National University of Singapore
School of Computing
CS3243 Introduction to Artificial Intelligence

Project 3: Adversarial Search

Issued: 14 October 2024 Due: 10 November 2024, 2359hrs

1 Overview

In this project, you will **implement an adversarial search algorithm to find valid and good moves for a modified chess game**. Specifically, you are tasked to implement the **Alpha-Beta Pruning algorithm** to play a modified game of chess.

This project is worth 10% of your course grade, with an extra 2% bonus.

1.1 General Project Requirements

The general project requirements are as follows:

- **Individual** project: Discussion within a team is allowed, but **no code should be shared**
- Python Version: $\geq$ **3.12**
- Deadline: **10 November 2024, 2359hrs**
- Submission: Via **Coursemology** – for details, refer to the Coursemology Guide on Canvas

1.2 Academic Integrity and Late Submissions

Note that any material that does not originate from you (e.g., is taken from another source) should not be used directly. You should do up the solutions on your own. Failure to do so constitutes plagiarism. Sharing of code between individuals is also strictly not allowed. Students found plagiarising will be dealt with seriously.

For late submissions, there will be a 20% penalty for submissions received within 24 hours after the deadline, 50% penalty for submissions received between 24-48 hours after the deadline, and 100% penalty for submissions received after 48 hours after the deadline. For example, if you submit the project 30 hours after the deadline and obtain a score of 92%, a 50% penalty applies, and you will only be awarded 46%.

2 Background: Fairy Chess

Fairy chess is a modified chess game that includes pieces with nonstandard movements[1]. In this project, you will be creating an agent that can find the best move for a given fairy chess board position. Your agent must be implemented using the Alpha-Beta pruning algorithm.

2.1 Rules of the Game

2.1.1 Board

For ease of implementation, we will not use standard chess board notation. Instead, squares on the chessboard will be represented as a `Tuple[int,int]`, representing the row and column of the square. The top-most row is row 0, and the left-most column is column 0. An example chessboard is depicted in Figure 1 below.

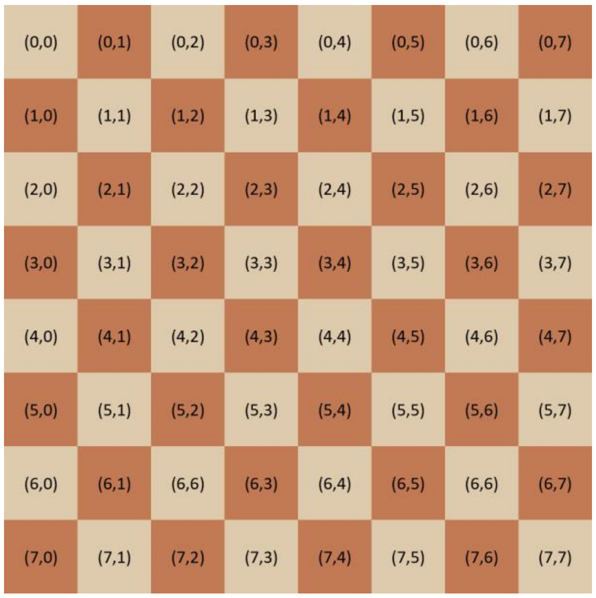

Figure 1: Chess board with labelled squares

2.1.2 Movement of Chess Pieces

The classic chess pieces that will be used are as follows.

- **King**: The king can move one square in any direction.
- **Rook**: A rook can move any number of squares along a rank or file but cannot leap over other pieces.
- **Bishop**: A bishop can move any number of squares diagonally but cannot leap over other pieces.
- **Knight**: A knight moves in an "L"-shape, i.e., two squares vertically and one square

[1] https://en.wikipedia.org/wiki/Fairy_chess

horizontally, or two squares horizontally and one square vertically. Note that the knight can leap over other pieces.

In addition to the standard chess pieces mentioned above, two other fairy chess pieces will be defined.

