Programming Project 2: Adversarial Search

The minimax search is especially known for its usefulness in calculating the best move in two-player games where all the information is available, such as chess or tic-tac-toe. It consists of navigating through a tree that captures all the possible moves in the game, where each move is represented in terms of loss and gain for one of the players. It follows that this can only be used to make decisions in zero-sum games, where one player's loss is the other player's gain. Theoretically, this search algorithm is based on von Neumann's minimax theorem which states that in these types of games there is always a set of strategies that leads to both players gaining the same value and that seeing as this is the best possible value one can expect to gain, one should employ this set of strategies.

Assignment Description

The task in this programming assignment is to implement an agent that plays **the Max-Connect4 game (https://en.wikipedia.org/wiki/Connect_Four)** using search. Figure 1 below shows the first few moves of a game. The game is played on a 6x7 grid, with six rows and seven columns. There are two players, player A (red) and player B (green). The two players take turns placing pieces on the board: the red player can only place red pieces, and the green player can only place green pieces.

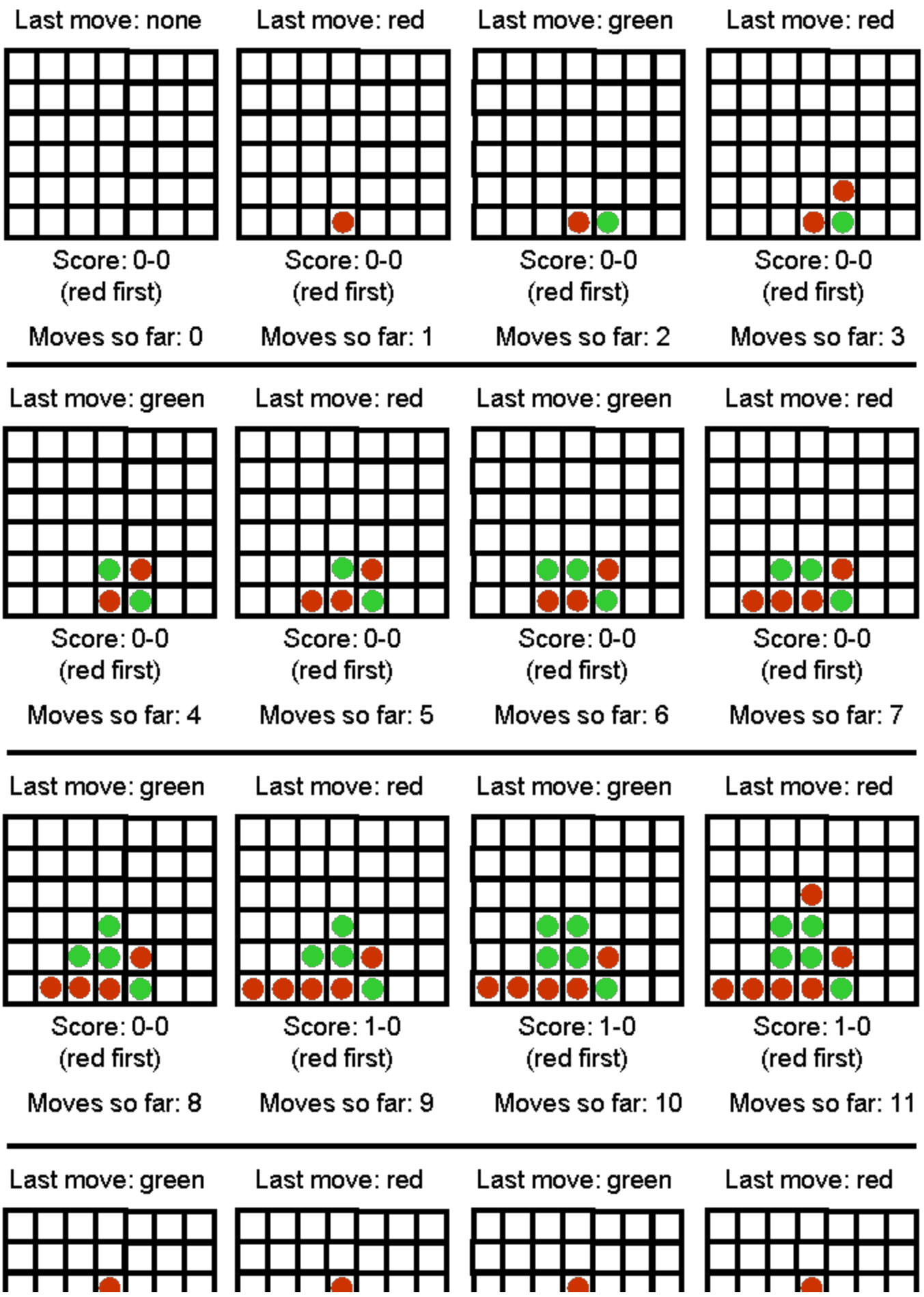
It is best to think of the board as standing upright. We will assign a number to every row and column, as follows: columns are numbered from left to right, with numbers 1, 2, ..., 7. Rows are numbered from bottom to top, with numbers 1, 2, ..., 6. When a player makes a move, the move is completely determined by specifying the COLUMN where the piece will be placed. If all six positions in that column are occupied, then the move is invalid, and the program should reject it and force the player to make a valid move. In a valid move, once the column is specified, the piece is placed on that column and "falls down", until it reaches the lowest unoccupied position in that column.

