Alpha-Beta vs Scout Algorithms for the Othello Game

Jorge Hernandez, Karen Daza, and Hector Florez ✉

Universidad Distrital Francisco Jose de Caldas
Bogotá, Colombia
jehernandezrodriguez@gmail.com, kgiselledaza@gmail.com,
haflorezf@udistrital.edu.co

Abstract. In the context of decision systems for board games; usually, nodes in a game tree are explored using a search algorithm. When the algorithm visits a node, it must evaluate that node through a function called heuristic. In order to design the evaluation function, the domain must be taken into account. In this paper, we discuss the factors that influence the design of a heuristic for a desire board game. Thus, we present an approach to find the best movement by deploying a game tree, with an implementation for the board game called Othello. The state of the board is used to obtain the desired factors and the best movement is obtained through an in-depth search, according to the designed heuristic. We experimented with two algorithms. The former is Mini-Max and its evolution to Alpha-Beta. The latter is Scout, which presents better performance regarding time. In addition, we present the results, rules, and implementation features.

Keywords: Scout algorithm · artificial intelligence · game tree · Alpha-Beta Pruning

1 Introduction

The states of a two-players game can usually be represented in a game tree. This represents the possible paths that an agent can take when trying to find the best solution. In games specification of complexity degree, the deployment of the nodes for each branch can become too extensive. Thus, it is important to find the most appropriate solution in less time. When the target endgame gets too large to fit into the main memory, fast memory-efficient algorithms are devised [21]. We could see this problem from the hardware point of view, where the finite tree is physically deployed in a database through retrospective analysis. Thus, we could consult the best movement or perform the search deeply as presented for AlphaGo[1]. AlphaGo has asynchronous processes for its calculations searching the tree using the Monte Carlo algorithm. When searching game trees, especially in a competitive setting, significant benefits can be achieved by pruning branches which under no circumstances can affect the decision being made at the root [15].

[1] https://deepmind.com/research/alphago/

2019 ICAI Workshops, pp. 65–79, 2019.

Heuristic search has been applied both to two-player games and single-agent problems [5,10]. The search algorithms use an evaluation function $F(n)$ where n is the node to evaluate. The heuristic influences the exploration because based on it, a value is assigned to each space. In games, this value can be assigned to the board to obtain the best value enabling the algorithm to make the best decision when performing a movement. In order to find a sequence of actions from an initial state to a goal state efficiently, this heuristic function has to guide the search towards the goal. However, it is important to have in mind that a static heuristic can present two problems: 1) each node in the exploration is different, since the movement must be evaluated and 2) the strategy in each game is different. Therefore, it is necessary to find the most relevant factors in the design of the heuristic for a desired domain.

In this work, we compare two algorithms at runtime. The former is the Alpha-Beta algorithm, which prunes the branches that do not meet the value $-\infty$ or ∞ depending on the player (max or min). The latter is the Scout algorithm, which aims to solve the problem of branching the Alpha-Beta algorithm. The nodes explored in the two algorithms had the same evaluation criteria in the heuristic.

Then, we propose a heuristic that depends on certain factors for the two-player game called Othello. We assign static values to each board position to calculate some factors. We assign static values to each board position to calculate some factors. To be able to demonstrate the approach, Othello is good enough case study since it is two-players strategy game played on an 8 x 8 board. There are 64 identical pieces which are white on one side and black on the other [9].

The paper is structured as follows. Section 2 presents the definition of the Othello game. Section 3 presents the main concepts of search game tree. Section 4 presents the related work. In section 5, we illustrate the proposed approach. Section 6 presents the results. Finally, section 7 concludes the paper.

2 Othello Game

Othello is classified as a territorial and abstract game. The generalizations are natural in the sense that the board is extended by planar $n \times n$ locations. In Othello, an arbitrary position of each game is given, and no further rules are modified [6]. Othello is a popular game in the world, which handles two-colored pieces, usually black and white. The goal is to reverse the opponent's pieces in order to finish the game with more pieces than the opponent. Fig. 1 shows an example board with the description of the movements. In this state, the turn is for the black player and the legal movements are indicated in gray. Thus, six different legal movements are identified indicated by numbers 1 to 6. These movements are: two diagonal, two vertical and two horizontal.

