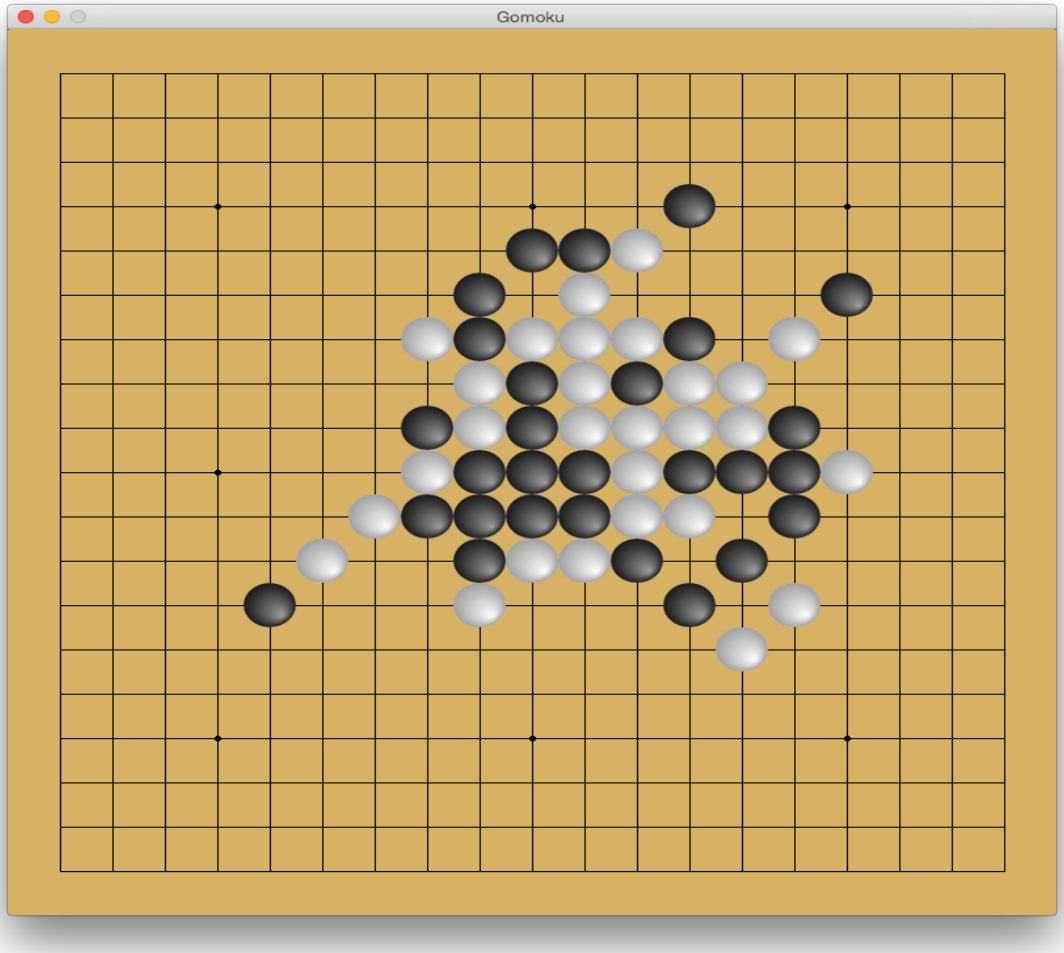
CSCI 561: Foundations of Artificial Intelligence
Instructor: Sheila Tejada
Homework #1: Adversarial Search
Due Date: June 12, 11:59 pm

In this homework, you will create a program to determine the next move for a player in **Go-Moku** game using Greedy, Minimax, and Alpha-Beta pruning algorithms.

