Assignment A1: Team 40

Fleur Ensink op Kemna Luc Siecker Leon Vreling

December 1, 2022

1 Agent Description

The Artificial Intelligence agent that was created employs a straightforward Minimax implementation with the extension of iterative deepening. Thus, creating a strategy in which a depth-first search is run iteratively over a depth-limited tree. This depth increases on every iteration. The best move is detected for depth 1, this move is saved and then during every iteration, all the branches one depth deeper are analyzed to see if the move is optimal. The best move is updated accordingly. Below the pseudocode for the initialization of the iterative deepening search is depicted.

Algorithm 1: Iterative deepening search

Result: Proposed move
initialization;
for *depth in range(*0*, number of empty cells)* **do**
  move, value = MINIMAX(game state, isMaximisingPlayer, depth);
  propose move;
end

Before the minimax function was created, several other functions were defined. Firstly, to ensure that there is always a move proposed within the time frame, the first possible move from the list of possible moves is proposed. Then, the `computeBestMove` function is employed. This function checks the board for empty cells and returns these in a list. Furthermore, it contains the `getAllPossibleMoves` function, which finds a list of all possible moves for a given game state. This is done by checking whether the given value at the given position is valid. To check for validity, a check for every row, column, and block is done in the given game state.
Furthermore, an `assignScore` function is implemented in order to assign scores to moves. This was done by checking whether the given position is the only empty cell in a row, column, or block. These functions return either true or false for every move. After which, the number of true and false statements for that move are added to each other, this determines the score of the move proposed to the function.
The implemented `evaluate` function finds the best move for a given game state. The best move is initialized as the first possible move from the `getAllPossibleMoves` function. After which, for every move in all possible moves, a value is assigned to the move utilizing the assign scores function, this value is then evaluated. If the value is better than the value corresponding to the original best move, then the original best move is replaced by the newly proposed move.
Then, the `minimax` search function is implemented. This function creates a tree to a given depth and returns the move a node has to make in order to achieve a certain score. The function takes depth as a parameter, which is by default set to zero, as the depth will start at depth zero. After this, the depth will iteratively be updated in order for the tree to look one depth deeper. Furthermore, the function takes a boolean parameter that determines whether the player is maximizing or minimizing. In this way, it can determine the best move for the opponent and evaluate an overall best score for the agent itself. It also takes in the parameters max depth and current score, which determine the depth at which to terminate the tree search, and a score value that defines the score of the parent

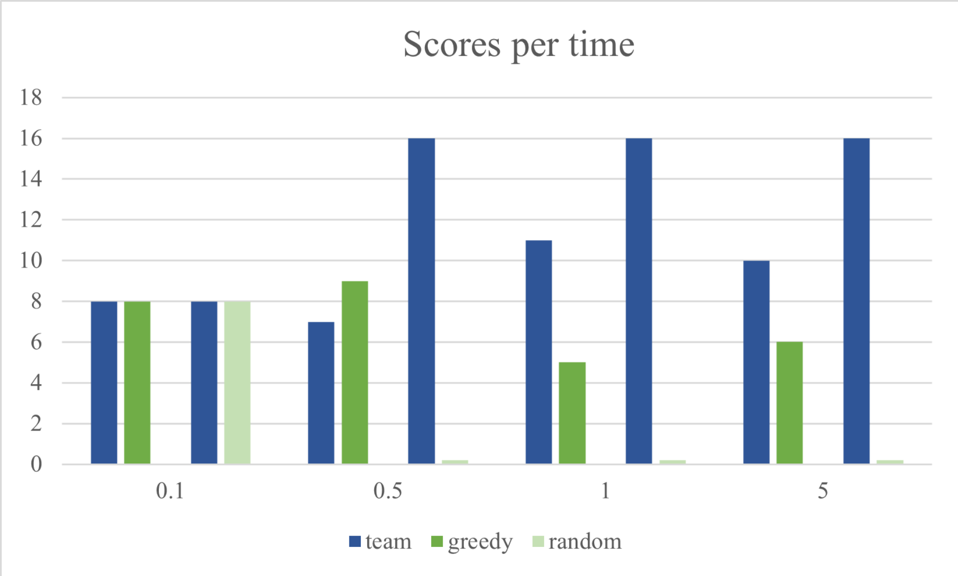

Figure 1: Cumulative wins of a player grouped by the compute-time of the game.

node respectively.
The function starts by stating that when there is no valid move the maximizing player should return a value of minus infinity and the minimizing player a value of infinity. Furthermore, if the tree is in the final leaf it returns a move and a value.
The function loops over all the possible moves in the state in order to search the tree for all the children of the current node, after which the scores of all of those children are stored in a list together with the move that initiated them.
These scores are then evaluated for both the maximizing player and the minimizing player, taking the maximum and the minimum value from the list of scores, respectively. The function returns the move and the determined value is added to the original current score. This score together with the move that initiated the branch for this score is returned back and this move is proposed in the game.