3 Search in game tree

Search in two-players games has been an important topic in Artificial Intelligence (AI). The range of search strategies investigated stretch from application-

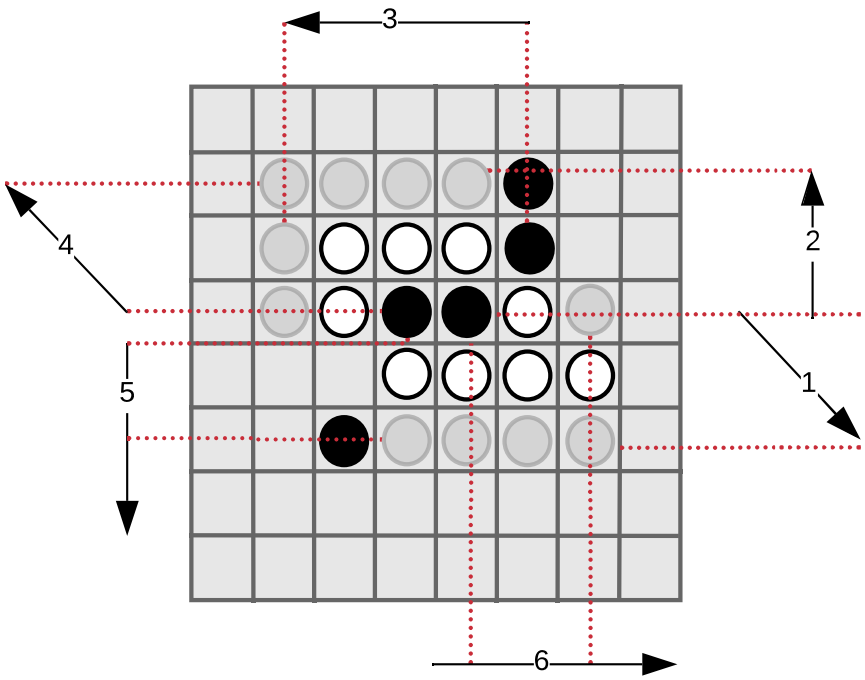

Fig. 1. Example game state

independent methods to knowledge-intensive methods. The former has the promise of general applicability, the latter of high performance [14]. Each movement generates a state (node) in the set of spaces. Multiple algorithms have been created to search the best node. Minimax algorithm was the pioneer and one of the most used in games with AI; nevertheless, this algorithm examines more board states than necessary [4]. Thus, it is more convenient to use Alpha-Beta algorithm. Another algorithm is Scout, which is able to find the best result with a better performance.

3.1 Two-players games

We consider a class of two-players perfect information games in which two players, called max and min, take alternate turns in selecting one out of legal moves [13]. They are games that use strategy models defined by movements. The two-players games we are dealing with can be characterized by a set of *positions*, and by a set of rules for moving from one position to another, where the players make their movements alternately [7]. An important feature is the fact that the board can be represented as a set of values, where each piece represents a value. Many AI systems use evaluation functions for guiding search tasks. In the con-

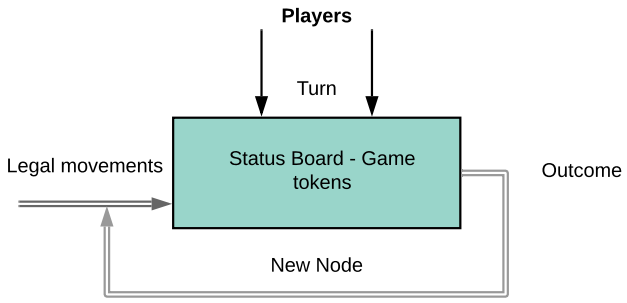

Fig. 2. Components in two-players games

text of strategy games, they usually map game positions into the real numbers for estimating the winning chance for the player to move [2]. Fig. 2 shows the main components of this type of game. The two players alternately have a turn to select a piece of the board, the configuration or state of this board has a value for each player, and each movement generates a new set of legal movements until reaching the end of the game or a terminal state.

3.2 Alpha-Beta algorithm

The pruning Alpha-Beta algorithm is an improved technique of the Minimax algorithm in which it is possible to calculate an objective state without the need to traverse all the nodes of the game tree. This type of algorithm is usually used for any type of deep search tree and whole subtrees. The effectiveness of Alpha-Beta is maximized if the best move is considered first at all interior nodes of the search tree [18]. The problem that exists using the Minimax algorithm is that the number of nodes to explore is exponential to the number of movements. Pruning does not influence the decision of the movement; however, it significantly reduces exploration, improving performance.

