A Prolog implementation of the Italian Checkers

Fundamentals of Artificial Intelligence

Nicola Fiorentino - A.Y. 2021-2022

"Games are not chosen because they are clear and simple, but because they provide maximum complexity with minimum initial structures."

Marvin Minsky

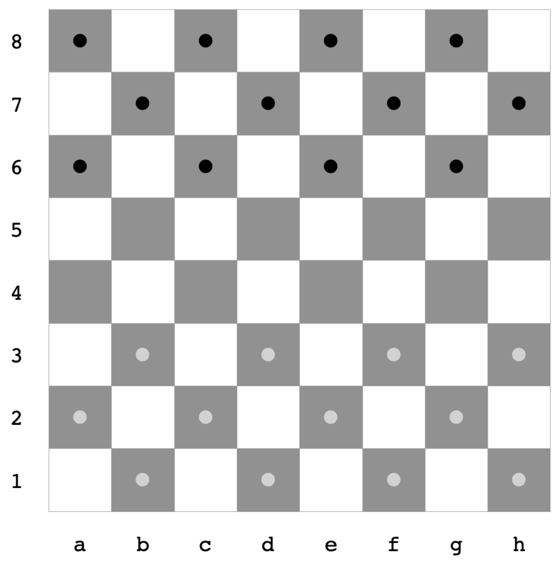
I. Objectives

Games represent a profitable domain to explore machine intelligence. On the one hand, their simplified structure allows a clear representation of the states; the agents are restricted to a small number of actions whose effects are defined in advance; the success or failure can be easily measured.
On the other hand, complex games cannot be solved algorithmically and require a tentative approach. Solutions are to be searched in the space of possible game states. Moreover, the effectiveness of the search is reduced by the presence of opponents who aim to hinder the agent. This requires a contingent strategy that makes the problem harder than a classic search.
Games often have an incredibly large search space. An effective use of knowledge is essential to reduce the combinatorial explosion of the states. Due to complexity and real-time constraints, we may not be able to select the best solution, having to settle for a good one.

In this work we examine game playing from the perspective of a particular game: Italian Checkers. The goal is to design and implement an artificial agent that is able to compete with human players. Logic programming will be used for the implementation.

II. Analysis

Checkers is an example of two-player, turn-taking game. It is deterministic since game states are determined solely by player actions; it is characterized by perfect information as players have access to the complete state of the game at each point in time; it is zero-sum since what is good for one player is just as bad for the other.
Opponents play on opposite sides of a checkered board and only the dark squares are used. One player has black pieces and the other has white pieces. A player cannot move opponent's pieces. Many variants of Checkers are played in the world. In this work we consider the Italian variant with the following rules:



- The board is made up of 64 squares. The last square on the bottom right is black.
- Each player starts with 12 men, placed on the first three rows of his side of the board.
- White always starts playing.

- Man moves one step diagonally forwards into an empty square. When a man reaches the last row, it is promoted and becomes a king.
- Man captures an adjacent opponent's man by jumping over it and landing on the next (empty) square. Men can jump only forwards. In a single turn multiple enemy pieces can be captured by successive jumps made by the same piece. Captured pieces are removed from the board.
- King moves one step diagonally, in all possible directions, capturing both the opponent's men and kings.
- When possible, it is mandatory to capture an opponent's piece.
- Having more chances, it is mandatory to capture as many pieces as possible.
- A player wins when he captures or blocks all of his opponent's pieces.

The game proceeds until one of the two opponents wins. The possibility of a draw is not considered.

III. Design

Theoretical knowledge in this chapter is mainly taken from *Artificial intelligence: A modern approach* [3].

