Homework 1

Student: Alessandro Mattei

Matricola: Non Disponibile

Email: alessandro.mattei1@student.univaq.it

Introduzione

Nel Homework 1 è stato richiesto di creare due giochi/puzzle. Ho scelto di proporre il gioco degli **Scacchi** e il **15-Puzzle**

In entrambi i giochi 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 utilizzate per risolvere entrambi i giochi nei file:

- AlessandroMattei_ChessGame.AIhw1.ipynb
- AlessandroMattei_FifteenPuzzleGame.AIhw1.ipynb

Nella fase finale viene mostrata e descritta una analisi statistica dei risultati delle varie sperimentazioni eseguite con istanze di stadi iniziali diversi, diversi algoritmi di ricerca e differenti euristiche per entrambi i giochi.

Implementazione

Agente - Agent Class

```
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, , 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" ad 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

A*

L'algoritmo A* è un algoritmo di ricerca informata utilizzato per la ricerca del percorso più breve da un punto iniziale a un punto finale. Esso combina i vantaggi della ricerca in ampiezza (garantendo una soluzione ottima) con quelli della ricerca guidata euristica (per ridurre il numero di esplorazioni).

Ad ogni nodo nel horizonte, associa un valore g, che rappresenta il costo per raggiungere quel nodo dal nodo iniziale, un valore h, che è l'euristica (una stima del costo per raggiungere il nodo finale da quel nodo) e un valore f, dove f = g + h.

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

Questo algoritmo per essere istanziato accetta come parametri la classe di Euristica e la classe Gioco

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

```
def evaluate(self, state):
    """
    Computes the cost (g value), heuristic value (h value), and combined
cost (f value) for a given state.

    The g value is the actual cost to reach the state, the h value is the
estimated cost from the state to the goal,
    and the f value is the sum of the g and h values.
```

```
    :param state: The state for which the costs need to be computed.
    """
    if state.g is None:
        if state.state_parent is None:
            state.g = 0
        else:
            state.g = state.state_parent.g + 1  # fixed cost set to 1

        state.h = heuristic.h(state)

        state.f = state.g + state.h
```

La valutazione di ogni stato inizia controllando se lo stato in questione è lo stato iniziale e gli viene attribuito un valore g = 0, altrimenti se si tratta di uno stato intermedio g viene calcolata sommando il valore g del suo stato *"state_parent"* ed applica il costo della mossa del gioco più il vaolore dell'euristica *"state.f = state.g + state.h"*.

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

```
def pick(self):
    """
    Selects the state with the lowest combined cost (f value) from the
horizon.
    :return: The state with the lowest combined cost (f value).
    """
    return min(self.horizon, key=lambda state: state.f)
```

pick() ha il compito di scegliere la mossa che costa meno, quindi più efficiente Di seguito possiamo vedere **search()**:

```
def search(self, state):
    """
    Expands the search from the given state by exploring its neighbors.

    The method evaluates neighboring states of the given state, calculates
their costs,
    and adds them to the search horizon if they haven't been explored
before. It also updates the
    set of explored states.
    :param state: The state from which the search should be expanded.
    """
    if state not in self.horizon:
        self.evaluate(state)
        self.horizon.add(state)

    self.explored.add(state)

    neighbors = self.game.neighbors(state)

    for neighbor in neighbors:
        if neighbor not in self.horizon and neighbor not in self.explored:
            self.evaluate(neighbor)
            if neighbor.is_end_game():
                return neighbor
            self.horizon.add(neighbor)

    self.horizon.remove(state)

    if len(self.horizon) > 0:
```

```
            return self.pick()
        else:
            return None
```

Il metodo valuta gli stati confinanti di uno stato dato, ne calcola i costi, e li aggiunge all'orizzonte di ricerca se non sono stati esplorati prima. Aggiorna anche il insieme di stati esplorati.

BestFirst

L'algoritmo Best-First Search è una tecnica di ricerca informata che esplora un grafo dando priorità ai nodi in base a una funzione euristica.

In altre parole, anziché esplorare in maniera "cieca" come farebbe una ricerca in ampiezza o in profondità, Best-First Search tenta di andare "direttamente" verso l'obiettivo, basandosi su una stima di quale nodo potrebbe essere il "migliore" da esplorare dopo.

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

Questo algoritmo per essere istanziato accetta come parametri la classe di Euristica e la classe Gioco.

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

