

# Introduction to Artificial Intelligence - 236501

## Homework #2: Adversarial Games



| Name         | ID        | Email                           |
|--------------|-----------|---------------------------------|
| Matan Kutz   | 205488679 | matankutz@campus.technion.ac.il |
| Yotam Granov | 211870241 | yotam.g@campus.technion.ac.il   |



Faculty of Computer Science  
Technion - Israel Institute of Technology  
Winter 2022/23

## Question 1 - Getting to Know the Environment

### Part 1.1

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.

#### Solution:

Here is our printout for the final state for this part:

```

+-----+-----+-----+
|      |      | M2  | B2  |
+-----+-----+-----+
S1  S2 | B1  | B1  |      |
+-----+-----+-----+
|      |      | B2  | S1  |
+-----+-----+-----+
time for step was 10.836672067642212
winner is: 1

```

### Part 1.2

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.

#### Solution:

Yes, it indeed is possible to cause a certain agent's turn to perform an action that will make the other agent win, and this can happen when the agent picks up one of their pieces under which there is a piece of the other agent, which then grants a victory for that other agent. For example, the last two moves of a game could end this way (where blue's move immediately causes yellow to win):

```

player 1
insert action
insert pawn: B2
insert location: 0
+-----+-----+-----+
|      |      | B2  | M2  |
+-----+-----+-----+
S1  S2 | B1  |      |      |
+-----+-----+-----+
|      |      |      |      |
+-----+-----+-----+
time for step was 10.456164360046387
player 0
insert action
insert pawn: B1
insert location: 8
+-----+-----+-----+
|      |      | B2  | M2  |
+-----+-----+-----+
S1  S2 | B1  |      |      |
+-----+-----+-----+
|      |      |      |      |
+-----+-----+-----+
time for step was 5.637104511260986
winner is: 2

```

### Part 1.3

What is the maximal number of options for different actions (assuming each action is valid) that can be done in a turn?

#### Solution:

The maximum number of options for different legal actions in a turn is  $6 \cdot 9 = 54$ . The maximum number of pieces an agent can choose is 6, and the maximum number of places on the board the piece can move to is 9. Such a situation exists at the beginning of the game.

## Question 2 - Improved Greedy

### Part 2.1

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.

#### Solution:

We will define our heuristic as follows:

$$smart\_heuristic(state, agent\_id) = \begin{cases} 10 & \text{Winning state for agent} \\ 0 & \text{Tie} \\ -10 & \text{Losing state for agent} \\ two\_together\_agent - two\_together\_other & \text{Otherwise} \end{cases}$$

where we define the auxiliary parameters:

- **two\_together\_agent**: The total number of rows, columns, and diagonals in which the agent has 2 pieces
- **two\_together\_other**: The total number of rows, columns, and diagonals in which the other agent has 2 pieces

### Part 2.2

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).

Solution: See `submission.py`

## Part 2.3

Explain the motivation for the changes you made to the heuristics. Do you think that the greedy agent based on your heuristics (`greedy_improved`) will defeat the original greedy agent? If you answered yes, explain why.

### Solution:

First, the heuristic gives a value of 10 to a winning situation, this is a higher value than any non-winning situation, and likewise with -10 to a losing situation. This change ensures that the agent will choose a winning move and avoid a losing move if they exist. In any non-finite situation, the heuristic will evaluate how beneficial the situation is according to the number of options for the agent to obtain triplets (three pieces in a row) and likewise for the other agent, where the motivation is to increase the number of victory options for the agent and decrease the number of victory options for the opposing agent. Therefore, we believe that a greedy agent based on our heuristic will beat the original greedy agent.

## Part 2.4

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:

- `greedy` against `random`
- `greedy_improved` against `random`
- `greedy_improved` against `greedy_improved`

### Solution:

Our results appear as follows:

- `greedy` against `random`

```
ties: 1.0 % greedy player1 wins: 94.0 % random player2 wins: 5.0
```

- `greedy_improved` against `random`

```
ties: 0.0 % greedy_improved player1 wins: 99.0 % random player2 wins: 1.0
```

- `greedy_improved` against `greedy_improved`

```
ties: 0.0 % greedy player1 wins: 0.0 % greedy_improved player2 wins: 100.0
```

## Question 3 - RB Heuristic MiniMax

### Part 3.1

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.

