Computer Science 384 March 5, 2025
St. George Campus University of Toronto

Homework Assignment #3: Game Tree Search
**Due: March 21, 2025 by 11:00 PM**

---

**Late Policy**: Please refer to the Course Information sheet posted on Quercus. Please note that grace tokens will be applied **automatically** as long as you submit your assignment during the grace token period.

**Total Marks**: This assignment represents 10% of the course grade.

**Handing in this Assignment**

*What to hand in on paper:* Nothing.

*What to hand in electronically:* You must submit your assignment electronically. Download `A3.zip` which contains `agent.py` other related files from the A3 web page. Modify `agent.py` so that it solves the problems specified in this document. **Submit your modified files:** `agents.py` You will submit your assignment using MarkUs. It is your responsibility to include all necessary files in your submission. You can submit a new version of any file at any time, though your grace tokens will be used if you submit after the deadline.

We will test your code electronically. You will be supplied with a testing script that contains a test using sample boards. You can add your own tests to this script. It's up to you to figure out further test cases to further test your code – that's part of the assignment! You will also have access the same tests on MarkUs so that you can test to ensure that your code executes correctly on the MarkUs instance.

When your code is submitted, we will run a more extensive set of tests. You have to pass all of these more elaborate tests to obtain full marks on the assignment.

Your code will not be evaluated for partial correctness, it either works or it doesn't. It is your responsibility to hand in something that passes at least some of the hidden tests on MarkUs.

- *Make certain that your code runs on teach.cs using python3 (version 3.12) using only standard imports.* This version is installed as "python3" on teach.cs. Your code will be tested using this version and you will receive zero marks if it does not run using this version.
- *Do not add any non-standard imports from within the python file you submit (the imports that are already in the template files must remain).* Once again, non-standard imports will cause your code to fail the testing and you will receive zero marks.
- *Do not change the supplied starter code.* Your code will be tested using the original starter code, and if it relies on changes you made to the starter code, you will receive zero marks.

**Clarifications**: Important corrections (hopefully few or none) and clarifications to the assignment will be announced on Quercus.
You are responsible for monitoring Quercus announcements for any clarifications or corrections.

**Help Sessions**: There will be several help sessions for this assignment. Dates and times for these sessions will be posted on Quercus.

**Questions**: Questions about the assignment should be posed on Piazza.

# Introduction

**Acknowledgments:** This project is based on one used in Columbia University's Artificial Intelligence Course (COMS W4701). *Special thanks to Dr. Daniel Bauer, who developed the starter code that we've extended.*

**Othello** is a 2-player board game that is played with distinct pieces that are typically dark on one side and light on the other side, each side belonging to one player. Our version of the game is played on a chess board of any size, but the typical game is played on an 8x8 board. Players (dark and light) take turns placing their pieces on the board.

Placement is dictated by the rules of the game and can result in the flipping of colored pieces from light to dark or dark to light. The rules of the game are explained in detail at `https://en.wikipedia.org/wiki/Reversi`.

**Objective:** The player's goal is to ensure that, by the end of the game, the majority of the pieces displayed on the board are of their color.

**Game Ending:** Our version of the game differs from the standard rules described on Wikipedia in one minor point: The game ends as soon as one of the players has no legal moves left.

**Rules:** The game begins with four pieces placed in a square in the middle of the grid, two light pieces and two dark pieces (see Figure 1, at left). The dark player always makes the first move.

- A **move** is defined as a player placing a piece on an unoccupied square on the board during their turn.
- A move is considered **legal ONLY** if the move "brackets" one or more opponent pieces in a straight line along at least one axis (vertical, horizontal, or diagonal).