```
def f_value(self, states):
    """
    Computes the heuristic value for a given set of states.
    :param states: A set of states for which the heuristic values need to be
computed.
    """
    for state in states:
        if state.f is None:
            state.f = self.heuristic.h(state)

def evaluate(self, state: StateFifteenPuzzleGame):
    """
    Computes the heuristic value for a specific state.
    :param state: The state for which the heuristic value needs to be
computed.
    :return:
    """
    if state.f is None:
        state.f = self.heuristic.h(state)
        state.h = state.f
```

La funzione evaluate e f_value si occupano di richiamare l'euristica per valutare uno stato se esso non è stato già valutato in precedenza. Di seguito possiamo vedere **pick()**:

```
def pick(self):
    """
    Selects the state with the lowest heuristic value from the horizon.
    :return: The state with the lowest heuristic value.
    """
    return min(self.horizon, key=lambda state: state.f)
```

pick() ha il compito di scegliere la mossa che costa meno, quindi più efficiente Di seguito possiamo vedere **search()**:

```
def search(self, state):
    """
    Expands the search from the given state, evaluating neighboring states
and adding them to the horizon.
    :param state: The state from which the search should be expanded.
    :return: The state with the lowest heuristic value. None otherwise.
    """
    if state not in self.horizon:
        self.evaluate(state)
        self.horizon.add(state)

    self.explored.add(state)

    neighbors = self.game.neighbors(state)

    for neighbor in neighbors:
        if neighbor not in self.horizon and neighbor not in self.explored:
            self.evaluate(neighbor)
            if neighbor.is_end_game():
                return neighbor
            self.horizon.add(neighbor)

    self.horizon.remove(state)

    if len(self.horizon) > 0:
        return self.pick()
    else:
        return None
```

pick() ha il compito di espande la ricerca dallo stato dato, valutando gli stati vicini e aggiungendoli all'orizzonte.

MinMax

L'algoritmo MinMax è una tecnica utilizzata principalmente per prendere decisioni in giochi a turni tra due giocatori, come appunto gli scacchi, la dama e il tris (tic-tac-toe).

L'obiettivo dell'algoritmo è di massimizzare le possibilità di vittoria del giocatore attuale minimizzando al contempo le migliori mosse possibili del suo avversario.

L'algoritmo MinMax analizza strategie ottimali per i due giocatori durante il gioco. Iniziando dai risultati finali, esamina ogni mossa in una prospettiva retrograda.

In ogni passaggio, si assume che il giocatore 1 (noto come giocatore max) miri a massimizzare le sue opportunità di trionfo, mentre il giocatore 2 (il giocatore min) si sforzi di minimizzare le probabilità di successo dell'avversario.

Pertanto, mentre il giocatore max cerca di elevare il punteggio complessivo della partita, il giocatore min ambisce a diminuirlo.

Grazie all'algoritmo MinMax, entrambi i giocatori possono minimizzare i rischi, optando per strategie che prevengono mosse pericolose o che potrebbero consegnare la vittoria nelle mani

dell'opponente.

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.

```
def __init__(self, game, heuristic, max_depth=1):
    """
    Initializes an instance of the MinMax class.
    :param game: The game for which the search is performed.
    :param heuristic: The heuristic used to evaluate the game states.
    :param max_depth: The maximum depth of the search. Default is 1.
    """
    self.game = game
    self.heuristic = heuristic
    self.max_depth = max_depth
    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()**, **evaluate()**, **pick()** e **search()**.

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

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

    This static method selects the best game state from a list of states
based on their heuristic values.
    The selection is determined by whether it's the maximizing player's turn
or the minimizing player's turn.

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

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.

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

```
def evaluate(self, states, parent_turn):
    """
    Evaluates a list of game states using the Minimax algorithm.

    This method evaluates a list of game states using the Minimax algorithm,
which is a decision-making algorithm in
    game theory for minimizing the possible loss for a worst-case scenario.
It assigns heuristic values to
```

```
    each state based on the algorithm's calculations.

    :param states: List of game states to evaluate.
    :param parent_turn: Indicates whose turn it is: True for the player
trying to maximize and False
                        for the player trying to minimize.
    """
    for state in states:
        if state.can_claim_draw():
            state.h = 0.0  # Set the heuristic value to 0 if the game can be
claimed as a draw.
        else:
            # Calculate heuristic value using Minimax.
                state.h = self.__minmax(state, self.max_depth - 1, not
parent_turn)
```