### Solution:

Using an easy-to-calculate heuristic can be advantageous given that the heuristic can often be calculated much faster than a difficult-to-calculate heuristic. The same consideration explains why the

difficult-to-calculate heuristic can be disadvantageous, given that it has a more temporally-expensive nature (and our resources are limited).

Using an easy-to-calculate heuristic can be disadvantageous given that it's often not as informed as a difficult-to-calculate heuristic might be, and this can diminish the quality of its results. The same consideration explains why the difficult-to-calculate heuristic can be advantageous, given that it has a more informed nature.

### Part 3.2

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.

#### Solution:

While we might initially posit that there is a mistake in Danny's algorithm, it turns out that it is indeed possible that his algorithm was valid and simply made a poor decision (in the sense of achieving victory). This type of phenomenon can occur if the heuristic that Danny defined does not prioritize victories, but rather is attempting to optimize some other parameter. This is the case, for example, with the `dumb_heuristic2` function given in our code, which gives greater value to states where the agent has a larger number of uncovered pieces on the board - not to states which are victorious. In this case, our algorithm might very well prefer states that are not victorious even when there exists a state that guarantees immediate victory - we must be cautious not to define heuristics that do this, if we are prioritizing victories for our agents.

### Part 3.3

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?

#### Solution:

In the anytime search approach, we return the best move computed within  $t$  seconds, rather than just the best move in general. We allow our MiniMax algorithm (or any of its variants, like Alpha-Beta MiniMax or RBH-MiniMax) to run as many iterations as it can (with increasing depth values) until  $t$  seconds, at which time it will return the solution obtained by the last completed iteration before the time ended.

This approach allows us to hardcode time constraints into our algorithm (and thus deal with time limits), all the while providing a bonus in that the error of evaluation functions will have less impact on our results as our search goes deeper and deeper in the game tree.

One problem with this approach is that, on average, the algorithm will terminate in the middle of the last iteration, and this can be a huge waste of resources (especially for games with large branching factors). While we could just take the value from the previous iteration, another possible solution to the last-iteration problem would be to order the children according to some heuristic, and then in the last iteration expand only the "heuristically-better children" while using the evaluation function for the rest.

### Part 3.4

Let's say there were  $K$  players instead of 2 (think of a general game and 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?

**Solution:**

(a) In the first case, we have  $K$  agents and every one of them just wants to win, and doesn't care only about making us lose. In such a case, we can modify our MiniMax algorithm such that instead of conducting repeating iterations of two layers (where one agent tries to maximize the utility and the other agent tries to minimize it), it will conduct repeating iterations of  $K$  layers. In each such layer, the  $i$ -th agent will try to maximize its own utility, and will not try to minimize that of any of the other  $K - 1$  agents.

(b) In the second case, we have  $K$  agents and every one of them hates us and only cares about ensuring that we don't win. In such a case, we can modify our MiniMax algorithm once again such that instead of conducting repeating iterations of two layers (where one agent tries to maximize the utility and the other agent tries to minimize it), it will conduct repeating iterations of  $K$  layers. In this case, though, for each such layer corresponding to an agent that isn't us (there are  $K - 1$  such agents), that layer's agent will try to minimize our utility and will not try to maximize its own utility. In our layers, we will still try to maximize our own utility.

**Part 3.5**

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).

**Solution:** See `submission.py`

**Question 4 - Alpha-Beta****Part 4.1**

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.

**Solution:** See `submission.py`

**Part 4.2**

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?

**Solution:**

The new agent can behave differently in terms of runtime. The new agent performs pruning, so it can run more efficiently and use less runtime (given that it will consider less nodes). In terms of choosing moves, the two agents will behave exactly the same, since only unprofitable moves will be pruned from the game tree.

**Part 4.3**

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?

**Solution:**

Using the opening/closing libraries method, we will memorize game sequences in common situations at the start/end of the game and use them instead of recalculating the alpha-beta values of the branches in the tree corresponding to these moves every single time.

## Part 4.4

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?

### Solution:


When using status tables, we will record the minimax values of common states in the table, so that we can use them later if we encounter those same states again. This method helps to prune parts of the tree for which we have already encountered the same situation, since there is no need to calculate the alpha-beta values again. We will usually prune branches from the tree only if their search depth is less than or equal to the search depth of the identical state we encountered before, because the alpha-beta values of the first state will be more informed, and thus we will maintain correctness.

## Part 4.5

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 efficiently 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.

