Assignment A3: Team 33

Lucas van Osenbruggen   Martin Miksik   Luca Mainardi

1 Agent Description

We implement an agent for the game *Competitive Sudoku*, in which two players must complete a Sudoku puzzle one turn at a time and receive points for completing regions (Krak [2023]). Two important rules are: A) The agent cannot suggest *taboo moves*, which are listed by the game engine. Making such move, which would render the puzzle unsolvable, results in losing the game. B) If a player makes a move that makes the Sudoku unsolvable and the move is not yet marked as a taboo move, that move is added to the taboo list, and the turn passes to the opponent. We design two agents, based respectively on the *Minimax* algorithm and on a *Monte Carlo tree search (MCTS)* strategy.

1.1 Iterative Deepening Minimax Agent

The Minimax agent, which can be found in Listing 2, uses a search strategy based on the *Minimax* algorithm (Polak [1989]). This is implemented in the function `minimax`. This strategy tries to maximise the score of the player, taking the potential moves of the opponent into account by searching a *game tree* of all subsequent moves. Since the turn time is indeterminate, we cannot reliably search to arbitrary depth. For a board of size $N \times N$, there are $\mathcal{O}(N^3)$ initial moves possible for the first turn, and $\mathcal{O}(N^3!)$ possible games. To deal with this time constraint, our agent will always start by proposing a random legal move. After this, the agent searches the game tree until some iteratively increasing depth – a strategy which is called *iterative deepening* (Korf [1985]). This is implemented in the function `compute_best_move`. Searching until a limited depth requires an evaluation function of an incomplete game. To this end, we define the following evaluation score:

$$\Delta(s_{max}, s_{min}) = s_{max} - s_{min} \tag{1}$$

Where s_{max} and s_{min} represent the game scores at the maximum search depth of the maximising and minimising player respectively. This evaluation score is optimal if the agent can search the entire game tree. This iterative deepening strategy ensures that the agent will propose the best move it can within the time limit. Furthermore, given more time, the agent will find better moves.

To address the limited time and improve performance, we implement several heuristics:

Alpha beta pruning is a strategy in which branches of the search tree that cannot lead to a better score than found so far are not searched (Knuth and Moore [1975]). This heuristics reduces the search space, while still producing optimal moves.

Move ordering heuristics is used once per turn to sort the list of initial moves based on the number of regions they allow to complete. Moves that complete three regions have the highest priority and are placed at the front of the list, followed by those that complete two, those that complete one, and finally, those that complete no regions. Once the sorting of the moves has been completed, the first move in the list is proposed (the one that *likely* leads to a greater increase in score) and the Minimax search with iterative deepening

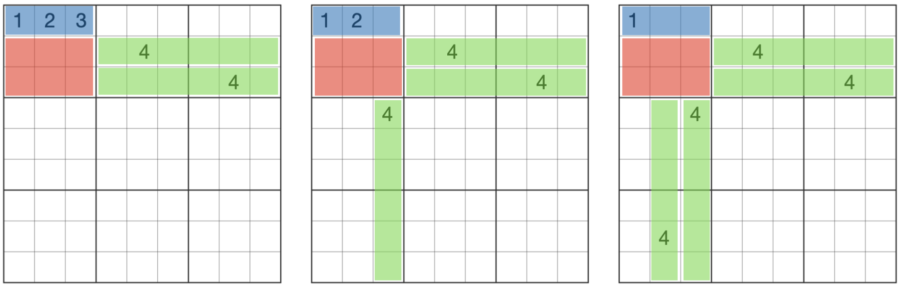

Figure 1: Examples of unsolvable Sudoku when the examined block row (blue region) has 3 filled cells (left), two filled cells (centre) and one filled cell (right). In all three cases the value 4 makes Sudoku impossible.

begins with a depth limit 2; that allows the agent to consider the opponent's next move. Te search starts from the moves that are presumably best, since the list of moves is sorted. This heuristics is implemented in the function `order_moves`.

Propose unsolvable moves heuristic consists of calculating a list of moves that make the Sudoku unsolvable (which we call *unsolvable moves*) and proposing one of them every time, the best score returned by the Minimax search is negative. The agent, thus, exploits the game rule that allows the player to skip a turn, deliberately proposing a move that makes the game unsolvable if all evaluated moves benefit the opponent. Furthermore, unsolvable moves are not evaluated by the *Minimax* to reduce the search space and save computation time.