La funzione evaluate() viene richiamata per valutare gli stati neighbors prima di chiamare la funzione pick() all'interno della funzione search(). Prima di chiamare la funzione helper __minmax(), per stimare il valore dello stato, controlla se tale stato è considerabile come un pareggio associandogli il valore 0.0

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

```
def __minmax(self, state, depth, turn):
    """
    Recursive helper method to perform the Minimax search.

    This private method performs a recursive Minimax search on a game tree
to determine the heuristic
    value of a given game state.

    :param state: The current game state.
    :param depth: The current depth in the search.
    :param turn: Indicates whose turn it is: True for the player trying to
maximize and False for the player
                trying to minimize.
    :return: Heuristic value of the provided game state.
    """
    self.eval_count += 1  # Increment evaluation count.
    neighbors = self.game.neighbors(state)  # Get neighboring states from
the current state.

    # Base cases: If the search depth is 0 or if the game is in an endgame
state, return the heuristic value.
    if depth == 0 or state.is_endgame():
        return self.heuristic.h(state)

    if turn:
        value = -np.inf  # Initialize value for maximizing player to
negative infinity.
        for child in neighbors:
            value = max(value, self.__minmax(child, depth - 1, False))  #
Recursively maximize.
        return value
    else:
        value = np.inf  # Initialize value for minimizing player to positive
infinity.
        for child in neighbors:
```

```
            value = min(value, self.__minmax(child, depth - 1, True))  # Recursively minimize.
        return value
```

la funzione helper **__minmax()** è la funzione ricorsiva che calcola il valore di ogni stato utilizzando max per il primo giocatore e min per il secondo giocatore.

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

```
def search(self, state):
    """
    Initiates the Minimax search for a given game state.

    This method initializes the Minimax search process for a given game state.
    It calculates the heuristic values for the neighboring states and selects the best next state based
    on the Minimax algorithm.

    :param state: The game state to start the search from.
    :return:  Best next game state based on the Minimax algorithm.
    """
    neighbors = self.game.neighbors(state)  # Get neighboring states from the current state.
    self.evaluate(neighbors, state.turn())  # Calculate heuristic values for the neighbors.
    return self.pick(neighbors, state.turn())  # Select the best next state using the Minimax algorithm.
```

search() che viene chiamata dall'agente per ogni mossa e si occupa di trovare gli stati neighbors e richiamare le funioni evaluate() e pick() descritte in precedenza.

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 game states using the Minimax algorithm with Alpha-
Beta pruning.

    This function evaluates a list of game states using the Minimax
algorithm with Alpha-Beta pruning.
    It assigns a heuristic value to each state based on its evaluation at a
specified depth in the game tree.
    The depth of the evaluation is determined by the 'max_depth' attribute
of the object.

    :param states: List of game states to evaluate.
    :param parent_turn: Indicates whose turn it is: True for maximizing
player and False for minimizing player.
    """
    for state in states:
        if state.can_claim_draw():
            # If the state can claim a draw, assign a heuristic value of 0.
            state.h = 0.0
        else:
            # Otherwise, use the Minimax algorithm with Alpha-Beta pruning
to assign a heuristic value.
            state.h = self.__minmax_alpha_beta(state, self.max_depth - 1, -
np.inf, np.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):
    """
    Recursive helper method to perform Minimax search with Alpha-Beta
pruning.

    This private method performs a recursive Minimax search with Alpha-Beta
pruning to find the optimal move
    in the game tree.
    It evaluates the provided game state and returns a heuristic value based
on the current player's turn.

    :param state: Current game state.
    :param depth: Current depth in the search.
    :param alpha: Best already explored option for the maximizer.
    :param beta: Best already explored option for the minimizer.
    :param turn: Indicates whose turn it is: True for maximizing player and
False for minimizing player.
    :return: Heuristic value of the provided game state.
    """
    self.eval_count += 1  # Count the number of state evaluations.
    neighbors = self.game.neighbors(state)  # Generate possible successor
states.

    if depth == 0 or state.is_endgame():
        # Base case: If the maximum depth is reached or the state represents
an endgame, return the heuristic value.
```

```
            return self.heuristic.h(state)

        if turn:  # Maximizing player
            value = -np.inf
            for neighbor in neighbors:
                value = max(value, self.__minmax_alpha_beta(neighbor, depth - 1,
alpha, beta, False))
                alpha = max(alpha, value)  # Update alpha with the maximum value
found so far.
                if alpha >= beta:  # Alpha-Beta pruning: Stop evaluating if
alpha is greater than or equal to beta.
                    self.prune_count += 1  # Count pruned branches.
                    break
            return value
        else:  # Minimizing player
            value = np.inf
            for neighbor in neighbors:
                value = min(value, self.__minmax_alpha_beta(neighbor, depth - 1,
alpha, beta, True))
                beta = min(beta, value)  # Update beta with the minimum value
found so far.
                if beta <= alpha:  # Alpha-Beta pruning: Stop evaluating if beta
is less than or equal to alpha.
                    self.prune_count += 1  # Count pruned branches.
                    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.

