Assignment 2: Solving Checkers Endgames using Search

Document update history:

- Added a section on "Solving the Puzzles"

Overview

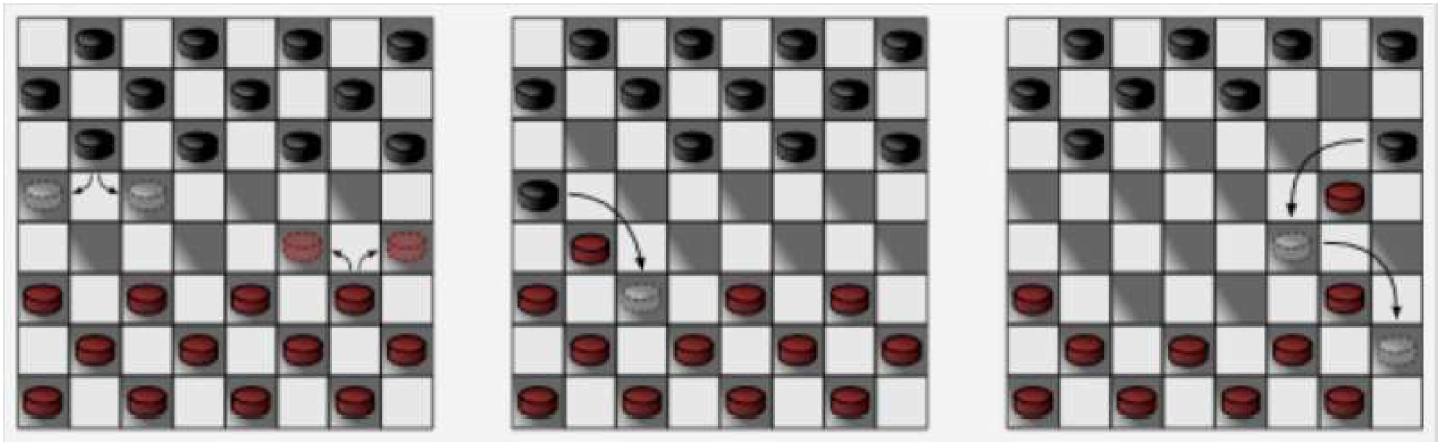
In this assignment, you will implement search algorithms to solve several Checkers endgame puzzles, where a winning solution can always be found, given a strong enough AI.

The history of Checkers AI is a great story and a Canadian one! 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/)**.

Game Rules

Checkers is a two-player board game played on an eight by eight chess board. One player's pieces are **black**, and the other player's pieces are **red.** The players take turns moving pieces on the board. The **red** player moves first.

We recommend that you start by playing some games of checkers to understand the game better. You can **player checkers at this link (https://www.mathsisfun.com/games/checkers-2.html)**.



We will be coding the version of Checkers called **Standard English Draughts (https://en.wikipedia.org/wiki/English_draughts)**. You can find the full rules at the link and also below.

- **Starting Position:** Each player starts with 12 pieces on the dark squares of the three rows closest to that player's side. The **black** pieces start from the **top three** rows, and the **red** ones start from the **bottom three**. The **red** player makes the first move.

- **Move Rules:** There are two different ways to move.
  - **Simple move** (see the left image in the figure above): A simple move involves moving a piece one square diagonally to an adjacent unoccupied dark square. Normal pieces can move diagonally forward only; kings can move in any diagonal direction. (For the black player, forward is down. For the red player, forward is up.)
  - **Jump** (see the middle image in the figure above): A jump consists of moving a piece diagonally adjacent to an opponent's piece to an empty square immediately beyond it in the same direction (thus "jumping over" the opponent's piece front and back.) Normal pieces can jump diagonally forward only; kings can jump in any diagonal direction. A jumped piece is "captured" and removed from the game. Any piece, king or normal, can jump a king.
  - **Jumping is mandatory**. If a player has the option to jump, they must make it, even if doing so results in a disadvantage for the jumping player.
  - **Multiple jumps** (see the right image in the figure above): After one jump, if the moved piece can jump another opponent's piece, it must keep jumping until no more jumps are possible, even if the jump is in a different diagonal direction. If more than one multi-jump is available, the player can choose which piece to jump with and which sequence of jumps to make. The sequence chosen is not required to be the one that maximizes the number of jumps in turn. However, a player must make all the available jumps in the sequence chosen.
- **Kings**:
  - If a piece moves into the last row on the opponent's side of the board, it becomes a king and can move both forward and backward. A red piece becomes king when it reaches the top row, and a black piece becomes king when it reaches the bottom row.
  - If a piece becomes a king, the current move terminates; The piece cannot jump back as in a multi-jump until the next move.
- **End of Game:** A player wins by capturing all the opponent's pieces or when the opponent has no legal moves left.

