ARTIFICIAL INTELLIGENCE 236501

Homework #2

Submission by:

- Alexander Shender : 328626114
- Samuel Panzieri: 336239462

1.

To define the Successor function, we first define the Current location of the agent 1, and agent 2, as following:

$$1 = [1_i, 1_j]$$

$$2 = [2_i\ , 2_j]$$

Since the state variable doesn't contain exact information on the agent 1 and agent 2 locations, we have to loop through $s \in \{-1, 0, 1, 2\}$ and find a tile with values 1 and 2.

$$1 = [1_i, 1_j]\ \ if\ \{S[1_i, 1_j] = 1\}$$

$$2 = [2_i, 2_j]\ \ if\ \{S[2_i, 2_j] = 2\}$$

We have to check that the following conditions hold:

- Player can move to a tile which isn't gray, or other player
- Player can move to a tile which isn't a wall

Defining a group of Moves which can be done in a game:

$$Moves = \{\forall Player \in \{1,2\}, \forall Direction \in \{[1,0], [-1,0], [0,1], [0,-1]\} :\ [Player, Direction]\}$$

For example, a move of first player upwards is: $Move = [1, [1,0]]$, where:

$$Move(0) = 1;\ Move(1) = [1,0]\ ;\ \ Move(1)_i = 1\ ;\ \ Move(1)_j = 0$$

Defining a condition which will be checked:

$$CanMove(S, Move) = \left\{ \begin{array}{c} S\left[\underbrace{Move(0)_i + Move(1)_i}_{player\ i\ next\ location}\ , \underbrace{Move(0)_j + Move(1)_j}_{player\ j\ next\ location}\right] \\ not \in \left\{-1,\ \ \underbrace{1 - Move(0)}_{index\ of\ other\ player}\right\} \end{array} \right\}$$

Now, we can define the successor function:

$$Succ(s) = \left\{ \forall Move \in Moves \right.$$

$$\left. : \; S\left[\underbrace{Move(0)_i + Move(1)_i}_{player\ i\ next\ location}, \underbrace{Move(0)_j + Move(1)_j}_{player\ j\ next\ location}\right] \;\; | \; CanMove(S, Move) \right\}$$

2.

Win(s,i) ⇔ Succ(s,i) ≠ ∅ ∧ ∀ s’ ∈ Succ(s,i): Succ(s’,(i+1)mod 2) = ∅

Tie(s) ⇔ Succ(s,1) = ∅ ∧ Succ(s,2) = ∅

3.

The branching factor can be 4 in the initial turn of any player, if surrounded by unvisited or white tiles. From second turn of every player (third depth of a tree), it will be 0-3, since we can never visit the tile which we have already visited.

4.

Simple player iterated on all the directions (4 directions – UP, DOWN, LEFT, RIGHT), checking if they are feasible. If a direction is feasible, the 'state_score' is being calculated. The move with the best score is performed. If state_scores are equal, FIRST direction which was calculated wins.

According to the 'state_score', it will prefer the state with the LEAST number of further states available for that player, as long as this will not lead to 0.

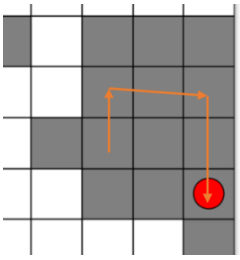
PROS:

- Given a certain area with no obstacles (a large space), the agent will for sure fill all the tiles

CONS:

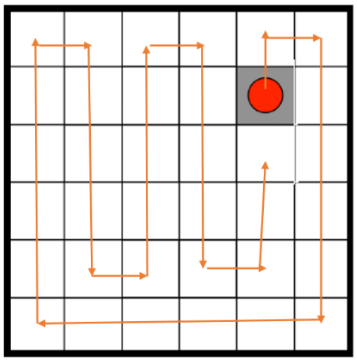
- Sees only 1 step into the future
- Doesn't take enemy location into account
- Can run into the dead end, while being able to easily avoid it (connects to the first CON)

Example of a dead-end (map number 3)



5.

