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

Project 3: Adversarial Search

Issued: 18 October 2023 Due: 12 November 2023

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:

1. Implement the **Alpha-Beta Pruning** algorithm to play a modified game of chess.

This project is worth 10% of your module grade.

1.1 General Project Requirements

The general project requirements are as follows:

- **Individual** project
- Python Version: **3.10** or later[1]
- Submission deadline: **TBC**
- Submission folder: **Canvas** > **CS3243** > **Assignments** > **Project 3 Submission folder**
- Submission format: One standard (non-encrypted) **zip file**[2] containing only the necessary project files. In particular, it should unzip to give one folder with one `.py` file: `AB.py`. More information is given in the Submission Details section below.

As to the specific project requirements, you must complete and submit the following:

- Task 1 (Alpha-Beta): Implement your Alpha-Beta Pruning algorithm in `AB.py`.

Note that you are tasked to implement your own board, chess pieces and states for the game.

---

[1] CodePost runs on Python 3.7, avoid using match-case and walrus operators in your submission to CodePost

[2] Note that it is the responsibility of the students to ensure that this file may be accessed by conventional means.

1.2 Academic Integrity and Late Submissions

Note that any material that does not originate from you (e.g., is taken from another source, with the exception of the P2ST (ChatGPT) app), should not be used directly. You should do up the solutions on your own. Failure to do so constitutes plagiarism. Sharing of materials between individuals is also strictly not allowed. Students found plagiarising or sharing their code will be dealt with seriously. Additionally, sharing of code on any public repository is strictly not allowed and any students found sharing 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 to play a game of fairy chess. Your agent must be implemented using the Alpha-Beta pruning algorithm. Your agent will play against other AI agents from differing starting positions.

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 given below.

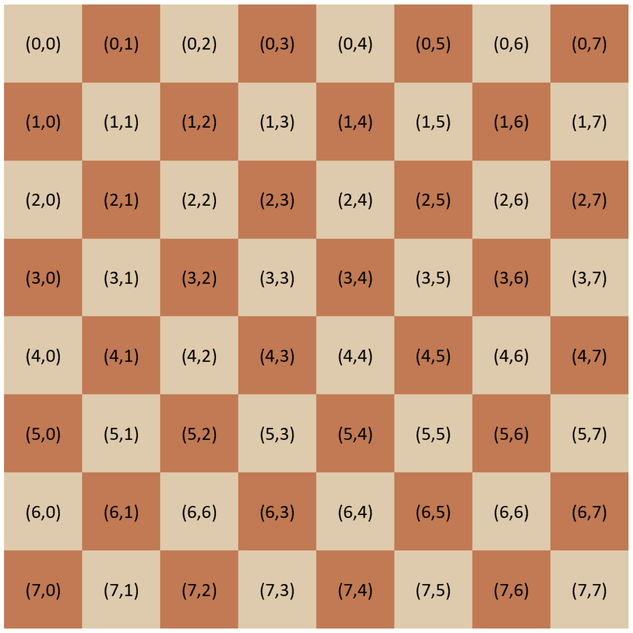


Figure 1: Chess board with labelled squares

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

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: two squares vertically and one square 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 orthorgonally 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).

The moves of various pieces are illustrated in Figure 2.

2.1.3 Winning condition

In this variant, the game is won by capturing your opponent’s King, and is lost when your King is captured. Note that this means the concepts of check, checkmate, and stalemate does not apply here. 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 Draw

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. A draw will also be declared if the game is still ongoing after 50 turns. That is to say, 50 moves have each been made by white and by black.

