Project 1: Tic Tac Toe

2024-05-10

Tic Tac Toe is a classic two-player game that is simple to learn but offers a rich avenue for strategic thinking. Typically played on a 3x3 grid, the game involves two players who take turns marking the spaces in a grid with their respective symbols: one player uses 'X' and the other uses 'O'. The objective is to be the first player to place three of their symbols in a horizontal, vertical, or diagonal row. The game ends either when one player achieves this goal, which is known as a "win", or when all nine squares are filled without anyone achieving three in a row, resulting in a "draw". In this project, you will have the chance to implement various AI algorithms for this game.

The maximum score for this project will be 100/100 points. The coding part takes 80/100 points and the written part takes 20/100 points. You will submit 4 `.py` files for each player and the written part in PDF.

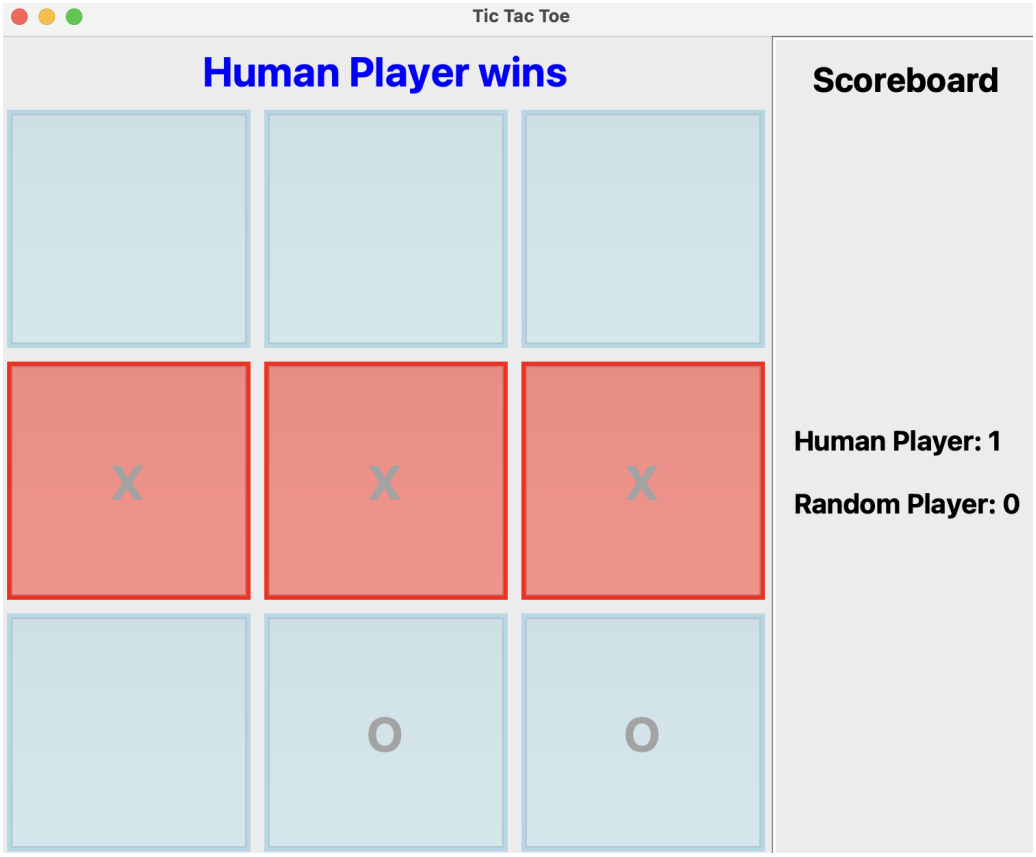

Figure 1: Tic Tac Toe game

Environment

To get started, you need to understand the game environment. In the project, `game.py` includes the game backend, where players can interact with the game. `player.py` defines an abstract class `Player` that defines whether the player plays 'X' or 'O'. To create AI agents, you will inherit from `Player` and implement the `get_move` method and return the **move** for 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.

