MCTS with Reward Shaping for GoMoKU

Team Members:

Aryan	160050053
Chaithanya Naik	160050102
Sushant Tarun	160050055

Introduction:

Computer board games have been the focus of artificial intelligence research for a long time. Gomoku is a popular two-player strategy board game. It is traditionally played with Go pieces (black and white pieces) on a board with 15x15 intersections. The winner is the player who first obtains an unbroken row of five pieces horizontally, vertically or diagonally. It can be seen that the state-space of the game is too large to be modeled by traditional MDP.

Inspired by the core idea of AlphaGo, we use Monte Carlo Tree Search (MCTS) algorithm for Gomoku and combine it with Reward Shaping and Prior, as the search space is huge, and the exact search is computationally infeasible[1]. Due to the lack of enough compute and simpler nature of the game (as compared to Go), we used simple heuristics which proved to be quite effective. Our main contributions are these heuristics which aid in guiding the selection of an appropriate action during the selection phase of MCTS. These heuristics include the addition of a prior to MCTS and giving additional rewards based on reasonable heuristics. We found out that these strategies worked better than the vanilla MCTS.

Related Work:

Reinforcement Learning (RL) has been a very intricate part of many games for a very long time. Be it board games, or two-dimensional Atari games, RL has proven its merit in all aspects. Especially for board games, MCTS is very popular as a search algorithm to facilitate the selection of real-time good action by taking random simulations in the action space and building a search tree according to the results. It has a long history within the numerical algorithm and significant successes in various AI games, such as Scrabble[3] and Go[4]. The game Gomoku itself is a very popular game in the field of RL. Numerous papers have been published on it which create agents using models which employ various well-known algorithms and different types of learning networks and methods. There is also a worldwide tournament of AI playing Gomoku called Gomocup which has led to the betterment of the AI agents playing Gomoku.

Problem Statement:

About the game

- Gomoku also called Five in a Row, is an abstract strategy board game for two players. It is traditionally played with Go pieces (black and white pieces) on a Go board. It can be played using the 15×15 board or the 19×19 board. Because pieces are typically not moved or removed from the board, Gomoku may also be played as a paper-and-pencil game. The game is known in several countries under different names.
- Players alternate turns to place a piece of their color on an empty intersection. Usually, black and white pieces are used. The winner is the first player to form an unbroken chain of five pieces horizontally, vertically, or diagonally.

The problem

Create an agent to play Gomoku using MCTS and explore various aspects of the algorithm.

Our approach:

The optimal strategy for a larger board size (n) and length of consecutive pieces (p) is an open problem[2]. Our aim in this project is to make an AI agent for this game for the case of p same-colored pieces consecutively on the board of size $n \times n$.

We have implemented three methods for the aforementioned game:

1. We have used the standard MCTS algorithm with a random rollout policy. This serves as a baseline. We are giving a reward of +1 on winning the game, -1 on losing the game and 0 on a draw. As we know that the game will end in at most n^2 steps, so, we use a discount factor (γ) of one. In each selection step, the node picks up the children with the highest UCT value, which is given as:
   $$UCT(child) = -value(child) + C \times \sqrt{\frac{2\log(T+1)}{visits(child)+1}}$$
   T is the total number of visits of all the nodes. C is the exploration coefficient (a hyperparameter). The $'-'$ sign before the *value* is an indicator of the fact that every child, in essence, is playing against its parent. We call this vanilla approach.
2. The problem with the previous method is that the rewards are very sparse (only +1, -1, 0 at the end of the game). So, along with the original rewards which are based on the result of the game, we have given a reward ($r_{potential}$) at each step.
   $r_{potential} = \alpha p$, where p is the potential of the node (described below) and α is a hyperparameter.
   The potential of a node in the search tree is the maximum length of a straight unbroken chain passing through the position where the agent played the last move.

This method was also not giving expected results. Our hypothesis is that the shaped reward was way larger in magnitude than the reward given by win/loss/draw. So, the agent was very greedy towards lengthening its chain rather than breaking it's opponents' chain (even if the opponent wins if its chain is not broken). We tried decreasing α but unfortunately, we weren't able to fix this behavior. But, we would like to point out that this was still better than the vanilla approach. We call this reward shaping approach.

3. Instead of changing the vanilla value, we thought that it might be helpful if we use the potential p (defined in the previous part) as a prior to help guide the search. In this case, the UCT value of each child becomes:
   $UCT(child) = -value(child) + C \times \sqrt{\frac{2 \log(T+1)}{visits(child)+1}} + \frac{z \times p}{visits(child)+1}$ (z is another hyperparameter)
   We are saying to the model to explore places with high potential more often (we are treating the potential as prior). We are decreasing the weight of this prior with the increasing number of visits to ensure that all children get sufficient attention. We call this prior approach.

