Homework 1

Student: Alessandro Mattei

Matricola: 295441

Email: alessandro.mattei1@student.univaq.it

Introduzione

Nel Homework 2 è stato richiesto di modificare l'algoritmo MinMaxAlphaBetaPruning in tre modi:

- Utilizzando l'H0 cutoff
- Utilizzando l'Hl cutoff
- Utilizzando un regressore non lineare

Tutto questo per verificare se con queste modifiche si migliorasse l'algoritmo MinMax Alpha Beta Pruning e se si riuscisse ad arrivare a profondità maggiori della prima sperimentazione effettuata nel primo Homework.

Nel gioco troviamo una classe Agent generica che si può utilizzare per qualsiasi gioco che prende in input un **search_algorithm** e un **initial_state**. L'Agente tramite la funzione **do_action** ritornerà un nuovo stato di gioco tramite l'utilizzo del **search_algorithm** e di un euristica. L'Agente dopo n iterazioni risolverà entrami i giochi proposti.

Più avanti verrà mostrato e descritto l'implementazione delle classi e delle funzioni scritte in Python, maggiori dettagli li troverete nel file:

- AlessandroMattei_ChessGame.Alhw2.ipynb

Nella fase finale viene mostrata e descritta una analisi statistica dei risultati delle varie sperimentazioni eseguite con istanze di diversi algoritmi di ricerca o configurazioni di essi.

Implementazione

Agente - Agent Class

```python
class Agent:
    """
    Represents an agent that can act based on a given search algorithm and its current view of the world.

    Attributes:
        search_algorithm: A search algorithm that the agent uses to make decisions.
        view: The agent's current view of the world.
        old_view: The agent's previous view of the world.
    """

    def __init__(self, search_algorithm, initial_state):
        """
        Initializes the Agent with a search algorithm and an initial state.

        :param search_algorithm: The search algorithm to be used by the agent.
        :param initial_state: The initial state of the world as perceived by the agent.
        """
        self.search_algorithm = search_algorithm
        self.view = initial_state
        self.old_view = None

    def do_action(self, current_state_world):
        """
        Updates the agent's view based on the current state of the world and the search algorithm.
        :param current_state_world: The current state of the world.
        :return: The updated view of the agent.
        """
        self.view = self.search_algorithm.search(current_state_world)
        self.old_view = current_state_world
        return self.view
```

La classe agente è indipendente dal tipo di gioco o problema che si vuole risolvere. Si occupa di richiamare l'algoritmo di ricerca (**search_algorithm**), tramite il metodo **do_action**, il quale ritornerà uno stato successivo passando come parametro lo stato attuale. L'agente viene chiamato dalla funzione "main" a ogni mossa e restituisce lo stato successivo migliore (secondo l'euristica scelta) che verrà a sua volta impiegato nel successivo ciclo come parametro fino alla fine dell'esecuzione.

Algoritmi di Ricerca Implementati

MinMaxAlpha-BetaPruning

L'algoritmo MinMax con Alpha-Beta Pruning è una ottimizzazione dell'algoritmo MinMax tradizionale, utilizzato nei giochi a due giocatori come Scacchi.

Il suo obiettivo principale è ridurre il numero di nodi valutati nell'albero di ricerca, "tagliando" rami che non influenzeranno la decisione finale. Questo permette di esplorare alberi più profondi in meno tempo, migliorando le prestazioni.

La strategia si basa sull'utilizzo di due parametri chiave: alpha e beta. Immaginando che il giocatore 1 sia quello che mira a massimizzare il punteggio e il giocatore 2 a minimizzarlo:

- Alpha simbolizza il punteggio minimo che il giocatore 1 può assicurarsi nella posizione attuale. Sebbene parta dal valore peggiore per il giocatore 1, si aggiorna costantemente in base alla mossa più vantaggiosa che il giocatore 1 potrebbe fare.
- Beta, al contrario, rappresenta il punteggio ottimale che il giocatore 2 può aspirare a ottenere. Anch'esso inizia dal valore peggiore per il giocatore 2, ma si rinnova considerando la mossa migliore individuata per il giocatore 2 fino a quel punto.

La dinamica procede seguendo la struttura della ricerca MinMax, con aggiornamenti continui di alpha e beta ad ogni nodo esaminato. Se, in una certa fase dell'analisi, alpha dovesse superare beta, l'esplorazione del ramo attuale viene interrotta, permettendo all'algoritmo di concentrarsi su percorsi alternativi. Così facendo, l'intero sotto-albero legato a nodi in cui i valori di alpha e beta si "incrociano" viene bypassato, ottimizzando l'efficienza dell'analisi.

L'algoritmo per essere istanziato ha bisogno dell'euristica, del gioco e della profondità alla quale deve lavorare. La variabile *eval_count* è una variabile che conta il numero degli stati valutati utile per stampare i risultati e la variabile *prune_count* è una variabile che ci dice quanti elementi sono stati potati

```
def __init__(self, game, heuristic, max_depth=1):
    """
    Initializes an instance of the MinMaxAlphaBetaPruning class.
    :param game: The game for which the search is performed.
    :param heuristic: The heuristic to evaluate the game states.
    :param max_depth: Maximum depth of the search. Default is 1.
    """
    self.game = game
    self.heuristic = heuristic
    self.max_depth = max_depth
    self.prune_count = 0
    self.eval_count = 0
```

Come si può vedere dal file *AlessandroMattei_ChessGame.AIhw1.ipynb*, nel quale è contenuta l'intera implementazione, possiamo notare la presenza di tre metodi **__minmax_alpha_beta()**, **evaluate()**, **pick()** e **search()**.

Di seguito possiamo vedere **pick()**:

```
def pick(states, parent_turn):
    """
    Picks the best state based on the heuristic values.

    This function evaluates a list of game states and selects the state that optimizes
    the current player's position.
    If it is the maximizing player's turn (parent_turn is True), the state with the highest heuristic
    value is chosen.
    Otherwise, if it is the minimizing player's turn (parent_turn is False), the state with the lowest heuristic
    value is chosen.

    :param states: List of game states to pick from.
    :param parent_turn: Indicates whose turn it is: True for maximizing player and False for minimizing player.
    :return: The best state based on the heuristic value.
    """
    if parent_turn:
        # If it's the maximizing player's turn, select the state with the highest heuristic value.
        return max(states, key=lambda state: state.h)
    else:
        # If it's the minimizing player's turn, select the state with the lowest heuristic value.
        return min(states, key=lambda state: state.h)
```

La funzione pick() restituisce, in base al turno (True per giocatore 1 e False per giocatore 2), lo stato con valore estimate massimo o minimo tra gli stati neighbors per ogni mossa. È uguale alla funzione che troviamo nel MinMax

Di seguito possiamo vedere **evaluate()**:

```
def evaluate(self, states, parent_turn):
    """
    Evaluates a list of states and updates their heuristic values.

    :param states: A list of game states to evaluate.
    :param parent_turn: A flag indicating if it's the parent player's turn.
    """
    for state in states:
        # If a draw can be claimed in the current state, set heuristic value to 0.0.
        if state.game_board.can_claim_draw():
            state.h = 0.0
        else:
```

```
            # Otherwise, evaluate the state using the Minimax algorithm with Alpha-Beta pruning.
            state.h = self.__minmax_alpha_beta(state, self.max_depth - 1, float("-inf"), float("inf"),
                                               not parent_turn)
```

La funzione evaluate() di MinMaxAlphaBetaPruning ha in più rispetto all'algoritmo MinMax le due variabili alpha e beta.

Di seguito possiamo vedere la funzione helper **__minmax_alpha_beta()**:

```
def __minmax_alpha_beta(self, state, depth, alpha, beta, turn):
    """
    Private method implementing the Minimax algorithm with Alpha-Beta pruning.

    :param state: The current game state.
    :param depth: The current depth in the game tree.
    :param alpha: The alpha value for Alpha-Beta pruning.
    :param beta: The beta value for Alpha-Beta pruning.
    :param turn: Flag indicating if it's the maximizing player's turn.
    :return: The heuristic value of the state.
    """
    self.eval_count += 1

    # Base case: if maximum depth is reached or the game is over, return the heuristic value of the state.
    if depth == 0 or state.game_board.is_game_over():
        return self.heuristic.h(state)

    # Generate all possible moves (neighbors) from the current state.
    neighbors = self.game.neighbors(state)

    if turn:  # If it's the maximizing player's turn.
        value = float("-inf")
        for neighbor in neighbors:
            # Recursively call the function to evaluate the neighbor state, updating the value and alpha.
            value = max(value, self.__minmax_alpha_beta(neighbor, depth - 1, alpha, beta, False))
            alpha = max(alpha, value)
            # Alpha-Beta pruning: if alpha is greater or equal to beta, prune this branch.
            if alpha >= beta:
                self.prune_count += 1
                break
        return value
    else:  # If it's the minimizing player's turn.
        value = float("inf")
        for neighbor in neighbors:
            # Similarly, for the minimizing player, update the value and beta.
            value = min(value, self.__minmax_alpha_beta(neighbor, depth - 1, alpha, beta, True))
            beta = min(beta, value)
            # Alpha-Beta pruning: if beta is less or equal to alpha, prune this branch.
            if beta <= alpha:
                self.prune_count += 1
                break
        return value
```

Nella funzione helper **__minmax_alpha_beta()** dell'algoritmo MinMaxAlphaBetaPruning, viene integrata la fase di "pruning", che verifica la convenienza di uno stato. Se questo stato risulta non vantaggioso, l'analisi del ramo corrispondente viene interrotta. Le variabili *eval_count* e *prune_count* servono rispettivamente a monitorare il numero di stati esaminati e il numero di potature realizzate.

MinMaxAlpha-BetaPruning H0 CutOff

Questa versione di MinMax con Alpha-Beta Pruning è una delle tre versioni che ottimizzano i tempi di valutazione e che ha lo scopo di vedere in profondità in un breve periodo di tempo.

Il suo obiettivo principale è ridurre il numero di nodi valutati nell'albero di ricerca, "tagliando" i primi rami usando una valutazione statica con un euristica H0, che nel mio caso è l'euristica SoftBoardEvaluationChessGame.

Questo permette di esplorare alberi più profondi in meno tempo, migliorando le prestazioni di "scoperta".

L'algoritmo per essere istanziato ha bisogno dell'euristica di evaluation, dell'euristica di taglio h0, del gioco e della profondità alla quale deve lavorare. La variabile *eval_count* è una variabile che conta il numero degli stati valutati dal semplice minmax alpha beta utile per stampare i risultati e la variabile *prune_count* è una variabile che ci dice quanti elementi sono stati potati dal semplice minmax alpha beta, abbiamo poi anche i valori *eval_h0_cut_count* e *eval_h0_cut_count* che svolgono la stessa funzione ma sono riferiti al processo h0.

È importante notare che è stato introdotto un dizionario contenente le valutazioni già effettuate. Questo significa che se ci troviamo di fronte a uno stato e una profondità già calcolati in precedenza, eviteremo di ricalcolare la valutazione, ottimizzando così il processo.

```
def __init__(self, game, heuristic, h0_cut, k=5, max_depth=1):
    """
    Initializes the MinMaxAlphaBetaPruningH0Cut class with game settings, heuristics, and search parameters.

    :param game: The current state of the chess game.
    :param heuristic: Main heuristic function used for evaluating game states.
```

```
    :param h0_cut: Secondary heuristic function used for h0 cutoff.
    :param k: Number of states to consider after applying the h0 cutoff. Defaults to 5.
    :param max_depth: Maximum depth for the Minimax search. Defaults to 1.
    """
    self.game = game  # The current state of the chess game.
    self.heuristic = heuristic  # Main heuristic function used to evaluate game states.
    self.h0_cut = h0_cut  # Secondary heuristic used for the h0 cutoff.
    self.k = k  # Number of states to consider after applying the h0 cutoff.
    self.max_depth = max_depth  # Maximum depth for the Minimax search.
    self.prune_count = 0  # Count of pruned branches in the main search.
    self.eval_count = 0  # Count of evaluations in the main search.
    self.eval_h0_cut_count = 0  # Count of evaluations for the h0 cutoff.
    self.prune_h0_cut_count = 0  # Count of pruned branches due to the h0 cutoff.
    self.memoization = {}  # Dictionary for storing previously calculated states.
```

Come si può vedere dal file *AlessandroMattei_ChessGame.AIhw2.ipynb*, nel quale è contenuta l'intera implementazione, possiamo notare la presenza dei metodi **__minmax_alpha_beta()**, **__h0_cut()**, **evaluate()**, **pick()** e **search()**.

