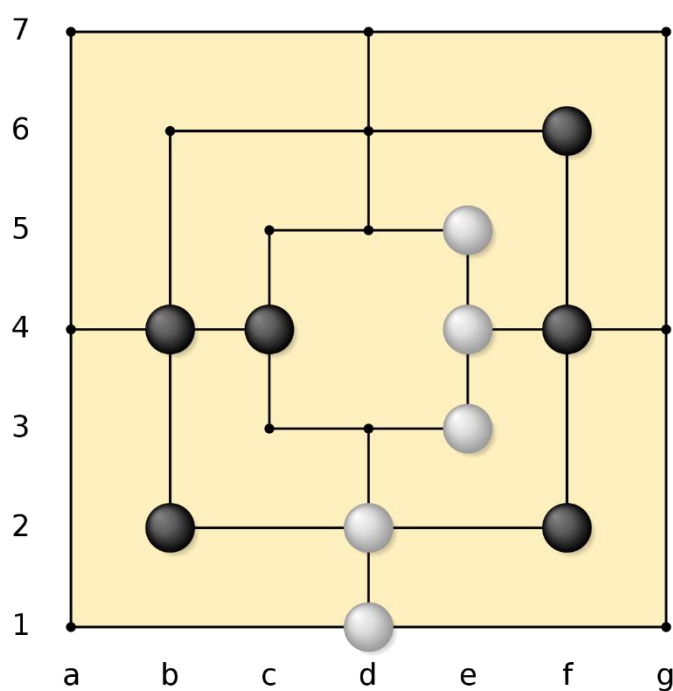


HW2

Nine Men's Morris



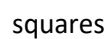
Inquiries to the staff regarding homework will be through the piazza only (there will be no response via email)

Link to piazza: piazza.com/technion.ac.il/winter2022/236501

In addition, before asking a question, one should check if it has been asked in the past. Questions asked in the past will not be answered

You must implement a variation of the game: 'Mill'. This is an abstract strategy game for two players, which has its roots in antiquity.

The game is played on a board containing 24 nodes (numbered 0-23) drawn on the board by 3 squares of different sizes connected by 4 lines coming out from the center of the sides of the



Each player has 9 soldiers (we count them 0-8)

The game is divided into two stages.

first step:

The game starts with a blank board. In the first stage, each player in turn places a soldier at a free intersection. When a player creates a sequence of three of his soldiers in one of the straight lines on the board (a sequence called a "mill"), he removes from the board a soldier of the opponent (if possible). This soldier is out of the game.

second level:

After each player has placed nine soldiers on the board, the second phase of the game begins in which the players move, one in turn, one soldier. A soldier is moved from the junction where he is to a nearby (*) free junction. Even at this point a player who has created a mill removes from the board a soldier of the opponent. A player left with less than three soldiers or unable to move when it is his turn loses the game.

(*) Adjacent junction: In the figure above, near junction 0 there are junctions 1 and 3. And near junction 9 there are junctions 1, 10, 8, 17.

The Purpose of the game

Leave the opponent with less than 3 soldiers (2 or less) or block the opponent from making moves.

Links

To read about the game: https://en.wikipedia.org/wiki/Nine_men%27s_morris

To play the game: <https://toytheater.com/nine-mens-morris/>

(Please note, when you read about the game / play the game, we will implement a simpler creation of the game, the third stage, in which one of the players has less than 3 soldiers is not relevant for us.

Running games

Before starting to compute the agents, it is recommended to experiment with the game. Run the following lines in the terminal in the folder where the code attached to the exercise is located:

To play as the two agents, run the line:

```
python main.py -player1 LivePlayer -player2 LivePlayer
```

After running the lines the game board will appear in the terminal and you can start playing.

To play against a SimplePlayer player (a player whose realization is in the code files you received) run:

```
python main.py -player1 LivePlayer -player2 SimplePlayer
```

Later we will implement the players MinimaxPlayer, AlphaBetaPlayer and more. To run a game between any two players run a similar line with the names of the corresponding players.