Following are some other minor heuristics that we have used in every method:

1. If a node's child is definitely going to lose, then we have removed all other children from that node. Also, we have marked that node as winning and set our value as 1.
2. When we have chosen a step, we aren't starting the next MCTS from scratch; instead, we are simply changing the root to the selected child and removing the part of the tree which is not a descendant of the new root. This increased the utilization of the simulations done in previous iterations.
3. We carry out rollouts in a parallel manner. Each time we reach the leaf, we perform a fixed number of rollouts (using a random policy) using a fixed number of workers and consolidate their outcomes into the value of the leaf).

We have analyzed the capability of the AI as developed by the three methods in terms of factors like convergence time, better strategies, depth of the search, exploration co-efficient.

Experiments and Results:

We used the following hyperparameters for all the experiments (unless otherwise specified):

w = 64, r = 96, $g = 1$, t = 5 sec, number of games = 35

w $\rightarrow$ number of workers (for parallelizing rollouts)

r $\rightarrow$ number of rollouts carried at each leaf

t $\rightarrow$ time limit to make one move. (This is different from T in UCT).

g $\rightarrow$ gamma

Other parameter naming conventions:
d → Max depth (nodes at this depth are considered as leaves)
c → exploration coefficient
a → coefficient for reward shaping
z → coefficient for prior
We have provided the config files for all the configuration of all the experiments. Please look at those for a full description of all the hyperparameters. In config, the parameters in lowercase correspond to Player 1 and those in uppercase corresponds to Player 2. Parameters that haven't been passed for Player 2 take the same values as Player 1. See *appendix* for how to run the code.

Since player 1 has an undue advantage in the game we have considered both scenarios in which either agent is player 1.
Between every pair of agents, we conducted 35 matches.
Also, **Wins, Lose and Draw: all of them correspond to Player 1.**

We have analyzed the following aspects of our models:

1. Strategy (described in "Our approach" section above)

   We conducted a round-robin tournament of the three agents (we call them Vanilla, Reward Shaping, and Prior). The following are the results:

Player 1	Player 2	Wins	Lose	Draw	Total matches
Prior	Reward Shaping	29	6	0	35
Reward Shaping	Prior	11	23	1	35
Prior	Vanilla	29	6	0	35
Vanilla	Prior	12	22	1	35
Reward Shaping	Vanilla	21	14	0	35
Vanilla	Reward Shaping	13	21	1	35

2. Time(T) allocated to make one move
   We conducted a round-robin tournament of the three agents with different time to make a move using the Prior approach. Here are the results:

Player 1	Player 2	Wins	Lose	Draw	Total matches
T = 1	T = 4	13	20	2	35
T = 4	T = 1	30	5	0	35
T = 4	T = 8	17	12	6	35
T = 8	T = 4	22	10	3	35
T = 1	T = 8	5	26	4	35
T = 8	T = 1	33	2	0	35

3. Maximum depth(D):
   We have used two approaches for search-depth analysis. The first one uses a fixed time for selections while the second one uses a fixed number of selections. The results displayed here seem to be performing better for lower depths because of the prior set in our method which dominates the value and exploration term if the number of visits is low which is the case with deeper nodes. Thus defying our intuitive result.

   a. Using a fixed time of 5 seconds:

Player 1	Player 2	Wins	Lose	Draw	Total matches
D = 3	D = 5	22	12	1	35
D = 5	D = 8	28	5	2	35
D = 8	D = 10	24	10	1	35
D = 10	D = 5	15	20	0	35

b. Using a fixed number of selections:

Player 1	Player 2	Wins	Lose	Draw	Total matches
D = 3	D = 5	32	3	0	35
D = 5	D = 3	13	22	0	35
D = 5	D = 8	33	2	0	35
D = 8	D = 5	17	18	0	35
D = 8	D = 10	33	2	0	35
D = 10	D = 8	9	26	0	35
D = 10	D = 5	1	34	0	35
D = 5	D = 10	34	1	0	35

4. Exploration coefficient (C):
We varied C from C = 0.02 to 0.35. Theoretically, C should have been 1[5]. But, we found out that lower values of C were performing better. This might be because of sparse rewards.