Soffermiamoci a vedere solo le parti cambiate dal canonico MinMax Alpha-Beta. Gradiamo prima come è stato implementata la funzione **__h0_cut()**

```
def __h0_cut(self, states, turn):
    """
    Applies the h0 cutoff heuristic to limit the number of states considered.

    :param states: A list of game states.
    :param turn: Flag indicating the current player's turn.
    :return: A list of states after applying the h0 cutoff.
    """
    initial_count = len(states)
    # Evaluate states using the h0 heuristic and count evaluations.
    for state in states:
        state.h0 = self.h0_cut.h(state)
        self.eval_h0_cut_count += 1

    # Sort and select the top k states based on the h0 heuristic value.
    sorted_states = sorted(states, key=lambda state: state.h0, reverse=turn)[:self.k]
    # Count how many states were pruned by this process.
    self.prune_h0_cut_count += initial_count - len(sorted_states)

    return sorted_states
```

Questo metodo è privato e applica la euristica h0 cutoff per limitare il numero di stati considerati durante la ricerca. Prende una lista di stati possibili **states** e una variabile **turn** che indica il turno del giocatore corrente. Per ciascuno degli stati nella lista, calcola un valore euristico h0 utilizzando la funzione **h0_cut.h(state)** e tiene traccia delle valutazioni tramite **eval_h0_cut_count**. Successivamente, ordina gli stati in base ai valori h0 in ordine decrescente (o crescente, a seconda del turno) e restituisce i primi k stati, dove k è il numero di stati da considerare dopo l'applicazione dell'h0 cutoff. Questo metodo tiene anche traccia del numero di stati che sono stati "potati" dalla lista iniziale tramite **prune_h0_cut_count**

Guardiamo **__minmax_alpha_beta()**

```
def __minmax_alpha_beta(self, state, depth, alpha, beta, turn):
    """
    Private method implementing the Minimax algorithm with Alpha-Beta pruning and memoization.

    :param state: The current game state.
    :param depth: The current depth in the game tree.
    :param alpha: The alpha value for Alpha-Beta pruning.
    :param beta: The beta value for Alpha-Beta pruning.
    :param turn: Flag indicating if it's the maximizing player's turn.
    :return: The heuristic value of the state.
    """
    self.eval_count += 1

    # Check if the state is already evaluated and stored in memoization.
    if (state, depth, turn) in self.memoization:
        return self.memoization[(state, depth, turn)]

    # Base case: if maximum depth is reached or the game is over, return the heuristic value.
    if depth == 0 or state.game_board.is_game_over():
        return self.heuristic.h(state)

    # Generate possible moves (neighbors), applying the h0 cutoff.
    neighbors = self.game.neighbors(state)
    top_neighbors = self.__h0_cut(neighbors, state.game_board.turn)

    if turn:  # Maximizing player's turn.
        value = float("-inf")
        for neighbor in top_neighbors:
```

```
            # Recursively evaluate the state, update value and alpha.
            value = max(value, self.__minmax_alpha_beta(neighbor, depth - 1, alpha, beta, False))
            alpha = max(alpha, value)
            # Alpha-Beta pruning: prune if alpha >= beta.
            if alpha >= beta:
                self.prune_count += 1
                break
        self.memoization[(state, depth, turn)] = value
        return value
    else:  # Minimizing player's turn.
        value = float("inf")
        for neighbor in top_neighbors:
            # Similar evaluation for the minimizing player.
            value = min(value, self.__minmax_alpha_beta(neighbor, depth - 1, alpha, beta, True))
            beta = min(beta, value)
            # Prune if beta <= alpha.
            if beta <= alpha:
                self.prune_count += 1
                break
        self.memoization[(state, depth, turn)] = value
        return value
```

Questo metodo privato implementa l'algoritmo Minimax con potatura Alpha-Beta, con memorizzazione degli stati e depth valutati e utilizza __h0_cut per valutare ed esplorare solo gli stati più promettenti. Prende come argomenti lo stato corrente state, la profondità corrente nella ricerca depth, i valori alpha e beta per la potatura Alpha-Beta, e un flag turn che indica se è il turno del giocatore massimizzante. Questo metodo valuta ricorsivamente gli stati nel gioco, utilizzando la memorizzazione per evitare di valutare più volte gli stessi stati con la stessa depth. Applica la potatura Alpha-Beta per ridurre il numero di stati da considerare e calcola il valore euristico del miglior stato possibile. Questo metodo tiene traccia del numero di valutazioni effettuate tramite eval_count.

Guardiamo **__search()**

```
def search(self, state: StateChessGame):
    """
    Public method to start the search with Alpha-Beta pruning and h0 cutoff.

    :param state: The current state of the chess game.
    :return: The best next state for the current player.
    """
    # Generate possible moves, applying the h0 cutoff.
    neighbors = self.game.neighbors(state)
    top_neighbors = self.__h0_cut(neighbors, state.game_board.turn)
    # Evaluate the top neighbors and choose the best move based on the player's turn.
    self.evaluate(top_neighbors, state.game_board.turn)
    return self.pick(top_neighbors, state.game_board.turn)
```

Questo metodo pubblico avvia la ricerca utilizzando l'algoritmo Minimax con potatura Alpha-Beta e l'h0 cutoff. Prende uno stato state come input, genera mosse possibili applicando l'h0 cutoff, quindi valuta queste mosse e restituisce la migliore mossa possibile in base al turno del giocatore corrente. La valutazione viene effettuata utilizzando il metodo evaluate, e la scelta della mossa migliore viene fatta utilizzando il metodo pick.

I restanti metodi non sono cambiati.

MinMaxAlpha-BetaPruning Hl CutOff

Questa versione di MinMax con Alpha-Beta Pruning è una delle tre versioni che ottimizzano i tempi di valutazione e che ha lo scopo di vedere in profondità in un breve periodo di tempo.

Il suo obiettivo principale è ridurre il numero di nodi valutati nell'albero di ricerca, "tagliando" i primi rami usando una valutazione dinamica hl cioè valutando i primi stati in profondità l, vine anche usato il "taglio" h0 che usa l'euristica SoftBoardEvaluationChessGame all'interno delle depth del minmax.

Questo permette di esplorare alberi più profondi in meno tempo, migliorando le prestazioni di "scoperta".

L'algoritmo per essere istanziato ha bisogno dell'euristica di evaluation, dell'euristica di taglio h0, del gioco e della profondità alla quale deve lavorare. La variabile *eval_count* è una variabile che conta il numero degli stati valutati dal semplice minmax alpha beta utile per stampare i risultati e la variabile *prune_count* è una variabile che ci dice quanti elementi sono stati potati dal semplice minmax alpha beta, abbiamo poi anche i valori *eval_h0_cut_count* e *eval_h0_cut_count* che svolgono la stessa funzione ma sono riferiti al processo h0 e *eval_hl_cut_count* e *eval_hl_cut_count* al processo hl.

È importante notare che è stato introdotto un dizionario contenente le valutazioni già effettuate. Questo significa che se ci troviamo di fronte a uno stato e una profondità già calcolati in precedenza, eviteremo di ricalcolare la valutazione, ottimizzando così il processo.

```
def __init__(self, game, heuristic, h0_cut, k=5, l=3, max_depth=1):
    """
    Initializes the MinMaxAlphaBetaPruningHlCut class with game settings, heuristics, and search parameters.

    :param game: The current state of the chess game.
    :param heuristic: Main heuristic function used for evaluating game states.
    :param h0_cut: Heuristic function used for the h0 cutoff.
    :param k: Number of states to consider after applying the h0 and hl cutoffs. Defaults to 5.
    :param l: Depth for the hl cutoff calculation. Defaults to 3.
    :param max_depth: Maximum depth for the Minimax search. Defaults to 1.
    """
```

```
    self.game = game  # The current state of the chess game.
    self.heuristic = heuristic  # Main heuristic function for evaluating game states.
    self.h0_cut = h0_cut  # Heuristic function used for the h0 cutoff.
    self.k = k  # Number of states to consider after applying the h0 and hl cutoffs.
    self.l = l  # Depth for the hl cutoff calculation.
    self.max_depth = max_depth  # Maximum depth for the Minimax search.
    self.prune_count = 0  # Count of pruned branches in the main search.
    self.eval_count = 0  # Count of evaluations in the main search.
    self.eval_h0_cut_count = 0  # Count of evaluations for the h0 cutoff.
    self.prune_h0_cut_count = 0  # Count of pruned branches due to the h0 cutoff.
    self.eval_hl_cut_count = 0  # Count of evaluations for the hl cutoff.
    self.prune_hl_cut_count = 0  # Count of pruned branches due to the hl cutoff.
    self.memoization = {}  # Dictionary for storing previously calculated states.
```

Come si può vedere dal file *AlessandroMattei_ChessGame.Alhw2.ipynb*, nel quale è contenuta l'intera implementazione, possiamo notare la presenza dei metodi **__minmax_alpha_beta()**, **__h0_cut()**, **__hl_cut()**, **__minmax_alpha_beta_hl()**, **evaluate()**, **pick()** e **search()**.

Soffermiamoci a vedere solo le parti cambiate dal canonico MinMax Alpha-Beta. Analizziamo prima la funzione helper **__hl_cut()**:

```
def __hl_cut(self, states, turn):
    """
    Applies the hl cutoff heuristic to further limit the number of states considered.

    :param states: A list of game states.
    :param turn: Flag indicating the current player's turn.
    :return: A list of states after applying the hl cutoff.
    """
    initial_count = len(states)
    # Evaluate states using a deeper level of the Minimax algorithm (hl cutoff).
    for state in states:
        state.hl = self.__minmax_alpha_beta_hl(state, self.l - 1, float("-inf"), float("inf"), not turn)
    # Sort and select the top k states based on the hl heuristic value.
    sorted_states = sorted(states, key=lambda state: state.hl, reverse=turn)[:self.k]
    # Count how many states were pruned by this process.
    self.prune_hl_cut_count += initial_count - len(sorted_states)
    return sorted_states
```

Questo è un metodo privato che applica l'euristica di taglio hl per limitare ulteriormente il numero di stati considerati durante la ricerca. Prende una lista di stati possibili states e una variabile turn che indica il turno del giocatore corrente. P er ciascuno degli stati nella lista, calcola un valore euristico hl utilizzando il metodo __minmax_alpha_beta_hl(state, depth, alpha, beta, not turn). Questo valore hl viene utilizzato per valutare e ordinare gli stati. Successivamente, restituisce i primi k stati in base ai valori hl (in ordine decrescente o crescente, a seconda del turno) e tiene traccia del numero di stati che sono stati "potati" dalla lista iniziale tramite prune_hl_cut_count.

Gradiamo ora il metodo **__minmax_alpha_beta_hl()**:

```
def __minmax_alpha_beta_hl(self, state, depth, alpha, beta, turn):
    """
    Implements a deeper level of the Minimax algorithm for the hl cutoff.

    :param state: The current game state.
    :param depth: The current depth in the game tree.
    :param alpha: The alpha value for Alpha-Beta pruning.
    :param beta: The beta value for Alpha-Beta pruning.
    :param turn: Flag indicating if it's the maximizing player's turn.
    :return: The heuristic value of the state.
    """
    self.eval_hl_cut_count += 1

    # Base case: if maximum depth is reached or the game is over, return the heuristic value from h0_cut.
    if depth == 0 or state.game_board.is_game_over():
        return self.h0_cut.h(state)

    neighbors = self.game.neighbors(state)

    if turn:  # Maximizing player's turn.
        value = float("-inf")
        for neighbor in neighbors:
            # Recursively evaluate the state for hl cutoff, update value and alpha.
            value = max(value, self.__minmax_alpha_beta_hl(neighbor, depth - 1, alpha, beta, False))
            alpha = max(alpha, value)
            # Alpha-Beta pruning for hl cutoff.
            if alpha >= beta:
                self.prune_hl_cut_count += 1
                break
        return value
    else:  # Minimizing player's turn.
        value = float("inf")
```

```
        for neighbor in neighbors:
            # Similar evaluation for the minimizing player for hl cutoff.
            value = min(value, self.__minmax_alpha_beta_hl(neighbor, depth - 1, alpha, beta, True))
            beta = min(beta, value)
            # Prune if beta <= alpha in hl cutoff.
            if beta <= alpha:
                self.prune_hl_cut_count += 1
                break
        return value
```

Questo è un metodo privato che implementa una versione più profonda dell'algoritmo Minimax con potatura Alpha-Beta per il taglio hl. Prende come argomenti lo stato corrente state, la profondità corrente nella ricerca depth, i valori alpha e beta per la potatura Alpha-Beta, e un flag turn che indica se è il turno del giocatore massimizzante. Questo metodo valuta ricorsivamente gli stati nel gioco utilizzando la profondità specificata l e calcola il valore euristico del miglior stato possibile. Questo metodo tiene traccia del numero di valutazioni effettuate tramite eval_hl_cut_count.