### Solution:

To improve the efficiency of the algorithm, we will save the minimax value of the tree as the alpha value of the root, and select a move with this value. With this method, the rerun can be more efficient because there is a possibility that more pruning will be done.

In the best case, the first move we check will have the minimax value, and we will choose it without going through the rest of the tree. In the worst case, only the last move will have the minimax value, and the rest of the values will be arranged so that no more pruning is done  than the original move. In the general case, this method will result in more cuts than the original run and will save some of the run time.

## Question 5 - ExpectiMax

### Part 5.1

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 problem you described?

### Solution:

Like most algorithms, the MiniMax method is not perfect and will not be optimal in all situations. One of the most basic assumptions of the MiniMax approach is that its opponent is a perfect rational agent, which will choose actions that are necessarily optimal. This is an idealistic approach, though, as there can indeed be cases where the adversary does not play perfectly optimally every single turn. For example, human beings are imperfect creatures and can sometimes make moves in a game that are not optimal (and they often do so). In such cases, the MiniMax algorithm will cause the agent to play too conservatively, and thus possibly produce poor results.

The Expectimax algorithm seeks to solve this problem throwing away the assumption that its opponent always behaves optimally. Instead, it will build a probabilistic model of the opponent's moves and use their expected utility (rather than the optimal utility) to decide its own moves each turn. By doing so, Expectimax doesn't play excessively conservatively against imperfect opponents, and thus on average can produce better results than MiniMax for such cases.

## Part 5.2

Assuming you use Expectimax algorithms against an agent who plays completely randomly, what probability will you use? Why?

### Solution:

In the event that we are using the Expectimax method against an agent playing completely randomly (and we are aware of this behavior), then we know that the opponent will choose a potential move out of the set of all possible moves for it with equal probability to all the others (for example, if it has 4 possible moves in a turn then the probability of choosing any of them is just 0.25). Modelling this behavior probabilistically is straightforward, since we can just tell our Expectimax algorithm to assign equal probabilities for all of the opponent's possible moves (i.e. children of a node corresponding to our turn) when it is trying to calculate the expected utility (which in this case is just their unweighted average). The Expectimax agent will then try to maximize utility during our turns, based on the expected utility values obtained for the opponent's moves.

## Part 5.3

For probabilistic games such as backgammon, in which there is a resource limit, the RB-Expectimax algorithm is used. We assume that it is known that the heuristic function  $h$  in the RB-Expectimax 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  $h(s) = -6$

