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

**Project 3: Adversarial Search**

Issued: 12 October 2022 Due: 6 November 2022

## Overview

In this project, you will implement an adversarial search algorithm to find valid and good moves for a minichess game. Specifically, you are tasked to:

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

---

**This project is worth 10% of your module grade.**

---

## General Project Requirements

The general project requirements are as follows:

- **Individual** project
- Python Version: **3.7**
- Submission deadline: **6 November 2022 (Wednesday) 11:59pm**
- Submission folder: **LumiNUS** > **CS3243** > **Projects** > **Project 3 Submission folder**
- Submission format: One standard (non-encrypted) **zip file**[1] 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. **You are strongly encouraged to reuse your implementation from Project 1/2.**

---

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

## 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 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%.

---

## Background: Gardner's Minichess

A board needs to be five squares wide to contain all kinds of chess pieces (except the pawn) on the first row. In 1969, Martin Gardner[2] suggested a chess variant on a 5×5 board[3]. However, in this project, we will be playing **a game of minichess on a customised 7x7 chess board that includes the fairy pieces** from previous projects.
You will implement the Alpha-Beta Pruning algorithm to beat various AI agents at the game of minichess. That is, to perform a checkmate on the opposing player's King. The rules of the game are defined in the next section.

## Rules of the Minichess Game

### Board Set-up

In this game, the chess pieces are divided into two different colored sets, namely White and Black. Each set consists of 14 pieces: one King, one Queen, one Rook, one Bishop, one Knight, one Empress, one Princess, two Ferz and five Pawns.
The game is played on a square board of 7 rows and 7 columns. The rows start from row 0 at the top to row 6 at the bottom and the columns start from column 'a' as the leftmost column to column 'e' as the rightmost column. The White pieces are placed at rows 0 and 1 while the Black pieces are placed at rows 5 and 6. The initial configuration and position of the pieces on the board can be seen below:

[2] https://en.wikipedia.org/wiki/Martin_Gardner
[3] https://glukkazan.github.io/checkmate/gardner-chess.htm

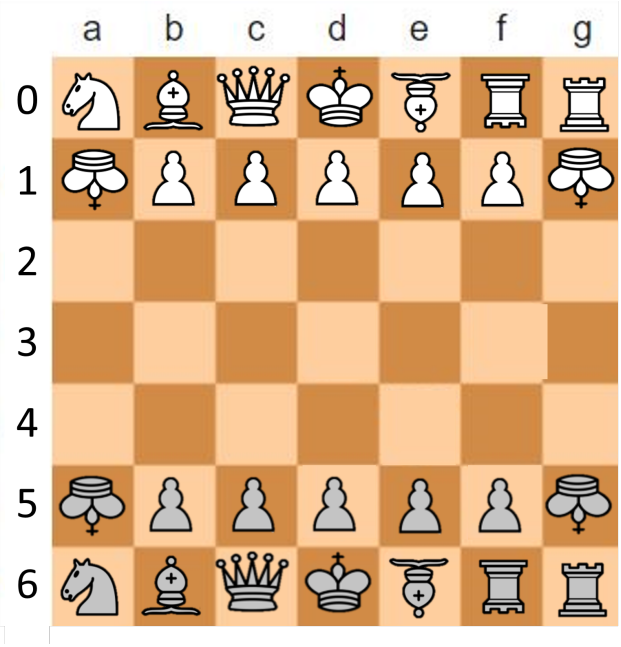


Figure 1: The starting configuration of a 7x7 minichess board with rows and columns labelled.

