CONCORDIA UNIVERSITY
ARTIFICIAL INTELLIGENCE

MAGNETIC CAVE

JEFF HOW [redacted]
ADDISON RODOMISTA [redacted]

DECEMBER 10TH, 2012
LEILA KOSSEIM

FINAL REPORT

PROGRAM DESCRIPTION

Our version of Magnetic Cave was written in Java using the Eclipse IDE. In order to run the program, simply load the project into any Java compatible IDE and compile. The user will immediately be prompted with a welcome screen.

1. **Single Player Game**

   If the user wishes to play against an AI opponent, they must enter 1 in the main menu. They will then be prompted with which color they wish to be. Black will be represented by an X, and will get the first move. White is represented by an O, and will move second.

2. **Multiplayer Game**

   The user can also choose to play a multiplayer game by selecting 2 from the main menu. Immediately following the selection of this mode, the game will begin and the first player must select their move.

CLASS STRUCTURES

Our program is composed of four classes.

PositionState

The foundation of our logic, PositionState is an enum with the values EMPTY, PLYR1 and PLYR2. It represents a lone possible square (or tile), which can either be empty or owned by player 1 or 2.

GameState

Represents the game board with a 2D of position states. Also, this class contains the methods used to calculate the score for our heuristic. (See Heuristic Section for further details).

GameState: Creates a new empty board

updatePosition: Updates the PositionState of a particular tile on the board.

retractMove: Uses a GameMove to retract a previously made move.

toString: Draws game board so screen

getScore: Sums the horizontal, vertical, and diagonal scores

horScore: Sums the number of adjacent same colored tiles

rowScore: Calculates the score for an entire row

horGrpScore: Calculates the score for individual 5-squared group horizontally

vertScore: Sums the number of vertically adjacent same colored tiles

colScore: Calculates the score of an individual column

` **vertGrpScore**: Calculates score of 5-square group vertically

diagScore: Calculates the number of adjacent diagonal same colored tiles

SEGrpScore: Score of south east 5 squared group

SWGrpScore: Score of south west 5 squared group

GameMove

Encapsulates the information required to make a distinct move. Specifically speaking, this includes the row, column and PositionState. GameMove objects are often used as input for functions, such as updatePosition and retractMove in the GameState class.

GameMove: Stores row, column and state information.

toString: Prints the GameMove on console. Used as a helper function while debugging.

Interface

This class makes the game runnable. It prints out all the menus, allows user input, validates player moves, and contains the logic for player and AI turns.

Interface: Main Constructor, sets the current gameState.

validateInput: Specifies whether a given input is within a specified range.

getCol: Given a specified character, returns an integer representation.

getRow: Takes in an ASCII value of a character, and identifies its array placement.

validatePlayerMove: Given a user input, checks to see whether or not the move is a valid entry.

printPlayerTurn: Used for the UI, prints which players turn it is.

welcomeScreen: Handles intro screen and gameplay type selection.

playerTurn: Prompts the human player for input regarding their move.

aiTurn: Uses the minimax function from the Game class to calculate and

execute the best possible next move.

playGame: Game loop that allows the game to continue playing until one of the player has won the game.

Game

main: Main gameloop, initializes the interface

minimax: Static method. Given a current GameState it uses getMoves to obtain all potential legal GameMoves. Then it calculates and chooses the best heuristic score for all potential moves. Depending on argument given for depth, may use recursion to go look a few turns ahead deeper down the tree (metaphorically speaking) of GameStates.

getMoves: Used as a helper function for the minimax method. Gets the possible legal GameMoves given the current game state and current player.

HEURISTIC

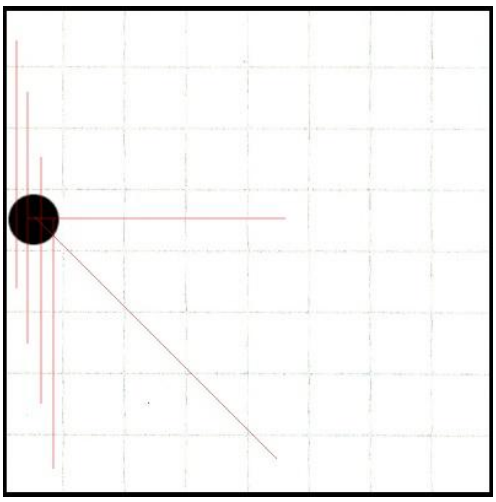
To begin our initial understanding of the heuristic, let us introduce a new concept. A "group" is a row of five consecutive squares. A single group can be horizontal, vertical or diagonal. A player can "own" a group if at least one of his tokens occupies one or more squares in the group AND no tokens of the opposing player lie within the group. In other words, only one player can occupy a single group. If both players have tokens within a group, neither player can own it. There are ninety-six potential groups on the board: 32 vertical, 32 horizontal and 32 diagonal. For the vertical ones, there are four groups per column by eight total columns. Similarly, for the horizontal ones, there are four groups per row by eight total rows. The number of diagonal groups was obtained by counting them manually. Note that one square can be a part of multiple groups.