As stated in PRO, the agent will fill the whole given area in an optimal way, for example here, I have marked it's path:



6.

The heuristic value includes only the number of successors of a certain state for a certain player. CONs can be similar to the "simple player" tactics, and more:

- This heuristic is not admissible – Sometimes going into a child state which has only 1 successor can lead to victory, while still having other children with more successors, which will result in higher heuristic
- It does not take into account the enemy location
- It looks only 1 step further.

7.

As described earlier, the heuristic is based on 3 main parts:

$$\begin{cases} H = a \cdot (MaxGround(s)) + b \cdot StateScore(s) - c \cdot EnemyDistance(s);\ iff\ EnemyDistance(s) > 0 \\ H = a \cdot (MaxGround(s)) + b \cdot StateScore(s);\ iff\ EnemyDistance(s) == -1\ or\ iff\ EnemyDistance(s) < DistFromOppRevelent \end{cases}$$

Where:

- $EnemyDistance(s)$:
  the shortest distance from the enemy-as found through the empty tiles. If there is no connection to the enemy, return -1
- $MaxGround(s)$:
  the tiles advantage of a player against the opponent. Calculates the difference of the number of tiles which are closer to the agent than to the opponent.
- $StateScore(s)$
  Calculates the number of available moves from a state where the agent is located. Similar to the SimplePlayer, but is the opposite! E.g., when there are 3 free tiles available, it will return 3. If none available, returns -1
- a, b, c – scaling factors
- $DistFromOppRevelent$ – the margin (threshold) customizable by user.

The strategy:

- When the distance to the enemy is large (perhaps in the beginning of the game), the agent will go in the direction of minimizing this distance (pay attention that there is a 'minus' sign).
  - If there is no connection through tiles to the enemy agent, don't use this heuristic (it loses any point going towards enemy if its unreachable).
  - If the distance to the opponent is less than $DistFromOppRevelent$, we aren't interested in getting closer to opponent anymore (but rather gain territory control), so we don't use it.
- The MaxGround heuristic will prioritize the states, where the tiles advantage of the player over its rival is the maximum. (Or the disadvantage is minimum). We want to have more tiles which we can reach before the enemy can, this way gaining control.
- The StateScore will try to bring the agent to the place with the maximum available tiles. The maximum value of this heuristic is 3, so it will be relevant towards the end of the game, when the EnemyDistance(s) heuristic is not used, and the MaxGround(s) value is not significant. (same scale as StateScore(s))
- We also use the Utility function. If some state is found as "final", it calculates who wins after this state. If player wins, a value of "99999" is returned, and "-99999" if an opponent wins. The value of "-9999" is returned if the result is a tie, so that the agent will prefer choosing another choice which is not final, and has some heuristic value returned.

8.

The “anytime” variation of the Minimax algorithm consist in running the usual minimax algorithm with an additional time parameter, and demanding that the algorithm returns the best answer available in that time.

For this purpose we run the rb-minimax algorithm with increasing depths, until the available time runs out.

At the end of each iteration the chosen step is saved and at the end of the allocated time we interrupt the search and return the last chosen step.

This technique is called iterative deepening, and the problem presented in the lecture regarding this technique, is that on average, the time will run out during the last iteration calculation, thus we will return the previous iteration step, but still most of the resources will be used by the last iteration, so we will use a lot of resources, for no gain since we will not manage to terminate the last iteration.

9.

The proposed solution for this problem in the lecture is that in each iteration, we can store the minimax value for each of the sons of the upper level

On the last iteration, thus, half of the sons will represent a search at the deepest level, whereas the rest will represent the search at the previous depth level (one before the deepest). This will solve the problem partially, since at least for half of the sons we will have used our resources in order to see into the next depth level.

10.