Introduction to Go-Moku:

Go-Moku or *Five in line* is a traditional oriental game, originally from China. The game consists of a grid of squares which can be any size as seen in the image below. Usually,



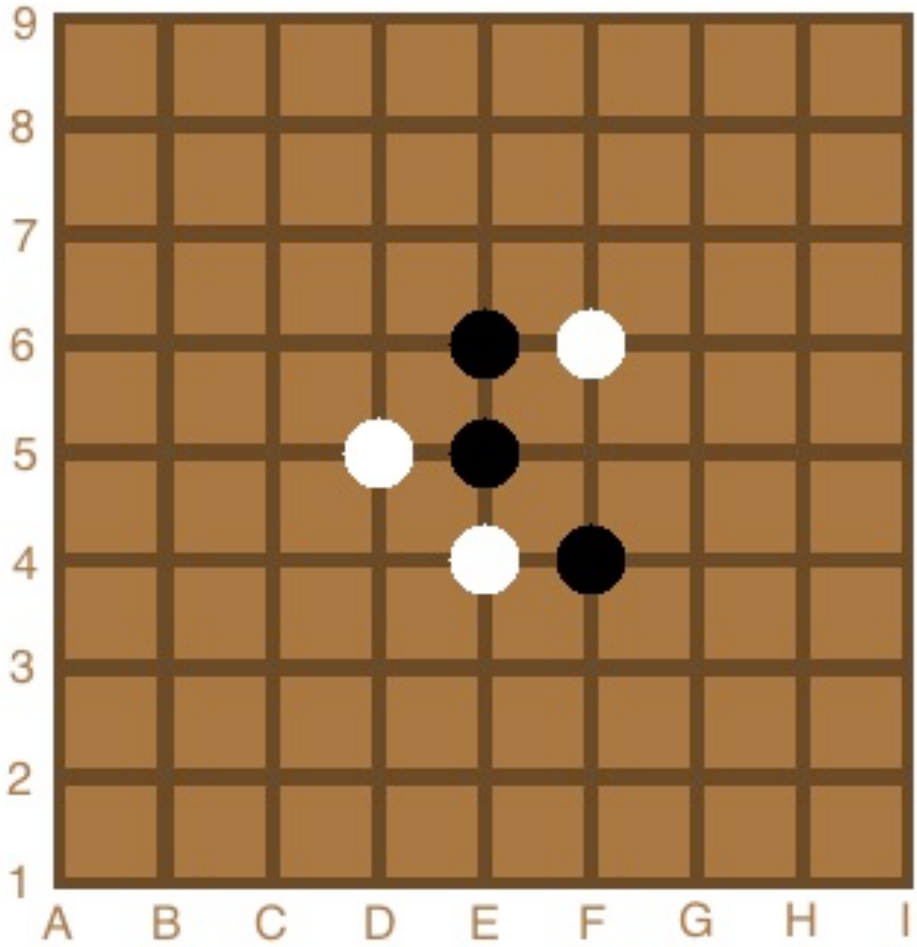
it is 19x19 and no smaller than 15x15. Players choose colors of their stones and decide who will play first. When playing with traditional white and black stones, black always goes first. The board starts empty. The first player then plays a single stone on the center intersection of the grid. Players then alternate playing 1 stone per turn. **For this homework we would impose the restriction that the new stone has to be placed adjacent (horizontally, vertically, or diagonally) to any of the existing stone.** This is different from the traditional play-style where the new stone can be placed anywhere on the board. The objective of the game is to get five or more stones in a row (horizontally, vertically, or diagonally). The first player to achieve this wins the game.

