Hex Game

Ebrahim Golriz

December 2022

This report details the development of a Hex game implemented in Java. Hex is a two-player abstract strategy board game played on a hexagonal grid. The goal for each player is to form a connected path of their stones linking opposite sides of the board. This project aimed to create a playable version of Hex, incorporating both a text-based interface and a Graphical User Interface (GUI), along with an AI opponent capable of playing the game against a human player. The AI utilizes the Negamax algorithm with a heuristic function based on shortest path calculations to determine its moves. This report will cover the design, implementation, functionality, and testing of the Hex game.

- **Main.java**: This file contains the core game logic, AI implementation, and the text-based game rendering. It manages the game state, player input in the text mode, AI move generation, win condition checks, and the game loop. The main method initiates the game by calling the playgame() function. It also includes functions for:

  - Setting the starting player (setstarter()).
  - Creating the game board as a list of Node objects.
  - Implementing the Negamax algorithm (NegaMax()) for AI decision making.
  - Calculating the heuristic value of a board state (heuristic()) based on shortest paths for both players.
  - Determining shortest paths using a modified Dijkstra-like approach (shortestpathBlue(), shortestpathRed()).
  - Setting up neighbor relationships between nodes (setNeighbors()).
  - Handling player input in text mode (input(), checkInput()).
  - Updating the game state based on player and AI moves (update()).
  - Rendering the game board in text format (render(), toppannel(), printboard(), bottompannel()).
  - Checking for win conditions (winCheck()).
  - Managing the game loop (gameloop()).
  - Utility functions like copying node lists (copylist()) and resetting node heuristics (resetnodesHeuristic(), resetpath()).

- **HexGUI.java**: This file implements the Graphical User Interface for the Hex game using Swing. It provides a visual representation of the hexagonal game board, allows user interaction through button clicks, and displays AI decision-making information. Key components and functionalities include:

    - HexGUI class extending JFrame for the main window.
    - JButton[][] buttons array to represent hexagonal tiles on the board.
    - JLabel statusLabel to display game messages and player turn information.
    - JPanel aiDecisionsPanel (currently hidden) intended for visualizing AI decisions.
    - JPanel[][] sidebarHexPanels and JLabel[][] sidebarHeuristicLabels for displaying a smaller hexagonal grid in the sidebar, visualizing heuristic values.
    - Methods for setting up the GUI (setupGUI()), creating the game board panel (createGameBoard()), creating the sidebar panel (createSidebar()), creating hexagonal buttons (createHexButton()), and creating sidebar hex panels and labels (createSidebarBoard(), createSidebarHexPanel(), createHeuristicLabel()).
    - Game initialization (initializeGame()) and starting the game process (startGame()).
    - Handling player moves through button action listeners (handlePlayerMove()).
    - Implementing the AI move execution using a SwingWorker to prevent UI blocking (aiMove()).
    - Updating the game state visually (updateGameState()) and the sidebar heuristics (updateSidebarHeuristics(), updateSidebarHeuristicsInSwingWorker()).
    - Checking for game over conditions and displaying win/loss messages (checkGameOver()).
    - Disabling buttons after the game ends (disableAllButtons()).
    - The main method to launch the GUI application.

- **Node.java**: This class defines the Node object, representing a single hexagonal cell on the game board. Each Node stores:

