AIHW1 PROJECTS

Adam Bouafia – AIConDa UNIVAQ

TABLE OF CONTENTS

1. PROJECT DESCRIPTION:

The Hex game is a strategy-based board game for two players, played on a hexagonal grid. The objective of the game is for each player to form a connected path between their two sides of the grid. The game concludes when one player forms such a path, thereby emerging as the winner.

In the Hex game, there are two modes of gameplay: AI vs Player and Player vs Player. In the AI vs Player mode, the AI employs the Minimax and AlphaBeta pruning algorithms to determine its moves.

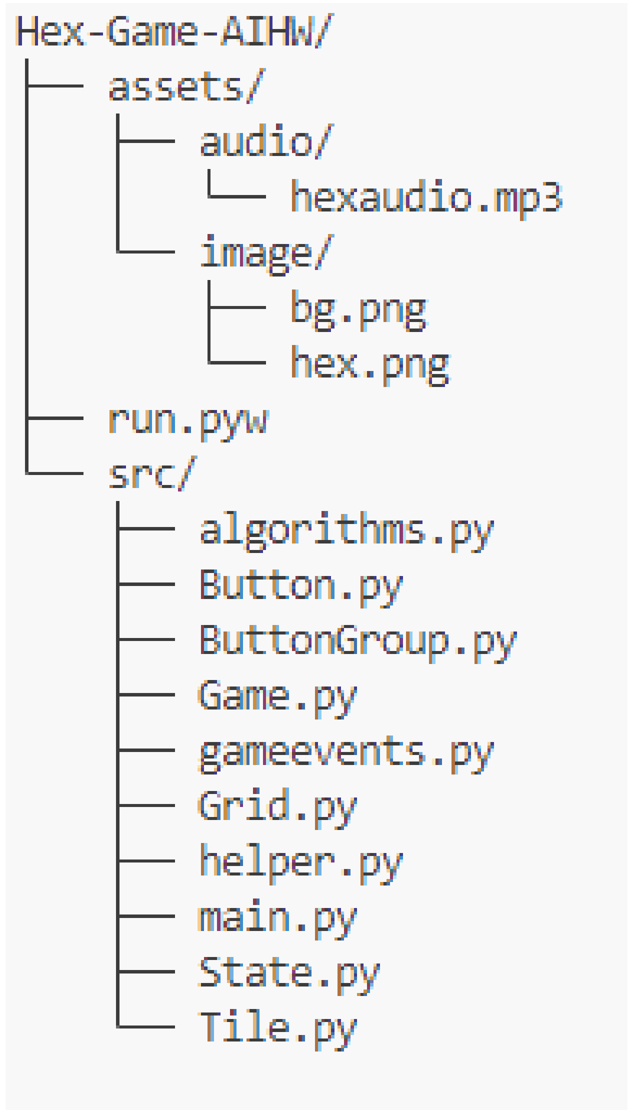
The Minimax algorithm is a decision-making algorithm used in game theory and artificial intelligence. It helps the AI to evaluate all possible future moves and select the one that maximizes its chance of winning. The AlphaBeta pruning is an enhancement of the Minimax algorithm that considerably reduces the number of nodes that the algorithm needs to evaluate. When it can be determined that a move is worse than a previously examined move, AlphaBeta pruning stops evaluating that move. This saves computational resources and time.

1. PROJECT DESCRIPTION:

In the Player vs Player mode, two players can compete against each other. This mode serves as an excellent platform for players to practice and improve their Hex strategies.