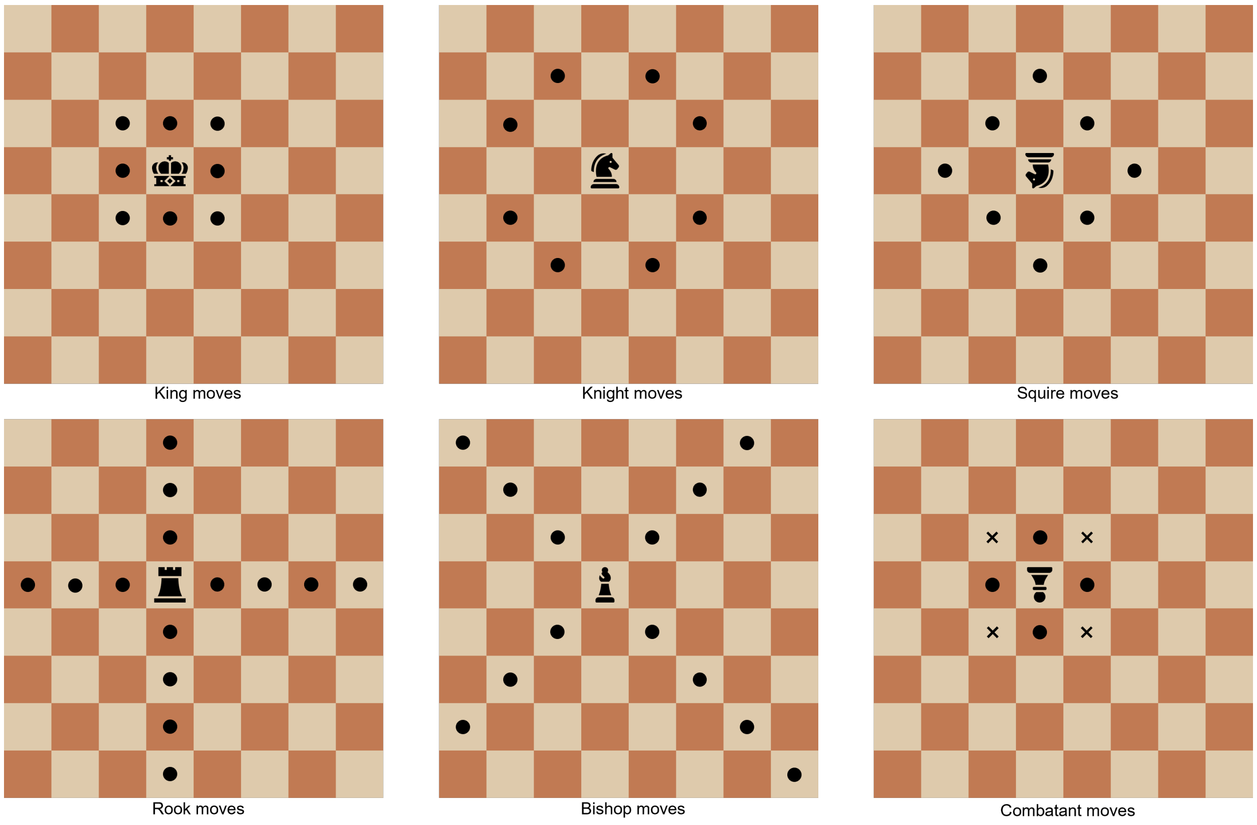


Figure 2: Movement of chess pieces

2.2 Getting Started

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

1. `studentAgent(gameboard)`: **DO NOT REMOVE THIS FUNCTION.** This function takes in a parameter `gameboard`. When called, this function must return a valid **Move**. The output will be used to make a move on the game. More details on the output will be given in the later sections.
2. `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).)
3. `setUpBoard()`: **Optional.** This functions allow you to take in a tuple of pieces as an argument to set up the gameboard. You may not have to use this function as the gameboard will be given to you as a parameter in the `studentAgent(gameboard)` function.
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. During testing, `studentAgent(gameboard)` from `AB.py` will be called by the autograder.

2.3 Winning a fairy chess game using Alpha-Beta Pruning

The objectives of this project are:

1. To gain exposure in the implementation of algorithms taught in class.
2. To learn to apply Minimax in 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 in `AB.py` to win a game of fairy chess.

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 color 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:

$$(pos_1,\ pos_2)$$

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 pos_2 is occupied by an opponent piece, it is guaranteed that the piece at pos_2 is captured after the move.

It is your responsibility to ensure that all the moves output by your `studentAgent()` function is legal. The autograder will check for the legality of the move, and if any move is found to be illegal, the test case will be considered failed. Illegal moves includes scenarios like pos_1 not being occupied by a piece, the piece at pos_1 not being able to move to pos_2 under the rules defined in Figure 2, capturing friendly pieces, moving your opponent's pieces, or moving off the board.

An example of the function output is shown below:

```
print(studentAgent(gameboard))
```

Sample output (that represents moving your Knight at (3,4) to (5,5)):

```
((3,4), (5,5))
```

3.1.3 Assumptions

In your implementation, you may assume the following:

- The starting position will be different for every test case, unlike a regular game of chess, in which the starting position is well defined. However, there is guaranteed to be one and only one white King and black King on the board.
- You are also guaranteed the following in terms of the starting board configuration:
  - No more than one rook per player on the board.
  - No more than two bishops per player on the board. If there are two bishops for one player, they are guaranteed to be starting on opposite colored squares.
  - No more than two squires per player on the board. If there are two squires for one player, they are guaranteed to be starting on opposite colored squares.
  - No more than two knights per player on the board.
  - No more than four combatants per player on the board.
  - No more than ten pieces per player on the board, including the King.
  - Both players have the exact same set of pieces.
- Your `studentAgent(gameboard)` function will only be required to find the best move for white in any given position.

