APS 105 — Computer Fundamentals
Lab #7: Simplified Connect6 (Recursion)
Fall 2013

Important note: The course material necessary for completing this lab will be found in your lecture notes and in the Carter text up to the end of Chapter 8.

This is a two-week lab. You must use the `submitaps105f` command to electronically submit your solution by 11:59pm on Saturday, November 23, 2013.

Objective

In Lab #6, you developed a program that represents the board of a Connect6 game, and that computes, for a given occupied square on the board, the number of contiguous stones that intersects with that square horizontally, vertically and diagonally. In this lab, you will build upon your work in Lab #6 to create a Connect6 game where a human can play against the computer, with the computer making (intelligent!) decisions on where to place stones.

Simplified Connect6

Connect6 is played with two players: one plays the black stones and one plays the white stones. The rules of Connect6 are as follows:

- The black player always plays first, laying down a single stone anywhere on the board.
- Turns then alternate between black and white players.
- In each turn, a player lays down **one** stone[1]. Each time a stone is laid down is called a *move*.
- The first player to lay down six contiguous stones wins.
- If all squares on the board are filled and there is no winner, the game is a draw.

There are two parts to this lab. In the first part, you must implement a specific strategy for the computer moves, as described in the next section. In the second part, you may implement a strategy of your own choosing. It is expected that much of your code will be the same for the first and second parts.

Part I – A Basic Computer Player

Your program must first ask the user the board dimensions, which you may assume will range from 7 to 19. Then, the program asks the user if the computer is to be the black or white player. If the computer is black, then the computer should make the first move; otherwise, the program prompts the human player to make the first move. The board is printed after every move.

[1] In actual Connect6, each player lays down two stones per turn, however, in our simplified version, each player lays down just one stone per turn.

Once the first move is out of the way, the turns alternate, with each player playing one stone per turn. After each turn, your program must print the board, and must detect whether the game has been won, or whether there is a draw. If your program detects the game is over (i.e. a win or a draw), a message is printed and the program terminates. The specific messages to print are: "White player wins.", "Black player wins." or "Draw!". If the human player attempts to play a stone on an occupied tile, an error message is printed ("That square is occupied.") and the human is asked to re-enter his/her move. See the example execution below for the specific input/output formatting.

How Should the Computer Make Moves?

The computer makes moves by using recursion to "look ahead" and predict the consequences of any potential move. Specifically, say your program is considering placing a stone at square $P_{current}$, the computer will look into the "future" by **one** move beyond that move to assess the goodness of placing a stone at $P_{current}$. That is, your program must consider how the human will respond to the computer's move at $P_{current}$. Let's refer to these two moves as: the *current*, *next* moves.

Say that the computer plays a stone at $P_{current}$ and the human responds by laying a stone at P_{next}, the score of this move sequence, from the computer's perspective, is defined as follows:

$$Score = LongestSequence(P_{current}) - LongestSequence(P_{next}) \quad (1)$$

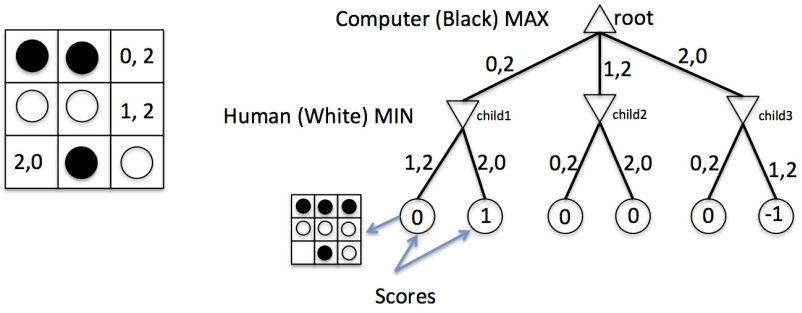
Here $LongestSequence(P_{current})$ is the longest number of stones that the computer has connected to the location $P_{current}$, and $LongestSequence(P_{next})$ is the longest number of stones that the human opponent has at location P_{next}. Higher scores reflect a better situation for the computer; lower scores represent a better situation for the human. Thus, the computer wishes to make a move that maximizes the score, whereas, it is expected that a human will respond with a move that minimizes the score. You should use your function from Lab #6 to find the length of the longest contiguous sequence of stones that intersects with a square.

Your program should use recursion to generate all possible two-move sequences (that is, one for each player) from a given board. Each two-move sequence will be scored using the equation above.

