

Adversarial search with alfa-beta pruning

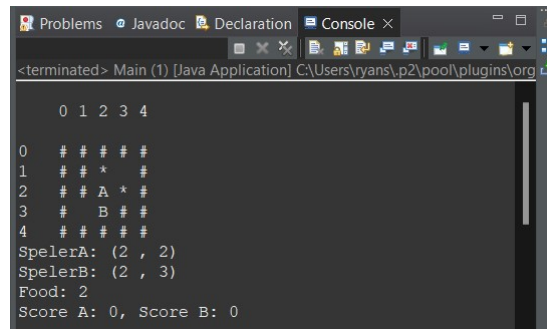
Ryan Sleuwaegen, Sjouk Ketwaru
s3122166, s3287297

10 oktober 2022

1 The Game Board

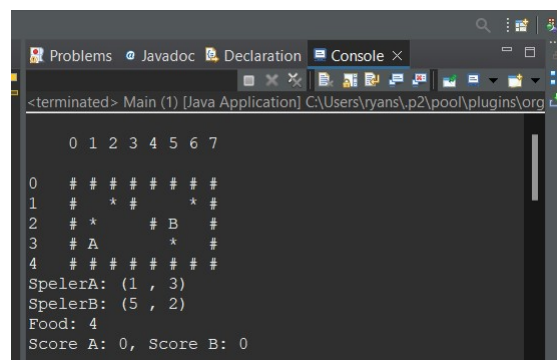
1.1 public String toString

We chose to put white spaces between each character for more clarity.



```
<terminated> Main (1) [Java Application] C:\Users\ryans\p2\pool\plugins\org
0 1 2 3 4
0 # # # # #
1 # # * # #
2 # # A * #
3 # # B # #
4 # # # # #
SpelerA: (2 , 2)
SpelerB: (2 , 3)
Food: 2
Score A: 0, Score B: 0
```

Figuur 1: Board 1



```
<terminated> Main (1) [Java Application] C:\Users\ryans\p2\pool\plugins\org
0 1 2 3 4 5 6 7
0 # # # # # # # #
1 # * # # # * #
2 # * # # B # #
3 # A # * # # #
4 # # # # # # #
SpelerA: (1 , 3)
SpelerB: (5 , 2)
Food: 4
Score A: 0, Score B: 0
```

Figuur 2: Board 2

1.2 public State copy

The method copy makes a copy of the current state of the game. First it creates a new state S, and then it makes copies of all important variables, this includes:

1. The coordinates of both players
2. The score of both players
3. Who's turn it is
4. The amount of food still that's not eaten yet
5. The width and height of the board
6. The entire board itself
7. The moves done by both players

Some variables are arrays and some are vectors, so they need to be copied in a for-loop to copy every index. After everything is copied the method returns the state S.

1.3 public Vector<String> legalMoves

The method legal moves checks what moves are legal for the current agent. At first it makes a vector to put all the moves in. It then checks what all the surrounding cells are. If those cells are not walls, the program adds the action to the vector.

After the actions up, down, left and right are checked, it checks if the agent can eat or block. It does this by checking what the current cell is. If it is food the action eat is added to the vector. If it is not food it checks if the agent is not at the same cell as the opponent, and that it did not block already in its previous turn. If all of those requirements are met, the action block is added to the vector.

After everything is checked it returns the vector.

1.4 public Void execute

The method executes simply executes the given argument. It does not check whether the action is legal or not. It first checks the coordinates of the current agent and makes a backup of it. Then it checks if the current cell is food, this is necessary since there is a possibility that a agent goes over a food cell without eating it. It then empties the old cell and performs the new action.

If the agent moves the coordinates are changed, if the agent eats the total amount of food gets updated and his score gets updated, if the agent blocks the current cell becomes a wall.

After the case the coordinates of the agent are updated. If an agent is on the same cell as a wall or food, the boolean 'under' gets activated, this is to not print the agent. Then it checks if an agent went over a food cell without eating it, if this is the case a '*' will be placed back at the old cell. This is needed since we first empty the old cell before performing the action. That is why we made a backup of the old coordinates. At last it checks if the agent previous action was a block, since that wall would get deleted by moving, so we place back the wall.

