Final Project - Omok

Foundations Artificial Intelligence Spring 2022

Ukhyeon Shin (shin.uk@northeastern.edu)
Vitaliy Shkolnik (Shkolnik.v@northeastern.edu)

Abstract

The project aims to find a winning solution for a game called Gomoku ("Omok") by using Alpha/Beta (A/B) and Approximate Reinforcement Learning (RL) algorithms. The two separate implementations are then compared against each other in terms of speed, efficiency, and win-rate. The A/B algorithm was ideal to investigate because there are only two players, and each move can be evaluated in simple min or max logic. Meaning that the AI can optimize their strategy, while minimizing their opponents. The second implementation involved approximate RL. This method relies on an AI agent learning by experience/exploration and creating an optimal policy to follow against the other player. The results revealed that with small board sizes (7X7) A/B agent performed better than the RL agent. However, as the board size increased, the A/B agent struggled with the search space and was outperformed by the RL agent in terms of time efficiency. It should be noted that A/B was implemented over MiniMax to save on space and time by pruning. This would reduce the tree search significantly by stopping the exploration of dead-end branches.

Introduction

Omok is similar to Tic-Tac-Toe, but has a much larger search space, no optimal path to winning, and involves significantly more strategy. A typical board size is 15X15, with players alternating turns by placing a "stone" on an available spot. The game is over when there are no more available moves, or one player is able to get 5 stones in a row.

Omok exists in a complexity class known as PSpace-complete (Reisch, 1980). This means that this problem is solvable in polynomial time and space. Our motivation for solving Omok is that the problem appears very simple to find the solution when the search space is small, but as the grid size increases the problem quickly becomes unmanageable. This relationship would allow for good experimental design and clearly illustrate the principles of search.

At first glance this game appears like a perfect candidate for A/B. There are two players alternating turns and each turn can be thought of maximizing or minimizing the value of a move. However, the typical board size the game is played on is 15X15 or sometimes 19X19 and the computation space becomes unattainable.

Another approach is using approximation RL. This methodology is superior in some ways, for example, RL is able to understand similar states even though they are technically different. Additionally, once the RL agent has been trained, its moves are faster than the Alpha/Beta agent in large gameboard sizes. With RL you could also improve its performance by increasing its training time and adjusting its reward and feature systems to better align with the problem at hand

Background

- **Environment: Omok and AI background**

Omok, also known as Gomoku, has a history of studies trying to implement AI to this game. Previous attempts involved using genetic algorithms and tree search algorithms (Zheng, 2016). However, after Alpha-Go's success with Go, some studies have looked into deep learning. For example, one paper proposed an improvement to the training data of a convolutional neural networks (CNN) by providing an alternative to the optimal solution. This would resolve rare cases where the optimal solution suggests placing a stone in an available spot and minimize the forgetting problem. (Gu et Sung, 2021).

- **Alpha/Beta Algorithm:**

This algorithm is referred to as "maximin", where you maximize the minimum gain. There also exists other versions of MiniMax, like including alternating moves (Russell et Peter, 2003). However, Omok has simple rules, where a player can only win, lose, or draw with predictable win conditions.

Therefore, we were able to implement the simple version. It's important to note that A/B functions by searching recursively into trees. Meaning that as the board size increases, the search space increases exponentially. This would this limit the depth layer Mini-Max can dive into unless you have limitless storage and time.

- **Approximate Reinforcement Learning (RL):**

RL is an area of machine learning where intelligent agents take actions based on the notion of a cumulative reward. Reinforced Learning was a better option to use here than supervised learning because we wouldn't have to create an original dataset and be limited by the ELO rating the player was in that dataset. In RL, we explore various states while inputting random actions and assign rewards to the (state, action) pair based on an evaluation function. This principle allows you to learn a game by a game by search a smaller game space and learn from these outcomes (Burnetas et Katehakis, 1997).

There are several downsides to RL such as the credit assignment problem, where it's tricky determining what actions we're relevant for the AI in getting to the maximum reward. There are also issues with sparce reward, getting to the goal state is too improbable, and alignment, the agent found an undesired/unpredicted solution.