The starting positions of the chess pieces are shown below, where the first tuple represents the position the specific piece is at, while the second tuple represents the piece type and the piece colour:

```
('a', 1): ('Ferz', 'White'), ('a', 5): ('Ferz', 'Black'), ('g', 1):
('Ferz', 'White'), ('g', 5): ('Ferz', 'Black'), ('b', 1): ('Pawn',
'White'), ('b', 5): ('Pawn', 'Black'), ('c', 1): ('Pawn', 'White'),
('c', 5): ('Pawn', 'Black'), ('d', 1): ('Pawn', 'White'), ('d', 5):
('Pawn', 'Black'), ('e', 1): ('Pawn', 'White'), ('e', 5): ('Pawn',
'Black'), ('f', 1): ('Pawn', 'White'), ('f', 5): ('Pawn', 'Black'),
('a', 0): ('Knight', 'White'), ('a', 6): ('Knight', 'Black'), ('b',
0): ('Bishop', 'White'), ('b', 6): ('Bishop', 'Black'), ('c', 0):
('Queen', 'White'), ('c', 6): ('Queen', 'Black'), ('d', 0): ('King',
'White'), ('d', 6): ('King', 'Black'), ('e', 0): ('Princess', 'White'),
('e', 6): ('Princess', 'Black'), ('f', 0): ('Empress', 'White'),
('f', 6): ('Empress', 'Black'), ('g', 0): ('Rook', 'White'), ('g',
6): ('Rook', 'Black')
```

### Movement of Chess Pieces

**As the chess pieces (except the Pawn) are the same as Project 1, you may refer to the Project 1 specifications PDF for the movement rules of each piece.** Note that there are no obstacles in

this game in Project 3.

New piece in Project 3: **Pawn**

- A Pawn can move forward to the unoccupied square immediately in front of it on the same column (black circle in the picture). Additionally, a Pawn can capture an opponent's piece on a square diagonally in front of it by moving to that square (X crosses in the picture below).
- **Note that we do not consider special moves for Pawns in this game** (i.e., no en passant capture, no promotion, and no advancing of two squares along the same column on its first move).

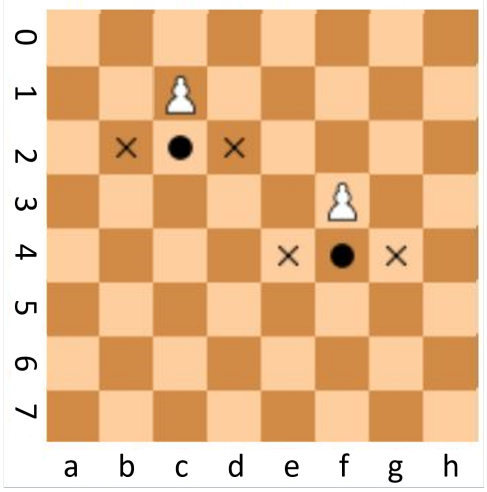


Figure 2: Movement of Pawn.

- In the game in Project 3, the White Pawn can only move in the downwards direction (from row 0 to row 6) while the black pawn can only move in the upwards direction (from row 6 to row 0).

Additionally, in Project 3, we only consider the basic movements for each piece and we do not consider any special moves for any of the pieces. For example, there is no castling for Rook and no special moves for Pawns as mentioned above.

### Check

**When a King is under immediate attack (threatened by an opponent piece), it is said to be in check.** In the standard game of chess, a move in response to a check is legal only if it results in a

position where the King is no longer in check (i.e., the player cannot make any other move unless its King is taken out of the check). This can involve capturing the checking piece; interposing a piece between the checking piece and the King (which is possible only if the checking piece is a Queen, Rook, or Bishop and there is a square between it and the King); or moving the King to a square where it is not under attack.
In the standard game of chess, it is also illegal for a player to put its own King in check. **However, in this project, this rule is not enforced;** i.e., you may play against some novice agents that may make unintelligent moves that will place its own King in check, allowing you to capture the agent's King in the next turn. However, if you accidentally place your own King in check, a smarter agent may also capture your King to win. (referred to as "King Capture" in the next section)