At last it updates the players turn.

1.5 public Boolean isLeaf

The method `isLeaf` checks if the current state is a leaf. It does this by checking multiple things. The first thing it checks is if there are any `legalMoves` left, if this vector is empty, the method returns true.

The second thing it checks is if there is any food left. It does this by simply checking if the class variable `food` is 0.

The third thing it checks is also to cut the amount of leaves it has to check. What we did was to check if a player could win in the current state, or that there was not even enough food left to get more points than the opponent. So for example if agent 0 has 1 point, and agent 1 has 6 points, if there is only 2 food left, it is impossible for agent 0 to get more points than agent 1.

1.6 public Double value

The method `value` checks if a the given agent in the argument wins, loses or plays a tie. If the score of the agent is higher than the score of the opponent, the method returns 1, if the score of the agent is lower than the score of the opponent, the method returns -1, and if neither are the case, both players have an equal score so the method returns 0.

We noticed that sometimes with minimax and alfabeta a state would be printed that was not completed at first. What happened here was that the game was not finished and the `maxDepth` was reached, so the game checked the value. If the given agent had 1 point and the opponent had 0 points, the method would still return 1, so that's why we now also check if there is no food, if that is the case the game can return 1 or -1, otherwise it will return 0 if the scores of the players are equal, by default is the return -1 so that all options are still open at the start of the program.

2 A Basic Minimax Algorithm

2.1 public State minimax

The method `minimax` suggests the best move for the current player, in this instance for Agent, in other words player 0. The base case for this recursive version is if the node is a leaf or if the depth is equal to the `MaxDepth` (which is 11 by default). The algorithm then checks if the agent needs to be maximized (for agent 0) or minimized (for agent 1). After that there is a for loop in which it checks every move that the current agent can do and goes in the recursion in search of the best state for every move.

Finally the algorithm takes the maximum state for the agent 0 (may that be a win, a loss or a draw) or minimum state for agent 1.

2.2 increase maxDepth

When you increase the `maxDepth` the best found states differ. This happens because, when you only have a `maxDepth` of 7, the 2 players can only execute 7 moves in total together, so both players do not have much moves to play. Because of this it might not be possible yet for the agent to win, because he can not reach the food before a depth of 7 is reached.

When you increase the `maxDepth` you also increase the amount of actions the players can execute, so it is more likely to find a better state for the agent.

What we noticed was that in some cases with different boards it was possible that it could not find a complete state of the board, so it prints the board without an end state if this is not met yet.

3 Adding The alfa-Beta Heuristic

3.1 public State alfabeta

The method alfabeta takes the code from minimax and cuts a lot of unneeded branches. For example there is a branch with value 0 (draw), then alfabeta does not calculate the branches with a lower score, but does calculate those with a higher score. This is all done by adding a simple if-statement for comparing alfa (in case of agent 0) to the current value of the current state in the recursion, changing the alfa when there is a higher value possible. And another if-statement (in the case of agent 0), comparing the best of agent 1 to the current value and if it is lower or the same the for loop breaks.

3.2 Comparison Minimax And Alfabeta

Search depth	Minimax	Alfabeta
7	1841	180
9	11197	663
11	65419	1962
13	382693	6810
16	5085743	34627

Tabel 1: table of nodes with search depth of Minimax and Alfabeta

Reduction factor:

- for 7 is 10,23
- for 9 is 16,89
- for 11 is 33,34
- for 13 is 56,20
- for 16 is 146,87

The average reduction factor from 7 to 13 is:

- $146,87 / 10,23 = 14,36$

These results show that the change of the reduction factor makes sense, because it is a square root relation.

3.3 Extra method public void nodes

We created an extra method called nodes in the Game class. In this method we run minimax and alfabeta 4 times each with increasing maxDepth to count the amount of nodes we needed.