In our experience we had to adjust the evaluation function several times because the probability of the agent finding a state with 5 same color stones in a row was too insignificant. A newer function was devised that rewarded states with 4 or 3 or fewer stones in a row. With each condition offering different rewards. One assumption to test is that this evaluation function could be better improved if trap moves were incorporated.

Approach

We reused Mini-Max (Alpha-Beta) and Approximate Reinforcement Learning algorithms that were modeled from the Pac-Man programming assignments. We performed multiple experiments with varying conditions to evaluate and compare each implementation. We investigated the impact the board size would have on both agents, the impact depth would have on Mini-Max time and performance, first turn advantage, and the time spent training the RL agent:

- Experiment 1: Test small board-size (7X7) on Mini-Max and RL agents. Investigated impact training time and depth had on outcomes.
- Experiment 2: Test medium board-size (9X9)
- Experiment 3: Test large board-size (11X11)

Minimax (Alpha-beta pruning)

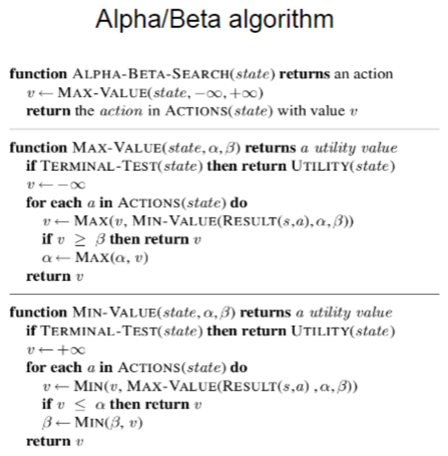
Parameters: This algorithm was run on board sizes 7X7, 9X9, and 11X11. We tested two different depths 2, and 4. The evaluation function was calculated using board states that contained 3X, or 4X stones in a row. If the agent found a state with 3X stones of its own color the sum value was multiplied by 2. 4 stones in a row resulted in another multiplication factor of 2. These values were inversed for the opposite color. With the condition of open spots to make 5 in a row.

The main functions and parameters we used are:

- AlphaBetaAgent(MultiAgentSearchAgent)
- MinimaxAgent(MultiAgentSearchAgent)
- scoreEvaluationFunction(currentGameState):

$$V(s) = \max_{s' \in successors(s)} V(s')$$

Equation 1: Alpha/Beta equation.



Alpha/Beta algorithm

```
function ALPHA-BETA-SEARCH($state$) returns an action
    $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
    return the $action$ in ACTIONS($state$) with value $v$

function MAX-VALUE($state, \alpha, \beta$) returns a utility value
    if TERMINAL-TEST($state$) then return UTILITY($state$)
    $v \leftarrow -\infty$
    for each $a$ in ACTIONS($state$) do
        $v \leftarrow$ MAX($v$, MIN-VALUE(RESULT($s,a$), $\alpha, \beta$))
        if $v \geq \beta$ then return $v$
        $\alpha \leftarrow$ MAX($\alpha, v$)
    return $v$

function MIN-VALUE($state, \alpha, \beta$) returns a utility value
    if TERMINAL-TEST($state$) then return UTILITY($state$)
    $v \leftarrow +\infty$
    for each $a$ in ACTIONS($state$) do
        $v \leftarrow$ MIN($v$, MAX-VALUE(RESULT($s,a$) , $\alpha, \beta$))
        if $v \leq \alpha$ then return $v$
        $\beta \leftarrow$ MIN($\beta, v$)
    return $v$
```

Figure 1: Pseudocode for Alpha/Beta

Approximate Reinforcement Learning

Parameters: The training iteration size was set to 1000 on board sizes 7X7 and 9X9. Rewards were set to 100 for a win, -100 for a loss, and 10 for draw. The features were calculated based on high value game states where 3X or 4X stones were discovered with available empty spots to make 5.

