Technion - Israel Institute of Technology

Introduction to Artificial Intelligence

Professor Oren Salzman

TA Tal Swisa

Homework 2

Pietro BRACH DEL PREVER 921210282

Yaacov VAKSMAN 316153261

January 17, 2021

Academic Year 2021/2022

1 Dry part

1.1

Considering the board game *nine men's morris*, a possible heuristic function for the evaluation of a state is given by:

$$h(s) = number\ of\ player's\ incomplete\ mills\ -\ number\ of\ rival's\ incomplete\ mills$$

The heuristic function has some pros and cons, listed below.

Pros

- it is very easy to compute;
- it is very informative in the first part of the game when each player has to stop the rival from completing a mill and each player can place a mill wherever on the board.

Cons

- it does not take into account the fact that during the second part of the game the soldiers cannot be placed wherever on the board but must be moved from another position by means of a one-square move;
- it does not take into account the fact that the mobility of the soldiers is important also because a player who cannot move loses the game;
- it does not take into account the number of soldiers each player has on the field, which is another important parameter to take into consideration;
- it does not give a very good indication of the position evaluation in general.

1.2

We define a possible heuristic for Nine Men's Morris as a weighted sum[1] based on the following rule:

$$h(s) = Evaluation(player) - Evaluation(rival) = \sum_{i}^{N} c_i \times R_i$$

If $h(s) > 0$ white is supposed to be in a better condition, else black is.

The heuristic function we define is based on 8 contributions ($N = 8$):

[1] The motivation of using that heuristic is from the paper: Simona-Alexandra PETCU, Stefan HOLBAN, *Nine Men's Morris: Evaluation Functions*

- R1: returns 1 if a mill was closed by our player in he last move, returns -1 if a mill was closed by the opponent n the last move, returns 0 otherwise;
- R2: returns the difference between the number of my player's mills and the number of the rival's ones;
- R3: returns the difference between the number of my players' blocked pieces and the number of the rival's ones;
- R4: returns the difference between the number of my player's pieces and the number of the rival's ones;
- R5: returns the difference between the number of my player's 2-piece configurations and the number of the rival's ones (a 2-piece configuration is made by two soldiers and a free cell in a row), not accounting for the number of 2-piece configurations that compose the 3-piece configurations;
- R6: returns the difference between the number of my player's 3-piece configurations and the number of the rival's ones (a 3-piece configuration is made by two 2-piece configuration that shares a soldier);
- R7: returns the difference between the number of my player's double mills and the number of the rival's ones (a double mill is made by two mills that share a soldier);
- R8: returns 1 if a mill was closed by our player in he last move, returns -1 if a mill was closed by the opponent n the last move, returns 0 otherwise.

Different coefficients can be set for the different phases of the game.

1.3

1.3.a

The advantage of alpha-beta pruning algorithm is that it decreases the number of sub-trees explored and leaves evaluated, although the selected move will be the same selected by the minimax algorithm. Therefore, the optimality of the algorithm is maintained, but the runtime is decreased. By pruning branches, and eliminating all the computational cost of their expansion and leaves evaluation, a deeper search can be performed. This advantage is achieved by avoiding searching sub-trees of moves that will not be selected according to the current knowledge of the search tree.

Fig. 1 shows a scheme of how the alpha-beta pruning algorithm works. The algorithm explanation[2], in the cut-off version, follows (it references to Fig. 1):

[2]Based on: Hsu TSAN-SHENG, *Alpha-Beta Pruning: Algorithm and Analysis*

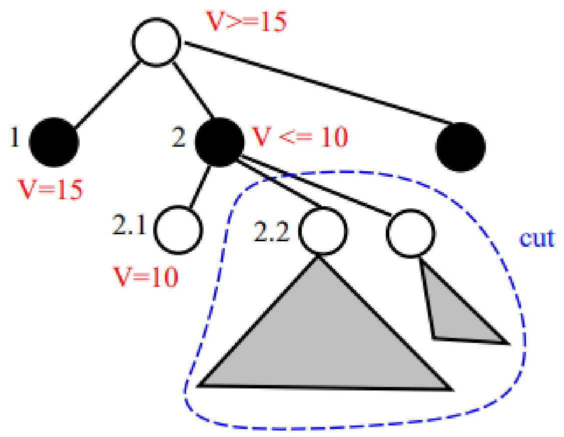


Figure 1: Explanation of the alpha-beta pruning algorithm (cut-off version).

- you explore branch at 1 and obtain the best value from it and call it *bound* (10 in the case of the figure);
- You now search the branch at 2 by first searching the branch at 2.1;
- branch at 2.1 returns a value that is $\leq$ *bound*;
- in such a case there is no need to evaluate the branch at 2.2 and all later branches of 2, if any, at all since the best possible value for the branch at 2 must be $\leq$ *bound*;
- branch 2 cannot return a value better than the one returned from the branch at 1.

1.3.b

Optimal pruning happens when the extreme (min or max) value of every node is found first: in nodes where we need to maximize the result, the highest minimax value should be found in the first child; in the nodes where we need to minimize the result, the lowest minimax value should be found in the first child, too.