In addition you can set:

- Limit on each move time by the -move_time flag
- Global time limit for all player turns by the -game_time flag

Dry part (45 pts)

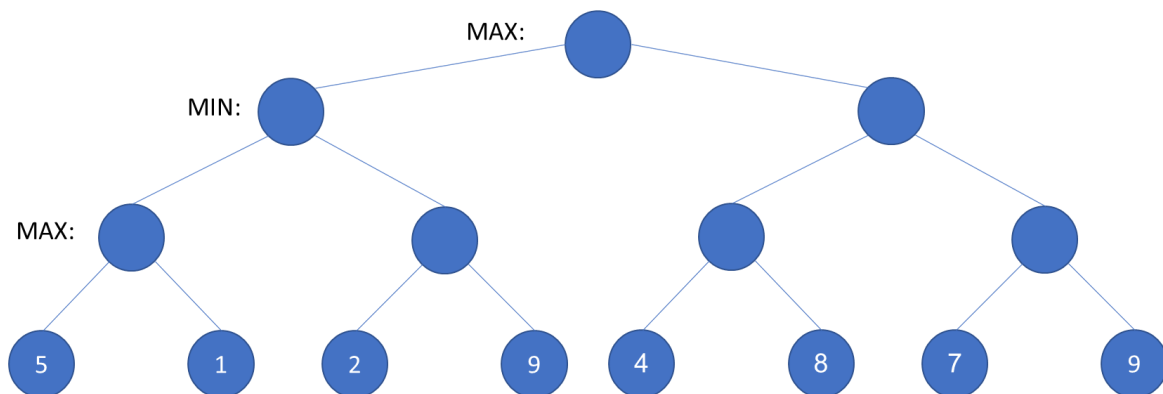
Questions about the heuristics

- (1.5 points) We will define the following heuristics:
Given state s ,
$$h(s) = \text{number of player incomplete mills} - \text{number of rival incomplete mills}.$$

The heuristic value of the state is the number of configurations in which a trio has 2 soldiers (of the agent) + an empty cell.
What are the disadvantages and what are the advantages of this heuristic?
- (2 points) Define a heuristic consisting a combination of at least four components. (Will help you get good results in the wet part). Formally present the heuristic in your document. Explain your motivation for defining heuristics, and a brief explanation of each of the parameters you defined in heuristics.

Questions about minimax and alpha-beta

- (2 pts) What is the advantage of adding alpha-beta pruning to the minimax algorithm, and how is this advantage achieved?
 - (2 pts) How would you sort the tree nodes in order to get maximum pruning?
 - (2 pts) The tree below represents a situation in the game. Now it's max's turn. Mark each leaf that is pruned by alpha beta.
In addition, mark the optimal route for the continuation of the game given the knowledge in the tree.



- (2 pts) True or False: If you use a alpha beta algorithm with heuristics, have you created an agent that plays with an optimal strategy? Give an example.
- non zero-sum game: Suppose some game in which there are finite states s_1, s_2, s_3, s_4 .
 - (2.5 points) Define a utility function for the final states that does not define a zero-sum game according to which Minimax is optimal.

