2ID90 International Draught assignment
Report

Group 98 :	Lucas Giordano	1517317
	Auguste Lefevre	1517600

September 27, 2020

Contents

1 Introduction

In this assignment we were asked to build a computer player for the game of *International Draughts*. In short, International Draughts is a two player strategy game played on a 10x10 board of which only the dark fields are used to initially position 20 white pieces and 20 black pieces. The players in turn move a piece of their own color diagonally over the board. The white player starts the game. Pieces of the opponent may be captured by jumping over them to an empty field. Finally, the player without any remaining moves looses the game.

Figure 1: An international draughts board with pieces in initial position

The goal of this assignment is then to write a draught playing agent using different IA method. Our work can be decomposed into the following 3 parts :

- Firstly we have to implement and extend the basic *Alpha-Beta Algorithm*.
- Secondly we must implement an *evaluation function* able to evaluate the state of the draughts game.
- Finally we integrate these two functions in framework for playing a draughts tournament.

To achieve these objectives we based our work on different resources. Our two main resources were of course the lecture notes and the book *Introduction to Artificial Intelligence* [1]. The presentation made by Wieger Wesselink on *Computer Draught Evaluation* [4] also give us some insights on how we can improve our program by implementing custom extensions based on pro-player tactical. For example we learned that material advantage was very important (way more than in chess), that formation is also an important factor or that *tempi* need to be taken into account. Finally we also used an very well written article on Draughts AI [3]. We will discuss these features and their implementation later. For the moment, we will talk about the main algorithm of our Draught player : *the alpha-beta algorithm*

2 Alpha-Beta

In computer science, especially in IA and game theory, the alpha-beta pruning is an algorithms that seeks to decrease the number of node that are evaluated by the min-max algorithms in its search tree. This type of algorithm is commonly use for machine playing a 2 players games (chess,go etc.), in our case the International Draughts game.

It is an upgrade of the simple min-max algorithm in the way that its purpose is to skip branches of which we know will not influence the decision. For this, during the tree search, two variables alpha and beta are used to store the minimum score the player is currently assured to have and the maximum score the opponent is assured to have. In our case the score is computed by our evaluation function (cf. Evaluation). It makes sense as the player wants to maximize the value, it will not play any move scoring below alpha. That means that when $beta \leq alpha$, it's not worth exploring the rest of the branch anymore. Then, the benefits of this algorithms resides in the fact that the search time can be limited to the more promising subtree, and a deeper search can be performed in the same time. With an (average or constant) branching factor of b, and a search depth of d we obtain a worst case performance of $O(b^d)$ because the maximum number of leaf node positions evaluated is equal to $O1(b \times ... \times b)$.

Our implementation Alpha-Beta pruning algorithm is divide in 3 parts. The first one automatically chooses the white player as maximizing player and the black player as minimizing player. See Algorithm 1

Algorithm 1 AlphaBeta

```
1  int AlphaBeta(node, alpha, beta, depth) {
       if it is white to move then do /* determine if it's whit player to move or black one */
3          return alphaBetaMax(node, alpha, beta, depth); /* Case we are playing the white player, maximize */
       else do
5          return alphaBetaMin(node, alpha, beta, depth); /* Case we are playing the black player, minimize */
   }
```

Now, in the case we are playing with the white pieces we want to maximize our evaluation. To do so we use the 2 AlphaBetaMax Algorithm. As said before we will iterate over all possible moves and evaluate which one is the more benefits depending of our evaluation function (cf. Evaluation). You can observe at line 15 that we verify the condition $beta \leq alpha$ to be able to prune nodes when possible (to decrease the running time as described before and avoid useless computation).

Algorithm 2 AlphaBetaMax

```
int AlphaBetaMax(node, alpha, beta, depth) {
2         state := get the current state of the node   /* first we get the current state */
          bestMove := null
4         /* end condition */
          if state is the final state do return -WINNING_SCORE
6         if depth<=0 do return evaluation of the current state
          else do {
8             moves = state.getMoves()
              /* Do not decrement current depth if ther is only 1 move possible /*
10            if moves.size() == 1 do depth += 1
              /* consider each possible move */
12            for each move M from the list of all possible moves at this state) do {
                  state do move M
14                childNode := clone of the actual state  /* get a fresh copy of state */
                  eval := alphaBetaMin(new DraughtsNode(child), alpha, beta, depth-1)
16                if eval > alpha do {
                      alpha = value /* update alpha */
18                    bestMove = M /* updtae the best move */
                      node.setBestMove(M) /* set the best move of this node to M*/
20                }
                  state.undoMove(M) /*undo the move to renitialize the state */
22                if alpha >= beta do return beta /* prunning/cut off */
              }
24        }
          return beta
26  }
```

Finally, in the case we are playing the black pieces we want to minimize our evaluation. To do so we use the 3 AlphaBetaMin Algorithm. You can observe that this algorithm is very similar to 2 AlphaBetaMax Algorithm.

