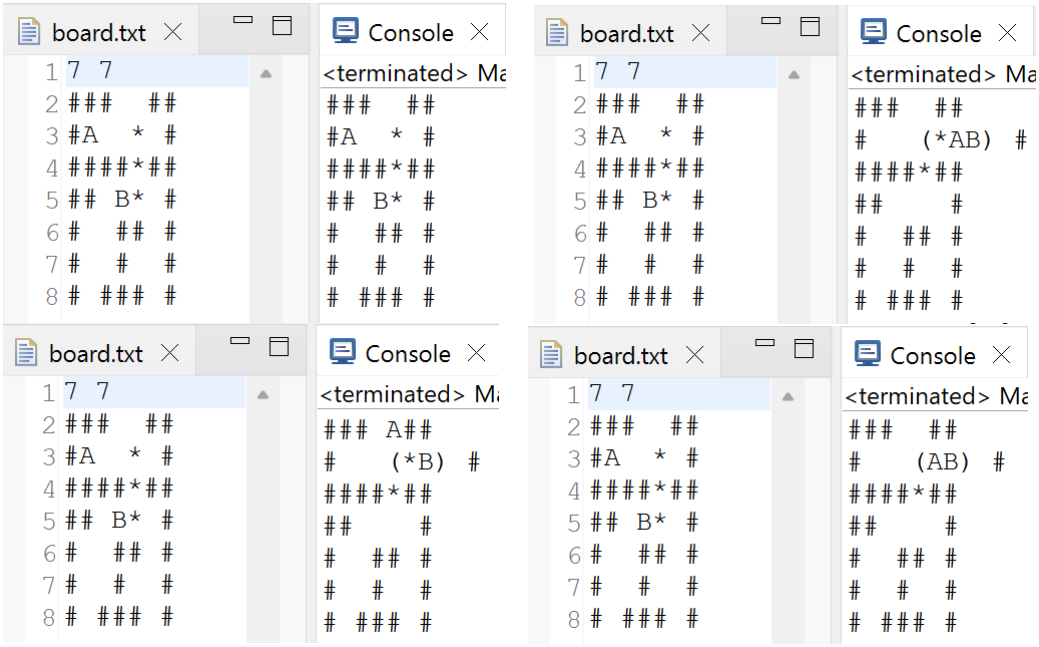
Adversarial search

Group: 43
Members: Maksym Lytovka (s3705609), Ivan Banny (s3647951)

Task 3 (stringification)

In this section, we show the result of the method `public String toString()`:



We are not actually recording the position of the agent in the **public char**[][] board object. That is done to avoid possible overlapping with the food and losing the coordinates of the last. Also, agents' positions are already stored in `State.agentX` and `State.agentY`. To show all the contents of a non-empty cell when an agent (or two) goes there we use brackets with all the insides of the cell when printing it.

Task 4 (copy method)

To implement copying of the object we used serialization. It means, that we decompose the object into an array of bytes and then compose this array into a new object. Consequently, a new object will have all the same properties as the initial object. We chose this option instead of using a constructor due to its compactness and scalability: there is no need to rewrite anything in case you add a new variable. Also, to implement that we had to define class State as a serializable object. We did it in the line:

```
13 public class State implements Serializable {
14     private static final long serialVersionUID = 1L; // the variable used for the serialization
```

Task 5 (legal moves)

For this task, we just implemented a basic check for all the problem's conditions with a series of if's. We also check whether the resulting move makes a character go outside of the board to prevent an error.

Task 6 (move execution)

Here we simply apply the changes to the corresponding variables of the State class: delete food from the board / change agent location / decrease food count / increase agent score / add executed move to the moves vector / pass the agent turn.

Task 7 (leaf detection)

Here we just check whether the game is done, which is either if the current agent has no legal moves or when there is no food left on the map. The first check has to be applied after the move is executed (which is exactly how it's done further in the code) and the food check can be applied at any time because the agent who wins is the one who has the best score.

Task 8 (state valuation)

Here we just return 1 if the other agent is out of moves, -1 if this agent is out of moves and if the food is gone, we return 1 if this agent has a bigger score (has eaten more food) and -1 if the other agent has a bigger score.