1.3.c

Fig. 2 shows how the algorithm works in practice on the example provided in the handout. No sub-tree or leave is pruned by the algorithm in this case.

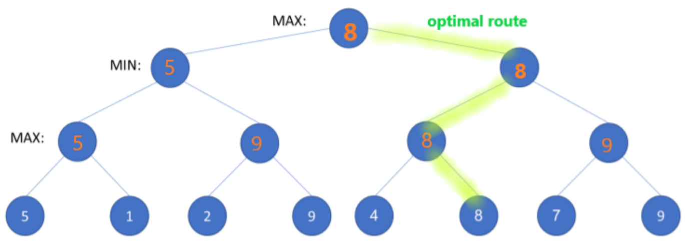


Figure 2: Example of pruning according to the alpha-beta pruning algorithm.

1.4

The alpha-beta algorithm is based on the minimax algorithm: it returns the same result, but in a shorter time thanks to the pruning of some sub-trees. Minimax is an algorithm (often implemented in a recursive form as an iterative-deepening DFS) which is used to choose an optimal move for a player assuming that the other player is also playing optimally. Minimax strategy is safe, since it discourages taking any risks. Minimax generates the whole game tree, down to the leaves. Alpha beta pruning has no effect on the evaluation of the terminal states and returns the same result of minimax, i.e. the optimal result.

However, exploring the whole search tree is impossible in most cases, because it is too large and it would require too much resources, both in terms of time and of memory. What is performed instead is a search until a certain depth d (often by means of an iterative-deepening DFS algorithm, as in our case) and the evaluation of the leaves of such trees (the last states reached and not expanded in the search). The evaluations is based on some heuristic functions. If we had a perfect heuristic we would need to perform the search and then the evaluation only one move ahead, but in reality the evaluation functions are always imperfect. The quality of the minimax algorithm, and therefore of the alpha-beta pruning algorithm, depends on its heuristic. The statement is therefore false: the alpha-beta algorithm i would be optimal only with a perfect heuristic, and this is not the case. To sum up, the terminal utilities of the final states are replaced by the evaluation function for non-terminal positions. Performing this change in the algorithm makes it non-optimal (but feasible).

Fig. 3 shows an example of a search-space where alpha-beta algorithm evaluates states at $depth = 1$ and returns the optimal decision $a1$ according the evaluation (50 is the maximum value among the values computed for the child states), while the algorithm run until the terminal states (no heuristic, no evaluation, just the utility of the terminal states) of the search tree ($depth = 2$) would have returned $a3$: considering an optimal rival, $a1$ eventually returns $utility = 40$, while $a3$ returns a terminal state utility of 42.

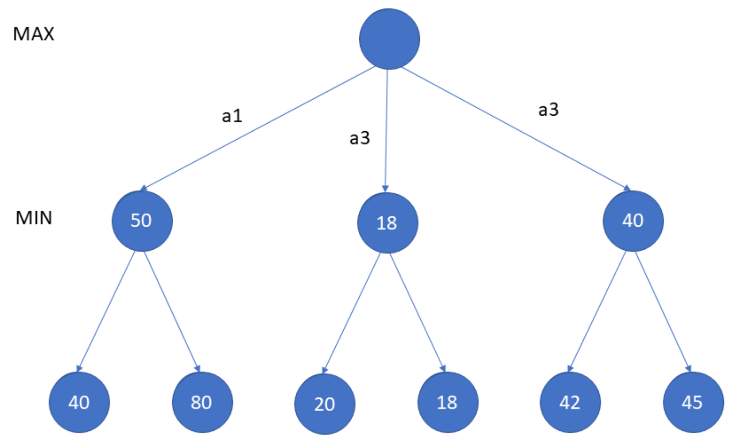


Figure 3: Example of alpha-beta algorithm with heuristic and evaluation at $depth = 1$.

1.5

Let us suppose the finite states $s1, s2, s3, s4$. In zero-sum games $U(s,1) = -U(s,2)$ or $\sum_k^N U(s,k) = 0$ sum of the utilities of all the players is zero for all states. For simplicity, let $U(s) = [U(s,1), U(s,2)]$.

1.5.a

For the sake of the reasoning, we will assume that player 2 is optimal and that the following values represent the final utilities of the four states for each player:

$$\begin{aligned} U(s1) &= [1,1] \\ U(s2) &= [5,-3] \\ U(s3) &= [0,0] \\ U(s4) &= [1,-1] \end{aligned}$$

The diagram in Fig. 4 shows the corresponding non-zero sum game.

In the example of Fig. 4, minimax still returns the optimal value and the optimal move. Minimax is optimal because there is a negative correlation between the two players i.e. what is good for player 1 is bad for player 2. Player 1 can see that, for the opponent's optimal moves, the game can terminate either in state $[1,1]$ or in state $[0,0]$: indeed, player's 2 best moves are represented by the edges leaning to left in both cases (orange arrows). So minimizing player's 1 utility in the opponent move nodes was the same as maximize opponent utility. Hence, minimax is optimal in such an example.

