Symbolic AI

Assignment 1: Simpel adversarial search

By: Rajiv Jethoe [3490750] and Steven Wang [3730816]

Introduction

In this report we will discuss the assignment given in the course “Symbolic AI”, specifically the assignment on simple adversarial search using the minimax algorithm and alpha-beta pruning. We will discuss our implementation of the minimax algorithm and its optimized version with alpha beta pruning implemented as well.

Debugging state

For the purpose of debugging we had to implement a “ToString” method that overrides the normal “ToString” behavior. Now it returns the complete state of a State object.
Below are two examples, as requested, of two different states where the size of the board and the agent positions are different.

```
# # # # #
# # *   #
# # # * #
#   # # #
# # # # #
Current players turn: 1
Current value of the state: 0.0
Is state a leaf: false
AgentX Position [2, 3]
AgentY Position [3, 2]
Moves made so far: [A : block , B : block , A : right ]
Current players legal moves: [left]
Opponents legal moves: [up, eat]
Current score: [0, 0]
Food left: 2
```

(*Figure 1: 5x5 board*)

```
# # # # # # # #
#           #
# #     #   * #
#       # #   #
#   * #   |   #
#           #
# #     #     #
# # # # # # # #
Current players turn: 1
Current value of the state: 0.0
Is state a leaf: false
AgentX Position [4, 5]
AgentY Position [1, 2]
Moves made so far: [A : up , B : right , A : right ]
Current players legal moves: [right, down, left, block]
Opponents legal moves: [right, down, left, block]
Current score: [0, 0]
Food left: 2
```

(*Figure 2: 8x8 board*)

Copying state for use in minimax

For the assignment we needed to implement a “copy” method in the “State” class that will provide a proper copy of the class. This means no references at all to the obwe want to copy.

To do this we need to fully understand how immutability works. In many languages and also Java, primitives are immutable. Primitives are “Strings”, “Int’s”, “Double’s” etc. This means that when a primitive is created and a programmer wants to re-assign the value of an immutable object a completely new object needs to be created and the new value is assigned to that new object.

An obvious way to create a copy came to mind shown in the figure below.

```
public State copy() {
    return new State(this.board, this.agentX, this.agentY, this.score, this.turn, this.food, this.moves);
}
```

(*Figure 3: a possible copy implementation*)

This however is not correct. If we look back at the beginning of this copy topic we talked about primitives. The board is a matrix, the agents and score are arrays and “moves” is a vector. These are not primitive objects in Java. What happens when we try to make a copy with this code is that a new char matrix and new arrays and a vector will be created in memory with their own memory address, but the data structures will contain references to the original data structure’s primitives that the code is copying from. This means editing one data structure will

result in the editing of the data structure you copied this one from as well which is not the behavior of a true copy of an object.

To truly copy the state with truly new objects with no references to the old state class we are copying from we need to create helper functions for matrices, arrays and vectors. For the matrix the method looks like the figure below.

```
private char[][] deepCopyBoard() {
    char[][] newBoard = new char[board[0].length][board.length]
    for (int i = 0; i < board[0].length; i++) {
        for (int j = 0; j < board.length; j++) {
            newBoard[i][j] = board[i][j];
        }
    }
    return newBoard;
}
```

(*Figure 4: proper copy of the board*)

We create a new board with the width and height of the current board. We loop through our old board and insert all items from the old board into the new board. This will give a true copy of the board which will not be referencing any of the primitives of the old board. Copying a mutable data structure like this will give you a true copy with no double references.

Similar helper functions were written for arrays and vector in the state class, but for these the reader can look at the code that was turned in. This explanation was to illustrate the core issue of copying objects with the goal of a truly new object with no references to the item that is being copied.

Fetching legal moves

For the agent to be able to move there arose a need to implement a function that searches for legal moves to make within the boundaries of our gameboard. Firstly we create three new data structures. A string vector and two int arrays.

```
Vector<String> possible_moves = new Vector<>();
int[] current_agent;
int[] loopPos;
```

(*Figure 5: three new data structures*)

The vector is used to store all legal moves an agent can make, this will also be returned in the end. The first array is the agent this method is called with. We assign “current_agent” to whichever agent is passed into this method call, this is a simple if-statement the reader can look at in the code.

The last array is to compare the position of the loop through the board with our agent's coordinates. If these two are equal to each other it means we are at the position in the board where our agent is located. From here we can check what this agents' legal moves are.