In all other cases, we return 0 (game not finished yet / draw).

Task 9 (Minimax)

The `minimaxLastState` method is a recursive function that explores the game by traversing a tree to a certain depth using depth-first search. In other words, `minimaxLastState` is an implementation of the minimax algorithm, which returns the best last state for the given agent. Important to note, that in contrast to the lecture slides it returns the state, but not the value itself. It will be useful in the future explanation. The algorithm searches for the best outcome the following way: when it's the turn of the agent you're looking for (`forAgent`), it maximizes its evaluation by considering all legal moves and recursively exploring their outcomes. In the case of the opponent's move, it minimizes the evaluation. It's important to clone the state before applying moves to it. That's because otherwise other children states will get a state, which was deformed by the previous children and all their children, and the search will fail.

The evaluation of the states is based on the leaf's values for the starting agent. In other words, for the whole algorithm, we take the perspective of the `forAgent` agent and stick to it. Therefore, we try to maximize results when it's our turn and minimize them otherwise.

That is also important to mention, that we shuffle the possible moves in the state. In other words, we shuffle the order of consideration of the children of the parent state. That is an additional feature, which is not essential for the proper algorithm execution. However, it was done for the purpose of preventing the game from falling into an infinite loop. This idea is described more extensively in the "Discussion" section below.

Finally, the `minimax` method is the main method of the algorithm. It heavily relies on the `minimaxLastState` method and may be considered as a nice modification of the last. As discussed earlier, `minimaxLastState` returns the last best state of the search tree, which is not that handy. The main job of the `minimax` method is to extract the next state of the game from the `minimaxLastState` output. It implements that by looking into the `.moves` attribute, which is basically a log of moves from the state, provided to the `minimaxLastState` as a parameter, to the final state of the search tree. Consequently, the first move in the `.moves` vector is the most optimal move for the provided search depth. Important to note though, that as we are taking the 1st move from the log, we need to provide a state with an empty `.moves` attribute to the `minimaxLastState` method. That's why we create a copy of the state, provided for the `minimax` method, and clean the log:

```
public State minimax(State s, int forAgent, int maxDepth, int depth) {
    State sCopy = s.copy();
    sCopy.moves = new Vector<String>();
    State nextState = s.copy();
    State lastState = minimaxLastState(sCopy, forAgent, maxDepth, depth);
    nextState.execute(lastState.moves.get(0));
    return nextState;
}
```

Task 10 (maxDepth change)

By changing `maxDepth` we observed, that the bigger `maxDepth` is, the more purposeful the agents became. That's since with the small `maxDepth` they couldn't always see all the winning states, which leads to choosing a non-optimal strategy.

Although in some cases with the growth of `maxDepth`, the game becomes longer. This happens if some agent starts in a position far away from the food (but it's still accessible) and the other agent. This way, the winning agent calculates that any move will still result in a win (1) and does something random. Unfortunately, the losing agent assumes the same, and also behaves randomly. Occasionally, when the losing agent comes close enough, the winning agent wins the game by eating all the food. Interestingly, something similar happens when the losing agent starts close enough to block a tile of food and the other agent can not eat it first. This way, they both assume that the game will end with 0 (tie) and do something random until occasionally the game is done.

Task 11 (Alphabeta)

The alpha-beta algorithm works the same way as the minimax, but with some enhancement at the end of each maximization and minimization respectively:

```
if (alpha < value)
    alpha = value;
if (beta <= alpha)
    break;
```

```
if (beta > value)
    beta = value;
if (beta <= alpha)
    break;
```

After each consideration of a child state, the alpha value (the minimum score that the maximizing agent is assured of at or above the current level) is maximized with the child state's value if it's the maximizing agent's turn. Conversely, if it's the minimizing agent's turn - the beta value (the maximum score that the minimizing agent is assured of at or above the current level). If beta falls below alpha (beta <= alpha) then the maximizing agent doesn't need to explore any more child nodes for this particular node since they will not come into play during the actual game.