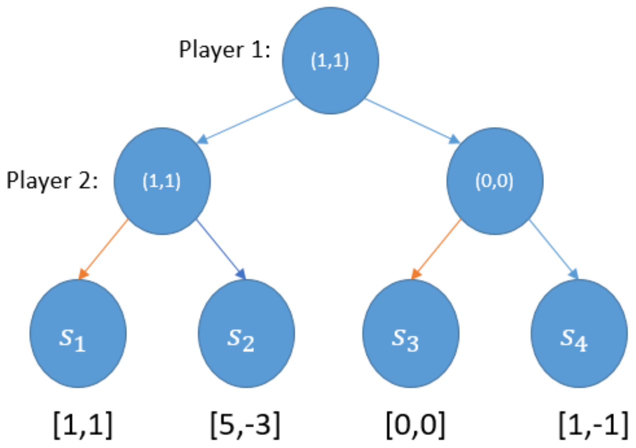


Figure 4: Example of a non-zero sum game where the optimality of minimax holds.

1.5.b

We now assume that the following values represent the final utilities of the four states for each player:

$$U(s1) = [1, 1]$$
$$U(s2) = [-1, -1]$$
$$U(s3) = [2, 0.5]$$
$$U(s4) = [-1.5, 0]$$

The diagram in Fig. 5 shows the corresponding non-zero sum game.

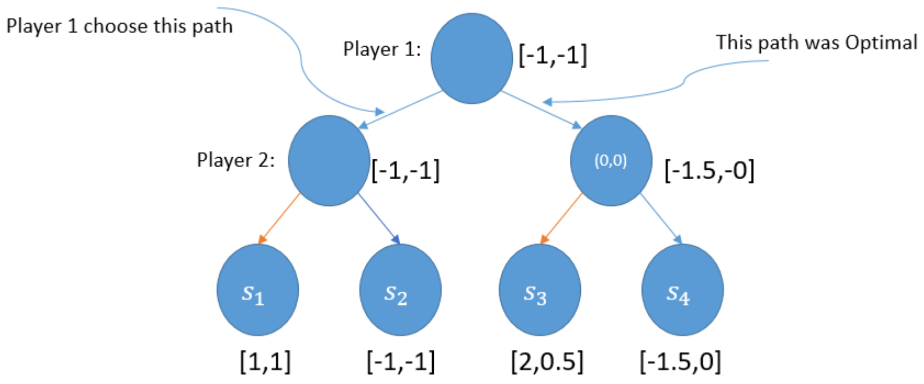


Figure 5: Example of a non-zero sum game where the optimality of minimax does no hold.

The example in Fig. 5 shows a case where minimax is not optimal, i.e. does not perform the best moves. Minimax minimizes the utility of player 1 at the opponent nodes, but the optimal path instead would be given if it tried to maximize the opponents utility at the opponent's nodes.

Considering the specific example under the scope, the minimax algorithm would compute $[-1, -1]$ and $[-1.5, 0]$ and would therefore chose to go left and get minimax value of -1, instead of -1.5. However, the opponent goes left in his nodes (orange arrows) during its turn, because that is how it can maximizes its terminal utility. Thus, considering this gameplay, player 1 will get the terminal utility of 1. After a deeper analysis, the best move for player 1 was to go right, because then the opponent will take the lest node $[2, 0.5]$ and player 1 terminal utility could be 2.

1.6

A student implements an alpha-beta algorithm and in a game against the software she noticed that the computer didn't take a winning move (in the next step) and chose another move instead.

1.6.a

Such a situation is possible because the heuristic can mislead the computer decisions and therefore avoid the wining move. In particular, the winning move has a lower heuristic value than the move eventually chosen. Probably, the used heuristic doesn't evaluate if the position represents a victory for the computer.

1.6.b

The change that we can apply is quite simple: if the alpha-beta algorithm gets a state at $depth = 1$ that is winning for the computer, then it returns that move. In Lst. 1 we can see the regular alpha-beta algorithm pseudo-code with the slight modification required.

```
alphaBeta(state):
    for successor in state.getSuccessors(): # added
        if successor.is_victory_for_player(): # added
            return successor # added
    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
```

Listing 1: Modified code: lines 2 to 4 have been added.

With such modifications we can guarantee that the winning move will always be returned.

1.7

Fig. 6 shows the search tree considered.