HEURISTICS

ManhattanDistancePuzzleGame -

L'euristica del **Manhattan Distance** (o Distanza di Manhattan) è una misura utilizzata in informatica, particolarmente nell'intelligenza artificiale e nella robotica, per stimare la distanza tra due punti in una griglia. Il suo nome deriva dalla struttura a griglia delle strade di Manhattan, a New York. Qui, per spostarsi da un punto A a un punto B, è possibile muoversi soltanto orizzontalmente o verticalmente tra le strade, escludendo gli spostamenti in diagonale.

La formula per calcolare la Distanza di Manhattan tra due punti ((x_1, y_1)) e ((x_2, y_2)) in una griglia 2D è: [\text{Manhattan Distance} = |x_1 - x_2| + |y_1 - y_2|]

In termini semplici, corrisponde alla somma delle differenze assolute tra le loro coordinate x e y.

Come si può vedere dal file *AlessandroMattei_FifteenPuzzleGame.Alhw1.ipynb*, nel quale è contenuta l'intera implementazione, possiamo notare la presenza di due metodi **h()** che è l'euristica vera e propria e **"index_to_position()"** che è una funzione di helper.

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

```
def h(self, state: StateFifteenPuzzleGame):
    """
    Computes the heuristic value for a given state based on the Manhattan
distance of tiles.

    The heuristic value is the sum of the Manhattan distances of each tile
from its goal position.
    :param state: The current state of the Fifteen Puzzle game.
    :return: The heuristic value for the given state.
    """
    distance = 0  # Initialize the total distance to 0.

    for i in range(16):
        # Iterate through each tile on a 4x4 board (16 tiles in total).
        if state.game_representation.game_board[i] == 0:  # Ignore the empty
tile.
            continue

        # Convert the current tile's index to its (x, y) position on the
board.
        current_position = self._index_to_position(i)
        # Find the goal position of the current tile and convert its index
to (x, y) position.
        goal_position =
self._index_to_position(self.goal_position.index(state.game_representation.game_

        # Calculate the Manhattan distance between the current position and
the goal position.
        # The Manhattan distance is the sum of the absolute differences of
their coordinates.
        distance += abs(current_position[0] - goal_position[0]) +
abs(current_position[1] - goal_position[1])

    return distance  # Return the total Manhattan distance.
```

Questa euristica calcola la distanza Manhattan di tutte le celle dalla sua posizione attuale alla posizione goal e la somma per stimare uno stato.

MisplacedTittlesPuzzleGame - Misplaced Tittles

L'euristica dei Misplaced Tiles (in italiano "Piastrelle Fuori Posto") è comunemente utilizzata nel contesto di puzzle come il 15-puzzle (o puzzle a scorrimento).

L'idea di base di questa euristica è piuttosto semplice: si calcola il numero di piastrelle (o tessere) che non si trovano nella loro posizione corretta o finale.

Dato un certo stato del puzzle, confrontiamo la posizione di ogni piastrella con la sua posizione nella soluzione finale.

Ogni volta che una piastrella non si trova nella posizione dove dovrebbe essere nella soluzione finale, incrementiamo un contatore.

Il valore finale del contatore rappresenta il numero di piastrelle fuori posto e, quindi, l'euristica dei Misplaced Tiles per quel particolare stato del puzzle.

Come si può vedere dal file *AlessandroMattei_FifteenPuzzleGame.AIhw1.ipynb*, nel quale è contenuta l'intera implementazione, possiamo notare la presenza di due metodi **h()** che è l'euristica vera e propria.

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

```
def h(self, state: StateFifteenPuzzleGame):
    """
    Computes the heuristic value for a given state based on the number of
misplaced tiles.

    The heuristic value is the number of tiles that are not in their goal
position, excluding the empty tile (0).
    :param state: The current state of the Fifteen Puzzle game.
    :return: The heuristic value for the given state.
    """
    return sum(
        1 for s, g in zip(state.game_representation.game_board,
self.goal_position)
        if s != g and s != 0
    )
```

Questa euristica calcola il numero di celle posizionate in modo errato rispetto allo stato goal per valutare ogni stato.