Alpha = .2
Gamma = .5
Iteration = 1000
Exploration = .7 with time decay (i*0.05)
Rewards: win = 100, lose = -100, draw = -10
Features: 3X stones with available spot to make 4 = 1 point
4X stones with available spot to make 5 = 1 point

$$\text{TD-error:}\quad \text{difference} = \left[r + \gamma \max_{a'} Q(s', a')\right] - Q(s, a)$$

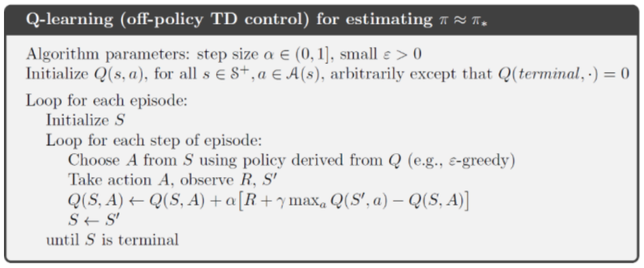
Equation 2: Calculating TD-error

$$w_i \leftarrow w_i + \alpha\,[\text{difference}]\, f_i(s, a)$$

Equation 3: Updating weights based on TD-error

$$V(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

Equation 4: Value determination using approximation



Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

```
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
        $S \leftarrow S'$
    until $S$ is terminal
```

Figure 2: Pseudo code for q-learning (Russell et Peter, 2003)

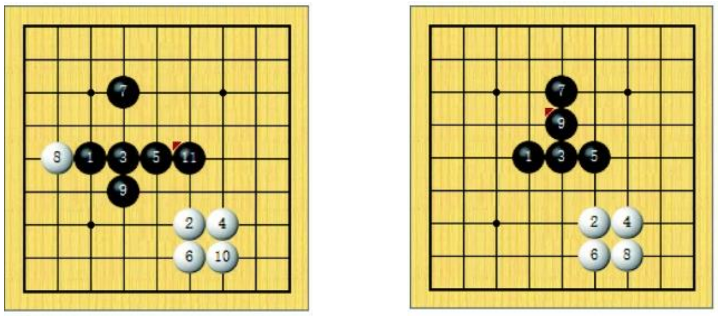


Figure 3: Black Advantage (Trap State) from (Li, He, Wu, Zhao, 2020)

File lists

A brief introduction to the main files we used.**:**

- **main.py:** Select the game mode (1: Human vs RL, 2: Human vs MinMax, 3: RL vs MinMax, 4: Quit). Initialize parameters and learning agents.
- **gameState.py:** Define a gameState class and a grid class and support related functions.
- **multiAgents.py:** Define the Minimax agent and related algorithm.
- **qlearningAgents.py**: Define the Q-learning agent and related algorithm.
- **featureExtractor.py:** Define the features that is used for ApproximateQAgent.

Results

Omok has a significant first mover advantage. We controlled for that by comparing both the wins and loses based on who went first or second.

Who`s 1st	Win	Lose	Draw
Alpha/Beta	860	100	40
RL	730	180	90

Chart 1. Draw game with RL after 3000 times learning

Experiment 1: Alpha-Beta agent vs. RL agent on boardsize (7X7).

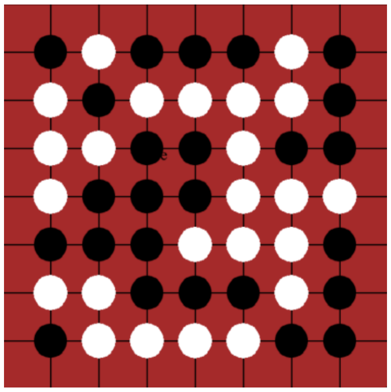


Fig 1. Draw game with Alpha-Beta agent

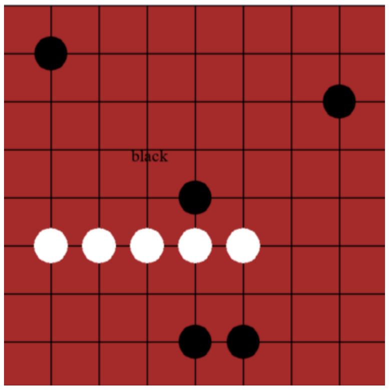


Fig 2. RL after learning 100 iterations

Outcome: Alpha-Beta agent outperformed our RL agent, refer to chart 1. Out of 1000 games where Alpha/Beta went first it won 860 times compared to RL's 730. A 16% difference. However, the efficiency of Mini-Max became a significant problem as the size of the board increased. At 11x11, the game was unplayable.