### Solution:

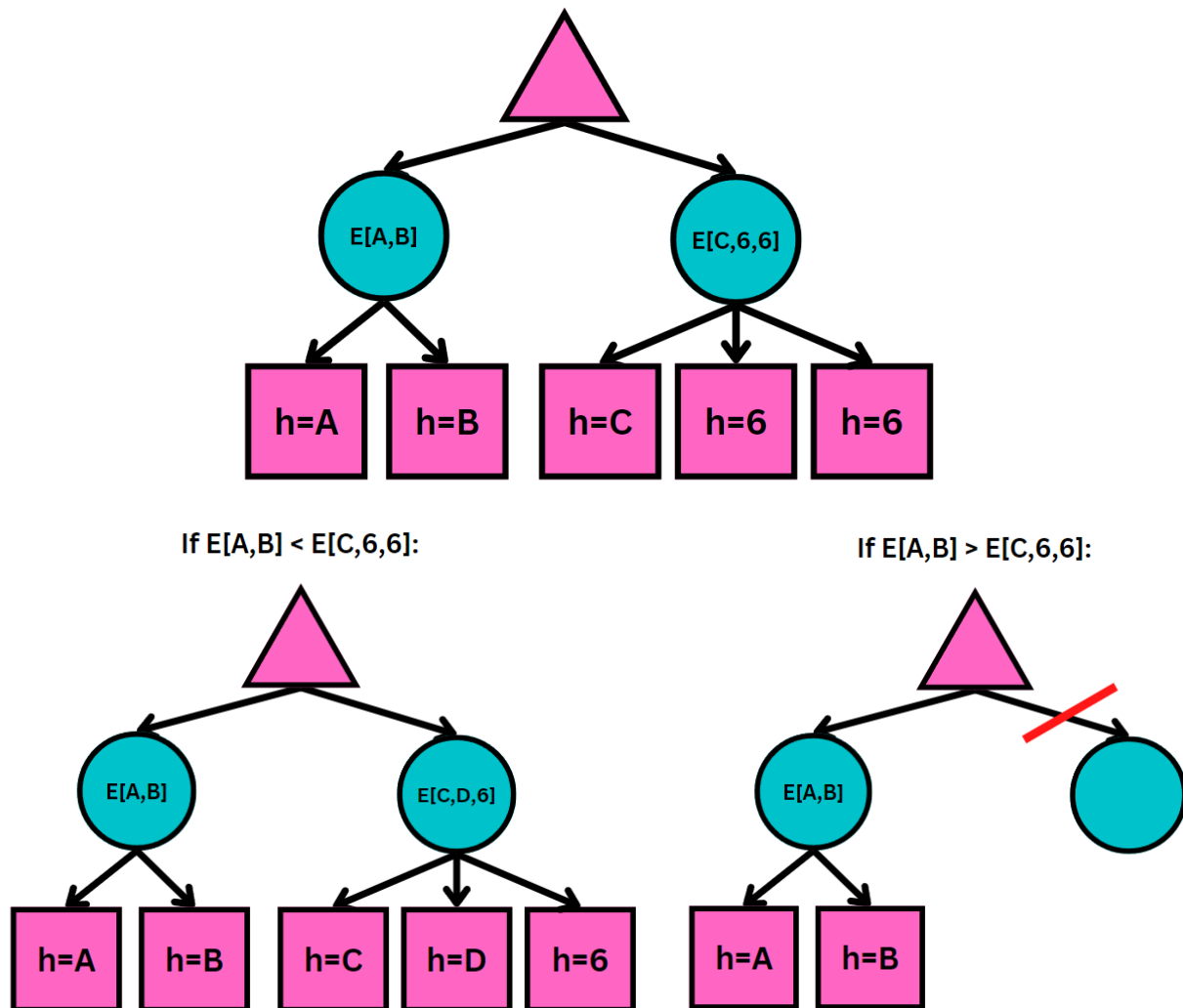
(a) Despite the fact that we are no longer conducting minimization steps, we still conduct maximization steps, and thus we can prune the RB-Expectimax algorithm using alpha-pruning (without beta).

We will explain our algorithm for pruning through an example. Let's say we are looking at a root node (maximizer) which has two child chance nodes, the first of which has two child nodes (one with utility  $h = A$  and one with utility  $h = B$ ), and the second of which has three child nodes (one with utility  $h = C$ , one with  $h = D$ , and one with  $h = E$ ). We will begin by calculating the expected utility of the first chance node  $E[A, B]$ , and we'll treat this value as our  $\alpha$ . With this value at hand, we can attempt to prune the second chance node. We will first consider the true utility value of only the first child of the chance node ( $C$ ), and we will initialize the values of all of the other children ( $D, E$ ) to the highest possible heuristic value ( $h = 6$ ). We will then calculate the expected utility of the second chance node  $E[C, 6, 6]$ , and this should be an overestimate of the actual expected utility of that chance node. In order to prune this node, we must require  $\alpha \geq E[C, 6, 6]$  - in this case, even our overestimation of the expected utility of the second chance node is not as good as the expected utility of the first chance node, and so we the actual expected utility of the second chance node certainly won't offer any improvements - thus we can prune it completely from the tree.

On the other hand, if  $\alpha < E[C, 6, 6]$  (which should often be the case), then we will return the next child node ( $D$ ) to its original value and then recalculate the overestimation of the expected utility of the second chance node ( $E[C, D, 6]$ ). We again compare this result to  $\alpha$ , and from here onward we follow the same steps as before. By conducting such iterations, we are guaranteed to discard nodes (and avoid superfluous child nodes) which definitely don't offer us improvements over ones that we've already checked.



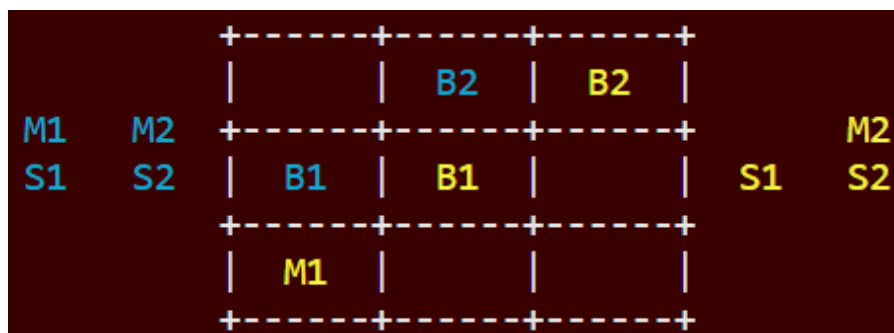
This algorithm can be seen in action in the following figure:



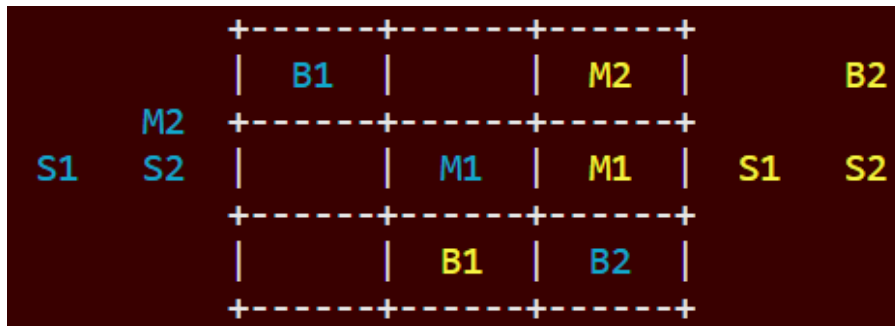
(b) Such a heuristic  $h$  for the game in our exercise would actually be relatively easy to define, and we could so as follows:

$$h(\text{state}, \text{agent\_id}) = \begin{cases} 6 & \text{Winning state for agent} \\ -6 & \text{Losing state for agent} \\ 0 & \text{Otherwise} \end{cases}$$

Granted, this is not the most intelligent way to define the heuristic, but it'll get the job done. We would get the first case ( $h = 6$ ) if our Expectimax agent (which controls the yellow pieces) wins the game, an example of which can be seen below:



We would get the second case ( $h = -6$ ) if our Expectimax agent (which controls the yellow pieces) loses the game, an example of which can be seen below:



We would get the third case ( $h = 0$ ) for any other state, i.e. in non-terminal states or states where there is a tie between the agents.

### Part 5.4

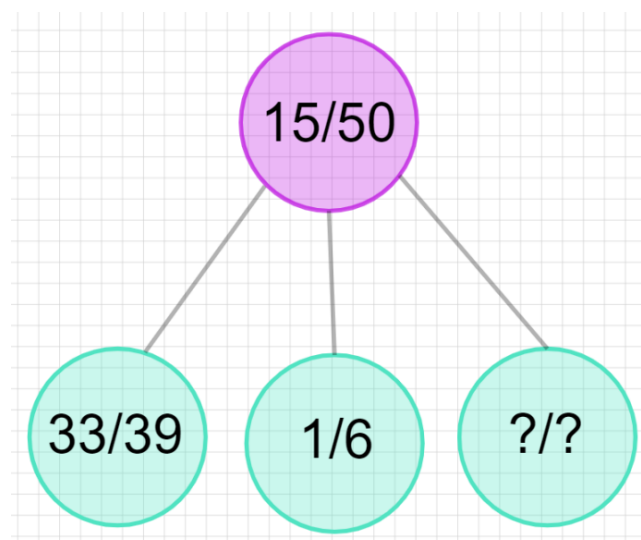
Now you will implement an Expectimax agent by implementing the `expectimax` function in the `submission.py` file, under the assumption that our opponent likes action and decides that he performs all actions with equal probability except for two types of actions:

- (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

**Solution:** See `submission.py`

### Question 6 - Monte Carlo Tree Search (Open Question)

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).



**Part 6.1**

Explain the following terms:

- (a) Exploitation
- (b) Exploration

**Solution:**

Exploitation is a behavior in which our agent places its focus on the most promising moves when choosing which nodes to select next, as part of algorithms like MCTS. Exploration, on the other hand, is a behavior in which our agent places its focus on moves where uncertainty about the evaluation is high due to smaller numbers of trials. There is a tradeoff between these behaviors when it comes to MCTS, since excessive exploration can cause us to sacrifice rewards that we could have gotten had we chosen more promising moves, while excessive exploitation could cause us to get stuck with sub-optimal values (i.e. we'll focus too much on local extrema and miss global extrema).

**Part 6.2**

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.