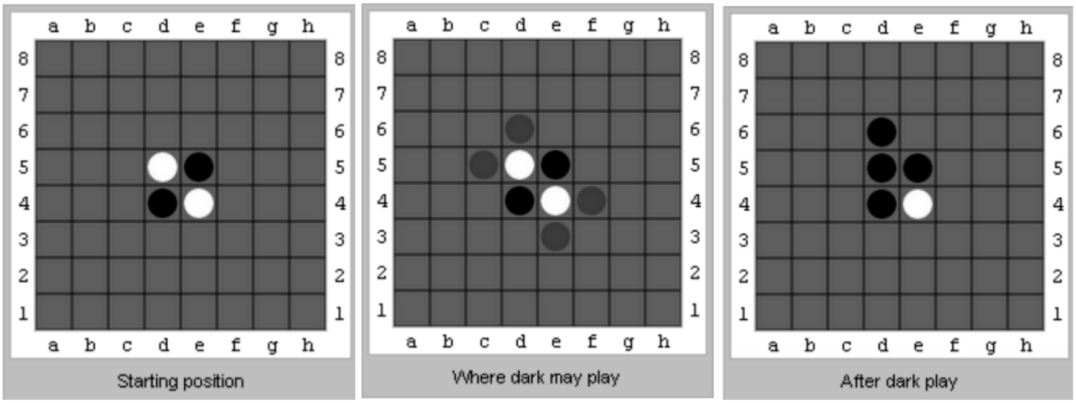


Figure 1: On the left, the initial state. In the middle, possible moves of dark player are shown in grey. At right, the board state after the dark player has moved.

For example, from the initial state the dark player can achieve this bracketing by placing a dark piece in any of the positions indicated by grey pieces in 1 (in the middle). Each of these potential placements would create a Dark-Light-Dark sequence, thus "bracketing" the Light piece. Once the piece is placed, all opponent pieces that got bracketed by the move are flipped to become the same color as the current player's.

Returning to our example, if the dark player places a piece in Position 6-d in the middle panel of Figure 1,

the light piece in position 5-d will become bracketed and consequently flipped to dark, resulting in the board depicted in the right panel of Figure 1.

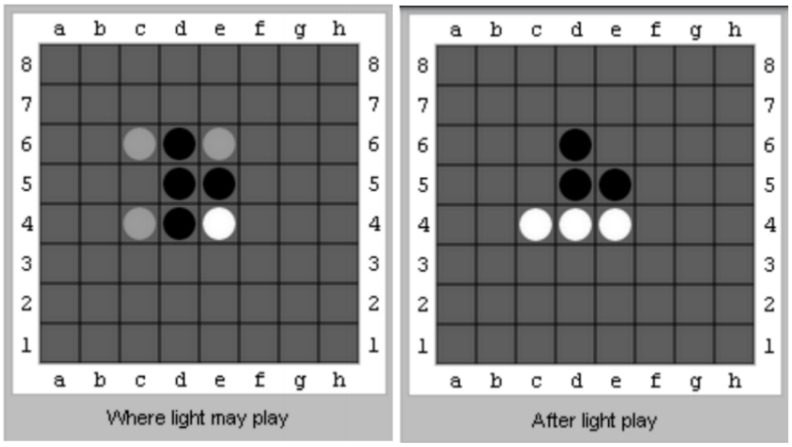


Figure 2: On the left, possible moves of the light player are shown in grey. At right, the board state after the move of the light player is shown.

Now it's the light player's turn to play. All of the light player's possibilities at this time are shown as grey pieces in 2 (at left). If the light player places a piece on 4-c, it will cause the dark piece in 4-d to be bracketed, resulting in the 4-d piece being flipped to light, as shown in 2(at right). To summarize, a legal move for a player is one that results in at least one of its opponent's pieces being flipped. Our version of the game ends when one player no longer has any legal moves available.

# 1 Starter Code

**The starter code contains 5 files:**

1. `othello_gui.py`, which contains a simple graphical user interface (GUI) for Othello.
2. `othello_game.py`, which contains the game "manager". This stores the current game state and communicates with different player AIs.
3. `othello_shared.py`, which contains functions for computing legal moves, captured disks, and successor game states. These are shared between the game manager, the GUI and the AI players.
4. `randy_ai.py`, which specifies an "AI" player (named Randy) that randomly selects a legal move.
5. `agent.py` - The file where you will implement your game agent.

**Game State Representation:** Each game state contains two pieces of information: the current player and the current disks on the board. Throughout our implementation, Player 1 (dark) is represented by the integer 1, and Player 2 (light) by the integer 2.

The board is represented as a tuple of tuples. The inner tuple represents each row of the board. Each entry in the rows is either an empty square (integer 0), a dark disk (integer 1), or a light disk (integer 2). For example, an 8x8 initial state looks like this:

```
(
(0, 0, 0, 0, 0, 0, 0, 0),
(0, 0, 0, 0, 0, 0, 0, 0),
(0, 0, 0, 0, 0, 0, 0, 0),
(0, 0, 0, 2, 1, 0, 0, 0),
(0, 0, 0, 1, 2, 0, 0, 0),
(0, 0, 0, 0, 0, 0, 0, 0),
(0, 0, 0, 0, 0, 0, 0, 0),
(0, 0, 0, 0, 0, 0, 0, 0)
)
```

**Running the code:**

You can run the Othello GUI by typing:

```
python3 othello_gui.py -d board_size -a agent.py
```

where the parameter `board_size` is an integer that determines the dimension of the board upon which you will play, and `agent.py` is the game agent you would like to play against.