Analizziamo il metodo **__h0_cut()**:

```
def __h0_cut(self, states, turn):
    """
    Applies the h0 cutoff heuristic to limit the number of states considered.

    :param states: A list of game states.
    :param turn: Flag indicating the current player's turn.
    :return: A list of states after applying the h0 cutoff.
    """
    initial_count = len(states)
    # Evaluate states using the h0 heuristic and count evaluations.
    for state in states:
        state.h0 = self.h0_cut.h(state)
        self.eval_h0_cut_count += 1

    # Sort and select the top k states based on the h0 heuristic value.
    sorted_states = sorted(states, key=lambda state: state.h0, reverse=turn)[:self.k]
    # Count how many states were pruned by this process.
    self.prune_h0_cut_count += initial_count - len(sorted_states)

    return sorted_states
```

Questo è un metodo privato che applica l'euristica di taglio h0 per limitare il numero di stati considerati durante la ricerca. Il suo funzionamento è simile al metodo __hl_cut, ma applica l'euristica h0 invece di hl e tiene traccia del numero di stati "potati" tramite prune_h0_cut_count.

Guardiamo **__minmax_alpha_beta()**

```
def __minmax_alpha_beta(self, state, depth, alpha, beta, turn):
    """
    Private method implementing the Minimax algorithm with Alpha-Beta pruning and memoization.

    :param state: The current game state.
    :param depth: The current depth in the game tree.
    :param alpha: The alpha value for Alpha-Beta pruning.
    :param beta: The beta value for Alpha-Beta pruning.
    :param turn: Flag indicating if it's the maximizing player's turn.
    :return: The heuristic value of the state.
    """
    self.eval_count += 1

    # Check if the state is already evaluated and stored in memoization.
    if (state, depth, turn) in self.memoization:
        return self.memoization[(state, depth, turn)]

    # Base case: if maximum depth is reached or the game is over, return the heuristic value.
    if depth == 0 or state.game_board.is_game_over():
        return self.heuristic.h(state)

    # Generate possible moves (neighbors), applying the h0 cutoff.
    neighbors = self.game.neighbors(state)
    top_neighbors = self.__h0_cut(neighbors, state.game_board.turn)

    if turn:  # Maximizing player's turn.
        value = float("-inf")
        for neighbor in top_neighbors:
            # Recursively evaluate the state, update value and alpha.
            value = max(value, self.__minmax_alpha_beta(neighbor, depth - 1, alpha, beta, False))
            alpha = max(alpha, value)
            # Alpha-Beta pruning: prune if alpha >= beta.
            if alpha >= beta:
                self.prune_count += 1
                break
```

```
            self.memoization[(state, depth, turn)] = value
            return value
        else:  # Minimizing player's turn.
            value = float("inf")
            for neighbor in top_neighbors:
                # Similar evaluation for the minimizing player.
                value = min(value, self.__minmax_alpha_beta(neighbor, depth - 1, alpha, beta, True))
                beta = min(beta, value)
                # Prune if beta <= alpha.
                if beta <= alpha:
                    self.prune_count += 1
                    break
            self.memoization[(state, depth, turn)] = value
            return value
```

Questo metodo privato implementa l'algoritmo Minimax con potatura Alpha-Beta, con memorizzazione degli stati e depth valutati e utilizza __hl_cut e __h0_cut per valutare ed esplorare solo gli stati più promettenti. Prende come argomenti lo stato corrente state, la profondità corrente nella ricerca depth, i valori alpha e beta per la potatura Alpha-Beta, e un flag turn che indica se è il turno del giocatore massimizzante. Questo metodo valuta ricorsivamente gli stati nel gioco, utilizzando la memorizzazione per evitare di valutare più volte gli stessi stati con la stessa depth. Applica la potatura Alpha-Beta per ridurre il numero di stati da considerare e calcola il valore euristico del miglior stato possibile. Questo metodo tiene traccia del numero di valutazioni effettuate tramite eval_count.

Guardiamo **__search()**

```
def search(self, state: StateChessGame):
    """
    Public method to start the search with Alpha-Beta pruning, h0, and hl cutoffs.

    :param state: The current state of the chess game.
    :return: The best next state for the current player.
    """
    # Generate possible moves, applying the hl cutoff.
    neighbors = self.game.neighbors(state)
    top_neighbors = self.__hl_cut(neighbors, state.game_board.turn)
    # Evaluate the top neighbors and choose the best move based on the player's turn.
    self.evaluate(top_neighbors, state.game_board.turn)
    return self.pick(top_neighbors, state.game_board.turn)
```

Questo è un metodo pubblico che avvia la ricerca utilizzando l'algoritmo Minimax con potatura Alpha-Beta, insieme alle euristiche di taglio hl e poi h0. Prende uno stato state come input, genera mosse possibili applicando il taglio hl, quindi valuta queste mosse e restituisce la migliore mossa possibile in base al turno del giocatore corrente. La valutazione viene effettuata utilizzando il metodo evaluate, e la scelta della mossa migliore viene fatta utilizzando il metodo pick.

I restanti metodi non sono cambiati.

MinMaxAlpha-BetaPruning Hr CutOff (MLPRegressor)

Questa versione di MinMax con Alpha-Beta Pruning è una delle tre versioni che ottimizzano i tempi di valutazione e che ha lo scopo di vedere in profondità in un breve periodo di tempo.

Il suo obiettivo principale è ridurre il numero di nodi valutati nell'albero di ricerca, "tagliando" i primi rami usando un Regressore non-lineare per stabilire quali stati andranno esplorati.

Questo permette di esplorare alberi più profondi in meno tempo, migliorando le prestazioni di "scoperta".

L'algoritmo per essere istanziato ha bisogno dell'euristica di evaluation, dell'euristica di taglio h0, del gioco e della profondità alla quale deve lavorare. La variabile *eval_count* è una variabile che conta il numero degli stati valutati dal semplice minmax alpha beta utile per stampare i risultati e la variabile *prune_count* è una variabile che ci dice quanti elementi sono stati potati dal semplice minmax alpha beta, abbiamo poi anche i valori *eval_hr_cut_count* e *eval_hr_cut_count* che svolgono la stessa funzione ma sono riferiti al processo hr. Una nota importante è obbligatorio aver creato un modello prima di eseguire questo MinMax Alpha-Beta.

È importante notare che è stato introdotto un dizionario contenente le valutazioni già effettuate. Questo significa che se ci troviamo di fronte a uno stato e una profondità già calcolati in precedenza, eviteremo di ricalcolare la valutazione, ottimizzando così il processo.

```
def __init__(self, game, heuristic, k=5, max_depth=1):
    """
    Initializes the MinMaxAlphaBetaPruningHrCut class with game settings, heuristics, and search parameters.

    :param game: The current state of the chess game.
    :param heuristic: Main heuristic function used for evaluating game states.
    :param k: Number of states to consider after applying the hr cutoff. Defaults to 5.
    :param max_depth: Maximum depth for the Minimax search. Defaults to 1.
    """
    self.game = game  # The current state of the chess game.
    self.heuristic = heuristic  # Main heuristic function used to evaluate game states.
    self.k = k  # Number of states to consider after applying the h0 cutoff.
    self.max_depth = max_depth  # Maximum depth for the Minimax search.
    self.prune_count = 0  # Count of pruned branches in the main search.
    self.eval_count = 0  # Count of evaluations in the main search.
    self.eval_hr_cut_count = 0  # Count of evaluations for the h0 cutoff.
    self.prune_hr_cut_count = 0  # Count of pruned branches due to the h0 cutoff.
    self.memoization = {}  # Dictionary for storing previously calculated states.
```

```
    self.mlp_regressor = joblib.load('./mlp_regressor_model.joblib')  # Load the ML regressor model.
    self.observation = ObservationBoard(normalize_result=True)  # Initialize the observation board.
```

Come si può vedere dal file *AlessandroMattei_ChessGame.AIhw2.ipynb*, nel quale è contenuta l'intera implementazione, possiamo notare la presenza dei metodi **__minmax_alpha_beta()**, **__hr_cut()**, **__regressor_eval()**, **evaluate()**, **pick()** e **search()**.

Soffermiamoci a vedere solo le parti cambiate dal canonico MinMax Alpha-Beta. Analizziamo prima la funzione helper **__hr_cut()**:

```
def __hr_cut(self, states, turn):
    """
    Applies the hr cutoff using the ML regressor to limit the number of states considered.

    :param states: A list of game states.
    :param turn: Flag indicating the current player's turn.
    :return: A list of states after applying the hr cutoff.
    """
    initial_count = len(states)

    for state in states:
        observations = self.observation.h_piccoli(state.game_board)  # Get observations from the board.
        state.hr = self.__regressor_eval(observations)  # Evaluate state using the ML regressor.
        self.eval_hr_cut_count += 1

    # Sort and select the top k states based on the hr value.
    sorted_states = sorted(states, key=lambda state: state.hr, reverse=turn)[:self.k]
    # Count how many states were pruned by this process.
    self.prune_hr_cut_count += initial_count - len(sorted_states)

    return sorted_states
```

Questo è un metodo privato che applica l'euristica di taglio hr utilizzando un modello di regressione di machine learning per limitare il numero di stati considerati durante la ricerca. Prende una lista di stati possibili states e una variabile turn che indica il turno del giocatore corrente. P er ciascuno degli stati nella lista, estrae le osservazioni dalla scacchiera utilizzando l'oggetto ObservationBoard e quindi valuta lo stato utilizzando il metodo __regressor_eval(observations) per ottenere un valore hr. Successivamente, restituisce i primi k stati in base ai valori hr (in ordine decrescente o crescente, a seconda del turno) e tiene traccia del numero di stati che sono stati "potati" dalla lista iniziale tramite prune_hr_cut_count.

Guardiamo **__regressor_eval()**

```
def __regressor_eval(self, observations):
    """
    Evaluates a state using the ML regressor.

    :param observations: The observations extracted from the chess board.
    :return: The predicted value from the ML regressor.
    """
    colonne = ['h1', 'h2', 'h3', 'h4', 'h5', 'h6', 'h7', 'h8', 'h9', 'h10',
               'h11', 'h12', 'h13', 'h14', 'h15', 'h16', 'h17', 'h18', 'h19',
               'h20']
    df = pd.DataFrame([observations], columns=colonne)
    return self.mlp_regressor.predict(df)[0]  # Predict and return the first value.
```

Questo è un metodo privato che valuta uno stato utilizzando un modello di regressione di machine learning (ML). Prende le osservazioni estratte dalla scacchiera come input e restituisce il valore previsto dal modello di regressione per quel particolare stato.

Guardiamo **__minmax_alpha_beta()**

```
def __minmax_alpha_beta(self, state, depth, alpha, beta, turn):
    """
    Private method implementing the Minimax algorithm with Alpha-Beta pruning.

    :param state: The current game state.
    :param depth: The current depth in the game tree.
    :param alpha: The alpha value for Alpha-Beta pruning.
    :param beta: The beta value for Alpha-Beta pruning.
    :param turn: Flag indicating if it's the maximizing player's turn.
    :return: The heuristic value of the state.
    """
    self.eval_count += 1

    # Check if the state is already evaluated and stored in memoization.
    if (state, depth, turn) in self.memoization:
        return self.memoization[(state, depth, turn)]

    # Base case: if maximum depth is reached or the game is over, return the heuristic value.
    if depth == 0 or state.game_board.is_game_over():
        return self.heuristic.h(state)

    # Generate possible moves (neighbors), applying the hr cutoff.
    neighbors = self.game.neighbors(state)
```

```
    top_neighbors = self.__hr_cut(neighbors, state.game_board.turn)

    if turn:  # Maximizing player's turn.
        value = float("-inf")
        for neighbor in top_neighbors:
            # Recursively evaluate the state, update value and alpha.
            value = max(value, self.__minmax_alpha_beta(neighbor, depth - 1, alpha, beta, False))
            alpha = max(alpha, value)
            # Alpha-Beta pruning: prune if alpha >= beta.
            if alpha >= beta:
                self.prune_count += 1
                break
        self.memoization[(state, depth, turn)] = value
        return value
    else:  # Minimizing player's turn.
        value = float("inf")
        for neighbor in top_neighbors:
            # Similar evaluation for the minimizing player.
            value = min(value, self.__minmax_alpha_beta(neighbor, depth - 1, alpha, beta, True))
            beta = min(beta, value)
            # Prune if beta <= alpha.
            if beta <= alpha:
                self.prune_count += 1
                break
        self.memoization[(state, depth, turn)] = value
        return value
```

Questo metodo privato implementa l'algoritmo Minimax con potatura Alpha-Beta, con memorizzazione degli stati e depth valutati e utilizza __hr_cut (un MPLRegressor) per valutare ed esplorare solo gli stati più promettenti. Prende come argomenti lo stato corrente state, la profondità corrente nella ricerca depth, i valori alpha e beta per la potatura Alpha-Beta, e un flag turn che indica se è il turno del giocatore massimizzante. Questo metodo valuta ricorsivamente gli stati nel gioco, utilizzando la memorizzazione per evitare di valutare più volte gli stessi stati con la stessa depth. Applica la potatura Alpha-Beta per ridurre il numero di stati da considerare e calcola il valore euristico del miglior stato possibile. Questo metodo tiene traccia del numero di valutazioni effettuate tramite eval_count.