SoftBoardEvaluationChessGame - Simple Chess Board Evaluation

Questa euristica combina varie euristiche. È l'euristica più semplice che è presente per il gioco Scacchi. Ritorna la somma dei risultati delle singole euristiche

Euristiche Combinate:

- evaluate_board: Valuta la qualità complessiva del consiglio di amministrazione.
- material_evaluation: Valuta la Scacchiera in base al materiale presente.
- piece_square_evaluation: Valuta la scacchiera in base alla posizione dei pezzi.
- mobility_evaluation: Valuta la Scacchiera in base alla mobilità dei pezzi.
- king_safety_evaluation: Valuta il tabellone in base alla sicurezza del re.
- center_control_evaluation: Valuta il controllo del tabellone dei quadrati centrali.

Per capire e vedere nel dettaglio come sono state implementate le singole euristiche vedere il file *AlessandroMattei_ChessGame.AIhw1.ipynb*

HardBoardEvaluationChessGame - More Complex Chess Board Evaluation

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

Euristiche Combinate:

- king_safety: valuta la scacchiera in base alla sicurezza del re.
- all_piece_values_and_piece_square_tables: valuta la qualità complessiva della tavola.

- center_control: Valuta il controllo delle caselle centrali sulla scacchiera.
- mobility: Valutare la mobilità dei pezzi sulla scacchiera.
- attack_value: Valuta il valore degli attacchi dei pezzi sulla scacchiera.
- rooks_on_open_files: Valuta la presenza di torri su file aperti nella scacchiera.
- check_forks: Valuta la presenza di opportunità di fork nella posizione degli scacchi.
- check_pins: Valuta la presenza di pezzi bloccati nella posizione degli scacchi.

Per capire e vedere nel dettaglio come sono state implementate le singole euristiche vedere il file *AlessandroMattei_ChessGame.AIhw1.ipynb*

GAMES

FifteenPuzzleGame - 15 Puzzle Game

Il 15 Puzzle, noto anche come "Puzzle a scorrimento" o "Gioco delle quindici", è un puzzle di tipo combinatorio inventato intorno al 1870. È composto da un piccolo quadro suddiviso in caselle, 4x4, che contiene 15 piastrelle numerate da 1 a 15, lasciando una casella vuota nel nostro caso 0.

OBBIETTIVO

L'obiettivo del gioco è riorganizzare le piastrelle, inizialmente mescolate in modo casuale, in ordine crescente, utilizzando la casella vuota per far scorrere le piastrelle adiacenti.

MODALITÀ DI GIOCO

L'agente può spostare spazio vuoto verso una tessera in quell'area, creando una nuova configurazione dei pezzi.

IMPLEMENTAZIONE

Per risolvere il puzzle richiede pianificazione e strategia. Alcune configurazioni iniziali potrebbero essere più difficili da risolvere di altre.

RAPPRESETAZIONE

Il gioco viene modellato tramite la classe **FifteenPuzzleRepresentation**, questa classe fornisce metodi per verificare se il tabellone di gioco è risolvibile e per contare il numero di inversioni nello stato attuale del tabellone, le mosse che si posso fare e altre funzioni utili.

Il gioco è implementato usando le tuple al posto degli array.

Un esempio di metodi che troviamo è **game_is_solvable()**, che ci dice se il puzzle è risolvibile o meno. Di seguito possiamo vedere la funzione helper **game_is_solvable()**:

```
def game_is_solvable(self):
    """
    Determines if the current game board configuration is solvable.
    A game board is solvable if the number of inversions is even.
    :return: True if the game board is solvable, False otherwise.
    """
```

```
    # Counts the number of inversions in the state.
    inversions = self.count_inversions()
    # If the number of inversions is even, the state is solvable.
    return inversions % 2 == 0
```

Per maggiori dettagli su come è stato implementato la rappresentazione vedere il file *AlessandroMattei_FifteenPuzzleGame.AIhw1.ipynb*

