Computer Science 384
St. George Campus

Monday, June 22, 2020
University of Toronto

Homework Assignment #3: Game Tree Search
Due: July 21, 2020 by 10:00 PM

Silent Policy: *A silent policy will take effect 24 hours before this assignment is due, i.e. no question about this assignment will be answered, whether it is asked on the discussion board, via email or in person.*

Late Policy: 10% per day after the use of 3 grace days.

Overview: Assignment #3 asks you to implement a game playing agents for the game of Othello (or Riversi).

Total Marks: This assignment has a total of 100 marks and represents 13% of the course grade.

What to hand in on paper: Nothing.

What to submit electronically: You must submit your assignment electronically. Download the assignment files from the A3 web page. Modify `agent.py` appropriately so that it solves the problems specified in this document. You will submit the following files:

- `agent.py`
- `acknowledgment_form.pdf`

In addition, if you would like to see your agent compete against others, submit the following file (this is optional):

- `agent_YOURID.py` (where your teach.cs ID replaces "YOURID")

How to submit: If you submit before you have used all of your grace days, you will submit your assignment using MarkUs. Your login to MarkUs is your teach.cs username and password. 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 the lateness penalty applies if you submit after the deadline. For the purposes of determining the lateness penalty, the submission time is considered to be the time of your latest submission. More detailed instructions for using Markus are available at: http://www.teach.cs.toronto.edu/~csc384h/summer/markus.html.

- *Make certain that your code runs on teach.cs using python3 (version 3.7) using only standard imports.* 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.

Clarification Page: Important corrections (hopefully few or none) and clarifications to the assignment will be posted on the Assignment 3 Clarification page:
http://www.teach.cs.toronto.edu/~csc384h/summer/Assignments/A3/a3_faq.html.
You are responsible for monitoring the Assignment 3 Clarification page.

Questions: Questions about the assignment should be posted to Piazza:
https://piazza.com/utoronto.ca/summer2020/csc384/home.

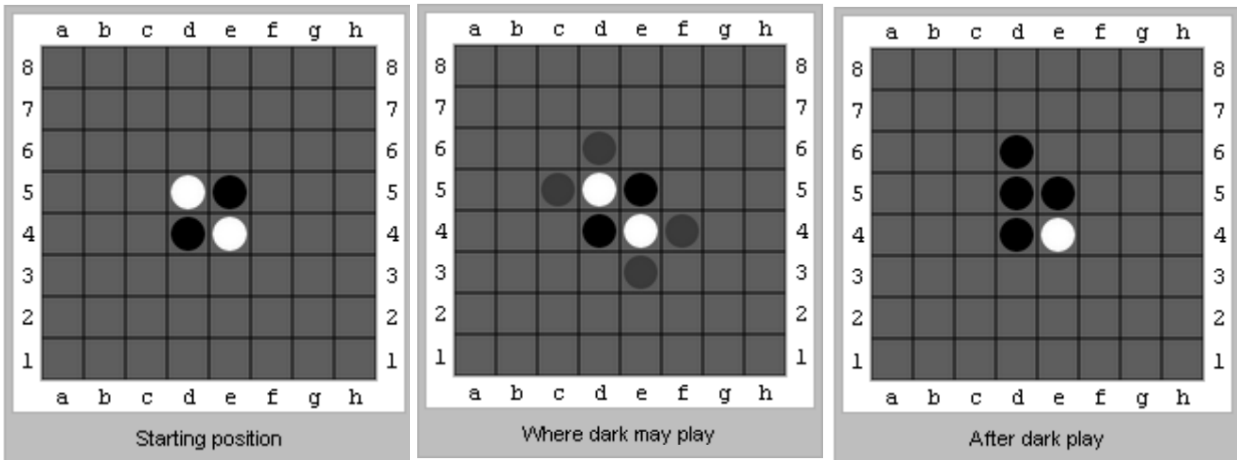

Figure 1: At left, the initial state. In the middle, possible moves of black player are shown in grey. At right, the board state after the black player has moved.

Introduction

Acknowledgements: This project is based on one used in Columbia University's Artificial Intelligence Course (COMS W4701). Special thanks to Daniel Bauer, who originally developed the starter code.