When the loop is at the position of the coordinates of the agent we hit a code block with six if-statements. These six if-statements check whether or not the agent is allowed to make these moves. These if-statements are formulated with the game rules in mind. They are also written such that we don’t get “index out of bounds errors” in certain cases.

What happens in these statements that we want to highlight is that when we enter the if-statements’ body, so if it is a legal move, we add the name of the move to our vector which contains all legal moves for that turn of the agent.

When the for loop finishes running we return the vector containing all legal moves for the agent the method was called with for this turn only.

Fetch legal moves: Helper function

This helper method looks like the figure below.
The helper function simply returns the return value of the main legalMoves(int agent) function with the agents turn already pre-filled in.

```
public Vector<String> legalMoves() {
    return legalMoves(turn);
}
```

(*Figure 7: Legal moves helper function*)

Executing a legal move

Before executing a move, we check which agent's turn it is. If the turn equals 0, we put the coordinates of agent A in the agent array and set the agentChar variable to A so we know which character to append to our "moves" vector so it's immediately clear who made what move. And we do the same for agent B if the turn is 1.

After that, we begin by looping through the whole board until we get to the position of the current agent (because the move needs to be executed from the position of the current agent) from there on, we check for the action and execute it. The i value represents the y coordinate and the j value represents the x coordinate.

We check if the action parameter string is equal to one of the actions in the order of block, up, down, right, left and eat. If so we execute that move and set the turn to the opponent. After each move, the move is added to the moves vector for bookkeeping the moves.

(We realize after implementing legal movies and execute that we didn't have to loop through the matric to find what moves are legal or to execute but it's to close to the deadline for us to risk failure upon change.)

Isleaf method

To tell if the current state is a leaf or not, we start by checking if there is food left. If none is left, we have reached a leaf so we return true.
If agent0 has no legal moves left or when agent1 has no moves left, we have reached a leaf so we return true in those cases as well. Whichever happens first. According to the rules, those are the only game ending scenarios meaning those are the only possible leaves.

```
public boolean isLeaf() {
    if(this.food == 0){
        return true;
    }
    if(legalMoves(agent: 0).isEmpty()) { return true; }
    else return legalMoves(agent: 1).isEmpty();
}
```

Figure 8: isLeaf method.

Local State value

To return the value of the current state we implemented a method that takes as argument an agent and returns a double of either -1, 0 or 1. -1 means the agent is losing in this current state, 0 means in the current state there are no winners and 1 means that the agent is winning in the current state.

We started by checking the amount of food left on the board first. If there is no more food left, the current winner will be determined by which agent has eaten the most food.

We made a function to check the winner based on the score (Figure 9). So for instance if the method is called with agent0, the opponent is set to agent1. We then compare the value of the score array at index of agent0 with agent1 and if it is higher, we return 1, if the score of the opponent is higher, agent0 is losing and we return a -1 and else, the score is even so we return a 0 (tie).

```
private int winnerBasedOnScore(int agent){
    int opponent = minimaxOpponent(agent);

    if(score[turn] > score[opponent]) {return 1;}
    else if (score[opponent] > score[turn]) { return -1; }
    else { return 0; }
}
```

Figure 9: Winner based on score method.

We continue in the value function if there is still food left on the board. We then check on whether the current agent has legal moves left. If not, we return -1 for losing. Else we check if the opponent has legal moves left, if not, we return 1. If none of the cases above are true, it means there is still food on the board and both agents have legal moves left and per rules defined, it will return a 0 for tie as can be seen in the figure below.

```
public double value(int agent){
    if(food == 0) {
        return winnerBasedOnScore(agent);
    }

    int opponent = minimaxOpponent(agent);

    if(legalMoves(agent).isEmpty()) { return -1; }
    else if(legalMoves(opponent).isEmpty()) { return 1;}
    else {return 0;}
}
```

Figure 10: Value method.

Minimax algorithm

Now that the boilerplate is set up we can implement the minimax algorithm as shown in the lectures. There are two main parts to this algorithm, or three, depending how you want to look at it.

- The cut-off for recursion
- The main maximizing and minimizing loop

Cut off for recursion.

