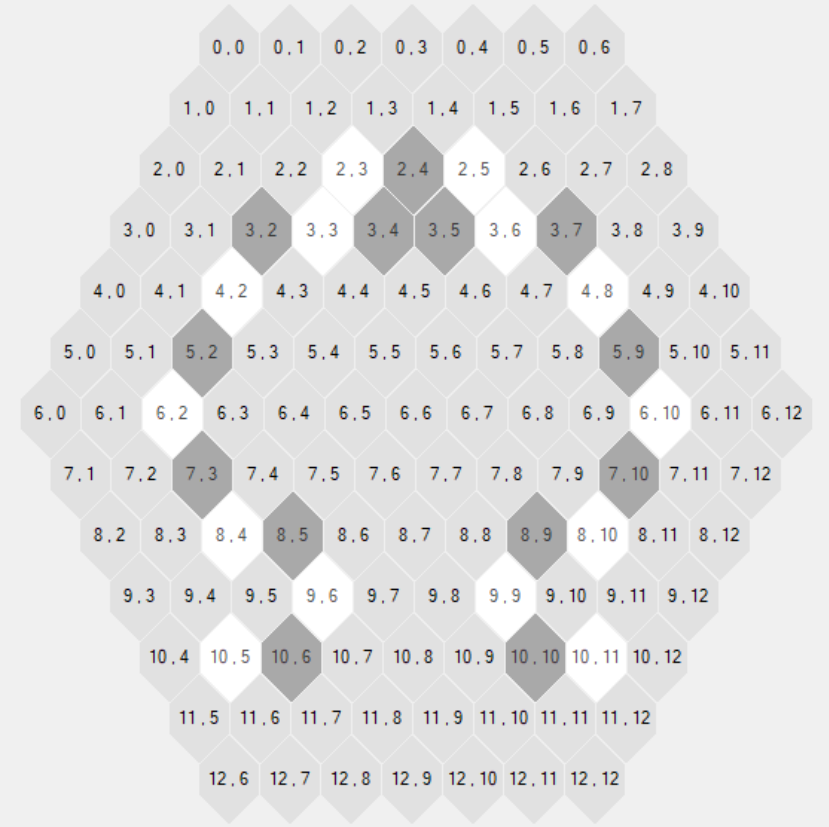
Maastricht University

Department of DKE

Intelligent Searches and Games

Omega implementation in C#



Student name: Aron Zoltan Varadi (i6202342)

About Omega

Omega is a simple board game, developed by Néstor Romeral Andrés. My software is made for the simplest version of the game, therefore here I describe only that.

The game is played on a board made of hexagonal fields, and the shape of the board is also a hexagon. The game can be played by two players, white (player 1) and black (player 2). The players take turns, in each turn, player one places two stones on the board, one black and one white. Then the second player does the same. This continues, until there is not enough empty space for both players to place stones. The score is then calculated by multiplying the sizes of the groups of the same colours.

General approach

I wrote my program in C# using Microsoft Visual Studio. I generate the board from "Field" objects, which is based on the button class to make it clickable. The Fields' shape are also modified to be hexagonal, then they are added to the user interface to create the board. On click events, they change their background color depending of the number of the previous steps (one step in this case is placing a stone).

The groups of each color are counted using the Union Find algorithm. The algorithm is implemented that the root of each group is always the last stone placed in that group and beside the roots the previous roots are also stored. These modifications enable the program to undo moves, which are very useful when implementing the alpha-beta framework.

When the computer is set to play the black (Black AI) or the white player (white AI), the AI is triggered in each turn by the second click event of the other player (except for white AI in the first turn, when it is triggered by the start button). The AI uses NegaMax search with alpha-beta windows. The algorithm starts with alpha by choosing first a black then a white field from the empty fields. Then invites itself, decreasing the depth, negating and switching alpha and beta, and negating the color variable. Then, when depth reaches 0 or a terminal node is reached, it evaluates the node.

The evaluation function is for simply the score, except when the size of the board is 5. In that case the game is divided into 4 phases:

- In the first two rounds, the corner pieces worth the most, and as we get closer to the center, the score decreases. In this phase, creating new groups are also promoted
- In the next phase (between round 3 - 10), the position-based score still holds, but the creating new groups are punished, and the ideal number of groups is set to be three. In this phase the game score is also a factor.
- In the third phase (between round 11 - 25) before NegaMax, the AI checks, if it can connect the two biggest group of the enemy in one move, or separate two of its biggest group by an enemy

piece. If it can, it does, and then starts the search by going through only their color in the first iteration.