Defining the $f(l, last\_iteration\_time)$ function:

Looking at the Minimax basic agent (or Alpha-Beta with NO PRUNING (worst case possible, even with child sorting)), the next depth will develop $l \cdot b$ leaves, where b is a branching factor.

Define:

$T(d) - number\ of\ leaves\ developed\ at\ the\ depth\ d$

We know, that in BFS,

$$T(d) = b^d$$

Thus,

$$T(d+1) = b^{d+1}$$

Defining number of children developed at current depth as l_d, we get

$$l_d = b^d\ ; l_{d+1} = b^{d+1} \rightarrow \quad l_{d+1} = \frac{b^{d+1}}{b^d} l_d = b \cdot l_d$$

Assuming that the time to develop each leave is the same among every iteration, we get:

We define the time to develop leaves at the depth d as $time_d$

$$\frac{time_{d+1}}{time_d} = \frac{l_{d+1}}{l_d} = b$$

We take the maximum branching factor possible (after the first move), which is $b = 3$.

Thus, we get:

$$time_{d+1} = time_d \cdot 3$$

We also need to account for the time required to develop all the previous tree again, thus adding $time_d$ to the total time for the calculation of the whole tree at level $total\_time_{d+1}$:

$$total\_time_{d+1} = time_{d+1} + time_d = \ time_d \cdot 4$$

This is of course the worst case, where no pruning occurs, and that EVERY leaf has 3 successors.

Note:

In any case, the actual time may be lower because the worst case for the branching factor was taken.

When using pruning, the actual time required for the next step maybe even less, since some of the branches may be pruned, thus we will save time on their calculation time.

11.

In the Anytime Contract, the Alpha-Beta variation is supposed to be more efficient. We have to remember that it prunes only the branches that will anyway not affect the decision of the root node. Thus, those resources (more time that became available) will be used to reach deeper depth of the tree.

12.

When the depth will be limited, both algorithms will return the exact same result. As mentioned before, Alpha-Beta prunes only branches which won't affect the Minimum or Maximum choice of the parent node.

13.

The heuristic for the Minimax player is the same heuristic as defined in the Question 7 in this report. The scale factors were chosen statistically and experimentally.

14.

The heuristic function used is the one defined in Answer no. 7.

The whole algorithm is as explained in the lecture:

When a certain 'child' is being processed, first we check if the state is final. If it is, check whether the result is Win, Lose, or Tie. Return '-99999' for loss, '+99999' for win, '-9999' for tie.

If state is not final, check if the depth is 0. (meaning we reached the max depth). If it is, use the heuristics to return the value. If not, find the children of the state, switch player and use same function on each child.

15.

The theoretical explanation was given in Question 10. The implementation was made as described in the homework assignment paper, since it was found to be efficient, and indeed there were no times when the time limit was broken.

Additionally, to make move faster, the depth was limited to the number of available while tiles on the board (we cannot step twice on the same tile).

Also, when the best value returned is “-99999”, or “99999”, which is actually the maximum Minimax value found, the agent does not iterate further and returns the move which resulted in that value

- If it’s “-99999”, meaning best possible move is when the agent loses. No need to iterate more
- If it’s “99999”, that the move which guarantees victory, go for it!

16.

The expected behavior : the Alpha-Beta Player is expected to reach further depth in a given amount of time than the Minimax Player, thus giving a better estimated Minimax value and its corresponding move.

We can set players to play one against the other, or rather we can play against both of them, performing same actions, which is what we do. This way they will have same starting conditions. Those are our custom outputs from the game environment.