Tasks:

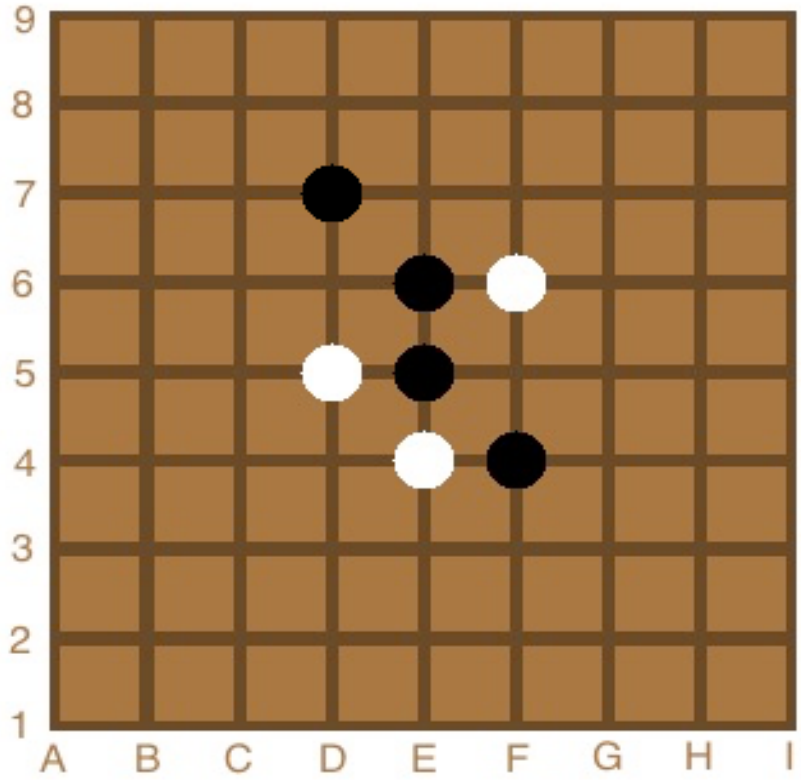
In this homework, you will wrote a program to determine the next move by implementing the following algorithms:

- Greedy
- Minimax
- Alpha-Beta

Board representation & Legal moves:



For this homework we will assume that Player-1 places black stones and goes first and Player-2 places white stones. The vertical lines are named in alphabetical order from left to right starting with A and the horizontal lines are named in numerical order from bottom to top starting from 1. As you can see in the given board condition above, both players have placed 3 stones each and the next turn will be Player-1's. Players can only place stones on the intersection of the grid. Any intersection can be identified with the combination of the letter of the column and number of the row it belongs to. For example, the center of the grid shown above will be named "E5". We will use the same naming convention for specifying any moves made by any player. As mentioned above, the new stone has to be placed adjacent (horizontally, vertically, or diagonally) to any of the existing stones. So in the current board state, the legal moves for Player-1 for his next turn would be "C4", "C5", "C6", "D3", "D4", "D6", "D7", "E3", "E7", "F3", "F5", "F7", "G3", "G4", "G5", "G6", and "G7". If Player-1 makes the move "D7" then the board state becomes as following:



End game:

The game ends when a player has five or more stones in a row (horizontally, vertically, or diagonally) or when a player cannot make any legal move (when this happens the game is considered to be a draw).

Evaluation function:

The goal of the game is to get five stones in a row and also preventing your opponent to get five stones in a row. The evaluation function explained here doesn't scan the entire board to evaluate the move, but rather scans the affected row, column, and diagonals when a stone is placed. **We will assume that the evaluation value for the input board state is 0.** Now, the evaluation value for any subsequent state after making a series of moves is calculated as the summation of the evaluation value of all the moves for their respective board states. Suppose, in a game tree of depth 2, we want to evaluate the value of the board state after max-player makes a move *m1* and min-player makes a move *m2.* The value for the leaf move of this game thee for the series of moves *m1, m2* would be (0 + evaluation value of *m1* on initial board state + evaluation value of *m2* on the board state after making the move *m1* from initial board state). Thus, the evaluation value of any board state is equal to the summation of the evaluation value of the previous board state and the evaluation value of the move taken from the previous board state that resulted in the current board state. Now, the evaluation value for any move on a given board state is calculated using the combination of the following categories.

- ***win : 50000 for max-player & -50000 for min-player***

  If this move wins the game. This happens when making this move gets you five or more stones in a row.

