Ex2 Introduction to AI

Gobblet Gobblers

(enhanced TicTacToe)

Introduction and Administration

General instructions

- date of submission: **5.1.2023**
- The assignment must be submitted in pairs. special requests should be approved by the teaching assistant in charge (Sapir Tuble)
- The assignment must be submitted typed only - a handwritten solution will not be checked.
- The answers should be written in Hebrew or English.
- You can send questions about the exercise through the piazza.
- Teaching assistant in charge of this exercise: **Ofek Gottlieb**
- Justified late submissions should be sent only to the teaching assistant in charge (Sapir Tuble)
- During the exercise there may be updates - a message will be published accordingly in this case.
- The updates are mandatory and it is your responsibility to stay up to date on them by the time the exercise is submitted. Updates will appear in yellow in the form.
- Copying will be dealt with severely.
- The assignment score includes a dry part and a wet part. In the dry part we will check that your answer is correct, complete, legible and organized. In the wet part, your code will be comprehensively tested using automatic tests - which will compare your implementation with ours. It is important to carefully follow the instructions of the exercise since deviations from the requested implementation may lead to a failure in the automatic test (even if the implementation is "mostly correct"). During the automatic test, a reasonable time will be given for each run - so as long as you follow the instructions, your implementation will meet the time restrictions and return results good enough for the test.
- It is recommended to look at the code yourself. Basic questions about Python that do not pertain to the exercise should be checked online before you ask in Piazza. It is recommended to read the given code in order to understand how it works - in case there are things that are not understood. (For this purpose there are many comments and even an extended explanation about the environment!)
- It is recommended not to postpone the exercise to the last minute since the implementation and writing of the report may take longer than expected.

- References in the masculine, feminine or plural refer to all genders.
- **The maximum score for the exercise is 110 because there is a bonus - details about the bonus at the end of the wet part**

Submission instructions

inside a zip file with the name: HW2_AI_id1_id2.zip
the dry report in the format: id1_id2.pdf
and the file: subbmision.py (where you implement your algorithms)

Introduction with the game

Board and Pawns

a grid board of size 3x3
two players
each player has six pawns
2 of size small
2 of size medium
2 of size big

The game, in general:

An enhanced version of TicTacToe, the winning player is the one who managed to fill a row, column or diagonal with his marks (the size does not .(matter
Players can devour other players and thus change the color of an existing .square
Below is a video that helps to understand the essence of the game in general:

How to Play Gobblet Gobblers

The game specifically:

- Each player in his turn places a goblin on the board, either on an empty square or on a smaller goblin
- Instead of placing a new goblin on the board, a player in his turn can choose his own goblin that is on the board and move it to a valid location
- A goblin can devour any goblin smaller than it (even if they are the same color)
- 3 goblins can be placed on top of each other (large on medium with the medium on small)
- Important emphasis: in the original game there is no step limit and in our version we limit the number of turns in the game. (If there is no victory and we exceed the limit of steps, we will consider it a draw)
- Important emphasis: in the original game it is important to remember what is under each goblin on the board, we always know (the computer has a good memory 🙂)

Here is the official rule book of the game: Gobblet gobblers rules

So, what did you get?

You got four main files:

1. Gobblet_Gobblers_Env.py - the game's environment. Will be explained later in the document.
2. submission.py - the file where you implement your agents - and the one which you are submitting.
3. game.py contains two functions for self-testing which we will elaborate on later in this document.
4. main.py also used for self checking

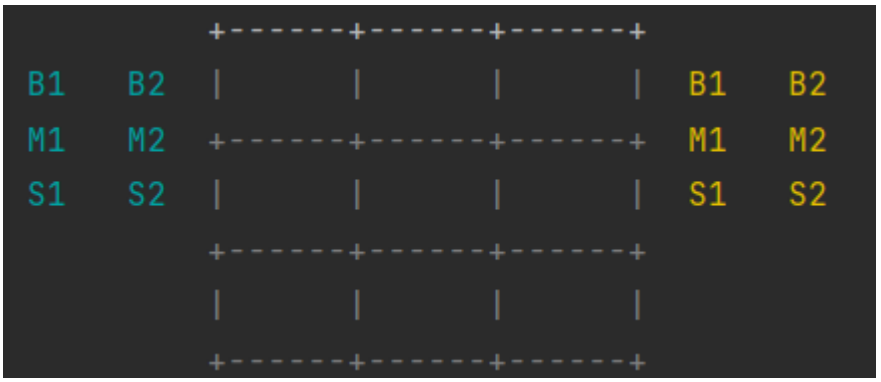
Getting to know the environment

The environment you will work with is implemented in the file Gobblet_Gobblers_Env.py. It is based on a gym environment like the environment you worked with in hw1.

