Homework 1 - Artificial Intelligence

Introduction

This report explores two popular games, the 15 Puzzle and Chess, implemented using Python. The 15 Puzzle is a classic sliding puzzle that challenges players to arrange numbered tiles in a specific order. Chess, a strategic board game dating back centuries, involves intricate moves and tactics to outmaneuver opponents. By implementing these games in Python, we gain insights into how programming can simulate complex gaming scenarios and the role of algorithms in decision-making. This report discusses the process of creating these games in Python, highlighting the key strategies and programming concepts used in their development.

Chess game

Search algorithms

The implemented code integrates a fundamental algorithm, minimax, to enhance decision-making and optimize gameplay in the context Chess game.

MINIMAX

For Chess, the minimax algorithm is used for strategic decision-making and evaluating the best moves based on a turn-based gameplay scheme. This algorithm relies on a recursive search for the best possible scenario, taking into account the opponent's moves, in order to maximize the advantage in each turn.

Heuristics

The code employs three crucial heuristics, namely numberOfMoves, materialH, and numberPieces, to provide insightful evaluations of game states and facilitate intelligent decision-making strategies for the agents involved.

numberOfMoves

The numberOfMoves heuristic evaluates the number of legal moves available in a given game configuration. It provides an estimation of the player's mobility in a particular position and guides the decision-making process towards increased gameplay flexibility.

materialH

The materialH heuristic considers the overall value of pieces on the chessboard to evaluate a player's relative position. It takes into account the current material advantage and guides the player towards a strategy that maximizes the acquisition of opponent pieces.

numberPieces

The numberPieces heuristic evaluates the total number of pieces held by each player. It provides a general indication of the overall strength of a position and guides the player towards a strategy that maximizes occupation of the chessboard with their pieces.

Implementation

The Python implementation is structured into two primary classes, 'Heuristics' and 'Agent,' alongside a 'Chess' class to manage the game flow. The 'Heuristics' class encapsulates the calculation of various heuristic metrics, including the number of legal moves, material advantage, and the count of pieces, crucial for guiding the decision-making process within the game.

1. Heuristics:

- The `**Heuristics**` class includes various static methods that are used to evaluate the chessboard state based on different criteria. These methods calculate the number of legal moves available, the material advantage for a given color, and the difference in the number of pieces compared to the opponent's color.

2. Agent:

- The `**Agent**` class represents an entity that makes moves in the chess game and includes methods for implementing the minimax algorithm with alpha-beta pruning to select the best move. The class is initialized with a specific heuristic, color, maximum search depth for the minimax algorithm, and a folder to save board state images.
- The `**minimax**` method implements the minimax algorithm, recursively evaluating board states to determine the best move for the agent.
- The `**makeMove**` method selects the best move using the minimax algorithm and alpha-beta pruning, considering the current chessboard state and the defined search depth.

3. Chess:

- The `**Chess**` function simulates a chess game between two agents, allowing the agents to make moves based on the implemented algorithms and heuristics. The function tracks various game termination conditions, including checkmate, stalemate, insufficient material, and fifty-move rules, and prints the outcome of the game along with the number of moves made by each agent.

Libraries Used:

1. `chess`:
- The `**chess**` library is utilized for implementing chess-specific functionalities and managing the chessboard state. It provides essential features for representing chess positions, generating legal moves, and validating game termination conditions. The library enables seamless integration of chess-related logic and facilitates the simulation of chess games between agents.

2. `os`:
- The `**os**` module is used to provide a way to interact with the operating system, enabling functionalities such as creating directories, accessing file paths, and managing file operations. It is utilized in the code to define the path for saving board state images and handle file-related operations during the execution of the program.

3. `**time**`:
- The `time` module is employed to measure the execution time of specific sections of the code. It enables the tracking of the time elapsed during the chess game simulation, providing insights into the performance of the agents and the overall game duration.

