Softwarepraktikum SS 2021

Assignment 3 Report

Group 3

Jamie Anike Heikrodt	394705	anike.heikrodt@rwth-aachen.de
Thomas Pollert	406215	thomas.pollert@rwth-aachen.de
Jascha Austermann	422571	jascha.auster@rwth-aachen.de
Silyu Li	402523	silyu.li@rwth-aachen.de

Contents

Exercise 1

Our code for Minimax relies on the pseudo code from the lecture "Artificial Intelligence" by Prof. Lakemeyer.
We do not work on a tree structure but use recursion. The function takes the cut-off depth, the current depth and a map state as arguments.
The termination condition is whether the current depth is equal to the cut-off depth. Then we return the heuristic value.
If we have not reached the cut-off depth yet, we first check whether the current player can make a move and otherwise we call the function for the next player. If the current player can make a move, we recursively calculate the Minimax values for every possible successor map. If the current player is us, then we return the best of the resulting values and otherwise the worst.
On the first level, we remember the move that belongs to the best seen Minimax value of a successor state and finally we return this move.

Exercise 2

Our code for Alpha-Beta Pruning relies on the pseudo code from the lecture "Artificial Intelligence" by Prof. Lakemeyer. We do not work on a tree structure but use recursion.
We did one modification for the first level of the search tree. The pseudo code suggests to calculate the alpha-value for the root and then return the action that has this value as its own alpha-value. However, our methods for calculating alpha and beta only return the value and no corresponding move.
Modifying it to return both seemed unnecessary as we only need the move for level 1. Therefore, we calculate level 1 in an own method and remember the move that belongs to the best alpha-value seen.
We also changed our code such that there can be several enemy players by checking whether we first player is us (then we calculate the alpha-value for the next level) or someone else (then we calculate the beta-value).

Exercise 3

3.1 Discussions

The third task deals with benchmarks. For this assignment we wanted to create a basic foundation for our benchmark strategy and use it to test and compare Minimax and AlphaBeta.

3.2 Solutions

To benchmark our programm, we use *System.currentTimeMillis().* To measure the time methods take in our programm, we just calculate the distance between the time before and after the method call. Some results of this regarding MiniMax and AlphaBeta will be presented in our group-meeting.
Future benchmarks will require to test different parts of methods. We will use the same benchmark technique as for simple method calls. This requires making clones of our methods, but gives us the chance to benchmark most parts of our code easily.
The data obtained when benchmarking is written in a CSV-file and then plotted with "datplot" by Michael Vogt. This gives us the chance to write all obtained data of one benchmark in a file and plot by different values later.

General changes

4.1 Saving neighbours

4.1.1 Problem

Before we used to save the extra transitions in a hash map. When looking for a neighbour, we checked whether there was an entry in the hash map for this transition. However, on maps with many extra transitions, this turned out to be quite inefficient as we needed to check the transitions of all tiles.

4.1.2 Solution

Each tile now saves their own neighbours in an ArrayList containing Integer arrays. Therefore, we can specifically check the neighbours for this tile and don't need to look at every extra transition of the whole map.

4.2 Changing directions

4.2.1 Problem

We did not notice that taking extra transitions could change the direction of a path. Therefore, our successor maps and move validation did not always calculate the correct results.

4.2.2 Solution

For each step that we take, we check whether it uses an extra transition and update the direction accordingly.

4.3 Endless loops

4.3.1 Problem

In our successor map and move validation methods, we encountered endless loops. This was because of a mistake in the break condition. We breaked the loop if we returned to the tile where we started. However, it could be possible to have a loop at a later point that was not noticed by our program.

4.3.2 Solution

We have a new 2-dimensional Array called track which tracks the tiles that we already saw. It has the size of the map and track[x][y] corresponds to tile (x,y) of

the map. At the beginning of each path calculation, we reset the array. Then, for each tile that we checked we would update the value in the array. By looking at the array we could check whether we already were at this tile and therefore ran into a circle.
At first, the values stored in the array were boolean. It was initialized with false and updated to true if we visited the tile. However we noticed that it would be possible to visit a tile from different directions without being in a circle. We only need to break if we visited the tile from the same direction twice. Boolean values could not express that. However, storing 8 different values for each direction for each tile seemed like too much space. Therefore, we came up with a solution that only needed one Integer value per tile. Each entry is initialized with 1. For each direction that we visited a tile from, a distinct prime number is multiplied to the entry. We use the 2nd to 9th prime numbers for that. We can now check whether we visited a tile from a certain direction by checking if the corresponding prime number is a factor of the array entry. The starting tile is initialized with 9699690, which is the product of the 2nd to 9th prime numbers, because we want to break if we reach the starting tile from any direction.

4.4 Who did what?

- Exercise 1 (Minimax): Jamie, Silyu
- Exercise 2 (Alpha-Beta): Jamie
- Exercise 3 (Benchmark): Thomas
- Implementation of command line flags: Jascha, Jamie
- General improvements and debugging: Jamie