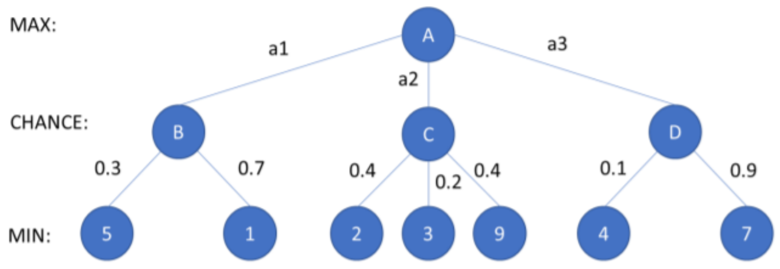


Figure 6

1.7.a

The expectimax of a chance node is calculated by a weighted sum of the utility and the probability to get it.

$$U(B) = 0.3 \cdot 5 + 0.7 \cdot 1 = 2.2$$
$$U(C) = 0.4 \cdot 2 + 0.2 \cdot 3 + 0.4 \cdot 9 = 5$$
$$U(D) = 0.1 \cdot 4 + 0.9 \cdot 7 = 6.7$$

The max operator takes the path with the best value at the chance node:

$$U(A) = U(argmax(U(i)) = U(D) = 6.7$$

1.7.b

After this computation, it derives that action D is the one we choose, therefore opting to explore the sub-tree through edge $a3$.

1.7.c

It is not possible to trim in expectimax in the same way we prune sub-trees in alpha-beta algorithm. This is because it is not possible to calculate the expectimax value until all the nodes are evaluated. The pruning can be performed only if a higher and a lower bound of the leaves are available. The regular expectimax is calculated by means of Eq. 1, where p, u are the probability and utility respectively.

$$Expectimax(state, action) = \sum_{i \in succ(state,action)} p_i u_i \tag{1}$$

If we got an upper and lower bound of the utility (u_{min}, u_{max}) we can predict the bounds of Expectimax node after revealing one possible successor:

$$Expectimax(state, action) \in [p_1 \cdot u_1 + (1 - p_1) \cdot u_{min},\ p_1 \cdot u_1 + (1 - p_1) \cdot u_{max}]$$

On the other had, if an upper and lower bound are not available, it is not possible to prune. An example is shown in Fig. 7. Assume the first action was been evaluated and the

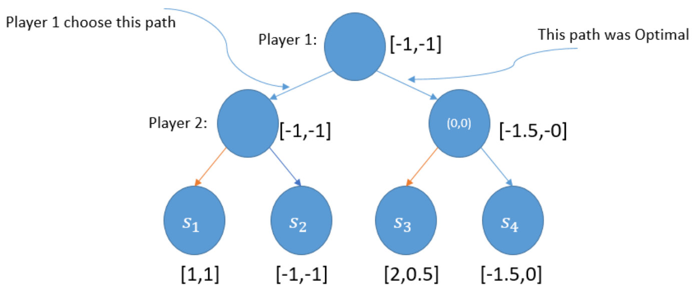


Figure 7: Example where pruning in expectimax compromises the correctness of the algorithm.

expectimin returned $5 \times 0.3 + 1 \times 0.7 = 2.2$. Now we need to evaluate the next action ($a2$). The first successor looks promising with high probability and value we already get 2.4 on our $a2$ chance node and it seems that we do not need to check the next leaves. However, what if the value in D was negative and equals to -2? Than the result of the expectimin operator on the second action would be equal to 2 (smaller then $a1$ action). That's why a naive approach will generally not work. Instead, if we had some bound on the leaves (e.g. positive value utility), we could say,as in the case of the above example, that after checking one leaf at the $a2$ action we could guarantee that action $a1$ is better.

1.8

Lst. 2 shows the modification to the pseudo-code of alpha-beta to introduce a harder pruning, paying with returning the non-best results, worsened possibly by the factor ϵ.

```
1      alphaBeta(state):
2          return maxValue(state, -INFINITY, INFINITY, 0)
3
4      maxValue(state, alpha, beta, depth):
5          if cutoffTest(state, depth):
6              return utility (state)
7          value = -INFINITY
8          for successor in state.getSuccessors():
9              value= max(value, minValue(successor, alpha, beta, depth + 1))
10             if value + eps >= beta: # modified
11                 return value  + eps # modified
12             alpha= max(alpha, value).
13         return value
14
15     minValue(state, alpha, beta, depth):
16         if cutoffTest(state, depth):
17             return utility(state)
18         value = INFINITY
19         for successor in state.getSuccessors():
20             value=min(value, maxValue(successor, alpha, beta, depth + 1)
21             if value - eps <= alpha: # modified
22                 return value - eps # modified
23             beta min(beta, value)
24         return value
```

Listing 2: Modified code: the changes are in lines 110-11 and 21-22.

The new algorithm returns a result different from the minimax in the case shown in Fig. 8. In this case, alpha-beta prunes away the sub-tree after edge $a3$, and returns $a1$ with the optimal minimax value of 40. Minimax, on the other hand, returns $a3$ with optimal minimax value of 42.

