Assignment 2: Game Tree Search

Document History

- **Validation scripts (https://q.utoronto.ca/courses/278996/files/22286080?wrap=1)** (https://q.utoronto.ca/courses/278996/files/22286080/download?download_frd=1) provided Oct 4 2022
- Released Sept 27 2022

Overview

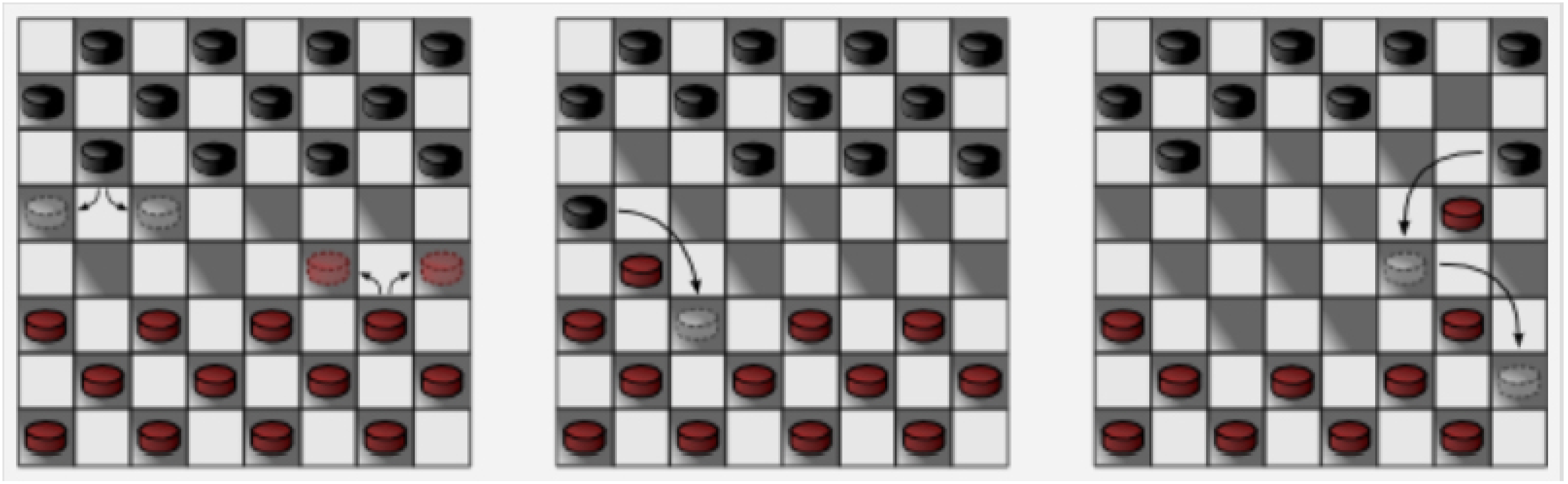
This assignment is an opportunity to implement the game AI algorithms described in class on a classic 2-player game: Checkers! The goal of this assignment is to solve a series of checkers endgame puzzles, where a winning solution can always be found, given a strong enough AI.

The history of Checkers AI is a great story, and a Canadian one as well! Check out the following article for the full tale: How Checkers Was Solved (https://www.theatlantic.com/technology/archive/2017/07/marion-tinsley-checkers/534111/) .

How the Game Works

Checkers is a 2-player board game played with distinct pieces, where one player's pieces are black and the other player's pieces are red. The standard version of the game is played on an 8x8 chess board. The players take turns moving pieces on the board, with the red player typically moving first.

Moving and capturing pieces are dictated by the game's rules. An overview of Checkers is provided here (https://en.wikipedia.org/wiki/Checkers) . We will be coding the version of Checkers called Standard English Draughts, which includes mandatory captures.