3.3 Scout algorithm

Scout was proposed by Judea Pearl in the 80s. The idea is to reduce the computational effort by first checking the inequality $V(S) > v$, where S is the current node, v is some reference value and $V(S)$ is the heuristic function. The power of the algorithm is due to its pruning capacity, where branches that do not lead to a better solution should not be explored. Scout needs a method to check inequalities called Test. Test is a predicate that is part of the Scout process. It returns true if a value is a bound on the Minimax value of a game tree [12] and it will be responsible for resolving inequalities. The Test input parameters are: the node to apply the test, the fixed value of the comparison, and the inequality to be applied (greater). Its output is a boolean value to indicate whether or not the inequality is met. Algorithm 1 shows an experimental approach of this function. The main

```
Input  : board, score, player (condition)
Output: True or false
moves ← getLegalMoves(board);
if moves == 0 or endGame(board) then
    if evaluateNode(board) > Score then return true ;
    else return false ;
end
else
    if player is max then
        foreach move in moves do
            newBoard ← fixedPosition(move);
            if test(newBoard,score,player) then
                return true
            end
        end
    end
    else
        foreach move in moves do
            newBoard ← fixedPosition(move);
            if ! test(newBoard,score,player) then
                return false
            end
        end
    end
end
// no node meets the condition
if player is max then return true;
else return false;
```

Algorithm 1: Test Function in Scout

input is the node. If it is not a terminal state, the sub-nodes are explored to obtain the best score. The score is evaluated with the function `evaluateNode`, which is the heuristic of the approach. `evaluateNode` receives the status of the board and returns the value of the node. Depending on the condition (player), Test validates whether the current branch should be explored.

In addition, Scout algorithm uses the Eval function to calculate the Minimax value of a node, using Test to check whether or not it is necessary to explore a particular branch. Its main input is the node to evaluate. Algorithm 2 generates the legal movements according to the node or board it receives. The first node is evaluated and from there, according to Test, the branches with the best values are explored. Finally, the best move to make according to the heuristic configuration in the function `evaluateNode` is returned.

```
Input  : board or node , depth, player
Output: optimal move and best score
moves ← getLegalMoves(board);
score ← 0;
if moves == 0 or endGame(board) or Depth == 0 then
    score ← evaluateNode(board);
end
else
    newBoard ← fixedPosition(moves− > item(0));
    score ← eval(depth − 1, player, newBoard) → bestScore;
     moves→ removeItem(0) ;
    if player is max then
        foreach move in moves do
            newBoard ← fixedPosition(move);
            if test(newBoard,score,playerMax) then
                score ← eval(depth − 1, playerMax, newBoard) →
                  bestScore;
                bestMove ← move;
            end
        end
    end
    else
        foreach move in moves do
            newBoard ← fixedPosition(move);
            if test(newBoard,score,playerMin) then
                score ← eval(depth − 1, playerMin, newBoard) →
                  bestScore;
                bestMove ← move;
            end
        end
    end
end
return score, bestMove;
```

Algorithm 2: Eval Function in Scout

4 Related work

Defining the heuristic with general factors for games represents the possibility of extending the design and implementation for various games. The following works try to solve the problem of defining a heuristic in a general context.

Buro [3] proposes two phases to build an evaluation function. Selecting features and combining them. In addition, he creates a generalized linear evaluation model (GLEM), where it is described as combining conjunctions of Boolean characteristics linearly. Combined with an efficient minimum square weight adjustment, GLEM greatly facilitates the programmer task of finding significant characteristics and assigning the most suitable weights.

Sephton et al. [20] perform a heuristic to determine the best move for the game Lords of War creating a set of functions when examining a state. They experimented in two ways: using heuristics that extracted statistics from the cards in the game and using heuristics that used heat maps to prioritize specific positions. As they concluded, the first category proved to be the most effective, which was the simplest heuristic since it simply counted the number of cards. Heat map heuristics were generally ineffective; however, they showed the greatest relative improvement in state extrapolation.

Sanchez et al. [16] present a proposal to create an agent in the game Quoridor based on some improvements to the graphics of the board. The strategy is done by displaying the tree of the game. The nodes are stored in a NOSQL graphs database. The work focuses on the representation of the nodes. In this way, they present the tools and techniques to reduce the great impact of processing in the exploration of the nodes.

