Project Report | L Game

James Dyer | https://github.com/James-Dyer/L-game

Sources Used:

Course materials and lecture notes.

Russell, S., & Norvig, P. (2021). *Artificial Intelligence: A Modern Approach* (4th ed.). Hoboken, NJ: Pearson Education.

Python Software Foundation. (n.d.). *Python 3 Documentation*. Retrieved December 13, 2024, from https://docs.python.org/3/

Wikipedia contributors. (n.d.). *L game*. In Wikipedia. Retrieved December 13, 2024, from https://en.wikipedia.org/wiki/L_game

Summary of Features:

- Multiple Gameplay Options:
  - Play against the computer, another human, or watch computer vs. computer.
  - Set the maximum depth for the AI's minimax algorithm to control it's difficulty level or set the AI to move randomly for an easier experience.
  - Set the initial board state for a customized experience.
- Inter-project compatible API:
  - The formatting of game's input/output is standardized, so compatible projects can play games against each other.
- AI Strategies:
  - Implemented Minimax and Alpha-Beta Pruning with a customizable depth.
  - Developed a heuristic evaluation function for efficient decisions.
  - Optimized performance using appropriate data structures and algorithms, as well as caching strategies.

Design Decisions

1. Data Structures

Game State:

I used a dictionary to hold the positions and orientations of players and neutral pieces, along with the turn information. The dictionary structure was chosen due to its ability to store diverse data types in a single container, making it easy to group all relevant game data together.

Board Representation:

I implemented the game board as a 4x4 grid represented by a 2D list. This choice was intuitive because it closely aligns with the spatial structure of the game board, where each cell corresponds to a specific location. The 2D list format simplifies visualization and operations such as determining valid moves, detecting occupied positions, and placing pieces.

Legal Actions:

I chose to store all the possible legal actions in a set, so the lookup of any given action can be done in O(1).

2. Key Functions

- **minimax:** implements the Minimax algorithm with alpha-beta pruning (up to a customizable depth) to evaluate the best possible move for the current player. When the maximum depth is reached, it uses a heuristic evaluation function, evaluate_state, as a fallback to score each possible state.

  A significant improvement I made in my implementation was introducing caching, which improved the efficiency of the algorithm by avoiding evaluations of previously encountered states. Since each state has eight equivalent symmetries (four rotations and four reflections), I saved all the symmetries with their respective evaluation scores and best actions to the cache for fast lookup when I encountered an equivalent state in the future. In my tests, using a maximum depth of 3, caching reduced the computation time for each turn from 6 seconds to 3 seconds on my machine, effectively halving the time required to make a move.

- **evaluate_state:** Calculates a heuristic value for a given game state, reflecting the favorability of the state for the current player. See 3. The Hurestic

- **buildBoard:** Takes the current game state and translates it into a 2D list representing the board. This function is for any operation requiring a 2D array, such as determining valid moves, detecting occupied positions, and placing pieces.
- **playerTurn:** Handles the logic of displaying the board, prompting for input, validating moves, and updating the game state for each players turn.
- **computerTurn:** Similar to playerTurn, it runs for each computer's turn and handles the logic of displaying the board, prompting for input, validating moves, and updating the game state. This function, however, automates the computer's move using the minimax algorithm and updates the game state accordingly.
- **getLegalActions:** Computes all possible legal moves for the current player based on the game state.

3. The Heuristic

The heuristic function evaluates the favorability of a game state by calculating the difference in the number of legal moves available to the current player and their opponent. The idea behind this heuristic is that it will always avoid being "trapped" by the opponent, as it will maximize the number of moves it can always escape to. At the same time, it will attempt to minimize the number of possible moves for the opponent, restricting the opponent as much as possible until they make a fatal mistake.

I experimented with manually favoring moving the neutral pieces to "aggressive" positions, aiming to encourage the AI to end games quickly against a logically weaker opponent. The idea was to favor positioning neutral pieces on the edges of the board, which is necessary to end the game.

However, I ultimately decided against this approach as I was worried the focus on aggressive neutral piece placements could expose a potential weakness in the AI, making it more vulnerable to errors or exploitation.

4. Relevant Issues

I found that the typical branching factor was around 70, with 14 being the smallest possible branching factor greater than 0. This is because for every L piece move, there are 14 possible neutral piece moves.

5. Extra commands and flags

I added some extra functionality to my program, both for debugging and for better user experience. When launching the game from the command line, you can use the following flags:

- **--randomCPU:** Makes the player 2 CPU choose moves at random, letting you test your AI against a strategically weak opponent.
- **--debug:** Enables various statistics about the game such as nodes evaluated, execution time (per turn), execution time (in total), etc. It also enables printing the traceback of exceptions.
- **--stepbystep:** When playing computer vs computer mode, enabling this flag will pause the game between each computer's turn, letting the user examine the board and any relevant information, then proceed when they are ready.

In addition, you can use the following commands at the command line while playing a game:

- **help():** Provides the player with a possible legal move.
- **save():** Prints a string that represents the current game state. This string can later be used to reinitialize the board using the set initial state feature.
- **quit():** Allows the user to gracefully exit the program the game is active.
- **0:** Typing '0' will return a list of all the possible moves the current player can make.