The calculation of the list of unsolvable moves is done only in the *mid game*, i.e., when more than 20% and less than 85% of the cells on the board are full[1].

The calculation of the list is computationally expensive, therefore we save the result to disk. At each subsequent turn, the list is loaded from the disk and updated (e.g. removing illegal moves). If the list is empty, a new list of unsolvable moves is re-generated[2]. Nonetheless, the computation time associated with unsolvable moves is still an issue, given that the duration of a turn is unknown. If there isn't sufficient time to complete the list of unsolvable moves. Under these circumstances, the agent repeatedly attempts to calculate this list in subsequent turns. As a result, it never moves to the Minimax search. To address this, the agent includes a check at the beginning of each turn. It determines whether it previously started but failed to complete the list. If this is the case, the agent repetitively increases the required cell fill rate by 10%.

This heuristics is implemented in the function `compute_best_move`. The algorithm that determines whether a game state (with an N x N board) is impossible is outlined in Algorithm 1 and is implemented in the function `is_unsolvable`.

In short, we examine all the rows of the blocks (3 rows made up of three cells in the case of a 9x9 Sudoku with 3x3 blocks). Any of the values not present in the row (blue region in Figure 1) could invalidate the Sudoku. If one of these values is not present in the same block as the row considered (red region in Figure 1), but is present in each of the rows not considered and of the columns corresponding to the empty cells (green regions in Figure 1), then none exists legal cell for that value in the block, and therefore Sudoku is impossible.

It is important to note that not all impossible states are detected by this algorithm. For example, the same check that was done on the block rows could also be done on the columns. However, for our purposes full list of unsolvable moves is not needed and

[1] Our testing showed that outside this range there are very rarely moves that make Sudoku unsolvable.

[2] As noted later our unsolvable moves list is not exhaustive

Algorithm 1 Impossible Sudoku. Function name correspond to the implementation in Listing 4

```
function is_impossible(board)
for all block in the board do
  for all block_row in the block do
    for all value in range [1, N] not present in the block_row do
      if if value is not in the red region and value is in all green regions then
        return true
      end if
    end for
  end for
end for
return false
```

would further increase the computation time required. Additionally, our tests show that the described algorithm can find *at least one* unsolvable move at almost any point in the game.

1.2 Tree-saving MCTS Agent

The second agent uses a *Monte Carlo tree search (MCTS)* strategy (Swiechowski et al. [2021]). Contrary to the Minimax agent from Section 1.1, which is an exhaustive search method on a game tree, the MCTS agent plays complete games with random moves, i.e., *random playouts*, and thus randomly samples different paths in the game tree. Since MCTS completes entire games, games are evaluated by their final score Δ as follows:

$$\Delta(s_{max}, s_{min}) = \begin{cases} 1 & \text{if } s_{max} > s_{min} \\ 0 & \text{else} \end{cases} \tag{2}$$

Where s_{max} is the score of the maximising player and s_{min} of the minimising player. MCTS estimates the quality of different moves based on the average score of a given move and improves this estimate after every playout. Given a longer turn time, this agent is thus expected to find better moves.

Furthermore, the MCTS agent makes better use of the available turn time. This is best illustrated with a hypothetical example of a game with a 1 second turn time. We will assume that the board is such that the Minimax agent can search until a depth of 5 in this case. However, because it does not know this, the agent will first search until lower depths, and thus spend its time searching to depths 2 and 3 before losing the turn while searching depth 4. In this case, the Minimax agent searched too shallowly and also wasted time. MCTS on the other hand, will play 99 random playouts, proposing a better move each time, before losing the turn in the 100th playout. In this example MCTS did not do any duplicate work, nor is there a high cost of losing the turn at the wrong moment.

The implementation of the MCTS agent can be found in Listing 3. The agent follows the basic structure of Monte Carlo tree search, which is outlined in Algorithm 2 with the function names corresponding to those in the listing. Because of the difference in strategy, the implementation is quite different from the Minimax agent, although the generation of legal moves and updating of the state are similar. The agents do share some utility functions which are listed in Listing 4. These functions perform game-specific logic that is not related to the strategy of the agent. Unit tests have been written for these functions as well, to ensure their correctness.

In MCTS a *Monte Carlo game tree* describes the quality of different states by their cumulative score q and how often they were evaluated n. Our agent extends basic MCTS by saving this data structure between turns. When the turn starts the agent loads the data structure and moves in the tree according to the move it proposed in the previous turn and the move performed by the other agent[3]. When these moves are not in the tree it creates a new root node. This allows the agent to reuse work the previous turn if the state evaluation tree is deeper than 2 levels.