Algorithm 3 AlphaBetaMin

```
int AlphaBetaMin(node, alpha, beta, depth) {
2          state := get the current state of the node   /* first we get the current state */
          bestMove := null
4         /* end condition */
          if state is the final state do return -WINNING_SCORE
6         if depth<=0 do return evaluation of the current state
          else do {
8             moves = state.getMoves()
              /* Do not decrement current depth if ther is only 1 move possible /*
10            if moves.size() == 1 do depth += 1
              /* consider each possible move */
12            for each move M from the list of all possible moves at this state) do {
                  state do move M
14                childNode := clone of the actual state  /* get a fresh copy of state */
                  eval := alphaBetaMax(new DraughtsNode(child), alpha, beta, depth-1)
16                if eval < beta do {
                      beta = value /* update alpha */
18                    bestMove = M /* updtae the best move */
                      node.setBestMove(M) /* set the best move of this node to M*/
20                }
                  state.undoMove(M) /*undo the move to renitialize the state */
22                if alpha >= beta do return alpha /* prunning/cut off */
              }
24        }
          return alpha
26  }
```

Using these 3 algorithms 1, 2 and 3, we can determine what is the best move to win the game either we play the white or the black pieces. But if we look closely to the Algorithms 2 or 3 we observe that we use 2 different methods not explained yet. The first one and the most evident one is the evaluate function. In fact we said that the goal of the alpha-beta algorithm is to maximize the score of our player. But how do we compute the score ? With an evaluation function. We will talk about it later, but first I would like to discuss a bit about the second method used which is an improvement done to the alpha-beta algorithm : the iterative deepening.

3 Iterative Deepening

As described in the book [1], '*The search for a solution in an extremely large search tree presents a problem for nearly all inference systems. From the starting state there are many possibilities for the first inference step. For*

each of these possibilities there are again many possibilities in the next step, and so on.' In class we analysed different algorithms of search in tree, the Breadth-first search, the Uniform cost search, Depth-first search and finally the Iterative Deepening. As 1 picture is worth 1000 words we decided to not re-do a summary of those algorithms and add a relevant figure (2) instead.

	Breadth-first search	Uniform cost search	Depth-first search	Iterative deepening
Completeness	Yes	Yes	No	Yes
Optimal solution	Yes (*)	Yes	No	Yes (*)
Computation time	b^d	b^d	∞ or b^d	b^d
Memory use	$\boldsymbol{b^d}$	$\boldsymbol{b^d}$	$\boldsymbol{bd}$	$\boldsymbol{bd}$

Figure 2: Comparison of the uninformed search algorithms. (*) means that the statement is only true given a constant action cost. d s is the maximal depth for a finite search tree

Observing the figure 2, we can easily say that Iterative Deepening is the winner cause it gets the best grade in all category. This is the reason why we used it in our alpha-beta pruning algorithm. The idea behind Iterative Deepening is to begin with the depth-first search with a depth limit of 1. If no solution is found, we raise the limit by 1 and start searching from the beginning, and so on. To do so we need to add a *depth* parameter in our algorithms (that you can find in algorithms 1, 2 and 3). In a classic implementation we also have to add a limit attribute. It is also say that the end condition of the while loop (in our case the for loop) should be similar to *NewNodes != null And Depth < Limit.*

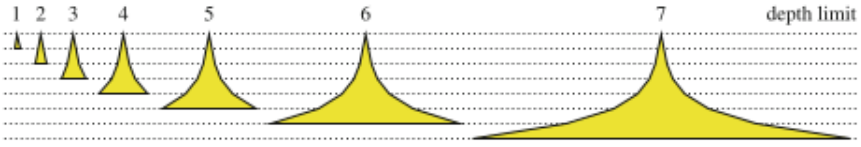

Figure 3: Schematic representation of the development of the search tree in iterative deepening with limits from 1 to 7. The breadth of the tree corresponds to a branching factor of 2

As there is no proper Iterative Deepening algorithm implemented in our project but more an integration of it inside the alpha-beta algorithm and the getMove method we will not rewrite those algorithms but directly cite them to see when and how we implemented the iterative deepening search inside those algorithms.