A. Game states

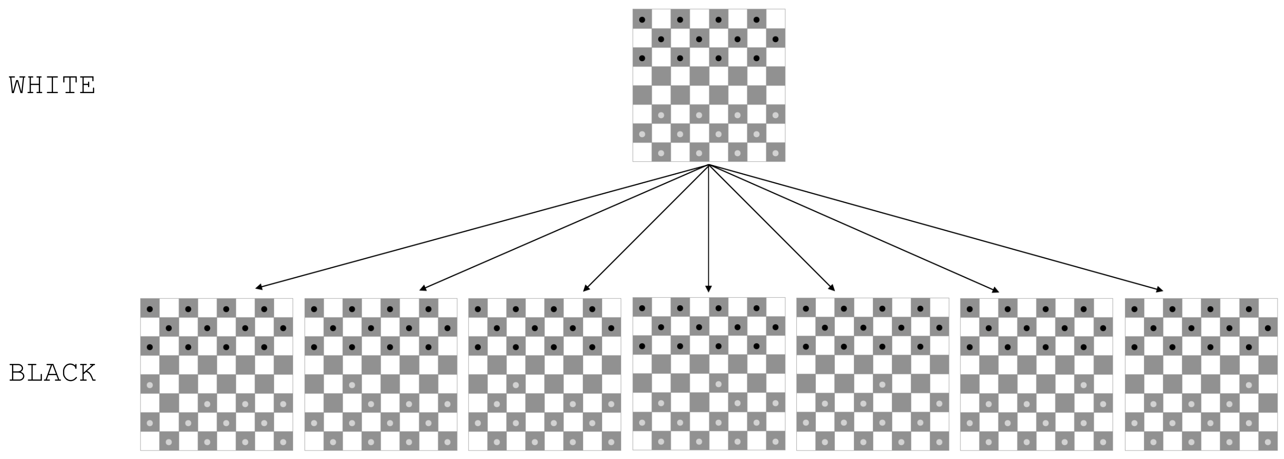
In order to provide a concise and effective conceptualization, the game is described by the pieces in play instead of considering the configuration of the whole board. In this way, a game state is defined by a set of $N \in [1,24]$ pieces and an indication of which player will make the next move. Each piece is defined by the following features:

- $CoordinateX \in \{a, b, c, d, e, f, g, h\}$.
- $CoordinateY \in \{1,2,3,4,5,6,7,8\}$.
- $Player \in \{white, black\}$.
- $Figure \in \{man, king\}$.

Pieces can be added and removed to transit from a game state to the next. At any moment, a successor function will define legal moves and resulting states. A termination test will determine if the game is over. In this case, the player who made the last move wins the game.

B. Game tree

The initial state and the successor function define a game tree in which the vertices are game states and the edges are moves. This representation allows to use a search algorithm to determine what is the optimal move to make. The following figure shows the first level of the tree rooted in the initial state. Note that white moves first.



In the start, white has seven possible moves. Play alternates between white and black until reaching leaf nodes that correspond to terminal states. Each leaf node will be associated with a value expressing its utility from white's point of view. High values are good for white (which is therefore called Max) and bad for black (which is therefore called Min).

C. Minimax search

White would like to find a sequence of moves leading to a win, but black will try to hinder him. White's optimal strategy is obtained by computing for each node a value expressing the utility of being in that state. The value of a leaf node reflects the utility of a win or a loss. In a nonterminal state, white prefers to move to a node of maximum value and black prefers to move to a node of minimum value (that is, minimum value for white and thus maximum value for black).

A search algorithm can find the optimal strategy for white by choosing the move that leads to the state with the highest value. The algorithm proceeds up to the leaves and then backs up the values through the tree. White's optimal strategy assumes that also black plays optimally.

The minimax algorithm performs a complete depth-first exploration of the tree. If the maximum depth of the tree is m and there are b legal moves at each point, the time complexity is $O(b^m)$. This makes the pure minimax algorithm unsuitable for complex games like Checkers.

D. Alpha-beta pruning

A first optimization, known as alpha-beta pruning, allows to find the optimal move without examining every node of the tree. It involves pruning parts of the tree that do not affect the final decision. The general principle is that a player will never select a node *n*, if a better choice is available either at the same level or at any point higher up in the tree. So once the algorithm has found out enough about *n* (by examining some of its descendants) to reach this conclusion, *n* can be pruned.

The algorithm exploits two parameters that define the bounds to decide whether to abandon a path: α is the minimal value that white is already guaranteed to achieve; β is the maximal value that white can hope to achieve, that is the worst value for black that black is already guaranteed to achieve. Thus, the actual value lies between α and β, and this interval can get narrower but never wider. The algorithm updates the values of α and β during search and prunes the remaining branches at a node as soon as the value of the node is known to be worse than the current α or β for white or black, respectively. The effectiveness of the pruning depends on the order in which the nodes are examined. For a random ordering the complexity is reduced to $O(b^{3m/4})$.

E. Heuristic search

Even with pruning, the game tree remains too big to be explored exhaustively. A solution (proposed by Claude Shannon in 1950) is to consider all possible moves up to a certain depth and then to cut off the search. A heuristic evaluation function will be applied to leaf nodes in order to estimate their expected utility. For terminal states the evaluation function returns their actual utility, whereas for nonterminal states its value is between a loss and a win.

