Programming Assignment 2: Checkers AI

Due Jul 6 by 10pm **Points** 100

Released: June 6, 2021

Updated on June 13: autograder.py and driver.py modified!

Table of Contents.

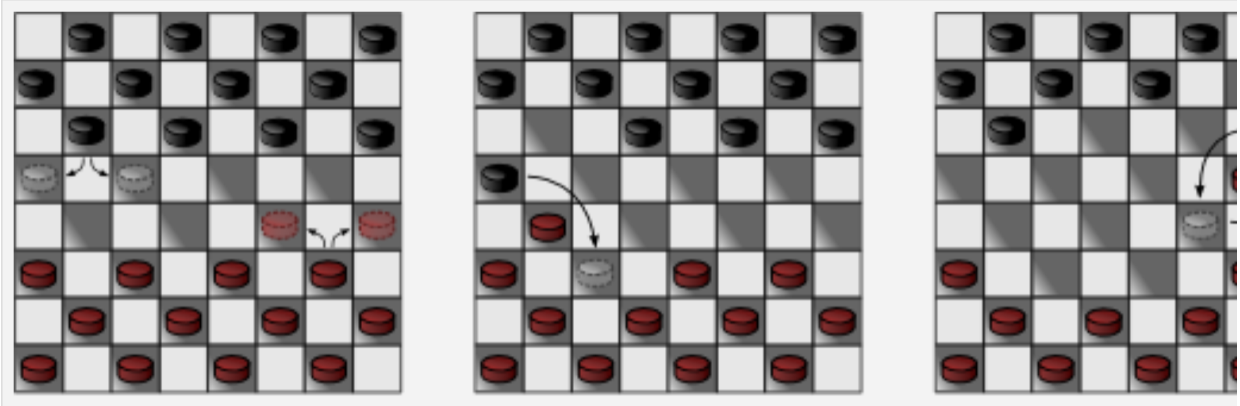

Fig 1. At left, the initial state and the possible moves for two of the pieces. In t middle, a sample position with the possibility of one capture. At right, a samp position with the possibility of two captures.

Warning (Please read this)

Warning (Please read this)

As ever, we are aware that solutions to this problem, or related ones, may exist on the internet. **Do not use these solutions as this would be plagiarism.** To earn marks on this assignment you must develop your own solutions. Also please consider the following points, as you did when implementing prior assignments:

- **Do not add any non-standard imports in the python files you submit** (all imports already in the starter code must remain). All imports that are available on teach.cs are considered to be standard.
- **Do not change any of the supplied files except for** `agent.py` **and** `agent_competition.py` **(if you choose to enter the competition).**
- **Make certain that your code runs on teach.cs using python3**. You should all have an account on teach.cs and you can log in, download all of your code (including all of the supplied code) to a subdirectory of your home directory, and use the command `python3 autograder.py` and test it there before you submit. Your code will be graded by running it on teach.cs, so the fact that it runs on your own system but not on teach is not a legitimate reason for a regrade.
- The test cases used in the autograder reflect the test cases we will use during marking. The test cases you have are of similar difficulty, so the grade shown to you by the autograder will be a reasonable predictor of your final grade on the assignment. We will also look for certain things in the assignments (e.g., running them through code plagiarism checkers, looking at assignments that fail all tests, etc.). If we have **good reasons** we will change your grade from that given by the autograder either up or down.

Introduction

Acknowledgements: This project is based on ones used in Columbia University's Artificial Intelligence Course (COMS W4701) as well as the University of Pittsburgh's Artificial Intelligence Course (CS2710). Special thanks to Dr. Daniel Bauer and Dr. Janice Wiebe for sharing their code and their ideas on the assignment.

Checkers is a 2-player board game that is played with distinct pieces that are typically black on one side and red on the other side, each side belonging to one player. Our version of the game is played on an 8x8 chess board, which is standard. Players (red and black) take turns moving pieces on the board.

Moving the pieces is dictated by the rules of the game, and can result in capturing the opponent's pieces. An overview of Checkers is provided at **https://en.wikipedia.org/wiki/Draughts. (https://en.wikipedia.org/wiki/Draughts)** Note that the version of Checkers that we will be coding is Standard English Draughts; **the rules are explained here. (https://en.wikipedia.org/wiki/English_draughts)** English Draughts includes mandatory captures.

Objective: The player's goal is to have a majority of the remaining pieces.

Game Ending: The game ends as soon as one of the players has no pieces remaining or no legal moves left.

Rules: At the beginning of the game, each player has 12 pieces of red or black colour. The pieces are placed on the dark squares of the board across the first three rows on each side (Figure 1, at left). The

placed on the dark squares of the board across the first three rows on each side (Figure 1, at left). The black player makes the first move.

Each piece can only move forward diagonally in two ways: 1) moving one space, if the adjacent space is empty (Figure 1, at left), or 2) moving two spaces if the adjacent space is occupied by the opponent, and space beyond that is empty (Figure 1, in the middle). In the second case, the player 'captures' the opponent's piece that was jumped over. If a move leads to capture, and the piece ends up sitting at a position that can jump over another opponent's piece, the player can move it again and capture another piece of the opponent. This sequence of moves does not need to be on a straight line and can be 'zigzag'.