1.9

$\boldsymbol{d_2 = \mathcal{O}(2 \cdot d_1)}$

1.9.a

The time complexity of minimax is $O(b^d)$, being function of the number ot leaves in the deepest searching layer. If we get the `rival_move` we don't need to expand the whole opponent possible moves, but we just need to run this procedure and we already know among all the possible child nodes which one will be chosen by the rival through `rival_move`. Thus the tree

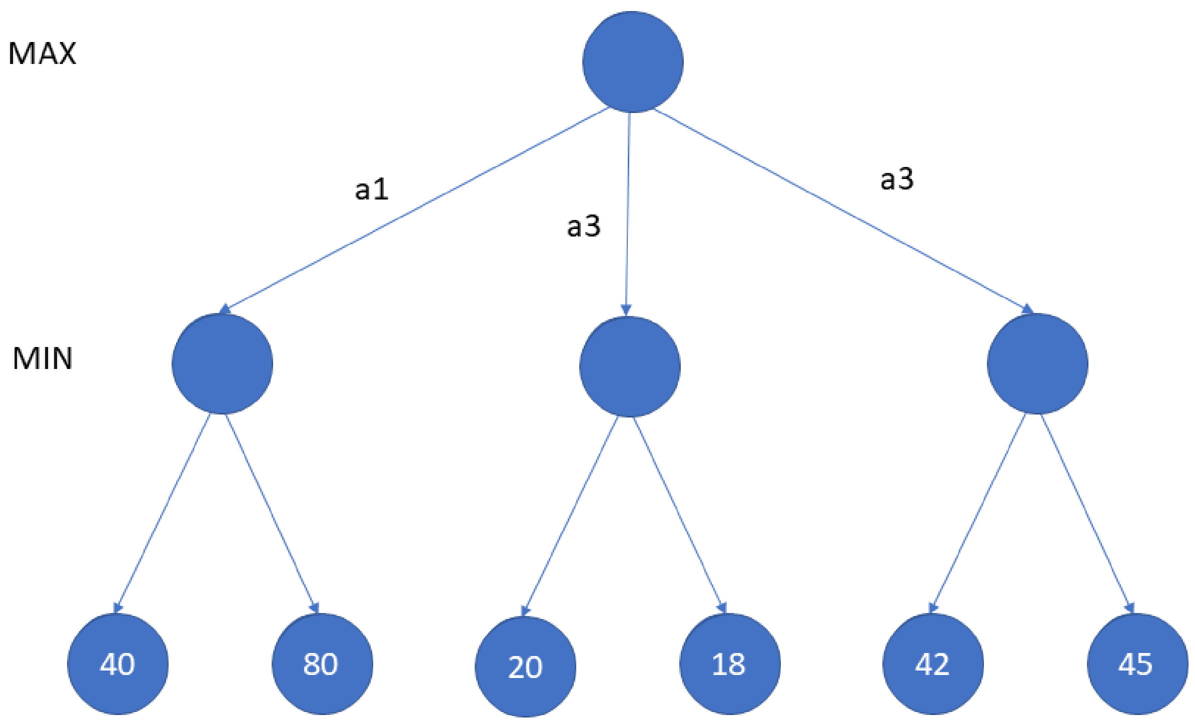


Figure 8: With $\epsilon = 2.5$, $42 - \epsilon = 39.5 < 40$ and therefore alpha-beta modified prunes away the sub-tree child of edge $a3$.

size multiply by b every two layers of actions: the branching of the tree, i.e. the increase of the number of branches by a factor b, happen every two layer, and not one as before, since the opponent moves do not change the number of branches.

$$O(b^{d_1}) = O(b^{d_2/2})$$
$$d_2 = 2 \cdot d_1$$

1.9.b

The minimax value is the highest value that the player can be sure to get without knowing the actions of the opponent, i.e it is the lowest value the rival can force the player to receive. Given state s, the ratio between the value of the minimax with use of the procedure and the value of the minimax without the use of the procedure when both runs are limited to depth d is give by Eq. 2.

$$v_i = max(min(v_i)) \tag{2}$$

Indeed, minimax strategy assumes a optimal opponent, i.e. it assumes that the rival will chose its best scebario, which corresponds to our worst sceanrio, but the rival does not necessarily chooses the action with the lowest utility (the one with minimum minimax value). Thus, minimax value without using `rival_move` procedure is always less or equal to the value of minimax with using `rival_move` procedure.

1.10

1.10.a

The student is wrong. The algorithm may have always returned local non-global minima each time he ran the SAHC algorithm. The convergence on the global minimum depends only on the starting point, and there is asymptotic guarantee of convergence for the number of starting points tending to infinity. However, it is not possible to state if the global maximum can be reached within a finite number of searches.

In this specific case, the information available about the search space and the results of some run is available:

- the search space has a size of 10×12;
- two local minima are found after 1000 searches, and their values are 0.9 and 5.8;
- the average of the value of the maxima found is 3.2 (which means 0.9 was found approximately 531 times, while 5.8 was found 469 times);
- the algorithm took on average 5 steps to converge.

However, this data is not informative in terms of probability that 5.8 is actually the global maximum. For instance, Fig. 9 shows an example of a search space corresponding to the information provided (size 10×12), where there are 3 local maxima, two of them corresponding to the ones found by the student running SAHC, and one which is the globabl maximum and is not found by the student. If the algorithm run by the student randomly chooses only points of value 0.1 and 0.5 as starting point (in red), then only two local maxima would be reached (in green) following the path in according to the experience of the student, bu non of them would correspond to the global maxima which is 15 (in blue). Again, there is no information that ensures us that this condition does not correspond to what happen in the hypothetical situation of he student. Therefore, we conclude that the maxima found by the student could not correspond to the global maximum.