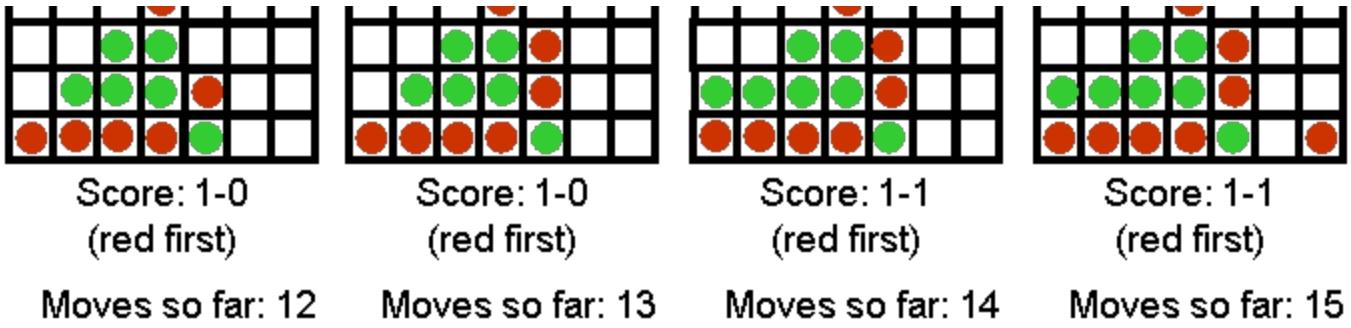
The game is over when all positions are occupied. Obviously, every complete game consists of 42 moves, and each player makes 21 moves. At the end of the game, the score is determined as follows: consider each quadruple of four consecutive positions on board, either in the horizontal, vertical, or each of the two diagonal directions (from bottom left to top right and from bottom right to top left). The red player gets a point for each such quadruple where all four positions are occupied by red pieces. Similarly, the green player gets a point for each such quadruple where all

four positions are occupied by green pieces. The player with the most points wins the game.

Your program will run in two modes:

- The interactive mode, which is best suited for the program playing against a human player, and a one-move mode, where the program reads the current state of the game from an input file, makes a single move, and writes the resulting state to an output file.
- The one-move mode can be used to make programs play against each other. Note that in this mode, the program may be either the red or the green player. This will be specified by the state, as saved in the input file. The description of the format of the input file is provided below very soon.


Last move: none
Score: 0-0
(red first)
Moves so far: 0
Last move: red
Score: 0-0
(red first)
Moves so far: 1
Last move: green
Score: 0-0
(red first)
Moves so far: 2
Last move: red
Score: 0-0
(red first)
Moves so far: 3
Last move: green
Score: 0-0
(red first)
Moves so far: 4
Last move: red
Score: 0-0
(red first)
Moves so far: 5
Last move: green
Score: 0-0
(red first)
Moves so far: 6
Last move: red
Score: 0-0
(red first)
Moves so far: 7
Last move: green
Score: 0-0
(red first)
Moves so far: 8
Last move: red
Score: 1-0
(red first)
Moves so far: 9
Last move: green
Score: 1-0
(red first)
Moves so far: 10
Last move: red
Score: 1-0
(red first)
Moves so far: 11
Last move: green
Last move: red
Last move: green
Last move: red