- **Squire**: A squire can move to any square a Manhattan distance of 2 away. A squire can also leap over other pieces.
- **Combatant**: When moving (without capturing), the combatant moves one square orthogonally in any direction (denoted by points in Figure 2). When capturing, the combatant captures one square diagonally in any direction (denoted by crosses in Figure 2).

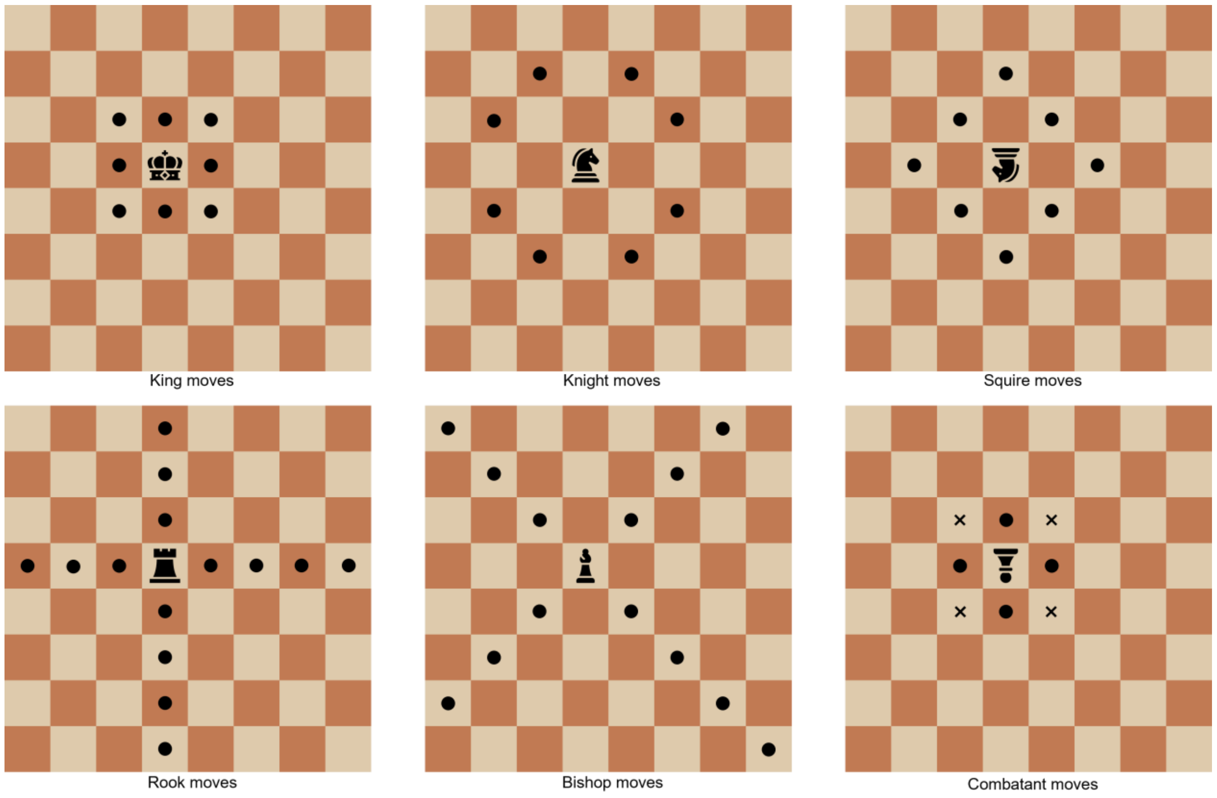

Figure 2: Chess piece movement

2.1.3 Winning Conditions

In this variant, the game is won by capturing your opponent's King; it is lost when your King is captured. Note that this means the concepts of **check**, **checkmate**, and **stalemate** ***do not apply***. An agent may (either accidentally or being forced to) put its own King under attack, and if a King is under attack, an agent is not forced to defend it. The game ends when any King is captured, hence, if both Kings are under attack, the player that captures the opponent King first wins.