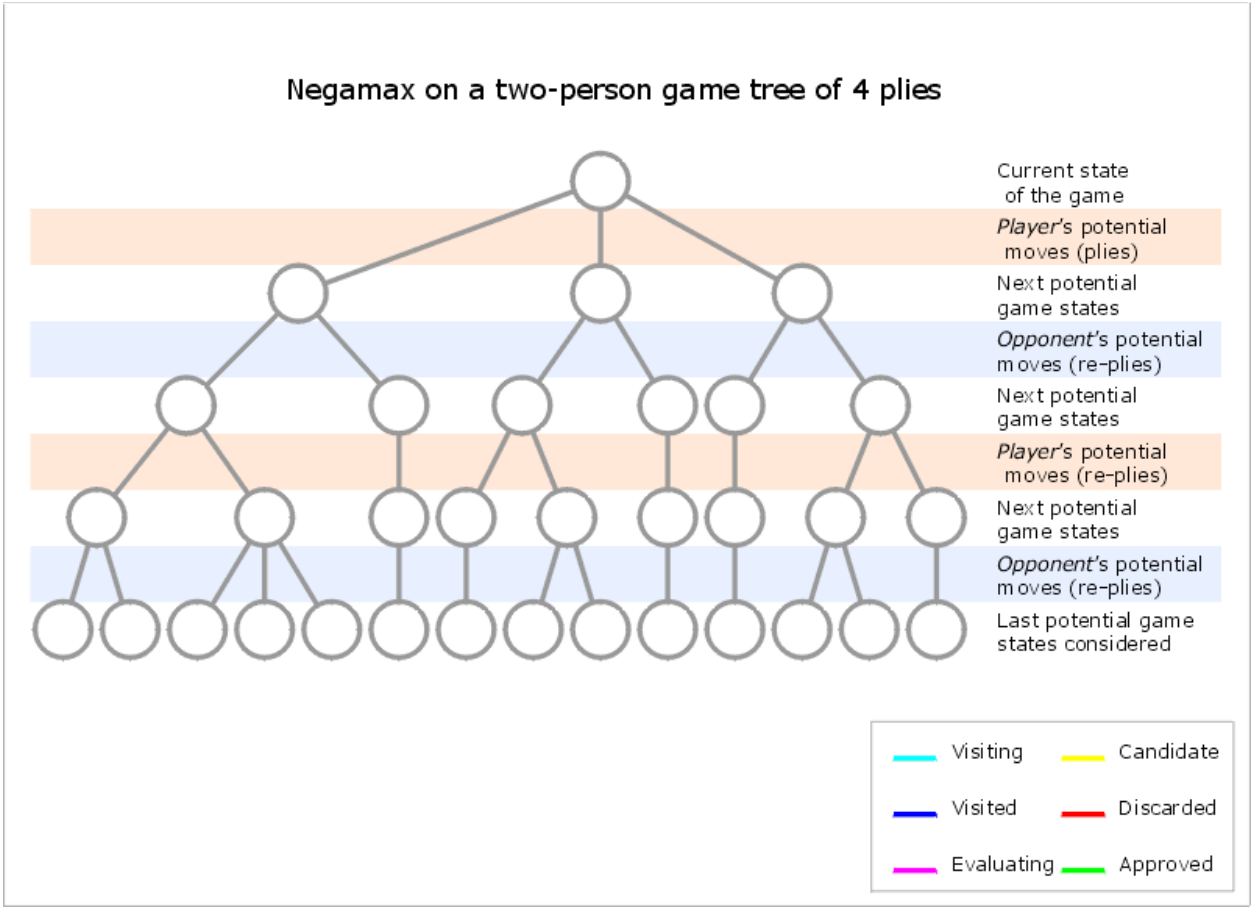
    - char color: Represents the color of the stone placed on the node ('R' for Red, 'B' for Blue, '\0' for empty).
    - int x, int y: Coordinates of the node on the grid (1-indexed).
    - int shortestpathtothisnode: Used in shortest path calculations, initialized to a large value (1000).
    - boolean visited: Flag used during shortest path algorithm to track visited nodes.
    - int heuristicvalue: Stores the heuristic value calculated for this node by the AI.
    - List<Node> neighbors: List of adjacent Node objects.
    - Constructors and getter/setter methods for relevant attributes.

Algorithm and AI Strategy

The AI opponent in this Hex game employs the **Negamax algorithm** with alpha-beta pruning to determine its moves. Negamax is a variant of minimax specifically designed for two-player zero-

sum games. It simplifies the minimax algorithm by always maximizing the score from the perspective of the player whose turn it is, using the negation of the score for the opponent.

- **Negamax Algorithm (NegaMax()):**
  - This function recursively explores the game tree to a fixed depth (currently 3).
  - The base case for recursion is reaching the specified depth, at which point the heuristic value of the current board state is returned.
  - For each possible move (placing a blue stone on an empty node), it generates a child game state and recursively calls NegaMax() with a reduced depth and swapped player turn (indicated by turnBlue parameter, 1 for Blue, -1 for Red).
  - Alpha-beta pruning is implemented to optimize the search by eliminating branches of the game tree that are guaranteed to be worse than already found solutions.
  - The function returns the maximum negated value from the recursive calls, effectively representing the best move for the current player.



