Project 2: Gomoku

2024-06-03

In the last project, you implemented AI algorithms for a simple game Tic Tac Toe with a limited search space. Let's level up by implementing AI algorithms for a harder game: **Gomoku**.

Gomoku, also known as Five In A Row or Caro, is an extension of Tic Tac Toe because the game ends when a player successfully places **five of their stones** in a row horizontally, vertically, or diagonally. The game is typically played on a 15x15 or 19x19 grid board. Therefore, the game is more challenging than Tic Tac Toe because of the enormous state space and complex winning strategies. In this project, you will use your logic and creativity to implement your own AI agents that can be deployed to compete with your friends!

The project has two parts. The Programming part has 75 points with an additional 15 bonus points available. The Written Report part has 25 points. You will submit 5 (or 6 if you complete the extra question) `.py` files and 1 written report in PDF format.

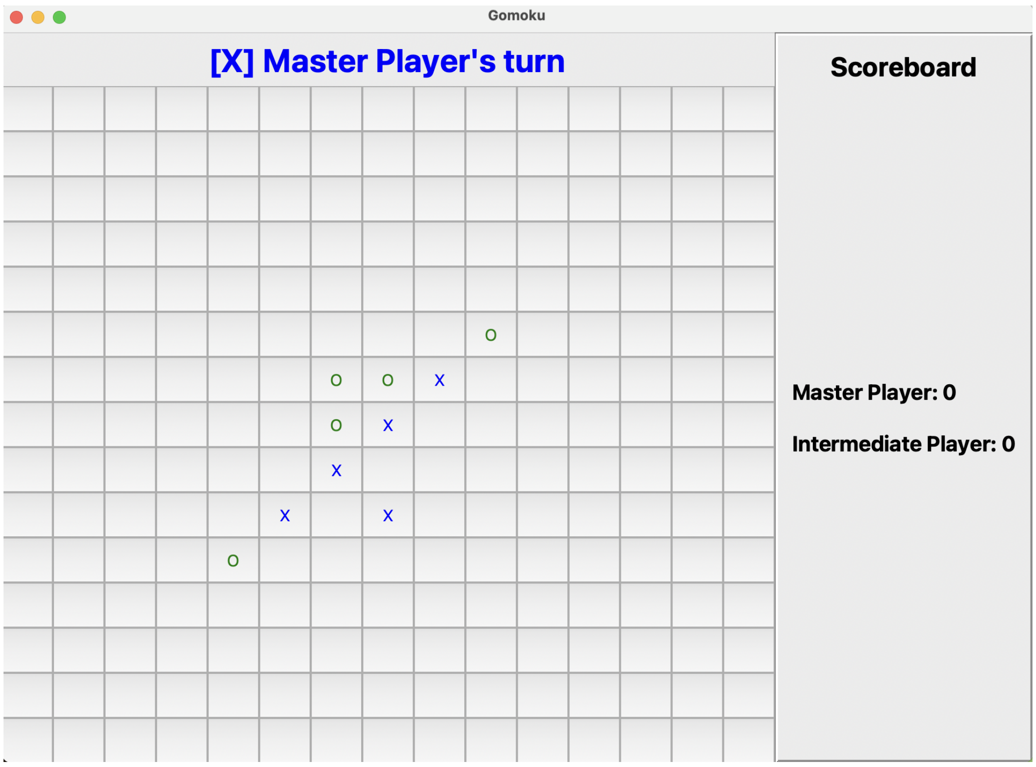

Figure 1: Gomoku game

Environment

Class `Gomoku` in `game.py` controls the game logic. It is built on the same abstract class `Game` as Tic Tac Toe. Similarly, to create AI agents, you will inherit from the abstract class `Player` to implement the `get_move` method that returns the **move** for the given game state. The move is the coordinates (x, y) of the selected cell on the board. You can ignore the file `gameplay.py` which controls the gameplay and UI elements for the game.