Guardiamo **__search()**

```
def search(self, state: StateChessGame):
    """
    Public method to start the search with Alpha-Beta pruning and hr cutoff.

    :param state: The current state of the chess game.
    :return: The best next state for the current player.
    """
    # Generate possible moves, applying the h0 cutoff.
    neighbors = self.game.neighbors(state)
    top_neighbors = self.__hr_cut(neighbors, state.game_board.turn)
    # Evaluate the top neighbors and choose the best move based on the player's turn.
    self.evaluate(top_neighbors, state.game_board.turn)
    return self.pick(top_neighbors, state.game_board.turn)
```

Questo è un metodo pubblico che avvia la ricerca utilizzando l'algoritmo Minimax con potatura Alpha-Beta, insieme all'euristica di taglio hr. Prende uno stato state come input, genera mosse possibili applicando il taglio hr, quindi valuta queste mosse e restituisce la migliore mossa possibile in base al turno del giocatore corrente. La valutazione viene effettuata utilizzando il metodo evaluate, e la scelta della mossa migliore viene fatta utilizzando il metodo pick.

HEURISTICS

HardBoardEvaluationChessGame - Complex Chess Board Evaluation

Questa euristica combina varie euristiche. È l'euristica più complessa che è presente nel gioco. Ritorna la somma dei risultati delle singole euristiche.

Euristica Combinate:

- evaluate_board_without_king: Componente di valutazione che si concentra sulla scacchiera senza considerare la posizione del re.
- evaluate_central_control_score: Componente di valutazione incentrata sul controllo delle caselle centrali.
- evaluate_king_safety: Componente di valutazione che si concentra sulla sicurezza del Re.
- evaluate_mobility: Componente di valutazione incentrata sulla mobilità dei pezzi.
- evaluate_pawn_structure: Componente di valutazione che si concentra sulla struttura dei pedoni.
- evaluate_piece_positions: Componente di valutazione che si concentra sulle posizioni di tutti i pezzi tranne il re.

Per capire e vedere nel dettaglio come sono state implementate le singole euristiche vedere il file *AlessandroMattei_ChessGame.AIhw2.ipynb*

SoftBoardEvaluationChessGame - Simple Chess Board Evaluation

Questa euristica combina varie euristiche. È l'euristica più semplice che è presente per il gioco Scacchi e viene usata per tagliare gli stati in h0 e hl. Ritorna la somma dei risultati delle singole euristiche

Euristica Combinate:

- evaluate_board_without_king: Componente di valutazione che si concentra sulla scacchiera senza considerare la posizione del re.
- evaluate_central_control_score: Componente di valutazione incentrata sul controllo delle caselle centrali.
- evaluate_king_safety: Componente di valutazione che si concentra sulla sicurezza del Re.
- evaluate_piece_positions: Componente di valutazione che si concentra sulle posizioni di tutti i pezzi tranne il re.

Per capire e vedere nel dettaglio come sono state implementate le singole euristiche vedere il file *AlessandroMattei_ChessGame.AIhw2.ipynb*

Creazione di un Regressore per il MinMax Alpha Beta con Hr CutOff

Creazione del dataset

Prima di creare un regressore da utilizzare per il MinMax Alpha Beta, ho deciso di scaricare un dataset collaudato disponibile su Kaggle: Chess Evaluations.

Questo dataset contiene oltre 12 milioni di righe e due colonne principali:

- La colonna "FEN" che identifica la posizione della scacchiera in formato FEN.
- La colonna "Evaluation" che rappresenta il valore calcolato utilizzando Stockfish 11 con profondità 22.

Per velocizzare la computazione e semplificare il processo di debug, ho suddiviso il dataset totale in 130 file.

Per creare il file CSV, è stata sviluppata una classe dedicata in grado di generare valutazioni a partire da una specifica configurazione della scacchiera. In totale, sono state generate 20 valutazioni per ciascuna configurazione. Il nuovo generatore CSV produce un dataset con 21 colonne:

- Le colonne h1, h2, h3, h4, h5, h6, h7, h8, h9, h10, h11, h12, h13, h14, h15, h16, h17, h18, h19, h20 identificano le singole osservazioni.
- La colonna "HL" identifica il valore calcolato da Stockfish.

Questo dataset sarà fondamentale per addestrare il nostro regressore per il MinMax Alpha Beta.

Vediamo il generatore:

```
import os
from concurrent.futures import ProcessPoolExecutor

import chess
import pandas as pd

from chessgame.heuristics.ObservationBoard import ObservationBoard


def eval_fen(csv_row):
    fen = csv_row['FEN']
    hl = csv_row['Evaluation']
    observation = ObservationBoard(normalize_result=True)
    evaluation = observation.h_piccoli(chess.Board(fen))
    return evaluation + [hl]


def generate_csv():
    # Numero di Lavoratori
    number_of_workers = os.cpu_count()

    heuristic_csv = pd.DataFrame(
        columns=['h1', 'h2', 'h3', 'h4', 'h5', 'h6', 'h7', 'h8', 'h9', 'h10', 'h11', 'h12', 'h13', 'h14', 'h15', 'h16',
                 'h17', 'h18', 'h19', 'h20', 'HL'])

    csv_num_files = 130
    # Percorso della cartella dove si trovano i file
    directory = '../csv/chessdata'

    for i in range(1, csv_num_files + 1):
        csv_file = f"{directory}/chessData_partizione_{i}.csv"
        df_chunk = pd.read_csv(csv_file)
        print(f"\ncarico il csv: {csv_file}")
        # Processa il chunk
        with ProcessPoolExecutor(max_workers=number_of_workers) as executor:
            res = list(executor.map(eval_fen, df_chunk.to_dict('records')))
```

```
        heuristic_csv = pd.concat([heuristic_csv, pd.DataFrame(res, columns=heuristic_csv.columns)])
        print("file elaborato")

    print(f"\nelaborati tutti i {csv_num_files} file. Scrivo il csv finale\n")
    # Salva il nuovo DataFrame in un file CSV
    heuristic_csv.to_csv('../csv/eval_dataset.csv', index=False)
    print("csv finale scritto\n")

    # Mostra le prime righe del nuovo DataFrame
    print(heuristic_csv.head())


if __name__ == '__main__':
    generate_csv()
```

Per capire e vedere nel dettaglio la generazione delle osservazioni e su come sono state implementate vedere il file *AlessandroMattei_ChessGame.AIhw2.ipynb*

Addestramento di un Regressore MLP per Hr con dati da CSV

In questo processo, ho adottato un approccio per addestrare un regressore basato su una rete neurale MLP (Multilayer Perceptron) per stimare Hr. ho utilizzato i dati dall file CSV generato prima che contiene osservazioni della scacchiera e le corrispondenti valutazioni HL ottenute da Stockfish.

```
import datetime

import joblib
import pandas as pd
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor


def train_regressor():
    df = pd.read_csv('../csv/eval_dataset.csv')
    # Separare le features e il target
    X = df.drop('HL', axis=1)  # Features: h1 a h20
    y = df['HL']  # Target: HL

    # Divisione del dataset in set di addestramento e di validazione
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

    # Creazione del modello MLPRegressor
    mlp_regressor = MLPRegressor(hidden_layer_sizes=(100, 50),
                                 activation='relu',
                                 solver='adam',
                                 alpha=0.0001,
                                 learning_rate_init=0.001,
                                 max_iter=500,
                                 early_stopping=True,
                                 validation_fraction=0.1,
                                 n_iter_no_change=10,
                                 random_state=42,
                                 verbose=True)

    # Addestramento del modello
    mlp_regressor.fit(X_train, y_train)

    # Valutazione del modello sul set di validazione
    y_val_pred = mlp_regressor.predict(X_val)
    mse = mean_squared_error(y_val, y_val_pred)
    r2 = r2_score(y_val, y_val_pred)

    print(f"Errore Quadratico Medio sul set di validazione: {mse}")
    print(f"Coefficiente di determinazione (R²) sul set di validazione: {r2}")

    # Salvare il modello addestrato
    joblib.dump(mlp_regressor, 'mlp_regressor_model_c_64.joblib')

if __name__ == '__main__':
    start_time = datetime.datetime.now()
    print(f"Addestramento iniziato a: {start_time.strftime('%Y-%m-%d %H:%M:%S')}")

    train_regressor()
```

```
    end_time = datetime.datetime.now()
    print(f"Addestramento terminato a: {end_time.strftime('%Y-%m-%d %H:%M:%S')}")
    print(f"Durata totale: {end_time - start_time}")
```

In Dettaglio: Questo programma legge un file CSV contenente dati strutturati, tra cui caratteristiche (features) e un obiettivo da predire (target). Le caratteristiche vengono estratte dal dataset, mentre il target viene isolato. Il dataset viene quindi diviso in due parti: un set di addestramento e un set di validazione per valutare le prestazioni del modello.

Un regressore basato su una rete neurale MLP viene creato con parametri specifici, tra cui le dimensioni dei livelli nascosti, la funzione di attivazione, il metodo di ottimizzazione e altri. Il modello viene quindi addestrato utilizzando il set di addestramento per imparare la relazione tra le caratteristiche e il target.

Dopo l'addestramento, il modello effettua previsioni sul set di validazione e calcola due metriche di valutazione principali: l'Errore Quadratico Medio (MSE) e il Coefficiente di Determinazione (R^2). Queste metriche forniscono informazioni sulle prestazioni del modello nel predire il target in base ai dati di input.

Infine, il modello addestrato viene salvato su disco utilizzando la libreria "joblib", e vengono registrati l'orario di inizio e di fine dell'addestramento, insieme alla durata totale dell'operazione.

ChessGame - Chess

Il gioco degli scacchi, rinomato e antico, è uno degli esempi più pregevoli di strategia tra i giochi da tavolo. La partita si dispiega su una scacchiera composta da 64 caselle, disposte in un alternarsi di colori chiari e scuri (tipicamente bianche e nere). La scacchiera viene posizionata tra i contendenti in modo che la casella situata in basso a destra sia di colore chiaro.

Pezzi Ogni giocatore inizia con 16 pezzi:

- 1 Re: Si muove di una casella in qualsiasi direzione.
- 1 Regina (o Dama): Si muove di qualsiasi numero di caselle in linea retta, sia in orizzontale, verticale, che diagonale.
- 2 Torri: Si muovono in linea retta, ma solo in orizzontale o verticale.
- 2 Cavalli: Si muovono in una forma a "L", ovvero due caselle in una direzione e una perpendicolare a quella.
- 2 Alfieri: Si muovono di qualsiasi numero di caselle, ma solo in diagonale.
- 8 Pedoni: Si muovono in avanti di una casella alla volta, con l'eccezione del primo movi

OBBIETTIVO

L'obiettivo principale è mettere in "scacco matto" il re avversario, creando una condizione in cui il monarca è minacciato e non può evitare la cattura. Il gioco può finire in pareggio, o "patta", in diverse circostanze, come quando nessuno dei giocatori ha sufficienti pezzi per dare scacco matto, o se si verifica una posizione ripetuta tre volte.

MODALITÀ DI GIOCO

Due agenti si sfidano spostando i pezzi sulla scacchiera, facendo un turno alla volta.

CLASSE GIOCO

All'interno del progetto per questo gioco troviamo una classe chiamata **ChessGame** che ha il compito di fornire i metodi per ottenere i vicini di un dato stato, che sono i possibili stati che possono essere raggiunti effettuando mosse valide dallo stato corrente.