When one of the pieces reaches the last row of the board (or the first row from the opponent's side), it turns into a "king" and will be granted a unique power: it can move both forward and backward. In our GUI a king is a piece with a yellow border.

We recommend that you start this assignment by playing some games of checkers to develop a better understanding of how the game works and what strategies can give you an advantage. An online version of the game that allows you to compete against an AI can be found at this link. (https://www.mathsisfun.com/games/checkers-2.html)

Starter Code

The starter code contains 6 files. You can download all the code and supporting files as a **Zipped file archive**. In that archive you will find the following files:

Files you can use to test your solution:

`autograder.py`	A file with some basic tests of the functions you must write.

Files you'll edit and submit on Markus:

`agent.py`	The file where you will implement your game agent; this will be graded.
`agent_competition.py`	A file you can use if you choose to help write an agent for the Checkers competition (which is optional).

Files you might want to look at (look but don't modify):

`driver.py`	This contains a simple graphical user interface (GUI) for Checkers; and the game ”manager” that stores the current game state and communicates with different player AIs.
`checkers_game.py`	This contains functions for computing legal moves, captured pieces, and successor game states. These are shared between the game

checkers_game.py	and successor game states. These are shared between the game manager, the GUI, and the AI players.
randy.py	A simple agent that picks moves at random that you can use to see the game in operation.

Game State Representation: Each game state contains two pieces of information: The current player and the current pieces on the board. Throughout our implementation, Player 1 (black) is represented using a 'b', and Player 2 (red) is represented using an 'r'.

The board is represented as a list of lists. The inner lists represents each row of the board. Each entry in the rows is either an empty square (indicated with a '.'), a black piece (a 'b'), or a red piece (an 'r'). Black pieces that are kings are indicated with a 'B' and red pieces that are kings are indicated with an 'R'. For example, an 8x8 board might look like this at the start of the game:

```
[ ['b','.','b','.','b','.','b','.'],\
['.','b','.','b','.','b','.','b'],\
['b','.','b','.','b','.','b','.'],\
['.','.','.','.','.','.','.','.'],\
['.','.','.','.','.','.','.','.'],\
['.','r','.','r','.','r','.','r'],\
['r','.','r','.','r','.','r','.'],\
['.','r','.','r','.','r','.','r'] \
]
```

Each move is represented as a list of tuples. For example, a move that takes a piece from position 5,1 to position 4,2 will be represented as:
[(5, 1), (4, 2)]
A move that contains a jump and which takes a piece from position 6,3 to position 4,5 and then to 2,7 via jumps will be represented as:
[(6, 3), (4, 5), (2, 7)]
Note that your starter code (checkers_game.py) contains a class of type Board that has two attributes: a board (which is a list of lists) and a move (which is a list of tuples). Be mindful of the distinction between an object that is of type Board and the board representation itself, which is a list of lists.
Running the code: You can run two AI agents against one another via the Checkers GUI. Running the code will bring up a game window. Each agent will alternate turns.

Our GUI takes two AI programs as command line parameters. When two AIs are specified at the command line you will be able to watch them play against each other. We have included one AI named Randy (and which is coded in the file randy.py) that will select checkers moves at random. To see Randy play against itself, type

```
$python3 driver.py randy randy
```

You will write your own agent in the agent.py file. Once you create an agent, you may want to try playing it 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.

Time Constraints: When we test your AI player, we will be expecting it to make a move within 10 seconds. If no move has been selected, the AI will lose the game.

Mark Breakdown

Minimax [30 pts]

The first thing you will do is to write a function compute `utility(state, color)` that computes the utility of a final game board state. Note that state will be an object of type Board; to access the board as a list of lists you will want to evaluate the board attribute of that state (i.e. state.board which will be in the format described above). The utility should be based on the number of pieces of the player's colour minus the number of pieces of the opponent. Make each regular piece worth one point and each king worth 2. If your agent has a single king and six regular pieces, assign this a value of 2*1 + 6 = 8. If your agent's opponent has two kings and 3 regular pieces, assign this a value of 2*2 + 3 = 7 points. The difference in points between agents, and the result of compute_utility, will therefore be 8-7 = 1.

Then, implement the method `select_move_minimax(state, color, limit = 5, caching = 0)`. For the time being, you can ignore the limit and caching parameters that the function will also accept; we will return to these later. This will accept a state (which is an object of type Board) as well as the player's colour ('r' or 'b'). It will return a single move that corresponds with the minimax strategy. More specifically, your function should select the action that leads to the state with the highest minimax value. The return value should be a list of tuples that represent the move, as above. Implement minimax recursively by writing two functions `minimax_max_node(state, color, limit, caching)` and `minimax_min_node(state, color, limit, caching)`. Again, you can just ignore the limit and caching parameters for now.

Hints: Use the successors(state,player) function in `checkers_game.py`, which returns all the the successive states (which includes moves and successive board representations) for a given player. Pay attention to which player should make a move for min nodes and max nodes.