**Solution:**

In every branch of the tree, the total number of trials must be the same. Thus, we can equate the total number of trials in the first and second tiers in order to find the total number of trials for the unknown node  $N_?$ :

$$50 = 39 + 6 + N_? \rightarrow N_? = 50 - (39 + 6) = 50 - 45 = 5$$

As for the number of successful trials of each agent, the sum of this number across two consecutive tiers (one for the first agent, the second for the other) must also equal the total number of trials. This is because the total number of successes of the first agent plus the total number of successes of the second agent must equal the total number of trials. We can use this property to calculate the missing success value for the unknown node  $U_?$ :

$$50 = 15 + 33 + 1 + U_? \rightarrow U_? = 50 - (15 + 33 + 1) = 50 - 49 = 1$$

Thus, we see that the missing values for the unknown node should be:  $U_?/N_? = 1/5$ .

**Part 6.3**

Calculate the  $UCB1(s)$  for the blue nodes with a value of  $C = \sqrt{2}$ , 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  $UCB1$ .

**Solution:**

The  $UCB1(s)$  formula is defined as follows:

$$UCB1(s_i) = \frac{U(s_i)}{N(s_i)} + C \sqrt{\frac{\ln(N(s_i.parent))}{N(s_i)}}$$

Here, we know that  $C = \sqrt{2}$  and the total number of trials done is  $N(s_i.parent) = 50$  for all three child nodes that we are evaluating here ( $\forall i \in [1, 3]$ ). For the leftmost blue node, we see that  $U(s_1) = 33$

and  $N(s_1) = 39$ , thus:

$$UCB1(s_1) = \frac{U(s_1)}{N(s_1)} + C \sqrt{\frac{\ln(N(s_1.parent))}{N(s_1)}} = \frac{33}{39} + \sqrt{2} \sqrt{\frac{\ln(50)}{39}} = 1.294$$

For the middle blue node, we see that  $U(s_2) = 1$  and  $N(s_2) = 6$ , thus:

$$UCB1(s_2) = \frac{U(s_2)}{N(s_2)} + C \sqrt{\frac{\ln(N(s_2.parent))}{N(s_2)}} = \frac{1}{6} + \sqrt{2} \sqrt{\frac{\ln(50)}{6}} = 1.309$$

For the rightmost blue node, we see that  $U(s_3) = 1$  and  $N(s_3) = 5$ , thus:

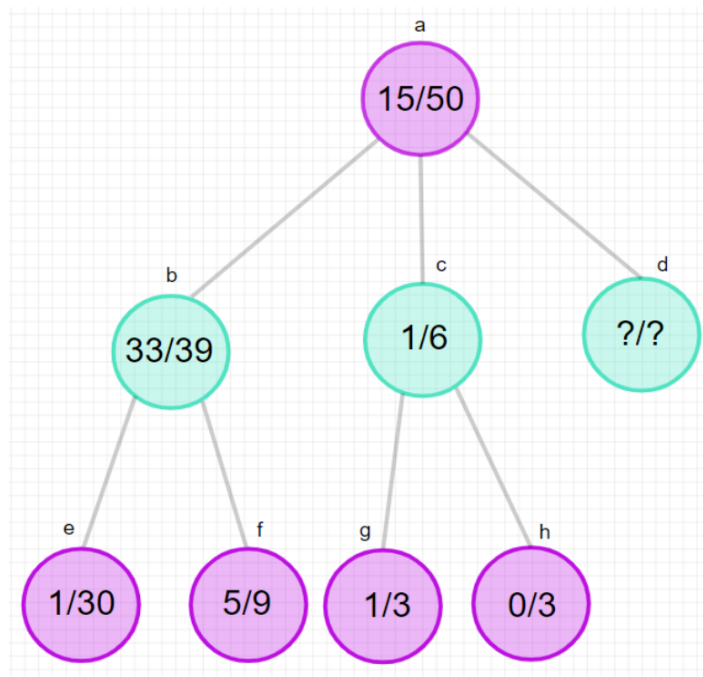
$$UCB1(s_3) = \frac{U(s_3)}{N(s_3)} + C \sqrt{\frac{\ln(N(s_3.parent))}{N(s_3)}} = \frac{1}{5} + \sqrt{2} \sqrt{\frac{\ln(50)}{5}} = 1.451$$

We obtained interesting results here. We see that the most promising node (i.e. the one with the highest value - the leftmost node) has the lowest  $UCB1$  value, meaning it is the least preferable one to expand next. We also see that the node with the lowest number of trials (and the second-lowest value) received the highest  $UCB1$  value, meaning it is the most preferable one to expand next. This is an example of how the  $UCBS$  might choose exploration over exploitation when determining which node to select next, since we choose to explore the node with the least number of previous attempts over exploiting the node that has shown to be most promising so far.