If you type:

```
python3 othello_gui.py -d board_size -a randy_ai.py
```

you will play against an agent that selects moves randomly (named Randy). Playing a game should bring up a game window. If you play against a game agent, you and the agent will take turns. We recommend that you play against Randy to develop a better understanding of how the game works and what strategies can give you an advantage. Remember that `othello_gui.py` has been written so that if you want to play against an agent, you should supply the agent file only after `-a` (so that the agent will play as light and you as dark). If you use `-b` to introduce an agent, the game will not detect it and will think that you want to play both sides yourself.

The GUI can also take two AI programs as command line parameters. When two AIs are specified at the command line, you can watch them play against each other. Again, remember that the agent introduced with `-a` will play as the dark player, and the agent introduced with `-b` will play as the light player. To see Randy play against itself, type:

```
python3 othello_gui.py -d board_size -a randy_ai.py -b randy_ai.py
```

You may want to try playing the agents you create against those that are made by your friends. The GUI is rather minimalistic, so you need to close the window and then restart to play a new game.

**Communication between the Game Host and the AI:**

*NOTE:* This is a technical detail that you can skip if you are not interested.

Functions for communicating with the game manager are provided as part of the scaffolding code. Note, however, that the game manager communicates with agents via `stdout`. If you want to print **debugging**

statements, you should print to **stderr** instead. You can do this by using the `eprint` command found in `agent.py`.

The AI and the Game Manager/GUI run in different Python interpreters. The Game Manager/GUI spawns a child process for each AI player. This makes it easier for the game manager to let the AI process time out and also ensures that if an AI crashes, the game manager can keep running. To communicate with the child process, the game manager uses pipes: it reads from the AI's standard output and writes to its standard input. The two programs follow a precise protocol to communicate:

- The AI sends a string to identify itself (e.g., Randy AI sends the string "Randy." You can come up with a fun name for your AI).
- The game manager sends back "1" or "2," indicating whether the AI plays dark or light.
- The AI then waits for input from the game manager. When it is the AI's turn, the game manager will send two lines: the current score (e.g., "SCORE 2 2") and the game board (a Python tuple converted to a string). The game manager then waits for the AI to respond with a move (e.g., "4 3").
- At the end of the game, the game master sends the final score (e.g., "FINAL 33 31").

**Time Constraints:**

Your AI player is expected to make a move within 10 seconds. If no move is selected within that time, the AI loses the game. This time constraint does not apply to human players. You may change the time constraint by editing `line 32` in `othello_game.py`: `TIMEOUT = 10`. However, during grading and the Othello competition, your AI will be run with a timeout of 10 seconds.

# Mark Breakdown

## 1. Minimax [30 pts]

You will want to test your Minimax implementations on boards that are only 4x4 in size. This restriction makes the game somewhat trivial: it is easy even for human players to think ahead to the end of the game. When both players play optimally, the player who goes second always wins. However, the default Minimax algorithm, without a depth limit, takes too long, even on a 6x6 board.

Write a function `compute_utility(board, color)` that computes the utility of a final game board state (in the format described above). The utility should be calculated as the number of disks of the player's color minus the number of disks of the opponent.

**HINT:** The function `get_score(board)` returns a tuple (number of dark disks, number of light disks).

Then, implement the method `select_move_minimax(board, color, limit, caching)`. For now, you can ignore the `limit` and `caching` parameters; we will return to these later. Your function should select the action that leads to the state with the highest minimax value. The parameter `board` is the current board (in the format described above), and `color` is the color of the AI player (using 1 for dark and 2

for light). The return value should be a (column, row) tuple representing the move. Implement minimax recursively by writing two functions: `minimax_max_node(board, color, limit, caching)` and `minimax_min_node(board, color, limit, caching)`. (Again, ignore `limit` and `caching` for now.)

**HINT:** Use the function `get_possible_moves(board, color)` in `othello_shared.py` to get the list of legal moves for the player, and use `play_move(board, color, move)` to compute the successor board state.