- ***blockClosedFour : 10000 for max-player & -10000 for min-player***

  If this move blocks win for the opponent. This happens when the opponent has four stones in a row which are blocked at one end by one of your stone and making this move will block the four stones from both ends.

- ***createOpenFour : 5000 for max-player & -5000 for min-player***

  If making this move will get you four stones in a row which are not blocked at either end by the opponent's stone.

- ***createClosedFour : 1000 for max-player & -1000 for min-player***

  If making this move will get you four stones in a row which are blocked at one end by the opponent's stone.

- ***blockOpenThree : 500 for max-player & -500 for min-player***

  This happens when your opponent has three stones in a row which are not blocked at either end by your stone and making this move will block the three stones from one of the end.

- ***blockClosedThree : 100 for max-player & -100 for min-player***

  This happens when your opponent has three stones in a row which are blocked at one end by your stone and making this move will block the three stones from both ends.

- ***createOpenThree : 50 for max-player & -50 for min-player***

  If making this move will get you three stones in a row which are not blocked at either end by the opponent's stone.

- ***createClosedThree : 10 for max-player & -10 for min-player***

  If making this move will get you three stones in a row which are blocked at one end by the opponent's stone.

- ***createOpenTwo : 5 for max-player & -5 for min-player***

  If making this move will get you two stones in a row which are not blocked at either end by the opponent's stone.

- ***createClosedTwo : 1 for max-player & -1 for min-player***

  If making this move will get you two stones in a row which are blocked one end by the opponent's stone.

As explained above, the value for any move will be the combination of the all the categories it falls into. For example, if a move satisfies both *blockOpenThree* and *createClosedFour*, then the value for that move would be 1500 for max-player or -1500 for min player. If a move does not fall into any of the category listed above then the value for that move will be 0 for both max/min player.

Note: You can assume that your player is always "max" player.
Note: Boundaries of the board can also block stones in a row. For example, if making a move gets you four stones in a row (horizontally, vertically, or diagonally) which are not blocked at one side and are blocked by the boundary of the board on the other side then this move will fall into the category of *'createClosedFour'*.

Expand order and Tie breaking:

You must evaluate the legal moves in alphanumeric order while expanding the game tree. This means:

- "A1" would be evaluated before "A5"
- "A3" would be evaluated before "B1"
- "A9" would be evaluated before "A13"

It is possible that more than one moves have the best value for being the next move. In this case ties between such moves are also broken in alphanumeric order following the rules for expansion order. For example, if moves "A4" and "C8" both have the best value for being the next move, then move "A4" should be reported as the best move.

Board size:

The board size will be NxN, where N represents the number of horizontal and vertical rows and $15 \leq N \leq 25$.

Pseudo code:

Greedy: It is a special case of Minimax. The cut-off depth is always 1. Thus, the algorithm is very simple. You need to pick the move which has the highest evaluation value.

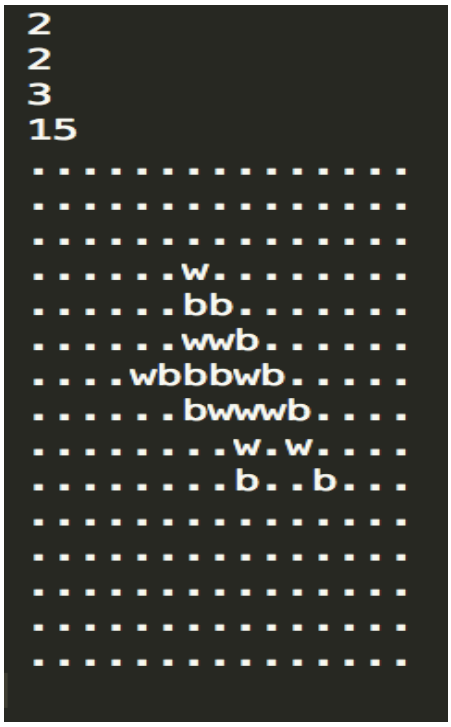
Minimax: AIMA Figure 5.3 (Minimax without cut-off) and section 5.4.2 (Explanation of Cutting off search)
Alpha-Beta: AIMA Figure 5.7 (Alpha-Beta without cut-off) and section 5.4.2 (Explanation of Cutting off search)