The full rules of the game are explained [here](https://en.wikipedia.org/wiki/English_draughts) (https://en.wikipedia.org/wiki/English_draughts) , but the quick summary is:

- **Objective**: Each player's goal is to capture all of their opponent's pieces.
- **Game Ending:** The game ends when one of the players has no pieces remaining or has no legal moves left. In that latter case, if it's a player's turn but they can't move any of their pieces, that player loses the game.
- **Setup:** At the beginning of the game, one player has 12 red pieces and the other player has 12 black pieces. The pieces are placed on the dark squares of the board across the first three rows on each side (Figure 1. at left). For this assignment, your AI will play the red pieces moving up the board while our AI will play the black pieces moving down the board. The red player (that's you) makes the first move.
- **Moves:**
    - Each piece can only move forward diagonally in two ways:
        - Moving one space, if the adjacent space is empty (Figure 1, at left), or
        - Moving two spaces if the adjacent space is occupied by the opponent, and space beyond that is empty (Figure 1, in the middle).
    - In the second case, the player ’captures’ the opponent’s piece that was jumped over.
    - **Multiple captures:**
        - If a move leads to capture, and the piece ends up sitting at a position that can jump over another opponent’s piece, the player can move it again and capture another piece of the opponent (Figure 1, at right).
        - This sequence of moves does not need to be on a straight line and can be ’zigzag’.
    - **Kings:**

- When one of the pieces reaches the opponent's end of the board (the last row, or the first row from the opponent's side), it turns into a ”king” and will be granted a unique power: it can move both forward and backward.

We recommend that you start this assignment by playing some games of checkers to develop a better understanding of how the game works and what strategies can give you an advantage. An online version of the game that allows you to compete against an AI can be found at this link (https://www.mathsisfun.com/games/checkers-2.html) .

What you need to do

You will implement an AI agent that can solve a checkers endgame puzzle, where the agent uses minimax, alpha-beta pruning, and any additional techniques of your choosing. Your agent will return the best move possible, given the endgame puzzle we provide and the time available.

We will use multiple endgame boards to test the correctness and performance of your implementation. An endgame is a board configuration where a winning solution is guaranteed, given the right set of moves. These are considered challenging because the solutions are not obvious or require a large number of moves to arrive at.

Evaluation

Your solution will be marked based on the number of moves your AI agent takes to end the game (compared to the optimal number of moves from a perfect AI agent). When counting the number of moves (also called the **solution length**), we will count moves from both your agent and the opponent. For example, if your agent takes 4 moves to end the game with the opponent taking 3 moves in between, the total number of moves would be 7.

Input Format

The board configuration will be provided in text files with names `input0.txt`, `input1.txt`, etc. Each line in this input file will represent a row from the checkers board, with 8 characters in each row, one for each square on the board. There are 5 possible values for these characters:

- ’`r`’ denotes a red piece,
- ’`b`’ denotes a black piece,
- ’`R`’ denotes a red king,
- ’`B`’ denotes a black king, and

- '`.`' (the period character) denotes an empty square.
- As an example, the contents of the file puzzle1.txt is as follows:

```
........
....b...
.......R
..b.b...
...b...r
........
...r....
....B...
```

For these puzzle layouts, assume that **your AI controls the red pieces and that it's the red player's turn to move**.

Your main program should take two command-line arguments: one for the input file and one for the output file:

```
    python3 checkers.py <input file> <output file>
```

For example, the command `python3 checkers.py input1.txt output1.txt` will take in the board layout stored in `input1.txt` and store the resulting board in file `output1.txt`. The input and output files will both be in standard text format.

Output format

Your program needs to return the best move possible, given the input layout that we provide. This move will be represented by the 8x8 board layout that will result once your AI completes its move. This 8x8 board will be saved in a file whose name is provided by the second command-line argument (as shown above). Once your program finds a solution (or the optimal solution), save it in a file using the following format. The names of the files are explained in the corresponding sections.

For instance, given the endgame layout shown above, your output file would contain the following:

```
........
....b...
.......R
..b.b.r.
...b....
........
...r....
....B...
```

Validation Script

A **validation script (https://q.utoronto.ca/courses/278996/files/22286080?wrap=1)** (https://q.utoronto.ca/courses/278996/files/22286080/download?download_frd=1) has been provided to help you check that you're satisfying the input/output requirements. There are a few test boards provided, but keep in mind that this is a subset of the test cases that we'll use to test your submission

Time and Space Constraints

Your AI agent has at most 5 minutes to make each move on the teach.cs server. Because the number of moves grows exponentially with the depth of the game tree, your program might consume a significant amount of memory if you don't prune or implement any other game tree techniques. Our autotester will timeout after a few minutes, however, your program should be able to run without causing any memory exceptions on the teach.cs servers under typical conditions (we typically test your submission when there is a light load of jobs running on the teach.cs server).

Suggestions

The following are optional suggestions you can follow to implement a Checkers AI agent. In addition to your `checkers.py` file, you must also submit a `heursitics.pdf` file in which you describe the game tree strategies that you implemented.