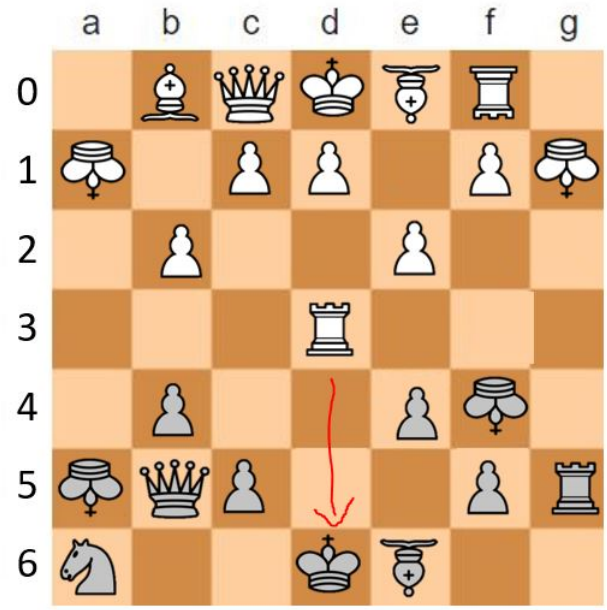


Figure 3: Black King at d6 is checked by White Rook at d3.

### Checkmate (Winning Conditions)

The objective of the game is to checkmate the opponent; in this variant, there are multiple ways to checkmate an opponent.

1. **Standard Checkmate:** This occurs when the opponent's King is in check, and there is no

legal way to get it out of check. Since it is illegal for a player to make a move that puts or leaves its own King in check, if it is not possible to get its King out of check, then the player cannot make any other moves and the King is considered checkmated (and the game is over).

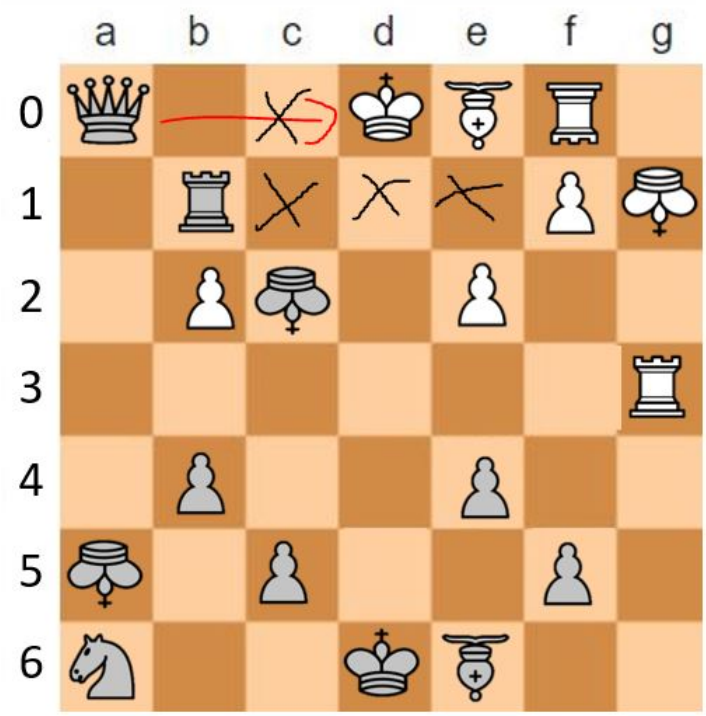


Figure 4: White King at d0 checkmated by Black Queen at a0. The 'X's marks the available positions that the King can take. However, since any of these positions are threatened by both the Black Queen at a0 and the black Rook at b1, the White King cannot escape check, resulting in checkmate.

2. **Out of Valid Moves:** This occurs when the opponent cannot make any valid moves during his turn, as any move made by the opponent will cause its **King to be placed in check,** resulting in a self-checkmate in the next turn as we can capture his King piece.