Othello is a 2-player board game that is played with distinct pieces that are typically black on one side and white 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 (black and white) take turns placing their pieces on the board.

Placement is dictated by the rules of the game, and can result in the flipping of coloured pieces from white to black or black to white. The rules of the game are explained in detail at https://en.wikipedia.org/wiki/Reversi.

Objective: The player's goal is to have a majority of their coloured pieces showing at the end of the game.

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 white pieces and two black pieces (Figure 1, at left). Black makes the first move.

At each player's turn, the player may place a piece of their own colour on an unoccupied square, if it "brackets" one or more opponent pieces in a straight line along at least one axis (vertical, horizontal, or diagnonal). For example, from the initial state black can achieve this bracketing by placing a black piece in any of the positions indicated by grey pieces in Figure 1 (in the middle). Each of these potential placements would create a Black-White-Black sequence, thus "bracketing" the White piece. Once the piece is placed, all opponent pieces that are bracketed, along any axis, are flipped to become the same colour as the current player's. Returning to our example, if black places a piece in Position 6-d in the middle panel of Figure 1,

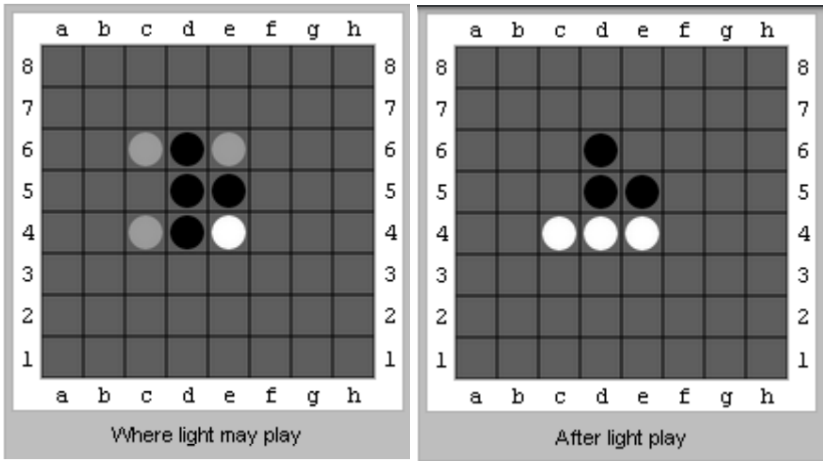

Figure 2: At right, possible moves of white player are shown in grey. At left, the board state after the move of white player.

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

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

Getting Started

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 using the integer 1, and Player 2 (light) is represented using the integer 2.

The board is represented as a 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, and that is 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.

The GUI can take 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. 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:

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 will therefore want to print to stderr instead. You can do this by using the `eprint` command that is found int `agent.py`.

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

- The AI sends a string to identify itself. For example, 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 if the AI plays dark or light.
- Then the AI sits and waits for input form the game manager.
  When it is the AI's turn, the game manager will send two lines: The current score, for example "SCORE 2 2" and the game board (a Python tuple converted to string). The game manager then waits for the AI to respond with a move, for example "4 3".
- At the end of the game, the game master sends the final score, for example "FINAL 33 31".

Time Constraints:

Your AI player will be expected to make a move within 10 seconds. If no move has been selected, the AI loses the game. This time constraint does not apply for human players.

You may change the time constraint by editing line 32 in othello_game.py: TIMEOUT = 10

However, we will run your AI with a timeout of 10 seconds during grading and for the Othello competition.

Part I. 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 the 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)`. For the time being, you can ignore the limit and caching parameters that the function will also accept; we will return to these later. Your function should select the action that leads to the state with the highest minimax value. The 'board' parameter is the current board (in the format described above) and the 'color' parameter is the color of the AI player. We use an integer 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)` and `minimax_min_node(board, color)`. Again, just ignore the limit and caching parameters for now.

Hints: Use the `get_possible_moves(board, color)` function in othello_shared.py, which returns a list of (column, row) tuples representing the legal moves for player color. Use the play_move(board, color, move) function in othello_shared.py, which computes the successor board state that results from player color playing move (a (column, row) tuple). Pay attention to which player should make a move for min nodes and max nodes.