Important methods in `game.py` that you may need for your implementation:

- `set_move`: Player set the move on the board.
- `valid_move`: Check if the move is valid or not.
- `wins`: Check if the player wins.
- `game_over`: Check if the game ends.
- `copy`: Create a new copy of the current game. Since the board is modified in-place, this method should be used to avoid modifying current board state.

Questions

Q1: Minimax Player (30 points)

Minimax is classical search algorithm that can be used for adversarial games like Tic Tac Toe. The core idea of the minimax algorithm is to recursively search through the game tree to determine the best move that minimize the potential loss for a worst-case scenario while maximizing potential gain. You will implement your Minimax algorithm with the evaluation functions in `minimax.py`.

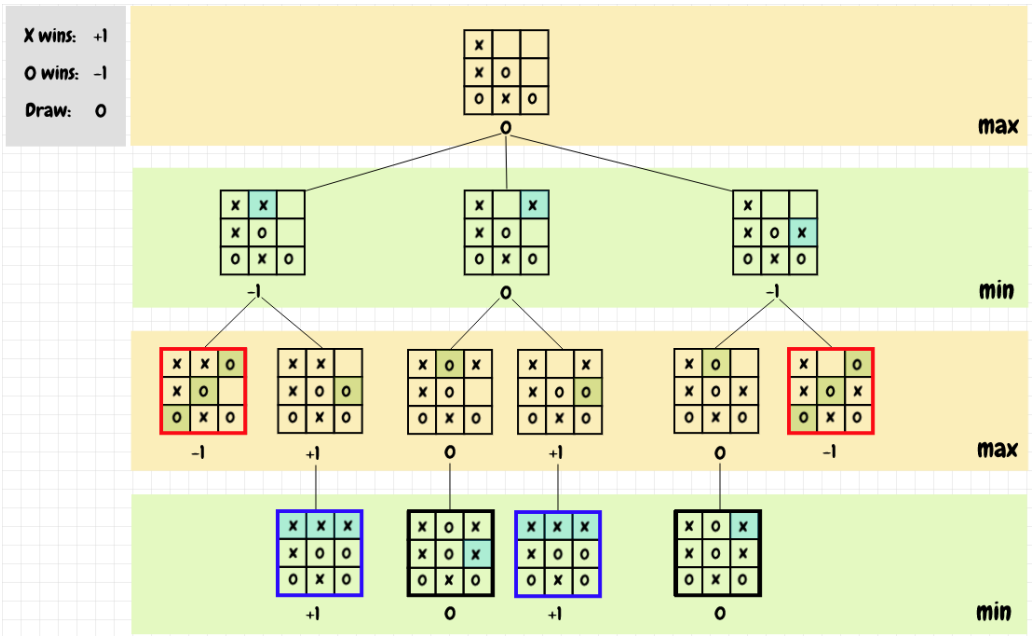

Figure 2: Minimax for Tic Tac Toe

Written part: Play 100 games between Minimax and Random Player, 100 games between Minimax and Minimax Player itself, 10 games between Minimax and Human Player (yourself). Record the result. Does the Minimax Player play optimally?

Q2: AlphaBeta Player (10 points)

Implement Minimax Player with the addition of alpha-beta pruning in `alphabeta.py`.

Written part: Play 20 games between AlphaBeta and Minimax Player. Does AlphaBeta Player give the same performance as Minimax Player? Record the runtime per move of Alpha-Beta and Minimax. Does alpha-beta pruning result in an improvement in the runtime?