This principle is also the reason that the node with value  $1/6$  has a  $UCB1$  value close to that of the node with value  $33/39$  (the difference is less than 0.015), despite the fact that their values are so different from one another (the higher value is over 5 times the lower value) - the exploitation-exploration tradeoff is evidently taken into consideration by this method.

## Part 6.4

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



Find and specify which node will be expanded next.

**Solution:**

In this case, Andrey wants to decide which node to expand next, and he'll do so using the *UCBS* method as we did in the last step. The *UCB1* values for each of the 4 nodes are calculated as follows (given that  $C = \sqrt{2}$  again):

$$UCB1(s_1) = \frac{U(s_1)}{N(s_1)} + C \sqrt{\frac{\ln(N(s_1.parent))}{N(s_1)}} = \frac{1}{30} + \sqrt{2} \sqrt{\frac{\ln(39)}{30}} = 0.528$$

$$UCB1(s_2) = \frac{U(s_2)}{N(s_2)} + C \sqrt{\frac{\ln(N(s_2.parent))}{N(s_2)}} = \frac{5}{9} + \sqrt{2} \sqrt{\frac{\ln(39)}{9}} = \mathbf{1.458}$$

$$UCB1(s_3) = \frac{U(s_3)}{N(s_3)} + C \sqrt{\frac{\ln(N(s_3.parent))}{N(s_3)}} = \frac{1}{3} + \sqrt{2} \sqrt{\frac{\ln(6)}{3}} = 1.426$$

$$UCB1(s_4) = \frac{U(s_4)}{N(s_4)} + C \sqrt{\frac{\ln(N(s_4.parent))}{N(s_4)}} = \frac{0}{3} + \sqrt{2} \sqrt{\frac{\ln(6)}{3}} = 1.093$$

We note a slight discrepancy in the problem definition, in that the rightmost two nodes ( $g, h$ ) appear to have incorrect utility values. This is because their parent node has a utility of 1 for 6 trials, meaning that its children should have a total utility of  $6 - 1 = 5$  over those 6 trials, and not  $1 + 0 = 1$  as is currently the case. Alternatively, the error could lie in the value of the parent node, which could be fixed to be  $5/6$  in order to amend this issue. Considering the former case (which affects our solution for this part, whereas the latter would not), and given that  $U(s_4) \leq N(s_4) = 3$ , we see that  $U(s_3) = N(s_3.parent) - U(s_3.parent) - U(s_4) \geq 6 - 1 - 3 = 2$ , and in such a case we would get that  $UCB1(s_3) \geq \frac{2}{3} + \sqrt{2} \sqrt{\frac{\ln(6)}{3}} = 1.760$ . This is the largest *UCB1* value obtained among all of the nodes, and thus the *UCBS* algorithm would surely choose to expand this node next, rather than the result we obtained above (node  $f$ ). Given that we do not know the correct values of  $U(s_3)$  and  $U(s_4)$  (assuming they are the ones that are incorrectly given), we will have to suffice with ignoring the effects of this discrepancy.

Back to our results, we see that the *UCBS* method returns the highest preference for the nodes with the lowest number of trials done so far (specifically  $f, g, h$ ) and chooses the node with the highest value among them ( $f$  in this case). We see here that this method consider both exploration (thus it gives preference to the rightmost three nodes), while using exploitation to choose from among those nodes given that they have similar exploration values (0.902 for  $f$  and 1.093 for  $g, h$ ).