The Minimax player is on the left and the AlphaBeta player is on the right.

```
Starting Game
LivePlayer VS MinimaxPlayer
Board 2
Players (besides LivePlayer) have 2.0 seconds to make a move
w
====================
Agent: MinimaxPlayer
Move No: 1
Depth reached : 12
Leaves developed: 94, Heuristics used : 45
Move chosen: (-1, 0)  Value = 1
====================
w
====================
Agent: MinimaxPlayer
Move No: 2
Depth reached : 12
Leaves developed: 93, Heuristics used : 36
Move chosen: (-1, 0)  Value = 3
====================
w
====================
Agent: MinimaxPlayer
Move No: 3
Depth reached : 14
Leaves developed: 169, Heuristics used : 53
Move chosen: (0, -1)  Value = 11
====================
w
====================
Agent: MinimaxPlayer
Move No: 4
Depth reached : 15
Leaves developed: 286, Heuristics used : 65
Move chosen: (0, -1)  Value = 11
====================
w
====================
Agent: MinimaxPlayer
Move No: 5
Depth reached : 11
Leaves developed: 79, Heuristics used : 23
Move chosen: (0, -1)  Value = 99999
====================
```

```
Starting Game
LivePlayer VS AlphaBetaPlayer
Board 2
Players (besides LivePlayer) have 2.0 seconds to make a move
w
====================
Agent: AlphaBetaPlayer
Move No: 1
Depth reached : 13
Leaves developed: 42, Heuristics used : 42, Branches pruned: 39
Move chosen: (-1, 0)  Value = 3
====================
w
====================
Agent: AlphaBetaPlayer
Move No: 2
Depth reached : 14
Leaves developed: 55, Heuristics used : 49, Branches pruned: 61
Move chosen: (-1, 0)  Value = 2
====================
w
====================
Agent: AlphaBetaPlayer
Move No: 3
Depth reached : 16
Leaves developed: 79, Heuristics used : 60, Branches pruned: 117
Move chosen: (0, -1)  Value = 11
====================
w
====================
Agent: AlphaBetaPlayer
Move No: 4
Depth reached : 21
Leaves developed: 130, Heuristics used : 70, Branches pruned: 195
Move chosen: (0, -1)  Value = 11
====================
w
====================
Agent: AlphaBetaPlayer
Move No: 5
Depth reached : 11
Leaves developed: 11, Heuristics used : 7, Branches pruned: 17
Move chosen: (0, -1)  Value = 99999
====================
```

“Value” symbolizes the Minimax value returned for this move. When this value is 99999, this means the agent has reached the Utility function which returns the “Victory” for the agent, thus there is no need to develop further depth, and it stops and returns the action which returned “Victory”.

We can see that AlphaBeta Player reaches deeper upon every step calculation.

We can clearly see that the AlphaBeta player prunes branches (it grows from turn to turn because the number of free tiles in the game diminishes, and heuristics calculation is made easier each consecutive turn), which allows him to reach bigger depth and prune more branches.

We can see that Minimax develops more “Leaves” (Nodes where heuristic or utility is calculated), which shows that it doesn’t prune any branches and develops them even though they don’t affect the Minimax value at the root.

Everything is as expected, and other boards show similar behavior.

We also let the agents play against each other, letting each agent to start first or second.

NOTE: After the children moves are being created for a specific leaf, their order is being randomized, so create a random behavior for the agent in case some states will return the same Minimax value (in AlphaBeta, the first one will be chosen, second one will be pruned).

We can see also in game that the AlphaBeta agent sees further, and reaches bigger depth, thus can see the outcome of the game before other agents. For example, here it calculated that the game will end with “Tie”, while the rival (Minimax) still used heuristics. This happened on multiple boards.

```
====================
Agent: AlphaBetaPlayer
Move No: 7
Depth reached : 58
Leaves developed: 55, Heuristics used : 0, Branches pruned: 142
Move chosen: (1, 0)  Value = -9999.0
====================
====================
Agent: MinimaxPlayer
Move No: 7
Depth reached : 26
Leaves developed: 206, Heuristics used : 48
Move chosen: (-1, 0)  Value = 5
====================
```

On the small boards there is a bigger chance of a game ending with a tie, since the heuristics give a good evaluation of the movement until the moment when the amount of free tiles is relatively small, so that both agents can calculate the whole tree until the “victory”, “loss”, or “tie” outcome.