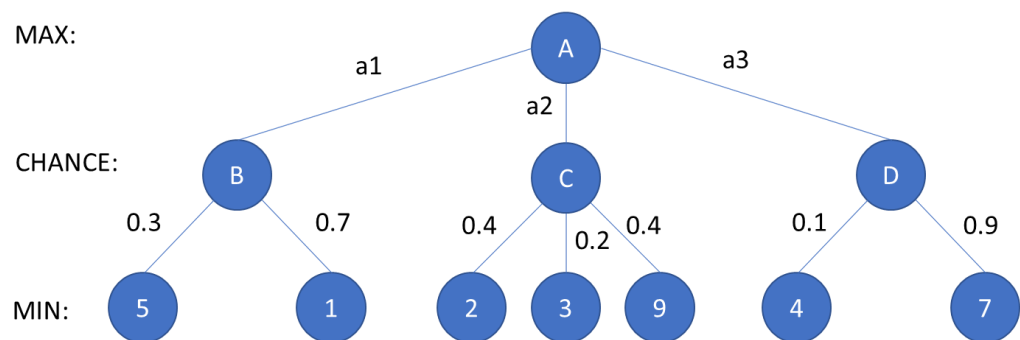
2. (2.5 points) Define a utility function according to which minimax is not optimal. Give an example of a search tree where the minimax algorithm does not fit.

6. A student implements the alpha-beta algorithm and the functions necessary for the 'mill' game. In a game she played against the software she noticed that although the computer could win the game in the next step, it refrained from taking that step and chose another step. The student checked and found that all the implementation is correct and there are no bugs in the software.

A. (2.5 pts) How is such a situation possible? Explain in detail.

B. (2.5 pts) Offer a change to alpha beta that will prevent such situations.

7. A. (2 points) Calculate the Expectimax values of nodes C, B, D. (Show your calculation.)



B. (2 pts) Which action MAX will choose? a1, a2, a3?

C. (2 pts) Pruning in the expectimax algorithm.

Can we trim in expectimax in the same way we want to trim in an alpha beta algorithm without compromising the correctness of the algorithm? An example should be given.

8. We want the alpha beta algorithm to prune more nodes and guarantee us a correct answer that deviates from the exact answer at most ϵ . Therefore, we will add the following change to it:

For each state s in the game, $|\text{Algorithm}(s) - \text{minimax}(s)| \leq \epsilon$

A. (2.5 points) The following is a pseudo code for their alpha beta algorithm:

```

alphaBeta(state):
    return maxValue(state, -INFINITY, INFINITY, 0)

maxValue(state, alpha, beta, depth):
    if cutoffTest(state, depth):
        return utility(state)

    value = -INFINITY
    for successor in state.getSuccessors():
        value = max(value, minValue(successor, alpha, beta, depth + 1))
        if value >= beta:
            return value
        alpha = max(alpha, value)

    return value

minValue(state, alpha, beta, depth):
    if cutoffTest(state, depth):
        return utility(state)

    value = INFINITY
    for successor in state.getSuccessors():
        value = min(value, maxValue(successor, alpha, beta, depth + 1))
        if value <= alpha:
            return value
        beta = min(beta, value)

    return value

```

Add a change to the code to get the requested behavior.

B. (2.5 points) Give an example for which the new algorithm returns a value different from the minimax value.

9. Before an important competition in the game Nine men's Morris, one of the players bribed the referee and managed to achieve the rival_move decision procedure of the opponent. The procedure gets a state s in the game, calculates the step the opponent will choose, and returns the state to which it will move.

Each player is assigned a fixed time T per turn in the game and it allows to do a minimax search to depth $d1$. The time it takes to run the rival_move procedure is negligible and weighs zero.

A. (2.5 pts) What depth $d2$ can we look for in time T if we use the rival_move procedure?

B. (2.5 pts) Given state s , what is the ratio between the value of the minimax with the use of the procedure and the value of the minimax without the use of the procedure when both are limited to depth d ?

Local Search Questions

10. A student was given a maximization problem with a 10^{12} search space. He chooses to solve the problem with the help of SAHC. He runs the algorithm 1000 times, each time he decides to start from a different random point. In all 1000 runs of the algorithm, the lowest value it received was 0.9, the highest 5.8 and the average was 3.2. It takes the algorithm 5 steps on average to converge and return a result. Because the student is satisfied with his search he reports that the optimum is at 5.8.
- A. (2.5 points) Is the student right that he reports that 5.8 is the global optimum?
- B. (2.5 points) If you had to verify the student's answer by further experiments. Which algorithm would you prefer to use?
- C. (2.5 pts) Create a search space where SAHC does not find an optimal solution with a high probability, but, the algorithm you chose in section B will find a high probability solution.

