INGI2261 – Artificial Intelligence
Assignment 3: Project: Adversarial Search

Group 16: Benoit Daloze Xavier de Ryckel

19 mars 2017

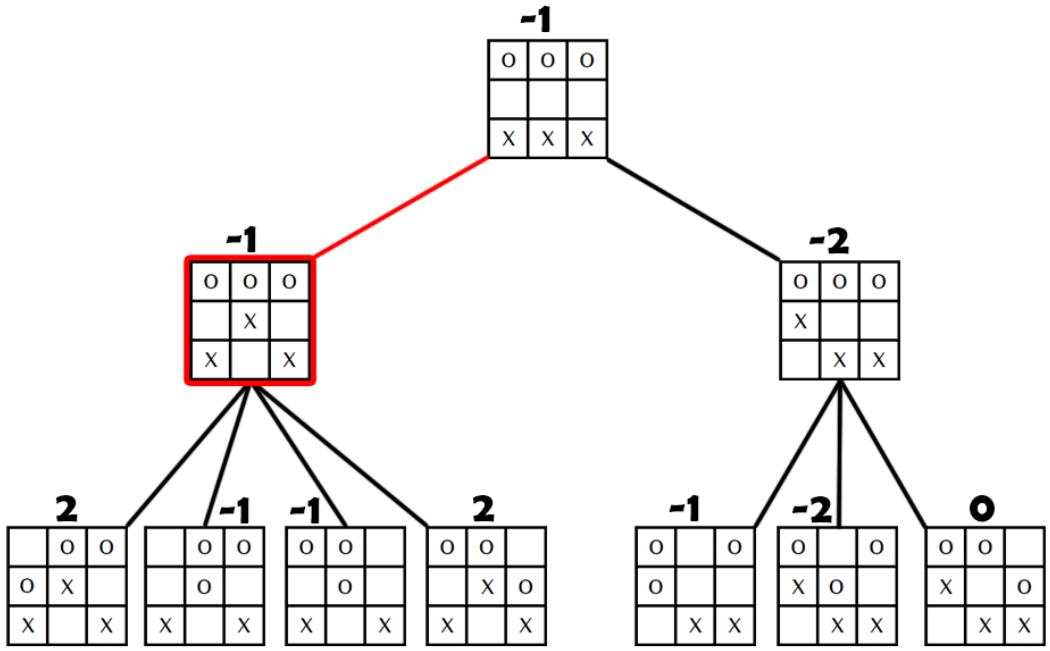
1 Scipion

1. Draw the game tree for a depth of 2, i.e. one turn for each player.

2. Evaluate the value of the leaves using the heuristic function (on the figure you draw in 1, you don't have to implement it !).

3. Using the MiniMax algorithm, find the value of the other nodes.

4. Circle the move the root player should do.



2 Alpha-Beta search

1. Perform the MiniMax algorithm on the tree in Figure 4, i.e. put a value to each node. Circle the move the root player should do.

