Artificial Intelligence Programming Exercise Report

Hannila Mikko
Lehto Mikko
Pietikäinen Niko

Cypress

Table Of Contents

1. Introduction

We started the project early on in February with the intention of beating all the other algorithms on the ranking list. Before starting work on the algorithm itself, we played several games of Othello and researched strategies for the game. In many stages of the project we developed multiple versions of the algorithm in parallel and in the end selected the best solution.

2. Search Tree

2.1 Basic functionality of the search tree

We started by implementing the minimax-algorithm and early on alpha-beta pruning was added to it (*wikipedia*). After we finished alpha-beta it was clear we didn't need to use the methods from the reversi libraries in the search tree, because we found out that our implementation didn't require saving all nodes in memory. Cypress estimates scores and builds the search tree simultaneously, as opposed to building the tree first and then estimating scores with the pre-given methods.

2.2 Minimum requirements for a stable algorithm

Once the basic functionality of search tree was ready, we had some problems with stability. Somehow the algorithm kept crashing sometimes. The bugs had to do with certain game states that Cypress did not account for. After we fixed the problems Cypress manages the following situations: when the algorithm or its opponent does not have movements at current turn; when neither has any possible moves (leaf node); when there are no empty squares on the table (terminal state); when the algorithm has reached terminal state (recursive search stops); when time is running out; and when the algorithm is able to get all of the disks during the game (very valuable state).

2.3 Improving performance of the search tree

After the search tree was fully functional we compared our algorithm to CheapBlue and noticed CheapBlue has less leaves than Cypress does. We chose to try a more effective pruning strategy to increase performance. We invented two strategies: we could remove obviously bad moves from the search tree; or maybe we could sort possible moves based on scores. To remove obviously bad movements we needed a sort function anyway so we began by creating the sorting alpha-beta algorithm.

2.3.1 Sorting alpha-beta

We realized that if we use some kind of quick heuristic to sort the possible movements of a given game state, alpha-beta would be more likely to find better moves faster and the pruning would kick in sooner. This would reduce the number of unnecessary branches and allow the algorithm to search deeper. This turned out to be very complex operation. All of the searched movements needed to be added to a container - which was an arraylist in this case. When the algorithm prunes something, all of the the pruned movements must be rated as a really bad moves. Sometimes the container has no movements after previous move then movements must be researched and added to the container, this can occur when algorithm have not searched depth before, or when alpha-beta has pruned these movements at previous iteration. After the sorting alpha-beta method was created we noticed that system is able to memorize hundreds of thousands game states without any problems ergo last version of Cypress saves all researched game states to containers. At this point Cypress often has less leaves than CheapBlue at the same depth
.

2.3.2 Removing obviously bad moves

Because we had spent a lot of time developing the search tree we chose to not implement this functionality in the final version. Removing bad moves would be a radical solution and it would increase the impact of the horizon effect.

2.3.3 Quiescence algorithm

During the project we also investigated the possibility of using a quiescence algorithm to complement the search tree. If the search tree reached maximum depth on a turn where the game score was fluctuating a lot, the quiescence algorithm would increase search depth for that branch until the game settled down. This would be a way to work around the horizon problem where Cypress sometimes made bad decisions because the search tree ended on a big swing turn. We had problems tuning this algorithm to actually increase results and in the end we decided not to implement it due to added complexity and time constraints. (*wikipedia*)

3. Heuristics

Good heuristics are crucial for the algorithms performance: a computationally worse algorithm with good heuristics can easily beat a computationally better one with bad heuristics. First versions of our algorithm only took into account the number of disks on the board, from which it was easy to note whether our search tree was functional or not. From counting disks we then moved to maximizing available moves in our heuristics and later on we switched to using scoreboard for each position and counting static disks.

3.1 Counting Static Disks

Static disks are marks that cannot be reversed anymore. This is common reversi strategy but we designed two different functions to solve this problem, whose performance was equal. Second function is used in final version. First method (which is not used) is in picture below.

```
void getStaticMarks(int[][] staticMarks, GameState state, int index, int x, int y)   // Updates board value for static marks using a flood fill algorithm. Must start from corners!!
{
    if ((x >= 0) && (x < 8) && (y >= 0) && (y < 8)) // Make sure coordinates are on the board
    {
        if (staticMarks[y][x] != 1) // Pass if the square has been evaluated already
        {
            if (state.getMarkAt(x, y) == index) // Check that a mark is on the square
            {
                if((x == 0) || (x == 7) || (staticMarks[y][x - 1] == 1) || (staticMarks[y][x + 1] == 1))
                {
                    if((y == 0) || (y == 7) || (staticMarks[y - 1][x] == 1) || (staticMarks[y + 1][x] == 1))
                    {

                        staticMarks[y][x] = 1;  // This square is static, mark it on the map
                        getStaticMarks(staticMarks, state, index, x - 1, y);    // Check each adjacent square if this square is static
                        getStaticMarks(staticMarks, state, index, x + 1, y);
                        getStaticMarks(staticMarks, state, index, x, y - 1);
                        getStaticMarks(staticMarks, state, index, x, y + 1);
                    }
                }
            }
        }
    }
    return;
}
```

Figure 1. Static-disk function

3.2 Pattern detection

In very final stage of project we noticed that there are two patterns that occur often in lost games. We designed a simple pattern detection method. For some reason when we added this method Cypress started to lose more games against CheapBlue so we did not implemented it in final version.

```
// pattern detection
int[][] checkBoard1 =       {{0,  1,  1,  1,  1,  1,  1, 0},
                             { 0,  0,  0,  0,  0,  0,  0, 0},
                             { 0,  0,  0,  0,  0,  0,  0, 0},
                             { 0,  0,  0,  0,  0,  0,  0, 0},
                             { 0,  0,  0,  0,  0,  0,  0, 0},
                             { 0,  0,  0,  0,  0,  0,  0, 0},
                             { 0,  0,  0,  0,  0,  0,  0, 0},
                             { 0,  0,  0,  0,  0,  0,  0, 0}};

int[][] checkBoard2 =       {{0,  0,  1,  0,  0,  0,  0, 0},
                             { 0,  0,  1,  0,  0,  0,  0, 0},
                             { 1,  1,  1,  0,  0,  0,  0, 0},
                             { 0,  0,  0,  0,  0,  0,  0, 0},
                             { 0,  0,  0,  0,  0,  0,  0, 0},
                             { 0,  0,  0,  0,  0,  0,  0, 0},
                             { 0,  0,  0,  0,  0,  0,  0, 0},
                             { 0,  0,  0,  0,  0,  0,  0, 0}};
```

Figure 2. If player have all squares marked by ‘1’ the position is good.

```
int temp1 = -1;
int temp1_1 = 0;
int temp1_2 = 1;
boolean pattern1 = true;

for(int i=0; i < 8; i++)
{
    for(int j=0; j < 8; j++)
    {
        if (checkBoard1[j][i] == 1)
        {
            temp1 = state.getMarkAt(j,i);
            if (temp1 == temp1_1 || temp1 == temp1_2 )
            {
                temp1_1 = temp1;
                temp1_2 = temp1;
             }
            else
            {
                pattern1 = false;
            }
    }
}



if (pattern1)
{
    if (temp1 == myIndex)
        score += 5000;
    else
        score -= 5000;
}
```

Figure 3. Main idea of pattern detection.

3.3 Terminal-state heuristics

The game ends when neither player has legal moves left. At this point the winner can be calculated simply by counting marks for both players. Firstly, the value of score is the mark count of Cypress multiplied by ten. If Cypress has more marks than the opponent, scores will increased by the number of (empty squares+1)*100 000. If the opponent has more marks than Cypress, scores are decreased by the same number. This procedure values early win more than late win, and late loss more than early loss. The scores for terminal states are inflated to be much more valuable than scores for intermediate game states derived from the heuristic function. Once the game has progressed enough for our algorithm to see the end, we can start playing “perfectly” and use winning or losing the terminal state as the only evaluation.

3.4 Heuristics from reference

We studied othello strategies and other algorithms and found an interesting web-site (*kartikkukreja*). We decided to use two methods from the site in our heuristic function: mobility and corner closeness. We thought that maybe the purpose of this exercises is not

to invent all reversi strategies by ourselves, but implement known strategies in our own algorithm.

3.4.1 Mobility

Mobility takes a ratio between our possible moves and opponents possible moves, then the ratio is multiplied by constant to produce a score.

3.4.2 Corner closeness

Corner closeness checks if a corner is empty and provides the value for every neighbouring squares. Then it calculates the ratio between our and opponents ‘bad disks’ and multiplies the ratio by a constant to produce a score.

4. Testing

The algorithm has been tested in many ways, e.g. printing game states from restricted depth and comparing algorithm’s performance to previous versions. Code has also been reviewed thoroughly again and again by all members of the team.

4.1 Choosing release version

Cypress uses very complex heuristic function so it was not easy to find best values for all constants and factors in it. Over 20 versions were made with different constants, scoreboards and heuristic methods, then over 200 games was played to limit the number of versions. The remaining versions of Cypresses was investigated even more and finally release candidate was chosen.

5. Conclusions

We didn’t achieve an algorithm that would always beat CheapBlue. Our job proceeded well until the code got so complicated that even the smallest changes required a lot of planning and testing. Eventually there was no more time for algorithm development. The project showed us that even the simplest AI can beat human intelligence in certain problem solving situations and that a person can create an algorithm that is better in the task at hand than they are.

6. References

http://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning
http://en.wikipedia.org/wiki/Quiescence_search
https://kartikkukreja.wordpress.com/2013/03/30/heuristic-function-for-reversiothello/