Source

- **Heuristic Function (heuristic()):**
  - The heuristic function evaluates the desirability of a board state from Blue's perspective. It aims to estimate Blue's advantage over Red.
  - It calculates the shortest path for both Red and Blue to connect their respective sides of the board.
  - For Blue, the shortest path is calculated from the right side (x=7) to the left side (x=1).
  - For Red, the shortest path is calculated from the top side (y=1) to the bottom side (y=7).
  - The heuristic value is calculated as the difference between Blue's shortest path and Red's shortest path (b - r).
  - A lower shortest path is considered better. Therefore, a positive heuristic value indicates a better state for Blue (shorter path for Blue, potentially longer for Red), and a negative value indicates a better state for Red.
  - The shortest path is calculated using a modified Dijkstra-like algorithm (shortestpathBlue(), shortestpathRed()) that prioritizes paths through stones of the player's color and penalizes paths through empty spaces.
- **Depth of Search:**
  - The AI currently searches to a fixed depth of 3. This depth limits the AI's lookahead and thus its strategic capability. Increasing the depth would improve the AI's strength but also increase computation time.

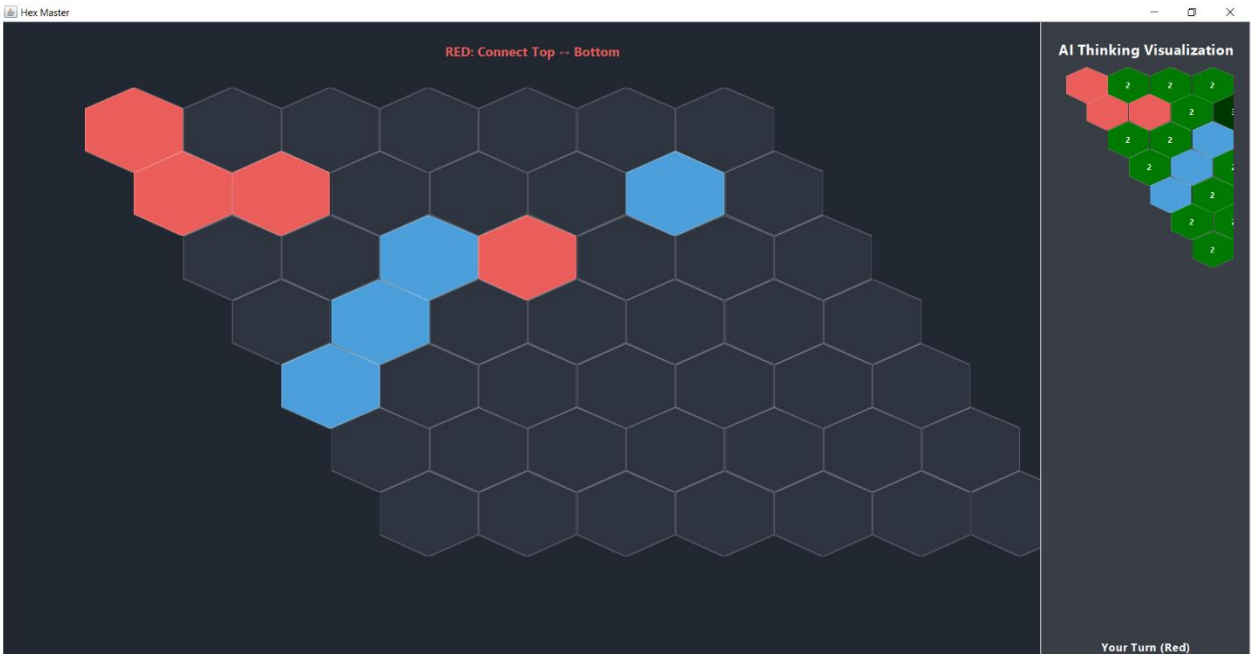
Game Logic and Implementation