Algorithm 2 Monte Carlo Tree Search of the best move in a given turn. Function names correspond to the implementation in Listing 3
.

function `iterate`(*root*)
while turn lasts **do**
  $leaf \leftarrow$ `select`($root$) ▷*Repeatedly select node with highest UCB value until a leaf*
  $leaf \leftarrow$ `expansion`($leaf$) ▷*Add new child nodes and select one under some conditions*
  `apply_move_on_node`($leaf$) ▷*Update state by applying all moves in path*
  $\Delta \leftarrow$ `simulation`($leaf$) ▷*Apply random moves until the game is over, then calculate the score (Equation 2)*
  `backpropagation`($leaf$, Δ) ▷*Update cumulative score and visited of nodes*
  $bestchild \leftarrow \arg\max_{child \in root.children}(\frac{child.q}{child.n})$
  return bestchild ▷*Propose move*
  if number of turns is a multiple of 10 **then**
    save *root* recursively
  end if
end while

To summarise, we propose an MCTS agent that estimates the best moves by randomly sampling complete games from the state space and saving the estimates between turns.

2 Agents Analysis

In our evaluation of AI agents Minimax and MCTS, we focused on their performance relative to computational time constraints and varying game board configurations. We run the Minimax agent against the A2 agent, Greedy Player, and MCTS agent across different boards at 0.1, 0.5, 1, and 2-second intervals for 20 games each. This resulted in 12 distinct outcomes. We also run a similar set of experiments for the MCTS agent. Additionally, we test the performance of hyper-parameters for the MCTS agent. Finally, we do an analysis of the number of Minimax calls and MCTS iterations as a function of the time limit, to test the agents' ability to make the most of the time available and their robustness to different time conditions

2.1 The Minimax Agent Analysis

Performance Over Time Figure 2 illustrates how the agents' average win rates vary with the amount of computation time provided. We observed the following:

- Against the A2 agent, the Minimax agent outperforms A2 when the computation time is limited, suggesting that the Minimax heuristics improve performance. With more computation time the impact of heuristics disappears and the agents end up in a draw.

[3]An exception is a taboo move, which was proposed but not accepted and thus does not change the state.

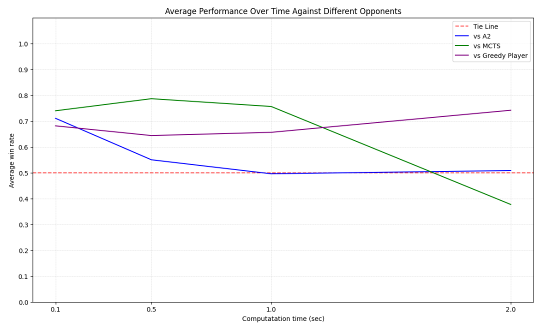

Figure 2: Average performance of A3 agent over time

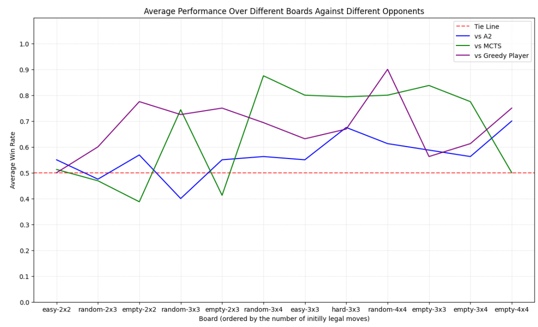

Figure 3: Average performance of A3 agent over different board configurations

- When facing the MCTS agent(C=2)[4] opponent, the Minimax agent starts with a high win rate which decreases as computation time increases. This suggests that MCTS may benefit more from additional computation time than A3. This is surprising since we would expect the MCTS to perform better with limited time. Minimax with enough time should provide *optimal* moves.
- Against the Greedy Player, the A3 agent's win rate increases over time, highlighting that the A3 agents can explore larger game trees and find *better* moves.

Performance Across Board Configurations Figure 3 presents the agents' win rates across different board setups' average overall times. We observed the following insights:

- Performance is generally inconsistent across different board configurations, suggesting that our A3 agent's implementation may not generalise well and thus its performance is board configuration dependent.
- The A3 agent seems to perform better on larger, empty boards (e.g., empty-3x4, empty-4x4) against the A2 agent. Indicating again that the A3 implementation better exploits the difference between *early, and late* game.
- The A3 agent generally outperforms all opponents (the win rate is above the tie line), with some exceptions, particularly on smaller boards. This indicates that the A3 agent performs better when the number of initial legal moves increases.