Q3: Monte Carlo Tree Search (20 points)

Monte Carlo Tree Search (MCTS) is a policy search algorithm that evaluates game states by sampling random simulations, balancing exploration and exploitation to find optimal moves in complex decision spaces. The procedures of MCTS is as follows: [1]

- **Selection:** Starting at the root of the search tree, we choose a move (guided by the selection policy), leading to a successor node, and repeat that process, moving down the tree to a leaf node.
- **Expand:** Expanding the search tree by generating a new child of the selected Node.
- **Simulate:** We perform a playout from the newly generated child node, choosing moves for both players according to the playout policy. Note that we can also perform simulation from the leaf node.
- **Back-propagation:** We now use the result of the simulation to update all the search tree nodes going up to the root

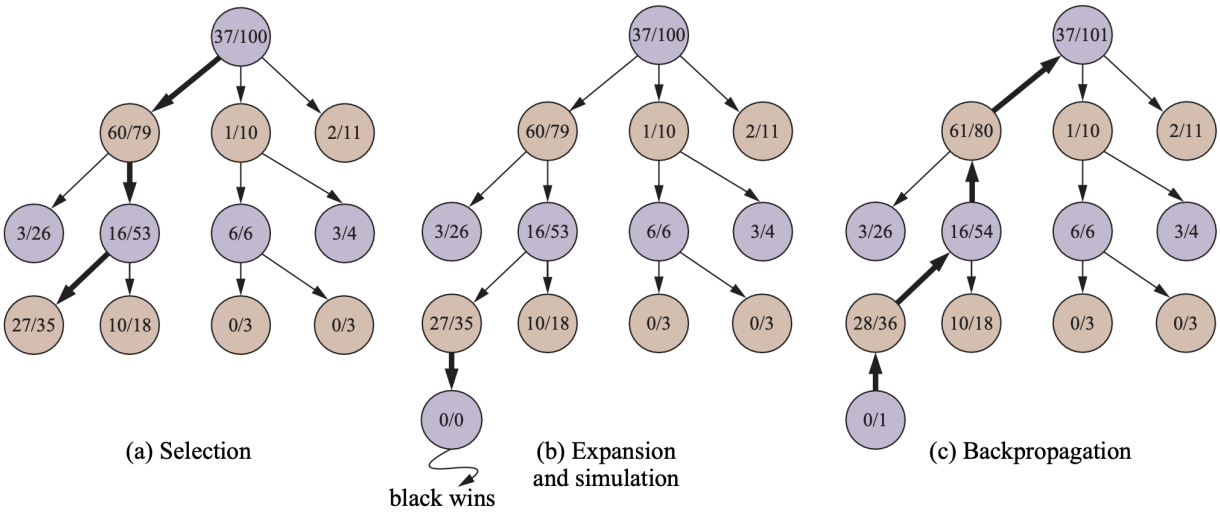

Figure 3: One iteration of MCTS.

Selection policy: A popular selection strategy to balance between exploration and exploitation is based on an upper confidence bound formula:

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

[1] Please refer to Chapter 6.4 in *Artificial Intelligence: A Modern Approach* for more details

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 a constant to control the amount of exploration.

The Algorithm for MCTS is given as follows:

function MONTE-CARLO-TREE-SEARCH(*state*) **returns** *an action*
  tree ← NODE(*state*)
  while IS-TIME-REMAINING() **do**
    leaf ← SELECT(*tree*)
    child ← EXPAND(*leaf*)
    result ← SIMULATE(*child*)
    BACK-PROPAGATE(*result*, *child*)
  return the move in ACTIONS(*state*) whose node has highest number of playouts

Figure 6.11 The Monte Carlo tree search algorithm. A game tree, *tree*, is initialized, and then we repeat a cycle of SELECT / EXPAND / SIMULATE / BACK-PROPAGATE until we run out of time, and return the move that led to the node with the highest number of playouts.

Figure 4: MCTS Algorithm (Chapter 6.4, *Artificial Intelligence: A Modern Approach*)

In this question, you will implement 4 procedures of pure MCTS in `mtcs.py`. If your implementation is correct, the agent should perform relatively similar to Minimax.

Written part: Play 100 games betwwen MCTS and Random Player, 100 games between MCTS and Minimax/AlphaBeta Player. Record the result and the runtime of MCTS and AlphaBeta players.