It contains the following methods related to the functioning of the environment:

- **init()** - The constructor of the environment (class). It can be used as follows: `env = GridWorldEnv()`

- **reset()** - Resets the game board. Put all the tools on the sides and the game board is empty. It is used as follows. `state = env.reset()` The function returns a state (to understand what it means to return a state, read the details on the get_state() function), below is the board in the initialized state:



*The letters represent the player's size (B - large, M - medium, S - small)

- **step()** - A function that receives an action and performs the action. The action should be a tuple in the format (pawn, location) where pawn represents the player you want to move from the options {B1,B2,M1,M2,S1,S2} and location the place on the board where you want to place the piece. location is a number between 0 and 8 that represents the slot on the board as follows:

An example of using the function `env.step(("B1", 7))`

Meaning: we will move goblin B1 to tile number 7.

- **render()** - Prints the game board example of using the function `env.render()`
- Note, pawns that are on top of other pawns will hide the pawns below them, to know exactly where each pawn is see ()get_state.
- Note that a pygame window will open for you with convenient graphics of the game where you can see the status of the board after you take (more precisely your agent will take 🙂) steps, in addition there are printouts for the console that you can use for debugging.

* There are many more internal functions, you are welcome to read their description, some of them can even help you in writing heuristics later, but there is no need to delve into them.

In addition, the environment contains the following functions related to its integration with the agents you are going to implement in this exercise:

get_state() - A function that returns the current state of the environment, it is used as follows: `env.get_state()`

And the form in which the state of the environment is returned is in the structure of State (which is defined in the file Gobblet_Gobblers_Env.py).

get_neighbors() -A function that accepts a state of type State and returns a list that contains all its neighbors (all the states that will be created from all the legal actions that can be performed on that state) in the list basically each member is a tuple of (action, state) so you can go through the states and easily return the action for the state you choose from a list the neighbors

is_final() - Confusing so it's important to pay attention! A function that receives a certain state and returns:

None – for a non-finite state.

0 - If there is a tie, if both players won the same step (normal situation) or if 100 turns have passed (50 for each player) there is no decision.

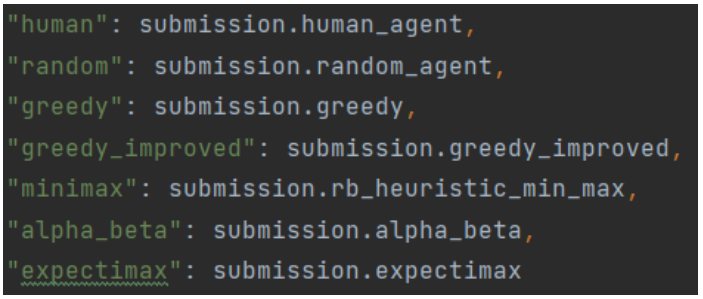
1 - The first player wins. pay attention! During the game, the first player's turn is indicated as 0 and not 1!

2 - Second player wins.

- **play_game()** - A function that runs a single game. The above function receives strings that describe each agent as described in the photograph of the dictionary in the following function. In

addition, there is an example below. The function returns who the winner is and prints it to the screen. Returns results like is_final() returns.

- **play_tournament()** - A function that runs a number of play_games according to the num_games value you give it (we will test your code with num_games = 50) and thus returns in percentages how many wins each player has and how many draws there were in the game. The above function accepts strings that describe each agent as described in the photograph of the above dictionary. Example later in the document.



* num_games is not the number of games that are actually running, actually 2 times more games are running. It represents how many games there are in which each player is the first player. (turn to play first)

Let's start writing code!

Part A - getting to know the environment (4 points)

1. (dry: 2 points) Go to the main file and run the line that is marked with a comment and above it is written PART1# The line runs the game with two human agents, play against each other until victory and attach the printout of the final state.
2. (dry: 2 points) Is it possible to cause a certain agent's turn to perform an action that will make the other agent win? If so, explain why such a situation can happen and include the last two steps in such a game, if not, explain why such a situation is not possible.
3. (dry: 1 points) What is the maximal number of options for different actions (assuming each action is valid) that can be done in a turn?