Evaluation

There are two ways that you can evaluate your agents:

- **Bots.** In `project/gomoku/bots`, we provided you with 4 implemented AI agents in increasing complexity: Beginner, Intermediate, Advanced, and Master Player. Notably, Advanced and Master players use threat search space algorithm Alus et al. [1996] that focuses on moves that create or respond to potential threats, rather than exploring all possible moves. Please see `project/__init__.py` to see the names of these bots in the command line. Example command to evaluate your agent against Advanced Player:

```
python main.py -p1 [YOUR_AGENT] -p2 advanced -n [NUM_GAMES] -m ui
```

- **Test cases.** We have constructed hand-crafted test cases to test your agents. The agent will be tested on its attack performance, defense performance, and intelligence (ability to find key moves that lead to a win). To evaluate your agent with the test cases:

```
python evaluation.py -p [YOUR_AGENT]
```

  The detailed evaluation will be saved in `evaluation` folder, inside `[agent_name].txt`.

Implementation Note

- Your agent should be fast in making a move. The default time limit for a move is set to 10 seconds. If an agent fails to make a move within the time limit, a random move will be taken.
- If you want to create a new agent, add the agent to `project/gomoku/__init__.py` and then initialize the agent in `project/__init__.py`.
- If your agent has memory between moves in a game (e.g., a winning sequence or move history), you may need to implement a `restart()` function for your agent to restart after each game to avoid bugs during game evaluation. Refer to `master.py` as an example. However, note that for a Q agent, the trained weight or Q table should not be reset.
- For Q-Learning agents, the Q-table (for Tabular Q-Learning) or the training weights (for Approximate Q-Learning) are automatically saved after training inside the `q_table` or `q_weights` folder respectively. The weight file has the format `NxN_NUM_SIMULATIONS.pkl` with N as the size of the board used for training. To load the training weight (both for games between agents and evaluations with test cases), please use the `--load` or `-l` command with the weight file.
- The solutions to the previous project, Tic Tac Toe, are available for your reference.

Programming

IMPORTANT:

- Evaluation criteria are specific to each question so you should read carefully.
- The scoring for each question is cumulative for each criterion listed.
- The default setting to compare your agent with the bots is **10 games with a time limit of 10 seconds per move**.

Q1: Reflex Player (15 points)

A reflex agent is an AI that makes decisions based solely on the current percept, using predefined condition-action rules without memory or learning. In `reflex.py`, you will implement a reflex agent by constructing a strategy to select a move that maximizes your winning chance.

Evaluation.

- **5 points**: The agent gets at least 30/75 correct test cases.
- **5 points**: The agent beats the Beginner player (i.e., has more wins than losses).
- **5 points**: The agent beats the Intermediate player.

Q2: Minimax (with Alpha-Beta Pruning) Player (20 points)

In `alphabeta.py`, you will implement your minimax player with alpha-beta pruning. You need to implement a heuristic evaluation for non-terminal leaves since an exhaustive search is not feasible within the time limit. The strength of the agent will depend on the quality of your evaluation function. You can use the provided Tic Tac Toe implementation as a guide to help you implement Alpha-Beta pruning on Gomoku.

Optimizations. You should try to optimize the code to incorporate a greater depth of minimax search. Some optimization approaches could be:

- **Alpha-Beta Pruning.** This technique helps to reduce the number of nodes that are evaluated by the minimax algorithm, significantly speeding up the search process.
- **Move Ordering.** Prioritize the evaluation of the most promising moves first to increase the chances of alpha-beta pruning.
- **Transposition Tables.** You could try to augment the table by rotation and flipping. Use these to store the evaluation of previously visited board positions to avoid redundant calculations. You could possibly **pre-train** the agent before playing games to store these values (similar to Q agent).
- **Iterative Deepening.** Perform a series of depth-limited searches, increasing the depth limit with each iteration, to ensure that the most important moves are evaluated first.

Evaluation.

- **5 points**: The agent beats the Beginner player.
- **5 points**: The agent beats the Intermediate player and gets at least 65/70 correct test cases.
- **5 points**: The agent beats the Advanced player.
- **5 points**: The agent beats the Master player.