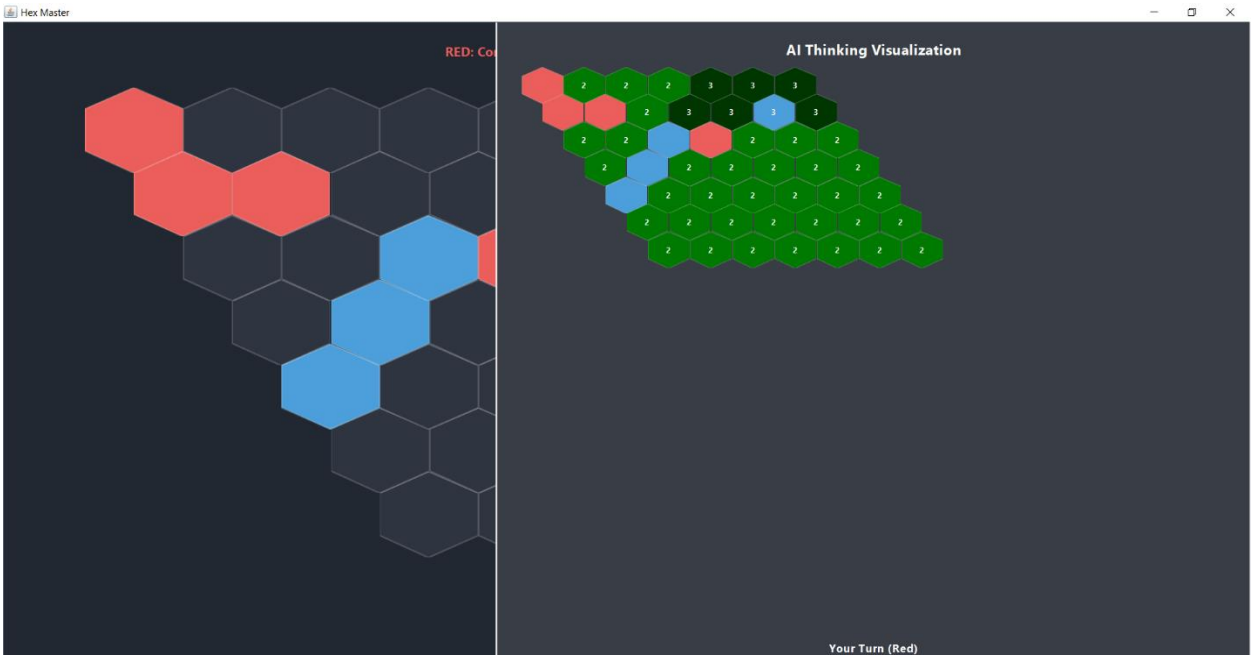
- **Board Initialization:**
  - The game board is represented as a List<Node> nodes in Main.java.
  - In playgame() and initializeGame(), the board is initialized as a 7x7 grid of Node objects.
  - Each Node is created with its x and y coordinates, and initially has no color (represented by the default char color value, which is effectively '\0').
  - Neighbor relationships between nodes are established using the setNeighbors() function. This function iterates through all nodes and determines their neighbors based on hexagonal grid adjacency rules.
- **Player Input and Move Validation:**
  - In text mode (input(), checkInput()), player input is taken as a two-digit integer representing coordinates (e.g., "11" for x=1, y=1).
  - checkInput() parses the input, validates if the coordinates are within the board range, and checks if the selected node is empty.
  - If the move is valid, the node's color is set to 'R' (Red), and Notvalid flag is set to false. Otherwise, Notvalid remains true, and an error message is displayed in the text-based render().
  - In GUI mode (HexGUI.java, handlePlayerMove()), player input is handled by clicking on hexagonal buttons. The handlePlayerMove() function performs similar validation logic as checkInput() and updates the game state accordingly.