Di seguito possiamo vedere la funzione helper **neighbors()** presente nella classe **ChessGame**:

```
def neighbors(self, state: StateChessGame):
    """
    Generates all possible next states (neighbors) from a given state.

    :param state: The current state of the chess game from which to compute neighbors.
    :return: A list of StateChessGame objects representing possible next states.
    """
    neighbors = []

    # Iterate through all legal moves from the current state.
    for legal_move in state.game_board.legal_moves:
        # Copy the current game board and make the legal move.
        new_game_board = state.game_board.copy()
        new_game_board.push(legal_move)

        # Create a new StateChessGame object for the resulting game state.
        neighbor = StateChessGame(game_board=new_game_board, state_parent=state, move=legal_move)
        neighbors.append(neighbor)
    return neighbors
```

La funzione **neighbors()** restituisce gli stati adiacenti a quello fornito, rappresentando tutte le configurazioni possibili dei pezzi ottenibili mediante mosse legali, sfruttando la libreria "chess".

STATES

Lo stato costituisce una chiave essenziale nel contesto del gioco o del problema, fungendo da fotografia istantanea della sua configurazione in un dato momento. All'interno dello stato, troviamo la rappresentazione dettagliata del tavolo da gioco o del contesto problematico, un riferimento al suo stato precedente o "state_parent", nonché una serie di parametri e valori numerici che forniscono una valutazione qualitativa e quantitativa di tale stato, aiutando nella sua interpretazione e nelle decisioni successive.

StateChessGame - STATE Chess

```
def __init__(self, game_board=None, state_parent=None, move=None):
    """
    Initializes a new game state.

    :param game_board: The current chess board configuration. If None, initializes a new chess board.
    :param state_parent: The parent state from which this state is derived.
    :param move: The move that led to this state.
    """
    self.game_board = game_board  # The current chess board (chess.Board object).
    self.parent_state = state_parent  # The parent state from which this state is derived.
    self.move = move  # The move that led to this state.
    self.h = None  # General heuristic value for the state.
    self.h0 = None  # Heuristic value used for h0 cutoff.
    self.hl = None  # Heuristic value used for hl cutoff.
    self.hr = None  # Heuristic value used for nonlinear regressor cutoff.

    # If no game board is provided, initialize a new chess board.
    if self.game_board is None:
        self.game_board = chess.Board()
```

Per maggiori dettagli vedere il file *AlessandroMattei_ChessGame.AIhw2.ipynb*

Chess Game Report

Vengono riportati i risultati dei test effettuati. (sono stati eseguiti 102 partite/test)

I test sono stati effettuati su un mini-pc con le seguenti caratteristiche:

- CPU Intel i7-12650H limitato in potenza a 50 Wat (Intel setta il limite di potenza a 125wat ma il pc in questione non supporta tale potenza)
- Ram 35gb DDR4 3600Mhz

In [1]:
```
import pandas as pd

df_minmax_normal = pd.read_csv('csv/min_max_alpha_beta_pruning_games/result.csv')
df_minmax_h0_cut = pd.read_csv('csv/min_max_alpha_beta_pruning_h0_cut_games/result.csv')
df_minmax_hl_cut = pd.read_csv('csv/min_max_alpha_beta_pruning_hl_cut_games/result.csv')
df_minmax_hr_cut = pd.read_csv('csv/min_max_alpha_beta_pruning_hr_cut_games/result.csv')
df_minmax_h0_cut_vs_hl = pd.read_csv('csv/min_max_alpha_beta_pruning_h0_vs_hl_cut_games/result.csv')
df_minmax_h0_cut_vs_hr = pd.read_csv('csv/min_max_alpha_beta_pruning_h0_vs_hr_cut_games/result.csv')
df_minmax_h0_cut_vs_normal = pd.read_csv('csv/min_max_alpha_beta_pruning_h0_vs_normal_games/result.csv')
df_minmax_hr_cut_vs_hl_cut = pd.read_csv('csv/min_max_alpha_beta_pruning_hr_vs_hl_cut_games/result.csv')
df_minmax_hr_cut_vs_normal = pd.read_csv('csv/min_max_alpha_beta_pruning_hr_vs_normal_games/result.csv')
df_minmax_normal_vs_hl_cut = pd.read_csv('csv/min_max_alpha_beta_pruning_vs_hl_cut_games/result.csv')
```

MinMax Alpha-Beta Pruning

Sono riportati i test effettuati con l'algoritmo base MinMax Alpha-Beta Pruning.

Dove Agent 1 ed Agent 2 si sfidano usando lo stesso algoritmo ed euristica ma con profondità diverse

In [2]:
```
df_minmax_normal.drop(columns=['Title']).head().transpose()
```

Out[2]:

	0	1	2	3
Algorithm Agent 1	MinMax Alpha Beta Pruning	MinMax Alpha Beta Pruning	MinMax Alpha Beta Pruning	MinMax Alpha Beta Pruning
Algorithm Agent 2	MinMax Alpha Beta Pruning	MinMax Alpha Beta Pruning	MinMax Alpha Beta Pruning	MinMax Alpha Beta Pruning
Heuristic Agent 1	HardBoardEvaluationChessGame	HardBoardEvaluationChessGame	HardBoardEvaluationChessGame	HardBoardEvaluationChessGame
Heuristic Agent 2	HardBoardEvaluationChessGame	HardBoardEvaluationChessGame	HardBoardEvaluationChessGame	HardBoardEvaluationChessGame
Max Depth Agent 1	3	3	4	4
Max Depth Agent 2	3	4	3	4
OUTCOME	CHECKMATE	CHECKMATE	FIVEFOLD_REPETITION	CHECKMATE
Winner	Black	Black	NaN	White
Total Time	130725.12ms	1309721.65ms	1105320.51ms	3404402.08ms
AVG Time Agent 1	1213.28ms	883.14ms	19000.20ms	14242.34ms
AVG Time Agent 2	1899.17ms	11957.22ms	1496.37ms	6946.11ms
Number of Moves Agent 1	42	102	54	161
Number of Moves Agent 2	42	102	53	160
States Evaluated Agent 1	182614	250043	4663763	6109260
States Evaluated Agent 2	285700	4011648	312121	3936330
Pruning carried out Agent 1	19283	28829	242417	676701
Pruning carried out Agent 2	19845	376326	28087	119565

MinMax Alpha-Beta Pruning H0 CutOff

Sono riportati i test effettuati con l'algoritmo base MinMax Alpha-Beta Pruning con taglio tramite valutazione statica H0, si notino i tempi di esecuzione più basi rispetto alla versione classica dell'algoritmo (se confrontiamo con le stesse profondità con la versione classica).

Dove abbiamo Agent 1 ed Agent 2 si sfidano usando lo stesso algoritmo ed euristica ma con profondità diverse e diversi parametri di taglio.

Vediamo a profondità da 3 a 4 per entrambi gli Agenti, cosi da poterli confrontarli con il classico MinMax Alpha-Beta Pruning.

In [3]:
```
df_minmax_h0_cut.drop(
    columns=['Title', 'Algorithm Agent 1', 'Algorithm Agent 2', 'Heuristic Agent 1', 'Heuristic Agent 2']).head(
    8).transpose()
```

Out[3]:

	0	1	2	3	4	5	6	7
Max Depth Agent 1	3	3	4	4	3	3	4	4
Number CutOff H0 Agent 1	5	5	5	5	3	3	3	3
Max Depth Agent 2	3	4	3	4	3	4	3	4
Number CutOff H0 Agent 2	5	5	5	5	5	5	5	5
OUTCOME	CHECKMATE	SEVENTYFIVE_MOVES	CHECKMATE	FIVEFOLD_REPETITION	CHECKMATE	CHECKMATE	CHECKMATE	FIVEFOLD_REPETITION
Winner	Black	NaN	Black	NaN	Black	Black	Black	NaN
Total Time	21242.10ms	479910.02ms	132536.65ms	228288.65ms	15483.22ms	85548.29ms	38063.41ms	99226.66ms
AVG Time Agent 1	252.97ms	317.58ms	1813.11ms	934.12ms	101.52ms	98.56ms	450.65ms	388.48ms
AVG Time Agent 2	432.12ms	1281.98ms	641.14ms	1433.94ms	397.80ms	831.22ms	525.26ms	929.02ms
Number of Moves Agent 1	31	300	54	97	31	92	39	76
Number of Moves Agent 2	31	300	54	96	31	92	39	75
States Evaluated Agent 1	2350	21310	15458	23539	840	2628	2422	4850
States Evaluated Agent 2	2365	83983	4855	34783	2325	27003	3066	23927
Pruning carried out Agent 1	511	2661	2221	3852	142	358	420	719
Pruning carried out Agent 2	536	10550	773	4233	500	3925	630	2916
States evaluated H0 Agent 1	18649	81330	158865	141488	7692	18513	34699	52469
States evaluated H0 Agent 2	31818	317249	55588	204792	29660	148940	40879	120305
Pruning H0 carried out Agent 1	14383	49886	135343	103353	6610	15197	31516	46313
Pruning H0 carried out Agent 2	27424	198485	47763	155726	25460	108717	35427	86610

Vediamo a profondità maggiore di 4 per entrambi gli Agenti

In [4]:
```
df_minmax_h0_cut.drop(
    columns=['Title', 'Algorithm Agent 1', 'Algorithm Agent 2', 'Heuristic Agent 1', 'Heuristic Agent 2']).iloc[
8:20].transpose()
```

Out[4]:

	8	9	10	11	12	13	14	15	16	17	18	19
Max Depth Agent 1	5	6	5	6	7	8	9	8	9	9	10	10
Number CutOff H0 Agent 1	5	5	5	5	5	3	3	3	3	3	3	3
Max Depth Agent 2	5	5	6	6	7	8	8	9	9	10	9	10
Number CutOff H0 Agent 2	5	5	5	5	5	3	3	3	3	3	3	3
OUTCOME	CHECKMATE	FIVEFOLD_REPETITION	CHECKMATE	CHECKMATE	CHECKMATE	SEVENTYFIVE_MOVES	CHECKMATE	CHECKMATE	CHECKMATE	CHECKMATE	CHECKMATE	SEVENTYFIVE_MOVES
Winner	Black	NaN	Black	Black	Black	NaN	Black	Black	Black	Black	Black	NaN
Total Time	280443.45ms	1365808.27ms	1039865.00ms	433161.97ms	934196.32ms	1592359.21ms	852883.62ms	1038700.11ms	673887.36ms	596961.34ms	736373.25ms	1700247.30ms
AVG Time Agent 1	2791.97ms	5590.87ms	3219.24ms	8165.26ms	11911.77ms	3117.17ms	12628.93ms	6145.47ms	12466.97ms	9632.21ms	22616.90ms	7686.70ms
AVG Time Agent 2	4587.96ms	2317.25ms	10112.12ms	9161.07ms	18223.53ms	2902.54ms	6323.92ms	13452.54ms	16832.29ms	23532.20ms	12448.50ms	6970.43ms
Number of Moves Agent 1	38	173	78	25	31	265	45	53	23	18	21	116
Number of Moves Agent 2	38	172	78	25	31	264	45	53	23	18	21	116
States Evaluated Agent 1	22521	267648	49268	52089	111781	215441	103912	61805	62569	38770	109896	357533
States Evaluated Agent 2	32235	107031	158864	70779	149704	208335	55263	131150	75869	93565	63783	333774
Pruning carried out Agent 1	6029	54916	12909	10826	33500	53167	30102	15518	18021	11055	28183	95795
Pruning carried out Agent 2	6559	22458	37746	14492	34936	53580	14273	35503	20067	26701	16796	92114
States evaluated H0 Agent 1	201216	2050291	514334	542868	1104577	1556021	1357226	884997	795457	492100	1398582	3301519
States evaluated H0 Agent 2	346991	858434	1605858	596696	1722537	1448289	696348	1913666	1051305	1194701	797128	3076692
Pruning H0 carried out Agent 1	157102	1589502	419160	451037	874448	1246376	1200159	795721	701259	433880	1239188	2772723
Pruning H0 carried out Agent 2	291977	669144	1316516	475188	1448785	1146378	616025	1719287	940418	1054968	703726	2580965

MinMax Alpha-Beta Pruning Hl CutOff

Sono riportati i test effettuati con l'algoritmo base MinMax Alpha-Beta Pruning con taglio con valutazione Hl, si notino i tempi di esecuzione più basi rispetto alla versione classica dell'algoritmo (se confrontiamo con le stesse profondità con la versione classica). Dove abbiamo Agent 1 ed Agent 2 si sfidano usando lo stesso algoritmo ed euristica ma con profondità diverse e diversi parametri di taglio.

Vediamo a profondità da 3 a 4 per entrambi gli Agenti e con profondità l da 2 a 3, cosi da poterli confrontarli con il classico MinMax Alpha-Beta Pruning.