CLASSE GIOCO

All'interno del progetto per questo gioco troviamo una classe chiamata **FifteenPuzzleGame** 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 **FifteenPuzzleGame**:

```
def neighbors(self, state: StateFifteenPuzzleGame):
    """
    Computes the neighboring states that can be reached from the given
state.
    The method calculates the possible states that can be reached by making
    the "UP", "DOWN", "LEFT", and "RIGHT" moves from the current state.
    :param state:  The current state of the Fifteen Puzzle game.
    :return: A set of neighboring states that can be reached from the given
state.
    """
    neighbors_state = set()
    moves = {
        "UP": state.game_representation.move_up(),
        "DOWN": state.game_representation.move_down(),
        "LEFT": state.game_representation.move_left(),
        "RIGHT": state.game_representation.move_right(),
    }
    for move, new_state in moves.items():
        if new_state is not None:
            neighbors_state.add(
                StateFifteenPuzzleGame(game_representation=new_state,
state_parent=state, move=move))
    return neighbors_state
```

Il metodo calcola i possibili stati raggiungibili effettuando le mosse "UP", "DOWN", "LEFT", and "RIGHT" quando possibili dallo stato corrente.

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.

RAPPRESETAZIONE

Il gioco viene modellato tramite la classe **ChessRepresentation**. Questa classe non solo offre una descrizione dello stato attuale della scacchiera durante una partita, ma mette anche a disposizione diversi metodi utili per interagire e valutare con il tavoliere.

Per maggiori dettagli su come è stato implementato la rappresentazione e quali sono i metodi utili vedere il file *AlessandroMattei_ChessGame.AIhw1.ipynb*

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):
    """
    Determines the neighboring states of the provided chess game state.
    :param state: The current state of the chess game.
    :return: A list of neighboring states for the given state.
    """
    neighbors = []

    # Iterate through all legal moves and compute the resulting game state
    for legal_move in state.game_representation.get_all_legal_moves():
        representation = state.game_representation.make_a_move(legal_move)
        neighbor = StateChessGame(game_representation=representation,
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.

StateFifteenPuzzleGame - STATE Puzzle Game

Per 15-Puzzle lo stato è composto dalla rappresentazione corrente del gioco **game_representation**, dal suo nodo parent **state_parent**, dalla mossa **move** che verrà effettuata e dai tre valori delle funzioni g(n), h(n) e f(n).

Di seguito possiamo vedere Inizializzazione nella classe **StateFifteenPuzzleGame**:

```
def __init__(self, game_representation=None, state_parent=None, move=None):
    """
    Initializes the StateFifteenPuzzleGame with a game representation,
parent state, and move.
    :param game_representation: The game board representation.
    :param state_parent: The parent state.
    :param move: The move that led to this state.
    """
    self.game_representation = game_representation
    self.state_parent = state_parent
    self.move = move
    self.h = None
    self.g = None
    self.f = None

    if self.game_representation is None:
        self.game_representation = FifteenPuzzleRepresentation()
```

Per maggiori dettagli vedere il file *AlessandroMattei_FifteenPuzzleGame.AIhw1.ipynb*

StateChessGame - STATE Chess

Nel contesto del gioco degli scacchi, lo stato riveste un'importanza cruciale. Per Chess lo stato è composto dalla rappresentazione corrente del gioco **game_representation**, dal suo nodo parent **state_parent**, dalla mossa **move** che verrà effettuata e dal valore della funzione h(n). Vengono forniti altri metodi utili al interno di questa classe.

```
def __init__(self, game_representation=None, state_parent=None, move=None):
    """
    Initializes the chess game state.
    :param game_representation: The chess board state.
           Defaults to a new chess board state if not provided.
    :param state_parent: The preceding state. Defaults to None.
    :param move: The move leading to this state. Defaults to None.
    """
```

```
    self.game_representation = game_representation
    self.parent_state = state_parent
    self.move = move
    self.h = None

    if self.game_representation is None:
        self.game_representation = ChessRepresentation()
```

Per maggiori dettagli vedere il file *AlessandroMattei_ChessGame.AIhw1.ipynb*

Report

Vengono riportati i risultati dei test effettuati per entrambi i giochi

Fifteen Puzzle Game

Stato Iniziale: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 0, 15, 13, 14)

SEARCH ALGORITHM	HEURISTIC	TIME RESULT	NUMBER OF STEPS	HORIZON	VISITED
BestFirst	ManhattanDistance	47.66 ms	42	323	283
BestFirst	MisplacedTittles	376.99 ms	120	2802	2689
AStar	ManhattanDistance	35.52 ms	16	187	175
AStar	MisplacedTittles	64.04 ms	16	567	548