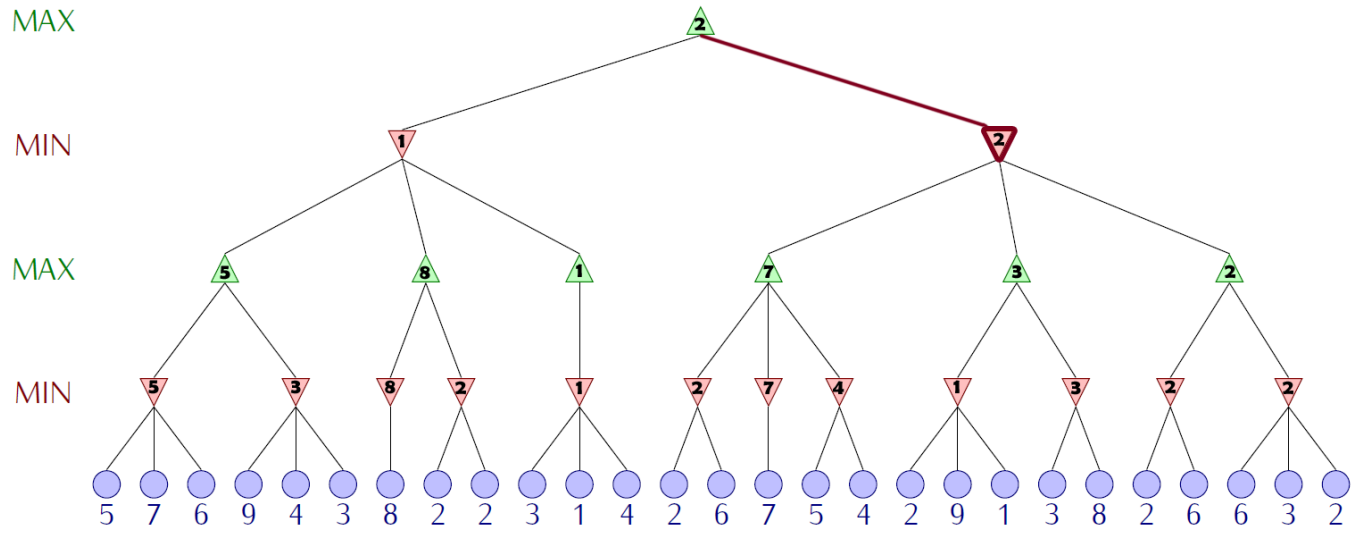


Figure 4: MiniMax

2. Perform the Alpha-Beta algorithm on the tree in Figure 5. At each non terminal node, put the successive values of α and β. Cross out the arcs reaching non visited nodes. Assume a left-to-right node expansion.