- Try to increase the number of simulations from 100 to 10000. How does the number of simulations impact the performance?
- The default value for the exploration constant c is $\sqrt{2}$. Try to play around with this value. How does c impact the performance?
- The default policy for rollout/simulation is to play randomly. This is a very naive approach to estimate the value of the state, which may not be efficient for a more complex game like Gomoku. Could you think of a better rollout policy?

Q4: Tabular Q-Learning (20 points)

In this question, you will implement reinforcement learning using tabular Q-learning for Tic Tac Toe in `q_learning.py`. The basic idea of tabular Q-learning is simple: We create a table consisting of all possible states on one axis and all possible actions on another axis. Each cell in this table has a Q-value. The Q-value tells us whether it is a good idea or not to take the corresponding action from the current state.

To update the Q-table, the agent learns by trial and error from interactions with the environment. Through such interactions, the agent earns rewards, and it updates the Q values for

(state, action) pair with following equation:

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma \max_a Q(s',a) - Q(s,a)) \tag{2}$$

where Q is the Q-table, s, a are current state and action taken from that state, r is the reward from taking action a, γ is the discount factor, α is the learning rate, $\max_a Q(s',a)$ is the maximum Q value of the next state.

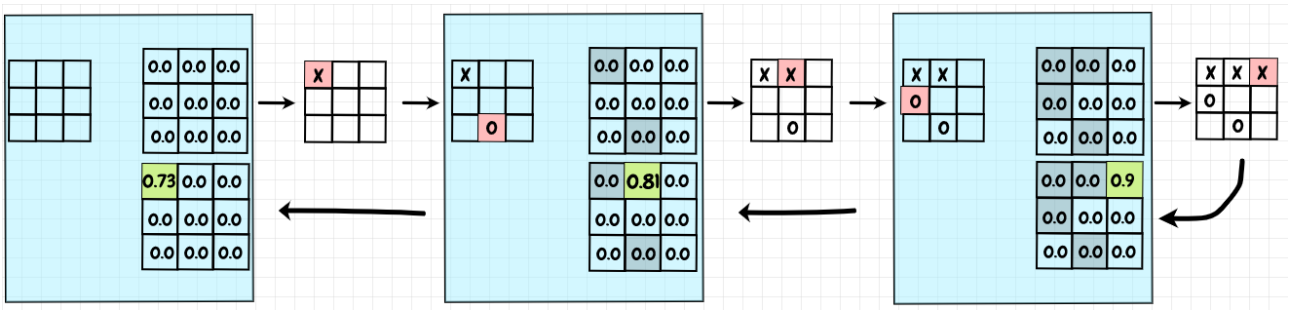

Figure 5: Q-table updates at the end of a game

Since Tic Tac Toe is a two-player games, we need to define a transfer player to play against our Q-Learning agent during training. By default, the transfer player is set to another Q-Learning Player, who will update the Q-table in the similar fashion to our Q-Learning Player, but with `-reward`. This transfer player can also be set to a Random Player, Minimax/AlphaBeta Player, MCTS Player, or a Human Player.

Note:

- Since for Tic Tac Toe, the reward is only received at the end of the game, we should only update the Q-table for all of the (state, action) pair in the game history (Figure 5). The update will happen at the end of each training game.
- In `q_learning.py`, we have provided a `hash_board` function to get the key value of current game board to be stored in Q-table.
- To encourage exploration during training, `choose_action` will implement ϵ-greedy strategy for selecting action. That is to randomly select an action once in a while.
- If your implementation is correct, the Tabular Q-Learning agent should have similar performance to Minimax and MCTS agents.

Written Part:

- Define an MDP for the game. Clearly define the states S, actions A, transition model P, and reward model R.
- Record 100 games between Q-Learning agent with every agent except Human Player.
- Try to change the transfer player to Random Player and do the evaluations again. Does training with Random Player help generalize to other players?
- Hyperparameter tuning: Try to vary the discount factor, the learning rate, and exploration rate, and the number of training epochs. What do you observe? What would be a good combination of the hyperparameters for Tic Tac Toe?