Kuhlmann et al. [8] describe how to create an agent for multiple prolog-style games. They built a heuristic from the characteristics identified in the description of the game. The heuristics of the candidate node are evaluated in parallel during the selection of actions to find the best movement. The agent identifies five elements of the game: relations of successors, counters, boards, markers, pieces, and quantities.

Schiffel et al. [19] present an approach to a general game combining reasoning about actions with heuristic search. The approach was based on automated analysis to build a heuristic search. They describe the following generalities: 1) Determining legal moves and their effects from a formal game description requires reasoning about actions. 2) Using non-uniform depth first. 3) Using Fuzzy Logic to determine the degree to which a position satisfies the logical description of a winning position. Strategic, goal-oriented play requires to automatically derive game-specific knowledge from the game rules. Then, terminal states are avoided as long as the objective is not met i.e., the terminal value has a negative impact on the status assessment if the objective has a low value and positive impact.

The works of Noe [12] and Pearl [13] present a comparison between Alpha-Beta and Scout for the kalah game. Both conclude that Scout has better performance.

5 Proposed Approach

Learning without any prior knowledge in environments that contain large or continuous state spaces is a tough task [1]. When implementing a game tree, we seek to explore all possible states; however, these implementations have become a bit obsolete because search algorithms usually take too much memory at runtime. In this section, we present the approach we proposed to design the heuristic and the search process to obtain the best movement for the Othello game.

5.1 Heuristic for two-players games

The proposed heuristic is intended to state global factors and discuss their use in different contexts. Each factor is set using a weight W. If the factor should not be implemented due to the domain, its weight will be 0 removing the heuristic factor. Developing heuristics for games or other search problems typically involves a great deal of testing by playing the game repeatedly under realistic time constraints against several different opponents [11]. The implemented heuristic has a set of factors that are taken into account when calculating the value of a node V. Factors details are as follows:

- **Parity of pieces.** This factor calculates the piece difference between the player and the machine. The important thing is to correctly calculate the difference of pieces between one player and the other. Some games need, as a strategy to keep fewer pieces during the start of the game. It is more efficient than having a large amount that could be in favor of the other player in a move.
- **Legal movements.** Mobility is one of the fundamental factors. Based on its calculation, it is possible to determine the outcome of the game. For the calculation of mobility, two types of movements are taken into account: the actual movement and the potential movement. On the one hand, the actual movement is the number of movements that the player can perform in his turn. On the other hand, the potential movement is the one that is developed when the game progresses depending on the movements made by the player and his opponent. Note that movements that are currently not legal, but might become legal in the near future are accounted for in the calculation of potential mobility. Hence, potential mobility captures the mobility of the player in the long term, while actual mobility captures the immediate mobility of the player[17].
- **Positions in the game.** Capturing some positions is an important factor. We see it in Othello, where it is of great importance to capture the corners since it is very likely that the player who puts a piece in these positions $[(0,0),(0,7),(7,0),(7,7)]$ can control the game. The corners are the most stable points of the board which means that the piece that occupies that position cannot be flanked. By occupying the corner positions, the player can build a stable game, which increases the chance of winning the game.
  The following source code represents the value we gave to each position in the board for the Othello game. The heuristic value for this factor for the player is calculated by adding the weights of the squares in which his pieces are present. The static board implicitly captures the importance of each frame on the board and encourages the game to tend to capture some positions. Dynamically changing these weights would mean that we would have to use heuristics to calculate the weight of a position based on its stability.

```
integer positions [][] =
{{99, -18, 8, 6,6, 8, -18, 99},
{-18, -24, -14, -12, -12, -14, -24, -18},
```

```
{8, -14, 15, 15, 15, 15, -14, 8},
{6, -12, 15, 10, 10, 15, -12, 6},
{6, -12, 15, 10, 10, 15, -12, 6},
{8, -14, 15, 15, 15, 15, -14, 8},
{-18, -24, -14, -12, -12, -14, -24, -18},
{99, -18, 8, 6, 6, 8, -18, 99}};
```

It is based on the possibility that a piece might be flanked. The stability has three states:

- Stable. A piece is stable when it is not possible to be flanked during the development of the game until its end. In Othello, the corners are the most stable positions of the game. In addition, as the game is established in the corners, the pieces become more stable.
- Semi-stable. A piece is semi-stable when it can be flanked at some point in the game, but not precisely on the next movement.
- Unstable. A piece is unstable when it is potentially flanked on the next movement. For Othello stability is an important factor.

