COSC 322 Implementation and Report

Team 02 L03:

- Taii Hirano
- Yuki Isomura
- Subaru Sakashita
- Justin Tom

Overview

This project is an implementation of an artificial intelligence (AI) agent for the board game "Game of Amazons". The goal is to create an AI agent that can play the game at a competitive level against human players. Game of Amazons is a two-player game that is played on a 10x10 board. Each player controls four Amazons, which are placed on the board at the beginning of the game. The objective of the game is to either block the opponent's Amazons or capture all of them.

Game Flow Breakdown

- Initialize the game player and get connected with the game client
- Retrieve the information of game state and actions
- Let heuristic find the optimized next move
- Send information about the next move to the client
- Keep playing until one of the players cannot move anymore

Game Structure

Class Extension Tree

```
Object
        ← TimerTask
                ← GameTimer
        ← JFrame
                ← BaseGameGUI
        ← JPanel
                ← AmazonBoard
        ← GameClient
        ← GameMessage
                ← AmazonGameMessage
        ← GameModel
                ← BoardGameModel
        ← GamePlayer
                ← Amazon
                ← Spectator
                ← HumanPlayer
                ← COSC322Test
        ← Tile
                ← Arrow
        ← Board
        ← Moves
        ← Node
        ← Queen
        ← Tree
```

Implementation Details

About Overall Game Flow

The game is controlled majorly by GameClient. When a user runs the program, the player instance is initialized based on the username and password he specified. Then, check that the player instance does not have the game GUI connected and call COSC322. `Go()` to get connected with the game server. In this process, GameClient takes over the game flow. Once the game is started, GameClient sends a message to the player instance. The player checks the message type to track the game flow and update the game state based on the message detail sent.
With updated information of the game state, let heuristic determine the next move reference to the score it calculated. As it is determined, send the information to GameClient and wait for the other player's move.

Structure

Our player implementation runs off of eight classes. Arrow, Board, Moves, Node, Queen, Tile, Tree and COSC322Test being the main class that handles the GameClient. The Arrow, Board, Queen, and Tile classes are our implementation of representing the game to our players. Lastly, the Moves, Node, and Tree classes are used to decide our next move.

COSC322Test

Aside from handling the GameClient, this main class uses the `handleGameStart()` and `handleGameMove()` methods to process and communicate which move the player would like to play. `handleGameStart()` is a method that will initialize a random move to be played on the first move, whereas `handleGameMove()` is a method that will first check if the game has reached a conclusion, before popping the optimal move from the decision tree, and checking if the move has ended the game again. These methods use the Node and Tree classes as representations of the decision tree used to decide which move to play.

Representation of the Game Board (Arrow, Board, Queen, Tile)

Our representation of the game board is in the Board class, which is achieved by using a 2D array of Tiles. Arrow and Queen are subclasses of Tile. In the board array, a blank tile is represented as null, and Queen objects are initialized to their starting positions. The algorithm for deciding which position to shoot an arrow is done within each Queen object. Once an arrow is shot, it is instantiated into the board array with the corresponding position as decided by the

Queen object. The Board class also contains methods used to make move decisions, such as `evaluateBoard()` and `inDanger()`.

Arrow (Inherited from Tile class)

Fields

- `int row`
  - Row number of itself
- `int col`
  - Column number of itself

Methods

- `move()`
  - Returns its position.

Board

Fields

- `Tile[][] board`
  - Initial board
- `Queen[] player`
  - Queen of our side
- `Queen[] opponent`
  - Queen of opponent side
- `Queen selected`
  - Queen that is selected to move
- `ArrayList<ArrayList<Integer>> makeMove`
  - List of positions where inner ArrayList holds the previous Queen position, new Queen position and the Arrow position

Methods

- `updateBoard(ArrayList<Integer> queenPrevPos, ArrayList<Integer> queenNewPos, ArrayList<Integer> arrowPos)`
  - Updates the board with the new positions of the queen and arrow.
- `randomMove()`
  - Generates the random moves from the given list of states.
  - When the list of allies moves, the list of states holds the same score.
  - It is also used to generate the opponent's moves.
- `gameOverCheck(boolean enemy)`
  - Checks whether the state instance is game over state for the selected side.
- `evaluateBoard()`
  - Evaluates the score of the board based on the number of next available positions and whether any queens are close to being trapped.

- `inDanger()`
    - Checks if any queens are close to being trapped.
    - Based on this score the board is adjusted.
- `printMove()`
    - Prints the selected move on the console.