Your Tasks

You will implement a program that can solve a Checkers endgame puzzle using alpha-beta pruning and additional techniques of your choosing.

Running Your Program

You will submit a file named **checkers.py**, which contains your program that uses alpha-beta pruning and other techniques to solve a Checkers endgame puzzle.

Your program **must use python3 and run on the teach.cs server** (where we run our auto-testing script).

We will test your program using several Checkers endgame puzzles. For each puzzle, we will run the following command.

```
python3 checkers.py --inputfile <input file> --outputfile <output file>
```

Each command specifies one plain-text input file and one plain-text output file.

For example, if we run the following command for an input file **puzzle1.txt**:

```
python3 checkers.py --inputfile puzzle1.txt --outputfile puzzle1_sol.txt
```

The solution to puzzle1.txt will be in **puzzle1_sol.txt**.

Input and Output Formats

We will represent each state in the following format.

- Each state is a grid of 64 characters. The grid has eight rows with eight characters per row.
- '`.`' (the period character) denotes an empty square.
- '`r`' denotes a red piece,
- '`b`' denotes a black piece,
- '`R`' denotes a red king,
- '`B`' denotes a black king.

Here is an example of a state.

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

Each input file contains one state. Your program controls the red pieces, and it is your turn to make a move.

Each output file contains the sequence of states until the end of the game. The first state is the same as the state in the input file. The last state is a state denoting the end of the game. There is one empty line between any two consecutive states.

Here are examples of **an input file (https://q.utoronto.ca/courses/293717/files/24641843?wrap=1)** **(https://q.utoronto.ca/courses/293717/files/24641843/download?download_frd=1)** and **an output file** (https://q.utoronto.ca/courses/293717/files/24641844?wrap=1) (https://q.utoronto.ca/courses/293717/files/24641844/download?download_frd=1) .

Solving the Puzzles

Your program controls both players. Both players use alpha-beta pruning with the same depth limit to determine their best move at their turn.

The game proceeds as follows.

- **Start by specifying a depth limit.**
- **The red player runs alpha-beta pruning with the depth limit to determine their move. Then, the red player carries out the move.**
- **Next, the black player runs alpha-beta pruning with the depth limit to determine their move. Then, the black player carries out the move.**
- **This process continues until the game ends.**

At each turn, perform alpha-beta pruning.

- **When alpha-beta pruning reaches a terminal state (i.e. the end of the game), return +infinithy for the winning player or -infinity for the losing player.**
- **When alpha-beta pruning reaches the depth limit, return a utility based on your evaluation function.**

Depth Limit:

  - **You need to determine a suitable depth limit for your submission.**
  - **Setting the depth limit comes with a trade-off. If your depth limit is too small, both players are weak and will likely not find the optimal solution (a way to win with the least number of moves). If your depth limit is too large, your program will take too long to run and may not terminate within the time limit.**

Testing Your Program

We will test your program on multiple puzzles of various difficulties.

Our tests will verify that:

- Your solution is valid.
  - The first state in the solution is the initial state of the puzzle. The last state in the solution is the end of the game. Every state is a successor of the previous state.
- Your solution is optimal.
  - Your agent makes the optimal move, given that your opponent is a strong enough AI agent.

- The efficiency of your program.
  - The less time it takes for your program to terminate, the more marks you will receive.
  - See the marking scheme section for details.

Your Program Design

Part of your assignment grade depends on the efficiency of your program. We encourage you to experiment with many techniques to optimize your program. In class, we mentioned two optimization techniques: node ordering and state caching.

Submit a one-page file named **design.pdf**. (We will grade the first page only if your file is longer than one page.) In this file, answer the three questions below.

- How does your program calculate the utility of each terminal state? Describe briefly.
- How does your program estimate the utility of each non-terminal state? Describe your evaluation function in a few sentences.
- Does your program perform other optimizations, such as node ordering or state caching? If so, describe each optimization in a few sentences.

Submission and Marking Scheme

You should submit two files on MarkUs.

- **checkers.py** has your Python program that executes alpha-beta pruning to solve a Checkers endgame puzzle.
- **design.pdf** is a one-page write-up answering a few questions regarding your program design.