Since the goal is to optimize the search, the evaluation function must be efficiently computed. In addition, it should reflect the actual chances of winning. To do this, it can consider various features of the state which are then combined into a weighted sum. Formally, the evaluation function can be expressed as follows:

$$eval(s) = \begin{cases} \infty & \text{If white wins in } s \\ -\infty & \text{If black wins in} s \\ w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) & \text{If } s \text{ is a nonterminal state} \end{cases}$$

Each f_i is a feature of the state and w_i is a weight expressing the importance of the feature. Features and weights are defined using knowledge about the game. A perfect evaluation function would require expanding only the immediate successors of the current state. But for complex games like Checkers, any evaluation function will provide just an estimate.

The simplest approach to control the amount of search is to set a fixed depth limit. This depth should be chosen so that a move is selected within the available time.

F. Iterative deepening

By setting a fixed depth limit, the time required to search for the optimal move varies according to the current state of the board. As the number of available moves grows, the search time increases. This forces the use of a shallow depth limit to prevent search from taking too long in complex game states.

A solution to this problem is given by an iterative deepening approach. Iterative deepening solves the problem of setting a fixed depth limit, by trying increasing depths until the available time runs out. The algorithm repeatedly performs an alpha-beta search, by increasing the depth on each iteration. When the time limit is reached, the best move according to the deepest search is played. This approach makes it possible to reach greater depths when the number of available moves is limited.

Iterative deepening entails some overhead since states near the top of the tree are generated multiple times. But often most of the nodes are in the bottom levels, so the advantages outweigh the drawbacks.

G. Graphical interface

To make the application user-friendly, it will be provided with a graphical interface. The user will be able to select the color to play with. When the game starts, the moves will be played by simply specifying the coordinate of the square containing the piece to be moved followed by the coordinate of the landing square.

IV. Implementation

The language used for the implementation is Prolog. It is a logic programming language that allows to describe a problem declaratively, as a set of clauses, instead of prescribing the sequence of steps to solve the problem. The adopted environment is SWI-Prolog.
In the implementation the "cut" was widely used to tell Prolog which previous choices it need not consider again when it backtracks. It is also used when not logically necessary for optimization purposes: the program operates faster if it does not attempt to satisfy goals that will never contribute to a solution, and it occupies less memory if backtracking points do not have to be recorded for later examination.
The whole application consists of five different modules, each with a specific responsibility: main, move, search, heuristics and draw.

A. Main

The main module manages the overall game process. It uses all the other modules and defines two dynamic facts: `p(X,Y,Player,Fig)` representing a piece of the board and `cpu(Player)` representing the color assigned to the CPU. The module initializes the board by declaring a fact for each piece in play. It allows the user to choose a color and manages game turns. During a CPU turn, the search for the optimal move begins with an initial depth equal to 4 and if the search time has not exceeded 3 seconds, it goes on with a greater depth. The module defines the following predicates:

`play`	Start a new game.
`initialize_board`	Initialize the game with 12 white men and 12 black men.
`select_color`	Ask the user to select the color to play with.
`opponent(Player,Opp).`	Match a player to his opponent.
`turn(Player)`	If the game is not over, start a turn for the player. Otherwise announce the winner.
`read_move(LegalMoves,Move)`	Read a legal move from the keyboard.

B. Move

The move module manages the movement of the pieces on the board. It identifies the legal moves, respecting their priority, and deals with the capture and promotion of the pieces. It defines the following predicates:

`legal_moves(Player,Moves)`	Find mandatory jump moves for the player. If no jump move is available, find non-jump moves. Fail if no legal move is available for the player.
`empty_square(X,Y)`	Check if a dark square of the board is empty.
`next_row(Player,Fig,Y1,Y2)`	Determine the next legal row for a given piece.
`next_col(X1,X2)`	Determine the next legal column.
`jump(Player,Fig,X1,Y1,X2,Y2, PrevJumps,Jumps)`	For a given piece, recursively jump over enemy pieces.
`longest_jumps(Moves,Longest)`	Select moves with longest jump sequence.
`move_piece(X1,Y1,X2,Y2,Jumps)`	Move a piece (possibly promoting it) and remove captured pieces.
`simulate_move(Move,Removed)`	Simulate a move and store the removed pieces.
`undo_move(Move,Removed)`	Undo a move and restore the removed pieces.
`save_state(State)`	Save the current state of the board.
`restore_state(State)`	Restore a given state of the board.

C. Search

The search module manages the search for the optimal move. The implementation of the alpha-beta search algorithm is adapted from *Prolog Programming for Artificial Intelligence* [4]. The module defines the following predicates:

`search_move(Player,MinDepth,MaxTime,BestMove)`
Search for the best move using an iterative deepening alpha-beta search, starting from a given depth and going on if the available time has not run out.