2 Agent Analysis

The performance of the agent is tested against two, by the course provided, agents. The AI agent plays against a greedy player and a random player two times, where the agent starts one time as first and one time as second. The time that is given to propose a move is varying between 0.1, 0.5, 1, and 5 seconds and the game is played on all 8 different boards. This results in 128 conditions for which a game is played (opponent x starting order x time x board)
For all conditions, one test is run and the results were measured by registering which player has won the game. The score (or difference in score between the players) is not registered.

2.1 Results

The cumulative wins per compute-time can be seen in figure 1. This table shows that the amount of wins is the same for time 0.1 when playing against both the random and greedy opponent.
When the agent plays against a random player, for all other compute times the AI agent wins all the games. When playing against a greedy player, the AI agent loses slightly more games when the compute time is 0.5 seconds. For 1 and 5 seconds of compute time, the agent wins from the greedy player.

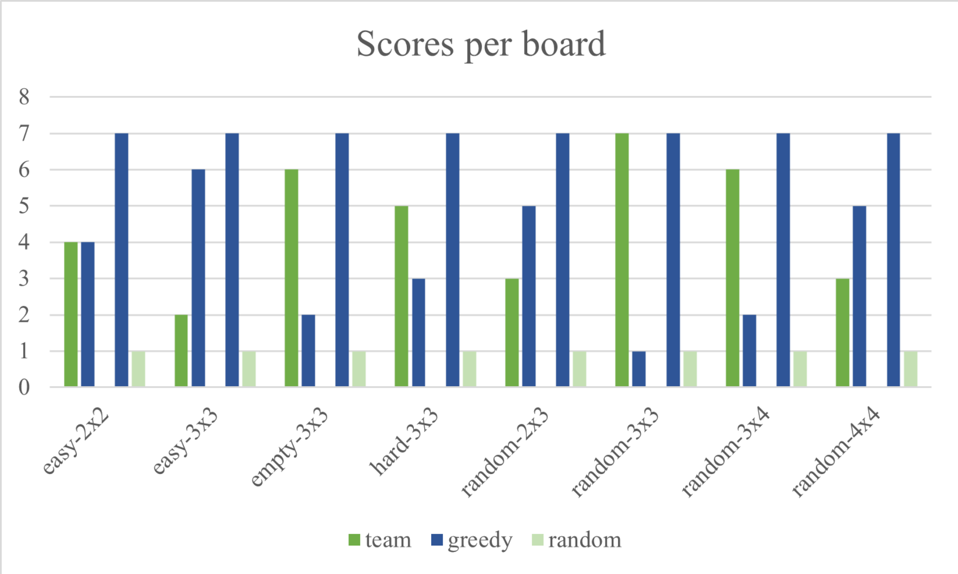

Figure 2: Cumulative wins of a player grouped by a board.

The cumulative wins of each player per board can be seen in figure 2. For all boards, it is clear that the AI agent wins from the random player with a win-loss score of 7-1. The performance per board of the AI agent versus the greedy player is less clear. On 4 out of 8 boards, the greedy player plays better than the AI agent and on 1 board the win-loss score is 4-4.

2.2 Interpretation

When we interpret the results we can see that at 0.1 seconds of compute time, the cumulative wins are equal for all players. This is due to the fact that none of the players nor the AI agent proposes a move within 0.1 seconds and by following the rules, the second player will then always win. For the compute times of 0.5, 1, and 5 seconds and playing against a random player, the AI agent wins every game and this is likely due to the fact that we are maximizing the score until a certain depth, which is determined by the compute time. When the AI agent plays against the greedy player, the cumulative wins at 0.5 seconds of compute time is lower for the AI agent. A possible explanation for this lower performance is that the proposed move on a depth of 0 is not the best move at that moment. However, when we have more compute time and therefore search more depths, the proposed move is better, and therefore, winning is more probable.

The difference in the performance of the AI agent between the different boards is not as clear. When playing against a random player, the agent wins everything. However, when the AI agent is playing against a greedy player, most of the games are lost on nearly all of the boards. A possible explanation is that the agent consistently loses on boards that have an odd number of cells in the width of a block. The only board that is not following this statement is the easy-3x3 board for which the AI agent wins against the greedy player. This observation can be explained by the fact that the agent can search the tree deeper since it is an easy-3x3 board. For the other boards, a possible explanation is that the sudoku is being filled in from the top left to the top right if it is not given enough time to search the tree, which causes the greedy player to be able to choose a move that gets more points.