4. `cairosvg` and `chess.svg`:
   - The **`cairosvg`** and **`chess.svg`** libraries are used for generating and saving images of the current chessboard state during the game simulation. They allow the program to create graphical representations of the chessboard and save the images in a designated folder. The libraries facilitate the visualization of the chessboard state at different stages of the game, enhancing the overall user experience and providing visual insights into the gameplay.

5. **`random`**:
   - The **`random`** module is utilized for generating pseudo-random numbers and making random selections. It is used in the code to select a random move from the best moves available to the agent. The module allows for random decision-making during the gameplay, introducing an element of unpredictability in the agent's move selection process.

Results

The results obtained from the simulations indicate various possible scenarios, including victories, draws, and defeats. The analysis of the employed heuristics highlights the importance of evaluating different aspects of the game, such as mobility, material advantage, and overall piece presence. Such considerations are crucial for developing winning strategies and improving decision-making capabilities in the context of chess games.
In the first simulation, where both agents utilize the 'numberOfMoves' heuristic, the game resulted in a draw due to the fivefold repetition rule, with Agent 1 making 25 moves and Agent 2 making 26 moves. The second simulation, utilizing the 'materialH' heuristic for both agents, also resulted in a draw due to the fivefold repetition rule. Agent 1 made 55 moves, while Agent 2 made 56 moves. Lastly, in the third simulation, with Agent 1 using the 'numberPieces' heuristic and Agent 2 using the 'materialH' heuristic, the game concluded with a checkmate, leading to Agent 1 (white) emerging as the winner after 35 moves each.

Conclusions

The implementation of the described algorithms and heuristics underscores the importance of a strategic approach in the game of chess. The integration of sophisticated algorithms and informative heuristics not only enriches the gaming experience but also empowers the decision-making abilities of players and involved artificial agents. Such implementation provides a solid foundation for developing competitive strategies and in-depth analysis of gameplay dynamics in the realm of chess. The varying outcomes of the simulated games demonstrate the significance of strategic decision-making and the impact of heuristic evaluations in determining the overall game progression and outcomes.

15 puzzle

Search algorithms

The implemented code integrates two fundamental algorithms, BFS and A*, to enhance decision-making and optimize gameplay in the context 15 puzzle.

Breadth-First Search (BFS)

This algorithm was employed to explore the state space in breadth. It utilizes a queue to maintain the nodes to be examined and ensures that all nodes of each level are visited before moving to the next level.

A* with Manhattan Distance Heuristic

A* is a search algorithm that uses both the actual cost to reach a specific state and an estimate of the remaining cost. In this case, the heuristic used is the Manhattan distance, which represents the sum of the horizontal and vertical distances between the current position of a piece and its goal position.

Heuristic

The heuristics are implemented within the `Heuristics` class

Manhattan Distance

Computes the Manhattan distance for a specific state, representing the sum of the horizontal and vertical distances between the current position of a piece and its goal position.

Misplaced Tiles

Computes the number of incorrectly positioned pieces with respect to the goal state.

Implementation

The code is implemented in Python and features three main classes: `State`, `Game`, and `Agent`. The `State` class represents a specific game state with attributes such as the matrix configuration, the parent, action, and cost. The `Game` class manages the game flow, including recognizing the goal state, generating possible actions, and transitioning between states. The `Agent` class implements functions to solve the puzzle using the algorithms and records relevant data.

1. State:

- This class represents the state of the puzzle, storing attributes such as the puzzle configuration, the parent state, the action taken, the cost, and the Manhattan distance.

- The `**calculate_manhattan**` method computes the Manhattan distance of the current state.

- The `**__lt__**` method allows the comparison of states for priority queue sorting.

2. Game:

- The `**Game**` class manages the 15-puzzle gameplay, offering methods to determine the goal state, available actions, and the subsequent state after a specific action.

- The `**is_goal**` method checks if the current state is the goal state.

- The `**get_possible_actions**` method returns a list of possible actions for the current state.