Stato Iniziale: (1, 2, 3, 4, 5, 6, 7, 9, 8, 10, 11, 12, 14, 13, 0, 15)

SEARCH ALGORITHM	HEURISTIC	TIME RESULT	NUMBER OF STEPS	HORIZON	VISITED
BestFirst	ManhattanDistance	68.74 ms	156	734	663
BestFirst	MisplacedTittles	45.76 ms	155	734	663
AStar	ManhattanDistance	68908.69 ms	31	47483	50728
AStar	MisplacedTittles	75661.41 ms	31	47483	50728

Stato Iniziale: (4, 2, 1, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0)

SEARCH ALGORITHM	HEURISTIC	TIME RESULT	NUMBER OF STEPS	HORIZON	VISITED
BestFirst	ManhattanDistance	152.31 ms	150	1905	1757
BestFirst	MisplacedTittles	142.83 ms	150	1905	1757
AStar	ManhattanDistance	16482.19 ms	28	23991	25937
AStar	MisplacedTittles	17389.41 ms	28	23991	25937

Stato Iniziale: (4, 2, 1, 3, 6, 7, 5, 8, 9, 10, 11, 12, 13, 14, 15, 0)

SEARCH ALGORITHM	HEURISTIC	TIME RESULT	NUMBER OF STEPS	HORIZON	VISITED
BestFirst	ManhattanDistance	37.09 ms	100	574	508
BestFirst	MisplacedTittles	32.84 ms	100	574	508
AStar	ManhattanDistance	252888.85 ms	32	78746	88588
AStar	MisplacedTittles	236101.01 ms	32	78746	88588

OSSERVAZIONI

Dai risultati sopra possiamo evincere che BestFirst impiega un tempo di soluzione nettamente inferiore, ma se guardiamo la qualità del risultato possiamo vedere che è inferiore. Infatti AStar produce una soluzione del gioco in meno mosse.

Per quanto riguarda le euristiche possiamo notare che per stati semplici MisplacedTittles è meno efficacie. Mentre per stati più complessi è leggermente più veloce se affiancato a BestFirst, risulta avere le stesse tempistiche di ManhattanDistance se affiancato ad AStar.

Chess Game

MinMaxAlphaBetaPruning - HardBoardEvaluationChessGame

AGENT	WINNER	DEPTH	TIME RESULT	NUMBER OF MOVES	STATES EVALUATED	PRUNING CARRIED OUT
AGENT 1 WHITE	FIVEFOLD_REPETITION	1	10583.87 ms	167	2128	0
AGENT 2 BLACK		1		167	1495	0
AGENT 1 WHITE	BLACK	1	40081.21 ms	45	538	0
AGENT 2 BLACK		2		45	19645	1
AGENT 1 WHITE	WHITE	2	50581.21 ms	28	17187	0
AGENT 2 BLACK		1		27	546	0
AGENT 1 WHITE	WHITE	2	59578.12 ms	38	20035	0
AGENT 2 BLACK		2		37	15004	2

OSSERVAZIONI

Dai risultati sopra possiamo evincere che quando facciamo giocare i due agenti con la stessa euristica a profondità diverse notiamo che vince chi ha più profondità. Caso speciale profondità 1-1 da risultato patta, invece 2-2 vince il Bianco.

MinMaxAlphaBetaPruning - SoftBoardEvaluationChessGame

AGENT	WINNER	DEPTH	TIME RESULT	NUMBER OF MOVES	STATES EVALUATED	PRUNING CARRIED OUT
AGENT 1 WHITE	WHITE	1	2715.75 ms	21	710	0
AGENT 2 BLACK		1		20	637	0
AGENT 1 WHITE	BLACK	1	44376.04 ms	18	453	0
AGENT 2 BLACK		2		18	22719	11
AGENT 1 WHITE	WHITE	2	50976.71 ms	44	26213	0
AGENT 2 BLACK		1		43	583	0
AGENT 1 WHITE	BLACK	2	80696.68 ms	53	23165	24
AGENT 2 BLACK		2		53	26395	0

OSSERVAZIONI

Dai risultati sopra possiamo evincere che quando facciamo giocare i due agenti con la stessa euristica a profondità diverse notiamo che vince chi ha più profondità. Caso speciale profondità 1-1 che da vittoria al Bianco, invece 2-2 vince il Nero che effettua anche una potatura.

