

**ICS 381: Principles of AI – 242**  
**PA 2 – Programming Instructions**

---

**General Helpful Tips:**

- Do not copy others' work. You can discuss general approaches with students, but do not share specific coding solutions.
- **NOTE1:** For this homework, we will be using **itertools** so import it.
- **NOTE2:** For this homework, we will be using **copy** so import it as well.
- **NOTE3:** For this homework, we will be using **numpy** and **chess** packages.
- **NOTE4:** Be sure to implement the functions as specified in this document. In addition, you can implement any extra helper functions as you see fit with whatever names you like.
- Submit the required files only: [**csp\_scheduler.py**, **games.py**]

Autograder Note: the autograder on gradescope will take around **100 seconds** to complete.

# Implementing Games

## python-chess package

Open the command line. Make sure to activate the `ai_env.yml` that we've been using for our assignments.

```
conda activate ai_env
```

You should have the [python-chess](#) package. We will use this package for simulating chess games. Check that package is installed by opening a python terminal and try importing chess.

## **import chess**

Also go over the [introduction](#) for this package. It is intuitive to understand how the engine works. You can also skim over the API for the [core](#) functions; you can see there are identifiers for the white and black player, chess pieces, the chess grid locations, Move functions, and Board functions. Of importance are the following Board functions:

- **legal\_moves:** note this is a property of the Board class. It generates list of legal moves for the current board position. Note that the board object keeps track of the current player, so the legal moves are generated for the current player.
- **turn:** note this is a property of the Board class. It returns the current player for this turn; returns True for White's turn and False for Black's turn.
- **outcome():** checks if the game is over from various reasons. If the game is not over, this will return None. If the game is over it will return an [Outcome](#) object. This will allow us to know if we are in a terminal state and know who is the winner.
- **is\_stalemate():** returns True/False if position is draw/stalemate. This will allow us to know if are in a terminal state.
- **push(move: [Move](#)):** Updates the board with the given move and puts it onto the move stack.
- **pop():** undoes the last move made. This is helpful if we want to backtrack when searching the game-tree.
- **pieces(piece\_type: *chess.PieceType*, color: *chess.Color*):** Gets pieces of the given type and color. This is helpful for us to get the current board position for both players (White and Black)

See the example code in the [introduction](#) to get some intuition on how to use these functions to simulate a chess game. Note that the game proceeds in turns with the White player starting first. So when you progress a turn by calling `board.move`, it is the next player's turn when you call move again.

For this assignment, we will implement minimax algorithm with heuristics and alpha-beta pruning to play chess. The MAX player will be the White player, with MIN being the Black opponent. For **terminal** states (checkmate) we will assign the utilities of: +1 when White wins, -1 when White loses, 0 when it is a stalemate/draw. The following code should give you an idea on how to check if the game has ended and who is the winner:

```
outcome = board.outcome()

if outcome is None: # if outcome is None the game is still going; i.e. non-terminal state
    print('game has not ended yet')
else: # terminal states
    winner = outcome.winner # True for WHITE player and False for BLACK player
    termination = outcome.termination

    if termination == chess.Termination.CHECKMATE:
        if winner:
            print('White player has won')
        else:
            print('Black player has won')

    else: # for other outcomes, the game is drawn
        print('the game is draw')
```

Recall from class that it would take too long to search the full game-tree for chess. So we will make use of heuristic to compute the estimated utility for **non-terminal** nodes. For this assignment we will start by implementing a heuristic for chess, then implement heuristic-minimax, then implement heuristic-minimax with alpha-beta pruning.

### game.py: Implement heuristic function for non-terminal nodes

Function Name: `heuristic_chess(board)`

Arguments: `board` is the current Board object. We can use this to get current board state information like pieces for both players and their positions.

Returns: `heuristic_value` this is the estimated utility of the current board state for WHITE player perspective (MAX).

#### **Implementation:**

If the current board state is a terminal state (use `board.outcome`), then return +1, -1, or 0 depending on the outcome and winner.