- The `**get_next_state**` method generates the next state after taking a specific action.

- The `**find_blank**` method finds the coordinates of the blank tile in the puzzle.

3. Heuristics:

- The `**Heuristics**` class provides two static methods, `**manhattan_distance**` and `**misplaced_tiles**`, to calculate the Manhattan distance and the number of misplaced tiles, respectively, for a specific state.

4. Agent:

- The `**Agent**` class initializes the agent with a specific game instance and offers methods to solve the puzzle using different search algorithms such as

Breadth-First Search (BFS) and A* with Manhattan distance or misplaced tiles heuristics.

5. Helper Functions:

- The `**randomise_4x4_matrix**` function generates a random 4x4 matrix.

- The `**solve_puzzle**` function solves the puzzle based on the specified search algorithm.

- The `**print_matrix**` function prints the matrix in a readable format.

- The `**reconstruct_path**` function reconstructs the path based on the solution.

Libraries Used:

The code leverages several libraries to streamline various operations and enhance the efficiency of the implemented algorithms

1. collections.deque:

- The `**deque**` class from the `**collections**` module is used for implementing the Breadth-First Search (BFS) algorithm. It provides an efficient way to manage a queue data structure with fast append and pop operations from both ends.

2. heapq:

- The `**heapq**` module provides an implementation of the heap queue algorithm, also known as the priority queue algorithm. It is used to implement the A* search algorithm with priority queues, facilitating efficient retrieval of the elements with the lowest cost.

3. random:

- The `**random**` module is utilized to generate random numbers and shuffle lists. It is used in the `**randomise_4x4_matrix**` function to randomize the initial 4x4 matrix for the 15-puzzle.

4. divmod:

- The built-in `**divmod**` function is used for obtaining the quotient and remainder from the division of two numbers. It is utilized to compute the target position coordinates for elements in the puzzle while calculating the Manhattan distance.

Results

The code produces a series of results showing the number of nodes explored by each algorithm for a specific initial configuration. In the presented test case, the BFS algorithm explored 41 nodes, while the A* algorithm with the Manhattan distance heuristic and A* algorithm with the misplaced tiles heuristic explored 9 and 12 nodes respectively. The solution was successfully found for the initial configuration of the 15 puzzle, demonstrating the effectiveness of the implemented algorithms.

Conclusion

A puzzle is considered solvable based on specific criteria, which include the following conditions: If N is odd, the number of inversions must be even, or if N is even, the blank must be in an even position counting from the bottom with an odd number of inversions, or the blank must be in an odd position counting from the bottom with an even number of inversions.
The implemented code demonstrates impressive efficacy in solving the 15-puzzle, as evidenced by the exploration of a specific initial configuration. The statistics generated from the code execution highlight the contrasting performance of different search algorithms, shedding light on their strengths and efficiency in exploring the search space.
In the presented test case, the Breadth-First Search (BFS) algorithm meticulously explored 41 nodes, showcasing its exhaustive approach in traversing the search space. On the other hand, the A* algorithm, when equipped with the Manhattan distance heuristic, explored a substantially smaller set of nodes, totaling 9, thus highlighting its efficiency in achieving optimal solutions. Similarly, the A* algorithm coupled with the misplaced tiles heuristic explored 12 nodes, showcasing its effectiveness in finding solutions through informed exploration of the search space.
These statistics not only underscore the diverse exploration strategies employed by the algorithms but also emphasize the critical role of heuristics in guiding the search process and improving the efficiency of the solution-finding process. The successful resolution of the 15-puzzle for the specified initial configuration serves as a testament to the robustness and effectiveness of the implemented algorithms and heuristics in solving complex problems efficiently.
The code's performance evaluation further underscores the significance of carefully selecting and implementing appropriate algorithms and heuristics, considering their respective strengths and limitations, to address specific problem contexts effectively and efficiently.

Lorenzo Tucceri Cimini
n. 294425