In [5]:
```
df_minmax_hl_cut.drop(
    columns=['Title', 'Algorithm Agent 1', 'Algorithm Agent 2', 'Heuristic Agent 1', 'Heuristic Agent 2']).head(
    18).transpose()
```

Out[5]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Max Depth Agent 1	3	3	4	4	3	3	4	4	3	3	4	4	3	3	4	4	4	4
Number CutOff Agent 1	5	5	5	5	5	5	5	5	3	3	3	3	3	3	3	3	5	5
Number of I Agent 1	2	2	2	2	3	3	3	3	2	2	2	2	3	3	3	3	2	3
Max Depth Agent 2	3	4	3	4	3	4	3	4	3	4	3	4	3	4	3	4	4	4
Number CutOff Agent 2	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
Number of I Agent 2	2	2	2	2	3	3	3	3	2	2	2	2	3	3	3	3	3	3
OUTCOME	CHECKMATE	SEVENTYFIVE_MOVES	CHECKMATE	FIVEFOLD_REPETITION	CHECKMATE	CHECKMATE	CHECKMATE	CHECKMATE	CHECKMATE	CHECKMATE	CHECKMATE	CHECKMATE	CHECKMATE	CHECKMATE	CHECKMATE	FIVEFOLD_REPETITION	CHECKMATE	CHECKMATE
Winner	Black	NaN	Black	NaN	Black	Black	Black	Black	White	White	Black	White	Black	Black	Black	NaN	White	Black
Total Time	26616.84ms	334068.54ms	59083.67ms	207182.49ms	4586132.50ms	1258624.10ms	1109962.76ms	1140042.33ms	48299.97ms	53666.30ms	30131.44ms	215182.08ms	378455.77ms	142967.78ms	1413181.54ms	437384.41ms	169348.18ms	1143032.63ms
AVG Time Agent 1	338.93ms	466.52ms	1023.45ms	1772.09ms	1025.19ms	1194.32ms	1914.36ms	6733.92ms	426.46ms	378.18ms	598.02ms	601.73ms	4241.87ms	2922.70ms	946.45ms	2682.93ms	1113.27ms	7155.60ms
AVG Time Agent 2	467.49ms	1348.81ms	383.21ms	1680.80ms	10948.64ms	5066.60ms	4021.05ms	11653.67ms	459.96ms	1524.83ms	560.71ms	1705.45ms	10313.97ms	5487.05ms	6530.39ms	5220.57ms	4313.60ms	11280.24ms
Number of Moves Agent 1	33	184	42	60	383	201	187	62	55	29	26	94	26	17	189	56	32	62
Number of Moves Agent 2	33	184	42	60	383	201	187	62	54	28	26	93	26	17	189	55	31	62
States Evaluated Agent 1	2920	15720	13368	19722	4517	4868	32146	18175	1596	835	1846	7115	754	469	4713	3760	8301	18175
States Evaluated Agent 2	2675	63037	3687	23775	11645	17999	11424	17467	3861	10243	2174	33538	1992	4906	7568	14691	11415	17467
Pruning carried out Agent 1	346	1526	1781	2930	1496	985	4796	2356	191	94	283	916	87	67	750	593	1290	2356
Pruning carried out Agent 2	464	6573	557	3007	4710	3458	1847	2547	670	1230	359	3929	428	818	1232	2082	1272	2547
States evaluated H0 Agent 1	16711	56883	90980	154544	77297	39692	193724	148964	19082	10664	21490	56289	7041	4140	52091	37759	71368	148964
States evaluated H0 Agent 2	22831	227213	24073	161020	260490	147983	77350	122907	28268	100954	17756	241845	26612	37776	77471	95188	122820	122907
Pruning H0 carried out Agent 1	12628	36025	71587	124654	68101	31485	145628	122421	17281	9739	19226	47856	6201	3586	46365	33102	58519	122421
Pruning H0 carried	18465	144387	18383	127609	231220	117801	59449	96166	22107	86777	14298	195971	23064	30006	65761	73482	107374	96166

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
out Agent 2																		
States evaluated HI Agent 1	13635	38057	22050	31998	262020	243988	266944	295933	39287	21250	16505	44153	142058	80135	148954	177686	28543	295933
States evaluated HI Agent 2	14656	36294	20942	27079	2697872	645620	496529	644594	31078	18748	16718	39160	365241	126271	828620	363910	126959	644594
Pruning HI carried out Agent 1	365	1209	665	1110	55610	34897	31620	35334	1914	962	654	2176	18271	8762	26351	19749	1226	35334
Pruning HI carried out Agent 2	766	2014	689	767	106826	42599	34587	37217	593	355	458	831	19476	9260	35133	20927	20721	37217

Vediamo a profondità maggiore di 4 per entrambi gli Agenti e con profondità I da 2 a 3

In [6]:
```
df_minmax_hl_cut.drop(
    columns=['Title', 'Algorithm Agent 1', 'Algorithm Agent 2', 'Heuristic Agent 1', 'Heuristic Agent 2']).iloc[
18:].transpose()
```

Out[6]:

	18	19	20	21	22	23	24	25	26	27	28	29
Max Depth Agent 1	5	6	5	6	7	8	9	8	9	9	10	10
Number CutOff Agent 1	5	5	5	5	5	3	3	3	3	3	3	3
Number of I Agent 1	2	2	2	2	2	2	2	2	2	2	2	2
Max Depth Agent 2	5	5	6	6	7	8	8	9	9	10	9	10
Number CutOff Agent 2	5	5	5	5	5	3	3	3	3	3	3	3
Number of I Agent 2	2	2	2	2	2	2	2	2	2	2	2	2
OUTCOME	CHECKMATE	CHECKMATE	CHECKMATE	CHECKMATE	CHECKMATE	CHECKMATE	CHECKMATE	SEVENTYFIVE_MOVES	CHECKMATE	CHECKMATE	SEVENTYFIVE_MOVES	CHECKMATE
Winner	Black	White	White	White	Black	Black	White	NaN	Black	Black	NaN	White
Total Time	254844.40ms	507903.08ms	1294699.54ms	1805126.84ms	2493912.13ms	1343823.05ms	2856083.16ms	4833464.86ms	1119994.20ms	3075336.30ms	3508353.95ms	4227955.24ms
AVG Time Agent 1	5328.36ms	7705.56ms	6812.01ms	12626.91ms	34141.44ms	10801.13ms	17651.79ms	8244.42ms	17556.54ms	17599.17ms	13537.07ms	20199.39ms
AVG Time Agent 2	5290.09ms	2102.09ms	22458.08ms	20567.40ms	21278.37ms	10195.67ms	8629.81ms	16373.94ms	18572.24ms	38316.00ms	5878.41ms	21878.07ms
Number of Moves Agent 1	24	52	45	55	45	64	109	197	31	55	181	101
Number of Moves Agent 2	24	51	44	54	45	64	108	196	31	55	180	100
States Evaluated Agent 1	21571	115812	51284	143416	367592	92804	348279	299619	90891	191590	666741	523461
States Evaluated Agent 2	24140	36620	169229	215796	258506	106838	171817	578332	81585	457841	325113	558973
Pruning carried out Agent 1	4996	23250	10100	29365	92158	22476	91895	70340	25930	53083	181898	146299
Pruning carried out Agent 2	5700	7478	32937	43247	71351	25900	40513	154987	23183	126200	90454	146019
States evaluated H0 Agent 1	232967	793773	555793	1345614	4007867	1152781	3698823	3114624	1232041	2614068	5926868	5859323
States evaluated H0 Agent 2	243064	246628	1793652	2228881	2474150	1097513	1764896	6245151	1220419	6021071	2767842	6525733
Pruning H0 carried out Agent 1	193228	601544	468689	1098134	3312270	1021060	3195523	2696885	1096005	2330919	4948513	5080833
Pruning H0 carried out Agent 2	198502	184119	1512497	1867295	1953361	947138	1526104	5400478	1097147	5348263	2288049	5722693
States evaluated HI Agent 1	14968	18388	22638	32582	24988	26064	37503	74020	15079	30203	48914	43627
States evaluated HI Agent 2	14704	15453	22837	29912	25552	25859	30219	52701	17272	29874	43200	36196
Pruning HI carried out Agent 1	487	907	1190	1506	1118	893	2569	2929	526	1206	3689	2746
Pruning HI carried out Agent 2	440	409	432	634	552	1296	784	2396	672	1048	1022	774

MinMax Alpha-Beta Pruning Hr CutOff (MPLRegressor)

Sono riportati i test effettuati con l'algoritmo MinMax Alpha-Beta Pruning con taglio con valutazione Hr (tramite il regressore). Si Noti che a basse profondità abbiamo molte partite che finiscono in pareggio.

Dove abbiamo Agent 1 ed Agent 2 si sfidano usando lo stesso algoritmo ed euristica ma con profondità diverse e diversi parametri di taglio.

Vediamo a profondità da 3 a 4 per entrambi gli Agenti, cosi da poterli confrontarli con il classico MinMax Alpha-Beta Pruning.

In [7]:
```
df_minmax_hr_cut.drop(
    columns=['Title', 'Algorithm Agent 1', 'Algorithm Agent 2', 'Heuristic Agent 1', 'Heuristic Agent 2']).head(
    5).transpose()
```

Out[7]:

	0	1	2	3	4
Max Depth Agent 1	3	3	4	4	5
Number CutOff Hr Agent 1	5	5	5	5	3
Max Depth Agent 2	3	4	3	4	5
Number CutOff Hr Agent 2	5	5	5	5	3
OUTCOME	INSUFFICIENT_MATERIAL	FIVEFOLD_REPETITION	CHECKMATE	FIVEFOLD_REPETITION	FIVEFOLD_REPETITION
Winner	NaN	NaN	White	NaN	NaN
Total Time	501344.70ms	431826.08ms	366489.69ms	701824.91ms	372908.43ms
AVG Time Agent 1	1046.43ms	670.20ms	3013.14ms	1408.16ms	1296.12ms
AVG Time Agent 2	445.57ms	1647.75ms	772.92ms	1327.77ms	2051.68ms
Number of Moves Agent 1	336	187	97	257	112
Number of Moves Agent 2	336	186	96	256	111
States Evaluated Agent 1	27452	15447	31351	73502	15966
States Evaluated Agent 2	30110	41884	8106	61096	16889
Pruning carried out Agent 1	4406	2503	3670	10019	3008
Pruning carried out Agent 2	3728	5677	1003	7618	3013
States evaluated Hr Agent 1	205456	87042	203615	284831	103984
States evaluated Hr Agent 2	90636	216164	52838	272582	162239
Pruning Hr carried out Agent 1	162018	63793	161809	183048	83424
Pruning Hr carried out Agent 2	52177	157592	41694	191290	140882

Vediamo i restanti due test:

- Il primo a profondità 6 e 6
- Il secondo a profondità 10 e 10

Altri test purtroppo per mancanza ti tempo no sono stati effettuati. Da notare che ha profondità 10 e 10 abbiamo un tempo totale di 6279055.89ms ~ 1.74 ore contro i 4227955.24ms ~ 1.17 ore della versione Hl contro i 1700247.30ms ~ 28.33 Minuti della versione H0.

Hr risulta essere il peggiore per quanto riguarda i tempi di esecuzione. E più avanti ci accorgermemo che sarà il pegiore pure nel scegliere le mosse.

In [8]:
```
df_minmax_hr_cut.drop(
    columns=['Title', 'Algorithm Agent 1', 'Algorithm Agent 2', 'Heuristic Agent 1', 'Heuristic Agent 2']).iloc[
5:].transpose()
```

Out[8]:

	5	6
Max Depth Agent 1	6	10
Number CutOff Hr Agent 1	3	3
Max Depth Agent 2	6	10
Number CutOff Hr Agent 2	3	3
OUTCOME	FIVEFOLD_REPETITION	CHECKMATE
Winner	NaN	White
Total Time	817093.23ms	6279055.89ms
AVG Time Agent 1	2819.90ms	48008.13ms
AVG Time Agent 2	3613.80ms	54927.21ms
Number of Moves Agent 1	127	61
Number of Moves Agent 2	127	61
States Evaluated Agent 1	44091	345956
States Evaluated Agent 2	43411	301017
Pruning carried out Agent 1	8277	89961
Pruning carried out Agent 2	7914	73561
States evaluated Hr Agent 1	336935	3824257
States evaluated Hr Agent 2	430877	4416521
Pruning Hr carried out Agent 1	280603	3349983
Pruning Hr carried out Agent 2	375731	4008184

MinMax Alpha-Beta Pruning H0 CutOff Vs MinMax Alpha-Beta Pruning Hl CutOff

Sono riportati i test effettuati con l'algoritmo MinMax Alpha-Beta Pruning con taglio e valutazione H0 contro l'algoritmo MinMax Alpha-Beta Pruning con taglio e valutazione Hl.