– **Node is a terminal.** If the node is a terminal state, it is validated if the player in function of the heuristic is the winner of the game to assign positive points. Otherwise, this factor will be negative. This factor, depending on the domain is one of the most important factors since it can indicate the most promising path to the search algorithm.
– **Most representative movements.** This factor consists in reviewing the legal movements of the board for the two players as a method of strategy for the next movement. The difference between the amount of legal movements for each player is calculated. In addition, the position of the movement is calculated by looking for its position. An additional list of positions is created, which are the most representative in the next movement. For Othello, the list consisted of the corners, along with the edges. If the legal movements have these positions, positive points are given for the player or if the opponent is the one who has a corner position, in that case, the factor will have negative points.
– **Intermediate tabs.** The amount of pieces between the enemy and the heuristic player is calculated against each enemy piece.
– **Parity.** The parity value for a factor is calculated by adding the weights of the other factors in which the player's pieces are present. In this way, the quadratic average of the other weights is calculated. This calculation is made by the equation 1, where w is the weight of each factor and N the amount of weights.

$$parity = \sqrt{\frac{1}{N}\sum_{1=0}^{n}(w_i^2)} \tag{1}$$

Finally, the node score is calculated adding all factors already mentioned using the equation 2, where K represents the amount of factors, $weight_{fi}$ is the weight of factor, and f_i is the value of factor.

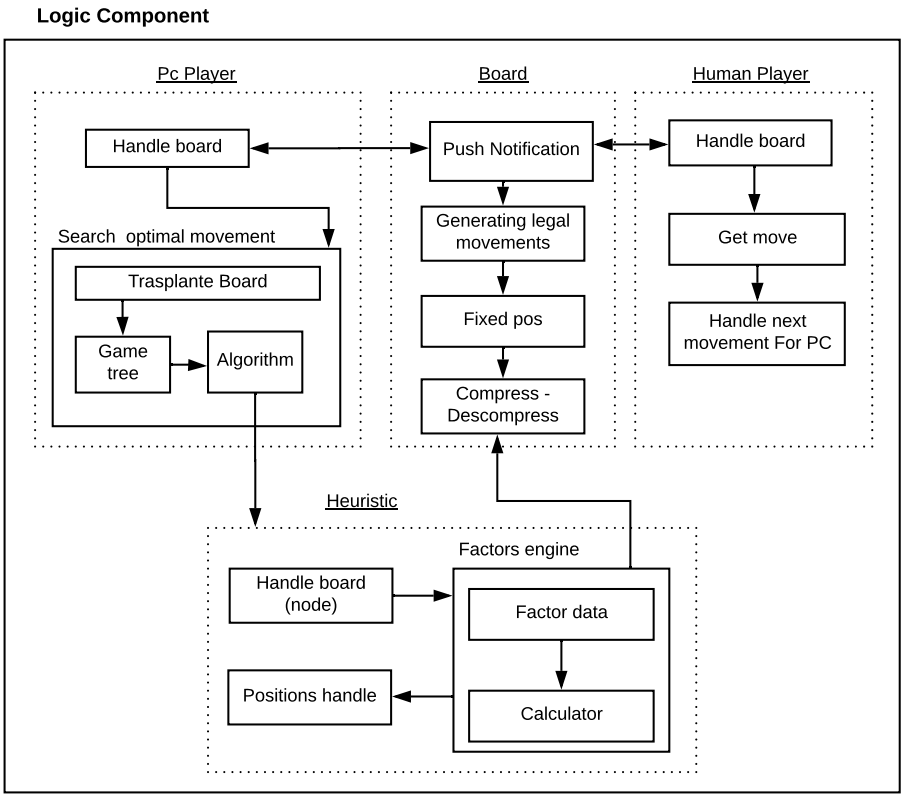

Fig. 3. Components Approach

$$score = \sum_{f=0}^{k} (weight_{fi} \times f_i) \tag{2}$$

5.2 Driving the game board

In the proposed approach, for a one-board node S $\in \{n_1, ...n_k\}$, where k is the number of nodes in the set of states, we define that the best position P in the legal movements of S should be evaluated by the proposed heuristic. This means that the best movement is generated within the legal movements for a reference board. We use the Scout algorithm to search S within P. Fig 3 presents the main components of the approach. The logic component is responsible for managing the interaction of the players with the board and the search. The details of the proposed architecture is as follows:

– **Logic component.** It is responsible for managing the board according to the type of player. The management of the board is done with a $n \times n$

matrix, where $n = 8$ for Othello. When saving the board, it is compressed into a character set with the following notation: *player-0-0*, *player-0-1*, ..., *player-i-j* where i represents the columns, j the rows, and player the value of the cell that can go from 0 to 1, where 0 is for white pieces and 1 for black ones. *Board* allows decompressing the format to an array and is listening for a human player movement using *push notifications*. Players *Pc player* and *Human player* have assigned white or black pieces. The management of the board (fixed pos, generate legal movements and compress-decompress) is done by *Handle Board*. *Pc player* is the implementation of the search algorithm. In the case that *Pc Player* is playing with white, the best move will be searched with black, the board will always be managed with black pieces, so the *Transplate Board* component is responsible for transposing the values. To obtain the movement, the tree of the game is deployed with the Scout algorithm; after that, the best movement is fixed. In *Human player* we analyze the legal movements the player can make with the *Get move* component. *Handle next movement for Pc* chooses a movement randomly within the legal movements of *Human player* as a candidate. Thus, we can anticipate and make the management of the *Pc player* (look for movement in the tree) to save time while the *Human player* chooses his movement. If we do not succeed in the movement of the *Human player*, *Pc player* calculates the best movement.

– **Heuristic and search optimal movement.** These two components interact to get the best movement. In *search optimal movement*, the algorithm is displayed, the legal movements for the search are obtained using *Handle Board*. The algorithm sets each legal movement and from there generates the branches of the tree. When evaluating a node, the *Heuristic* element is used. Heuristic gets all the data of each factor with *Factor data* according to those values such as weight, tab value, or board value. The calculation is made and the heuristic value is generated.

6 Experimental results

The great popularity of mobile devices makes an important influence on people. Thus, creating mobile apps might produce advantages because based on their use, people can develop skills that can boost active learning. With this in mind, we developed a mobile app in order to apply the proposed approach to the Othello game. This app can be used without an internet connection. When the mobile phone is connected to internet, the app has a connection to a non-relational database Firebase [2], which will be used to analyze the data of each game that is done with the algorithm. Based on this, it is possible to analyze the movements made in every game for both the machine and the player. The app has the game modes *vs computer*, which allows the user to play against the machine. The Scout algorithm has been implemented, as well as the Alpha-Beta algorithm to compare the two algorithms and obtain their performance.

[2] `https://firebase.google.com`

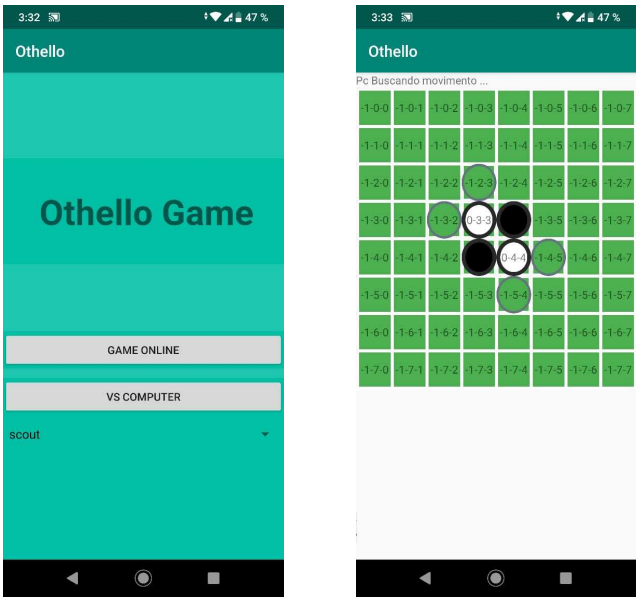

Fig. 4. Main Game Activity.

Fig. 4 shows two parts of the app. In the left side, the user can select the mode and the algorithm. In the right side, the Othello game is deployed. The board has the white pieces, the black pieces, and some gray guides that are the valid movements that the player could make in his turn.