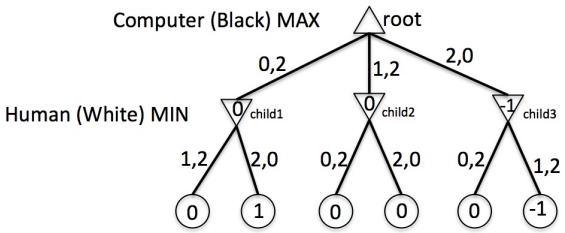
Your program should work in the following way[2]: Consider the 3×3 board in the figure below, which is a board part way through a game. Let's say that the computer is playing the black stones. The set of all possible two-move sequences can be represented using a tree, shown on the right side of the figure. This is called a *game tree*. The top of the tree (called its *root*) represents the computer's move. That is, the three lines (*edges*) coming from the root triangle show the three possible squares where the computer may lay a stone: (0,2), (1,2), or (2,0). The edges lead to three "child nodes" of the root. The child nodes represent and are connected to the human's response to the computer's move. For each computer move, there are two possible human responses (because there are only a total of 3 unoccupied squares on the board). The circles at the bottom of the tree (*leaf nodes*) represent each two-move sequence that is possible to make from the current board. The number shown inside each leaf is the score, computed according to the equation above. For example, the score of the child node labelled 1 in the second from the left leaf node is 1 because that node represents the combination of the computer moving to square (0,2)

[2]This approach is called the *minimax* algorithm. Google "minimax algorithm" to find out more about this approach – it is a typical way to write programs that play games.

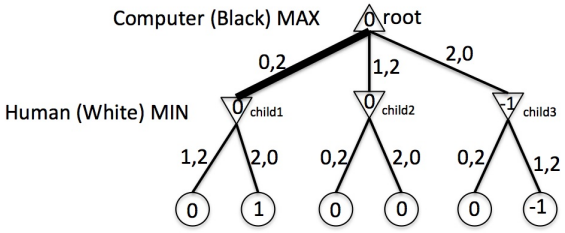
(giving a longest black sequence of 3), and the human choosing square (2,0) (which gives the longest sequence of white being 2), and 3-2=1.



Now, as mentioned above, the computer program should assume that the human will make the best move it can see for the human. That means it should choose the move for the human that minimizes the score. Thus, your program should give a score to each child node (respresenting a human's move) equal to the *minimum* score of each leaf below it in the tree. $Child1$, for example, is given a score of 0, since the two leaf nodes below it have scores of 0 and 1, making 0 the minimum. The game tree, with the child nodes of the root labelled with their scores is shown in the figure below. Notice that in the game tree, the triangles representing the root's children have their points on the bottom, indicating these are *min* nodes.



Finally, the computer should choose a move that gives the highest score, which corresponds to the maximum-score child. In this case, the children of the root have scores of 0, 0, and -1, the maximum of which is 0, as shown in the figure below (see the thick edge from the root). In this case, the computer would lay down a stone at the position (0,2). Notice that the triangle representing the root has its point at the top, indicating it is a *max* node.



If, in deciding a move, two moves have the same score, your program must choose the position with the smaller row. If there are two positions with the same score in the same row, your program must choose the solution with the smaller column.

Your program must be able to work with the computer being either the black or the white player.

Write your program in a file called `Lab7Part1.c`

Here is a sample execution of the program (your program must conform to this output):

```
Enter board dimensions (n): 7
UUUUUUU
UUUUUUU
UUUUUUU
UUUUUUU
UUUUUUU
UUUUUUU
UUUUUUU
Computer playing B or W?: B
Computer lays a stone at ROW 0 COL 0.
BUUUUUU
UUUUUUU
UUUUUUU
UUUUUUU
UUUUUUU
UUUUUUU
UUUUUUU
Lay down a stone (ROW COL): 0 0
That square is occupied.
Lay down a stone (ROW COL): 2 0
BUUUUUU
UUUUUUU
WUUUUUU
UUUUUUU
UUUUUUU
UUUUUUU
UUUUUUU
Computer lays a stone at ROW 0 COL 1.
BBUUUUU
UUUUUUU
WUUUUUU
UUUUUUU
UUUUUUU
UUUUUUU
UUUUUUU
Lay down a stone (ROW COL): 2 1
BBUUUUU
UUUUUUU
WWUUUUU
UUUUUUU
UUUUUUU
UUUUUUU
UUUUUUU
Computer lays a stone at ROW 0 COL 2.
BBBUUUU
UUUUUUU
WWUUUUU
UUUUUUU
```

```
UUUUUUU
UUUUUUU
UUUUUUU
Lay down a stone (ROW COL): 2 2
BBBUUUU
UUUUUUU
WWWUUUU
UUUUUUU
UUUUUUU
UUUUUUU
UUUUUUU
Computer lays a stone at ROW 0 COL 3.
BBBBUUU
UUUUUUU
WWWUUUU
UUUUUUU
UUUUUUU
UUUUUUU
UUUUUUU
Lay down a stone (ROW COL): 2 3
BBBBUUU
UUUUUUU
WWWWUUU
UUUUUUU
UUUUUUU
UUUUUUU
UUUUUUU
Computer lays a stone at ROW 0 COL 4.
BBBBBUU
UUUUUUU
WWWWUUU
UUUUUUU
UUUUUUU
UUUUUUU
UUUUUUU
Lay down a stone (ROW COL): 0 5
BBBBBWU
UUUUUUU
WWWWUUU
UUUUUUU
UUUUUUU
UUUUUUU
UUUUUUU
Computer lays a stone at ROW 2 COL 4.
BBBBBWU
UUUUUUU
WWWWBUU
UUUUUUU
UUUUUUU
UUUUUUU
```

```
UUUUUUU
Lay down a stone (ROW COL): 1 0
BBBBBWU
WUUUUUU
WWWWBUU
UUUUUUU
UUUUUUU
UUUUUUU
UUUUUUU
Computer lays a stone at ROW 1 COL 3.
BBBBBWU
WUUBUUU
WWWWBUU
UUUUUUU
UUUUUUU
UUUUUUU
Lay down a stone (ROW COL): 3 2
BBBBBWU
WUUBUUU
WWWWBUU
UUWUUUU
UUUUUUU
UUUUUUU
UUUUUUU
Computer lays a stone at ROW 3 COL 5.
BBBBBWU
WUUBUUU
WWWWBUU
UUWUUBU
UUUUUUU
UUUUUUU
UUUUUUU
Lay down a stone (ROW COL): 4 3
BBBBBWU
WUUBUUU
WWWWBUU
UUWUUBU
UUUWUUU
UUUUUUU
UUUUUUU
Computer lays a stone at ROW 4 COL 6.
BBBBBWU
WUUBUUU
WWWWBUU
UUWUUBU
UUUWUUB
UUUUUUU
UUUUUUU
Lay down a stone (ROW COL): 5 4
BBBBBWU
```

```
WUUBUUU
WWWWBUU
UUWUUBU
UUUWUUB
UUUUWUU
UUUUUUU
Computer lays a stone at ROW 1 COL 4.
BBBBBWU
WUUBBUU
WWWWBUU
UUWUUBU
UUUWUUB
UUUUWUU
UUUUUUU
Lay down a stone (ROW COL): 6 5
BBBBBWU
WUUBBUU
WWWWBUU
UUWUUBU
UUUWUUB
UUUUWUU
UUUUUWU
White player wins.
```