Player 1	Player 2	Wins	Lose	Draw	Total matches
C = 0.05	C = 0.02	21	10	4	35
C = 0.02	C = 0.05	21	11	3	35
C = 0.05	C = 0.15	24	7	4	35
C = 0.15	C = 0.05	16	14	5	35
C = 0.05	C = 0.25	27	7	1	35
C = 0.25	C = 0.05	9	23	3	35
C = 0.05	C = 0.35	27	7	1	35

C = 0.35	C = 0.05	2	29	4	35

Discussion and Conclusion:

Many approaches have been used to create agents for this game. We have created some MCTS based agents and explored various aspects of the algorithm. Our work suggests that the way used to select the next child is very crucial for the performance of the agent. We have also explored augmenting rewards for doing this. The search depth plays a key factor in determining the time and space complexities as they both tend to grow exponentially with increasing depth. This hugely limits the real-time performance of the agent. We have used a random policy for roll-out which can be replaced by a better-trained policy that uses regression or some other network resulting in more accurate representation for the leaf node values. Our implementation always starts the game from scratch, that is, each game played is independent of any other. To improve what we can do is self play the agents among themselves to tune them better. Doing this has the ability to result in a more sophisticated but better agent. Like AlphaZero, we can train a deep learning-based model to predict the value of each state, so that we can skip rollouts totally.

Acknowledgment:

We extend our thanks to our instructor, Prof. Shivaram K. for teaching us about various aspects of reinforcement learning and helping us decide the project in the first place.
We would also like to acknowledge the help of our friends and wingies for their help in testing the AI to make it better.

References:

[1] Z. Tang, D. Zhao, K. Shao, et al, “ADP with MCTS algorithm for Gomoku,” IEEE Symposium Series on Computational Intelligence, pp. 1-7, 2016.

[2] On p. 60 of József Beck's book Combinatorial Games: Tic-Tac-Toe Theory, he states the following problem concerning "unrestricted 5-in-a-row" (Gomoku on an infinite board):
Open Problem 4.1. *Is it true that unrestricted 5-in-a-row is a first player win?*

[3] A. Ramírez, F. G. Acuña, A. G. Romero, R. Alquézar, E. Hernández, A. R. Aguilar, and I. G. Olmedo, "A Scrabble Heuristic Based on Probability That Performs at Championship Level," MICAI 2009: Advances in Artificial Intelligence, vol.5845 Lecture Notes in Computer Science, pp. 112-123, 2009

[4] A. F. Smith and G. O. Roberts, "Bayesian computation via the Gibbs sampler and related Markov chain Monte Carlo methods," Journal of the Royal Statistical Society. Series B (Methodological), pp. 3-23, 1993

[5] H.S. Chang, M. Fu, J. Hu, and S.I. Marcus. An adaptive sampling algorithm for solving Markov decision processes. Operations Research, 53(1):126–139, 2005

Appendix:

Pitfalls and Bugs:

Earlier, we had coded in python which owing to its slow speed gave us very bad results and the number of iterations occurring at a node was also very less. So, we switched to C++, which increased the number of iterations manifold(almost 50-100 times). This gave us more freeway to experiment better and tune-up preferable outcomes.

One bug(not a concerning one) present in our code is that when the agent knows that, on playing any move there is a guarantee that the opponent has a move with which the opponent can win, the agent plays a random move.

Running the Code:

Before running, first compile the code by running: `make`

Run the following to run experiments mentioned in this report:

```
python3 play_arena.py <file_name>.config number_of_games
<file_name>.log <file_name>.parsed
```

Run `main_wrapper.py` without any arguments to play against AI. (Use full screen for a better view). Enter your move in the following format:

row_number<space>column number. e.g: 0 5

You can modify the following parameters as per your liking:

```
"-b", "--board_size", type=int, default=11
 "-l", "--line_size", type=int, default=5
 "-m", "--mode", type=str, default="01000001"
 "-v", "--verbose", type=int, default=1
 "-r", "--num_rollouts_1", type=int, default=100
 "-d", "--max_depth", type=int, default=5
 "-t", "--timeout", type=int, default=1
 "-w", "--num_workers", type=int, default=4
 "-c", "--exploration_coeff", type=float, default=1.0
 "-g", "--gamma", type=float, default=1.0
 "-z", "--beta", type=float, default=0.1
 "-a", "--alpha", type=float, default=0.1
```

-m is 8 bit string where rightmost 4 bits is for first player and leftmost 4 bits is for second player

-m abcdefgh abcd = 0001 => first player is human; efgh = 0100 => second player is agent

With -v = 1 the play state of the board is displayed after each move.
With -v = 2 will show the debug output with details of visits and UCT matrices of each position