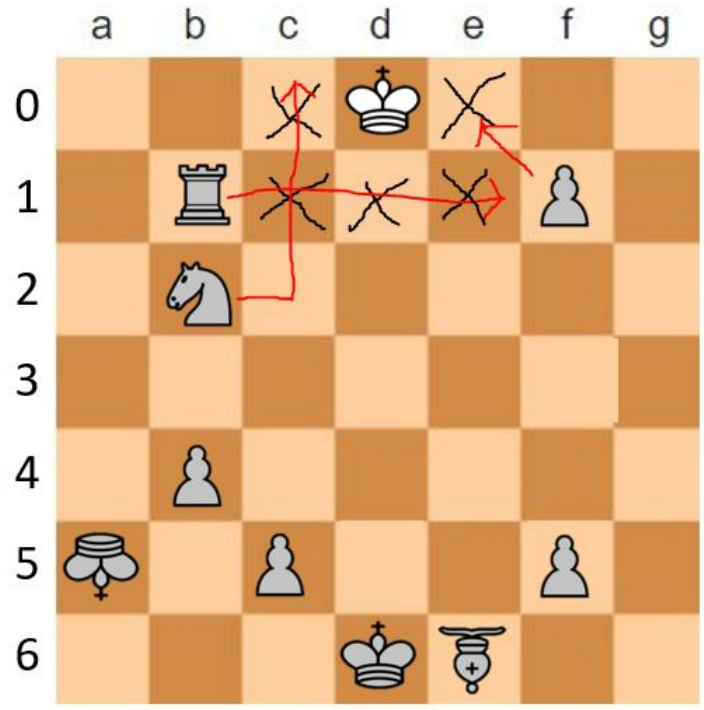


Figure 5: Although White King at d0 is not checked now, White cannot make any more valid moves as moving his King at d0 (the only White piece left) will cause his King to be checked by the Black pieces (c0 threatened by Black Knight at b2; c1, d1 and e1 threatened by Black Rook at b1; e0 threatened by Black Pawn at f1).

3. **King Capture:** This occurs when the opponent makes a move that will cause his King to be in check, or when his King is in check but he ignores it and makes a move that will cause his King to remain in check.

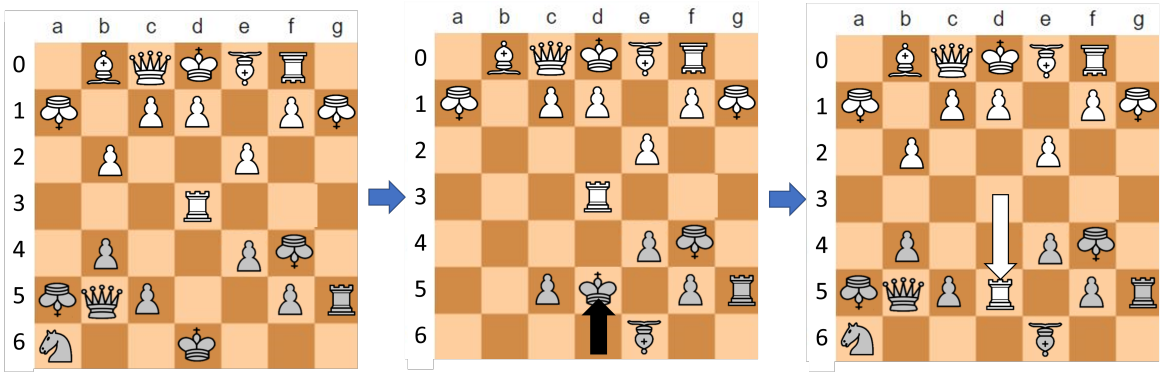


Figure 6: Black King at d6 is checked by White Rook at d3. It is now Black's turn to move (1/3). However, Black ignores the checking White Rook and moves its King from d6 to d5, causing his King to remain in check by White Rook at d3 (2/3). White Rook at d3 proceeds to capture Black King at d5, leading to White winning the game (3/3).

### Draw

In this game, we only consider the game to be a draw **if White and Black have the same number of pieces left after 50 consecutive moves.** This means that there is no change in the number of pieces in the board for 50 moves in a row, implying a Draw. Furthermore, this can only occur if both Kings are still left on the board.