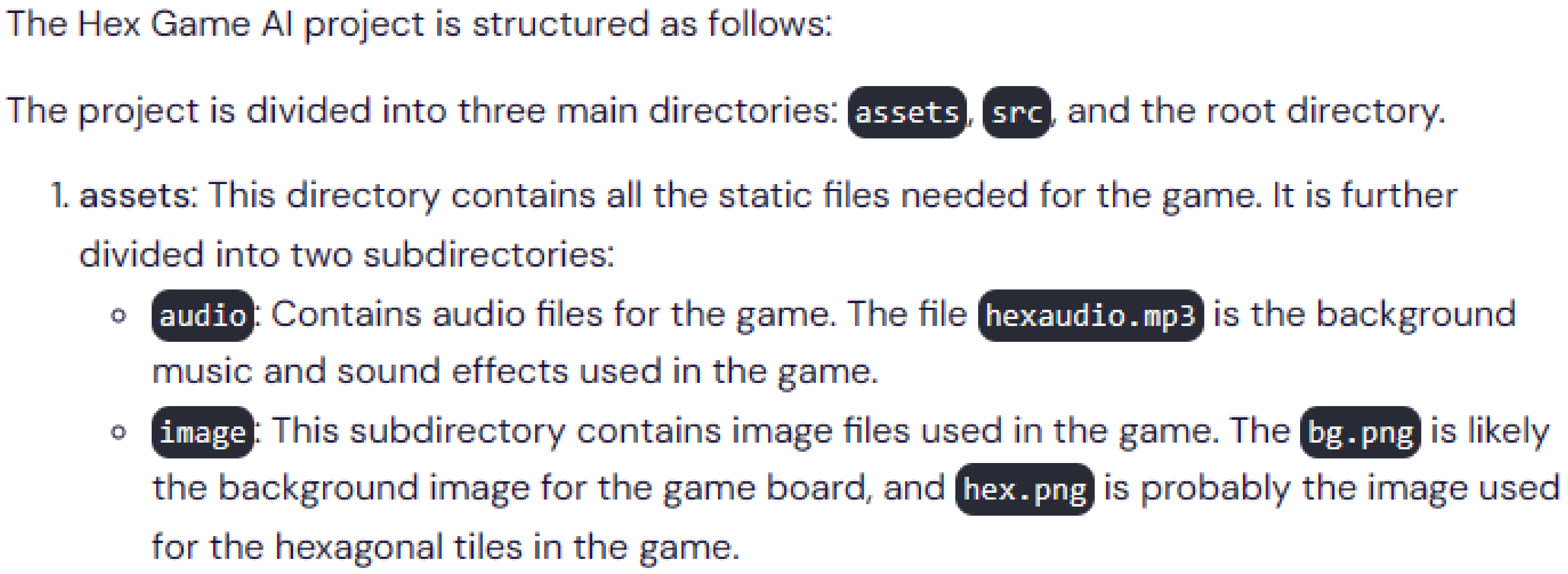
In summary, the Hex game is a practical application of AI techniques in board games. It offers an engaging user experience and a challenging AI opponent for Hex enthusiasts.

2. PROJECT STRUCTURE (DIRECTORY TREE)



```
Hex-Game-AIHW/
├── assets/
│   ├── audio/
│   │   └── hexaudio.mp3
│   └── image/
│       ├── bg.png
│       └── hex.png
├── run.pyw
└── src/
    ├── algorithms.py
    ├── Button.py
    ├── ButtonGroup.py
    ├── Game.py
    ├── gameevents.py
    ├── Grid.py
    ├── helper.py
    ├── main.py
    ├── State.py
    └── Tile.py
```

3. PROJECT IMPLEMENTATION DETAILS



The Hex Game AI project is structured as follows:

The project is divided into three main directories: `assets`, `src`, and the root directory.

1. assets: This directory contains all the static files needed for the game. It is further divided into two subdirectories:
   - `audio`: Contains audio files for the game. The file `hexaudio.mp3` is the background music and sound effects used in the game.
   - `image`: This subdirectory contains image files used in the game. The `bg.png` is likely the background image for the game board, and `hex.png` is probably the image used for the hexagonal tiles in the game.