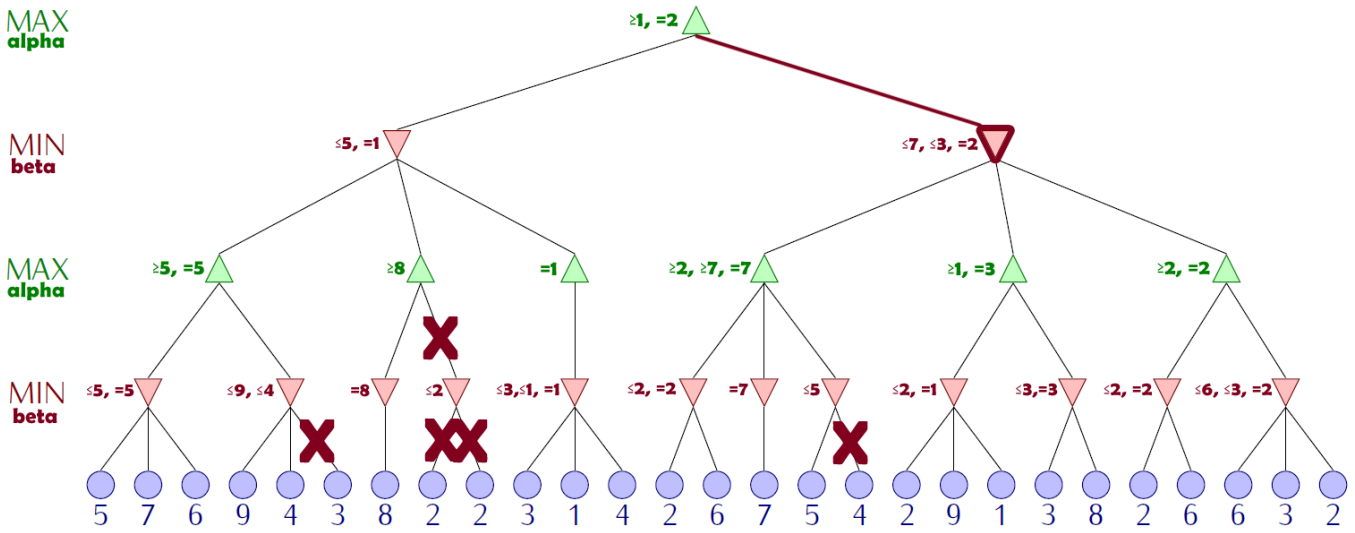


Figure 5: Alpha-Beta, left-to-right expansion

3. Do the same, assuming a right-to-left node expansion instead (Figure 6).

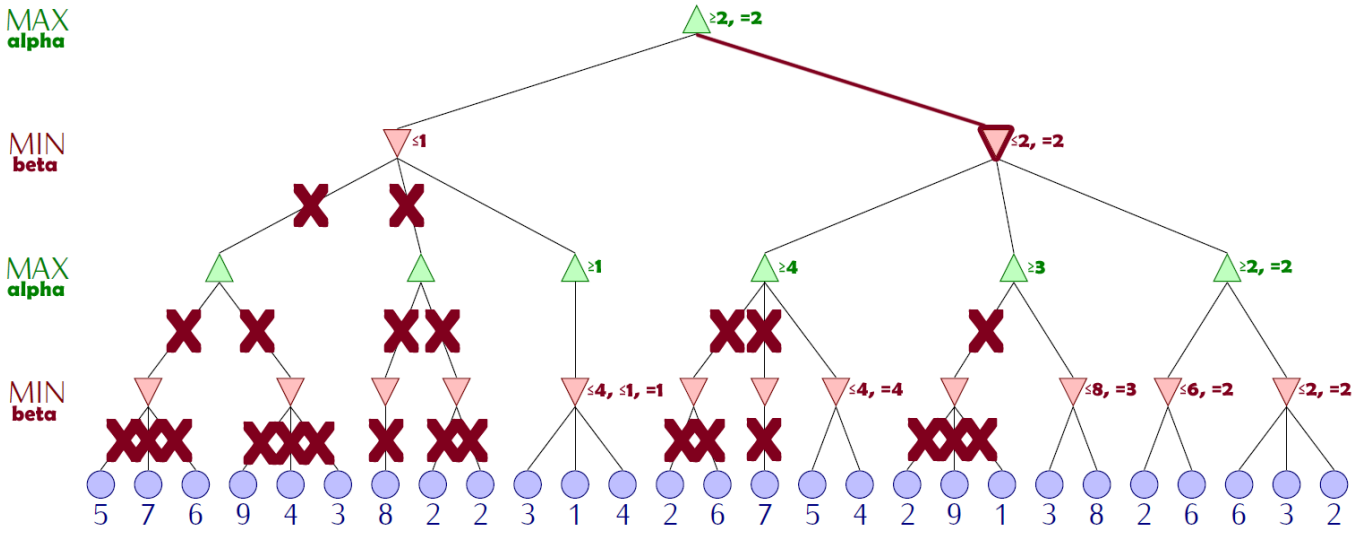
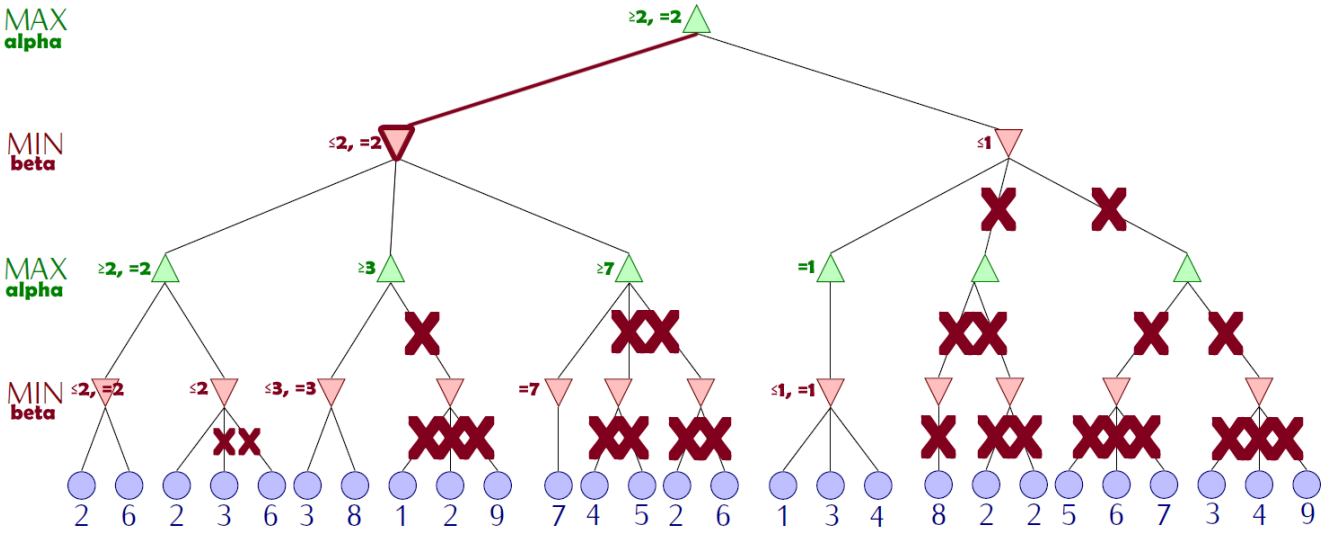


