Intelligent Search & Games

Omega

Nikoleta Acheila i6202817

1. Introduction

1.1 Omega

Omega is a multiplayer strategy board game (2 to 4 players) created by Néstor Romeral Andrés. In this game, the objective of each player it to maximize their individual score, while trying to minimize the scores of the opponents. The Omega board is shaped as a canonical hexagon which comprises of smaller canonical hexagons, while the size of the board can vary, depending on the desirable level of difficulty. Each player is denoted by a distinct color of stones in the board. During the game, every player has to add a stone of their own color in the board , as well as a stone of everyone of their opponents' color. The game ends when the board is full so as it is no longer possible for every participant to play their moves. The score of each player depends on the number of different groups of their color formed on the board and it is calculated by multiplying the sizes of each distinct group. In that way, the general strategy of each player is to form as many different groups of their color as possible, and at the same time to try to connect the different groups of their opponents.

1.2 Minimax, Alpha-Beta Pruning & Negamax

The Minimax algorithm is a search method designed for finding the best move that the current player of the game can do to maximize their chances of winning. The algorithm generates the next legal moves and recursively runs through all possible next rounds, where the number of rounds searched depends on the set depth of the search. At each round, the player's objective is to maximize their score, while the opponent tries to minimize it. Given one possible move, the next positions of a board can be represented in the form of a tree where the current state considered is a node, and the states where this move was chosen by the player and then the opponent played their move are the successor nodes. When the maximum depth is reached the algorithm evaluates the eligibility of each state considered in that depth based on an evaluation function.
This process is significantly time consuming considering that in an empty board of 61 hexagons one player can place their two stones in 61*60 possible ways, which means that 3,660 different game states have to be evaluated, and that is only if the search depth is equal to 1, as the number of possible states to be searched grows exponentially with the increase of the depth. For this reason, the Minimax algorithm is enhanced with the Alpha-Beta pruning method which provides a way to cut down the nodes (states) to be searched by computing whether a branch is worth exploring in terms of returning a better value than the one already ensured by the previously evaluated states. If that is not the case, then a pruning of the current branch occurs and the search continues with the next one.
The simplest way to implement Minimax with Alpha-Beta pruning is using the NegaMax algorithm. NegaMax follows the same principles as the ones described above, while avoiding to have to make a distinction amongst maximizing and minimizing rounds, by negating the values obtained by the evaluation function so that the algorithm only needs to maximize throughout the game.

1.3 Implementation

The current program is developed in Java 1.8.0_181 using Java's Swing and AWT libraries for the implementation of the GUI. Here, Omega is a two player game consisting of manual and/or user payers , where manual players add their moves manually to the GUI using the mouse, while computer players' moves are generated at each round and sequentially are added to the game board.

2. The Game

2.1 Menu

At initialization, a series of pop up options appear, where the user can select their preferences about the upcoming game. First, the user can select the size of the board by choosing the preferred side size out of the options provided (the options can be altered in the Menu class). Then, they can choose the preferred type of game out of the following options : User vs User, User vs PC, PC vs PC, where PC denotes an AI player. If the User vs Pc option is selected, then the user will have to choose whether the manual or the AI player will play first. If the user closes one (ore more) of the pop up selection boxes then the default option for this box is selected, which is board of size = 5, User vs User game and manual player plays first accordingly.

2.2 Game Board

The Omega Game Board is created using the java.awt and java.swing packages in the GameBoard class by the Init_HexGrid method and consists of individual hexagons placed is such a way that a larger hexagon is formulated as can be seen in the following figures. The individual hexagons are created one by one in sequence, starting from the first one on the upper left side of the board and continuing with the one next to it. Each individual hexagon is an object of the Hexagon class and it is defined based on the coordinates of its center, while the coordinates of its vertices are computed in the updatePoints method. The color of a hexagon is determined by the hasBeenSelectedBy variable of the Hexagon class which is by default -1 indicating that this hexagon has not been selected yet and should be painted green. Once a row of hexagons is completed, the same process continues with the next row, following constraints that reassure the final hexagonal shape of the board, until the game board is created.

Figure 1: Board of a 3-size game

Figure 2: Board of a 5-size game

2.3 Hexagons & Coordinates

The Hexagon object is created in the Hexagon class and it is defined by its center, its current color and its serial number. The serial numbers of the hexagon start from 0 denoting the most upper left hexagon of the board until n-1, where n is the total amount of hexagons in the board, with the most lower right hexagon of the board. The discernment of hexagons based on their serial number is easier from the developer's point of view, because it is easier to mentally place it on the board and thus create simplest structures such as list or arrays that each position corresponds with the hexagon of that number.
However this method of describing the hexagons makes some tasks much more complex and consequently the need for another system arose. The Coordinates class provides a method of describing the place of a hexagon by its cube coordinates. This system consists of three axes x, y and z and denotes the plane where x+y+z=0. At the center of the board we have that x=y=z=0, while the absolute values of x, y, z increase when moving away from the center and towards the edges, but always conforming with the equation of the plain. The use of this system helped when calculating the neighbors of a single hexagon, which was very complex if at all possible using only the serial numbers.

2.3 Players & Game Types

There are three types of players that are implemented in this project: manual player, AI player and random player. The manual player provides their input through mouse clicks which are passed to the program throughout a mouse listener. The AI player, which is described in detail later on, follows two different strategies, depending at the point of the game their in and its moves are generated and drawn on the board when so. The random player was implemented just for testing purposes as a RandomMove method which randomly places two stones one of each color to the board.
Depending on the type of game (i.e. User vs PC, User vs User etc.) the way those players are implemented differs. In a User vs User game the action listener handles a mouse click by checking if a move is eligible (a hexagon of the board was selected and this hexagon has not been previously selected by any player) and calling on the nextMove method. In this method the colors and turns are swapped to determine the next player and the order of the colors of the stones they will insert (each player first puts a stone of their own color and then a stone of the opponent's color) while there are no other background calculations throughout the game. In a User vs PC game, the user input is handled in the same way, but this time after the second turn of the manual player (when they put the opponents stone) causes an AIMove which consists of both turns of the AI player (two moves in total) and is described in the following section. In a PC vs PC mode the whole game is played out inside the GameBoard constructor by sequentially calling the AIMove function. At the end of ever game, the EndofGame method is called, that pops up a box declaring the winner (or draw) and providing the score of each player.