Once you are done you can run your MINIMAX algorithm via the command line using the flag `-m`. If you issue the command "`$python3 othello_gui.py -d 4 -a agent.py -m`" you will play against your

agent on a 4x4 board using the MINIMAX algorithm. The command "`$python3 othello_gui.py -d 4 -a agent.py`" will have you play against the same agent using the ALPHA-BETA algorithm, which you will implement next. You can also play your agent against Randy with the command "`$python3 othello_gui.py -d 4 -a agent.py -b randy_ai.py`"

Part II. Alpha-Beta Pruning [30 pts]

The simple minimax approach becomes infeasible for boards larger than 4x4. To ameliorate this we will write the the function `select_move_alphabeta(board, color)` to compute the best move using alpha-beta pruning. The parameters and return values will be the same as for minimax. Once again, you can again ignore the limit, caching and ordering parameters that the function will also accept for the time being; we will return to these later. Much like minimax, your alpha-beta implementation should recursively call two helper functions: `alphabeta_min_node(board, color, alpha, beta)` and `alphabeta_max_node(board,color, alpha, beta)`.

Playing with pruning should speed up decisions for the AI, but it may not be enough to be able to play on boards larger than 4x4. Use the command "$python3 othello_gui.py -d 4 -a agent.py" to play against your agent using the ALPHA-BETA algorithm on a 4x4 board.

Part III. Depth Limit [10 pts]

Next we'll work on speeding up our agents. One way to do this is by setting a depth limit. Your starter code is structured to do this by using the `-l` flag at the command line. For example, if you type `$python3 othello_gui.py -d 6 -a agent.py -m -l 5`, the game manager will call your agent's MINIMAX routine with a depth limit of 5. If you type `$python3 othello_gui.py -d 6 -a agent.py -l 5`, the game manager will call your agent's ALPHA-BETA routine with a depth limit of 5.

Alpha beta will recursively send the 'limit' parameter to both `alphabeta_min_node` and `alphabeta_max_node`. Minimax is similar and will recursively sends the 'limit' parameter to `minimax_min_node` and `minimax_max_node`. In order to enforce the depth limit in your code, you will want to decrease the limit parameter at each recursion. When you arrive at your depth limit (i.e. when the 'limit' parameter is zero), use a heuristic function to define the value any non-terminal state. You can call the compute_utility function as your heuristic to estimate non-terminal state quality.

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

Part IV. Caching States [10 pts]

We can try to speed up the AI even more by caching states we've seen before. To do this, we will want to alter your program so that it responds to the `-c` flag at the command line. To implement state caching you will need to create a dictionary in your AI player (this can just be stored in a global variable on the top level of the file) that maps board states to their minimax value. Modify your minimax and alpha-beta pruning functions to store states in that dictionary after their minimax value is known. Then check the dictionary,

at the beginning of each function. If a state is already in the dictionary and do not explore it again.

The starter code is structured so that if you type `$python3 othello_gui.py -d 6 -a agent.py -m -c`, the game manager will call your agent's MINIMAX routines with the 'caching' flag on. If instead you remove the `-m` and type `$python3 othello_gui.py -d 6 -a agent.py -c`, the game manager will call your agent's ALPHA-BETA routines with the 'caching' flag on.

Part IV. Node Ordering Heuristic [10 pts]

Alpha-beta pruning works better if nodes that lead to a better utility are explored first. To do this, in the Alpha-beta pruning functions, we will want to order successor states according to the following heuristic: the 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. Note that this is the same as the utility function, and it is okay to call the utility function to compute this value. This should provide another small speed-up.

Alter your program so that it executes node ordering when the `-o` flag is placed on the command line. The starter code is already structured so that if you type `$python3 othello_gui.py -d 6 -a agent.py -o`, the game manager will call your agent's ALPHA-BETA routines with an 'ordering' parameter that is equal to 1.

Taken together, the steps above should give you an AI player that is challenging to play against. To play against your agent on an 8x8 board when it is using state caching, alpha-beta pruning and node ordering, type `$python3 othello_gui.py -d 8 -a agent.py -c -o`.

Part V. Your Own Heuristic [10 pts]

The prior steps should give you a good AI player, but we have only scratched the surface. There are many possible improvements that would create an even better AI. To improve your AI, create your own game heuristic. You can use this in place of `compute_utility` in your alpha-beta routines.