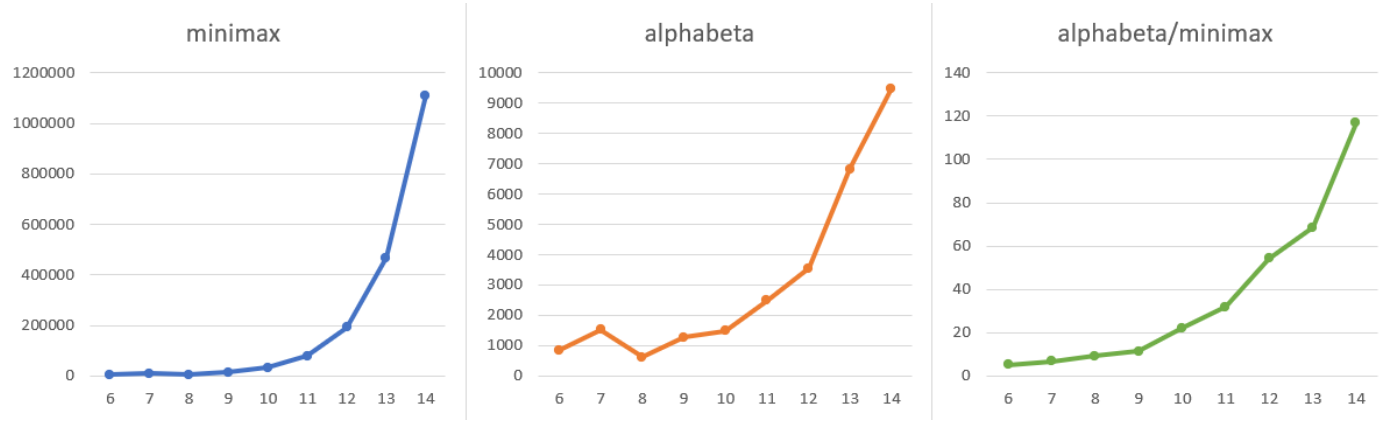
Task 12 (Alphabeta advantages)

For this part, we gathered data on a total of 2800 algorithm runs (100-200 for each algorithm and for each search depth). For each run, the move options are shuffled randomly, after which minimax/alpha-beta is executed.

Sample size	Search Depth	Minimax	Alphabeta	factor
200	6	4505	850	5.3
200	7	10500	1529	6.87
200	8	5696	624	9.13
200	9	14333	1270	11.29
200	10	33056	1495	22.11
100	11	79494	2497	31.84
100	12	191870	3541	54.18
100	13	466699	6817	68.46
100	14	1109798	9490	116.94

From the data above it can be concluded that both algorithms have exponential complexity (which makes sense because increasing the search depth by 1 will multiply the search space size by the average amount of valid moves for the leaf nodes).

Since both minimax and alpha-beta are exponential, the reduction factor will also be exponential, which is confirmed by the data.



Discussion

Shuffling children

As discussed above, shuffling was implemented to prevent the game process from falling into the infinite loop. Let's consider this idea in the example:

Take a look at the Board 1. If we are to run the code implemented by our group with such a configuration, the game will never end. Agent A will be going up and down, whilst B will take one step right and afterwards will also step only down and up. The loop occurs. The assumption was made, that the reason for that is the ordering of the considered options. In our code moves "up" and "down" are examined first. So, as we are traversing the tree with depth > 7 for agent B, we will always pick the first possible move. That's because the leaves of the tree take values only -1 or 0 (B can't win), and such values occur in the subtree, which has children of the root as roots. In other words, whichever of the possible moves B takes, it always will be a possible outcome of 0 value, that's why it considers the 1st possible move and cuts off all the rest. With sufficient depth of search same goes for A (but now there is always a winning outcome). In such a way both agents are stuck in the "first considered move" loop. To prevent that, we shuffle the order of examination of possible moves every time.

```
17 7
2###   ##
3# *A   #
4# ** ##
5##     #
6#  ####
7#  #B #
8# ### #
```

Board 1

Funny note: Due to the randomness in the order of moves scrutiny, sometimes B will decide to place a wall. After two such decisions, it will bury itself down.