Input:

You will be provided with an input file *"xxx.txt"* as a command line argument that describes the current state of the game. The structure of the input file will be as follows:

<Task#> Greedy=1, Minimax=2, Alpha-Beta=3
<Your player: 1 or 2>
<Cutting off depth: d> Cut-off depth for the search tree. 1≤d≤4
<N> Number of horizontal and vertical lines, 15≤N≤25
<Board state> N lines specifying the current board state

Example:



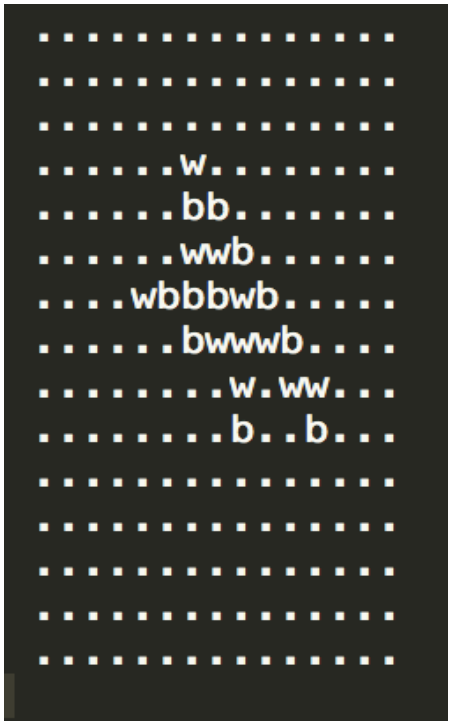
- The sample input file shown above gives you a description of a **15x15** Go-Moku board.
- The input file asks you to perform task #2, i.e. **Minimax**.
- You will be playing as **Player-2**, i.e you will be placing white stones.
- The **cutting off depth** for Minimax is given as 3. You should ignore the cutting off depth if the task is **Greedy**.

- Line-4 specifies the number of horizontal and vertical rows in the board which is this case is 15. This means that the next 15 lines in the input file will describe the state of the Go-Moku board.
- Each line of the board state will have N number of characters with no spacing between them. ‘w’ means that a white stone is placed on the intersection. ‘b’ means that a black stone is placed on the intersection. ‘.’ means that the intersection is empty.
- You can assume that the input file will have no format error.
- You can assume that there will be at least one legal move available for the current player.

Output:

Greedy:

The program should output one file named ***“next_state.txt”*** showing the *next state* of the board after the greedy move in the following format.



Each line represents one row of the board. ‘.’ represents empty intersections, ‘b’ represents black stones, and ‘w’ represents white stones on the board.

Minimax:

The program should output two files named “next_state.txt” showing the next state of the board after the minimax move and “traverse_log.txt” showing the traverse log of your program in the following format.

```
Move,Depth,Value
root,0,-Infinity
D8,1,Infinity
C7,2,-Infinity
B6,3,5
C7,2,5
B7,3,5
C7,2,5
B8,3,5
C7,2,5
C6,3,5
C7,2,5
C8,3,10
C7,2,10
C9,3,10
C7,2,10
D6,3,5
C7,2,10
D7,3,10
C7,2,10
D9,3,11
C7,2,11
…..
…..
…..
```

Note: The Minimax traverse log requires 3 columns. Each column is separated by ‘,’ (a single comma). Three columns are move, depth, and value. Everything is case sensitive.

“Move”: is the name of the move you are evaluating. For example, Player-1 chooses move “D8” as the first move to be explored as per the evaluation order. The depth of the move “D8” is 1. Then Player-2 chooses “C7” as his/her move as per the evaluation order and depth of the move “C7” becomes 2. “root” is the special name assigned to the root.