Part II – The Competition

Write another version of your Connect6 program using *any* strategy for the computer moves. Your objective here is to apply your wits, creativity and resourcefulness to produce the best computer player possible, using *any* approach you are aware of (even those that do not use recursion). Reminder that, as with any APS105 submitted work, your program must be entirely your own work.

The instructors are organizing a competition where the programs submitted for part will be played off against each other. The best programs will be selected as Finalists, and invited to the Finals stage. The Finals will be a public event, where all APS105 students are invited to watch the Finalists' programs play against one another on a large screen with graphics. You are encouraged to attend this event to cheer on your colleagues, so watch BlackBoard for an announcement. The will be prizes for the top 3 students in the course.

Competition Rules:

- Your program must conform to the identical output format as in Part I above.
- For a 19×19 board, your program *must* make a move (lay a stone) within 1 second on the computer `remote.ecf.utoronto.ca`. Otherwise, your program is disqualified from the competition.
- If your program makes an invalid move, it is disqualified from the competition.

Write your program in a file called `Lab7Part2.c`

Marking

This lab will be marked out of 10, with Part I being worth 8 marks, and Part II being worth 2 marks. A marking program will be used to automatically mark Part I of your lab. Part II will be marked by "facing off" your solution against other solutions with various levels of decision-making intelligence. The marking program will use the last version of the lab files you submitted using the `submitaps105f` command.

Full marks are given if your program works correctly, fewer if not, and zero if it cannot be compiled. Late submissions or submissions with an incorrect filename will result in a mark of 0 for the entire lab. The deadline will be strictly enforced, so avoid last minute submissions.

Important: Any submitted solution to Part I that does not use recursion will be assigned a mark of 0.

You can run a testing program, called `tester`, yourself to test the correctness of your solution. At the command line in your ECF account, run:

```
/share/copy/aps105f/lab7/tester
```

in the same directory as your solution file. The testing program will use a number of test cases to test your solution, and report success if your solution produces output that is identical to the expected output. Some of these test cases will be used by the marking program as well, but the marking program will also be using other test cases that are not included in the testing program to test the correctness of your program. This implies that even though you do not have access to the marking program, you will obtain at least partial marks if all of the test cases in the testing program report success with your solution. **Note:** the tester will only check Part I of this lab, since in Part II, you may use *any* approach you wish, it is therefore not possible for the tester to check "correctness".

Your mark for this lab will contribute 3% to your mark in the course.

What To Submit

When you have completed the lab, use the command

```
submitaps105f 7 Lab7Part1.c Lab7Part2.c
```

to submit your file. Make sure you name your file exactly as stated (including lower/upper case letters). Failure to do so will result in a mark of 0 being assigned. You may check the status of your submission using the command

```
submitaps105f -l 7
```

where `-l` is a hyphen followed by the letter *ell*. You can also download a copy of your submission by running the command:

```
/share/copy/aps105f/lab7/viewsubmitted
```