`iterative_deepening(Player,Depth,Time,CurrentBest,BestMove)`
Repeatedly perform an alpha-beta search with increasing depths, until the time limit is reached. Then return the best move according to the deepest search.
`alpha_beta(Player,Depth,Alpha,Beta,BestMove,BestVal)`
Perform a heuristic alpha-beta search.
`best(Player,Depth,Moves,Alpha,Beta,BestMove,BestVal)`
Select the best move from a list of candidates. Best is either maximum or minimum, depending on the player.
`good_enough(Player,Depth,Moves,Alpha,Beta,Move,Val,BestMove,BestVal)`
Check if a move is sufficiently good to make the correct decision. If not, try the next candidate move.
`new_bounds(Player,Alpha,Beta,Val,NewAlpha,NewBeta)`
Update the values of alpha and beta.
`is_better(Player,Move,Val,NewMove,NewVal,BestMove,BestVal)`
Determine which of two moves is better.

D. Heuristics

The heuristics module computes the heuristic evaluation function. The rationale for the choice of features and weights will be provided in the next chapter. The considered features are the following:

- *Men* is the difference of men between white and black.
- *Kings* is the difference of kings between white and black.
- *Center* is the difference of pieces in the center between white and black. The center of the board is defined by $X \in \{c, d, e, f\}$ and $Y \in \{3,4,5,6\}$.
- *Arrows* is the difference of arrows between white and black. An arrow is a formation of three pieces connected in the opponent's direction.
- *Progress* is the difference of progress of pieces between white and black. The progress of a piece is defined as the distance from the back row.
- *Back* is the difference of men in the back row between white and black.
- *Threats* is the difference of threats between white and black. A threat is a potential jump over an opponent's piece.

- *Mobility* is the difference of available non-jump moves between white and black.

The final value is computed as:

$$Value = 2Men + 3Kings + 0.2Center + 0.5Arrows + 0.2Progress + Back + 1.3Threats + 0.2Mobility$$

The module defines the following predicates:

`evaluate(Value)`	Evaluate the game state using a heuristic function.
`count_pieces(Player,Fig,N)`	For a given player, count pieces of a given type.
`count_center(Player,N)`	For a given player, count pieces in the center of the board.
`count_back(Player,N)`	For a given player, count men in the back row.
`count_arrows(Player,N)`	For a given player, count formations of three pieces connected in the opponent's direction.
`count_progress(Player,N)`	For a given player, count progress of pieces towards the opponent's side.
`count_threats(Player,N)`	For a given player, count potential jumps over the opponent's pieces.
`count_mobility(Player,N)`	For a given player, count available non-jump moves.

E. Draw

The draw module manages the graphical interface. This consists of Unicode characters. The module defines the following predicates:

`print_logo`	Print logo and instructions.
`print_board(Player)`	Print the current state of the board and highlight the legal moves for the player.
`print_row(Y,LegalMoves)`	Print a given row of the board.
`symbol(X,Y,LegalMoves,Char)`	Assign a symbol to each square of the board.