“Depth”: is the depth of the move. The depth of root is 0.

“Value”: is the evaluation value of the move. The value is initialized to “-Infinity” for the max move and “Infinity” for the min move, except for the leaf moves. The value of the moves will be update when its children return their value to the move. The value of the leaf node is the evaluated value as explained in the *‘evaluation function’* section, for example, move “B6” has value 5.

The algorithm traverses from root node. The log should show when:
1) The algorithm traverses down to the move.

2) The algorithm traverses upward to the move.

For example, the log shows value of move “C7” when traversing from “D8”. It shows “C7” again when the move is traversed back from its children “B6”, “B7”, “B8” and so on. You can relate the reporting of the traverse log to DFS traversal of the Minimax game tree.

Alpha-Beta:

The program should output two files named “next_state.txt” showing the next state of the board after the alpha-beta move and “traverse_log.txt” showing the traverse log of your program in the following format.

```
Move,Depth,Value,Alpha,Beta
root,0,-Infinity,-Infinity,Infinity
D8,1,Infinity,-Infinity,Infinity
C7,2,-Infinity,-Infinity,Infinity
B6,3,5,-Infinity,Infinity
C7,2,5,5,Infinity
B7,3,5,5,Infinity
C7,2,5,5,Infinity
B8,3,5,5,Infinity
C7,2,5,5,Infinity
C6,3,5,5,Infinity
C7,2,5,5,Infinity
C8,3,10,5,Infinity
C7,2,10,10,Infinity
C9,3,10,10,Infinity
C7,2,10,10,Infinity
D6,3,5,10,Infinity
C7,2,10,10,Infinity
D7,3,10,10,Infinity
C7,2,10,10,Infinity
D9,3,11,10,Infinity
C7,2,11,11,Infinity
…..
…..
…..
```

The Alpha-Beta traverse log requires 5 columns. Each column is separated by ‘,’ (a single comma). Five columns are move, depth, value, alpha, and beta. The description is same as with Minimax log. However, you need to show alpha and beta values in the Alpha-Beta traverse log.

Guidelines:

- You can use Python2.7 or Python3.4 to implement your code. Make sure that the code runs on Vocareum.
- The name of the code should be **"gomoku.py"** for Python 2.7 and **"gomoku3.py"** for Python3.4.
- The command to run your code would be "python gomoku{3}.py *inputfile*", where you will read the input file name from the command line argument.
- Input file is a text file ending with ".txt" extension.
- For any test case, it will be marked as correct only if both next state and traverse log are correct. (For greedy only next state will be checked).
- **Late submissions:** Late submissions will be allowed for additional 24 hours, i.e. till **June 13, 11:59 pm**. The late submission **penalty will be 50%** of the score you get. Homework will not be accepted after June 13.
- Go through **"Student Help - Vocareum.pdf"** uploaded on the course website to get started with Vocareum.
- If you have any issues with Vocareum with regards to logging in, submission, code not executing properly, etc., please contact any of the TAs.
- Multiple submissions are allowed, and your last submission will be graded. So you are encouraged to submit early and often in order to iron out any problems, especially issues with the format of the final output. The performance of you program will be measured automatically; failure to format your output correctly may result in very low scores, which will not be changed.
- This is an individual assignment. You may not work in teams or collaborate with other students. You must be the sole author of 100% of the code you turn in.
- You may not look for solutions on the web, or use code you find online or anywhere else.
- You may use external resources to learn basic functions of Python (such as reading and writing files, handling text strings, and basic math), but the computation performed by the program must be your own work.
- Failure to follow the above rules is considered a violation of academic integrity, and is grounds for failure of the assignment, or in serious cases failure of the course.
- Please discuss any issues you have on the discussion board. Do not ask questions about the homework by email; if we receive questions by email where the response could be helpful for the class, we will ask you to repost the question on the discussion boards.