Input Files

In the input files, a "0" stands for an empty spot, a "1" stands for a piece played by the first player, and a 2 stands for a piece played by the second player. The last number in the input file indicates which player plays NEXT (and NOT which player played last). Sample input files that you can download are **input1.txt (https://elearning.mines.edu/courses/36718/files/3983289?wrap=1)** ⬇ **(https://elearning.mines.edu/courses/36718/files/3983289/download?download_frd=1)** and **input2.txt (https://elearning.mines.edu/courses/36718/files/3983290?wrap=1)** ⬇ **(https://elearning.mines.edu/courses/36718/files/3983290/download?download_frd=1)** . You are free to make other input files to experiment with, as long as they follow the same format.

Interactive Mode

In the interactive mode, the game should run from the command line with the following arguments (assuming a Java implementation, with obvious changes for C++ and Lisp implementations):

java maxconnect4 interactive [input_file] [computer-next/human-next] [depth]

For example:

java maxconnect4 interactive input1.txt computer-next 7

- Argument interactive specifies that the program runs in interactive mode.
- Argument [input_file] specifies an input file that contains an initial board state. This way we can start the program from a non-empty board state. If the input file does not exist, the program should just create an empty board state and start again from there.
- Argument [computer-next/human-next] specifies whether the computer should make the next move or the human.
- Argument [depth] specifies the number of moves in advance that the computer should consider while searching for its next move. In other words, this argument specifies the depth of the search tree. Essentially, this argument will control the time takes for the computer to make a move.

After reading the input file, the program gets into the following loop:

1. If computer-next, goto 2, else goto 5.
2. Print the current board state and score. If the board is full, exit.
3. Choose and make the next move.
4. Save the current board state in a file called computer.txt (in same format as input file).
5. Print the current board state and score. If the board is full, exit.
6. Ask the human user to make a move (make sure that the move is valid, otherwise repeat request to the user).
7. Save the current board state in a file called human.txt (in same format as input file).
8. Goto 2.

One-Move Mode

The purpose of the one-move mode is to make it easy for programs to compete against each other, and communicate their moves to each other using text files. The one-move mode is invoked as follows:

java maxconnect4 one-move [input_file] [output_file] [depth]

For example:

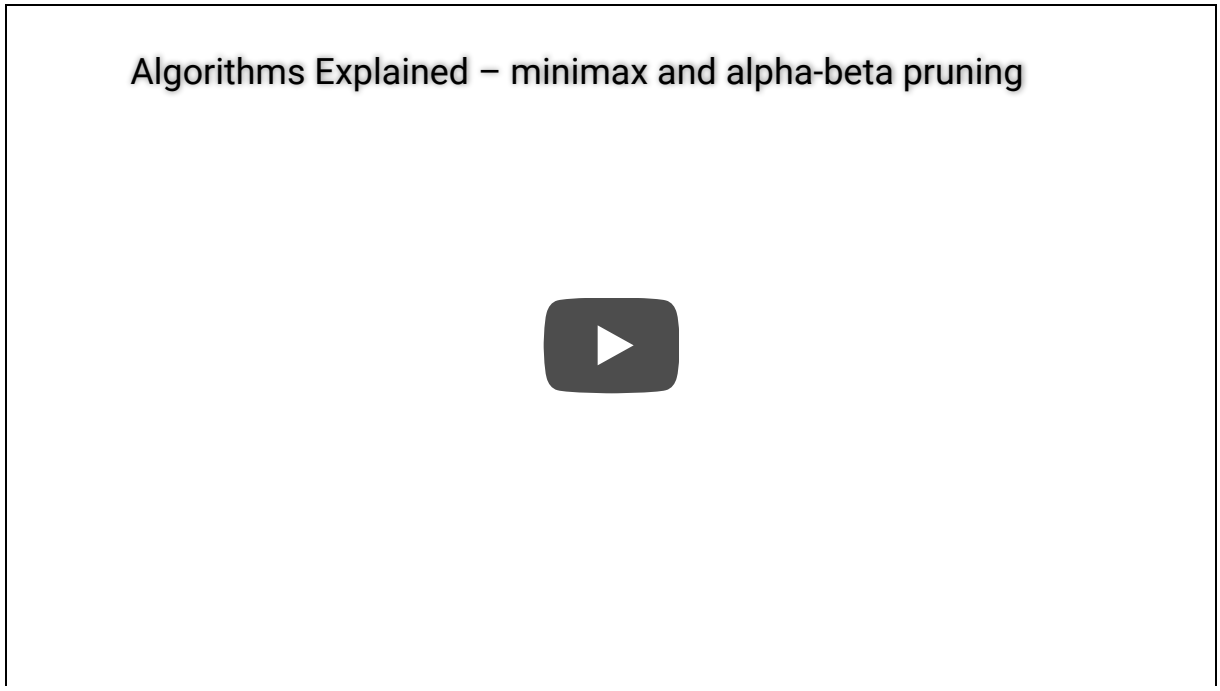
java maxconnect4 one-move red_next.txt green_next.txt 5