Part B - Improved Greedy

(15 points)

In the file submission.py that you got we implemented for you a greedy agent that uses a simple heuristic implemented in the `dumb_heuristic2` function, the heuristic returns the number of pawns that the player has that aren't hidden.

*pay attention! A heuristic called dumb_heuristic1 is implemented in submission, we don't use it anywhere, but what it does is: returns 0 if the state is not a final state. Returns 1 if we won and 1- if we lost or there was a draw. You are welcome to look up to understand how to use is_final().

1. (dry: 6 points) Define your own heuristics for evaluating game situations, write an explicit formula for heuristics. Feel free to add a diagram or a detailed breakdown of what the heuristic does given the state of the board and the player playing. Choose clear and meaningful names in the explanation.
2. (wet: 5 points) In the file submission.py implement a greedy agent in the function `greedy_improved` and implement your smart heuristic under the function `smart_heuristic.`(don't change the

signatures of the functions it will fail the tests!)
greedy_improved is a function that gets:

- curr_state - current state of the board in the struct of State
- agent_id - the player that plays right now - If this is the first player 0 will be transferred if this is the second player 1 will be transferred.
- time_limit - The time limit (at this point you can **ignore it**, it will be relevant in the following algorithms).

The function returns the action that the agent should choose to perform as a tuple (pawn, location) as detailed in the explanation of the environment (same as the input of the step() method).

3. (dry: 2 points) Explain the motivation for the changes you made to the heuristics Do you think the agent is greedy based on your heuristics (greedy_improved) will defeat the greedy agent? If you answerd yes explain why.
4. (wet: 2 points) Run the lines that are in the comment and above them PART2# and attach the results that will be printed to the screen as a result. You will actually run:
    - greesdy against random
    - greedy_improved against random
    - greedy against greedy_improved

Part C - RB heuristic MiniMax
(20 points)

1. (dry: 2 points) What are the advantages and disadvantages of using an easy-to-calculate heuristic versus a difficult-to-calculate heuristic given that the hard-to-calculate heuristic is more informed than the easy-to-calculate heuristic (gives better information regarding the question of what is a good situation)? Given that we are in min-max limited resources.
2. (dry: 3 points) Danny implemented the RB-heuristic-MiniMax algorithm and ran it from state s where there is a single action that results in a win. Danny was surprised to find that the algorithm did not choose this action. Is there necessarily a mistake in the algorithm that Danny wrote? If so, explain what the error is, otherwise, explain why the algorithm worked this way.
3. (dry: 3 points) You learned in lectures and tutorials an approach called anytime search. How does that approach deal with the time limit? What is a common problem in the last iteration and how is it solved?
4. (dry: 4 points) Let's say there were K players instead of 2 (think of a general game not only our game, but still a zero sum game). What changes will need to be made in implementing the Minimax agent?
    a. Given that every agent wants to win and doesn't care only about you
    b. Assuming that every agent hates you and the only thing he cares about is that you don't win

5. (wet: 8 points) you should implement the function `rb_heuristic_min_max` in the file submission.py. Note that the agent has limited resources, with the time_limit variable limiting the number of seconds the agent can run before returning an answer. (In this section it is forbidden to use pruning to solve the problem - don't worry in the next section you will have pruning).

* It is recommended to run your agents against each other as in part A and see that you get reasonable results

*notice that the time limit has changed, from 1 sec to 80
*the changes are in file game.py

Part D - Alpha_Beta
(20 points)

1. (wet: 10 points) You are now implementing an alpha beta agent, in the submission.py file implement the function alpha_beta so that it performs pruning as learned in the lectures and exercises.
2. (dry: 3 points) Will the agent you implemented in part D behave differently from the agent you implemented in part C in terms of running time and choice of moves?
3. (dry: 2 points) You have learned several methods for alpha_beta improvements one of these methods is called opening / closing libraries. What does this method do and why does it help to prune a piece of the tree?
4. (dry: 3 points) You learned about another approach called status tables. What does the method do, how does it maintain correctness, and why does it help to "prune" the tree?
5. (dry: 2 points) When running Alpha-Beta Minimax, the programmer mistakenly deleted the recommended step and was left with only the minimax value of the tree. How can you rerun the algorithm efficiency by improved simple changes to the algorithm? Describe the behavior of the modified algorithm. Explain how it will behave in the best case, the worst case, and the general case.