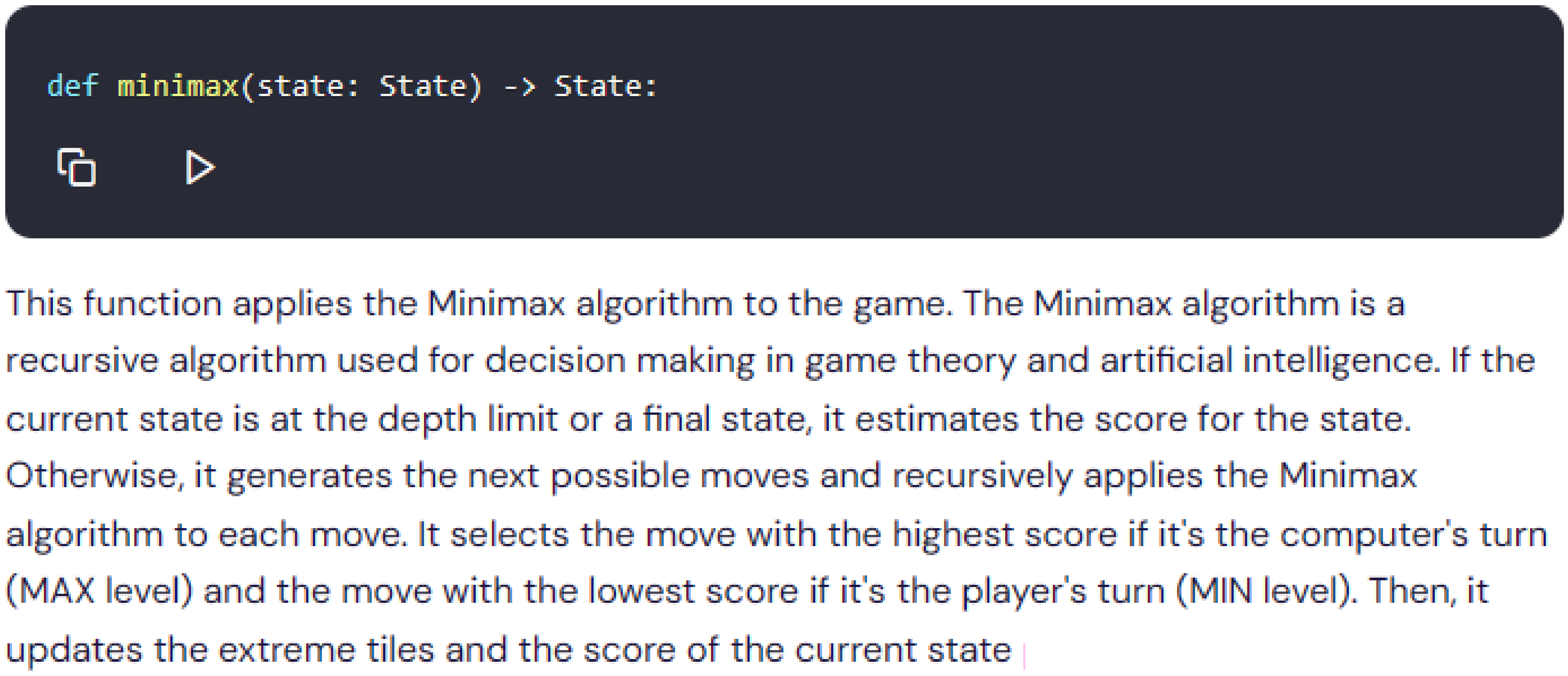
2. src: This directory contains the source code for the game :
    - `algorithms.py`: This file likely contains the implementation of the Minimax and AlphaBeta pruning algorithms used by the AI to decide its moves.
    - `Button.py` and `ButtonGroup.py`: These files probably contain the classes and methods related to the game's interactive buttons.
    - `Game.py`: This is likely the main game class that controls the game logic and manages the game state.
    - `gameevents.py`: This file likely handles the game events such as player moves and AI moves.
    - `Grid.py`: This file probably contains the class that represents the hexagonal grid used in the Hex game.
    - `helper.py`: This file likely contains helper functions used across the game.
    - `main.py`: This is the main entry point of the game. It likely initializes the game and controls the game loop.
    - `State.py`: This file probably contains the class that represents the state of the game at any given point.
    - `Tile.py`: This file likely contains the class that represents an individual tile on the game grid.

> 3. root directory: This contains the `run.pyw` script, which is the script that starts the game. It imports and runs the main function from `main.py`.