V. Test

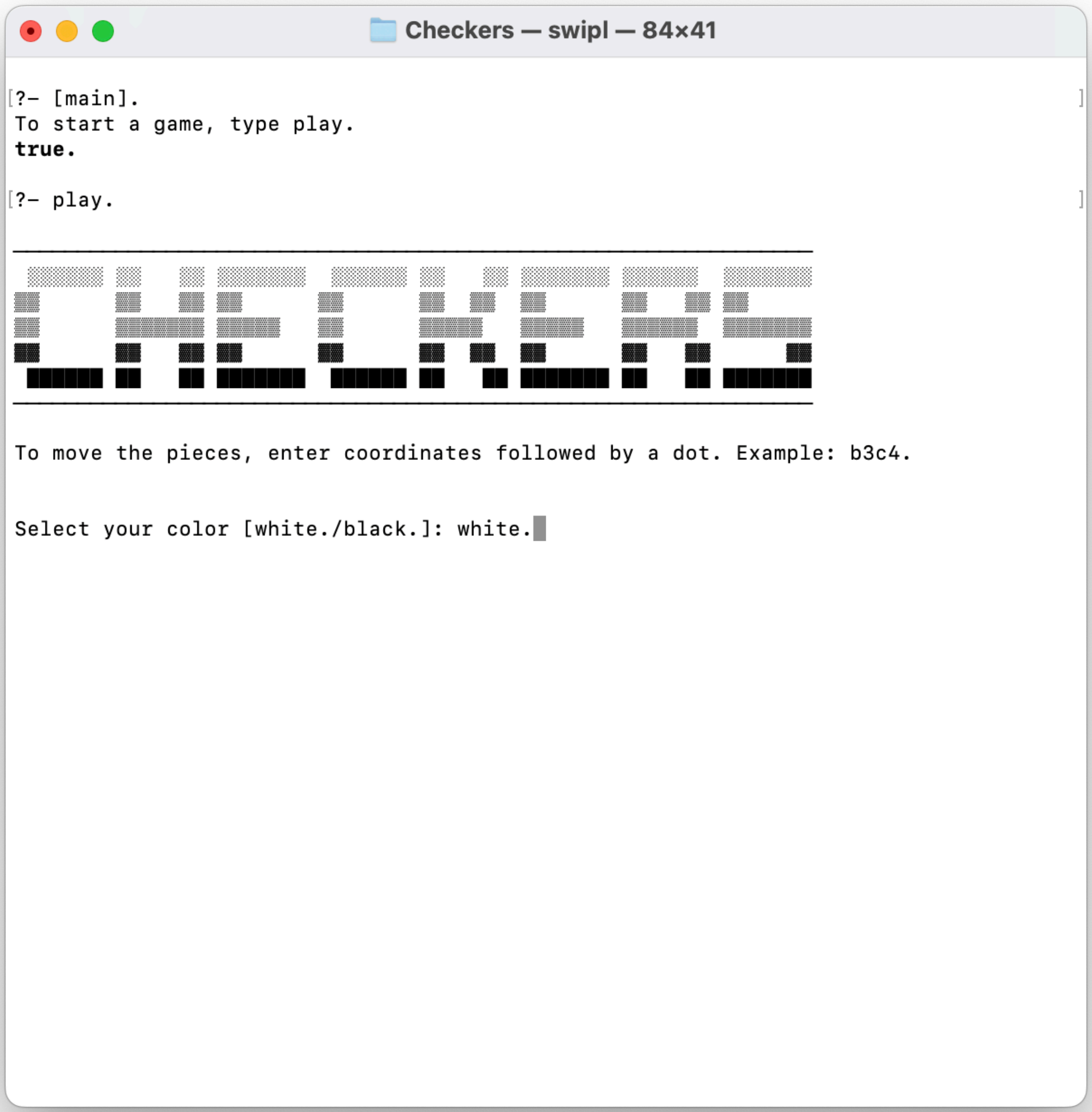
A. Heuristics tuning

To test the application, a simple heuristic function based on the amount of pieces of each player was initially used. Since the performance of this simple function was poor, a more accurate analysis of game strategies was performed, using personal knowledge of the game as well as general information found on websites, online forums and scientific papers [6]. When playing Checkers, you don't just aim to capture your opponent's pieces. It is essential to protect your own pieces and occupy the opponent's territory in order to increase the chances of promotion. These factors led to the eight features presented above.

To distinguish the importance of the features, a weight was assigned to each of them. This weight was determined empirically, based on game knowledge and system behavior. Specifically, the behavior was evaluated both by allowing the system to challenge itself using distinct heuristics, and by challenging (average level) human players. While not perfect, the final function is often able to defeat non-expert human users.

