Magnetive Cave

Jonathan Bergeron & Marc-André Faucher

December 7, 2012

1 Program Description

Our program is written in the Python scripting language. We used version 3.3 to write it. In order to run it, you will need the Python interpreter that can be downloaded for free from `http://www.python.org`. Once installed, the user should configure environment variables if running on Windows. Afterwards, the user can start the application by going in the directory where the files in which the files were extracted and type:

```
python Main.py
```

Our program spans different files so that it remains organized and easy to read. Those files are the following: `Main.py`, `Constants.py`, `GameBoard.py`, `Minimax.py`, `Node.py`, `Tree.py` and last, we used `Test.py` to facilitate debugging and testing. In this report, we will omit describing the last because it not an integral component of the application.

1.1 Main.py

This file contains the entry point of the application. It contains a series of instructions that compose the game's initialization and the game's main loop. It contains one other method.

Methods

- `GetCoord()`: Used to prompt the human player to enter coordinates.

1.2 Constants.py

This file contains many of the constants used in the program such as the game board's height and width, the max depth for the tree, the symbols used for player one and player two and other constants for the A.I. such as the bridge size (used to check the state of the game board) and various values for weights used to make decisions. It contains no methods.

1.3 Node.py

The underlying data structure that we used to create our A.I. is a tree composed of nodes. In this file, we can find the implementation of the node class. A node keeps track of a representation of the state of the game (Gameboard), a reference to its parent and to its children. Apart from the constructor, it has one other method.

Methods

- `IsLeaf()`: Returns true if the node is a leaf node. If it returns false, then the node is an internal node.

1.4 Tree.py

This file contains the implementation of the tree class. This class acts as a container for nodes. Appropriately, its methods are related to manipulating the nodes that compose it. At creation, it will initialize an empty root node.

Methods

- `AddNode(parentNode)`: Adds a node without having searching the tree.
- `SetRoot(node)`: Sets a new root node for the tree.
- `GetNode(currentBoard)`: Returns the node which corresponds to the player's move. When a player's turn passes, the move he played is kept on a gameboard object.
- `GetNodeByMove(move, player)`: Generates a board from a move object and a node object and calls `GetNode(currentBoard)`.
- `CountNodes()`: Used for testing. Counts nodes.
- `GetLeaves(node)`: Recursive method which returns all leaves starting at a specified node or root if the argument is not specified.
- `GenerateDepths(player, depth)`: Generates levels to the tree ADT.

1.5 Minimax.py

This file contains the minimax implementation of our game that uses alpha beta pruning. Using but a single call to one method, the A.I. player will get its next best possible move to play.

Methods

- `Heuristic(player)`: Returns the heuristic score for a specific node evaluated by a method from the `GameBoard` class.
- `Minimax(tree, player, depth)`: Will start the minimax algorithm and once the stack of calls empties out, will return the path in the tree that the A.I. should opt for.
- `Maxi(node, player, depth, alpha, beta)`: Will return the path to the node maximizing the A.I.'s ingame situation.
- `Mini(node, player, depth, alpha, beta)`: Will return the path to the node minimizing the opponent's ingame situation.

1.6 GameBoard.py

This file contains the GameBoard class definition. It contains methods to accomplish many differently natured tasks. Tasks range from printing the game board onto the screen, enforcing the game's rules, verifying if a player won the game, evaluating the heuristic for the game's current state and other methods to help the A.I.

Methods

- `Print()`: Prints the game board to the output console
- `PlaceSymbol(player, column, row)`: Places a symbol for the appropriate player on a given board cell.
- `CheckWinner()`: This method will call different various other methods in order to check if the game was won or if the game closed at a draw. Returns 1 if player one won, 2 if player two won, 0 if it's a draw and -1 if none of the previous events happened.
- `CheckWinColumn(column, row)`: Called by `CheckWinner()`. Checks for a vertical win starting at a given position.
- `CheckWinRow(column, row)`: Called by `CheckWinner()`. Checks for a horizontal win starting at a given position.
- `CheckWinDiagDown(column, row)`: Called by `CheckWinner()`. Checks for a diagonal (with upper left extremity) win starting at a given position.
- `CheckWinDiagUp(column, row)`: Called by `CheckWinner()`. Checks for a diagonal (with upper right extremity) win starting at a given position.
- `GetPlayer(column, row)`: Returns the player number corresponding to symbol in a game board's cell.
- `IsDraw()`: Used to determine if there is a cell that hasn't been played yet.