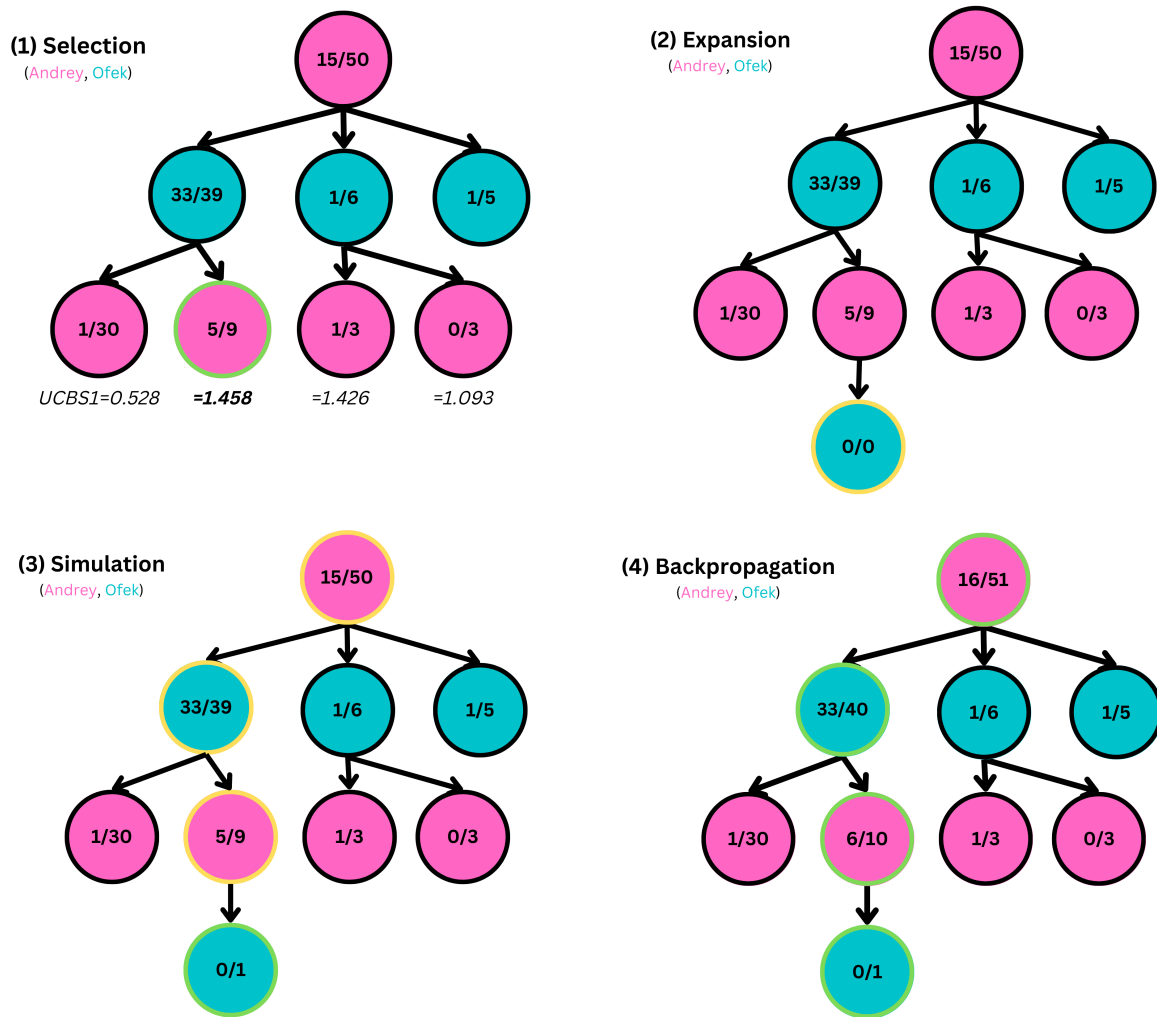
In short, Andrey should choose to expand node  $f$  (with value  $5/9$ ) next in his search.

**Part 6.5**

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.

**Solution:**

Our iteration of the MCTS algorithm using *UCBS* for node selection looks as follows:



In the first step, we select the node with the highest  $UCB1$  value, which is the second pink node from the right (with a value of 1.458, calculated in the last step). We then expand that node by adding a blue node (representing Ofek) with an initial value of 0/0. We then run one simulation for that child node, which we are told results in a loss for Ofek - thus, the value of this child node will update to being 0/1 (since Ofek has won 0 trials out of 1 trial attempted). In the final step, we work our way back up through the tree along the path that goes from the child node back to the root node, and we update the values of all nodes in that path according to the following rule:

- If the node is pink (i.e. it belongs to Andrey), then we add one win and one trial to its value:

$$\frac{n_i(t_{j+1})}{N_i(t_{j+1})} = \frac{n_i(t_j) + 1}{N_i(t_j) + 1}$$

- If the node is blue (i.e. it belongs to Ofek), then we add no wins and one trial to its value:

$$\frac{n_i(t_{j+1})}{N_i(t_{j+1})} = \frac{n_i(t_j)}{N_i(t_j) + 1}$$

By doing so, we will complete our backpropagation step and thus finish one iteration of the MCTS algorithm for our game.

Thank You!