The evaluation function is using only the score of the game.

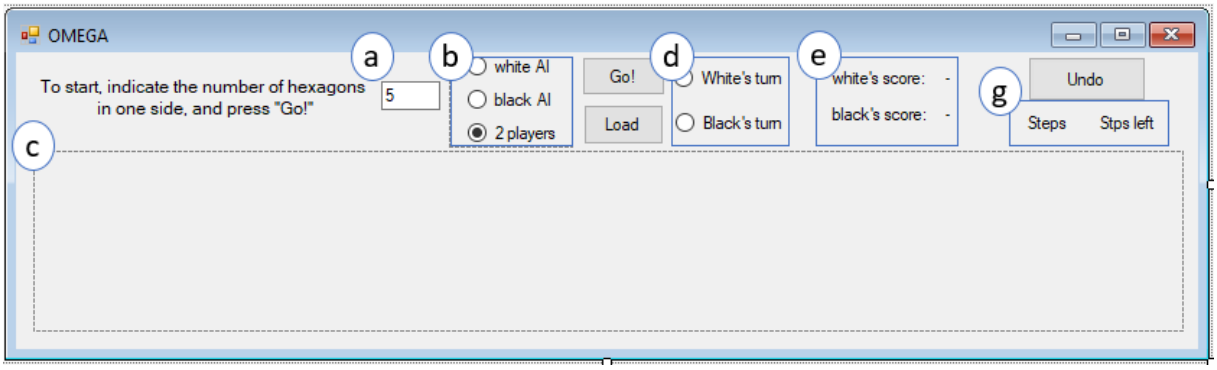
- In the endgame the search depth is increased, and only the score is used as the evaluation function.

After evaluation it returns to the previous depth, choses the bigger between the previous value and the new value for the new value and the best moves. After this it sets alpha to the bigger of value and alpha, then checks for prunings. When a pruning occurs, the next white stone is going to be searched. After examining all available options, the algorithm returns with the best moves. Then the best moves are played, and the human player can make his/her moves.

Upon initialization the program calculates the number of possible steps. When a stone is placed, this number is decreased by 1. When there are no more available steps left, the game ends, and a Message Box pops up.

Components

The user interface



1. Figure - Design view of Omega in Visual Studio

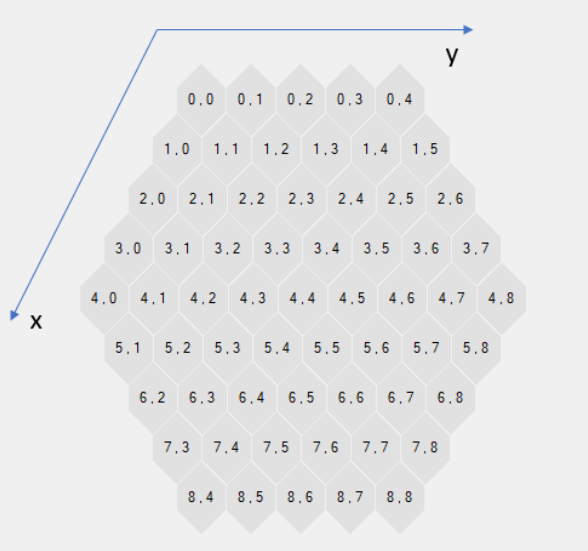
In the user interface, the player can set the size of the board (1.a), and when they want to play a 2 player game or they want to play against an AI (1.b). Then by pressing the “Go!” button, the game starts, by placing the fields inside the container (1.c). The radio buttons, called (“White’s turn” and “Black’s turn”) are showing whose time it is to play (1.d.) The score of the game is calculated after each move (1.e), and the number of steps left is also updated (1.g). The number of steps (1.g) is calculated at the beginning of the game and remain unchanged until the next game. The “Undo” button removes the last stone from the field, and increase the “Steps left” by one. The “Load” button is not yet functioning, but the underlying idea was to save every variable of the game into .txt files after each turn, and load the game from these.

Setting the board

After the player clicks on the "Go!" button, the program sets the board. This consist of Field objects, which inherit the button class, but have some additional properties:

- id (integer) – number of the field, starting with 1 at the coordinates (0,0)
- color (integer) – 0 if empty, 1 if it represents a white stone, 2 if it represents a black one
- row (integer) and column (integer)

For row and column, axial coordinates are used as shown on Fig. 2. I chose this, because they are easy to interpret and they can be easily converted to cube coordinates, if needed.



2. Figure - Axial coordinats in size 5 Omega

Calculating the score using Union Find

My implementation of Union Find uses a two-dimensional array (called searchTree), which for each field contains the ID of that field, the group size, the parent of the field, and the previous parent. For each field, the ID is set to its own ID, the group size is initially set to 1, the parent is to the ID and the previous parent to 0. Because the board is shaped like a hexagon, this array has more members than fields on the board. For these members, everything is set to 0.

I have a list of every white and every black root, and every white and every previous root. These lists are made of Root object, which have three parameters: ID, size, and which root removed them from the roots.

When I click on a field, the program calls the checkNeighbours method, which checks whether the field's neighbours and the field have the same colours. If it finds a neighbour it calls the unify AB method. This first gets the root of the neighbour, using the Find method. Then it checks whether they already belong to the same group. If no, it makes some changes in the checkNeighbours array:

neighbour's previous root = neighbour's root

neighbour's root = clicked field

clicked field's size = neighbour's size + 1

Then it removes the neighbour from the list of roots, and adds it to the list of previous roots, and sets the removed by property to the clicked field. The method also removes the clicked field from the list of roots, if it is already a member and then adds the clicked field to the list of roots.

This method guarantees, that the last member

The removeLast method has three parameters:

- toRemove – the ID of the last placed stone
- newRoots – list of roots for the same colour as the last placed stone
- oldRoots - list of previous roots for the same colour as the last placed stone

First the method restores the old roots. For this it looks at the last 3 elements of oldRoots, and if the removedBy parameter equals to toRemove, then restores the list member, by setting its removed by to 0, and setting its root in the searchTree array to the previous root.

Then it sets the root to itself, and the size to 1 for removedBy in the searchTree array and deletes it from the list of roots.

It is important to mention, that the remove last only works, if the last placed stone becomes the root. And if we want to use it multiple times in a row, we cannot use path compressing. This means, that we cannot use path compression in the alpha beta framework, but we could be still able to use it in the game (if we do not use the undo button carelessly). In the current version I do not implement this, but path compression can be turned on for each function, using the path compression parameter.

Calling the AI

In the user interface we can choose between the black AI, the white AI and the 2 players versions. If we set one of the AI players on, then the button click events are used to trigger the AI as is shown in the code snippet below:

```
dynamicButton.Click += (object sender1, EventArgs args) =>
{
  […] //setting variables
    if (player == true) //player is true when it is white's turn
    {[…]} //player steps with white stone
    else
    {[…]} //player steps with black stone
    player = !player;
    //Calculate result, writing reusult on GUI, calculating step left
    […]
    // Game end check
    if (stepsLeft == 0) { MessageBox.Show("The game has ended"); return; }

    if (countSteps % 4 == 2 || countSteps % 4 == 1)
    {
        radioButtonWhite.Checked = true;
        radioButtonBlack.Checked = false;
         if (whiteAIChecked) //white AI plays
        {[…] //calling NegaMax, execute step}
        if (blackAIChecked) // black AI plays
  {[…]//calling NegaMax, execute step }
   }
```

There is one exception: when white is played by AI and it is the first turn. In that case, the search is triggered by the click event of the start button.

NegaMax (with standard evaluation function)

In the center of my AI is a NegaMax search. The simplest version of the search code can be found in the attachment. It works by first checking if it is time to evaluate. If not then the function creates a list from the empty fields which is called emptyWhites. Then for each items of the empty whites, a list is created from the empty blacks (this is a copy of empty whites, minus the current item). Then a step is performed with the white item, and then foreach empty field NegaMax is called. Upon return, the value, the optimal steps, and alpha is calculated, and the program is checking for potential cutoff. If there is no cutoff, first the black is removed, then if the function evaluated / cut off all blacks, the white stone is removed. Both remove utilizes the removeLast method. After all whites have been visited, the function returns with the best move.

By moving with the white and the black pieces in the same search, I limited the cutoffs to the black pieces. The alternative would be to search each move separately, but that means, that at depth = 1 and depth = 2 the player maximizes, then at the next two depts it minimizes. This would create the opportunity to cut at white nodes, but also a lot of opportunity for hard-to-debug errors.

Optimizing for5x5 board

The goal of the Omega implementation was to compete against the other class members AIs. Since the competition was announced to be held on 5 size game, I payed special attention to this version. After playing many games against myself and against other, I realized a few guidelines, which then I could implement in my AI.

Most of the games I played ended with each player having between 1-5 groups. This means, that the AI should not try to aim for mathematically optimal sized (3 size big) groups, because, then instead of getting optimal results, many groups the payer will be connected by the enemy. On the other hand, if the player aims to have 3-4 groups, and creates more and more new group for the enemy, the enemy's groups might be connected in the endgame, and if the player manages to separate its groups with enemy stones, it can win. Therefore, in the final stages of the development, I tried to create a complex evaluation function, which implements this strategy.

For this to work I divided the game into 4 phases. In the first, my AI concentrates on getting the enemy pieces in the middle, and mine in the side, and creating 3-4 groups for myself. In the second phase, the AI tries to grow its groups without connecting them, and create many new groups for the enemy. In the next phase I implemented a trick: the AI can connect the two biggest enemy groups, or divide its two biggest groups by an enemy piece, do that, else do the full search. Here the evaluation function can be the score. In the endgame, the goal is to foresee the end result, therefore, instead of the connecting strategy the AI focuses on increasing the search depth.

Phase 1 (turn 1 -2)

For phase one, my original plan was that the evaluation function should be the combination of the placement points, the group size points, and the points for the Manhattan distance from own team members.

The first two rules was successfully implemented. Each group of one received the 7 points, each grup of two 4 points and the groups of threes 1 point. For the position score, an array was created where a score was assigned manually for each position (6 if corner, 1 if center). Then the score was calculated: score = `a*([white's score for group size] – [black's score for group size]) + b*([white's score from position] – [black's score from position]).`

The last one was not ready in time, but it would have worked by calculating the manhattan distance from the just placed stone and every pieces of the same color, giving a point from 1-10 for the smallest distance (0 if 0, 10 if 8).

Phase 2 (turn 3 -10)

Group size	Point
1	0
2	4
3	8
4	8
5	8
6	4
7	3
8	1

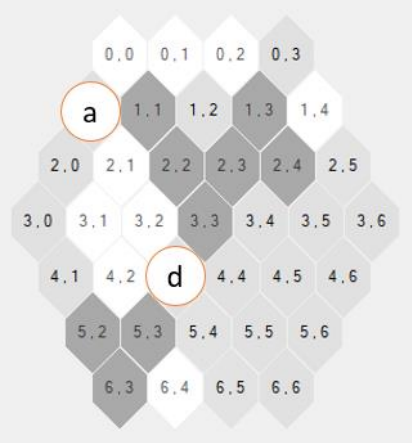
1. Table

Phase two was a slight variation of phase one. The position score was used here as well, but the score for the group sizes were changed, and the game score also started to play part. The rewards for group sizes are shown in Table 1.

This was made to decrease the chance of creating new groups for the AI ang creating more new groups for the enemy, but also to get the optimal number of groups. For optimizing the number of groups another function was also used, which calculated the number of groups and 10 points for three groups, 8 points for 4 group and 1 point for 5 group. The game score was calculated by having the square root of the whites' and the blacks' score and subtracting the two values, because this way the result will be in in a smaller range.

The score function was: `(a * [white's score for group size] – [black's score for group size]) + b * ([white's score from position] – [black's score from position]) + c * (int)(Math.Sqrt([game score for white]) - Math.Sqrt([game score for black])) + d * ([number of groups score white] - [number of groups score black]))`

Phase 3 (turn 11 – 25) and Endgame



3. Figure - Must-do-moves. It is the black player's turn, who could either attack (a) or defend (d)

For both phase 3 and for the endgame the evaluation function was the same: `[game score for white]) - Math.Sqrt([game score for black]` the difference was that for phase three, before calculating the store the game was checked for must-do-moves. The concept of this is similar to the killer move, but instead of using int to improve the move ordering, here I use it to automatically make a move.

A must-do-move is when the AI could connect the two biggest groups of the enemy, or put an enemy piece between two of its own biggest group (see Fig.3.). Originally, I planned to search for maximum three length distances between the biggest groups, but in the end, I chose the distance of one. I also have not implemented a method between choosing the attacking move or the defending move, my AI will choose the one it finds first.

The algorithm for the must do move works the following way: If it is at the first iteration of NegaMax, and we are in phase 3, then the algorithm finds the two biggest white, and the two biggest black groups. Then it searches all the empty fields, and searches for one, which has a neighbour from both the two biggest groups (from the same color). It searches all possibilities, and choses the last successful one (it does not differentiate between the attacking and the defending moves. then it continues with NegaMax, but if it has found a must-do, then that move is fixed, and that branch will not be searched.

Tuning during the competition

In both phase 3 and in the endgame, I used a simple method to increase the search depth. In each turn after the beginning of phase three, I measured the time it takes for the computer to calculate the search at a given depth, and if it was below 15 sec, the program increased the depth for the next round. However, this did not work as planned. The result was that after a fast search, the next search would have taken more than 10 minutes to be calculated. Therefore, later in the competition I set the depths at fix values, dividing phase three and the endgame into subcategories. This got the wished result, and the game became much faster, therefore I left this modification in the final version.

Potential improvements

Although the project is finished, there are many parts, where some improvements would be necessary. Here I will list the ones, I consider the most important.

Save and Load function: These functions are relatively easy to implement, I have already started with their development and I have a broad conception how these should work. I could write the variables into .txt files, using the StreamWriter method, after each successful turn. Then in case of failure, when the user clicks on the Load button, the program would read the variables, and basically copy the start the game method with some minor changes.

Evaluation function: During the competition, I saw a program which used group to group distance searches, and I strongly believe that this type of search could make my program better, even if it could only run at depth 1 search. One step easier than this is to create the distance-based evaluation in phase 1 using the Manhattan distance, and improve the must do move, by searching for 3 tile length path instead of one.

Move ordering: I think the biggest drawback of my software that I do not utilize move ordering. I currently cannot go more than 4-ply deep even in the endgame, and this could be solved by getting an reordering the moves using the previous search result, or, for example the position score in phase 1 and phase two.

Appendix – NegaMax shortened

```
public void NegaMax(List<Field> boardList, int stepsLeft,
int depth, int alpha, int beta, int color, out int value,
out Field stepWithWhite, out Field stepWithBlack)
        {
            Field bestWhite = new Field();
            Field bestBlack = new Field();

            if (depth==0 || stepsLeft==0)
            {
                //evaluate boardList (Standard evaluation function)
```

```
    Score(listWhites, listBlacks, out int evalW, out int evalB);
    value = color *( evalW - evalB );
    stepWithWhite = bestWhite;
    stepWithBlack = bestBlack;
    return;
}
value = -1000000;
//this will be the value of the 1 ply deeper search
int dummy;
// generating move for white stones
List<Field> emptyWhites = new List<Field>();

foreach (var field in boardList)
{
    if (field.color==0)
    {
        emptyWhites.Add(field);
    }
}

foreach (var white in emptyWhites)
{

    // generating move for black stones
    List<Field> emptyBlacks = new List<Field>();
    foreach (var empty in emptyWhites)
    {
        emptyBlacks.Add(empty);
    }
    emptyBlacks.Remove(white);

    //adding whites
    white.color = 1;
    checkNeighbours(white, false, listWhites, previousListWhites);
    foreach (var black in emptyBlacks)
    {
        //adding blacks
        black.color = 2;
        checkNeighbours(black, false, listBlacks, previousListBlacks);
        //evaluate or go deeper
        NegaMax(boardList, stepsLeft - 2, depth - 1, -beta, -alpha,
        -color, out dummy, out Field d1, out Field d2);
        //setting bestStep
        if (-dummy > value)
        {
            bestWhite = white;
            bestBlack = black;
            value = -dummy;
        }
        //value = Math.Max(value, -dummy);
        alpha = Math.Max(value, alpha);
        if (alpha >= beta)
        {
            black.color = 0;
            removeLast((byte)black.id, listBlacks, previousListBlacks);
            break; // cut-off
        }
        //removing blacks
        black.color =  0;
        removeLast((byte)black.id, listBlacks, previousListBlacks);
    }
    //removig whites
```

```
            white.color = 0;
            removeLast((byte)white.id, listWhites, previousListWhites);
        }
        stepWithWhite = bestWhite;
        stepWithBlack = bestBlack;
        return;
    }
```