- "depth and limit attribute" : Our algorithms 1, 2 and 3 all have a depth attribute to know in which situation are they. When we call the first time algorithm 1 we give it a depth value which is actually the limit of the iterative deepening search.
- "recursion" : The recursive part of the deepening iterative algortihm can be found inf the getMove method. In this method we implemented a for loop that repreduce exactly the effect shown in 3. The idea is to first search a solution in a tree of depth 1. After this search, if no excepetion have been raised (due to time or other thing) we search for the best move in a tree of depth 2 and so on until we reach the maxDepthlimit. We can also determine treshold and stop the iteration/search when a solution higher than this treshold is found, but we decided to implemented this feature because we could miss some 'bestMove Test' of the automatic grading.
- "end condition" : As said before, we want to terminate if either we reached the depth limit or if we are in a final state situation. This condition is described at the beginning of the 2 algorithms and 3 at line 5 with the if statement. As I explained before we did the iteration backward so the condition *depth < Limit* became *depth = 0.* You also find the end of the condition in the for loop of the getMove method. In fact this for loop start at depth = 1 and end at depth = maxSearchDepth.

The trick of using a backward implementation and only 1 additional attribute instead of 2 has been found on this github implementation [2] of an Iterative Deepening Alpha-Beta Algorithm done in python.
We have seen how we implemented the alpha-beta algorithm in our project; we also have seen how we optimized it using another well known algorithm : the iterative deepening. But for both of them we spoke of evaluation to find the best solution. It is now the time for us to explain in details what is an evaluation function and how we implemented it in our project.

4 Evaluation

We have seen that using the alpha-beta pruning algorithm combined with iterative deepening we obtained a very efficient "search tree" algorithms that decrease the number of nodes that are evaluated (*alpha-beta pruning)* and that searches at shallower depths, giving move-ordering hints and helping produce cutoffs for higher depth searches much earlier than would otherwise be possible (*iterative deepening*). But to estimate the value or goodness of a position (usually at a leaf or terminal node on the tree) we need an *evaluation function.* The goal of an evaluation function, also called heuristic evaluation function, is to represent the relative probability of winning if the game tree were expanded from that node to the end of the game. To do so, different strategies are possible.

As described in the book [1] there are fundamentally two approaches based on the same idea. We need 3 fundamentals parameters to represent our evaluation function :

- the value estimated at a certain state of the game or 'how good is the position' : $f(X)$
- a set of differents features used to evaluate the position : $\phi(X)$
- a vector of parameter (weights) to learn : ω

Using these 3 parameters we can define our evaluation function as : $f(X) = \phi(X)^\intercal \omega$

In our case, we used a lot of different features :

- number of pieces : ρ_{nbrp}
- number of king : ρ_{nbrk}
- compactness : ρ_c
- tempi : ρ_t
- center position : ρ_{cp}

We'll described these features in detail later. For the moment there is one question left, how to determine weights. As we said it before there are two fundamentals approaches to create the evaluation function and the main difference remains on how we compute weights. The classical way uses the knowledge of human experts and we choose this way as we got the opportunity to attend the lecture of Wieger Wesselink on *Computer Draught Evaluation* [4]. Of course with more knowledge and more time it is also possible to combine this way with some machine learning methods as reinforcement learning *(Chapter 10 of the book [1])* or gradient descent but we thoughts it was out of the scope of this project.

First, we defined the most important feature : the number of pieces ρ_{nbrp}. We attributed a weight of 10 for this feature which is our 'referential'. Then for the number of king ρ_{nbrk}, which is also very important, we attributed a weight of 30 as a king is worth 3 pieces. For the compactness feature ρ_c, which is less decisive than the number of piece / king we only attributed a weight of 2. For the tempi feature ρ_t we have to be careful as value may vary a lot. To limit the impact of tempi over other more important features we decided to attribute a weight of 1 only. For the center position feature ρ_{cp} we learned that this is useful at the end of the game but that sometimes this is also interesting to have pieces on the bounds of the board since they can not be captured so we decided to only attribute a weight of 2.