MinMax - HardBoardEvaluationChessGame

AGENT	WINNER	DEPTH	TIME RESULT	NUMBER OF MOVES	STATES EVALUATED
AGENT 1 WHITE	FIVEFOLD_REPETITION	1	10809.84 ms	167	2128
AGENT 2 BLACK		1		167	1495
AGENT 1 WHITE	BLACK	1	38039.19 ms	45	538
AGENT 2 BLACK		2		45	19652
AGENT 1 WHITE	WHITE	2	50581.21 ms	28	17187
AGENT 2 BLACK		1		27	546

AGENT	WINNER	DEPTH	TIME RESULT	NUMBER OF MOVES	STATES EVALUATED
AGENT 1 WHITE	WHITE	2	63341.93 ms	38	20035
AGENT 2 BLACK		2		37	15004

OSSERVAZIONI

Dai risultati sopra possiamo evincere che quando facciamo giocare i due agenti con la stessa euristica a profondità diverse notiamo che vince chi ha più profondità. Caso speciale profondità 1-1 da risultato patta, invece 2-2 vince il Bianco. (Stessi risultati del MinMaxAlphaBetaPruning)

MinMax - SoftBoardEvaluationChessGame

AGENT	WINNER	DEPTH	TIME RESULT	NUMBER OF MOVES	STATES EVALUATED
AGENT 1 WHITE	WHITE	1	2795.28 ms	21	710
AGENT 2 BLACK		1		20	637
AGENT 1 WHITE	BLACK	1	46144.33 ms	18	453
AGENT 2 BLACK		2		18	23135
AGENT 1 WHITE	WHITE	2	51206.81 ms	44	26213
AGENT 2 BLACK		1		43	583
AGENT 1 WHITE	BLACK	2	79177.83 ms	53	23598
AGENT 2 BLACK		2		53	26395

OSSERVAZIONI

Dai risultati sopra possiamo evincere che quando facciamo giocare i due agenti con la stessa euristica a profondità diverse notiamo che vince chi ha più profondità. Caso speciale profondità 1-1 che da vittoria al Bianco, invece 2-2 vince il Nero che effettua anche una potatura. (Stessi risultati del MinMaxAlphaBetaPruning)

MinMaxAlphaBetaPruning - HardBoardEvaluationChessGame vs SoftBoardEvaluationChessGame

AGENT	HEURISTIC	WINNER	DEPTH	TIME RESULT	NUMBER OF MOVES	STATES EVALUATED
AGENT 1 WHITE	HardBoardEvaluationChessGame	BLACK	2	172374.91 ms	46	32908
AGENT 2 BLACK	SoftBoardEvaluationChessGame		2		46	37245
AGENT 1 WHITE	SoftBoardEvaluationChessGame	BLACK	2	89737.79 ms	48	20741

AGENT	HEURISTIC	WINNER	DEPTH	TIME RESULT	NUMBER OF MOVES	STATES EVALUATED
AGENT 2 BLACK	HardBoardEvaluationChessGame		2		48	25283

OSSERVAZIONI

Dai risultati sopra possiamo evincere che quando facciamo giocare i due agenti con le euristiche diverse a profondità 2-2 non hanno lo stesso effetto. Possiamo dire che HardBoardEvaluationChessGame è uguale a SoftBoardEvaluationChessGame a profondità 2-2

MinMax - HardBoardEvaluationChessGame vs SoftBoardEvaluationChessGame

AGENT	HEURISTIC	WINNER	DEPTH	TIME RESULT	NUMBER OF MOVES	STATES EVALUATED
AGENT 1 WHITE	HardBoardEvaluationChessGame	BLACK	2	172003.17 ms	46	34111
AGENT 2 BLACK	SoftBoardEvaluationChessGame		2		46	37245
AGENT 1 WHITE	SoftBoardEvaluationChessGame	BLACK	2	90300.03 ms	48	20888
AGENT 2 BLACK	HardBoardEvaluationChessGame		2		48	25283

OSSERVAZIONI

Dai risultati sopra possiamo evincere che quando facciamo giocare i due agenti con le euristiche diverse a profondità 2-2 non hanno lo stesso effetto. Possiamo dire che HardBoardEvaluationChessGame è uguale a SoftBoardEvaluationChessGame a profondità 2-2

OSSERVAZIONI FINALI

MinMax e MinMaxAlphaBetaPruning hanno lo stesso risultato, ciò che cambia è la velocità con cui vincono. MinMaxAlphaBetaPruning risulta essere più veloce, come ci aspettavamo.