With this modified winning condition, the concept of stalemate also does not apply. A stalemated King is forced to move into danger on the next move, resulting in a loss for the stalemated player.

2.1.4 Draws

A draw will be declared if only Kings are left on the board. The game is drawn even if one King can capture the other on the very next move.

2.2 Getting Started

You are given one python file (`AB.py`) with the recommended empty functions/classes to be implemented. Specifically, the following files are given.

1. `studentAgent(gameboard)`: This function takes in a parameter `gameboard` and returns a move. More details on the output will be given in the later sections. **DO NOT REMOVE THIS FUNCTION.**
2. `get_legal_moves(gameboard,color)`: This function takes in two parameters, `gameboard` and `color`. This function should return a `list` of moves. **DO NOT REMOVE THIS FUNCTION.**
3. `ab()`: Implement the Alpha-Beta Pruning algorithm here. (You may change/remove this function, as long as you keep the `studentAgent(gameboard)` function and return the required value(s).)
4. `State`: A class storing information of the game state. (You may change/remove this function as desired, as long as you keep the `studentAgent(gameboard)` function and return the required value(s).)
5. `Board`: A class storing information of the board. (You may change/remove this function as desired, as long as you keep the `studentAgent(gameboard)` function and return the required value(s).)
6. `Piece`: A class storing information of a game piece. (You may change/remove this function as desired, as long as you keep the `studentAgent(gameboard)` function and return the required value(s).)

You are encouraged to write other helper functions to aid you in implementing the algorithms. Do note that during testing, the Autograder will call both `studentAgent(gameboard)` and `get_legal_moves(gameboard,color)`.

2.3 Winning a Fairy Chess Game Using Alpha-Beta Pruning

The objectives of this project are as follows.

1. To gain exposure in the implementation of the algorithms taught in class.
2. To learn how to apply the Minimax algorithm to games.
3. To learn the efficiency and importance of using Alpha-Beta Pruning.

3 Project 3 Task 1

Many years have passed since the last invasion and the people of CS3243 kingdom are living peacefully and happily. However, the council of CS3243 Kingdom detects an imminent threat from our enemy Kingdom, CS9999 and are fearful that a war may break out soon. In order to prepare for the war, the council seeks to recruit a strategic advisor. To assess the best candidate, the council decides to organise a fairy chess competition that anyone can join. The participants will compete against AI agents of various difficulty levels and the participant that wins against all the AI agents will be recruited as the strategic advisor. Being an avid fairy chess player in the kingdom, you decide to join the competition to challenge yourself!

3.1 Task 1: Alpha-Beta Pruning Algorithm

Implement the Alpha-Beta pruning algorithm to find the best move for a given fairy chess board state.

3.1.1 Input

The function `studentAgent()` takes in a parameter `gameboard` that is a `list` of all the pieces on the board. Each piece is represented by a `tuple` containing 3 elements. The first element is a string containing the name of the piece. The second element is either the string `'white'` or `'black'`, and this element represents the colour of the piece. The last element is another tuple, representing the position of the piece using the notation defined in Figure 1. An example of the `gameboard` is given below.

```
[('King','white',(7, 0)), ('King','black',(0, 7)), ('Knight','white',(3, 4))]
```

3.1.2 Output

When the `studentAgent()` function is executed, your code should return a valid **move** in the following format.

$$(\mathtt{pos}_1,\ \mathtt{pos}_2)$$

To be more precise, your `studentAgent()` function should return a tuple containing two other tuples. The first tuple represents the starting position of the piece you intend to move. The second tuple represents the position you intend to move the piece to. Both positions are denoted using the notation in Figure 1. Note that captures are not denoted differently from other moves, if $\mathtt{pos}_2$ is occupied by an opponent piece, it is guaranteed that the piece at $\mathtt{pos}_2$ is captured after the move.

In addition, you are tasked to define a `get_legal_moves(gameboard,color)` function. This function should take in a `gameboard` as described above, and the string `'black'` or `'white'`. This function should return a `list` all the legal moves available to the player whose colour is specified. An example is shown below.