3 Reflection

The AI agent that has been made has a few strong points. Firstly, the agent looks at the possible moves, excluding the currently known TabooMoves. Knowing these moves, it gains insight into how good a certain move is by searching a tree using iterative deepening. At each depth, a move is proposed and a new deeper tree is created until the compute time is over. These aspects all strengthen the agent's performance.

At this moment, a weakness of the AI agent is the first proposed move. The move is not found fast enough in order for it to be proposed before the compute time of 0.1 seconds is over. Furthermore, iterative deepening creates a whole new tree for each iteration, even if the parent nodes have already been computed once. In order to further improve the calculation time, α-β-pruning could still be implemented. This could be done in order to stop searching a branch that is with certainty not better than the other branches.

Python files

Underneath, the Python file for the AI agent can be seen, in order for it to be evaluated. This file (`sudokuai.py`) is located in the folder for the agent, which can be accessed as a player by the `simulate_game.py` file.

Code Listing 1: `sudokuai.py`.

```python
#   (C) Copyright Wieger Wesselink 2021. Distributed under the GPL-3.0-or-later
#   Software License, (See accompanying file LICENSE or copy at
#   https://www.gnu.org/licenses/gpl-3.0.txt)

import copy
import random
import time
import typing
from competitive_sudoku.sudoku import GameState, Move, SudokuBoard, TabooMove
import competitive_sudoku.sudokuai


class SudokuAI(competitive_sudoku.sudokuai.SudokuAI):
    """
    Sudoku AI that computes a move for a given sudoku configuration.
    """

    def __init__(self):
        super().__init__()

    def compute_best_move(self, game_state: GameState) -> None:
        N = game_state.board.N

        def checkEmpty(board) -> list[typing.Tuple[int, int]]:
            """
            Finds all the empty cells of the input board
            @param board: a SudokuBoard stored as array of N**2 entries
            """
            emptyCells = []
            for k in range(N**2):
                i, j = SudokuBoard.f2rc(board, k)
                if board.get(i, j) == SudokuBoard.empty:
                    emptyCells.append([i,j])
            return emptyCells

        def getAllPossibleMoves(state) -> list[Move]:
            """
            Finds a list of all possible moves for a given game state
            @param state: a game state containing a SudokuBoard object
            """

            def possible(i, j, value) -> bool:
                """
                Checks whether the given value at position (i,j) is valid for all regions
```

```python
        and not a previously tried wrong move
    @param i: A row value in the range [0, ..., N)
    @param j: A column value in the range [0, ..., N)
    @param value: A value in the range [1, ..., N]
    """

    def checkColumn(i, j, value) -> bool:
        """
        Checks whether the given value at position (i,j) is valid for the column
            i.e. finds if the value already exists in the column
        @param i: A row value in the range [0, ..., N)
        @param j: A column value in the range [0, ..., N)
        @param value: A value in the range [1, ..., N]
        """
        for col in range(N):
            if state.board.get(col, j) == value:
                return False
        return True

    def checkRow(i, j, value) -> bool:
        """
        Checks whether the given value at position (i,j) is valid for the row
            i.e. finds if the value already exists in the row
        @param i: A row value in the range [0, ..., N)
        @param j: A column value in the range [0, ..., N)
        @param value: A value in the range [1, ..., N]
        """
        for row in range(N):
            if state.board.get(i, row) == value:
                return False
        return True

    def checkBlock(i, j, value) -> bool:
        """
        Checks whether the given value at position (i,j) is valid for the block
            i.e. finds if the value already exists in the block which holds (i,j)
        @param i: A row value in the range [0, ..., N)
        @param j: A column value in the range [0, ..., N)
        @param value: A value in the range [1, ..., N]
        """
        x = i - (i % state.board.m)
        y = j - (j % state.board.n)
        for col in range(state.board.m):
            for row in range(state.board.n):
                if state.board.get(x+col, y+row) == value:
                    return False
        return True
```

```
            return not TabooMove(i, j, value) in state.taboo_moves \
                    and checkColumn(i, j, value) \
                    and checkRow(i, j, value) \
                    and checkBlock(i, j, value)

        return [Move(cell[0], cell[1], value) for cell in checkEmpty(state.board)
                    for value in range(1, N+1) if possible(cell[0], cell[1], value)]

    def assignScore(move, state) -> int:
        """
        Assigns a score to a move using some heuristic
        @param move: a Move object containing a coordinate and a value
        """

        def completeColumn(i, j) -> bool:
            """
            Checks whether the given position (i,j) is the only empty square in the column
            @param i: A row value in the range [0, ..., N)
            @param j: A column value in the range [0, ..., N)