Figure 6: Alpha-Beta, right-to-left expansion

4. Can the nodes be ordered in such a way that Alpha-Beta pruning can cut off more branches (in a left-to-right node expansion) ? If no, explain why ; if yes, give the new ordering and the resulting new pruning.



Alpha-Beta, left-to-right expansion

3 Sarena

3.1 A basic Alpha-Beta agent

1. Implement the successors method *without modifying* `minimax.py`. You shall return all the successors given by `Board.get_actions` in that order.

2. We want to explore the whole tree. The cut-off function thus only has to check if the given state is a terminal state. This can be done using the `Board.is_finished` method. Implement this in the cutoff method.

3. As we explore the whole tree, the evaluation function is in fact the utility function. This function shall return 1 if the agent is winning, -1 if he is loosing and 0 if it is a draw game. You can use `Board.get_score` to determine the winner, but beware that the agent is not always playing yellow.

3.2 Comparison of MiniMax and Alpha-Beta

4. What action will be played when using the MiniMax algorithm and when using the Alpha-Beta algorithm ? Is there a difference between both results ?

No, the same action will be chosen (0,0,1,0) because Alpha-Beta pruning only removes branches which cannot be chosen by MiniMax.

5. For both algorithms, give the time taken and the number of nodes that are visited when searching the tree. Does this meet your expectations ?

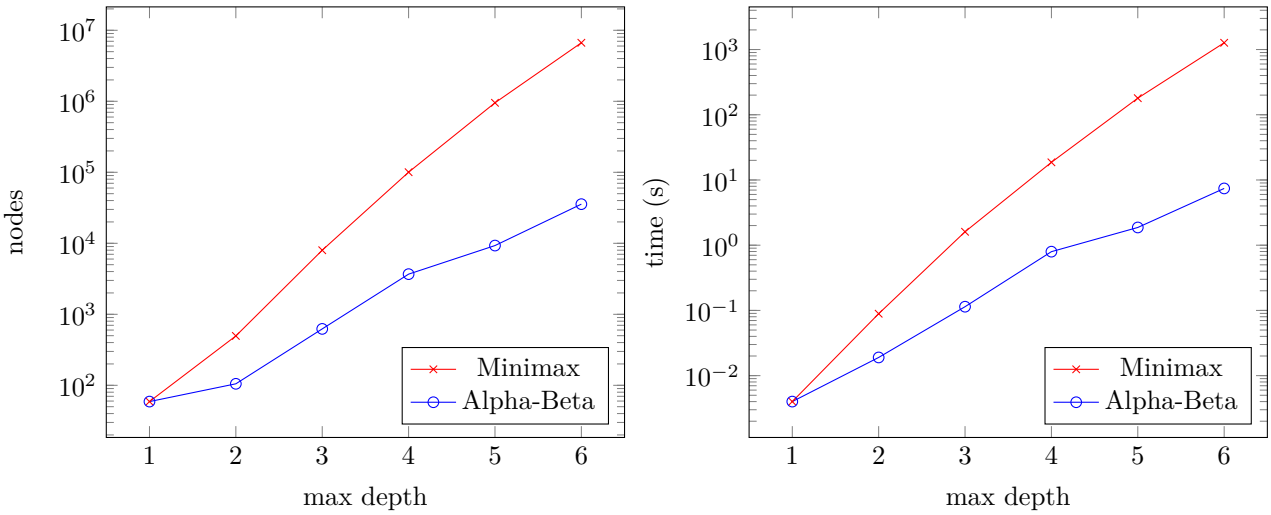
6. For both algorithms, report the number of nodes that are visited at each depth level of the tree. What do you observe ? Explain.