2.2 MCTS Agents Analysis

Performance Over Time Figure 4 showcases the fluctuation of the average win rates of the agents given different computation times. We observed the following:

- Against the Greedy Player, the MCTS agent shows an improvement in performance as the computation time extends, but in general, it does not win any matches.
- Competing with another MCTS, for time<0.5s the agents perform somewhat similarly. For time=1s, but with increased computation time decreasing the C hyper-parameter seems to have a positive effect on the performance.

Performance Across Board Configurations Figure 5 illustrates the agents' win rates over an average of all computation times across various board types. We observed the following:

[4]We chose to default to C=2 as per lecture slides

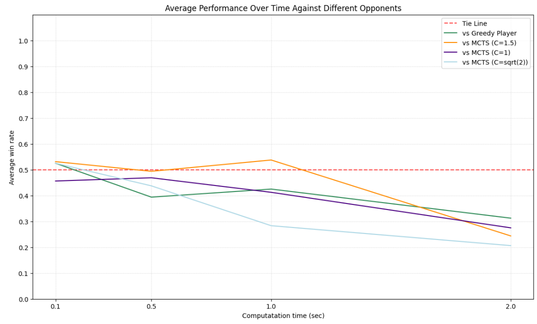

Figure 4: Average performance of MCTS (C=2) agent vs other agents (including varying C) over time

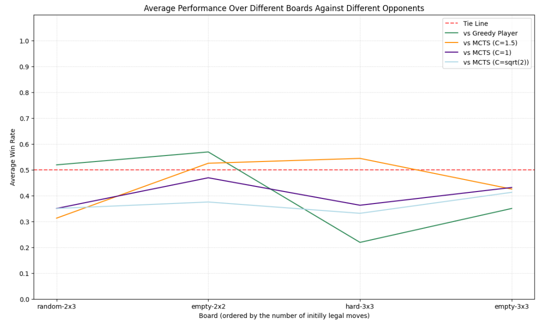

Figure 5: Average performance of MCTS (C=2) agent vs other agents (including varying C) over board configurations

2.3 Minimax Calls and Iterations of MCTS Analysis

Finally, an analysis was made of the difference in behaviour between Minimax and Monte Carlo Tree Search (MCTS), in particular examining the number of iterations as the time available to make a move varies. The number of calls of the Minimax algorithm and the number of MCTS iterations were measured by playing games on four boards (empty-2x2, random-2x3, hard-3x3, empty-3x3) with four different time limits (0.1s, 0.5s, 1s, 2s).

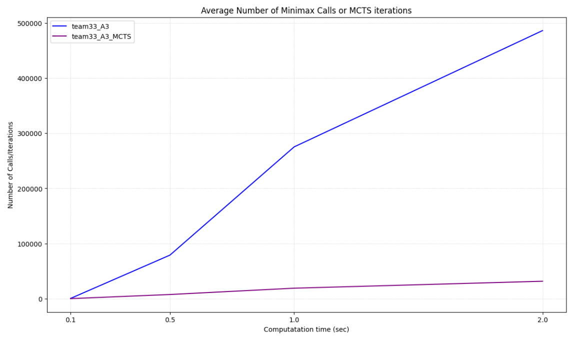

Figure 6: Average number of Minimax Calls and MCTS Iterations over time

The results in Figure 6 demonstrate that the number of calls of the Minimax function increases significantly as the time available for move selection increases, suggesting a tendency for the game tree to expand as the decision time extends. In contrast, the number of iterations of MCTS has a slight, almost constant slope, suggesting greater stability in carrying out iterations regardless of the time available. In other words, the analysis highlights a trade-off between deepening the decision tree and stability in the exploration of moves: Minimax adapts to the increase in available time by broadening its search, while MCTS maintains a more constant and risk-oriented strategy. Selective exploration, guided by Monte Carlo simulations.

3 Motivation & Reflection

In this section, we will reflect on several design decisions in the two agents and their strengths and weaknesses in the domain of Competitive Sudoku.