0	0.5	1	2	3	4	5.8	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	15	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0.9	0.7	0.6	0.3	0.2	0.1	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0

Figure 9: Example of a possible search space.

1.10.b

Verifying the answer of the student in such conditions wold be easy: instead of running the algorithm from random points, the algorithm should be run 120 times only, starting each time from a different point on the search space: in such a way, the algorithm would converge to the global minimum by the end of the run (and start from the global maximum once).

If we had to verify the student's result with a non-deterministic algorithm instead, another possible algorithm would be Simulated Annealing. The problem of SAHC algorithms is that they try to maximize the state and never makes gradient-descending moves toward states with lower value. In contrast, SA allows to not converge straight away to the local solution, but moving to states with lower value.

1.10.c

An example of search space where SAHC finds the solution with low probability is shown in Fig. 10. Given the total number of cells in the search space $n = 20$, the probability of convergence of SAHC to the globabl maximum 15 (red) is given by $2/n = 2/20 = 0.1$, while the probability to converge to the local minimum 5.8 (green) is equal to $1 - 2/20 = 1 - 0.1 = 0.9$.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5.8	1	15

Figure 10: Example of a possible search space where SAHC can find the global maxima with low probability.

Another example, with a non-deterministic algorithm instead, of a search space where SAHC does not find the optimal solution with high probability is shown in Fig. 11. Only if the initial starting point gets somewhere in the red interval (along x-axis) it will climb to the global maximum, otherwise it will never reach the global maximum. If the initial starting point will be somewhere along the purple lines - SAHC will try to maximize the value and reach a local maximum. So with high probability SAHC will reach one of the "sinusoidal" peak. In contrast, simulated annealing algorithm can descends the gradient and find the global maximum. Considering again the example of Fig. 11, SA can descend along the purple lines and reach the global maximum with much higher probability than SAHC.

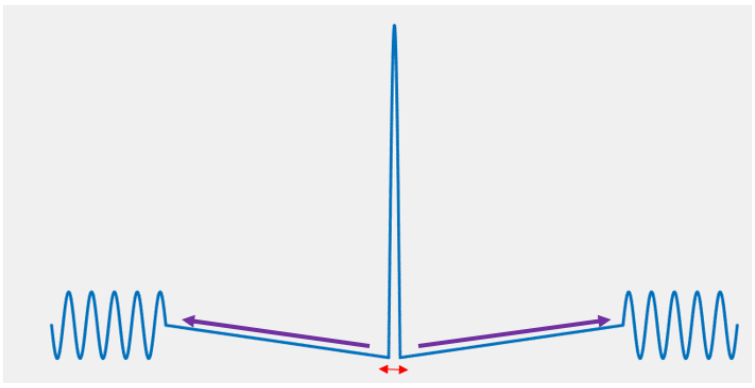


Figure 11: Example where SAHC does not find the optimal solution.

2 Wet part

2.1 Part E

2.1.a

The heuristic used for the Minimax player is based on 7 contributions:

- R1: returns 1 if a mill was closed by our player in he last move, returns -1 if a mill was closed by the opponent n the last move, returns 0 otherwise;
- R2: returns the difference between the number of my player's morrises and the number of the rival's ones;
- R3: returns the difference between the number of my players' blocked pieces and the number of the rival's ones;
- R4: returns the difference between the number of my player's pieces and the number of the rival's ones;
- R5: returns the difference between the number of my player's 2-piece configurations and the number of the rival's ones (a 2-piece configuration is made by two soldiers and a free cell in a row), not accounting for the number of 2-piece configurations that compose the 3-piece configurations;
- R6: returns the difference between the number of my player's 3-piece configurations and the number of the rival's ones (a 3-piece configuration is made by two 2-piece configuration that shares a soldier);
- R7: returns the difference between the number of my player's double mills and the number of the rival's ones (a double mill is made by two mills that share a soldier).

In our case we set different coefficients for the two different phases of the game. For the first phase we have that the heuristic value assigned to he evaluated states is given by:

$$E(s) = 18 \times R1 + 26 \times R2 + 1 \times R3 + 9 \times R4 + 10 \times R5 + 7 \times R6 + 0 \times R7$$

while for the second phase of the game we have:

$$E(s) = 30 \times R1 + 31 \times R2 + 10 \times R3 + 11 \times R4 + 5 \times R5 + 0.5 \times R6 + 8 \times R7$$

2.1.b

The Player submitted for the competition corresponds to the Alpha-beta player coded for the other exercises, with the strategy implemented for the global time management, as explained later in the section 2.1.c. No particular improvement was implemented in the heuristic, since we reckoned that the heuristic was already quite informative.