The scoring system for a game state works as follows:

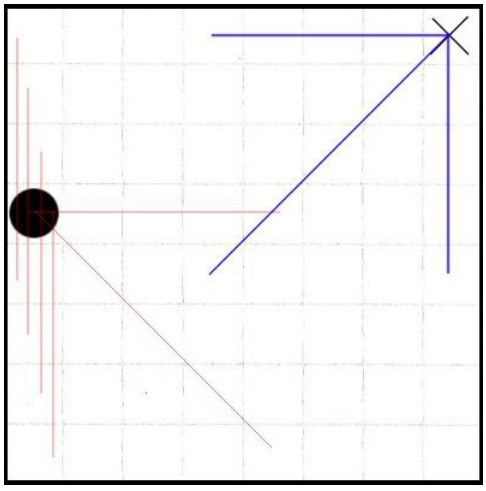
- Own a group: 1 point
- Two tokens in an owned group: 1 points
- Three tokens in an owned group: 2 points
- Four tokens in an owned group: 3 points
- Own entire group: 10000 points
- For Player 2, the above points would be negative.

The idea behind this scoring system is that we wanted to emphasize having tokens in a row but not at the cost of losing giving away ownership of potentially valuable tiles that relate to multiple free groups. Please refer to images below for examples.

Example 1: O owns 6 groups → Score: 6



Example 2: O owns 6 groups, X owns 3 groups → Score: 3

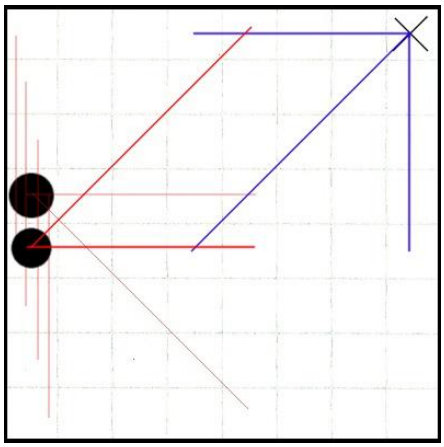


Example 3 (next page): O owns 8 groups = 8 points

O has two squares in a row in 4 groups = 4 points

X owns 3 groups = -3 points

Score → 9

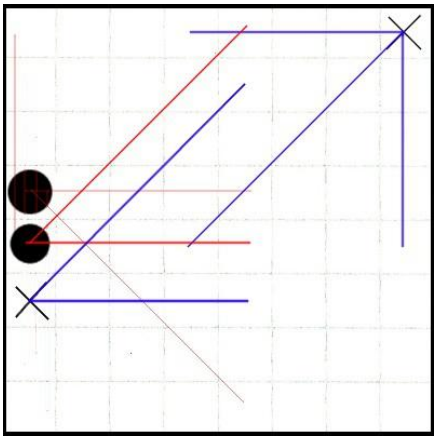


Example 4: O owns 4 groups = 4 points

O has 2 squares in a row in 1 group = 1 point

X has 5 groups = -5 points

Score = 0



TOURNAMENT RESULTS

While we were unable to attend the in lab tournament ourselves, we independently tested our program against our the tournament runner-ups. To emulate a tournament setting, we pitted our AI against theirs in 3 rounds.

Notes: *Our project was run using the computers in Lab H817 using Netbeans and Windows 7. Their project was run using the same computers, but executed in terminal on Linux.*

Round 1: Their AI First

```
Player 2 chose move: G3
  A  B  C  D  E  F  G  H
1 -  -  -  -  -  -  -  -
2 -  -  -  -  -  -  O  X
3 O  X  X  O  O  X  O  -
4 X  O  X  O  O  X  O  O
5 X  O  X  O  X  O  O  O
6 X  X  O  X  X  X  O  -
7 O  X  O  X  -  -  O  X
8 X  X  X  O  -  -  -  -

--------------------------------------
Player 2 won! Thank you for playing!
--------------------------------------
```

Winner: Ours

Round 2: Our AI First

```
Took 0.08 seconds to calculate move
Player 1 chose move: A6
  A  B  C  D  E  F  G  H
1 X  -  -  O  X  O  X  X
2 -  -  -  O  X  X  O  X
3 X  X  O  X  O  X  O  X
4 -  O  X  X  O  X  O  O
5 O  X  X  X  X  O  X  O
6 X  -  O  X  O  O  O  O
7 -  O  X  O  X  O  O  O
8 X  O  X  O  -  -  -  X

--------------------------------------
Player 1 won! Thank you for playing!
--------------------------------------
```

Winner: Ours

Round 3: Their AI First

```
Took 0.13 seconds to calculate move
Player 2 chose move: D3
  A  B  C  D  E  F  G  H
1 -  -  -  -  -  -  -  -
2 O  X  X  O  -  -  -  -
3 X  O  -  O  X  O  X  O
4 X  O  O  O  X  O  X  X
5 X  O  O  O  O  X  O  X
6 O  X  -  O  X  X  X  X
7 X  -  -  -  -  -  -  X
8 -  -  -  -  -  -  O  O

--------------------------------------
Player 2 won! Thank you for playing!
--------------------------------------
```

Winner: Ours

Abiding by the honor system, we were very pleased to see that our AI proved victorious in all 3 rounds, and were even more pleased at how quickly it was able to determine its next move. There was not an instance in all 3 matches that required more than 2 seconds. Judging by these results, we would have loved to pit our program against the class victors to see how it held up against the reigning champions.