Q3: Approximate Q-Learning Player (20 points)

You will implement an Approximate Q-Learning agent for the game Gomoku in `qlearning_v2.py`. Due to the enormous state space of Gomoku, storing Q values in a table, as done in the Tic Tac Toe game, is impractical. Instead, this version of Q-Learning uses a heuristic evaluation function to approximate the value of a (state, action) pair. This involves extracting key features from the current board and optimizing the feature weights through Temporal Difference (TD) Learning. You will design your feature extractor that inherits from `FeatureExtractor` class. The performance of the Q-Learning agent depends on the quality of the feature extractor.

Note.

- The current feature extractor is set to `IdentityExtractor`, which assigns 1.0 to every (state, action) pair. Approximate Q-Learning with `IdentityExtractor` will function exactly the same as Tabular Q-Learning.
- To improve the learning performance of the Q-Learning agent, you can train it with a strong `transfer_player` such as Master player or Intermediate player.
- The implementation for Tabular Q-Learning is given in `qlearning_v1.py`.
- You can use the option `-no_train` together with `-load` to only load the trained weights (or Q-table) without performing any training.

Evaluation.

- **5 points**: The agent beats the Beginner player.
- **5 points**: The agent beats the Intermediate player and gets at least 65/70 correct test cases.
- **5 points**: The agent beats the Advanced player.
- **5 points**: The agent beats the Master player.

Q4: Monte Carlo Tree Search (MCTS) Player (5 points)

In `mcts_v1.py`, you will implement the standard MCTS player for Gomoku with the following policies:

- Tree Policy (or selection strategy within the tree): Based on UCB1 formula:

$$UCB_1(n) = \frac{Q(n)}{N(n)} + c \cdot \sqrt{\frac{\log N(Parent(n))}{N(n)}} \tag{1}$$

  where $Q(.)$ is the total quality (value) of a node; $N(.)$ is the number of times a node is visited; $Parent(.)$ is the predecessor of the node; c is an exploration constant.

- Rollout Policy (or strategy for running simulation): Random sampling.

The implementation for standard MCTS should be the same as your implementation in the Tic Tac Toe game of Project 1.

Note. You should observe that MCTS is slow at making a move so it may exceed the time limit of 10 seconds on a 15x15 board. Therefore, to compare this naive MCTS agent against other agents, you should evaluate it on a smaller board size (e.g., 8x8 board).

Evaluations. You should get a full score if your implementation is correct. That is, your agent should get at least 40/70 correct test cases with 1000 simulations, and 63/70 with 10000 simulations. It should beat Beginner Player with at least 2000 simulations on an 8x8 board.

Optimizations. You can try multi-processing to run simulations in parallel.

Q5: Better MCTS Player (15 points)

You should observe that the standard MCTS player is not efficient and effective since Gomoku has enormous state space and action space. The standard rollout policy is random sampling, so MCTS might need to simulate many moves before receiving any feedback, making the search process inefficient. Let's try optimizing MCTS in this question. In `mcts_v2.py`, you will implement the improved version of MCTS player for Gomoku. Some possible improvements could be:

- **Heuristic-Based Rollouts**: Incorporate domain-specific heuristics into the simulation phase to improve the quality of the rollouts. For example, prioritize moves that create or block potential winning lines, such as open threes and fours.
- **Hybrid MCTS and Minimax**: Combine MCTS with a depth-limited minimax search to improve the evaluation of leaf nodes. When the MCTS reaches a certain depth, switch to a minimax evaluation for more accurate assessments.
- **Better Rollout policy**: Incorporate domain-specific heuristics specific to Gomoku during the simulation phase (rollouts). For instance, prioritize moves that block the opponent's potential five-in-a-row or complete your own.
- **Better Tree Expansion**: Implement progressive widening to control the expansion of the search tree by initially focusing on the most promising moves and gradually considering a wider set of actions as more simulations are performed. Introduce a progressive bias to favor more promising moves based on domain knowledge or heuristic evaluations.