We observe the Alpha-Beta algorithm is a lot faster and visit much less nodes than MiniMax. This is what we expected, as Alpha-Beta is just pruning some non-chosen branch and therefore not evaluating those nodes.

For the first movement of the yellow player :

max depth	1		2		3		4		5		6	
	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
Minimax	59	0.004	496	0.089	7988	1.606	100340	18.65	951956	179.63	6669444	1269.68
Alpha-Beta	59	0.004	105	0.019	623	0.114	3679	0.796	9292	1.87	35592	7.418

Here are two plots for this data, in semi-log scale.



3.3 Evaluation function

7. Describe your evaluation function.

We have 3 evaluation functions : `score_at()`, which evaluates the score of a single cell, `score()` which sums the scores of all cells and `incremental_score()` which computes the score of the new state from the parent state.

The initial score (for the given `precepts`) is computed by using `score()`. Scores for states in `successors()` are computed by `incremental_score()`.

Our main evaluation function, `score_at()` is based on a few observations :

- If there is no pile on the cell, we don't evaluate the cell and the score is 0.
- If there is a pile of 4 chips, on an arrows case, the top color will win.
- If there is a pile of 4 chips, on a normal case, the bottom color will win (unless all neighbors around can not move, but this is very rare).
- If a pile stands on an arrows case and has no neighbor, the top color will win (because once a normal case has no more chips, none can go again on this case).
- If a pile stands on a normal case, the color at the bottom of the pill may win, if reversed.
- If a pile is on an arrows case (and has neighbors), the top color might win (by taking the pile over another).

`incremental_score()` works in an incremental way in order to save execution time. Therefore, what we chose to do is to use the score of the previous played state in order to compute the score of the next state. We are also using a function called `score_at(state, i, arrows)`. This function is used to define the value of the score of one cell of the given state.

So, in `incremental_score`, first of all, we retrieve the previous score. Then, we subtract from this score the score of the old cell (the cell of id i) for the previous state (because there is no more pile on i, therefore yielding a score 0 for that cell). Then, we compute the final score for the current state by incrementing our score again. There are then two possibilities.

First, if the current cell is not on an arrows cell, it means that its direct surrounding neighbors cells contain arrows. So, we iterate over each of those cells, and for each of them, we remove from the main current score the score of the current neighbor cell for the previous state, and we add to the main current score the score of the current neighbor cell for its current state on the board. Note that we perform the same strategy also at the same time on the current new cell (n, the id of the target cell of the move, whereas i is the origin ; obviously i and n are neighbors) for the new state.

Second, if the current cell for the new state contains arrows, it means that instead of this process, we can simply update the score for cell n (subtract score in old state and add score in new state).

8. In Alpha-Beta, the evaluation function is used to evaluate leaf nodes (when the cut-off occurs). As seen in previous questions, the pruning of Alpha-Beta depends on the order of the successors. Explain how your evaluation function could be used to (we hope) obtain more pruning with Alpha-Beta. Are there any drawbacks to your approach ?

The idea is to sort successors in a way such that we can prune as many branches of the tree as possible. So, what we want to do is to associate each successor to a score so that we can compare them together and yield them in the right way by ordering them for a given state, for a left-to-right Alpha-Beta, from lowest to highest score at "min" depth, and from highest to lowest at a "max" depth.