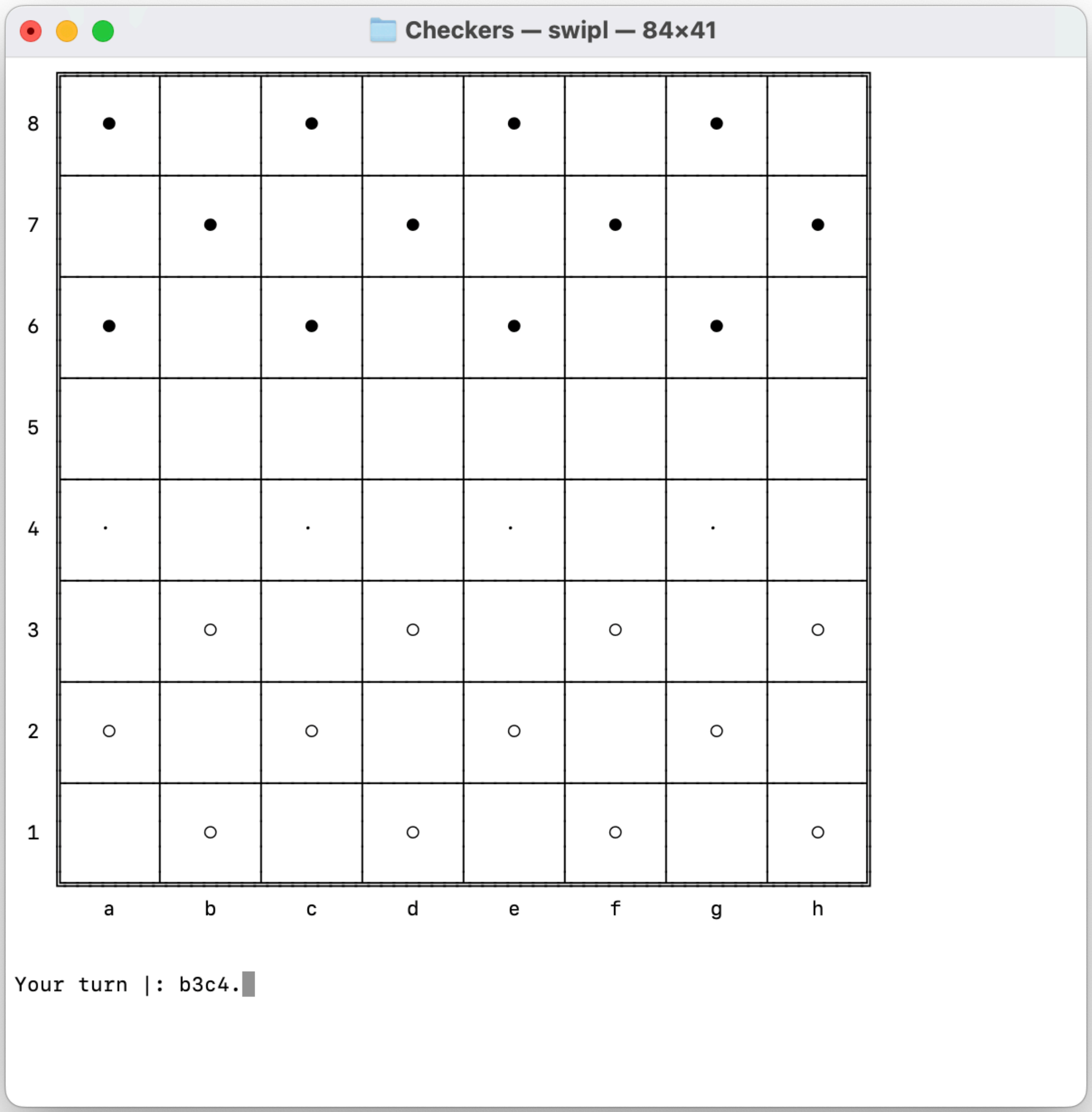
B. Use

The application has been tested on MacOS. SWI-Prolog must be installed on the system. For an optimal use, it is recommended to launch the application in full screen, using a light theme. To run the application, the user can consult the main module. A welcome message will suggest typing "`play.`" in order to start a new game. Then, the system will show logo and instructions and the user will be asked to select a color to play with.