6.1 Comparison Alpha-Beta and Scout

To start comparing the Scout and Alpha-Beta algorithms, we run exploration experiments with the same test field for both algorithms. In this way, we performed an in-depth comparison for both algorithms to see the number of nodes explored. The tests were carried out in mobile programming. In our context, Alpha-Beta allowed us to reach a maximum depth of 8 without ceasing to respond the mobile device due to a lack of memory. The tests were based on measuring the number of nodes exploring the time and depth given. Besides the memory capacity, the Alpha-Beta algorithm requires, its implementation takes a long time to make a decision to place the movement because it explores too much nodes based on the deployment of the decision tree branches. On the contrary, the Scout algorithm just takes a couple of seconds to obtain the right movement exploring much less nodes and making a better decision. Fig. 5 and Fig. 6 represent the experimental results of the tests. The Scout algorithm is detailed in red and Alpha-Beta blue (α - β).

Fig. 5 presents the number of nodes explored by the Scout algorithm and the time it took to explore them. The shortest time was 73 milliseconds and the

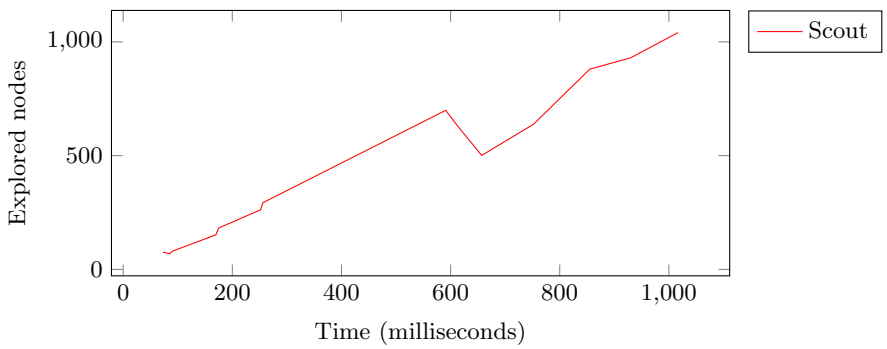

Fig. 5. Exploration for Depth 8 using Scout

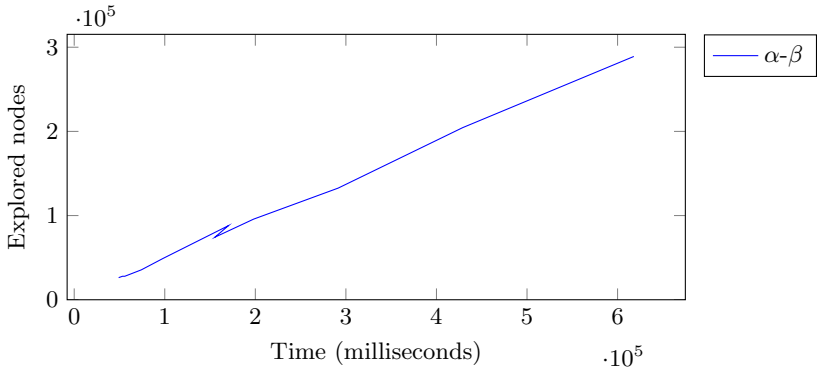

Fig. 6. Exploration for Depth 8 using Alpha-Beta

longest time 1014 milliseconds for 15 movements. The number of nodes ranges from 76 nodes explored to 1041. The performance is compared with Fig. 6, which presents the number of nodes scanned by the Alpha-Beta algorithm that is much greater than the Scout. Thus, Alpha-Beta starts deploying 26264 nodes, while Scout just 76, which represents a huge difference in their performance. for both algorithms, the test was performed with a depth of 8.

6.2 Test results

We carried out tests with 3 players. We classified players into three categories: expert, medium, and amateur. On the one hand, each player played 10 games against the Scout algorithm, where the expert player won 3, drew 5 and lost 2 games. In the same way, the medium player did not win any game, drew 2 and lost 8. Finally, the amateur player did not win or draw any game. As a result, the algorithm won 20 games, drew 7 games, and lost 3 games. On the other hand, the Alpha-Beta algorithm just won 6 games, drew 10 games and lost 14 games.

Consequently, we can asses that the Scout algorithm can beat players even with an expert level.

7 Conclusions and Future work

Based on the results obtained in the implementation of the two algorithms, Scout and Alpha-Beta, the Scout algorithm may have a better performance than Alpha-Beta for board games. Scout algorithm obtained better results in less time exploring fewer nodes, which improved the response time against the human player. The factors were fundamental to obtain these results. It is pertinent to note that the implementation of each algorithm was carried out under the same parameters. With this in mind, we can conclude that under the same conditions the Scout algorithm is more efficient than Alpha-Beta. The performance of each algorithm was tested with 30 games played against human players.