Additionally, we consider a game a draw **if there are no more available moves for a player.** For this scenario to happen, the player's pieces must not have any free moves it can make, regardless of whether a move will place its King in check.

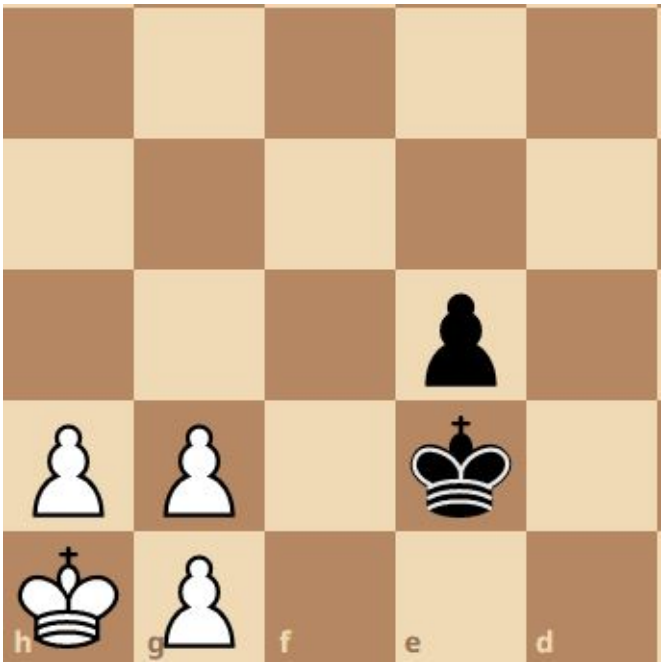


Figure 7: White cannot make any more moves as the King is trapped by 3 of its pawn and all of the pawns are not able to move as well. This results in a draw.)

---

## 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 `config.txt` file 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.

---

## Winning a minichess game using Alpha-Beta Pruning

The objectives of this part of the 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.

---

## 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 7x7 minichess 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 minichess player in the kingdom, you decide to join the competition to challenge yourself!

**Background:**

You are given the initial configuration of the minichess board. You are also assigned to be Player WHITE (White chess pieces) and you are given the privilege of making the first move. You will play against different AI agents and they are assigned as Player BLACK. The winning conditions against the different AI agents are described below:

- **Dummy Agent (Testcase 1):** This agent chooses the first available move that it sees and performs the action. It does not care if the move will cause its King to be checked.
- **Random Agent (Testcase 2):** This agent chooses a random move out of all the available moves that it sees and performs the action. It does not care if the move will cause its King to be checked.
- **Greedy Agent (Testcase 3):** This agent will choose a move based on a priority. Firstly, it will choose a move that will checkmate your King if there exists such a move. If there is no such move, it will then find and choose a move that will check your King. If there is no such move, it will choose any available move at random that does not cause its own King to be checked. If no such move exist, the Greedy Agent have lost by the condition of **Out of Valid Moves**.
- **Smart Agent (Testcase 4):** This agent will choose a move based on a utility value. It will pick the move that gives the highest utility. In general, a move that checkmates its opponent yields the highest utililty, followed by moves that checks its opponent's King AND capture another piece simultaneously, followed by moves that capture another piece ONLY and lastly, moves that checks its opponent's King ONLY. Capturing different types of piece gives different utility value as well.[1] Additionally, If there are no valid moves that will not cause the Agent's King to be checked, the Smart Agent have lost by the condition of **Out of Valid Moves**.
- **Minimax Agent (Testcase 5):** This agent will choose a move based on the Minimax algorithm with alpha beta pruning with a depth of 4. Its evaluation function is similar to the utility of the Smart Agent (Limited depth to improve runtime).

---

## Task 1: Alpha-Beta Pruning Algorithm

Implement the **Alpha-Beta pruning** algorithm in `AB.py` to solve the problem stated above.