**Optional:** There is an alternative approach to implement the minimax algorithm using only one function instead of two. Consider what other modifications might be needed.

Once you are done, you can run your MINIMAX algorithm via the command line using the flag `-m`. For example:

```
python3 othello_gui.py -d 4 -a agent.py -m
```

will let you play against your agent on a 4x4 board using the MINIMAX algorithm. The command

```
python3 othello_gui.py -d 4 -a agent.py
```

will have you play against the same agent using the ALPHA-BETA algorithm, which you will implement next. You can also play your agent against Randy with:

```
python3 othello_gui.py -d 4 -a agent.py -b randy_ai.py
```

## 2. Alpha-Beta Pruning [30 pts]

The simple minimax approach becomes infeasible for boards larger than 4x4. To address this, write the function `select_move_alphabeta(board, color, limit, caching, ordering)` to compute the best move using alpha-beta pruning. The parameters and return values are the same as for minimax. For now, ignore the `limit`, `caching`, and `ordering` parameters. Your alpha-beta implementation should recursively call two helper functions: `alphabeta_min_node(board, color, alpha, beta)` and `alphabeta_max_node(color, alpha, beta)`.

Play with pruning should speed up the AI decisions. Use:

```
python3 othello_gui.py -d 4 -a agent.py
```

to play against your agent using the ALPHA-BETA algorithm on a 4x4 board.

## 3. Depth Limit [10 pts]

To further speed up your agents, implement a depth limit by using the `-l` flag at the command line. For example:

```
python3 othello_gui.py -d 6 -a agent.py -m -l 5
```

calls your agent's MINIMAX routine with a depth limit of 5, and

```
python3 othello_gui.py -d 6 -a agent.py -l 5
```

calls your agent's ALPHA-BETA routine with a depth limit of 5.

Alpha-beta and minimax will pass the `limit` parameter recursively. When the depth limit (i.e., when `limit` equals zero) is reached, use a heuristic (such as the compute utility function) to estimate the value of the non-terminal state.

Experiment with different depth limits on boards larger than 4x4. What is the largest board you can play on without timing out after 10 seconds?

## 4. Caching States [10 pts]

To speed up the AI further, modify your program to respond to the `-c` flag for caching. Create a dictionary (a global variable in your AI file) that maps board states to their minimax value. Modify your minimax and alpha-beta functions to store computed values in this dictionary and check it before re-computing a state. When running:

```
python3 othello_gui.py -d 6 -a agent.py -m -c -l 5
```

the game manager will call your agent's MINIMAX routines with caching enabled. Similarly, omitting `-m` will use the ALPHA-BETA routines with caching. Note that caching may not always improve runtime significantly, but it should when searching deeply.

## 5. Node Ordering Heuristic [10 pts]

Alpha-beta pruning can be more efficient if nodes with higher utility are explored first. In your ALPHA-BETA functions, order the successor states according to the heuristic: nodes for which the number of the AI player's disks minus the number of the opponent's disks is highest should be explored first. (This is essentially the utility function.) Enable node ordering when the `-o` flag is placed on the command line. For example:

```
python3 othello_gui.py -d 6 -a agent.py -o -l 5
```

will have the game manager call your ALPHA-BETA routines with an ordering parameter of 1. To play against your agent on an 8x8 board using state caching, alpha-beta pruning, and node ordering, type:

```
python3 othello_gui.py -d 8 -a agent.py -c -o -l 5
```

## 6. Your Own Heuristic [10 pts]

The previous steps should give you a good AI player, but there is room for improvement. To create a better AI, design your own game heuristic. You can use this heuristic in place of the compute utility function in your alpha-beta routines.

**Some Ideas for Heuristic Functions for the Othello Game:**

1. Consider board locations where pieces are stable (i.e., cannot be flipped).
2. Consider the number of moves available to you versus your opponent.

3. Use different strategies for the opening, mid-game, and end-game.

To grade your heuristic, we put two minimax agents against each other. One uses the compute utility and one uses the compute heuristic function submitted by you as its heuristic function. A total of 10 games are played for depth limits 2 to 6 and both color assignments. Each game the agent with your heuristic wins, you get 1 point and with each tie, you get 0.5 points. Remember that the agents are developed by TAs and we only import the compute heuristic function from your file. Your grade only depends on this function and not anything else in your submission.

In addition, please include a (short) description of your heuristic as a comment at the start of your solution file.

---

**GOOD LUCK!**