Queen

Fields

- `int prevRow`
    - Previous row number of itself before the moves
- `int prevCol`
    - Previous column number of itself before the moves
- `int arrRow`
    - Previous row number of Arrow before the moves
- `int arrCol`
    - Previous column number of Arrow before the moves
- `boolean opponent`
    - Used to tell if the Queen is opponent or not
- `Moves moves`
    - List of possible moves for this Queen instance

Methods

- `bestArrowShot(Board board)`
    - Returns the best Arrow position from the list of available Arrow positions.
    - Prioritizes shooting arrows at positions adjacent to an enemy queen.

Tile

Fields

- `int row`
    - Row number of itself
- `int col`
    - Column number of itself

Methods

- Getters and setters

Decision-Making Process (Moves, Node, Tree)

In order to decide the next move, the program first retrieves information on available moves each queen can make. This calculation is done in the `getMoves()` function defined inside the Moves class. The function checks for the available moves in each direction, up, down, left, right, top-right, top-left, down-right, and down-left, and stores the coordinates in an ArrayList.

Within the `handleGameMove()` function, the tree is grown based on the previous move received from the GameClient, and that tree is used to make a decision on the move to play.

Our group decided to use the number of available moves in a given board state as the heuristic function. The `treeSearch()` function is the function used to decide which move to play. The function first takes the best move as decided by the heuristic function, checks if the board state of that move would put any queen in danger, and if it does, dissuade the player from making that move by subtracting the score of that move by 1000, if it doesn't, it returns that move.

Moves

Fields

- `List<ArrayList<Integer>> moves`
    - List of possible moves where inner ArrayList holds the index of row and column
- `List<ArrayList<Integer>> arrowShots`
    - List of possible Arrow position where inner ArrayList holds the index of row and column

Methods

- `getMoves(Board board, Queen queen)`
    - Generates all possible moves that a Queen can make on a given Board.
- `availableArrows(Board board, Queen queen)`
    - Generates all the available arrow shot positions for a given Queen on a given Board.

Node

Fields

- `Board board`
    - Current state of the Board
- `ArrayList<Node> children`
    - Children Nodes of itself
- `Node parent`
    - Parent Node of itself
- `int bf = 10`
    - Branching factor
- `int gameOver`
    - Initialized using the function gameOverCheck from Board
    - Used to tell if the state of the node is game over or not

Methods

- Getters and setters

Tree

Fields

- `Node root`
    - root Node
- `int depth`
    - depth of the Tree

Methods

- `growTree(Node curr, int toDepth)`
    - Generates a tree of nodes by recursively adding child nodes to the current node by BFS where the branching factor is 10 until the specified depth is reached.
    - For each child node, a random move is generated for either the enemy or the player, depending on the parity of the depth of the node.
    - If a generated child node is already a child of the current node, it is skipped to avoid duplicates.
    - time complexity is O(10^toDepth)
- `addChild(Node parent, Node child)`
    - Add a new Node and set the parent Node.
- `findPath(Node curr)`
    - Returns a Stack of Nodes representing the path from the current node to a leaf node in a tree.
    - It performs a tree search by repeatedly calling the `treeSearch()` method on the current node until it reaches a leaf node or a node with a game over state.
    - Then it backtracks from the leaf node to the current node by adding each node to the stack and setting the current node to its parent until the input node is reached.
    - If any node with game over state has been found, the path to that node will be returned.
- `treeSearch(Node curr)`
    - Performs a search on the current node's children and returns the best child node according to a heuristic evaluation function.
    - If a child node has a game over status of 1, it is immediately selected as the best node.
    - If a child node's board is in a dangerous state, its evaluation value is decreased by 1000.
    - The best child node is chosen based on the evaluation function, which takes into account the current node's evaluation value and the evaluation value of the child node.

Task Separation

- Taii:
    - Implement the Node
    - Implement the heuristic function
- Yuki:
    - Implement the Board
    - Implement the COSC322Test
    - Implement the heuristic function
- Subaru:
    - Implement the Tile
    - Implement the Moves
    - Keep track of scores
- Justin:
    - Implement the Arrow
    - Implement the Queen
    - Implement the Tree

Issues and Difficulties

- Unexpected cases while setting up the MAVEN Project.
- This project is not fit for Mac users. Some members are Mac users so it is quite difficult for them to work at home.
- Lack of information in the main method of the GameClient class.
- Lack of information on the rules of the tournament. Some opponent groups had heuristics that took more than 20 seconds for each move.
- Due to the limitation in time to operate the calculation, it is not possible to set the branching factor and maximum depth to explore as big numbers.