Those are the results of the games with Same heuristics for both players:

Board	Starting Agent	Result
0	Minimax	Tie
0	AlphaBeta	Tie
1	Minimax	Tie
1	AlphaBeta	Tie
2	Minimax	Tie
2	AlphaBeta	Tie
3	Minimax	Tie
3	AlphaBeta	Tie
4	Minimax	AlphaBeta wins
4	AlphaBeta	AlphaBeta wins

We can see that the difference in outcome starts being visible on big boards.

17.

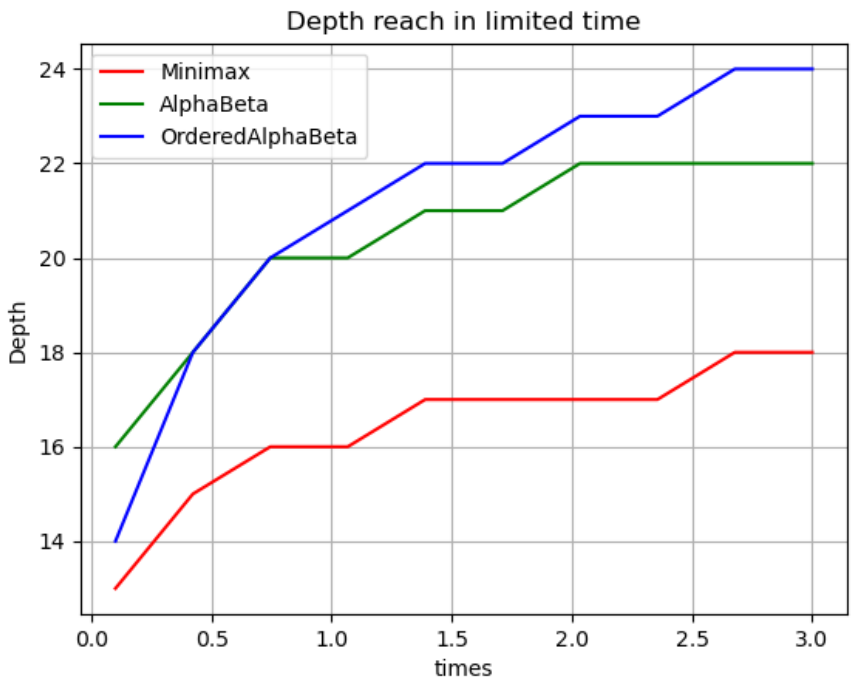
The example code was used and extended.

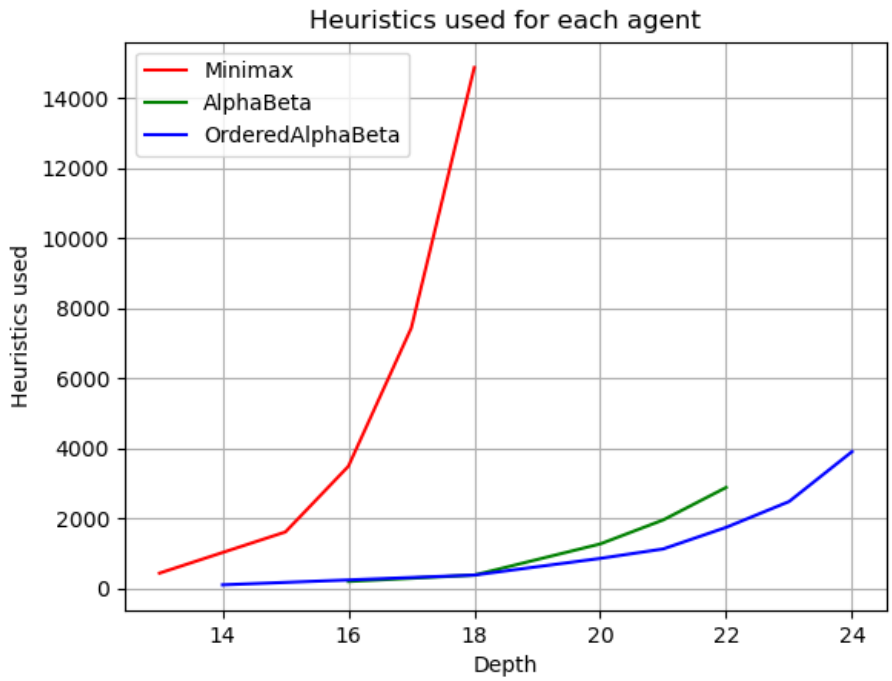
Before running the graphs, our expectations are:

- Given a certain amount of **time**, Ordered Alpha-Beta would reach biggest depth, then regular Alpha-Beta. Minimax would reach the most shallow depth.
- For a given **depth**, the Minimax Agent will use the most amount of heuristics, since it does not prune any branches, and has to develop the whole tree.
- For a given **depth**, Ordered Alpha-Beta will develop less branches than the regular Alpha-Beta, since branches will be pruned closer to the root, thus their children branches will never be discovered.
- For a given **depth**, The Ordered Alpha Beta will prune less branches than regular Alpha-Beta, out of the reasons states in the previous statement

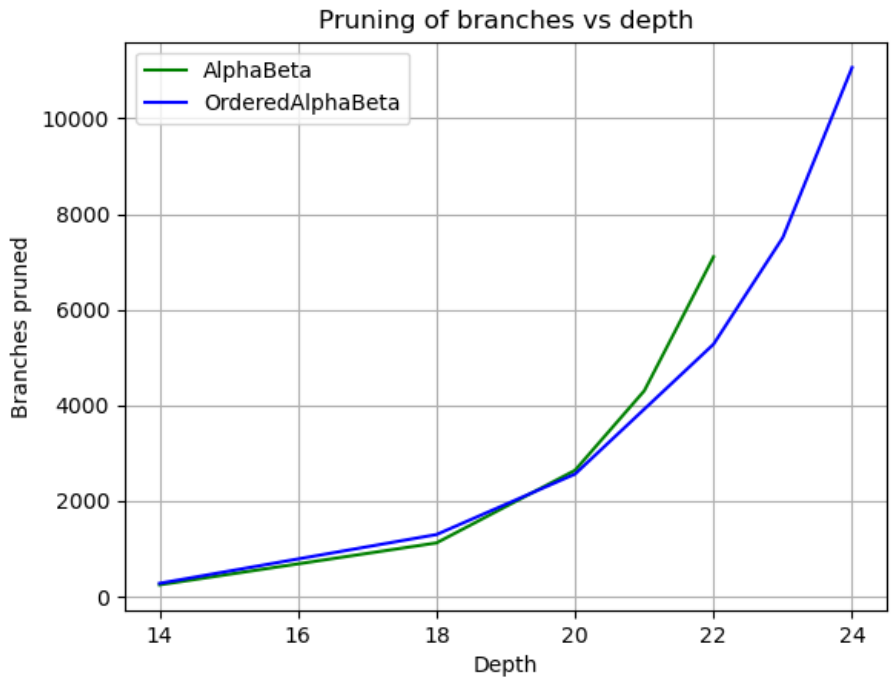
To verify those statements, we do the experiments. First, on more simple heuristic, consisting only of the StateScore value. The goal is to test the algorithms and see that they act correctly.

The maximum Mimimax value this heuristic will return is 3. So whenever some leaf will return this heuristic value (=3), the 'alpha' parameter at the root will be updated, every other branch will develop exactly 1 leaf at the specified current depth, since no heuristic value exists which is bigger than 3. So we expect considerable bigger depth for AlphaBeta and OrderedAlphaBeta.