Incremental best moves improvements In Competitive Sudoku, the state space explodes with the board size. With a short turn time, agents often cannot explore all move combinations and must search a subset of the state space, a challenge compounded by unknown time limits. Our agents tackle this by progressively improving the best move within their time limit. The Minimax agent searches the game tree to a certain depth, incrementally increasing this depth as time permits. This strategy leverages Competitive Sudoku's scoring system, where completing regions earns intermediate scores, guiding the agent towards moves that offer short-term score gains. Conversely, the MCTS agent exploits the game's characteristic of having relatively few turns in a game. It simulates numerous complete games to identify beneficial moves, continuously refining estimates after each simulated game.

We optimised the generation of new states by locally updating the boards based on the move, in the sense that applying a move at some position will only affect the row, column and block that position is in. This is helpful, as evaluating more states or games yields better results. For this reason, we measured this and the results are shown in Figure 6.

Data saving Both the MCTS and Minimax agents save data between turns to improve efficiency. The game state differs from the last turn only by the last two moves, making it unnecessary to recompute everything. The Minimax agent stores the computationally expensive list of *unsolvable moves*. The MCTS agent, maintaining a tree of potential best moves, updates its strategy by following the path of the last two moves in the saved game tree. This data retention in MCTS is particularly beneficial when the tree depth exceeds two layers. Longer turn times are hypothesised to enhance the MCTS agent's performance. Additionally, the agent's hyperparameter C, controlling the balance between exploration and exploitation, suggests that lower values (favouring exploitation) might lead to deeper trees and thus better performance. This was tested and the results are shown in Figure 4 and 5.

Intentional turn loss In Competitive Sudoku, some moves that would the Sudoku unsolvable are rejected, causing the agent to lose their turn. The Minimax agent exploits this by intentionally playing an unsolvable move when advantageous. For example, if the Minimax agent predicts that all explored moves will benefit the opponent, it may choose to forfeit its turn. This forces the opponent into a position where they might lose points. Additionally, the agent can avoid losing its turn when this would be disadvantageous. This strategic use of unsolvable moves adds a layer of complexity to the agent's decision-making process.

Early vs late game The Minimax strategy is less effective in the early stage of the game when Equation 1 is often zero and thus not informative. This is partially mitigated by heuristically ordering moves. In contrast, it is quite effective in the late game when it can search most of the state space. MCTS makes better early game decisions by simulating complete games, but it is less effective at predicting short-term consequences, focusing less on immediate future nodes and under-penalising poor outcomes. Essentially, Minimax is nearsighted and MCTS farsighted. A prospective hybrid approach could combine MCTS's early-game strength with Minimax's late-game precision.

Symmetry Many solutions to a Sudoku are mirror images of others. As a result, many states result in the same score, and this property is stronger for states in nearly empty boards. Future work could consider an agent that is able to find this symmetry, allowing it to prune a large part of the search space.

References

Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975. ISSN 0004-3702. doi:https://doi.org/10.1016/0004-3702(75)90019-3. URL https://www.sciencedirect.com/science/article/pii/0004370275900193.

Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985. ISSN 0004-3702. doi:https://doi.org/10.1016/0004-3702(85)90084-0. URL https://www.sciencedirect.com/science/article/pii/0004370285900840.

Thomas Krak. Competitive sudoku: Rules, 2023. URL https://canvas.tue.nl/courses/25392/files/5120538?module_item_id=522484. Accessed: 2023-11-29.

E. Polak. *Basics of Minimax Algorithms*, pages 343–369. Springer US, Boston, MA, 1989. ISBN 978-1-4757-6019-4. doi:10.1007/978-1-4757-6019-4_20. URL https://doi.org/10.1007/978-1-4757-6019-4_20.

Maciej Swiechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mandziuk. Monte carlo tree search: A review of recent modifications and applications. *CoRR*, abs/2103.04931, 2021. URL https://arxiv.org/abs/2103.04931.

Python files

In this appendix, you must include all your Python files, so that the reviewers can evaluate your code. So this must at least be your `sudokuai.py` file, but might include any support files. Make sure that the file name (and path) is explicitly mentioned, so that it is clear how the code is used. Avoid too-long code lines that need to be broken over multiple lines here.