While you can test some of the functions in your MINIMAX algorithm using the autograder, if you try to run it with driver.py you may end up with a stack overflow! This is because the game of checkers is large. We will have to implement a depth limit to realize an agent that can operate in a tractable amount of time and memory.

Depth Limit [10 pts]

To make your agent really functional, you must implement a depth limit. Your starter code is structured to do this by using the '-d' flag at the command line. For example, if you type

```
$python3 driver agent agent -d 6
```

the game manager will call your agent's MINIMAX routine with a depth limit of 6. Note that the code implements a default depth limit of 5, which you can override.

Change your Minimax code to recursively send the 'limit' parameter to both `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.

What is the largest depth limit you can play without needing more than 10 seconds per move?

Alpha-Beta Pruning [30 pts]

The simple minimax approach is quite slow. To ameliorate this we will write the function `select_move_alphabeta(state, color, limit=5, caching=0, ordering=0)` to compute the best move using alpha-beta pruning. The parameters and return values will be the same as for minimax. You can ignore the 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(state, color, alpha, beta, limit, caching=0, ordering=0)` and `alphabeta_max_node(state, color, alpha, beta, limit, caching=0, ordering=0).` As with Minimax, recursively sends the 'limit' parameter to `alphabeta_min_node` and `alphabeta_max_node.` When you arrive at your depth limit, call compute_utility to generate a utility value for each given state.

Playing with pruning should speed up decisions for the AI, but you may still need to use a fairly shallow depth. Use the command

```
$driver agent agent -d 6 -a
```

to play against your agent using the ALPHA-BETA algorithm with a depth limit of 6.

What is the largest depth limit you can play without needing more than 10 seconds per move?

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, or that checks values against stored alpha and beta parameters. Modify your Minimax and alpha-beta pruning functions to store states in that dictionary after their 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

```
$driver agent agent -d 6 -c
```

the game manager will call your agent's Minimax routines with the 'caching' flag on. If instead you type

```
$driver agent agent -d 6 -a -c
```

the game manager will call your agent's ALPHA-BETA routines with the 'caching' flag on.

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 computing utility in your alpha-beta or Minimax routines.

When you submit your code, however, please use compute_utility as the heuristic you use when you hit the depth limit in your game agent. This will facilitate marking! You can use your own heuristic in your agent_competition.py file, however.

Some Ideas for Heuristic Functions for Checkers Game

1. Consider board locations where pieces are stable, i.e. where they cannot be captured 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: https://www.ultraboardgames.com/checkers/tips.php).

In addition to engineering your own heuristic, **please include a (short) description that details your heuristic as a comment at the start of your solution file.**

Node Ordering Heuristic [10 pts]

Finally, note that 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, try using your heuristic to order the nodes that you explore. If your heuristic makes reasonably good estimates of non-terminal state values, this should lead to a little bit of a speed up because a good ordering will lead to more pruning. Note that you can use your own heuristic to order nodes when you submit your code. Use compute_utility to assess values of nodes when you hit the depth limit, however.

To run your code with node ordering, use the -o flag. More specifically if you type:

$driver agent agent -d 6 -a -c -o

You will run your agent against itself using alpha-beta with a depth limit of 6 and both caching and node ordering ON.

Checkers Competition (Optional)

If you want, you can enter your agent in the CSC384 Checkers competition. The finalists and the winners 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. A tentative plan is to run the competition in a league format on an 8x8 board and a time constraint of 10 seconds.

8x8 board and a time constraint of 10 seconds.

To submit an AI to the competition, simply include the file agent_competition.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, you can explore other gaming algorithms like Monte Carlo Tree Search (MCTS) as you develop your game agent for competition. MCTS was initially created for Go in 2006. Many tools and applications have been based on this algorithm, including AlphaGo. This is the 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.

To help you, we provide an 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 S0. 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, S1, 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 estimate the potential benefit to be derived from the state, we will simulate gameplay from that state. To do that, randomly assign the moves to generate subsequent states until a terminal state is reached. (For instance, beginning from S1, 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 any simulation, we need to update the simulation results with information about the reward that was derived from the simulation. To do this, rewards found at terminals will be back-propagated through parent nodes. The information must be updated at every ancestor of the terminal, one by one, until reaching the root (i.e the initial state, S0).

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

The agent_competition.py file also includes some MCTS function stubs to get you started, in case you would like to use these (you don't have to)!

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

- 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.
- By improving the efficiency of simulation, the algorithm should yield a better estimation of states more quickly. See this paper for more information: **https://doi.org/10.1145/1273496.1273531 (https://doi.org/10.1145/1273496.1273531)**
- UCB plays an important role in the selection stage. You may want to explore enhanced versions of UCB to further improve performance.
- 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 the 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.

What to Submit

You will be using MarkUs to submit your assignment. You will submit two files (one of which is optional):

1. Your modified `agent.py`
2. Your modified `agent_competition.py` (optional)

Note that while the assignment is due on July 6, we will continue to accept agents for competition until **August 10 at Midnight!**

GOOD LUCK!