NB : those weights have been tested to ensure the efficiency of our evaluation function (cf. Result)

Here is a more visual recap of the situation :

Features	Weights
number of pieces : ρ_{nbrp}	10
number of king : ρ_{nbrk}	30
compactness : ρ_c	2
tempi : ρ_t	1
center position : ρ_{cp}	2

4.1 Number of pieces

This is probably the most important feature and also the most obvious. Because a player loses the game when they have no pieces left and because we can assume their position is better when they have more pieces than the opponent, this feature is considered as the most relevant one. Also, Wieger Wesselink insisted on the fact that losing a piece usually means losing the game so piece sacrifices are rare. Here is how we can describe the evaluation function (1) of the number of pieces :

$$H(s) = \begin{cases} -\infty & \bigcirc(s) = 0 \\ \infty & \bullet(s) = 0 \\ \bigcirc(s) - \bullet(s) & otherwise \end{cases}, \tag{1}$$

where for board state s, $\bigcirc(s)$ and $\bullet(s)$ are the number of white and black pieces, respectively.

4.2 Number of Kings

As Wieger Wesselink explained to us during his presentation [4], a king is roughly equivalent to 3 pieces. Based on the same idea of the number of pieces, we added a feature which evaluate the number of kings. It seems also really obvious that this is a relevant feature. Here is the evaluation function (2) of the number of kings :

$$H(s) = \Big\{ \bigcirc(s) - \bullet(s) \quad every - time \ , \tag{2}$$

where for board state s, $\bigcirc(s)$ and $\bullet(s)$ are the number of white and black kings, respectively.

PS : to optimize the running time of our evaluation function we merged the number of Pieces / Kings together for the 2nd sumbission.

4.3 Compactness

As learned in the Draughts AI article [3], the more space is left between pieces, the more vulnerable they are. So by the *compactness* feature we mean counting all the free squares around one player's pieces and the other and then do the difference between the two formations. The result determines how compact one playerâs formation is compared to the other. A lower value means a more compact setup. As we count the number of free space, this feature is analyze as a *malus* rather than a bonus. We can describe the evaluation function (3) of this feature as follow :

$$H(s) = \Big\{ -(\bigcirc(s) - \bullet(s)) \quad every - time \ , \tag{3}$$

where for board state s, $\bigcirc(s)$ and $\bullet(s)$ are the number of free space next to white and black pieces, respectively.

For the second submission we decided to divide s, $\bigcirc(s)$ and $\bullet(s)$ by the number of white and black pieces respectively because we din't want to creat an imbalance du to the number of piece for each player since this parameter is already represented by the getPiece and getKings features

4.4 Tempi

The notion of *tempi* was introduced during the presentation of Wieger Wesselink. The idea behind tempi is that it is important to advance your pieces. The basic way of computing tempi is to count pieces on the first row as 1, on the second row as 2, etc. So in our case we compute it for both white and black pieces and then do the difference. Here is the pseudo-code of the Tempi Algorithm (4) :

Algorithm 4 Tempi

```
int tempi(state)
        white := 0
        black := 0
        board := place of pieces on the board    /*depends of state */
        for each row do
            sumRowWhite := 0;
            sumRowBlack := 0;
            for each colonn do
                if there is a white piece then do
                    increment sumRowWhite
                if there is a black piece then do
                    increment sumRowBlack

            black += sumRowBlack * row;       /*black occupies top part of the board*/
            white += sumRowWhite * (11-row)   /*white occupies bottom part of the board*/

        return white - black;
```

4.5 Center position

As described during the presentation of Wieger Wesselink, center position is also an important tactical to take into account. In fact, at the end of the game corner pieces can become very weak so we want to encourage move that put pieces on middle colons of the board. The ideq is then to compute the deviation of each piece with respect to the middle two columns. Then as for the compactness we decided to devide the deviation of each player by his number of piece left to not imbalance the result. To clarify this statement, here is the Algorithm 5 used :

Algorithm 5 Center Position

```
1  private int centerPosition(DraughtsState state)

3          whiteWeight := 0
           blackWeight := 0
5          nbrWhite := 0
           nbrBlack := 0
7          board = place of all pieces on the board
           for each dark fields of the board do
9              deviation := (column-3)*(column-3)
               if there is a white piece then do
11                 whiteWeight += deviation
                   nbrWhite += 1
13             if there is a black piece then do
                   blackWeight += deviation
15                 nbrBlack += 1

17         return -(5*(whiteWeight/nbrWhite - blackWeight/nbrBlack));
```

As you can see we decided to multiply the result by 5 since we wanted to increase the impact of this feature on the game. We also could have increase directly the weights via the vector W.

5 Results

In this section we decided to represent our result using a relevant table as, once again, 1 picture is worth 1000 words. We will first describe the experience we did and then present the results.