As future work, it is intended to implement a neural network, which will be responsible for assigning the weights to the factors of the heuristic. In this way, the heuristic will be trained through unsupervised learning. The items stored in the database will serve as analysis to reinforce the weights. In the factors, the weights are the basis of the approach and must be readjusted for each game. Then, the algorithm would obtain the necessary information to make the best movements against human players. Thus, it will be easier to build the heuristic regardless of the domain. The main input will be the state of the game, from there, we can obtain the factors that we propose in the paper.

References

1. Bentivegna, D.C., Atkeson, C.G.: A framework for learning from observation using primitives. In: Robot Soccer World Cup. pp. 263–270. Springer (2002)
2. Buro, M.: From simple features to sophisticated evaluation functions. In: International Conference on Computers and Games. pp. 126–145. Springer (1998)
3. Buro, M.: Improving heuristic mini-max search by supervised learning. Artificial Intelligence **134**(1-2), 85–99 (2002)
4. Duarte, V.A.R., Julia, R.M.S.: Mp-draughts: Ordering the search tree and refining the game board representation to improve a multi-agent system for draughts. In: 2012 IEEE 24th International Conference on Tools with Artificial Intelligence. vol. 1, pp. 1120–1125. IEEE (2012)
5. Geissmann, C.: Learning heuristic functions in classical planning (2015)
6. Iwata, S., Kasai, T.: The othello game on an n× n board is pspace-complete. Theoretical Computer Science **123**(2), 329–340 (1994)
7. Knuth, D.E., Moore, R.W.: An analysis of alpha-beta pruning. Artificial intelligence **6**(4), 293–326 (1975)
8. Kuhlmann, G., Stone, P.: Automatic heuristic construction in a complete general game player. In: AAAI. vol. 6, pp. 1457–1462 (2006)
9. Liskowski, P., Jaśkowski, W., Krawiec, K.: Learning to play othello with deep neural networks. IEEE Transactions on Games **10**(4), 354–364 (2018)

10. Mejia-Moncayo, C., Rojas, A.E., Dorado, R.: Manufacturing cell formation with a novel discrete bacterial chemotaxis optimization algorithm. In: Figueroa-García, J.C., López-Santana, E.R., Villa-Ramírez, J.L., Ferro-Escobar, R. (eds.) Applied Computer Sciences in Engineering. pp. 579–588. Springer International Publishing, Cham (2017)
11. Nešić, N., Schiffel, S.: Heuristic function evaluation framework. In: International Conference on Computers and Games. pp. 71–80. Springer (2016)
12. Noe, T.D.: A comparison of the Alpha-Beta and SCOUT algorithms using the game of Kalah. University of California, School of Engineering and Applied Science ... (1980)
13. Pearl, J.: Scout: A simple game-searching algorithm with proven optimal properties. In: AAAI. pp. 143–145 (1980)
14. Plaat, A., Schaeffer, J., Pijls, W., De Bruin, A.: Exploiting graph properties of game trees. In: AAAI/IAAI, Vol. 1. pp. 234–239 (1996)
15. Saffidine, A., Finnsson, H., Buro, M.: Alpha-beta pruning for games with simultaneous moves. In: Twenty-Sixth AAAI Conference on Artificial Intelligence (2012)
16. Sanchez, D., Florez, H.: Improving game modeling for the quoridor game state using graph databases. In: International Conference on Information Theoretic Security. pp. 333–342. Springer (2018)
17. Sannidhanam, V., Annamalai, M.: An analysis of heuristics in othello
18. Schaeffer, J., Plaat, A.: New advances in alpha-beta searching. In: ACM conference on Computer science. pp. 124–130. Citeseer (1996)
19. Schiffel, S., Thielscher, M.: Automatic construction of a heuristic search function for general game playing. Department of Computer Science pp. 16–17 (2006)
20. Sephton, N., Cowling, P.I., Powley, E., Slaven, N.H.: Heuristic move pruning in monte carlo tree search for the strategic card game lords of war. In: 2014 IEEE Conference on Computational Intelligence and Games. pp. 1–7. IEEE (2014)
21. Wu, P.h., Liu, P.Y., Hsu, T.s.: An external-memory retrograde analysis algorithm. In: International Conference on Computers and Games. pp. 145–160. Springer (2004)