Experiment 2: Performance in term of time

Minimax (Depth = 2)	
GridSize	first respond (s)
7X7	6.2
9X9	30.9
11X11	151.1

Chart 2. The first respond time of Alpha/Beta in depth 2 search

Minimax (Grid = 7x7)	
Depth	first respond (s)
2	6.2
3	75.5
4	1317.3

Chart 3. The first respond time of Alpha/Beta in depth 3 and 4

RL self-game (RL vs RL)	
GridSize	100 game iterates (s)
7X7	15.688
9X9	68.987
11X11	251.245

Chart 4. The times of 100 game iterates of RL

Outcome: As expected, the searching time for both Alpha-Beta agent and RL agent are significantly increased as grid size becomes larger. Note that the depth parameter is significantly more impactful on time than board size.

Experiment 3: Performance in term of winning rate
We did a comparison experiment by changing the value of the parameter that affects the performance of each agent.

RL vs Minimax (Grid = 7x7)			
Depth	Win	Lose	Draw
2	8	22	0
3	8	12	12

Chart 5. The Minimax win rate in terms of its depth

RL vs Minimax (Grid = 7x7)			
Exploration	Win	Lose	Draw
0.9	6	21	3
0.7	12	12	6
0.5	9	20	1
0.3	3	27	0
0.1	5	15	10

Chart 6. The RL win rate in terms of its exploration

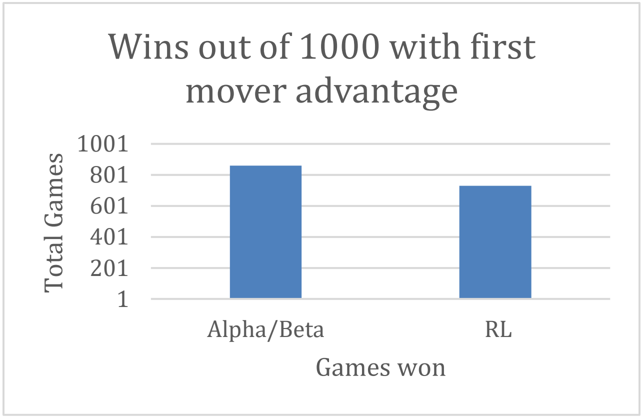
RL vs Minimax (Grid = 7x7)			
alpha	Win	Lose	Draw
0.9	7	21	2
0.7	9	21	0
0.5	10	17	3
0.3	6	16	8
0.1	9	18	3

Chart 7. The RL win rate in terms of its alpha

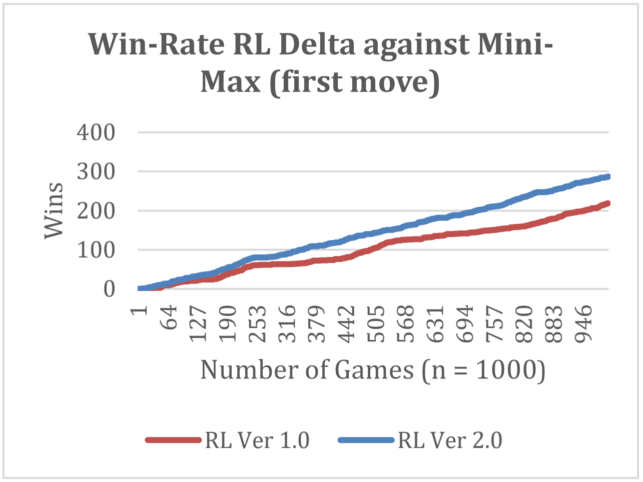
RL vs Minimax (Grid = 7x7)			
Learning time (games)	Win	Lose	Draw
10	4	22	4
100	8	19	3
1000	9	20	1
10000	10	18	2

Chart 8. The RL win rate in terms of its learning time

Outcome: In chart 5, Alpha-Beta agent with depth 3 shows better performance than depth 2. We did not test in depth 4 because it takes too much time as we have seen in the chart 3. For RL, we have changed exploration and alpha parameters, but results were not reliable and failed to obtain meaningful data. This is the tricky determining what actions we're relevant for the AI in getting to the maximum reward. However, in chart 6, we could observe that RL win rate is somewhat related with its learning time. The more learning time guarantees the better winning rate.



Graph 1: 1000 games with RL going first and 1000 games of Mini-Max going first were compared. The results show that the mini-max agent we built performed better than out RL agent.