- `WeightedH(player)`: Uses other methods to determine a weighted heuristic score for a player for a the game board.
- `CheckHColumn(column, row)`: Called by `WeightedH()`. This method calculates the heuristic score of player symbol placed in a vertical manner.
- `CheckHRow(column, row)`: Called by `WeightedH()`. This method calculates the heuristic score of player symbol placed in a horizontal manner.
- `CheckHDiagDown(column, row)`: Called by `WeightedH()`. This method calculates the heuristic score of player symbol placed in a downwards diagonal (upper left extremity) manner.
- `CheckHDiagUp(column, row)`: Called by `WeightedH()`. This method calculates the heuristic score of player symbol placed in a upwards diagonal (upper right extremity) manner.
- `ApplyWeights(control, size)`: Applies a corresponding value to a heuristic score found to alter its impact on the final score. This let's the A.I. prioritize certain possible moves.
- `IsLegal(column, row)`: Returns if a specified move is permitted or not.
- `IsOccupied(column, row)`: Checks if space is already occupied, returns true if occupied else, returns false
- `IsOutOfBounds(column, row)`: Checks that input corresponds to a cell inside the Game Board
- `IsStacked( column, row)`: Verifies if the space chosen is stacked on another occupied space or the left or right wall of the game board.
- `LetterToInt(letter)`: Used to convert a character to an integer value that can be used in the 2D array
- `IntToLetter(intchar)`: Used to convert an integer value to a character used by the Game Board
- `ClearOutBoard()`: Resets the game board to an "empty" state.
- `GetNextAvailablePlays()`: Returns a list of possible ingame moves that a player can during his or her turn.
- `PopulateForTest(setting)`: This method was implemented for testing winning conditions. It has no relevance in the final program.

2 Heuristic Function

The A.I. uses the Minimax algorithm, with alpha-beta pruning. In order to respect the limit of 3 seconds per move, the depth of the game tree is fixed at 3. The heuristic calculates the total number of possible lines of 5 cells available to each player, by evaluating all candidate lines (see Figure 1).

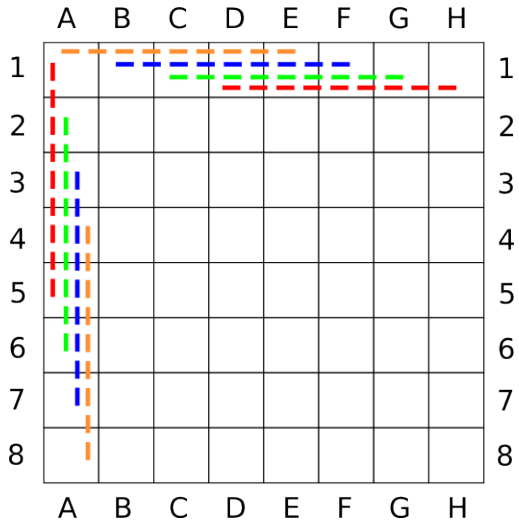

Figure 1: An example of candidate lines, vertically on the first column, and horizontally on the first row. Diagonal lines are also evaluated.

A candidate is rejected for one of two reasons:

1. a line contains bricks from both players, since the line is considered blocked;
2. a line is completely empty, since it is free for both players and it should not contribute to the heuristic;

All other lines are weighted according to the number of bricks the controlling player has placed on that line. The final heuristic function is the weighted sum of all possible lines.
By intuition, the weight of a line is worth much more if it contains even a single brick more. For this reason, we opted for an exponential weight function. Assuming the AI is playing player 1, the heuristic function is defined:

$$H(B) = \sum_{\text{lines} \in B} w_{p,i}$$

$$w_{p,i} = \begin{cases} n^i & \text{if } p = 1 \\ -n^i & \text{if } p = 2 \end{cases}$$

Where:

p = controlling player

i = number of bricks

This function is defined for $i = \{1, 2, 3, 4\}$. It is not defined for $i = 5$, since the heuristic for a winning board is automatically ± 1000. The variable n was initially set to 2 but, after some testing, this was changed to 4. This seemed to perform much better, since the A.I. would consider a line of 4 to be much more of a threat then even several lines of 3.

3 Tournament Results & Discussion

Our tournament record was: 2 won; 2 draw; and 1 lost.
The game seemed to be somewhat skewed towards the starting player. In fact, until meeting our match against the winning player, we won every game in which our program started. We lost one game due to a bug which allowed the opponent to win simply by placing bricks on squares A1 to A5. We were not able to find out why this occurred.
We opted for a more informed heuristic rather than implementing the program in a more optimised programming language. This was a successful strategy up to a point, since it was able anticipate winning moves before they were explored, and play accordingly.
The main weakness of our program is the speed of our implementation. Using alpha-beta pruning, allowed the Minimax Tree to explore up to a depth of 3. Using a depth of 4 was only possible very late in the game, when the number of possible moves was more limited. In the end, we lost against the winning player because he was using threads to explore the gamespace, and calculating the heuristic at variables depths. This allowed the program to run Minimax up to depths between 5 and 10.