Code Listing 1: Heuristics Minimax agent with heuristics search from Assignment A2. `team33_A2/sudokuai.py`.

```python
"""Competitive Sudoku AI.

Adapted from /naive_player/sudokuai.py

A1: iterative deepening minimax search with alpha beta pruning.
A2: heuristic search.
"""

import os
from random import shuffle

# from numpy import full

# Import types and libraries
from competitive_sudoku.sudoku import GameState, Move, SudokuBoard
from competitive_sudoku.sudokuai import SudokuAI

from .utils import block_range # Game specific logic
from .utils import block_index, calculate_move_score, is_illegal, next_player


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

    def __init__(self):
        super().__init__()
        self.transposition_table = []

    def update_legal(self, game_state: GameState, move: Move, moves: dict) -> list:
        """Update which moves are legal in the recursive minimax search.

        :param move: The move to be evaluated
        :param moves: A dictionary containing the inital set of moves, whether they are still legal and other properties of the moves.
            initial: list of initally legal moves, used as subset to avoid iterating over all moves
            legal: numpy array of shape (board_size, board_size, board_size + 1) where legal[i, j, k] is True if Move(i, j, k) is legal
            count: Counter for legal moves, avoid repeated iteration
```

```python
        :return: A list of moves that were invalidated by the move.
        """
        _moves_invalidated = []
        for row in range(game_state.board.board_height()):
            # move invalidates another move if not already illegal, avoid double counting
            if moves["legal"][row, move.j, move.value]:
                _moves_invalidated.append(Move(row, move.j, move.value))
                moves["legal"][row, move.j, move.value] = False  # Set legal status
        for column in range(game_state.board.board_width()):
            if moves["legal"][move.i, column, move.value]:
                _moves_invalidated.append(Move(move.i, column, move.value))
                moves["legal"][move.i, column, move.value] = False
        for row, col in block_range(row=move.i, col=move.j, board=game_state.board):
            if moves["legal"][row, col, move.value]:
                _moves_invalidated.append(Move(row, col, move.value))
                moves["legal"][row, col, move.value] = False
        moves["count"] -= len(_moves_invalidated)
        return _moves_invalidated

    def minimax(
            self,
            game_state: GameState,
            move: Move,
            moves: dict,
            current_player: int,
            maximizing_player: int,
            depth: int,
            *,
            alpha: float = float("-inf"),
            beta: float = float("inf"),
    ) -> float:
        """Returns the score of a given move.

        Minimax search that considers the perspectives of two players.
        Searches until some depth before returning the score.
        Uses alpha beta pruning to avoid searching branches which cannot lead to better results.

        :param game_state: The current game state. Describes the board, scores, taboo moves and move history.
        :param move: The move to be evaluated
        :param moves: A dictionary containing the inital set of moves, whether they are still legal and other properties of the moves.
            initial: list of initally legal moves, used as subset to avoid iterating over all moves
            legal: numpy array of shape (board_size, board_size, board_size + 1) where legal[i, j, k] is True if Move(i, j, k) is legal
            count: Counter for the number of legal moves.
```

```
            free: shows per region (row, col or block) what the number of free squares is, used in heuristics.
        :param current_player: The player who's turn it is. Will be the same throughout the turn. (0 or 1 for first or second player)
        :param maximizing_player: The player who's score is to be maximised. (0 or 1 for first or second player)
        :param depth: The maximum depth to search before returning the score.
        :param alpha: The highest so far value for alpha beta pruning. (initially -inf)
        :param beta: The lowest so far value for alpha beta pruning. (initially inf)
        :return: The score of the move (higher is better for maximizing player, lower is better for minimizing player)
        """
        block_indices = block_index(move.i, move.j, game_state.board)

        # Apply move, update resulting scores
        # Update legal moves and count newly invalidated moves
        game_state.board.put(move.i, move.j, move.value)
        _score_achieved = calculate_move_score(game_state, move)
        game_state.scores[current_player] += _score_achieved
        _moves_invalidated = self.update_legal(game_state, move, moves)
        # switches perspective to other player
        current_player = next_player(current_player)

        # Update properties used in heuristics
        moves["free"]["row"][move.i] -= 1
        moves["free"]["col"][move.j] -= 1
        moves["free"]["block"][block_indices[0]][block_indices[1]] -= 1

        # Search until game is finished or maximum depth is reached
        if depth == 0 or moves["count"] == 0:
            # evaluate the current board
            best_value = self.evaluate_state(
                maximizing_player, current_player, game_state, moves["free"]
            )
        else:
            if maximizing_player == current_player:  # maximising player
                best_value = float("-inf")
                for try_move in moves["initial"]:
                    if not moves["legal"][try_move.i, try_move.j, try_move.value]:
                        continue
                    # Recurse and find value up to some depth
                    value = self.minimax(
                        game_state,