We will test your program on several puzzles. For each puzzle, your program must terminate within 4 minutes. We strongly recommend testing your program on the **teach.cs** server to ensure that it terminates under the time limit.

We will provide two puzzles to you on MarkUs. Please use these to test the correctness and efficiency of your program.

The mark breakdown is as follows:

Marking Scheme

Component	Percentage
Solution correctness	30%
Program efficiency	60%
Descriptions of your program design	10%

The time limit for each test case is 4 minutes. The marking scheme for program efficiency is below.

Program Efficiency Marking Scheme

Runtime Criteria	**Percentage Earned (up to 60%)**
Your program terminates within 1 minute.	60%
Your program terminates within 2 minutes.	45%
Your program terminates within 4 minutes.	30%
Your program does NOT terminate within 4 minutes.	0%

Suggestions

Here are some suggested steps to tackle this assignment. Good luck!

Write a few classes and functions to simulate a Checkers game.

- Your first step is to write code to simulate a Checkers game. This is an important and necessary step if you want to solve a real problem using an AI algorithm. Spend some time playing checkers and understanding the rule. Then think about how to implement the game using multiple helper functions.
- Implement a class to represent a state.
  - The state contains a board, which is an eight-by-eight grid of characters. (Checkers is arguably easier to deal with than Huarongdao because you no longer have pieces of different sizes.)
  - Write a function to print the board for debugging.
- **Generate Successors:** This function takes a state and returns a list of its successor states.
  - There are two types of moves: simple moves and jumps. Be sure to consider both.
  - Sometimes many multi-jumps are possible. We recommend implementing a helper function to generate all possible jump sequences. This function may be recursive or iterative.
  - Our game has mandatory capture; if you can jump, you must jump. When generating possible moves, check whether a jump is possible. If a jump is possible, the player should not be allowed to make simple moves.

Write a few functions to implement Alpha-Beta Pruning.

- **Utility Function:** This function computes a player's utility of a **terminal** state.
    - **We strongly recommend returning +infinity for the winning player and -infinity for the losing player.**
- **Evaluation Function (to Estimate Utility):** This function estimates a player's utility of a **non-terminal** state.
    - Searching until the terminal states is often impossible when solving a Checkers endgame puzzle. Therefore, we must cut off the search at a pre-specified depth limit and estimate each player's utility using an evaluation function.
    - To construct this evaluation function, we suggest that you think about useful features of each state and incorporate these features into the function.
    - We start with a simple evaluation function to motivate each player to maximize the number of their pieces.
        - For example, suppose a normal piece is worth one point, and each king is worth two points. A player's utility is the player's total points minus the opponent's total. If your player has a single king and six regular pieces, and your opponent has two kings and three regular pieces. Then this simple utility value for your agent is (2 ∗ 1 + 6) - (2 * 2 + 3) = 8 - 7 = 1.
    - Here are some potentially useful features:
        - It is better to have pieces that are more advanced on the board. For example, the red player prefers red pieces in the top half of the board.
        - It is better to have pieces in the center of the board.
        - Considering board locations where pieces are stable (i.e. where they cannot be captured anymore). For example, a piece on the left or right edge is safe from capture.
        - Consider the number of moves you and your opponent can make, given the current board configuration.
        - You might find more ideas here (https://www.ultraboardgames.com/checkers/tips.php).
- **Alpha-Beta Pruning:** Implement the alpha-beta pruning by following the pseudocode in the slides. Make sure to implement a depth limit. When your program reaches the depth limit, apply the evaluation function.

Implement additional optimizations to speed up your program execution.

- **Node Ordering:** The alpha-beta algorithm performs more pruning if nodes that lead to a better utility are explored first. Try ordering the successor states using your heuristic function in the alpha-beta pruning functions.
    - The simplest approach is to order the successors by your evaluation function.
- **Caching States:** Try to speed up your program by remembering states we have seen before.

  - Create a dictionary (e.g. as a global variable) that maps each state to some useful information about the state.
  - You can decide what useful information to store in the dictionary. For example, you can store the state's minimax value and/or alpha and beta parameters.
  - Once you visit a state, store the information in the dictionary. Then, check the dictionary at the beginning of each function. If a state is already in the dictionary, then do not explore it again.
- **Implement a more sophisticated evaluation function.**
  - Consider incorporating more features into your evaluation function.
  - Note that a more complicated evaluation function is not necessarily better. You will have to experiment with different evaluation functions and decide which one to use.