Wet part (55 pts)

You will need to exercise agents who play the game.

Each player inherits from the AbstractPlayer class.

In the **AbstractPlayer** file, there is a class named Player, which has the following functions:

Please read them and their role carefully!

1. Init- Boot the player. Gets a game_time parameter which is the total time for all player turns.

Initializes the board to be an array of 24 lengths, and initiates a directions function which, given a soldier's location, returns the adjacent cells to which the soldier can move.

2. set_game_params - Set the game parameters needed by this player.

This function will be read by the code that runs the game once for each player at the beginning of the game, and its purpose is to pass to the players the board where the game will run (the starting board is always empty).

Parameters: The board of the game, this is a 24 length np.array where each number represents a slot on the board.

During the game the board will change and each entry in it will contain one of the following values:

```
1 (00)----- 2 (01)----- 0 (02)
|
|
|
|
| 0 (08)----- 0 (09)----- 0 (10) |
|
|
|
|
| 0 (16)----- 0 (17)----- 0 (18) |
|
|
|
|
0 (03)- 0 (11)--- 0 (19)          0 (20)- 0 (12)--- 0 (04)
|
|
|
|
| 0 (21)----- 0 (22)----- 0 (23) |
|
|
|
|
| 0 (13)----- 0 (14)----- 0 (15) |
|
|
|
0 (05)----- 0 (06)----- 0 (07)
```

-The value 0 denotes an empty square. (In the illustration above 2-23 empty squares)

- The value 1 symbolizes that in this box there is a soldier of player number 1. (Box 0 in the illustration above)

- The value 2 symbolizes that in this box there is a soldier of player number 2. (Box 1 in the figure above)

1. **make_move**- This function will be called by the code that runs the game in each player's turn, and its purpose is to get the next step from the player.

The function accepts as a parameter the time, `time_limit`, that the player has to make his turn, in case he exceeds this time frame the game will end and he will lose the game.

The function output is the following tuple: (`pos`, `soldier`, `dead_opponent_pos`)

`pos` - A position on the board where the player wants to put a soldier. (int between 0-23)

`Soldier` - which soldier the player chooses to put in this position (int between 0-8). (You choose which number each soldier).

(Note, this must be a soldier who can reach this position. This means that in stage 1 of the game a soldier has not yet been placed, and in stage 2 of the game, a soldier who is in a box next to this soldier.)

`dead_opponent_pos` - If the player has created a mill, returns the cell from which the player wants to remove an opponent's soldier. If the player has not created a mill, he must return "-1".

2. **set_rival_move** - This function will be called by the code that runs the game after each turn of the opposing player, and its purpose is to update your player in the step performed by the opposing player.

The function receives as a parameter the move `move` which is the tuple: (`rival_pos`, `rival_soldier`, `dead_my_pos`) which refers to the step taken by the opponent.

3. **isMill** - A function that checks if the player has created a mill (the mill function gets the cell where the player wants to check if he has created a mill and board (optional, see code).

Note that your board must be updated before calling this function.

The `checkNextMill`, `isPlayer` functions are auxiliary functions of this function.

Note: The `isMill` function is implemented.

You can add additional parameters and functions to this file.

In the **SearchAlgos** file:

Pay attention! This file is yours. You can change the details of the existing functions in this file and add any class you want.

A class called `SearchAlgos`. And has an `init` function.

1. **Init** - Receives:

`utility` - a utility function

`succ`- The follower function

`perform_move` - a function that performs a step (most of you probably will not use this parameter and it is initialized to none)

`goal`- A function that checks if we are in goal state

2. **Search**- Receives:

`State` - current state

`depth`- A depth limit of the minimax algorithm

`maximizing_player`- Boolean value if I am the maximizer its value will be true.

