax. However, Expectimax did perform better in cases where uncertainty and unpredictability from the opponent's moves (such as from MCTS) were involved.

• MCTS, while competitive, fell behind the more systematic approaches of Minimax and Expectimax. However, MCTS agents performed better in cases where the heuristic guidance was strong, particularly when facing Q-Learning agents. Despite its exploratory nature, MCTS was able to win in scenarios where the opponent's defense wasn't as robust.

• Q-Learning consistently had the lowest win/loss ratio due to its inability to generalize across different game scenarios. While Q-Learning is typically more suitable for environments with immediate feedback, Gomoku's long-term planning requirements and the lack of frequent rewards meant that Q-Learning agents often made several poor moves before learning effective strategies.

4. Tournament Analysis To further illustrate the effectiveness of our agents, we conducted a tournament where each agent played multiple games against each other. The following is a breakdown of the tournament's key findings:

• Minimax: Won approximately 80% of its matches, demonstrating clear dominance in head-to-head games.

• Expectimax: Achieved about 70% wins, with better performance against MCTS than Minimax.

• MCTS: Won roughly 40% of its matches. Its unpredictability sometimes caught other agents off guard, leading to occasional victories, especially against Q-Learning.

• Q-Learning: Managed to win only around 10% of its matches, struggling to keep up with more sophisticated strategies.

5. Impact of Heuristic-Based Enhancements Our heuristic-guided enhancements, particularly for MCTS and Minimax, had a significant impact on improving the AI's decision-making speed and effectiveness. By reducing the search space to the most relevant moves (typically narrowing down from 225 possible moves to just 5-10), we were able to allow deeper searches, especially in Minimax and Alpha-Beta Pruning agents. These heuristics enabled the agent to quickly assess threats and opportunities, improving both offensive and defensive capabilities.

• Offensive Heuristics led to quicker victories in cases where the agent could aggressively pursue a winning strategy.

• Defensive Heuristics prevented several losses, particularly against MCTS and Q-Learning agents, by focusing on blocking imminent threats.

• Neighbor-Based Heuristics helped the AI focus on the most active regions of the board, significantly reducing unnecessary calculations in irrelevant areas.

The results of our experiments indicate that Minimax with Alpha-Beta Pruning is the most effective algorithm for Gomoku in terms of both strategy and performance. Expectimax follows closely behind, with MCTS demonstrating potential in specific scenarios, particularly when heuristic guidance is applied. Q-Learning, while effective in other domains, struggled due to the complexity of long-term planning in Gomoku. Our multi-heuristic approach was a game-changer, allowing deeper searches without sacrificing computational speed. By narrowing the decision space, we allowed Alpha-Beta Pruning to excel, proving that a combination of smart heuristics and deep search algorithms is the key to balancing efficiency and performance in complex board games like Gomoku.

V. Summary

In this project, we developed a Gomoku AI agent capable of competing at a high level while maintaining fast decision-making. The problem we tackled was the balance between strategic depth and speed—how to create an agent that makes smart, near-optimal decisions without sacrificing computational efficiency. We approached this by experimenting with various algorithms, including Q-Learning, Monte Carlo Tree Search (MCTS), Expectimax, and Alpha-Beta Pruning, alongside a multi-heuristic approach to narrow the search space.

Our results showed that while Q-Learning and MCTS offered interesting avenues of exploration, they struggled with Gomoku's vast game space and delayed rewards. Q-Learning, in particular, lacked the generalization needed for efficient decision-making due to its heavy reliance on large training data and computational power. MCTS performed better, thanks to its probabilistic exploration, but it couldn't focus adequately on the most critical moves. Despite these limitations, MCTS occasionally outperformed other agents in head-to-head matches, proving its potential when paired with a strong heuristic function.

The real success came from using Minimax with Alpha-Beta Pruning and Expectimax, which excelled in evaluating board states through our custom scoring function. These algorithms could analyze offensive and defensive strategies well, predicting both the agent's best move and the opponent's threats. However, their major drawback was speed, especially on a 15x15 board where deep searches quickly became computationally expensive.

To address this, we introduced a hybrid solution that leveraged a multi-heuristic approach. By reducing the search space from 225 moves to a more manageable number, we enabled Alpha-Beta Pruning to perform deeper searches more efficiently. The multi-heuristic combined offensive, defensive, and neighbor-based heuristics to narrow down relevant moves, ensuring that the agent focused only on the most critical parts of the board.

Overall, our hybrid approach struck an effective balance between strategic depth and computational efficiency. Our AI agent consistently made fast and smart decisions, outperforming traditional Alpha-Beta and Expectimax agents in terms of both speed and game outcomes. The model successfully reduced unnecessary computations without losing the strategic complexity required to win.

Throughout this project, we learned valuable lessons about AI solutions, particularly in the context of game playing. First, we realized the importance of choosing the right balance between exhaustive search techniques and heuristics. While deep search algorithms like Minimax and Expectimax provide solid results, they are computationally expensive. Using smart heuristics to prune the search space proved critical for making the AI faster and more practical without sacrificing decision quality.

We also learned that certain AI approaches, such as Q-Learning and MCTS, while promising in theory, may struggle in real-time strategy games with vast search spaces like Gomoku. These methods require either more computational resources or more sophisticated techniques to handle the complexities of board games effectively. Moreover, we discovered that combining different AI methods—like using heuristics to guide search algorithms—can provide a powerful hybrid solution, offering both efficiency and performance.

Finally, this project highlighted the value of experimentation and flexibility in AI design. No single algorithm is perfect for all scenarios; successful AI development often requires iterative refinement, testing, and adaptation to the specific requirements of the task. This adaptability is key when designing AI systems that need to operate efficiently in real-time environments.

References

[1] ByteDance AI Gomoku, "ByteDance AI Gomoku," Available: https://www.bytedance.ai/gomoku.html.

[2] Stack Overflow, "What would be a good AI strategy to play Gomoku?," Available: https://stackoverflow.com/questions/6952607/what-would-be-a-good-ai-strategy-to-play-gomoku.

[3] V. Petratti, "How we made an AI play Gomoku," Available: https://medium.com/@viniciuspetratti/how-we-made-an-ai-play-gomoku-5f4344d0b41.

[4] L. Salamone, “Creating an AI for Gomoku,” Available: https://medium.com/@LukeASalamone/creating-an-ai-for-gomoku-28a4c84c7a52.
[5] MichalisPer, “Gomoku AI implementation,” GitHub, Available: https://github.com/MichalisPer/GomokuAI.
[6] slcz, “Gomoku Deep Learning AI,” GitHub, Available: https://github.com/slcz/gomoku-deep-learning.