                        try_move,
                        moves,
                        current_player,
                        maximizing_player,
                        depth - 1,
                        alpha=alpha,
```

```
                        beta=beta,
                        )
                    best_value = max(best_value, value)
                    alpha = max(alpha, best_value)
                    if beta < alpha:
                        break
            else:  # minimising player
                best_value = float("inf")
                for try_move in moves["initial"]:
                    if not moves["legal"][try_move.i, try_move.j, try_move.value]:
                        continue
                    # Recurse and find value up to some depth
                    value = self.minimax(
                        game_state,
                        try_move,
                        moves,
                        current_player,
                        maximizing_player,
                        depth - 1,
                        alpha=alpha,
                        beta=beta,
                        )
                    best_value = min(best_value, value)
                    beta = min(beta, best_value)
                    if beta < alpha:
                        break

        # Undo move and its effects
        current_player = next_player(current_player)
        moves["free"]["row"][move.i] += 1
        moves["free"]["col"][move.j] += 1
        moves["free"]["block"][block_indices[0]][block_indices[1]] += 1
        moves["count"] += len(_moves_invalidated)
        for inv_move in _moves_invalidated:
            moves["legal"][inv_move.i, inv_move.j, inv_move.value] = True
        game_state.scores[current_player] -= _score_achieved
        game_state.board.put(move.i, move.j, 0)

        # Recursion result
        return best_value

    def evaluate_state(
            self,
            maximizing_player: int,
            current_player: int,
            game_state: GameState,
            free: dict,
    ) -> float:
        """Heuristic evaluation of the current game state.

        Is used by minimax to evaluate the current game state which is most often not
```

```python
        a complete game.
        Base score is the difference between the scores of the two players since maximizing this will result in a win.
        Additional heuristic contributions are made for early game where score is often 0.

        :param maximizing_player: The player who's score is to be maximised. (0 or 1 for first or second player)
        :param game_state: The current game state. Describes the board, scores, taboo moves and move history.
        :param free: The number of free squares per region.
        :return: The score of the game state (higher is better for maximizing player, and vice versa)
        """
        score = (
            game_state.scores[maximizing_player]
            - game_state.scores[next_player(maximizing_player)]
        )

        if not os.environ.get("not_prefer_more_empty"):
            # Scale to avoid this additional heuristic dominating the score
            # Will result in an early game strategy avoiding a filled field.
            # Positive contribution, our player will thus prefer less filled fields.
            if current_player == maximizing_player:
                score += 0.1 * self.prefer_empty_regions(game_state, free)

        return score

    def prefer_empty_regions(self, game_state: GameState, free: dict):
        """Heuristic for evaluating a state.

        Prefer less filled out regions by counting the number of free moves
        Normalising using the the maximum number of squares in a region

        :param maximizing_player: The player who's score is to be maximised. (0 or 1 for first or second player)
        :param game_state: The current game state. Describes the board, scores, taboo moves and move history.
        :param free: The number of free squares per region.
        :return: The heuristic score, will be 1 for a completely empty field.
        """
        board_size = game_state.board.board_height()
        num_blocks = board_size / game_state.board.region_height()
        score = sum([count / board_size for count in free["row"]]) / board_size / 3
        score += sum([count / board_size for count in free["col"]]) / board_size / 3
        score += (
            sum(
                sum([count / board_size for count in block]) / num_blocks
                for block in free["block"]
            )
            / num_blocks
```

```
                / 3
        )
        return score

    def find_initial_moves(self, game_state: GameState) -> list[Move]:
        """
        Find all possible moves for a given state. This is copy of method used in A1
                                                    used for benchmarking purposes.
        @param game_state: GameState
        @return: list of moves
        """
        board_size = game_state.board.board_width()

        # Generate possible moves
        initial_moves = [
            Move(i, j, value)
            for i in range(board_size)
            for j in range(board_size)
            for value in range(1, board_size + 1)
            if not is_illegal(move=Move(i, j, value), state=game_state)
        ]

        # Shuffle moves to be less predictable
        shuffle(initial_moves)

        return initial_moves

    def find_initial_moves_heuristics(self, state: GameState) -> list[Move]:
        """
        Find all possible moves for a given state and order them by priority. The
                                                    priority is determined by the
        number of possible moves for a cell. Cells with fewer possible moves are
                                                    prioritised because they are
        more likely to result in a completed row, column, or block.