Depth reach in limited time
Minimax
AlphaBeta
OrderedAlphaBeta
Depth
times
24
22
20
18
16
14
0.0
0.5
1.0
1.5
2.0
2.5
3.0

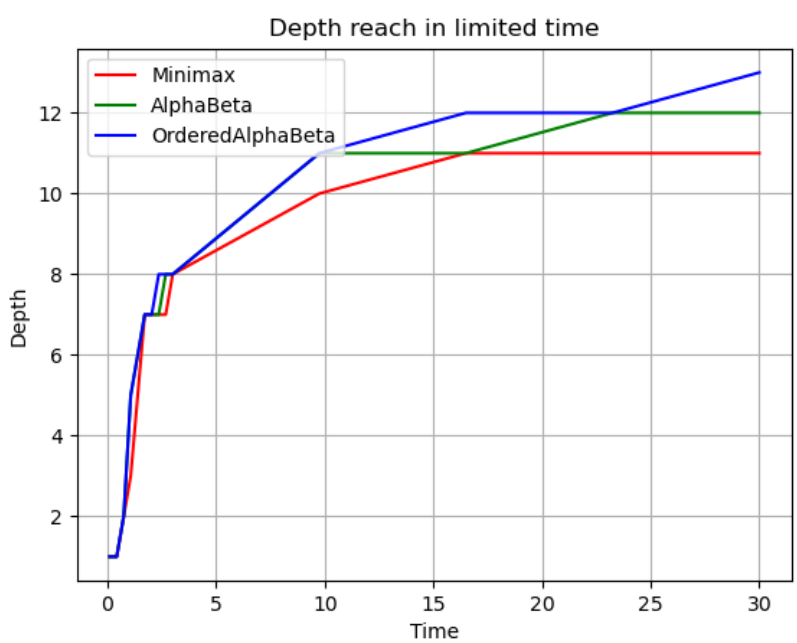


Interesting observation – where the branches are not pruned, and the Heuristic is used on every leaf, we can see that for each increase in depth, we have increase in a number of heuristics used by a value less than 3. The reason is the branching factor, which has a max val of 3.



We can indeed see that all those statements are verified on the simple heuristic.

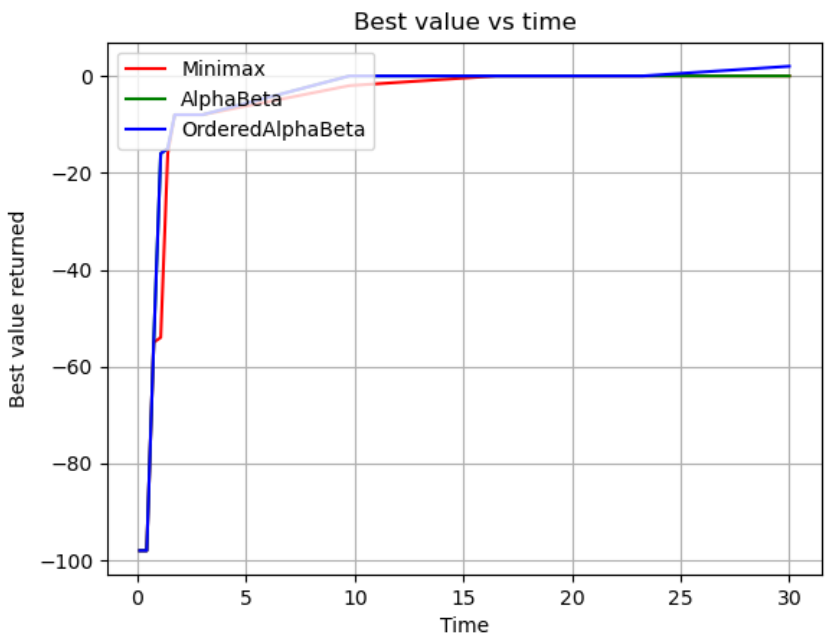
Now, running on the more complex heuristic:



We can observe the same tendency – the Ordered AlphaBeta player reaches the furthest depth, while the Minimax player reaches the shallowest. Another interesting graph to look at is the Minimax value

returned. We can see that Ordered AlphaBeta succeeds to return the biggest value for a given time, thus providing us with the most beneficial (according to heuristics) move.