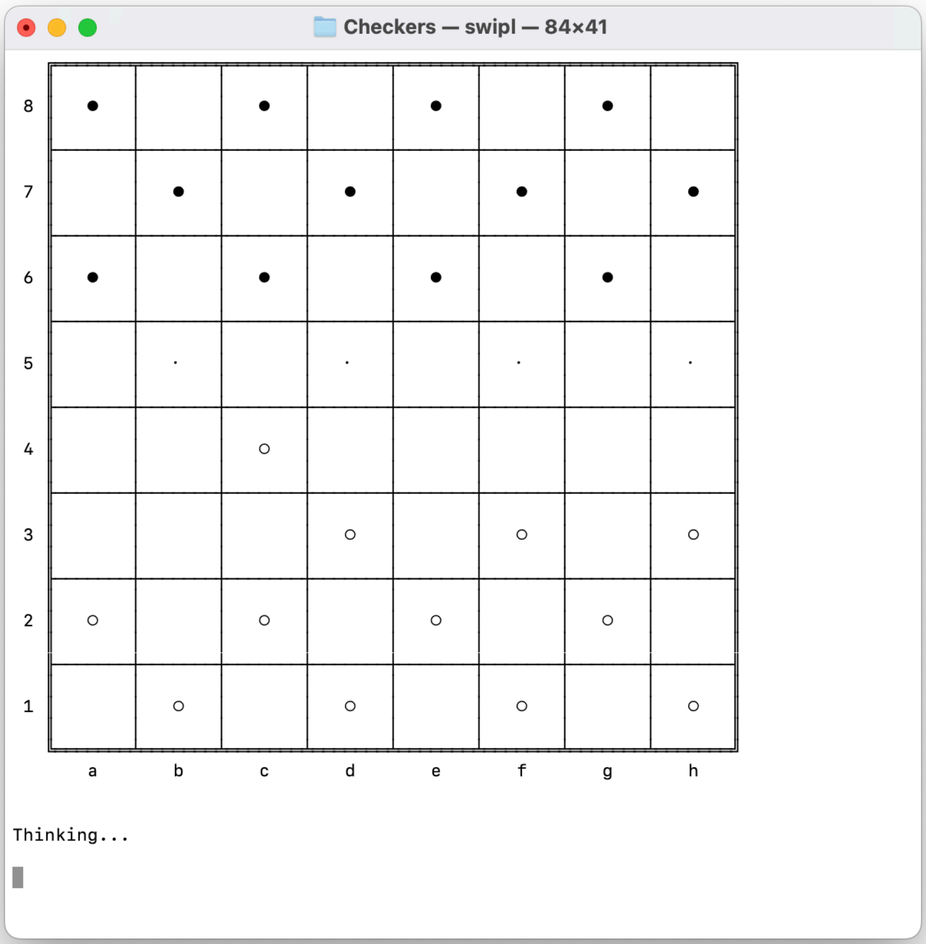
```
Checkers — swipl — 84×41

?- [main].
To start a game, type play.
true.

?- play.

───────────────────────────────────────────────────────────────
 ░░░░░░ ░░   ░░ ░░░░░░░  ░░░░░░ ░░   ░░ ░░░░░░░ ░░░░░░  ░░░░░░░
▒▒      ▒▒   ▒▒ ▒▒      ▒▒      ▒▒  ▒▒  ▒▒      ▒▒   ▒▒ ▒▒     
▒▒      ▒▒▒▒▒▒▒ ▒▒▒▒▒   ▒▒      ▒▒▒▒▒   ▒▒▒▒▒   ▒▒▒▒▒▒  ▒▒▒▒▒▒▒
▓▓      ▓▓   ▓▓ ▓▓      ▓▓      ▓▓  ▓▓  ▓▓      ▓▓   ▓▓      ▓▓
 ██████ ██   ██ ███████  ██████ ██   ██ ███████ ██   ██ ███████
───────────────────────────────────────────────────────────────

To move the pieces, enter coordinates followed by a dot. Example: b3c4.


Select your color [white./black.]: white.
```

The application will print the board, highlighting the available moves with a small point. The user plays a move by simply entering the coordinates of the square containing the piece to move, followed by the coordinates of the landing square (e.g.: “`b3c4.`”).

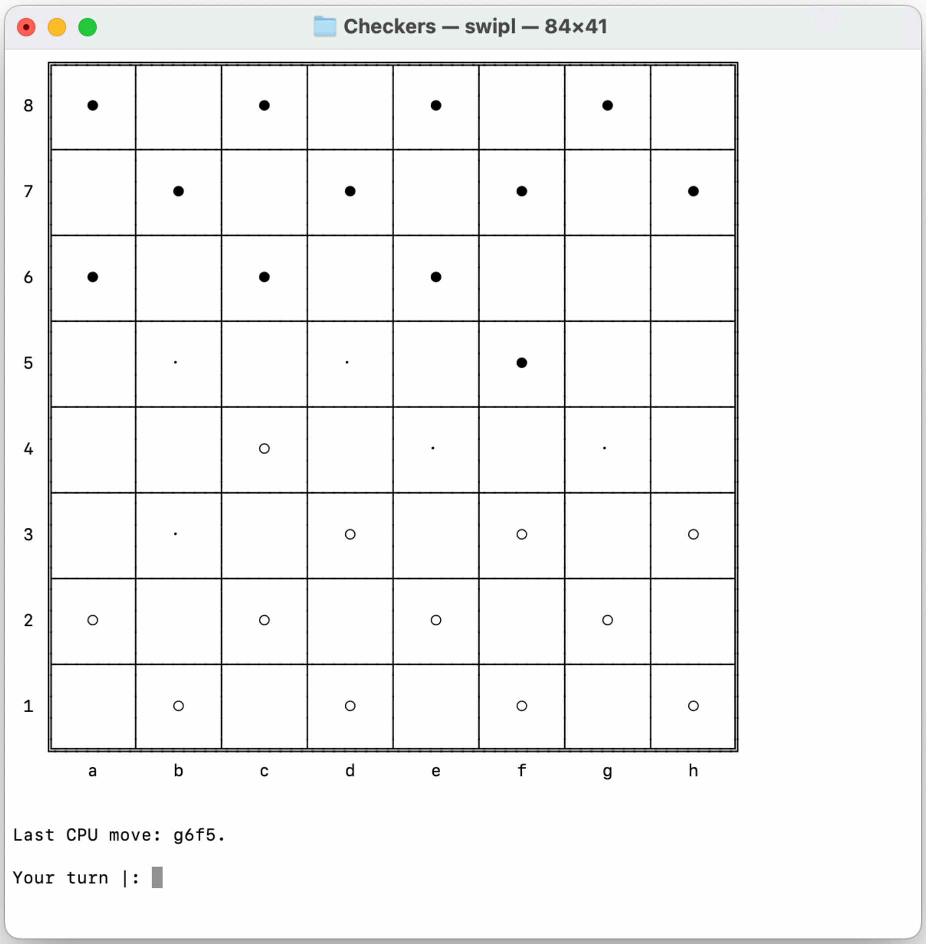


```
8  ●     ●     ●     ●
7     ●     ●     ●     ●
6  ●     ●     ●     ●
5
4  ·     ·     ·     ·
3     ○     ○     ○     ○
2  ○     ○     ○     ○
1     ○     ○     ○     ○
   a  b  c  d  e  f  g  h

Your turn |: b3c4.
```