Part E - Expectimax
(20 points)

1. (dry: 2 points) A Minimax agent (with or without improvements) assumes that the adversary chooses at each step the optimal action for him, what is problematic about this approach and how does the Expectimax algorithm overcome the problematic you described?
2. (dry: 3 points) Assuming you use Expectimax algorithms against an agent who plays completely randomly, what probability will you use? And why?
3. (dry: 5 points) For probabilistic games such as backgammon, in which there is a resource limit, the RB-Expectimax algorithm is used. We assumed that it is known that the heuristic function h in the Expectimax-RB algorithm satisfies $\forall s: -6 \leq h(s) \leq 6$
    a. How can this algorithm be pruned? Describe in detail the conditions of exaggeration, and explain the idea behind it.
    b. Present an example of such a heuristic for the game in our exercise and attach an example to the board for each of the following situations:
        - state s such that $h(s) = 6$
        - state s such that $h(s) = -6$
4. (wet: 10 points) Now you will implement an expectimax agent by implementing the expectimax function in the submission.py file, under the assumption that our opponent likes action and decides that he performs all actions with equal probability except for two types of actions:
    a. Actions where another soldier can eat have a 2x higher probability than other actions
    b. He prefers small soldiers (of size S) and therefore an action involving player S also has a 2x higher probability than the other actions

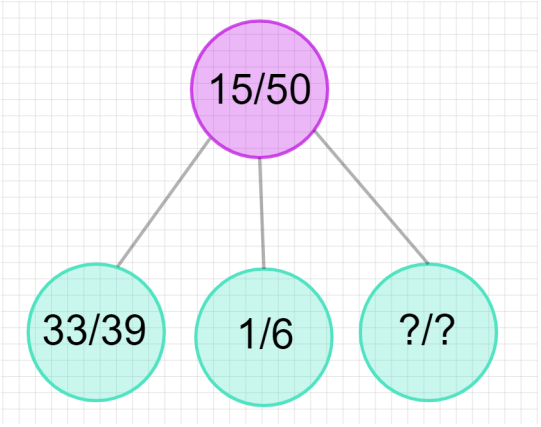
Bonus Bonus Bonus Bonus Bonus

As you know, the algorithms you implemented are adversarial, which means they compete with each other, so we invite you to write an agent that will compete with the agents of the other students in the course. You must implement it under the supre_agent function. The two pairs that win against the most other pairs will receive 10 bonus points for the final grade of this assignment. The four pairs after them will receive a 5 bonus points.

Open question - Monte Carlo Tree Search
(21 נק')

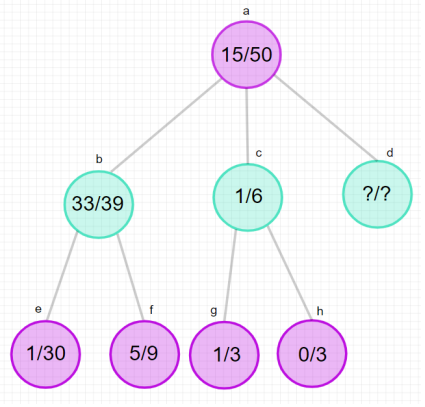
1. (2 points) explain the following terms:
    a. exploitation
    b. exploration

Andrey and Ofek decided to play Catan (let's say it's a game for two people), they decided that the game was too simple and decided to add the expansions: "Cities and Knights" and "Seafarers".
After they realized that they overdid it and now they are not sure what the smartest step is and because the branching factor is very large they decided to use MCTS (Monte Carlo Tree Search) (emphasis - this is a zero sum game that ends in the victory of one of the parties)
Below is a tree that describes two stages of the game. Blue is Ofek's turn and pink is Andrey's turn (less relevant for the first two questions)



2. (2 points) For the above tree that describes two stages in the game for the leaf with the (?) complete what the values should be in place of the question marks.
3. (7 points) Calculate the UCB1(s) for the blue nodes with a value of C = 2, compare the values you got in the UCB1(s) calculation, explain what they mean. Are there nodes (only from the blue ones) whose value is fundamentally different from each other but the UCB1(s) are similar? If so, explain why such a situation happens according to the principles of UCB1.

Now they reveal to us a continuation of the two stages we saw and the tree now looks like this:



4. (4 points) Find and specify who is the next node to be developed
5. (6 points) Assuming there is an only child from whom, after simulation, a loss of Ofek is obtained. Draw the newly created tree (change the values of the other nodes accordingly) after flashing the information about the simulation that took place.