- **AI Move Generation:**
  - The update() function in Main.java is responsible for generating the AI's move (Blue player).
  - It iterates through all empty nodes on the board.
  - For each empty node, it temporarily places a Blue stone on it in a copy of the board state (copylist()).
  - It then calls the NegaMax() function to evaluate the heuristic value of this potential move.
  - The heuristic value is assigned to the heuristicvalue attribute of the corresponding Node.
  - After evaluating all empty nodes, the nodes are sorted in descending order based on their heuristic values (nodes.sort(Comparator.comparing(Node::getHeuristicvalue).reversed());).
  - The AI chooses a move from the set of nodes with the highest heuristic value. In case of ties, a random node from the best options is selected to introduce some variability in AI play.
  - The chosen node's color is set to 'B' (Blue).
- **Win Condition Checking (winCheck()):**
  - The winCheck() function determines if either player has won the game.
  - It recalculates the shortest paths for both Red and Blue using shortestpathRed() and shortestpathBlue().
  - If either shortest path is 0, it indicates that a path connecting opposite sides has been formed.
  - If Red's shortest path is 0, win is set to true.
  - If Blue's shortest path is 0, loose is set to true.
  - gameover flag is set to true in either win or loss condition.
- **Game Loop (gameloop()):**
  - The gameloop() function in Main.java controls the game flow in text mode.
  - It calls input() to get player input.
  - If the input is valid and the game is not over, it calls update() to execute the AI's move.
  - It then calls render() to display the updated game board.
  - This loop continues until gameover flag is set to true.

GUI Implementation

The HexGUI.java file provides a user-friendly graphical interface for playing Hex.





- **Swing-based GUI:** The GUI is built using Java Swing components, providing a platform-independent graphical environment.
- **Hexagonal Button Representation:** The game board is visually represented using a grid of hexagonal JButton objects. Custom painting is implemented within the createHexButton() method to draw hexagons for the buttons, creating the characteristic Hex board appearance.
- **Sidebar for AI Decision Visualization:** The right-hand sidebar is designed to provide insights into the AI's decision-making process.
    - A smaller hexagonal grid (sidebarBoardPanel, sidebarHexPanels) is displayed in the sidebar, mirroring the main game board.
    - sidebarHeuristicLabels are used to display the heuristic values calculated by the AI for each empty cell. The labels are dynamically updated after each AI move to visualize the AI's evaluation of different possible moves.
    - Color coding is applied to the sidebar hexagons to visually represent the heuristic values. Green hues indicate positive heuristic values (better for Blue), red hues indicate negative values (better for Red), and the intensity of the color corresponds to the magnitude of the heuristic value.
    - The aiDecisionsPanel was initially intended to list the top AI decisions, but it is currently hidden in the GUI.
- **User Interaction and Game Flow:**
    - The GUI is event-driven. User interaction is primarily through clicking on the hexagonal buttons to place Red stones.
    - Action listeners are attached to each button to handle player moves (handlePlayerMove()).
    - The game flow is managed by the startGame(), handlePlayerMove(), and aiMove() methods.
    - SwingWorker is used for the AI's move calculation (aiMove()) to prevent the GUI from freezing during computation. This ensures a smooth and responsive user experience.
    - Game status messages (player turn, win/loss) are displayed in the statusLabel.
    - JOptionPane dialogs are used to prompt the user to start a new game and to display game over messages.

Testing and Results

Testing Methods

- AI tested via manual playtesting in text and GUI modes.

AI Performance

- Depth-3 AI offers a reasonable challenge but is still simple.

- Displays basic strategy (blocking, path-building) but struggles with long-term planning.
- Shallow search depth (3) limits strategic play.
- Heuristic function is basic and could be improved for better decision-making.

Possible AI Improvements

- **Increase Search Depth:** Enhances strategy but requires optimization.
- **Refine Heuristic Function:** Consider factors like path width, critical connections, and territorial control.
- **Iterative Deepening:** Allows the best move within a time limit.
- **Move Ordering:** Improves efficiency using heuristics for alpha-beta pruning.

Conclusion and Future improvements

AI Enhancements

- Increase search depth, refine heuristics, and implement iterative deepening/move ordering.
- Introduce difficulty levels with varying AI strength.

Game Features

- Different board sizes (e.g., 9x9, 11x11).
- Game saving/loading and undo feature.

Network Play

- Enable online multiplayer support.