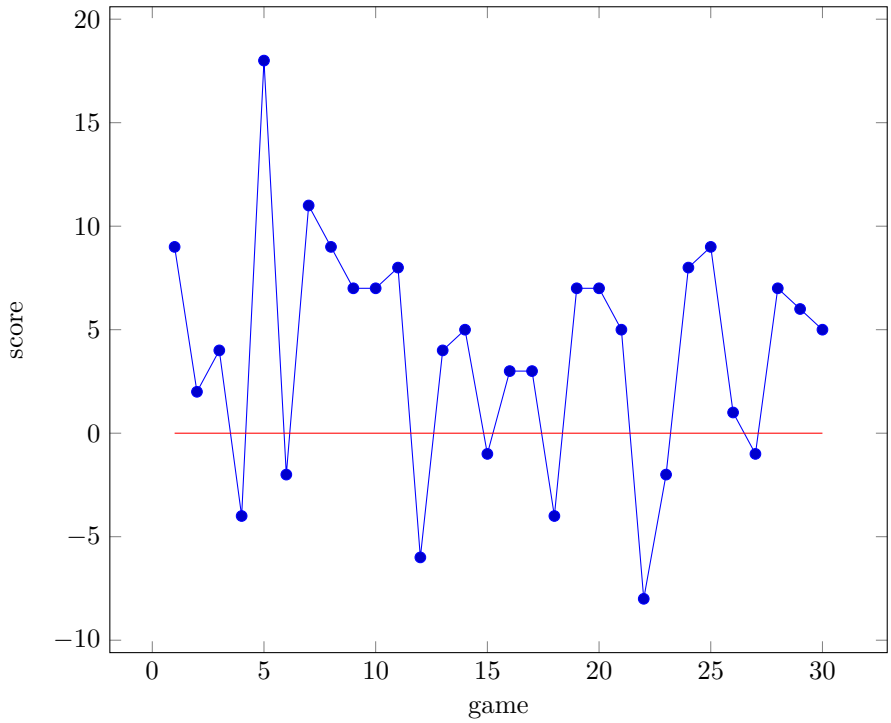
Our implementation does this efficiently by recording the score in the state itself.

The main drawback is that we will spend time to create each successor and compute the score for it. However our score calculation is pretty fast thanks to its incremental design. It is also a lot more efficient to not sort leafs, as they are never expanded and always need to be evaluated.

9. Make an agent using the successor function of the basic player (section 3.1), using your new evaluation function and cutting the tree at its root to use the evaluation function on its direct successors (you can achieve this by making `cutoff` always return `True`). Let this agent play against another similar agent using `Board.get_score` as evaluation function. Try out multiple matches and vary who plays first. How well does your evaluation function fare ?

Out of 30 games, the agent using our evaluation function won 22, the average score is 3.6 (in our favor) and the median score is 5.

Scores



3.4 Successors function

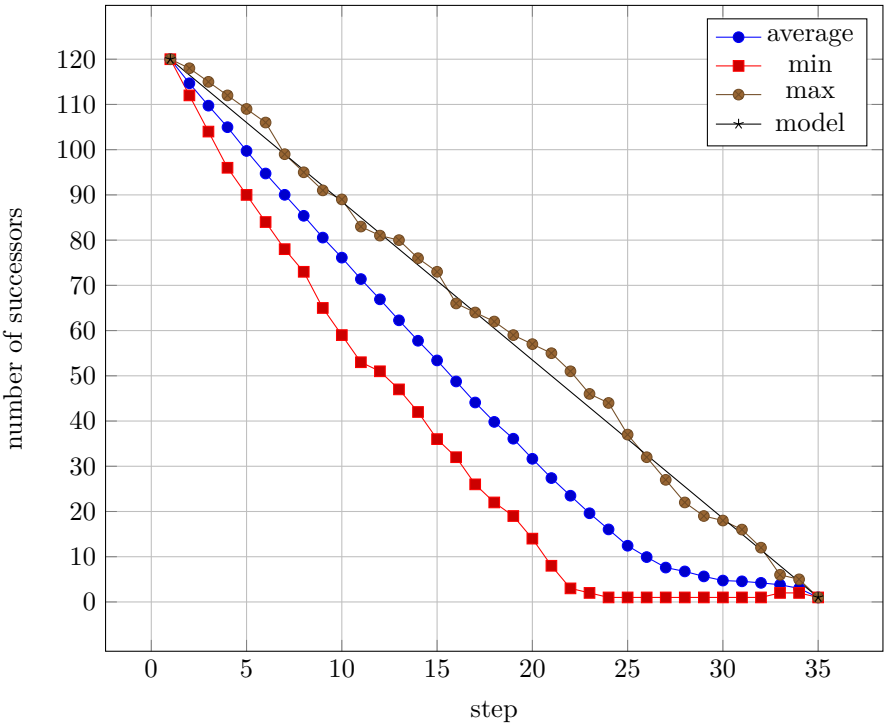
10. Give an upper bound on the number of successors for one state.