2.1.c

The runtime of the `make_move` method was managed according to the following ideas.

global_time limit strategy We get the global time and calculation average time for single move, assuming N=35 moves:

$$avg\_time = \frac{global\_time}{N}$$

We want to take into account the fact that for higher turns we need less time. so we are using a turn factor function to factorize the move time.

Factor function:

- We want a monotonically descending function
- at turn=0 we want to get $turn\_factor = 1$
- at turn=N (the average number of moves) we want to remain with 1 second for move, i.e. $turn\_factor(N) = 1/avg\_time$
- so we used an exponent $turn_factor(turn) = e^{-\lambda \cdot turn}$
- from the above conditions, easy to see that: $\lambda = -(\frac{1}{N})\ln(\frac{1}{avg\_time})$

With $move\_time$ calculated every turn we use it to set a time limit for a single turn.

one-move_time limit strategy We saw that as the turn number is higher most of the time it takes less time to perform a move. That was accounted in a Global Time strategy. In addition we saw that the difference between two adjacent depth is around 20. This fact is not so strange cause we know that asymptotically the time of DFS is $O(b^d)$ where b is the branching factor and d is the depth, so asymptotically the ratio between two adjoint depth is $\sim b$:

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

Fig. 12 shows how it resulted from experiments that the time ratio of two adjacent depths is bound by 25.

Our code strategy:

- Every depth we measure the time it took to perform the Minimax or Alpha-beta loop ($iter\_time$)
- We continue to the next depth if the $remain\_time \; > \; 40 \cdot iter\_time$
- We took the factor of 40 (to be on the safe side)

- o In addition, we took safety factors of few seconds to be sure that we have enough time to send our move to the game_wrapper

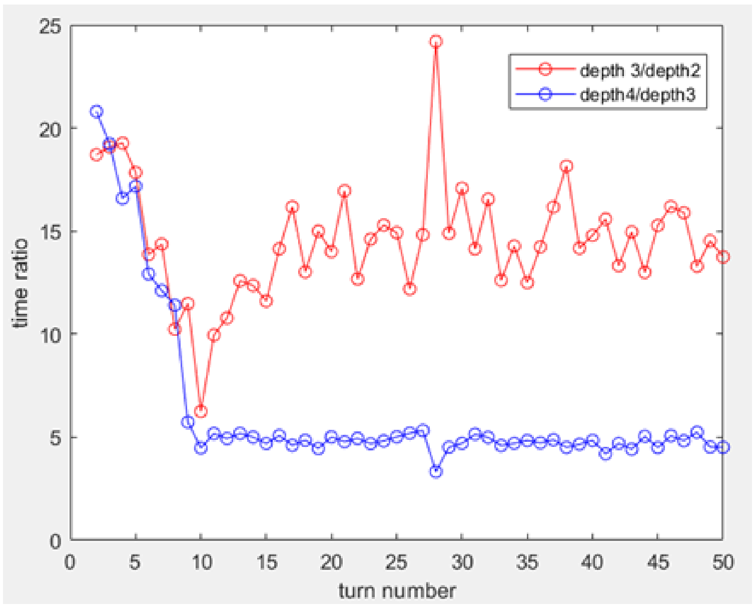


Figure 12: The time ration of two adjacent depths is bound by 25.

2.2 Part F

2.2.a

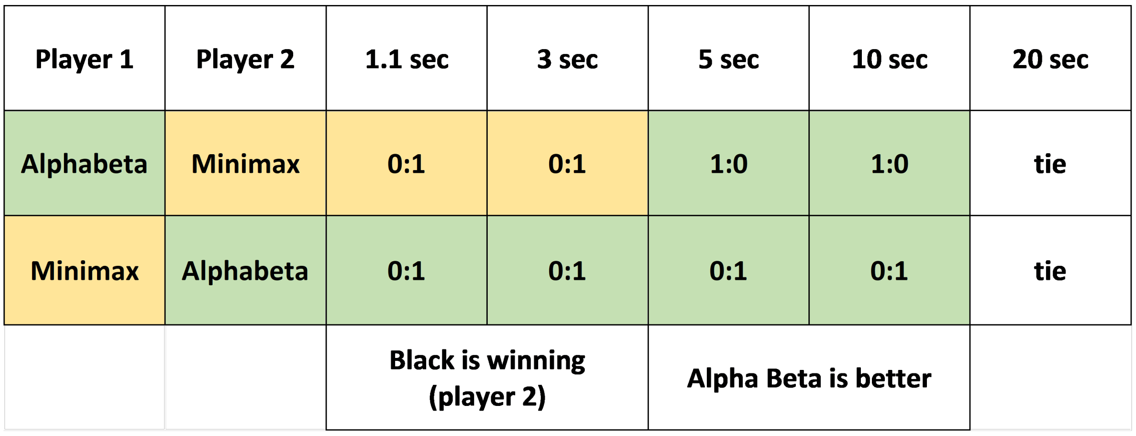
The value of the initial position has been proved to be a draw[3]. Therefore, if the time-per-move provided is long enough and if both players play with the same heuristic and manage to reach a depth of 3-4 in each turn search, then tey do not allow the rival to create a mill and the game will eventually end in a tie. This can explain the 20 sec result.