The game starts with the execution of `run.pyw`, which initializes the game and starts the game loop. The game state is managed by the `Game.py` class, and the player moves and AI moves are handled by the `gameevents.py` script. The AI uses the algorithms defined in `algorithms.py` to decide its moves. The game grid and individual tiles are represented by the `Grid.py` and `Tile.py` classes, respectively. The `Button.py` and `ButtonGroup.py` classes handle the interactive buttons in the game. The `helper.py` script contains helper functions used across the game. All the static files such as images and audio are stored in the `assets`

4. Heuristic Algorithms Implementation

Minimax Algorithm Function



```python
def minimax(state: State) -> State:
```

This function applies the Minimax algorithm to the game. The Minimax algorithm is a recursive algorithm used for decision making in game theory and artificial intelligence. If the current state is at the depth limit or a final state, it estimates the score for the state. Otherwise, it generates the next possible moves and recursively applies the Minimax algorithm to each move. It selects the move with the highest score if it's the computer's turn (MAX level) and the move with the lowest score if it's the player's turn (MIN level). Then, it updates the extreme tiles and the score of the current state

Alpha-Beta Pruning Function

```
def alpha_beta(alpha, beta, state):
```

This function applies the Alpha-beta pruning algorithm to the game. Alpha-beta pruning is an optimization technique for the Minimax algorithm. It reduces the number of nodes that need to be examined in the game tree by eliminating branches that don't need to be explored. If the current state is at the depth limit or a final state, it estimates the score for the state. Otherwise, it generates the next possible moves and recursively applies the Alpha-beta pruning algorithm to each move. It uses the alpha and beta values to cut off branches in the game tree that don't need to be explored

This implementation demonstrates the use of heuristic algorithms in game playing, specifically in the Hex game. The Minimax and Alpha-beta pruning algorithms are used to decide the best moves for the computer player. The `updateExtremeTiles` and `createPossibleMove` functions are used to update the game state and generate possible moves, respectively.

4. PERFERMONCE ANALYSIS

The performance analysis of the Hex game implementation using Minimax and Alpha-beta pruning can be discussed in terms of time complexity, space complexity, and game-playing performance.

Time Complexity

In terms of time complexity, the Minimax algorithm is known to have a time complexity of O(b^d), where b is the branching factor (the average number of successors of a node) and d is the depth of the game tree. In the worst-case scenario, the algorithm has to traverse all the nodes in the game tree.

However, the Alpha-beta pruning optimization can significantly reduce this time complexity. Alpha-beta pruning eliminates the need to explore unnecessary branches in the game tree, thereby reducing the number of nodes that need to be processed. The exact time complexity with Alpha-beta pruning depends on the order in which nodes are processed, but in the best case, it can reduce the time complexity to O(b^(d/2)).

Space Complexity

The space complexity of the Minimax and Alpha-beta algorithms is O(bd), as they perform depth-first search. Therefore, at any point, they only need to store a single path from the root to a leaf node, along with each node's siblings. This makes these algorithms space-efficient, as they do not need to store the entire game tree in memory.

Game-Playing Performance

The game-playing performance of these algorithms in the Hex game depends on the heuristic function used to evaluate the game states. In the provided code, the heuristic function is implemented in the `estimateScore` method of the `board` object. A well-designed heuristic function can significantly enhance the game-playing performance of the algorithms by providing accurate estimations of the likelihood of winning from a given game state.

Number of Nodes

The number of nodes processed by the algorithms can be used to gauge the efficiency of the Alpha-beta pruning optimization. The global variable `noOfNodes` counts the number of nodes processed. The fewer nodes processed, the more branches have been pruned from the game tree, indicating a more efficient search. However, the exact number of nodes processed will depend on the specific game states encountered and the order in which nodes are processed.

Random Move Generation

The `createPossibleMove` function introduces an element of unpredictability into the game by generating a random move when no other moves are available. This could potentially make the game more challenging for the human player. However, it may also lead to suboptimal moves if there are better moves available that are not considered.

4. CONCLUSION

In conclusion, the performance of the Minimax and Alpha-beta pruning algorithms in the Hex game is influenced by several factors, including the quality of the heuristic function, the order of node visitation, and the specific game states encountered. The Alpha-beta pruning optimization can significantly improve the time efficiency by reducing the number of nodes that need to be processed, while the space efficiency is relatively high due to the depth-first search strategy. The introduction of random move generation adds an element of unpredictability to the game.