If the current board state is a non-terminal state, then compute the following features of the board state:

- $f_{\text{pawn}} = (\text{num White pawns} - \text{num Black pawns})$
- $f_{\text{knight}} = (\text{num White knights} - \text{num Black knights})$
- $f_{\text{bishop}} = (\text{num White bishops} - \text{num Black bishops})$
- $f_{\text{rook}} = (\text{num White rooks} - \text{num Black rooks})$
- $f_{\text{queen}} = (\text{num White queens} - \text{num Black queens})$

Note that you can gather pieces info by making use of the [board.pieces](#) function. For example, to get number of black pawn pieces, you can use: `len( board.pieces(chess.PAWN, chess.BLACK) )`

Now compute the heuristic value of the board as:

$$\text{heuristic\_value} = \frac{1.25f_{\text{pawn}} + 3.5f_{\text{knight}} + 5f_{\text{bishop}} + 5f_{\text{rook}} + 10f_{\text{queen}}}{100}$$

We divide by 100 to scale it in the range between -1 and +1.

### **game.py: Implement is\_cutoff as helper for minimax**

Function Name: `is_cutoff(board, current_depth, depth_limit=2)`

Arguments: `board` is the current Board object. We can use this to get current board state information like pieces for both players and their positions.

`current_depth` is the current depth of the game-tree. We want to keep track of this to know when we should cut the search.

`depth_limit` This is the cutoff depth to stop searching and compute heuristics.

Returns: True or False

This is a simple function that will be used as a helper for minimax implementation. Given the arguments, do the following:

- If the current board state is a terminal state (use the check `board.outcome` is not None), then return True.
- If the `current_depth` is equal to `depth_limit`, return True.
- Else return False.

## game.py: Implement heuristic minimax algorithm

Function Name: `h_minimax(board, depth_limit=2)`

Arguments: `board` is the current Board object. We can use this to get current board state information like pieces for both players and their positions.

`depth_limit` This is the cutoff depth to stop searching and compute heuristics.

Returns: `heuristic_value, move` returns the heuristic value and move that should be made by MAX (White) player.

Note this will be a [`chess.Move`](#) object.

### Implementation:

Follow the pseudocode in slide 46 of Adversarial Search and Game slides. Implement the two helper functions:

- `max_node(board, current_depth, depth_limit)` to implement MAX node behavior.
- `min_node(board, current_depth, depth_limit)` to implement MIN node behavior.

Some notes for both helper functions:

- You can import numpy and use `np.inf` and `-np.inf` to represent  $+\infty$  and  $-\infty$ .
- You can loop on available actions by using `board.legal_moves`.
- `Current_depth` should start at 0 for the first MAX node at the root, then it should be incremented by 1 for each subsequent `max_node` or `min_node` call (as you move to a lower depth in the tree, increment by 1).
- Be sure to use `is_cutoff` function to stop the search, and use `heuristic_value` function to compute the estimated utility. For the move, just return `None` (null) like in the pseudocode in slide 46.
- **Important:** While looping on actions, to progress the board, you can use `board.push(move)`. Then you can call the MIN or MAX node call with the progress board. However, you definitely want to undo the move once a search call is done by calling `board.pop()`.

### **game.py: Implement heuristic minimax algorithm with alpha-beta pruning**

Function Name: `h_minimax_alpha_beta(board, depth_limit=2)`

Arguments: `board` is the current Board object. We can use this to get current board state information like pieces for both players and their positions.

`depth_limit` This is the cutoff depth to stop searching and compute heuristics.

Returns: `heuristic_value, move` returns the heuristic value and move that should be made by MAX (White) player.

Note this will be a [`chess.Move`](#) object.

#### **Implementation:**

Implement the two helper functions:

- `max_node_ab(board, current_depth, depth_limit, alpha, beta)` to implement MAX node behavior.
- `min_node_ab(board, current_depth, depth_limit, alpha, beta)` to implement MIN node behavior.

**NOTE:** I highly recommend copying your code from the previous task. You just need to augment the code with the alpha-beta parameters which requires few modifications to arguments and few additional lines of code. See the pseudocode in slide 47 of Adversarial Search and Game slides.



## **Test your code**

You can test your code by running `test_h_minimax.py`, `test_alpha_beta.py`, and `test_profiling.py`

The results of my implementation for these test files can be found in the `refouput*.txt` files.

Of interest is the results from `test_profiling.py`. This test code will run the python profiler to count the number of calls for min and max nodes. For a correct implementation, you should see that alpha beta pruning significantly reduces the number of calls (which reduces the search).