For a short time-per-move provided, both the players cannot get deep in the search and most of the time they stay in the same one (1.1 sec case). We assume that our heuristic makes it an easier game for Black (player 2), in the sense that it is better defend and stop the rival than attack and take initiative.

For medium-range time per move (5 sec and 10 sec) Alpha-beta is winning. It is enough time for alpha-beta to get more depth than the minimax player and winning the game.

The results are shown in Fig. 13.

[3]Ralph GASSER, *Solving Nine Men's Morris*



Player 1	Player 2	1.1 sec	3 sec	5 sec	10 sec	20 sec
Alphabeta	Minimax	0:1	0:1	1:0	1:0	tie
Minimax	Alphabeta	0:1	0:1	0:1	0:1	tie
		Black is winning (player 2)		Alpha Beta is better		

Figure 13: Rsults of the experiments for different time-per-move limits.

2.2.b

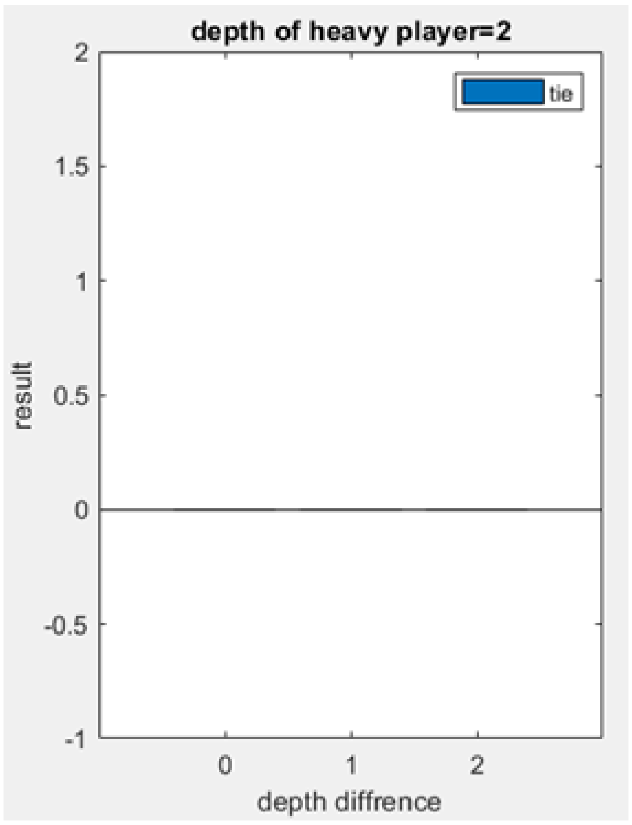
Results For *depth* = 3 we won with our Heavy Player in all the depth differences. For *depth* = all the games ended in a tie (Program paused).

Discussion Depth 3 experiments are understandable - the light heuristic was not informed enough - it does not have parameters such as – the difference between number of blocked soldiers, double mills, - piece configurations. So even the light player get higher depth, he is so far away from the endgame/terminal positions and without good evaluation of the positions he is loosing to the more informed player.

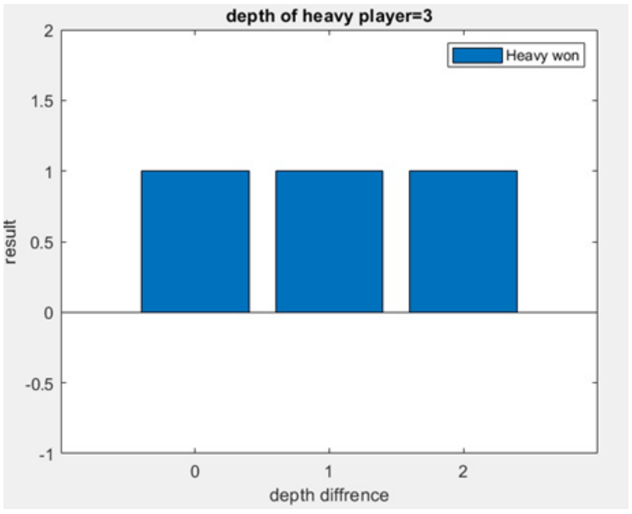
Depth 2 is strange, especially after the discussion of the depth 3 results. In all the games we stopped the game cause it's just gets in a loop where no one want to loose and no one going for a new mill. The more informed player is usually in a better position but he can't win. The explanation is that for winning he need to move one of the soldiers in a mill to construct a mill in the next move. But our heavy player is a two depth player so it only calculate his move and the rival's one, so he can't reach the mill (which constructed only in *depth* = 3).

The conclusion is that *depth* = 2 is not enough for our players and heuristic.

Te plots of the results concerning this are shown in Fig. 14.



(a) $depth = 2$



(b) $depth = 3$

Figure 14: Heavy player for $depth = 2, 3$.