Minimax: The first thing you will do is to write a function utility function that computes the utility of a game board state for a colour (i.e. whose turn it is). A simple utility could be based on the number of pieces of the player’s colour minus the number of pieces of the opponent, with each regular piece worth one point and each king worth 2. If your agent has a single king and six regular pieces, assign this a value of $2 * 1 + 6 = 8$. If your agent’s opponent has two kings and 3 regular pieces, assign this a value of $2 * 2 + 3 = 7$ points. The difference in points between agents, and the result of the utility function, will therefore be $8 - 7 = 1$.

Then, implement the method select move minimax which will accept a state (which is an object of type Board) as well as the player’s colour (’r’ or ’b’) and return a single move that corresponds with the minimax strategy. More specifically, your function should select the action that leads to the state with the highest minimax value. Implement minimax recursively by writing two functions minimax max node and minimax min node.

Hints: Write a successors function which returns all the successive states (which includes moves and successive board representations) for a given player. Pay attention to which player should make a move for min nodes and max nodes.

Depth Limit: To make your agent really functional, you must implement a depth limit. Change your minimax code to recursively send the limit parameter to both minimax min node and minimax max node. In order to enforce the depth limit in your code, you will want to decrease the limit parameter at each recursion. When you arrive at your depth limit (i.e. when the limit parameter is zero), use a heuristic function to define the value of any non-terminal state. You can call the compute utility function as your heuristic to estimate non-terminal state quality.

Alpha-Beta Pruning: The simple minimax approach can be quite slow. To improve its performance we can write the function select move alpha-beta to compute the best move using alpha-beta pruning. The parameters and return values will be the same as for minimax. Much like minimax, your alpha-beta implementation should recursively call two helper functions: alpha-beta min node and alpha-beta max node. As with minimax, recursively send the limit parameter to alpha-beta min node and alpha-beta max node. When you arrive at your depth limit, call compute utility to generate a utility value for each given state.

Caching States: We can try to speed up the AI even more by caching states we've seen before. To implement state caching you will need to create a dictionary in your AI player (this can just be stored in a global variable on the top level of the file) that maps board states to their minimax value or that checks values against stored alpha and beta parameters. Modify your minimax and alpha-beta pruning functions to store states in that dictionary after their value is known. Then check the dictionary at the beginning of each function. If a state is already in the dictionary, then do not explore it again.

Your Own Heuristic The prior steps should give you a good AI player, but we have only scratched the surface. There are many possible improvements that would create an even better AI. To improve your AI, create your own game heuristic. You can use this in place of computing utility in your alpha-beta or Minimax routines. Some ideas for heuristic functions for checkers could include:

1. Considering board locations where pieces are stable (i.e. where they cannot be captured anymore),
2. Consider the number of moves you and your opponent can make given the current board configuration.
3. Tuning your strategy towards the end game.

You can also do your own research to find a wide range of other good heuristics (for example, here (https://www.ultraboardgames.com/checkers/tips.php) is a good start).

Node Ordering Heuristic: Finally, note that alpha-beta pruning works better if nodes that lead to a better utility are explored first. To do this, in the Alpha-beta pruning functions, try using your heuristic to order the nodes that you explore. If your heuristic makes reasonably good estimates of non-terminal state values, this should lead to a little bit of a speed up because a good ordering will lead to more pruning.

Mark Breakdown

The expectation is that you will implement minimax as a bare minimum, and alpha-beta pruning as well to get the majority of the marks. The additional techniques and heuristics will earn you the rest of the marks, with part marks possible for partly correct implementations of certain functions. The formula for your final mark will be calculated as follows, with "correctness" representing the autotesting results:

A2 Mark = Correctness x Difficulty (where "difficulty" is specified below):

Marking Scheme

Option	Difficulty
Uniform Cost Search	10%
Search + Minimax (with a simple heuristic)	60%
Search + Minimax + Alpha-beta pruning	85%

In addition to this, 15% of your mark for this assignment is earned through the implementation of additional heuristics or techniques that you add to your game tree search. These are meant to be part of your solution and described in a file called `heuristics.pdf`, which you will submit on Markus along with your `checkers.py` file. This mark is assigned separately from your correctness mark (i.e. you can get full marks for your heuristics even if you don't get full correctness marks).