The `Minimax`, and `AlphaBeta` are inherited from `SearchAlgos`

Part A - Compute Minimax player

In this section a resource-limited Minimax player needs to be built in the Anytime contract variation. You must compute the player in the MinimaxPlayer.py file, in a class named Player.

The player you have has to implement the functions `set_game_params`, `make_move`, `set_rival_move`.

The `__init__` function receives a `game_time` parameter that this player does not use. The Minimax algorithm must be implemented in the `SearchAlgos.py` file under the Minimax class.

`set_game_params` - As we explained above.

`set_rival_move` - As we explained above.

`make_move` - This function will be called by the code that runs the game in each turn of your player and aims to get from your player the next step he takes.

Note that the `make_move` function has a time limit, `time_limit`, that the player must make his turn, if he exceeds this time frame the game will end and he will lose the game. A good way to deal with the time limit is through iterative deepening.

The Minimax algorithm must be implemented in the `SearchAlgos.py` file within the Minimax class.

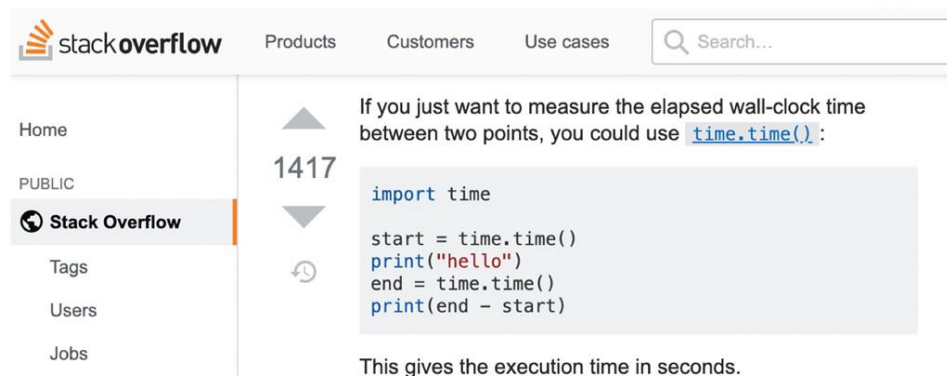
Utility function – heuristics

As we learned in the lectures, A Minimax agent with limited resources uses heuristics to assess the quality of the leaves in the tree he has developed and to determine the game strategy according to these values.

You must define and implement the heuristics for the Minimax player, the heuristics must consist of at least 4 components, ie you must define at least 4 values that are a function of the situation we want to evaluate, so that the heuristics will be composed of their combination.

tip

To deal with the time limit, you can use the time library:



The screenshot shows a Stack Overflow page. The left sidebar contains navigation links: Home, PUBLIC, Stack Overflow (selected), Tags, Users, and Jobs. The main content area shows a question titled "If you just want to measure the elapsed wall-clock time between two points, you could use `time.time()`:". Below the title is a code block with the following Python code:

```
import time

start = time.time()
print("hello")
end = time.time()
print(end - start)
```

Below the code block, it says "This gives the execution time in seconds."

Part B - Compute α - β Agent (12 pts)

In this section we will improve the Minimax agent we built to the α - β agent.

The player must be computed in a file named `AlphabetaPlayer.py` in a class named `Player`.

The player can be the same as the Minimax player, except that he will implement the α - β algorithm.

The `__init__` function receives a `game_time` parameter that does not use this player, so this array must be ignored.

The player must implement the `set_game_params`, `make_move`, `set_rival_move` functions as described above, and implement the alpha-beta algorithm in the `SearchAlgos.py` file under the `AlphaBeta.py` class.

Part C - Compute α - β Agent with Global Time Limit



Instead of a time limit for each turn, there is a general time limit for all player turns together, so a strategy must be devised that will spread the time wisely across the player turns.