Some Ideas for Heuristic Functions for Othello Game

1. Consider board locations where pieces are stable, i.e. where they cannot be flipped anymore.
2. Consider the number of moves you and your opponent can make given the current board configuration.
3. Use a different strategy in the opening, mid-game, and end-game.

You can also do your own research to find a wide range of other good heuristics (for example, here is a good start: http://www.radagast.se/othello/Help/strategy.html).

Part VI. Monte Carlo Tree Search and Othello Competition (Optional)

If you want, you can enter your agent in the CSC384 Othello competition. The finalists and winner of the competition will receive a shout out on the course website. We are planning to run most of the competition after the last day of class. An tentative plan is to run the competition in two rounds. The first round will use a 6x6 board and the second round will use an 8x8 board. Both have a time constraint of 10 seconds.

To submit an AI to the competition, simply include a file agent_YOURID.py (such as agent_sonyaa.py) with your homework submission. You can restructure the code in your submission as you please as this file will not be marked.

If you like, for this part of the assignment you can explore other gaming algorithms like Monte Carlo Tree Search (MCTS). MCTS was initially created for Go in 2006. Many tools and applications have been based on this algorithm, including AlphaGo. This is most advanced Go AI to date and it was developed by Deepmind. MCTS provided a foundation for AlphaGo, which was augmented by complex neural networks.

In this competition, you can implement a basic Monte Carlo Tree Search algorithm and attempt to improve on it. If you submit a vanilla version of MCTS, we will reward you with an extra 5 bonus points. If you submit an enhanced version that outperforms the vanilla version, we will reward you with an extra 10 bonus points.

To help you, we provide a overview of MCTS. There are four stages in each iteration of the algorithm that are detailed below. MCTS will iterate through each of these four stages while time allows:

1. Selection: This step builds a game tree with the current board as the initial state S_0 . The child with the highest probability of winning is then selected. We suggest that by default you use the Upper Confidence Bound (UCB) statistic to select the child with the highest probability of winning.

2. Expansion: After selecting a child state, S_1, we must determine if it is a terminal state. (the `get_possible_moves(board, color)` function returns an empty result at a terminal state). If the state is a non-terminal state, it will be expanded.

3. Simulation: We don't know the consequence of selecting this state. To have estimate the potential benefit to be derived from the state, we will simulate game play from that state. Do to that, randomly assign the moves to generate subsequent states until a terminal state is reached. (For instance, beginning from S_1, randomly make moves for both players until the game ends. If we win, make note of the fact that we have derived a reward of 1; otherwise, we have derived a reward of 0.

4. Backpropagation: Once we finish a any simulation, we need to update the simulation results with information about the reward that was derived from the simulation.rTo do this, rewards found at terminals will be back-propagated through parent nodes. Information must be updated at every ancestor of the terminal, one-by-one, until reaching the root (i.e the initial state, S_0).

Once time runs out, MCTS can select the move associated with the most simulations or highest average payoff.

To help you better understand it, we have an example to demonstrate this process. Click here to view slides.

The MCTS we introduced here is a 'vanilla' version that makes use of UCB. However, there are many improvements you can add to make this algorithm to perform better. For example, the simulation stage detailed above is purely random. This makes MCTS generalize to other games but at the cost of a high variance. It can take a very long time to compute a reliable result.

1. To address this issue directly, you can generate non-random moves during your simulations using heuristic functions. More specifically, you can create a heuristic function using expert knowledge, or you can explore training a heuristic based on previous games.

2. By improving the efficiency of simulation, the algorithm should yield better estimation of states more quickly. See this paper for more information: https://doi.org/10.1145/1273496.1273531

3. UCB plays an important role in the selection stage. You may want to explore enhanced versions of UCB to further improve performance.

4. Pruning sub-optimal moves from the search tree is also a feasible way to reduce complexity. The easiest way is to use expert knowledge to identify and exclude unattractive branches.

You are welcome to choose any of above directions to explore or you are free to come up with your own ideas. You can use external libraries in this part of the assignment, assuming they are installed on teach.cs. However, do not include any number of search trees directly in your submission. Parallel processing and GPU acceleration are also prohibited.

HAVE FUN and GOOD LUCK!