2.4 AI

The AI player is implemented through the AIMove method. For the first third of a game, the strategy followed by the AI player is placing their color's stones on the border of the board and the opponent's stones close to the center. This strategy not only saves a lot of time, as the AI moves are considerably much slower to compute, but also approximates the strategy followed in the first part of the original game. This way, the AI player attempts to increase the distance between the different groups that they are going to formulate later in the game, while concentrating the opponent's stones close to the center, which will facilitate the connection of the opponent's groups later on. When the number of moves played exceeds the number of moves corresponding to first third of the game, the moves are generated in the AI class instead.

This class receives as inputs a list of coordinates of all the hexagons in the board, a list of the color that occupies each hexagon and the current player for whom the best move must be computed. This is done by calling the AlphaBeta_Negamax function with iterative deepening.

Iterative deepening is a method where the depth of the search algorithm is not fixed so as to optimize the search. An initial depth is set to 3 (can be altered by the initial_depth variable) and the search runs first for this depth. Then, we set a timeout, which is currently set to 10000 milliseconds, and we rerun the search with an increasing depth until the running time reaches the timeout. As a result, the number of the increase we can do to the depth before reaching the timeout depends on the available moves of the game, which is correlated both with the size of the board and the state that the game is in when the method is called. Nonetheless, the maximum depth reached at the earliest stages of the game is smaller, which makes sense in such a complex game because we couldn't reach and evaluate a very later state of the game that would provide significant information for the next possible move, and it gets larger as we approximate the end of the game providing better quality moves.

As for the AlphaBeta_Negamax method, the implementation is based on the ISG lecture slides with mild modifications. The algorithm considers the possible moves recurrently, by placing stones in the empty cells playing out the following rounds of the game. When the depth equals to zero (it has gone in the maximum depth) or there are no more available moves, it returns a value for the state of the game studied that is computed with the evaluation function. This value, is compared to the previous values assessed to get the best possible move, and also the alpha and beta bound to determine whether the current branch is worth searching. When a child node of the root is reached the best calculated value so far is saved in the blackmove and whitemove variables. At the end of the function, those moves are returned to the AI constructor and are saved in the globalmove variables to ensure that those are the actual moves computed and have not been returned during a timeout in the middle of the search.

2.5 Score

The score of each player is computed in the ScoreCalculator class using the union method. In the beginning, the program initializes a parent array and a size array, where each position of those arrays denotes one hexagon of the board based on its serial number. The size array stores the size of the group that a hexagon is considered to belong, while the parent array stores the serial number of the hexagon that is the "parent" of that group, meaning that all hexagons that belong to the same group must point to the same parent hexagon. Subsequently, the algorithm goes through each hexagon of the board and creates a list of neighboring hexagons for that hexagon. Then, it iterates through the list of neighbors (myNeighbours) and unites the hexagons of the same color to the current hexagon using the Unite method. This method works in the following way:
-Two hexagons are inserted as input (the current hexagon considered and its current neighbor) that have the same color stone on them
-The hexagon with the smallest rank (size value) changes its size and its parent values to the one of the hexagons with the largest rank

-The parents of all other hexagons that previously pointed to that parent are updated to the new parent
-The new size of the group is computed

Next, two sets of lists are created with the parents and sizes of the hexagons of the two colors (players) and the size of each distinct group is calculated and added to a list. Finally, the two scores are computed by multiplying the sizes of groups of the same color for each player (score1 for the white player and score2 for the black player).

2.6 Evaluation function

The Evaluation function is called when a 0 depth is reached in the AlphaBeta_NegaMax function. It receives as inputs the list of colors capturing the hexagons (capturebyList), the list of the coordinates and the current player for whom we optimize. The value of a state of the game is computed in the following way:

Evaluation= c*[(score1-score2) + 4*(goodgroups1-goodgroups2)+2*(badgroups2-badgroups1)]

,where:

P is 1 for White Playerand 2 for Black Player
scoreP=score of player P
goodgroupsP=amount of groups of player P that have either two or three members (stones0
badgroupsP=amount of groups of player P that have 1 or >3 members
c=1 when optimizing for White Player and c=-1 when optimizing for Black Player

This evaluation function was chosen to indicate the importance of the number of stones in a group as the creator of the games suggest that 2 or 3 is the optimal size of each group.

3 Experiments & Results

It is interesting to study the effect of who plays first with respect to the outcome of the game. In a game board of size 3, 20 games were played between an AI player and a Random player. For the first 10 games the AI player played first, while in the next 10 games the sides were the opposite.

Board Size=3	**AI vs Random**		**Random vs AI**	
Average Score	13.45	8.36	9.72	8.72
Wins	8	1	5	2
Maximum Score	18	12	16	12
Minimum Score	5	4	7	6

In the games were the AI played first we can see that it did significantly good, when 8/10 game resulted in a win. The overall values scored were higher than the ones scored by the random player and only one game resulted in a draw. When the sides were exchanged, the results were not so good for the AI player with them winning only 2/10 game while three games ended in a draw. It can be seen that in this case, the difference in the values scores were not very high, so we can conclude that indeed the player that plays first has an advantage over the game.