In [9]:
```
df_minmax_h0_cut_vs_hl.drop(columns=['Title', 'Heuristic Agent 1', 'Heuristic Agent 2']).head(5).transpose()
```

Out[9]:

	0	1	2	3	4
Algorithm Agent 1	MinMax Alpha Beta Pruning H0 CutOff	MinMax Alpha Beta Pruning H0 CutOff	MinMax Alpha Beta Pruning H0 CutOff	MinMax Alpha Beta Pruning H0 CutOff	MinMax Alpha Beta Pruning H0 CutOff
Algorithm Agent 2	MinMax Alpha Beta Pruning Hl CutOff	MinMax Alpha Beta Pruning Hl CutOff	MinMax Alpha Beta Pruning Hl CutOff	MinMax Alpha Beta Pruning Hl CutOff	MinMax Alpha Beta Pruning Hl CutOff
Max Depth Agent 1	3	3	4	4	5
Number CutOff Agent 1	5	5	5	5	5
Max Depth Agent 2	3	4	3	4	5
Number CutOff Agent 2	5	5	5	5	5
Number of I Agent 2	3	3	3	3	3
OUTCOME	CHECKMATE	SEVENTYFIVE_MOVES	CHECKMATE	CHECKMATE	CHECKMATE
Winner	Black	NaN	Black	Black	Black
Total Time	95360.93ms	714725.30ms	259116.41ms	409949.13ms	524009.38ms
AVG Time Agent 1	222.48ms	375.28ms	994.12ms	925.44ms	2357.57ms
AVG Time Agent 2	4796.41ms	3160.77ms	7364.39ms	6022.81ms	13521.41ms
Number of Moves Agent 1	19	203	31	59	33
Number of Moves Agent 2	19	202	31	59	33
States Evaluated Agent 1	1483	13777	6764	14222	18089
States Evaluated Agent 2	1474	49175	2283	15643	25693
Pruning carried out Agent 1	256	1831	1225	1919	4818
Pruning carried out Agent 2	330	5770	459	2284	5295
States evaluated H0 Agent 1	10424	91865	72736	113068	177996
States evaluated H0 Agent 2	17410	285342	27865	100606	319823
Pruning H0 carried out Agent 1	7954	70522	61426	92101	142118
Pruning H0 carried out Agent 2	14763	217797	23960	77286	274871
States evaluated Hl Agent 2	124640	385798	266399	370148	449458
Pruning Hl carried out Agent 2	11261	47333	18462	29127	26568

In [10]:
```
df_minmax_h0_cut_vs_hl.drop(columns=['Title', 'Heuristic Agent 1', 'Heuristic Agent 2']).iloc[5:10].transpose()
```

Out[10]:

	5	6	7	8	9
Algorithm Agent 1	MinMax Alpha Beta Pruning H0 CutOff	MinMax Alpha Beta Pruning H0 CutOff	MinMax Alpha Beta Pruning H0 CutOff	MinMax Alpha Beta Pruning H0 CutOff	MinMax Alpha Beta Pruning H0 CutOff
Algorithm Agent 2	MinMax Alpha Beta Pruning Hl CutOff	MinMax Alpha Beta Pruning Hl CutOff	MinMax Alpha Beta Pruning Hl CutOff	MinMax Alpha Beta Pruning Hl CutOff	MinMax Alpha Beta Pruning Hl CutOff
Max Depth Agent 1	6	5	6	7	8
Number CutOff Agent 1	5	5	5	5	3
Max Depth Agent 2	5	6	6	7	8
Number CutOff Agent 2	5	5	5	5	3
Number of l Agent 2	3	3	3	2	2
OUTCOME	CHECKMATE	CHECKMATE	CHECKMATE	CHECKMATE	FIVEFOLD_REPETITION
Winner	Black	Black	Black	Black	NaN
Total Time	464246.86ms	414618.98ms	1045058.47ms	1068613.62ms	713133.09ms
AVG Time Agent 1	9959.99ms	2145.72ms	2264.87ms	16908.09ms	7275.24ms
AVG Time Agent 2	12146.91ms	13801.06ms	4434.07ms	15474.11ms	8932.24ms
Number of Moves Agent 1	21	26	156	33	44
Number of Moves Agent 2	21	26	156	33	44
States Evaluated Agent 1	48762	14371	107871	203735	55755
States Evaluated Agent 2	22091	49934	141840	175735	72428
Pruning carried out Agent 1	9635	3541	21445	55771	13810
Pruning carried out Agent 2	4256	10706	29922	44464	18245
States evaluated H0 Agent 1	529913	132797	949247	1902516	697904
States evaluated H0 Agent 2	263997	441109	967881	1872862	824035
Pruning H0 carried out Agent 1	447319	105122	764094	1499492	617838
Pruning H0 carried out Agent 2	226407	352812	723184	1530924	721660
States evaluated Hl Agent 2	280669	216425	474921	19976	25595
Pruning Hl carried out Agent 2	16143	16284	31827	798	892

In [11]:
```
df_minmax_h0_cut_vs_hl.drop(columns=['Title', 'Heuristic Agent 1', 'Heuristic Agent 2']).iloc[10:].transpose()
```

Out[11]:

	10	11	12	13	14	15
Algorithm Agent 1	MinMax Alpha Beta Pruning H0 CutOff	MinMax Alpha Beta Pruning H0 CutOff	MinMax Alpha Beta Pruning H0 CutOff	MinMax Alpha Beta Pruning H0 CutOff	MinMax Alpha Beta Pruning H0 CutOff	MinMax Alpha Beta Pruning H0 CutOff
Algorithm Agent 2	MinMax Alpha Beta Pruning Hl CutOff	MinMax Alpha Beta Pruning Hl CutOff	MinMax Alpha Beta Pruning Hl CutOff	MinMax Alpha Beta Pruning Hl CutOff	MinMax Alpha Beta Pruning Hl CutOff	MinMax Alpha Beta Pruning Hl CutOff
Max Depth Agent 1	9	8	9	9	10	10
Number CutOff Agent 1	3	3	3	3	3	3
Max Depth Agent 2	8	9	9	10	9	10
Number CutOff Agent 2	3	3	3	3	3	3
Number of I Agent 2	2	2	2	2	2	2
OUTCOME	SEVENTYFIVE_MOVES	CHECKMATE	CHECKMATE	CHECKMATE	CHECKMATE	FIVEFOLD_REPETITION
Winner	NaN	Black	Black	Black	Black	NaN
Total Time	2302339.93ms	231860.61ms	698797.37ms	1006601.33ms	845928.87ms	2603557.81ms
AVG Time Agent 1	4223.36ms	4396.69ms	7961.92ms	7431.49ms	17057.14ms	6226.41ms
AVG Time Agent 2	3237.96ms	13438.58ms	13875.40ms	20529.54ms	12112.75ms	6631.56ms
Number of Moves Agent 1	309	13	32	36	29	203
Number of Moves Agent 2	308	13	32	36	29	202
States Evaluated Agent 1	400600	10811	52025	67210	104275	519031
States Evaluated Agent 2	336338	33118	77072	185626	74680	591049
Pruning carried out Agent 1	117928	2641	16260	20677	26795	143424
Pruning carried out Agent 2	84576	8351	20168	54816	19573	166332
States evaluated H0 Agent 1	2666890	139247	661108	859254	1408235	4042817
States evaluated H0 Agent 2	2011525	443234	1154224	2340115	1028538	4375924
Pruning H0 carried out Agent 1	2069775	123830	580171	755804	1256295	3265353
Pruning H0 carried out Agent 2	1541780	395955	1041957	2059889	918918	3495986
States evaluated Hl Agent 2	51434	6917	19697	23455	16229	32451
Pruning Hl carried out Agent 2	3589	330	960	972	623	1248

Come possiamo notare l'Agente che usa Hl vince più spesso dell'agente che usa H0 a discapito però del tempo di esecuzione per mossa che per l'Agente H0 è migliore.

MinMax Alpha-Beta Pruning H0 CutOff Vs MinMax Alpha-Beta Pruning Hr CutOff (MPLRegressor)

Sono riportati i test effettuati con l'algoritmo MinMax Alpha-Beta Pruning con taglio e valutazione Hl contro l'algoritmo MinMax Alpha-Beta Pruning con taglio Hr.

In [12]:
```
df_minmax_h0_cut_vs_hr.drop(columns=['Title', 'Heuristic Agent 1', 'Heuristic Agent 2']).head(6).transpose()
```

Out[12]:

	0	1	2	3	4	5
Algorithm Agent 1	MinMax Alpha Beta Pruning H0 CutOff	MinMax Alpha Beta Pruning H0 CutOff	MinMax Alpha Beta Pruning H0 CutOff	MinMax Alpha Beta Pruning H0 CutOff	MinMax Alpha Beta Pruning H0 CutOff	MinMax Alpha Beta Pruning H0 CutOff
Algorithm Agent 2	MinMax Alpha Beta Pruning Hr CutOff	MinMax Alpha Beta Pruning Hr CutOff	MinMax Alpha Beta Pruning Hr CutOff	MinMax Alpha Beta Pruning Hr CutOff	MinMax Alpha Beta Pruning Hr CutOff	MinMax Alpha Beta Pruning Hr CutOff
Max Depth Agent 1	3	3	4	4	5	6
Number CutOff Agent 1	5	5	5	5	3	3
Max Depth Agent 2	3	4	3	4	5	6
Number CutOff Agent 2	5	5	5	5	3	3
OUTCOME	CHECKMATE	FIVEFOLD_REPETITION	FIVEFOLD_REPETITION	CHECKMATE	FIVEFOLD_REPETITION	FIVEFOLD_REPETITION
Winner	White	NaN	NaN	Black	NaN	NaN
Total Time	48805.98ms	38457.12ms	246826.98ms	54689.18ms	214169.30ms	41715.57ms
AVG Time Agent 1	122.16ms	70.35ms	641.54ms	305.16ms	206.28ms	317.00ms
AVG Time Agent 2	936.08ms	2676.51ms	640.63ms	2573.22ms	1140.61ms	3475.33ms
Number of Moves Agent 1	47	14	193	19	159	11
Number of Moves Agent 2	46	14	192	19	159	11
States Evaluated Agent 1	4236	967	52457	4840	18630	2372
States Evaluated Agent 2	4472	6314	19540	7313	24720	3760
Pruning carried out Agent 1	606	229	7088	828	4146	501
Pruning carried out Agent 2	538	500	2206	841	4257	633
States evaluated H0 Agent 1	29960	7195	365303	43328	139224	27416
States evaluated Hr Agent 2	40761	35108	122470	49655	192809	36777
Pruning H0 carried out Agent 1	23505	5315	288268	35397	113532	24104
Pruning Hr carried out Agent 2	34374	27465	96011	39736	161345	32127

Comparando l'Agnete che usa H0 con l'Agente che usa Hr notiamo che la stragrande maggioranza delle partite termina in un pareggio e quando c'è un vincitore è sempre l'Agente che usa H0. Probabilmente è dovuto al valore non del tutto corretto che restituisce il regressore.

MinMax Alpha-Beta Pruning H0 CutOff Vs MinMax Alpha-Beta Pruning

Sono riportati i test effettuati con l'algoritmo MinMax Alpha-Beta Pruning con taglio e valutazione H0 contro l'algoritmo base MinMax Alpha-Beta Pruning.

In [13]:
```
df_minmax_h0_cut_vs_normal.drop(columns=['Title', 'Heuristic Agent 1', 'Heuristic Agent 2']).head(6).transpose()
```

Out[13]:

	0	1	2	3	4
Algorithm Agent 1	MinMax Alpha Beta Pruning H0 CutOff	MinMax Alpha Beta Pruning H0 CutOff	MinMax Alpha Beta Pruning H0 CutOff	MinMax Alpha Beta Pruning H0 CutOff	MinMax Alpha Beta Pruning H0 CutOff
Algorithm Agent 2	MinMax Alpha Beta Pruning	MinMax Alpha Beta Pruning	MinMax Alpha Beta Pruning	MinMax Alpha Beta Pruning	MinMax Alpha Beta Pruning
Max Depth Agent 1	3	4	5	6	10
Number CutOff Agent 1	5	5	5	3	3
Max Depth Agent 2	3	3	3	2	2
OUTCOME	CHECKMATE	CHECKMATE	CHECKMATE	CHECKMATE	CHECKMATE
Winner	Black	White	Black	White	Black
Total Time	123630.51ms	63382.18ms	245204.27ms	28181.52ms	643101.30ms
AVG Time Agent 1	122.66ms	365.37ms	751.39ms	585.98ms	5544.33ms
AVG Time Agent 2	1126.11ms	1667.33ms	2517.96ms	140.21ms	47.84ms
Number of Moves Agent 1	99	32	75	39	115
Number of Moves Agent 2	99	31	75	38	115
States Evaluated Agent 1	8323	10052	37074	11053	336855
States Evaluated Agent 2	314978	196855	425620	24564	25776
Pruning carried out Agent 1	1314	1424	8686	2458	87381
Pruning carried out Agent 2	38212	22366	44138	10	0
States evaluated H0 Agent 1	63917	87047	348508	146088	3659395
Pruning H0 carried out Agent 1	50491	71937	280502	130611	3171787

Caso singolare è il confronto tra H0 e il normale MinMax Alpha-Beta Pruning, in questo caso non è detto che aumentando la profondità vinca l'agente che è più profondo. Se prendiamo in considerazione un Agente a profondità 10 che usa il CutOff H0 non sarà meglio del Agente che usa il normale MinMax Alpha-Beta Pruning. Come visto in precedenza il CutOff H0 è più veloce del normale MinMax Alpha-Beta Pruning (a parità di profondità).