            """
            for col in range(N):
                if state.board.get(col, j) == SudokuBoard.empty \
                        and col != i:
                    return False
            return True

        def completeRow(i, j) -> bool:
            """
            Checks whether the given position (i,j) is the only empty square in the row
            @param i: A row value in the range [0, ..., N)
            @param j: A column value in the range [0, ..., N)
            """
            for row in range(N):
                if state.board.get(i, row) == SudokuBoard.empty \
                        and row != j:
                    return False
            return True

        def completeBlock(i, j) -> bool:
            """
            Checks whether the given position (i,j) is the only empty square in the block
            @param i: A row value in the range [0, ..., N)
            @param j: A column value in the range [0, ..., N)
            """
            x = i - (i % state.board.m)
```

```python
        y = j - (j % state.board.n)
        for col in range(state.board.m):
            for row in range(state.board.n):
                if state.board.get(x+col, y+row) == SudokuBoard.empty \
                        and (x+col != i or y+row != j):
                    return False
        return True

    completedRegions = completeRow(move.i, move.j) + completeColumn(move.i, move.j) + completeBlock(move.i, move.j)

    if completedRegions == 0:
        return 0
    if completedRegions == 1:
        return 1
    if completedRegions == 2:
        return 3
    if completedRegions == 3:
        return 7

def evaluate(state) -> typing.Tuple[Move, int]:
    """
    Finds the best Move for the given game state
    @param state: a game state containing a SudokuBoard object
    """
    best_value = -1
    ## Initalize the best move as a random possible move
    best_move = random.choice(getAllPossibleMoves(state))
    for move in getAllPossibleMoves(state):
        value = assignScore(move, state)
        if value > best_value:
            best_move = move
            best_value = value
    return best_move, best_value

def minimax(state, isMaximizingPlayer, max_depth, current_depth = 0, current_score = 0) -> typing.Tuple[Move, int]:
    """
    Makes a tree to a given depth and returns the move a node needs to make to get a certain value
    @param state: a game state containing a SudokuBoard object
    @param isMaximizingPlayer: a boolean value which determines if the player is maximizing
    @param max_depth: a depth value which defines when to terminate the tree search
    @param current_depth: a depth value which defines the current depth
    @param current_score: a score value which defines the score of the parent node
    """
```

```
            # If there are no possible moves (when no move is valid), return a infinite value
            if len(getAllPossibleMoves(state)) == 0:
                if isMaximizingPlayer:
                    return None, float("-inf")
                return None, float("inf")

            # If the tree is in the final leaf, return a move and value
            if len(getAllPossibleMoves(state)) == 1 or current_depth == max_depth:
                move, value = evaluate(state)
                if isMaximizingPlayer:
                    return move, value
                return move, -value

            scores = []
            # Loop to search the tree for all children of the current node
            for move in getAllPossibleMoves(state):
                score = assignScore(move, state)
                if isMaximizingPlayer:
                    total_score = current_score + score
                else:
                    total_score = current_score - score
                state.board.put(move.i, move.j, move.value)
                result_move, result_value = minimax(state, not isMaximizingPlayer, max_depth, current_depth+1, total_score)
                scores.append((move, result_value))
                state.board.put(move.i, move.j, SudokuBoard.empty)

            # [print((str(i[0])) + " scores a value of " + str(i[1])) for i in scores]

            if isMaximizingPlayer:
                move, value = max(scores, key=lambda score: score[1]) # Return the state with the maximal score
                # print("Optimal move for depth " + str(current_depth+1) + " is " + str(move) + " with a total reward of " + str(value))
                return move, value + current_score
            move, value = min(scores, key=lambda score: score[1])
            # print("Optimal move for depth " + str(current_depth+1) + " is " + str(move) + " with a total reward of " + str(value))
            return move, value + current_score

        # start_time = time.time()

        # Intialize a random possible move as return
        # (to ensure we always have a return ready on timeout)
        self.propose_move(getAllPossibleMoves(game_state)[0])

        # Search the minimax tree with iterative deepening
        for depth in range(0, game_state.board.squares.count(SudokuBoard.empty)):
```

```
            move, value = minimax(game_state, True, depth)
            self.propose_move(move)
            # intermediate_time = time.time()
            # print("Proposed move: " + str(move) + " | Total reward: " + str(value))
            # print("\n\nTime for depth " + str(depth) + ": " + str(round(
                intermediate_time - start_time, 3)) + "
                seconds \n\n")
```