In this case, the program simply makes a single move and terminates. In particular, the program should:

1. Read the input file and initialize the board state and current score, as in interactive mode.
2. Print the current board state and score. If the board is full, exit.
3. Choose and make the next move.
4. Print the current board state and score.
5. Save the current board state to the output file IN EXACTLY THE SAME FORMAT THAT IS USED FOR INPUT FILES.
6. Exit

Sample codes

Here is a video tutorial on implementing the Minimax search algorithm and alpha-beta pruning.



Link of the video: https://www.youtube.com/watch?v=l-hh51ncgDI&t=474s

---

Suggestions

Pay close attention to all specifications on this page, including specifications about output format, submission format. Even in cases where the program works correctly, points will be taken off for non-compliance with the instructions given on this page (such as a different format for the program output, a wrong compression format for the submitted code, and so on). The reason is that non-compliance with the instructions makes the grading process significantly (and unnecessarily) more time-consuming.

Grading

The assignments will be graded out of 100 points.

- 20 points: Please provide 5 test cases for testing your codes.
- 25 points: Implementing plain minimax.
- 30 points: Implementing alpha-beta pruning.
- 25 points: Implementing the depth-limited version of minimax (if correctly implemented, and includes alpha-beta pruning, you also get the 25 points for plain minimax and 30 points for alpha-beta search, you don't need to have separate implementations for those). For full credit, you obviously need to come up with a reasonable evaluation function to be used in the context of depth-limited search. A "reasonable" evaluation function is defined to be an evaluation function that allows your program to consistently beat a random player. You can find some

reference evaluation functions **at (https://elearning.mines.edu/courses/36718/files/3983288?wrap=1) (https://elearning.mines.edu/courses/36718/files/3983288/download?download_frd=1) here (https://elearning.mines.edu/courses/36718/files/3983288?wrap=1) (https://elearning.mines.edu/courses/36718/files/3983288/download?download_frd=1)** .

- Bonus points (10 points): You can get up to 10 bonus points, if you implement alpha-beta pruning together with successor node prioritization. For example, the maximization player (the computer in the interactive mode) should start node expansion from the successors of the current nodes with the biggest minimax value or evaluation function value. In contrast, during the search the minimization player (human player in the interactive mode) should start node expansion from the successor node with the smallest minimax value or evaluation function value.

How to submit

Implementations in C, C++, Java, and Python will be accepted. If you would like to use another programming language, please first check with the grader via e-mail (Stephen Thoemmes: **sthoemmes@mines.edu (mailto:sthoemmes@mines.edu)**). Points will be taken off for failure to comply with this requirement.

The assignment should be submitted via Canvas. Submit a ZIPPED directory called project2.zip (no other forms of compression accepted, contact the instructor or the grader if you do not know how to produce .zip files). The directory should contain source codes. Including binaries that work on Windows 10 is optional. The submission should also contain a file called readme.txt, which should specify precisely:

1. Name and CSM Campus ID of the student.
2. What programming language is used.
3. What OS is used to compile and run the codes.
4. How the code is structured.
5. How to run the code, including very specific compilation instructions, if compilation is needed. Instructions such as "compile using g++" are NOT considered specific.

Insufficient or unclear instructions will be penalized by up to 20 points. Code that does not run on Windows 7/8/10/11, Mac OS, or Linux machines gets AT MOST half credit (50 points).

Submission checklist

- Does the submission include all required source code files?
- Does the submission include a readme.txt file, as specified above?