Assume that the following is executed.

```
gameboard = [  ('King','white',(7, 0)),
               ('King','black',(0, 7)),
               ('Knight','white',(3, 4)) ]
print(get_legal_moves(gameboard,'white'))
print(studentAgent(gameboard))
```

Consequently, the sample output representing your Knight at (3,4) moving to (5,5) is as follows.

```
# output of get_legal_moves:
[ ((7,0),(7,1)), ((7,0),(6,0)), ((7,0),(6,1)), ((3,4),(5,5)),
  ((3,4),(4,6)), ((3,4),(1,5)), ((3,4),(2,6)), ((3,4),(5,3)),
  ((3,4),(4,2)), ((3,4),(1,3)), ((3,4),(2,2)) ]
# output of studentAgent:
( (3,4),(5,5) )
```

3.1.3 Assumptions

In your implementation, you may assume the following.

- Both players will have a King on the board.
- With the exception of the first four test cases (where the `get_legal_moves` function is evaluated), every other test case is a board position such that one move is clearly better than all other moves (i.e., only one move leads to a forced win or leads to winning material[2]).
- In the board position given, you will only play as "white".

You may assume that the opponent plays optimally.

[2] "Winning material" refers to either capturing an opponent piece for free or trading your lower valued piece for an opponent's higher valued piece.

3.2 Grading (Total: 10 marks + 2 bonus marks)

3.2.1 Evaluating Your Agent

Your agent will not be playing a game. Instead, your agent will be tasked to find the best move for several given board positions. However, do feel free to let your agent play against itself to view the performance of your agent in a real game.

The test cases will test for the following:

- **Test cases b1 - b4**: Find all the legal moves for a given board position.
- **Test cases m1_1 - m1_2**: Find the best move for a board position where there is a forced win in one move (i.e., you may capture the black King in one move).
- **Test cases m3_1 - m3_4**: Find the best move for a board position where there is a forced win in three moves (i.e., two moves by white, one move by black).
- **Test cases m5_1 - m5_4**: Find the best move in a board position where there is a forced win in five moves (i.e., three moves by white, two moves by black).
- **Test cases e3_1 - e3_2**: Find the best move in a board position where there is a guaranteed way to win material in three moves.
- **Test cases e5_1 - e5_4**: Find the best move in a board position where there is a guaranteed way to win material in five moves.
- **Bonus test cases e5_5 - e5_6**: Find the best move in a board position where there is a guaranteed way to win material in five moves (note: these are very tricky test cases).

3.2.2 Marks Distribution

The grading rubrics are as follows:

- **Test cases b1 - b4** (to test the `get_legal_moves` function): **0.5m each [total: 2m]**
- **Test cases m1_1 - e5_4**: **0.5m each [total: 8m]**
- **Bonus test cases e5_5 - e5_6**: **1m each [total bonus: 2m]**

---

4 Submission

4.1 Details for Submission via Coursemology

Refer to **Canvas** > **CS3243** > **Files** > **Projects** > **Coursemology_guide.pdf** for submission details.

---

5 Appendix

5.1 Allowed Libraries

The following libraries are allowed:

- Data structures: queue, collections, heapq, array, copy, enum, string
- Math: numbers, math, decimal, fractions, random, numpy
- Functional: itertools, functools, operators
- Types: types, typing

5.2 Tips and Hints

- **IMPORTANT: Do not spend too much time optimising the heuristic!** A simple weighted sum of piece values is sufficient. Evaluation of more complicated heuristics may slow down your code significantly due to the frequency with which the evaluation function is called. You may search online to determine how to define a good evaluation function.
- With an average computer, you may expect the slowest test cases to run in under 2 seconds (locally).
- ChatGPT is quite useful for generating the skeletal code for minimax, as well as the piece movements. You may use that as a starting point. For Project 3, you are allowed to use ChatGPT to assist you.