The system will think about what is the optimal move to make. Then it will move the piece, ending its turn.



```
8 | ● |   | ● |   | ● |   | ● |   |
7 |   | ● |   | ● |   | ● |   | ● |
6 | ● |   | ● |   | ● |   | ● |   |
5 |   | · |   | · |   | · |   | · |
4 |   |   | ○ |   |   |   |   |   |
3 |   |   |   | ○ |   | ○ |   | ○ |
2 | ○ |   | ○ |   | ○ |   | ○ |   |
1 |   | ○ |   | ○ |   | ○ |   | ○ |
    a   b   c   d   e   f   g   h

Thinking...
```



```
8 | ● |   | ● |   | ● |   | ● |   |
7 |   | ● |   | ● |   | ● |   | ● |
6 | ● |   | ● |   | ● |   |   |   |
5 |   | · |   | · |   | ● |   |   |
4 |   |   | ○ |   | · |   | · |   |
3 |   | · |   | ○ |   | ○ |   | ○ |
2 | ○ |   | ○ |   | ○ |   | ○ |   |
1 |   | ○ |   | ○ |   | ○ |   | ○ |
    a   b   c   d   e   f   g   h

Last CPU move: g6f5.

Your turn |:
```

C. Limits and future developments

When searching for the best move, the algorithm starts with a depth of 4 and if the search time has not exceeded 3 seconds, it goes on with a greater depth. Although reasonable, these values could be limiting in the final stages of the game, when a deeper analysis could be decisive to win.

Relying on a good evaluation function is crucial to improve the quality of the game. A systematic analysis of possible features and their weights could lead to a significant improvement in performance. Moreover, adding up the features implies that each of them is assumed to be independent of the others. For this reason, also nonlinear functions should be considered.

Anyway, the approximate nature of the evaluation function leads to some known problems. First, it should be applied only to quiescent states, where there is no pending move that would wildly swing the evaluation. Secondly, the program may run into the horizon effect problem. This arises when the program faces an opponent's move that causes damage and is unavoidable, but can be temporarily delayed. In this case, the program makes non useful moves that just pushes the inevitable damage beyond the search limit. These problems can be alleviated by extending the search in turbulent positions beyond the depth limit, until a quiescent state is reached.

VI. References

1. Administrator. (2023, March 15). *Le Regole per giocare a dama*. Sito della Federazione Italiana Dama. http://www.federdama.it/cms/la-dama
2. Wikipedia contributors. (2023, March 15). *Checkers*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Checkers
3. Russell, S., & Norvig, P. (2021). *Artificial intelligence: A modern approach, global edition* (4th ed.). Pearson Education.
4. Bratko, I. (2011). *Prolog Programming for Artificial Intelligence* (4th ed.). Addison-Wesley Educational.
5. Mellish, C. S., & Clocksin, W. F. (2003). *Programming in Prolog: Using the ISO Standard* (5th ed.). Springer.
6. Zhoufeng, Y., Pengyao, Z., Yajie, W., Fei, L., & Hongkun, Q. (2017). Research and implementation of static evaluation algorithm for checkers. *2017 29th Chinese Control And Decision Conference (CCDC).*