A recursive function always needs a cut-off point. Since we are implementing a minimax algorithm with a maxdepth parameter we also include this in our cut-off point.

```
if(depth == maxDepth) {
    return s;
} else if(s.isLeaf()) {
    return s;
}
```

The first thing we need to check for is if maxdepth is hit. If so we hit the maximum depth of the tree we specified for a run. When this is done we immediately return the state and start evaluating from here.

Our second cut-off check is to see if we hit a leaf state. If so, we need to start evaluating all leaf states until all siblings of this leaf are evaluated and we go up and we repeat the recursion once again.

Maximizing and minimizing

Now we come to the meat of the algorithm, the maximizing and minimizing steps. Firstly, we check if “forAgent” is equal to 0, in our implementation this means this is the current player. Since you always want to maximize for yourself this is a logical first step. From here we create a new double, called “best” which is set to “negative_infinity”. This is because a maximizing player needs to start by comparing his first state with the absolute worst possible value for him, which in turn is negative infinity in case of maximizing. The same was implemetned for the else-statement, but here we create the double with value “positive_infinity”. For the minimizer the situation is the opposite of the maximizer. As the minimizer your absolute worst possible value is infinity, since you are trying to get your evaluation value as low as possible.

Next up we need to create another “best” value, but this time for a “best state”. This is because we here differ from more conventional implementations of the minimax algorithm. We do not return a number, or a single action that needs to be performed, we return an entire class object. Here we set our new “bestState” variable to “null”. Since this “bestState” variable is never used for evaluation it will always be overridden no matter what is in it given that “bestState” can never be a leaf state. We can view it as the variable “double best”. It will always be overridden on the first evaluation run.

Next up we fetch all legal moves (children) of the parent (node). We iterate over them in a DFS manner. Meaning, if we have a tree data structure (which this problem is), we go all the way down on the left side of the tree (first child of every parent node) until we hit a leaf state or we hit the specified max depth, if so we backtrack all the way back up to the top and repeat this process until we run out of children.

Following the start of our iteration process, we first start with creating a new variable called “copyState” which is used to assign a copy of the current state we are in. This procedure is facilitated with the “copy” method we discussed earlier in this report. We copy this state to execute, possibly nonoptimal moves on. We only want to continue with the state that gives us the best outcome, the “bestState (referring to our variable we created)”, so we create a copy of the current state to execute our moves on and evaluate. If it turns out the move we executed was not optimal in leading us to the best optimal state we simply discard this variable and start over with a fresh copy of our state we are iterating over.

Moving on, we execute the current move we are at in the iteration process on the copied state (very important). This is a fairly well discussed topic we talked in depth about in an earlier chapter so we refer you back to this part in the report.

Now we make a recursive call to our method minimax with our copied state on which we executed our move. Recursively calling minimax with this new state will repeat the process described above until either a leaf state is hit or the max depth variable specified is reached. Every time we recursively call minimax we also set the “forAgent” variable to our opponent, because this process needs to be repeated for our opponent's moves in the tree and since we are maximizing for ourselves we need to minimize for our opponent. Another important note, we increase the depth by one. We increase the depth by one so we can keep track of how deep we are in our tree since we do want to cut-off the search at some point. Eventually we hit a leaf state and we’ll return a state (which is a leaf state). This state is directly assigned while making the recursive call. We use this variable “minimaxEvalState” to evaluate.

Eventually after traversing the tree we hit our max depth or we encounter a leaf state. This is one of the cut-off conditions described at the start of this chapter. Now the leaf state is returned so it can be evaluated. The “value” method is called on the returned state with “forAgent” as input. Now the value method can check if this is a winning, losing or a tie state for “forAgent”. The method will return either 1.0, -1.0 or 0.0 respectively, these values are stored in a variable, double “minimaxEvalState”.

The algorithm now needs to check if the evaluated state is bigger or equal to negative infinity in the first iteration and after that, the evaluated state is compared to the current best for the maximizing player and for the minimizing player, we check if the evaluated state is smaller or equal to positive infinity in the first iteration and after, the evaluated state is compared to the current best state for the minimizing player. If it is either bigger or equal or smaller or equal respectively we set the evaluated state value as the new best and also set our new evaluated state object as our new “bestState” object. Since this implementation requires a state to be returned we need to return our best possible state, that is why we keep track and update the “bestState” variable.