Graph 2: This represents the increased performance of the RL agent against Mini-Max after improving the features. Suggesting further improves could impact performance. First move advantage outcomes demostrated similar results.

Conclusion

Our initial results demonstrated that our Alpha-Beta agent outperformed our RL agent (graph 1). To control for first mover advantage, we played the agents against each other 1000 times and switched between who went first. These results were measured to be significant.

We think our RL agent performed worse than Alpha/Beta because it wasn't fully optimized. We hypothesize that with further improvements to the features, rewards, training time, and tweaking the parameters (alpha, gamma, exploration) we would be able to close the gap. There are currently two iterations of our RL agent.

The next study would aim to adjust the features and rewards based on information found in related studies. Our own findings suggested that we were already using optimal RL parameters (chart The first idea is to better differentiate between high value gamestates. Instead of giving equal rewards to states with 3X stones and 4X stones (with open end spots), we would increase the rewards of states with 4X more stones. Additionally, states with 4X stones with both sides having available spots should get even more value. Finally, there are several winning moves known to Omok and those should also be incorporated (figure 3). Finally, more data should be collected and better statistical analysis should be performed. For example, STD and variance could have been added.

A couple of clear benefits of the RL agent over Alpha/Beta's are memory and the time between turns. As we used 'Approximate agent', the search spaces in the boards were significantly decreased. Also, with RL the main downside was the time spent on originally learning. This could be resolved by saving the dataset and reloading the learned values.

Once the agent learned, the moves between turns was almost instant. However, with the Alpha/Beta agent as the size of the board increased, the time between moves became too long and this implementation was not scalable. With a 11X11 board, the Alpha-Beta agent took around 2 minutes per move (Table 2). Furthermore, if you increased the depth the response time increased exponentially. With a 15X15 board and a depth of 4, we were only able to figure that the time for one turn would be longer than 24 hours.

The results also indicated that increasing the game board size did negatively impact the performance of the RL agent. But training the RL agent longer compensated. This is likely because there are significantly more gamestates with larger boards and consequently the exploration step finds less meaningful states. Which results in an agent being more random.

The paper *Game Model for Gomoku Based Deep Learning and Monte Carlo Tree Search* stated that their learning model required millions of games and the time span for learning was days. Additionally, it's worth noting that our agents were going to be hardware limited. According to this and some other papers, the more optimal approach is to use a deep learning agent (AlphaZero) and combine it with Monete Carlo Tree Search.

Work Cited

1. Zheng, P.M.; He, L. Design of gomoku ai based on machine game. Comput. Knowl. Technol. 2016, 2016, 33.

2. Gu B, Sung Y. Enhanced Reinforcement Learning Method Combining One-Hot Encoding-Based Vectors for CNN-Based Alternative High-Level Decisions. Applied Sciences.2021;11(3):1291.https://doi.org/10.3390/app11031291

3. Russell, Stuart J.; Norvig, Pe-ter (2003), Artificial Intelligence: A Modern Approach (2nd ed.), Upper Saddle River, New Jersey: Prentice Hall, pp. 163–171, ISBN 0-13-790395-2

4. Stefan Reisch (1980). "Gobang ist PSPACE-vollstandig (Gomoku is PSPACE-complete)". Acta Informatica. 13: 59–66. doi:10.1007/bf00288536.

5. Burnetas, Apostolos N.; Katehakis, Michael N. (1997), "Optimal adaptive policies for Mar-kov Decision Processes", Mathematics of Operations Research, 22: 222–255, doi:10.1287/moor.22.1.222’’

6. Zhang, H., Yu, T. (2020). AlphaZero. In: Dong, H., Ding, Z., Zhang, S. (eds) Deep Reinforcement Learning. Springer, Singapore. https://doi.org/10.1007/978-981-15-4095-0_15

7. Li, X., He, S., Wu, L., Chen, D., Zhao, Y. (2020). A Game Model for Gomoku Based on Deep Learning and Monte Carlo Tree Search. In: Deng, Z. (eds) Proceedings of 2019 Chinese Intelligent Automa-tion Conference. CIAC 2019. Lecture Notes in Elec-trical Engineering, vol 586. Springer, Singapore. https://doi.org/10.1007/978-981-32-9050-1_10