The player must be implemented in a file named `GlobalTimeAB.py` in a class named `Player`.

The `__init__` function receives a `game_time` parameter which is the total time for all player turns.

The player must implement the `set_game_params`, `make_move`, `set_rival_move` functions as described in Part B, except for the change in the `make_move` function which refers to the global time and not the `time_limit` parameter.

To test this agent you have to set in the row that runs the game, the `game_time` parameter in different sizes, and the `move_time` parameter so that it is greater than or equal to the `game_time` parameter.

Part D – Competition



Each pair is required to compute a single player which will represent them in the competition, in which all pairs will participate. Students whose players excel in the competition will receive a bonus for the final grade of the course:

5 points for the winning pair, 3 points for second place, and 2 points for third place.

The competition is defined so that each player has a global time for all the turns together, as defined in section D.

Your competition player must be computed in the `ContestPlayer.py` file, in a class called `Player`.

The class must implement the functions, `make_move`, `set_rival_move`.

You can submit the player you exercised in Part D or exercise a new player.

If the player you submit does not run, you will lose points in the exercise. Participation in the competition is mandatory.

Rules:

- A player may not use the Internet or any local network.
- The player must not use any type of parallel code.
- Any player who tries to cheat or disrupt the opponent's code or system will be disqualified, and points will be dropped to the submitters in the exercise.
- It is forbidden to use information that is pre-processed and stored in the file.
- It is forbidden to use packages / directories that are not understood in Python, except for `networkx`, `collections` and the packages in which import is performed in the code provided to you.
- Any calculation to determine the agent's next step should be made solely by calling the `make_move` function.

Part E - Writing a Report

Answer the sections below for questions in the dry section

1. (2 pts) Describe the heuristics you set for the Minimax player, explain.
2. (2 pts) Explain how your competition player works. If you have defined a new heuristic function, explain it.
3. (2 pts) Describe how you managed the run time of the make_move function. For two time limit cases.
(Limited turn time and global limited time).



Part VI - Performing experiments



Add your answers to the following sections of the report:

4. (1 points) Run games between the Minimax Agent and the AlphaBeta Agent (with a time limit for each turn) that you have implemented. What results did you get? Do the results match your expectations (each game should have a different time limit)?

In-depth comparisons between different players:

5. (4 pts) Define two new players, LightABPlayer.py, and HeavyABPlayer.py, without reference to time constraints, and must search at a fixed depth. The players' heuristics will be defined as follows: HeavyABplayer will be defined as a player with sophisticated heuristics (you can use the heuristics you have implemented for your player)
LightABPlayer will be set to be a player with simple heuristics.

You must perform the following experiment:

The HeavyABplayer player must be set to depth 3 regularly, with the search depth in the LightABPlayer player changing at each stage of the experiment.

There are 3 stages, where in stage i the depth of search for the player LightABPlayer will be $i-1 + \text{search_depth}(\text{HeavyPlayer})$. That is, in the first stage a search depth of three.
In each stage of the experiment, the stage score should be set to be the number of HeavyABplayer wins.

The experiment results should be displayed in the graph, with the x-axis set to be the difference between the search depths of LightABPlayer and HeavyABplayer and the y-axis set to be the step score.

The experiment should be performed again for a HeavyABplayer 2 depth search (now LightABPlayer will search for depths 2,3,4)

Explain the results of the experiment, is there a difference between the results of the two experiments? If so what do you think could be the reason for this?

Submission Instructions

All files (code and report) must be submitted in a single zip file named AI2_<id1>_<id2> where instead of <id1> and <id2> the ID numbers of the submitters must be entered.

Except for the report, the zip file needs all the code files of the players you wrote (all players except LivePlayer.py, AbstractPlayer.py SimplePlayer.py,), and the code files SearchAlgos.py and utils.py.

When the folder hierarchy exists in the zip file (the players in the players folder, and the SearchAlgos.py and utils.py files outside the players folder