The maximum amount of successors for one state is 120. This is for the initial state, of course, because after that, this amount is decreasing. The 4×4 elements in the center of the board have 4 neighbors, so 4 moves are possible for each of them. The 4×4 elements on the borders of the boards (but not in the 4 corners) have 3 neighbors. The 4 elements in the corners have 2 neighbors. So, the total amount of successors is $16 \times 4 + 16 \times 3 + 4 \times 2 = 120$

11. From random games (at least 100), compute the average number of possible actions at each step of the game. Plot the results in a graph. What do you observe ?

The number of successors decreases at each step (or is equal, rarely). We saw the first move is one of 120 possible actions.

Number of successors :



12. Are all these successors necessary to be exhaustive (think about symmetry) ? Why ? If not, how will you consider only the necessary states ?

They are symmetric states, but these represent about 20% of all nodes in an Alpha-Beta search. In our experiment, it was not significantly better in performance to handle them.

13. If the number of successors is still too large, can you think of states that mightbe ignored, at the expense of loosing completeness ?

The initial number of successors is very large (120). We can resign to have a small depth like 2 or 3, or choose to only explore some successors. We chose the small depth, as the first moves are usually not that important.

14. Describe your successors function.

We have the `gen_successors()` function which `yield` all possible successors.

It loops over each cell, and for each with a pile (that is, not empty) on it, it loops over each neighbor :

- If the neighbor cell is not empty and the sum of their heights is ≤ 4, we add the current pile on top of that neighbor.
- Otherwise (if the neighbor cell is empty), and it contains arrows, we reverse the current pile on that neighbor cell.

In any case, the original cell in the new state is empty, and the new one is either the pile over the neighbor pile or the reversed pile.

15. On average, how deep can you explore the tree made by your successors function starting from random initial states (as generated by the constructor of the `Board` class) in less than 30 seconds ? Use an Alpha-Beta agent with the basic evaluation function and increase progressively the cut-off depth.

For the first 10 steps we can explore up to depth 4 in a few seconds. From step 11 we can explore up to depth 5 in $\pm$10s.

3.5 Cut-off function

16. The `cutoff` method receives an argument called depth. Explain precisely what is called the depth in the `minimax.py` implementation. Illustrate on an example (draw a search tree and indicate the depths).

The depth is the number of successive moves explored, before evaluating the score. With a depth of 1, only the next immediate movement is examined.

For the illustration, we will take Figure 4, in question 2.1. The first node at the top has a depth of 0, the two children have a depth of 1, their children a depth of 2, the children of those a depth of 3 and finally the leafs have a depth of 4.

17. Explain why it might be useful (for the Sarena contest) to cut off the search for another reason than the depth.

We want to respect the time allocated to our agents, so a cut-off method could check the time to stop at an appropriate time.

18. Describe your cut-off function.

Our cut-off function is inlined and check the depth and calls `is_finished()`. When there are no more successors for the current state, it means there is not any possible action to play anymore and therefore the party is over.