Secondly, we observe that the gap became closer, the difference between the 3 algorithms became smaller. This may be explained by the following observation:



We can see that, as the agent has more time, the Minimax Value of the found solution gets higher. Thus, also, if the depth gets deeper, the value of the solution improves. Thus, at every new depth, we would find a better solution (due to heuristics). And it's logical – as the agent expands the search, it will find a node with a smaller distance to the enemy player – and this is one of our heuristics.

18.

Two agents were created

The difference in the Heuristics was only the MaxDistance parameter for the MaxGround heuristic calculation.

The radius of the MaxGround calculation was the following:

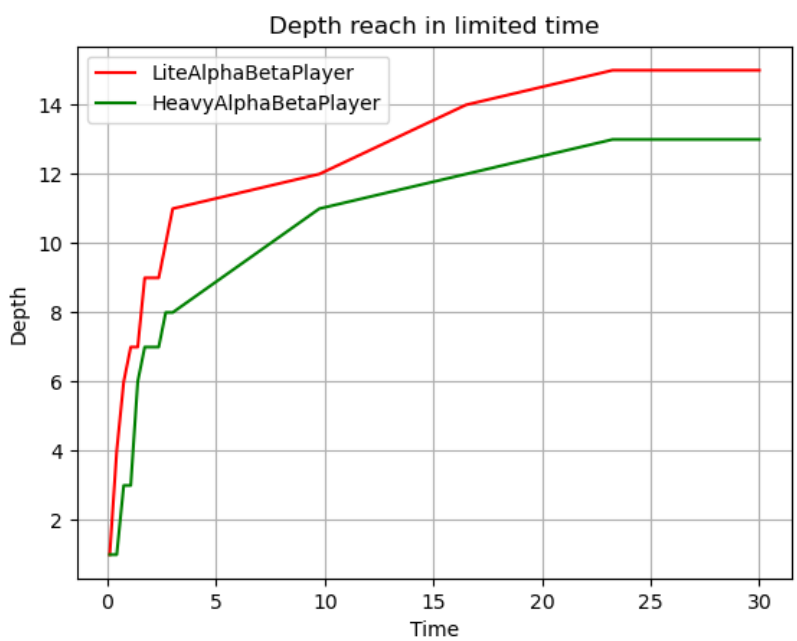
- Heavy : 15
- Light: 5

To remind, this heuristic calculates the difference between the amount of free tiles that are closer to the agent and the amount which is closer to the enemy. Only tiles below the MaxDistance are taken into account. The other 2 heuristics are light already, so they are left equal between the Agents.

What we expect:

- The Depth reached by the Lite player should be higher, since the Heuristics calculation time is smaller

And indeed:



We can see that this is the case. Running some games between the agents.

First of all, we can indeed see that the Agent with the Lite heuristic reaches bigger depth during the game:

```
Board 3
Players (besides LivePlayer) have 2.0 seconds to make a move
====================
Agent: LiteAlphaBetaPlayer
Move No: 1
Depth reached : 16
Leaves developed: 193, Heuristics used : 176, Branches pruned: 207
Move chosen: (1, 0)  Value = -6
====================
====================
Agent: HeavyAlphaBetaPlayer
Move No: 1
Depth reached : 14
Leaves developed: 110, Heuristics used : 102, Branches pruned: 119
Move chosen: (1, 0)  Value = -8
====================
```

The smaller value of the HeavePlayer is because the initial state has changed of course.

Board	Starting Agent	Result
0	Lite	Tie
0	Heavy	Tie
1	Lite	Tie
1	Heavy	Tie
2	Lite	Tie
2	Heavy	Tie
3	Lite	Tie
3	Heavy	Tie
4	Lite	Heavy wins
4	Heavy	Heavy wins

We can see that there is an advantage in the Heavy Heuristics. This is the agent which was chosen for the competition.

The end