If the evaluated state value is bigger than our best score for the maximizing player and smaller than our best score for our minimizing player it means this state can be discarded. The “bestState” is not updated and additionally we remove the move made from our “moves” vector, which keeps track of what moves are made to get to a certain state. This is done for the sake of clarity. If this was not done the vector would end up being immensely huge, especially if the max depth were to be increased. Now a vector of moves is returned that will specify how to get to the state of the returned state.

This process will repeat for the leaf states until there are no leaves left at the absolute bottom, now the backtracking starts all the way back up the branch until the root is reached and the process starts all over again until all children have been visited.

In the end we return the “bestState” that was being updated every time a better state came along.

Minimax various max depth value analysis

Depth	Outcomes
7	[A : block , B : block , A : right , B : left , A : eat , B : block]
9	[A : block , B : block , A : right , B : left , A : eat , B : block]
11	[A : block , B : block , A : right , B : left , A : eat , B : block]
13	[A : block , B : block , A : right , B : left , A : eat , B : block]

When the max depth is increased from 7 to 13 there is no observed difference in the end state and the moves to this state. To argue that this is a correct outcome we need to go over minimax and how it selects an end state. In the end the implemented minimax algorithm was implemented without any heuristics on what is a good state, a better state or a bad state or worse state. This means minimax always takes the last end state out of the tree. This is shown in our table, because block is the last child of A's first set of children at depth 0, this is the root. So apparently no matter how deep the tree is made, the last end state it sees for A is the one we keep getting back.

An observation was made that if the children of the tree were restructured the end state would wildly differ seemingly following this restructuring. The conclusion can be made that there is a relation between the structure of the tree and the final end state that is returned in the vanilla implementation of minimax.

Alpha-beta implementation

The implementation of alpha-beta pruning is the addition of a small block of code underneath the inner if-else-statement for the maximizer and minimizer. Alpha and beta are first added to the method signature and in the recursive calls as advised in the assignment. Alpha is set to negative infinity since this is the worst possible outcome for the maximizer and as such beta is set to positive infinity which is the worst possible outcome for the minimizer.

In code it can be observed that for the maximizer the alpha is being set to the maximum of alpha and the evaluated state's value. This makes sense since alpha was set to negative infinity and alpha is being compared and set for the maximizer. In the first run of evaluation, alpha will always be updated since nothing is worse than alpha's initial value.

Now it's the minimizers turn. For the minimizer beta is being set to the minimum of beta and the evaluated state's value. Since beta is positive infinity it's the worst possible outcome for the minimizer which is correct. Again, during the first evaluation step alpha will always be updated since nothing is worse than positive infinity in perspective of the minimizer.

Eventually, both the maximizer and minimizer are checking if beta is smaller or equal to alpha. If this is true we break out of the for loop which is akin to pruning the branch the algorithm is in at the moment. This means there is a better or equally good solution found earlier in the tree. So, considering the current branch is a waste of computation.

Reduction in visited search nodes with Alpha-beta

Search depth	Minimax O(b^d)	Alpha-Beta O(b^(d/2))
7	2016	246
9	12918	679
11	76700	1815
13	460348	4770
15	2602930	12247

Visited states at depth of 7: Minimax vs Alpha-beta difference. **2016 / 246 ≈ 10.**
Visited states at depth of 9: Minimax vs Alpha-beta difference. **12918 / 679 ≈ 20**.
Visited states at depth of 11: Minimax vs Alpha-beta difference. **76700 / 1815 ≈ 40**.
Visited states at depth of 13: Minimax vs Alpha-beta difference. **460348 / 4770 ≈ 96**.
Visited states at depth of 15: Minimax vs Alpha-beta difference. **2602930 / 12247 ≈ 212**

From the calculations it can be observed that there roughly seems to be an exponential growth difference between the visited states for minimax and alpha beta. This in turn means that there is an exponential reduction factor in visited nodes in respect to alpha beta. So the difference between the visited states from minimax and alpha beta are exponentially growing apart.

The reduction factor's optimal big O is O(b^(d/2)). We can see that we are fairly close to this complexity since we have an exponential growth difference. We can conclude from this that the tree is fairly well ordered since the Big O complexity shown above is of an ideally ordered tree.

Furthermore, in this implementation the branching factor is not a static number since some moves might be illegal at certain points. This makes a straight up comparison with the ideal complexity of the alpha-beta algorithm very difficult, but it seems like the growth difference between visited nodes is at least exponential and even a bit more in this implementation's case.