- **Threat-Based Search**: Focus the search on moves that create or respond to potential threats. Implement a threat-search space that prioritizes moves creating immediate winning threats or blocking opponent threats. This could reduce the search space and improve the efficiency of the MCTS algorithm.
- **Transposition Tables**: Use transposition tables to store previously visited states and their evaluations. This allows you to reuse information from identical or similar positions, preventing redundant calculations and speeding up the search process.
- **Improved Node Selection and Backpropagation**: Implement move ordering strategies to prioritize moves more likely to lead to a win. Use enhanced backpropagation techniques to update node values more accurately, such as weighted averages or incorporating uncertainty measures to better reflect the confidence in the evaluations.

You can refer to this link (Section 4) by Gelly et al. [2012] and this link (Methods, page 7) by Silver et al. [2016] for extensions of MCTS.

Note. You should tune `NUM_SIMULATIONS` adaptively to your implementation to ensure the agent stays within the time limit for each move.

Evaluation.

- **5 points**: The agent beats the Intermediate player on a 15x15 board and gets at least 65/70 correct test cases.
- **5 points**: The agent beats the Advanced player on a 15x15 board.
- **5 points**: The agent beats the Master player on a 15x15 board.

Extra question: Alphago MCTS Player (15 points)

This is a bonus question and you should only attempt it if you have done other questions. In this question, you will try to implement the famous *AlphaGo Zero* Silver et al. [2017] version of MCTS for the game Gomoku. You will implement it in `mcts_v3.py`. The AlphaGo Zero algorithm enhances MCTS with a deep neural network that evaluates board positions and suggests potential moves by outputting a probability distribution over possible moves (policy) and a value estimate of the current board position. During the MCTS process, the neural network guides the selection and expansion of nodes, while the value estimates are backpropagated to update the tree. The neural network is iteratively trained using self-play games, where the MCTS guided by the current network generates data, and the network is updated to minimize prediction errors and improve move predictions. This combination of MCTS and deep learning allows for a more efficient and effective exploration of the game state space in Gomoku.

Note. To speed up the training process, you can try to train the agent on a smaller board size (10x10), initialize a small neural network, and train on Google Colab or Kaggle's free GPUs. The training should typically take a few hours.

References. You could take a look at the following repositories for reference: github1 and github2.

Written Report (25 points)

You will prepare a written report (maximum 3 pages) for the project. You are recommended to write the report in Latex. The report will contain four sections:

1. Introduction: Briefly describe the game, the methods, and results.
2. Methodology: Describe the details of your approaches for each question. For example, you should describe your reflex agent in Q1, the heuristic evaluation function in Q2, the feature extraction in Q3, and the improvements you add to MCTS in Q5. You should also mention optimization techniques and other novel approaches if you have any.
3. Evaluation: This section contains evaluations of your implemented agents. Describe the scoring of your agents in the test cases. Compare your agent with each of the bots we've provided by recording the results of 10 games for each bot. You could also compare your agents with each other (e.g., Q agent trained with Minimax agent versus Minimax agent).
4. Conclusion: Summarize the key findings and insights gained from the project. Discuss any challenges encountered during the implementation and how they were addressed. Finally, provide suggestions for future work and potential areas for improvement.

IMPORTANT:

- We will stop reading your report after 3 pages
- Your report should be **clear** and **concise** as we will give you:
  - **10 points** for your report content
  - **10 points** for your report presentation
  - **5 points** if your code matches what you write in your report.

References

L. V. Alus, 1 H. J. Herik, and M. P. H. Huntjens. Go-moku solved by new search techniques. *Computational Intelligence*, 12, 1996. URL `https://aaai.org/papers/0001-fs93-02-001-go-moku-solved-by-new-search-techniques/`.

Sylvain Gelly, Marc Schoenauer, Michèle Sebag, Olivier Teytaud, Levente Kocsis, David Silver, and Csaba Szepesvari. The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions. *Communications of the ACM*, 55(3):106–113, 2012. URL `https://inria.hal.science/hal-00695370`.

David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016. URL `http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html`.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, Oct 2017. ISSN 1476-4687. doi: 10.1038/nature24270. URL `https://doi.org/10.1038/nature24270`.