The function `studentAgent()` takes in a parameter `gameboard` that is a dictionary of positions (Key) to the tuple of piece type and its colour (Value). It represents the current pieces left on

[1] https://www.chessprogramming.org/Material

the board. At your first turn, the `gameboard` will be a dictionary containing all the pieces on the board as shown in page 3. Another example of the `gameboard`:

$$\{('a', 0) : ('Queen', 'White'), ('d', 10) : ('Knight', 'Black'), ('g', 25) : ('Rook', 'White')\}$$

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

$$(pos_1, \ pos_2)$$

The values $pos_1$ and $pos_2$ represent a grid index tuple, $(x, y)$, where $x$ is the column index (i.e., a string), and $y$ is the row index (i.e., an integer), such that $(x, y)$ corresponds to a specific grid cell on the board that we wish to reference. **The move specifies that you wish to move the piece at $pos_1$ to $pos_2$.**
An example of the function output is shown below:

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

Sample output (that represents moving your White Queen at a0 to b1):

$$(('a', 0), \ ('b', 1))$$

**Configuration file input: Use sys.argv[1] to obtain the name of the configuration file and use `setUpBoard()` to configure the start state of the gameboard.** (Note that it is optional to read this config file as it is fixed at all times. This means that you can hardcode the initial starting positions and the board size of the game.)

---

## Grading Rubrics (Total: 10 marks)

| Requirements (Marks Allocated) | Total Marks |
|---|---|
| • Correct implementation of Minimax algorithm and Alpha-Beta pruning evaluated by **winning or drawing** against Dummy Agent consistently **(100% of the time)** for full credit (1m).<br>• Correct implementation of Minimax algorithm and Alpha-Beta pruning evaluated by **winning or drawing** against Random Agent consistently **(100% of the time)** for full credit (1m).<br>• Correct implementation of Minimax algorithm and Alpha-Beta pruning evaluated by **winning or drawing** against Greedy Agent **(90% of the time)** for full credit (2m).<br>• Correct implementation of Minimax algorithm and Alpha-Beta pruning evaluated by **winning or drawing** against Smart Agent **(90% of the time)** for full credit (3m).<br>• Correct implementation of Minimax algorithm and Alpha-Beta pruning evaluated by **winning or drawing** against Minimax Agent **(80% of the time)** for full credit (3m). | 10 |

## CodePost (Platform for Running Autograder)

We will be using CodePost as our platform to run the autograder and test your code. You should have already joined the Course's group on CodePost from Project 1. 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/student/CS3243%20AY2223%20S1/Fall%202022/

1. 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. If your implementation is within CodePost's time limit of 30s, you will see this:

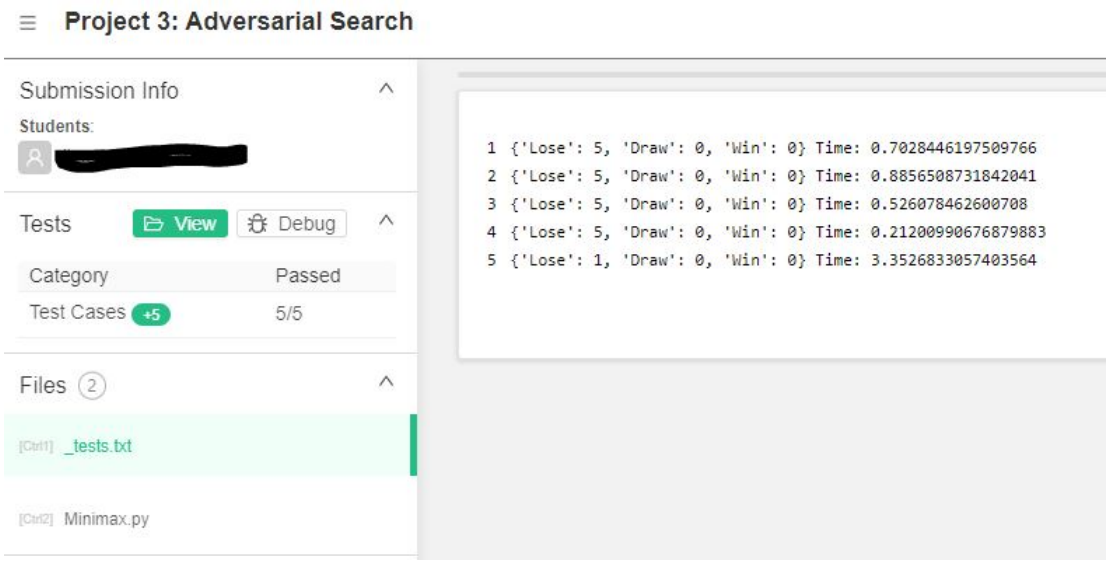


Figure 8: CodePost output for Testcase 1 - 5 in a sequential order. **The score shown on the left of the picture (Passed (5/5)) can be disregarded as we will look at the total wins/draws instead.**

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.

## IMPORTANT: Grading

**For this project, results/scoring on CodePost will not be counted as your actual grade. CodePost serves as a platform for you to test your code and check if your agent is winning against our agents.** Due to CodePost's timeout limitation of 30s, it is impossible to conduct full testing on it as 1 iteration of the game using two agents using Alpha-Beta Pruning takes around 15-25s. As such, we are unable to execute the full test on CodePost as we are unable to perform batch testing. For agents 1-5, we have provided 1 iterations of gameplay for you to test. You can choose to re-submit your code as many times on CodePost to see your agent's expected winnings (Around 10 - 20 times with consistent results should suffice). Additionally, to reduce the chances of operation timeout, you can try limiting your cut-off depth for AB-pruning to $< 5$.

Your code will be graded for technical correctness. Please do not change the name of the function `studentAgent`.

Testing of your code will be done locally. For each test case, the result of your agent will be recorded and the time taken for the execution of your code will be measured as well. **To ensure that we can measure your agent's performance fairly, your agent will play against each different agent for 50 iterations.** Your agent should fulfil the winning conditions against the different agents to receive full credit. Additionally, the time taken for your Alpha-Beta Pruning algorithm will be compared to our solution's benchmark. If the time taken for your code is above the time limit, it will be considered a failure of the test case. As such, you are encouraged to use efficient data structures when implementing your code (e.g., using a set/dictionary instead of a list). To account for the execution of code on different systems, we have provided additional buffers for the time limit.

There is one configuration file released to you via LumiNUS (together with the skeleton implementation file(s)). These files serve as a starting point for you to initialise the gameboard and starting pieces. **For this project, there will not be any public test cases since you will be playing against another program. Nonetheless, we have provided sufficient information of the agents that you will be playing against as shown above.** To aid in your own debugging and testing, you may wish to create different agents as mentioned above and use them to play against your own Minimax Agent. This will be extremely helpful as it removes the limitations of CodePost. You can refer to this link when designing your evaluation function: https://www.chessprogramming.org/Evaluation#Basic_Evaluation_Features

In summary, to pass a test case, two conditions have to be fulfilled for your Alpha-Beta Pruning algorithm implementation. They are:

1. Fulfilling the winning conditions for the different AI agents as shown above.
2. Time taken to run your algorithm is within the time limit.

---

## Other details

### Allowed Libraries:

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

- CLI arguments: sys
- Data structures: queue, collections, heapq, array, copy, enum, string
- Math: numbers, math, decimal, fractions, random

- Functional: itertools, functools, operators
- Types: types, typing

**Test Cases Debugging:**

**When submitting on CodePost, you should remove any print() functions, raising of exceptions and std.out() functions. You should also rename any functions/variables/string that contain those key words in your code to prevent the autograder from terminating. Commenting out the functions may not guarantee that the autograder will pass the check.**

---

## Submission Details via LumiNUS

- 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 Project 1 and Project 2.
- 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 LumiNUS.
- 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: **LumiNUS** $>$ **CS3243** $>$ **Projects** $>$ **Project 3 Submission Folder**