3.1.4 Test script

Your `studentAgent()` function will be tested via a script that is similar to the following:

```
gameboard = ...
while True:
    move = studentAgent(gameboard)
    if isLegal(gameboard, move):
        gameboard = makeMove(gameboard, move)
    if isGameOver(gameboard):
        break

    move = opponentAgent(gameboard)
    if isLegal(gameboard, move):
        gameboard = makeMove(gameboard, move)
    if isGameOver(gameboard):
        break
```

3.2 Grading (Total: 10 marks)

3.2.1 Agent functions

Your agent will be playing against 4 different opponent agents as follows:

- Agent 1: **Random Agent**. The random agent chooses a random legal move from the moves available and plays it.
- Agent 2: **Greedy Agent**. The greedy agent chooses the move that will capture the 'best' piece it can, in the order ['King' , 'Rook', 'Bishop', 'Knight', 'Squire', 'Combatant', No capture]. It does not consider future moves. If two equally good options are available, it will choose a random one to make.
- Agent 3: **Smart Agent**. The smart agent evaluates the board through an undisclosed evaluation function, and chooses the move that will maximise that evaluation. If two equally good options are available, it will choose a random one to make.
- Agent 4: **Minimax Agent**. The Minimax agent will run minimax on depth 4, and the evaluation function used is similar to the one used by the Smart Agent.

3.2.2 Marks distribution

The grading rubrics are as follows:

- **Winning** against **any agent** in a position where there is a forced win in 5 moves or less (ie: 3 moves by white and 2 moves by black. White captures the opponent King on their 3rd move). Four games will be played, starting from the four positions released on Canvas, with each game being worth 0.5m.
- **Winning** against the **random agent** within the time limit. Four games will be played, starting from four different predetermined starting positions, with each game being worth 0.25m.
- **Winning** against the **greedy agent** within the time limit. Four games will be played, starting from four different predetermined starting positions, with each game being worth 0.5m.
- **Winning** against the **smart agent** within the time limit. Four games will be played, starting from four different predetermined starting positions, with each game being worth 0.5m.
- **Winning or drawing** against the **minimax agent** within the time limit. Four games will be played, starting from four different predetermined starting positions, with each game being worth 0.75m.

3.2.3 Time limit

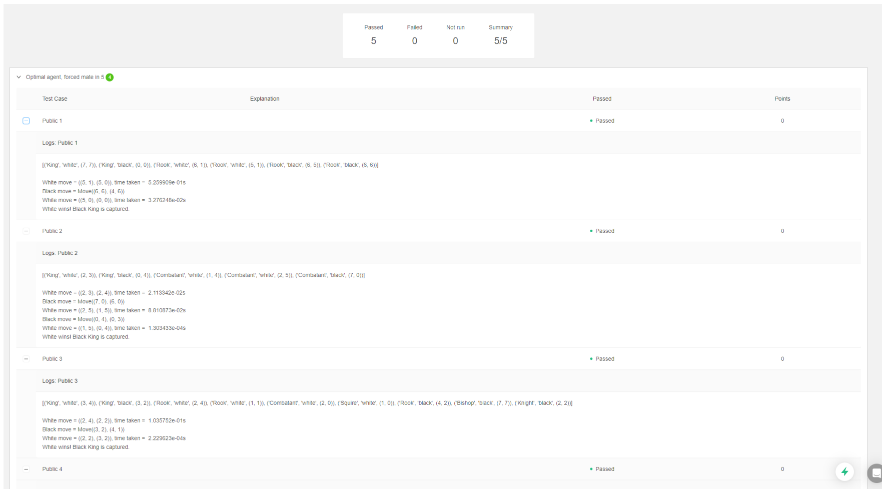
In each game, the total run time of the `studentAgent` function will be recorded. At the end of each game, if the total run time of the `studentAgent` function is less than 1 minute, you will be awarded credit in accordance to 3.2.2. If it is between 1 to 1.5 minutes, you will be awarded **80%** of the marks stated in 3.2.2. If it is between 1.5 to 2 minutes, you will be awarded **50%** of the marks stated in 3.2.2. Otherwise, no marks will be awarded for that particular game.

3.3 CodePost (Platform for Running Autograder)

We will be using CodePost as our platform to run the autograder and test your code. You should already be familiar with the platform from Project 1 and 2. If you need any assistance for CodePost signups, please feel free to reach out and email any of the TAs.