MinMax Alpha-Beta Pruning Hr CutOff (MPLRegressor) Vs MinMax Alpha-Beta Pruning Hl CutOff

Sono riportati i test effettuati con l'algoritmo MinMax Alpha-Beta Pruning con taglio Hr (MPLRegressor) contro l'algoritmo MinMax Alpha-Beta Pruning con taglio e valutazione Hl.

In [14]:
```
df_minmax_hr_cut_vs_hl_cut.drop(columns=['Title', 'Heuristic Agent 1', 'Heuristic Agent 2']).head(6).transpose()
```

Out[14]:

	0	1	2	3	4	5
Algorithm Agent 1	MinMax Alpha Beta Pruning Hr CutOff	MinMax Alpha Beta Pruning Hr CutOff	MinMax Alpha Beta Pruning Hr CutOff	MinMax Alpha Beta Pruning Hr CutOff	MinMax Alpha Beta Pruning Hr CutOff	MinMax Alpha Beta Pruning Hr CutOff
Algorithm Agent 2	MinMax Alpha Beta Pruning Hl CutOff	MinMax Alpha Beta Pruning Hl CutOff	MinMax Alpha Beta Pruning Hl CutOff	MinMax Alpha Beta Pruning Hl CutOff	MinMax Alpha Beta Pruning Hl CutOff	MinMax Alpha Beta Pruning Hl CutOff
Max Depth Agent 1	3	3	4	4	5	6
Number CutOff Agent 1	5	5	5	5	3	3
Max Depth Agent 2	3	4	3	4	5	6
Number CutOff Agent 2	5	5	5	5	3	3
Number of I Agent 2	3	3	3	3	2	2
OUTCOME	CHECKMATE	FIVEFOLD_REPETITION	CHECKMATE	CHECKMATE	CHECKMATE	CHECKMATE
Winner	Black	NaN	Black	Black	Black	Black
Total Time	32426.24ms	118036.28ms	224519.78ms	923112.32ms	50952.22ms	149754.71ms
AVG Time Agent 1	764.63ms	486.43ms	1850.09ms	1569.77ms	2285.72ms	4803.70ms
AVG Time Agent 2	2183.12ms	1217.17ms	1225.48ms	2375.08ms	395.81ms	742.73ms
Number of Moves Agent 1	11	70	73	234	19	27
Number of Moves Agent 2	11	69	73	234	19	27
States Evaluated Agent 1	1162	7162	22505	71186	3513	11945
States Evaluated Agent 2	955	26542	6279	73242	3058	11398
Pruning carried out Agent 1	149	885	2527	8704	629	2172
Pruning carried out Agent 2	176	2891	888	9140	645	2299
States evaluated Hr Agent 1	8048	33573	148302	367977	42301	136109
States evaluated H0 Agent 2	10921	129417	50830	433025	40462	128598
Pruning Hr carried out Agent 1	6419	23738	118509	271103	37816	120855
Pruning H0 carried out Agent 2	9331	93761	41441	330391	36284	113539
States evaluated Hl Agent 2	131685	244104	340313	679103	14681	19209
Pruning Hl carried out Agent 2	7508	21194	24827	82953	435	602

Comparando l'Agnete che usa Hr con l'Agente che usa Hl notiamo che la stragrande maggioranza delle partite termina con la vittoria del Agente che usa Hl. Un comportamento simile lo abbiamo visto con il test H0 vs Hr, li quando c'era una vittoria era sempre di H0. Probabilmente è dovuto al valore non del tutto corretto che restituisce il regressore.

MinMax Alpha-Beta Pruning Hr CutOff (MPLRegressor) Vs MinMax Alpha-Beta Pruning

Sono riportati i test effettuati con l'algoritmo MinMax Alpha-Beta Pruning con taglio Hr contro l'algoritmo base MinMax Alpha-Beta Pruning.

In [15]:
```
df_minmax_hr_cut_vs_normal.drop(columns=['Title', 'Heuristic Agent 1', 'Heuristic Agent 2']).head(6).transpose()
```

Out[15]:

	0	1	2	3
Algorithm Agent 1	MinMax Alpha Beta Pruning Hr CutOff	MinMax Alpha Beta Pruning Hr CutOff	MinMax Alpha Beta Pruning Hr CutOff	MinMax Alpha Beta Pruning Hr CutOff
Algorithm Agent 2	MinMax Alpha Beta Pruning	MinMax Alpha Beta Pruning	MinMax Alpha Beta Pruning	MinMax Alpha Beta Pruning
Max Depth Agent 1	3	4	5	6
Number CutOff Agent 1	5	5	5	3
Max Depth Agent 2	3	3	3	2
OUTCOME	CHECKMATE	FIVEFOLD_REPETITION	CHECKMATE	CHECKMATE
Winner	Black	NaN	Black	Black
Total Time	97399.52ms	277061.22ms	303703.65ms	209772.21ms
AVG Time Agent 1	637.39ms	1685.54ms	5956.40ms	4529.93ms
AVG Time Agent 2	3258.59ms	1672.67ms	1636.16ms	131.63ms
Number of Moves Agent 1	25	83	40	45
Number of Moves Agent 2	25	82	40	45
States Evaluated Agent 1	2451	28552	40779	18917
States Evaluated Agent 2	316640	345541	248668	25216
Pruning carried out Agent 1	298	3596	7339	3402
Pruning carried out Agent 2	16491	42314	16066	3
States evaluated Hr Agent 1	16315	150637	262496	208388
Pruning Hr carried out Agent 1	12912	110950	200669	184358

Comparando l'Agnete che usa Hr con l'Agente che usa il normale MinMax Alpha-Beta Pruning notiamo che la stragrande maggioranza delle partite termina con la vittoria del Agente che usa il nomrale MinMax Alpha-Beta Pruning. Un comportamento simile lo abbiamo visto con il test H0 vs Hr, li quando c'era una vittoria era sempre di H0 e con il test Hr vs Hl. Probabilmente è dovuto al valore non del tutto corretto che restituisce il regressore.

MinMax Alpha-Beta Pruning Vs MinMax Alpha-Beta Pruning Hl CutOff

Sono riportati i test effettuati con l'algoritmo base MinMax Alpha-Beta Pruning contro l'algoritmo MinMax Alpha-Beta Pruning con taglio Hr.

In [16]:
```
df_minmax_normal_vs_hl_cut.drop(columns=['Title', 'Heuristic Agent 1', 'Heuristic Agent 2']).head(6).transpose()
```

Out[16]:

	0	1	2	3	4
Algorithm Agent 1	MinMax Alpha Beta Pruning	MinMax Alpha Beta Pruning	MinMax Alpha Beta Pruning	MinMax Alpha Beta Pruning	MinMax Alpha Beta Pruning
Algorithm Agent 2	MinMax Alpha Beta Pruning Hl CutOff	MinMax Alpha Beta Pruning Hl CutOff	MinMax Alpha Beta Pruning Hl CutOff	MinMax Alpha Beta Pruning Hl CutOff	MinMax Alpha Beta Pruning Hl CutOff
Max Depth Agent 1	3	3	3	2	2
Max Depth Agent 2	3	4	5	6	10
Number CutOff Agent 2	5	5	5	3	3
Number of I Agent 2	3	3	3	3	2
OUTCOME	CHECKMATE	CHECKMATE	CHECKMATE	CHECKMATE	CHECKMATE
Winner	Black	Black	Black	White	White
Total Time	223899.14ms	81126.13ms	190409.61ms	89924.82ms	873085.01ms
AVG Time Agent 1	294.23ms	1005.82ms	1511.15ms	189.77ms	77.79ms
AVG Time Agent 2	1427.97ms	1380.19ms	3132.96ms	1998.84ms	11562.26ms
Number of Moves Agent 1	130	34	41	42	76
Number of Moves Agent 2	130	34	41	41	75
States Evaluated Agent 1	139231	126194	203683	35963	31806
States Evaluated Agent 2	5029	9486	27585	14711	523343
Pruning carried out Agent 1	20052	15365	25503	0	12
Pruning carried out Agent 2	962	1439	5551	2827	143328
States evaluated H0 Agent 2	56984	58300	325827	207613	5603600
Pruning H0 carried out Agent 2	48649	43986	278057	188212	4836578
States evaluated Hl Agent 2	663047	170098	409485	201107	23977
Pruning Hl carried out Agent 2	29655	12926	22957	23258	619

Un altro caso singolare è il confronto tra il normale MinMax Alpha-Beta Pruning e Hl, anche in questo caso non è detto che aumentando la profondità vinca l'agente che è più profondo. Se prendiamo in considerazione un Agente a profondità 10 che usa il CutOff Hl non sarà meglio del Agente che usa il normale MinMax Alpha-Beta Pruning. Come visto anche con il caso H0 vs normale MinMax Alpha-Beta Pruning. Notiamo anche che a parità di profondità Hl impiegherà mediamente più tempo per effettuare una mossa.

Considerazioni finali

Abbiamo visto l'implementazione di tre nuove versioni del MinMax Alpha-Beta Pruning:

- MinMax Alpha-Beta Pruning con taglio e valutazione H0
- MinMax Alpha-Beta Pruning con taglio e valutazione Hl
- MinMax Alpha-Beta Pruning con utilizzo del regessore

Il più veloce risulta essere H0 a discapito però della bontà di scelta della mossa rispetto al normale MinMax Alpha-Beta Pruning. Seguito da Hl che però è leggermente più lento del normale MinMax Alpha-Beta Pruning. Fanalino di coda è Hr che risulta essere più lento e meno bravo a scegliere delle buone mosse.

Per quanto riguarda il migliore I dai test effettuati, il migliore I quando L=10 è 2 in quanto maggiore di 2 avvevo un incremento temporale estremamente significativo infatti si passava da 14.55 minuti a 58.98 minuti. Anche se non cambiava l'esito della partita.

Confrontando queste versioni tra di loro ci accorgiamo che il migliore a scegliere è Hl seguito da H0, il peggiore rimane Hr.

Informazioni sui migliori motori di gioco

Stockfish

Stockfish utilizza un algoritmo di ricerca basato su Alpha-Beta Pruning, arricchito da molteplici ottimizzazioni. Tra queste, si annoverano le tabelle hash (per memorizzare e riutilizzare le valutazioni di posizioni già analizzate), la ricerca di quiescenza (per evitare di valutare posizioni in stato di flusso), e l'implementazione di euristiche come il Null Move Heuristic (che permette di ridurre l'albero di ricerca sotto certe condizioni).

Studiando questo motore ho cercato di riutilizzare l'idea delle tabelle hash per memorizzare e riutilizzare le valutazioni di posizioni già analizzate. Nel mio caso ho implementato un dizionario python. Ciò ha migliorato le tempistiche medie.

Leela Chess Zero (LCZero)

LCZero si basa su reti neurali e apprendimento rinforzato. Si ispira a AlphaZero di Google DeepMind, utilizzando una rete neurale per valutare le posizioni e selezionare le mosse. LCZero apprende giocando contro se stessa, migliorando nel tempo.

Apparentemente non usa la nostra idea di MinMax Alpha-Beta Hr.

Komodo

Komodo impiega l'algoritmo Alpha-Beta Pruning, con una serie di ottimizzazioni e euristiche proprie. È noto per il suo equilibrio tra forza tattica e comprensione strategica.

AlphaZero

Sviluppato da Google DeepMind, AlphaZero utilizza un approccio basato su reti neurali e apprendimento rinforzato. Invece di basarsi su database di aperture o librerie di mosse finali, AlphaZero apprende giocando contro se stesso, partendo da principi di gioco di base.

Ricercando su internet non ho trovato un motore che usi la nostra idea di MinMax Alpha-Beta Hr.

In [16]: