Assignment 1: Adversarial search with alpha-beta pruning

Joost Broekens

September 28, 2023

1 INTRODUCTION

Before you start, first read the *whole* assignment. In this assignment you will program an agent that can play a simple game against another agent. The game is an invented game called *Blocker*. Two agents compete to collect the most food in a maze. The game ends when all food is collected, or, when either of the agents has no legal move left. At the end, you win when you collected the most food, or when the other agent has no legal moves left; whichever happens first. You lose if the other has more food or you cannot make a legal move, again whichever happens first. In all other cases the game ends in a tie. The agent has 6 possible moves: up, right, down, left, eat, and block. Up, right, down and left move the agent one step in that direction, and are legal moves if there is no wall at the target location. Eat gathers food at the current agent's position, and is a legal move if the agent is on top of a food patch. Block drops a wall at the agent's position, making it impossible to move *to* that location, and is a legal move if the agent is on an empty spot (no food or wall present). Agents can only do legal moves. Each agent starts at a fixed position in the maze. Each turn an agent selects *one* action. The maze and both agent locations are fully observable. Agents can occupy the same location.

1.1 GETTING STARTED

To start, you need to first install Eclipse, the standard Java SDK. Download it here: https://www.eclipse.org/downloads/ Then you need to download the project .zip file from Assignment 1 on Brightspace. Unzip the project into a folder of your choice. Then open Eclipse

and import the project (File>Import>Existing projects). Choose the folder you unzipped the file to. This will open the project in Eclipse.

The project consist of two Classes (java files) and one data file (the game board text file). The first class is *Main*. The second class is *Game*. To run the code in Eclipse, click the green arrow with the popup text "Run Main". When you run the code now, it gives a warning (that you can ignore), then prints "hello world" and then stops. (to stop running a program while a program is running, click the red "Terminate" button in Java console).

For now, you can ignore the *Game* class and the two commented lines in the *Main* class. It has errors and you don't need it yet.

If you are not yet comfortable programming in Java, then you can use the following tutorial: https://www.w3schools.com/java/default.asp. Once you get the hang of it a little bit, you can continue with the assignment.

Please note that it is not allowed to change variable names and method headers (return type, name, arguments) when they are given in the exercise. This will result in us not being able to run your code, and will result in failing the exercise.

Also note that use of ChatGPT is limited to asking it questions about using Java syntax and Classes. You may not use it to do the exercsie for you.

2 THE GAME BOARD

Your first assignment is to program a class that is able to read in the starting board from a text file as found in the data folder in the project for this assignment. The file is called "board.txt". It has the following structure:

```
5 5
#####
##* #
##A*#
#*B##
#####
```

The first line defines the dimensions of the board (width, height). Then the board itself follows. The letter defines the start position of an agent (agent A and agent B). The * defines food. The # defines a wall.

2.1 QUESTIONS

1. Create a new Class called *State* in your project. Define the following variables in the class, and create the constructor of the class:

```
char [][] board; //the board as a 2D character array
int[] agentX; //the x-coordinates of the agents
int[] agentY; //the y-coordinates of the agents
int[] score; //the amount of food eaten by each agent
int turn; //who's turn it is, agent 0 or agent 1
int food; //the total amount of food still available
```

2. Implement a method *public void read(String file)* in the State class that takes as argument the file location (a String) and converts the information in the text file to the variables defined above. Try different variations and sizes of a starting board by copying "board.txt" and then editing the copy and loading that one. Don't edit the original, you will need it later. The method should be generic (handle different dimensions and starting locations of the two agents). You do not need to explain this method in your report.

3. To facilitate debugging, implement (again in the State class) a method *public String toString()* that returns the complete state as a nice printable string. Test it by using it in your Main class with a *System.out.println()* statement. Put the printed output of your method for two board states, each with a different size and different agent locations in your report.

4. As a preparation for what will come, implement a method *public State copy()* (again in the State class) that returns a proper copy of the class instance (object). Pay special attention to the copying of references (pointers in C++) versus actual content, because in Java variables that point to an object are in fact references. You want to be able to make changes to a copy without changing the original. They should be completely distinct instances. Explain the method in your report!

5. Now we are getting serious. Implement the method *public Vector<String> legalMoves(int agent)* (again in the State class) that will return a subset of actions from the set of possible actions *up,right,down,left,eat,block* that are currently valid for the <agent> given as argument. Do not implement any searching yet. You should return the moves that are legal at the current agent's position. Also implement the helper method *public Vector<String> legalMoves()* that returns legalMoves for the agent that has the current turn. Explain the method in your report!

6. Now that we know if a move is valid or not, we can also implement *public void execute(String action)* (again in the State class), which simply executes a valid move <action> for the agent who's turn it is as stored in <turn>. You may assume the algorithm does not cheat, so every call to execute() is with a legal move. To move the agent, change its coordinates. Don't forget to do food, maze and turn bookkeeping using the class

variables. Also keep a list of actions executed so far in a class variable *Vector<String> moves* by adding the current move to the vector. Explain the method in your report!

7. Now we have to implement a method telling us whether or not this State is a leaf. Implement *public boolean isLeaf()* (again in the State class), returning true if this State is a leaf, otherwise it returns false. Use the game rules as explained in the intro. Explain your leaf method in your report!.
8. Finally we implement a method that returns the *local* value for this State from the perspective of a particular agent. No search involved yet. Implement *public double value(int agent)* (again in the State class). Remember that we do *not* use heuristics, only win, tie, or loss values, and that winning for the one is losing for the other. So you should return one value from the set {1, 0, -1}. Also take the perspective of the agent <agent> into account. Use the game rules as explained in the intro. Explain your value function in the report!

You are now ready to test. Uncomment the two lines in Main.class and your Game.class should run. It prints a sequence of random moves and board states, until it reaches a leaf.

3 A BASIC MINIMAX ALGORITHM

You are now ready to implement the minimax algorithm. This is the prototypical adversarial tree-search algorithm. You need to understand the search lectures to be able to do so.

3.1 QUESTIONS

9. Implement the *public State minimax(State s, int forAgent, int maxDepth, int depth)* method in the Game class. The method should find the best state *s* for agent *forAgent* using the recursive depth-first minimax algorithm as explained on slide 16 of the adversarial lecture. Use maxDepth as cut-off to restrict the search depth. Note that in the slides a value is returned, while here we need a state to be returned. To test your algorithm, uncomment the line that calls and prints the result of the minimax method in Game.test(). Explain your code in the report. Hint: recursive calls need to operate on copies of states, but why?
10. Increase *maxDepth* from 7 to 13 by steps of 2. and observe the four outcomes of the minimax algorithm. How and why does the best found state differ depending on the maxdepth? Explain in your report.

4 ADDING THE ALPHA-BETA HEURISTIC

As you may have noticed, already at a depth of 13, the algorithm takes some time to finish. And this is only 7 steps ahead for the agent calling the minimax algorithm. So, larger mazes will be impossible to deal with (exponential growth of the tree size). Here you will make a

small change to the minimax algorithm enabling it to drastically reduce the branching factor. This speeds up the search.

4.1 QUESTIONS

11. Make a copy of the minimax method, and rename the copy to alfabeta (also rename the recursive calls inside the body of the copy to alfabeta!). Then add the alfa and beta arguments to this new method. The method should now look like this: *public State alfabeta(State s, int forAgent, int maxDepth, int depth, double alfa, double beta)*. Implement the method, and explain your code in the report.

12. Analyse how much gain in terms of visited search nodes (states) alfabeta gives as compared to minimax. Make a table in your report with three columns: search depth, minimax, alfabeta. What seems to be the reduction factor in visited nodes? Explain in short (a couple of lines) the behavior of the reduction factor, using the best-case complexity reduction of alfa-beta given in the lecture.

You have now completed this exercise. Upload the following to your brightspace assignment: *one* zip-file containing the pdf file for your report and the three java source files (Game.java, State.java and Main.java) from your project.