First we put in competition our AI *MyDraughtsPlayer98* against 2 given players : the *optimistic* and *uniformed* one. In both case our AI beat them with more or less difficulties (when playing both white and black pieces). But the biggest part of the experience is coming. The idea is to put in competition our IA against a test version of itself : *MyDraughtPlayerTest*. We let for both of them 2 fundamentals features : the the number of piece and the number of king. Then for each game we added a feature and we verify that the IA with this feature win the game to prove the efficiency of each future independently of each other. Also we do a final game with one IA having all the features and the other one having only the two fundamentals features to prove that the different features are also valid together. These are the results (figure 4) we obtained :

	pieces	king	compactness	tempi	center position	Winer/Loser	# of pieces left
MyDraughtsPlayer98	X	X	X			W	14
MyDraughtsPlayerTest	X	X				L	0
MyDraughtsPlayer98	X	X		X		W	5
MyDraughtsPlayerTest	X	X				L	0
MyDraughtsPlayer98	X	X			X	W	12
MyDraughtsPlayerTest	X	X				L	0
MyDraughtsPlayer98	X	X	X	X	X	W	15
MyDraughtsPlayerTest	X	X				L	0

Figure 4: Results of the experience described in the section Results

After analyzing these results, it is not hard to say that in fact the different optional features we implemented are useful and work correctly as well independently of each other than together. More over we performed 2 additional tests furnished by the Professor that consists to find the best move in 2 different positions. With our IA we are able to pass both tests playing either black or white pieces.

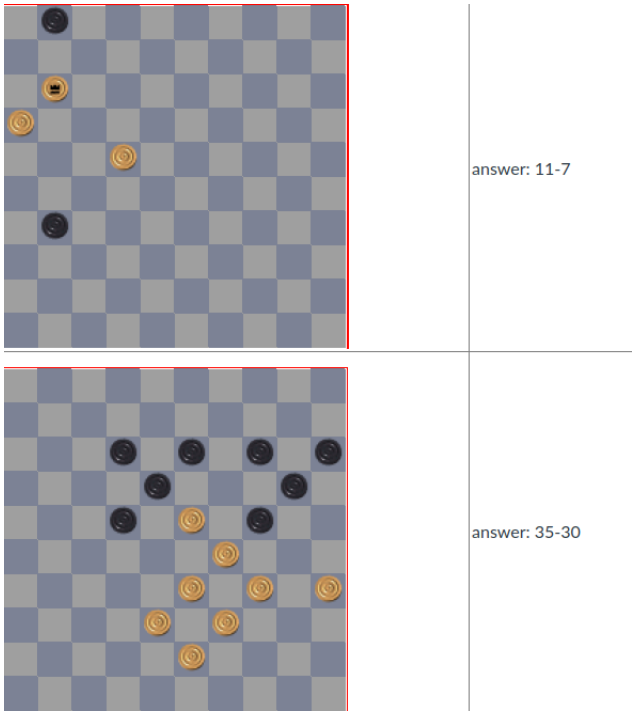

Figure 5: "Find best move" tests

6 Conclusions

In short, and in conclusion, we think that we globally took up the challenge of implementing and International Draughts plays using basic and fundamental artificial intelligence knowledge. We implemented and optimised the *Alpha-Beta* algorithm using *iterative deepening depth-first* search. We constructed an efficient evaluation function and developed many features. Our results are clear and prove the efficiency of our implementation. Still, we would like to go further in our thinking.
As we said, we used basic AI tools to develop our International Draughts Player, but while doing this project we also discovered more advanced methods. The most interesting one is probably the reinforcement learning method which can be used to find the most efficient weights in our evaluation function. Also, we discovered a lot of different features harder to implement but very interesting as the mobility of a player or the number of capture moved possible. We think that this project was a very good introduction to AI and could be use as a base to implement more advanced method in the future. For the moment we think this is out of the scope of the project and 'beyond our pay-grade', but we hope that we will be able to improve this project in the future, for our own interests.

7 Contributions

A statement on the contributions of each of the authors.

	implementation	**documentation**	**total #hours**
Lucas Giordano	60%	40%	40
Auguste Lefevre	40%	60%	40

References

[1] Wolfgang Ertel. *Introduction to Artificial Intelligence*. Springer Publishing Company, Incorporated, 2nd edition, 2017.

[2] kartikkukreja. Iterative deepening alpha-beta algorithm. 2015.

[3] TÃM LE MINH. Draughts ai. 2019.

[4] Wieger Wesselink. Draughts analysis. 2017.