Subsequent access: `https://codepost.io/login`

1. Choose an agent which you want to test your code against. Click on "Upload assignment", and upload your python file `AB.py` (do NOT rename the file).
2. **Note that you should not print any output in your Python files but only return the required values as this may interfere with the Autograder.**
3. After the submission has been processed, refresh the page and select "View feedback".
4. After following the above steps, you should see the following:



Passed	Failed	Not run	Summary
5	0	0	5/5

Optimal agent, forced mate in 5 4

Test Case	Explanation	Passed	Points
Public 1		Passed	0

```
Logs: Public 1

[('King', 'white', (7, 7)), ('King', 'black', (0, 0)), ('Rook', 'white', (6, 1)), ('Rook', 'white', (5, 1)), ('Rook', 'black', (6, 5)), ('Rook', 'black', (6, 6))]

White move = ((5, 1), (5, 0)), time taken =  5.259909e-01s
Black move = Move((6, 6), (4, 6))
White move = ((5, 0), (0, 0)), time taken =  3.276248e-02s
White wins! Black King is captured.
```

Test Case	Explanation	Passed	Points
Public 2		Passed	0

```
Logs: Public 2

[('King', 'white', (2, 3)), ('King', 'black', (0, 4)), ('Combatant', 'white', (1, 4)), ('Combatant', 'white', (2, 5)), ('Combatant', 'black', (7, 0))]

White move = ((2, 3), (2, 4)), time taken =  2.113342e-02s
Black move = Move((7, 0), (6, 0))
White move = ((2, 5), (1, 5)), time taken =  8.810873e-02s
Black move = Move((0, 4), (0, 3))
White move = ((1, 5), (0, 4)), time taken =  1.303433e-04s
White wins! Black King is captured.
```

Test Case	Explanation	Passed	Points
Public 3		Passed	0

```
Logs: Public 3

[('King', 'white', (3, 4)), ('King', 'black', (3, 2)), ('Rook', 'white', (2, 4)), ('Rook', 'white', (1, 1)), ('Combatant', 'white', (2, 0)), ('Squire', 'white', (1, 0)), ('Rook', 'black', (4, 2)), ('Bishop', 'black', (7, 7)), ('Knight', 'black', (2, 2))]

White move = ((2, 4), (2, 2)), time taken =  1.035752e-01s
Black move = Move((3, 2), (4, 1))
White move = ((2, 2), (3, 2)), time taken =  2.229623e-04s
White wins! Black King is captured.
```

Test Case	Explanation	Passed	Points
Public 4		Passed	0

Figure 3: Example CodePost Output

You may submit your files as many times as you like. Note that this platform is run by an external organisation – we are not responsible for any downtime.

3.4 IMPORTANT: Grading

In this project, CodePost will NOT show whether you pass or fail a test case. Instead, the outputs on CodePost will show the following:

- The starting position of the game.
- A sequence of moves made by your agent and an opponent agent, starting from the given starting position. This sequence of moves ends when a checkmate or draw occurs, or when the total running time of the chess game exceeds 25 seconds. Note that this termination after 25 seconds is due to a limitation of CodePost, and does not indicate whether your agent has won or lost.
- The time taken for your agent to make each move.

You are encouraged to use the output on CodePost to reconstruct the game played between your agent and the opponent agent. Unexpected behaviour of your agent can indicate a flaw in your algorithm implementation, or a suboptimal heuristic function.

3.5 Submission Details via Canvas

- For this project, you will need to submit 1 zip file containing 1 python file: `AB.py`

- Place the file in a folder, name the folder as `studentNo` and zip it as `studentNo.zip`. An example will be: `A0123456X.zip`.
- The format of submission is the same as the previous projects.
- In summary, your submission file should be a zip file and when unzipped, it will contain the 1 file in a folder: **A0123456Z.zip** $\rightarrow$ **unzip** $\rightarrow$ **A0123456Z folder** $\rightarrow$ **AB.py file**. There should not be any subfolders.
- Do not modify the file names.
- Make only one submission on Canvas.
- Please follow the instructions closely. If your files cannot be opened or if the autograder cannot execute your code, it will be considered failing the test cases as your code cannot be tested.
- Submission folder: **Canvas** > **CS3243** > **Assignments** > **Project 3 Submission**

4 Other details

4.1 Allowed Libraries:

To aid in your implementation, you are allowed to use these Python libraries:

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

4.2 Hints:

- When submitting on CodePost, you should remove any print() functions, raising of exceptions and std.out() functions. This may interfere with the displaying of the moves made.
- The board evaluation function and move ordering heuristic need not be very complicated (the smart agent and minimax agent uses a combination of material difference and number of moves available as a heuristic function). At the end of the day, this is a programming exercise, not a chess exercise. You may use the P2ST (ChatGPT) App to help you determine what features to implement in your heuristics.

- **You are highly encouraged to write your own agent functions, following the description in 3.2.1. This is as due to time limitations on CodePost, the outputs on CodePost likely cannot show the full game, especially for the minimax agents. Hence, writing your own agent functions can give you a much better idea of the long term behaviour of your code.**