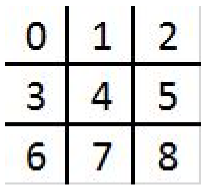
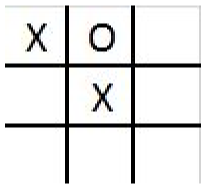
Lab 4: Games

The task in this lab is to create a TIC-TAC-TOE solver capable of predicting the result of a specific game when a board is provided. The solver has to use min-max to predict the outcome as if both players were playing optimally. Your program will receive an array of 9 (from 0 to 8) integers representing the squares of the tic tac toe board and a separate number indicating which player moves next (1 for X, 2 for O).



Indexes and positions in the board

The values you can find in the board are 0 for empty square, 1 for “X” and 2 for “O”. So this sample board:



Will be represented for the array: board=[1,2,0,0,1,0,0,0,0]

For this you are to complete two functions:

1. def minmax_tictactoe(board, turn) that returns an integer representing the winner of the game under optimal play, 1 for X, 2 for O and 0 for tie, using minmax to explore the <u>entire solution space.</u>
2. def abprun_tictactoe(board, turn) that returns an integer representing the winner of the game, 1 for X, 2 for O and 0 for tie, <u>using alpha-beta pruning to trim the solution space.</u>

Even though this functions only return the result of the game, it is important that you control the size of the space explored with the techniques explained in class. For this purpose we provided a function common.game_status(board) that counts the calls you make to evaluate different states of the game and a set of tests to give you a hint if you are hitting the mark, <u>you must use game_status() to look at every state expanded.</u>

Helper functions and constants provided:

- game_status(board) : receives a board and returns 1 if X wins, 2 if O wins, 0 for any other case (tie or incomplete game).
- get_cell(board, y, x) : receives a board and a coordinate and returns the value of the board in that coordinate.
- set_cell(board, y, x, s) : receives a board, a coordinate and a value and sets the square at that coordinate to the value.

- print_board(board): prints the board in the screen.
- Constants: X 1, O 2, NONE 0

No additional modules are to be used (don't import anything in your student code other than common). Run your code with Python3.

Required implementation:

- It is important that you follow the order of the squares when you explore the solution space to get to the expected results, this is : explore 0 first, and then 1, and then 2… and so on.
- You have to use game_status to check if a state is a terminal state, this is the way we observe how you explore the solution space. I.e. every node in the search tree will call game_status once.
- Use the exact implementation of Alpha-Beta pruning described in day 4 slides. Don't use additional tricks or knowledge about the game.

Considerations:

- We provide some boards to test your solution but grading will be done with another set.
- game_status() uses a global variable to count the boards explored, but the manipulation of that variable is completely forbidden (during the grading will not be available and your program will fail to compile if you use it anywhere)
- When game_status returns 0, it doesn't automatically mean that it is a tie! You also have to check if the board is complete!!
- The running time of your algorithm cannot be longer than 30 seconds for any board, otherwise it will fail the grading tests.
- All functions will be tested independently, so you will get credit for each one that returns the right results, but you have to make sure your program compiles and runs properly.
- Make sure you follow the academic honesty and plagiarism rules given on the first day of class and in the syllabus on canvas.