        Any cell with 2 possible moves is *de*-prioritised, because that means the
                                                    opponent could complete a row, column
        or block next turn.
        @param state: GameState
        @return: ordered list of moves
        """
        size = state.board.board_width()

        # Store moves in a dictionary with the number of possible moves for that cell
                                                    as key
        priority_dict = dict([(key, []) for key in range(0, size + 1)])

        for i in range(size):
            for j in range(size):
                possible_moves_for_cell = []
                for value in range(1, size + 1):
                    move_candidate = Move(i, j, value)
```

```python
                    if not is_illegal(move=move_candidate, state=state):
                        possible_moves_for_cell.append(move_candidate)

                priority_dict[len(possible_moves_for_cell)].extend(possible_moves_for_cell)

        key_order = sorted(priority_dict.keys())

        # Prioritise cells that have 2 possible moves,
        # because the opponent could complete a row, column or block
        if len(key_order) > 2:
            key_order.pop(2)
            key_order.append(2)

        # Return list of moves in order of search priority
        return [move for key in key_order for move in priority_dict[key]]

    def compute_best_move(self, game_state: GameState) -> None:
        """Computes the best move for the agent and proposes it.

        Will initially propose a random move, then evaluate different moves based on minimax search.
        Since the turn time is not known it will propose the best move found so far by iteratively deepening the search.

        :param game_state: The current game state. Describes the board, scores and move history.
        """
        board_size = game_state.board.board_height()
        num_blocks = board_size // game_state.board.region_height()

        # print(f"Avoid 2 moves: {not os.environ.get('not_avoid_2_moves')}")
        # print(f"Prefer empty regions: {not os.environ.get('not_prefer_more_empty')}")

        # Generate possible moves
        initial_moves = (
            self.find_initial_moves_heuristics(game_state)
            if not os.environ.get("not_avoid_2_moves")
            else self.find_initial_moves(game_state)
        )

        # Move cache
        moves = {}

        # List of initially legal moves, used as subset to avoid iterating over all moves
        moves["initial"] = initial_moves

        # Propose a random move (initial, avoid timeout)
        self.propose_move(moves["initial"][0])
```

```python
        # Avoid repeated regeneration of legal moves by tracking their status
        from numpy import full
        moves["legal"] = full(
            shape=(board_size, board_size, board_size + 1),
            dtype=bool,
            fill_value=False,
        )
        moves["count"] = len(moves["initial"])
        for move in moves["initial"]:
            moves["legal"][move.i, move.j, move.value] = True

        # Track some properties of the game to be used in statistical search
        # Count how many free squares there are per region
        # TODO do this above when generating moves?
        moves["free"] = {
            "row": [
                len(set(((m.i, m.j) for m in moves["initial"] if m.i == y)))
                for y in range(board_size)
            ],
            "col": [
                len(set(((m.i, m.j) for m in moves["initial"] if m.j == x)))
                for x in range(board_size)
            ],
            "block": [
                [
                    len(
                        set(
                            (
                                (m.i, m.j)
                                for m in moves["initial"]
                                if block_index(m.i, m.j, game_state.board) == (i, j)
                            )
                        )
                    )
                    for j in range(num_blocks)
                ]
                for i in range(num_blocks)
            ],
        }

        # Iteratively increase the search depth of minimax
        best_move: tuple[Move, float] | None = None
        for depth_limit in range(1, len(moves["initial"]), 1):
            # evaluate different moves based on minimax
            for move in moves["initial"]:
                # player_index is 0 or 1 (first or second player)
                player_index = game_state.current_player() - 1

                value = self.minimax(
                    game_state,
```

```
                    move,
                    moves,
                    player_index,  # Current player is self
                    player_index,  # Maximising own score
                    depth_limit,
                )

                if best_move is None or value > best_move[1]:
                    best_move = (move, value)
                    # Update proposed move (best so far, avoid timeout while find a better move)
                    self.propose_move(best_move[0])
                    # print(move.i, move.j, value)
```

Code Listing 2: Improved Minimax agent from Assignment A3. `team33_A3/sudokuai.py`.

```
"""
Competitive Sudoku AI.

Adapted from /naive_player/sudokuai.py

A1: iterative deepening minimax search with alpha beta pruning.
A2: heuristic search.
A3: heuristic search with unsolvable moves list.

"""

from random import shuffle

# Import types and libraries
from competitive_sudoku.sudoku import GameState, Move, SudokuBoard
from competitive_sudoku.sudokuai import SudokuAI

from .utils import block_range, StateMatrixT, SavedDataT, is_unsolvable  # Game specific logic
from .utils import (
    block_index,
    calculate_filling_rate,
    calculate_move_score,
    is_illegal,
    next_player,
)


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

    def minimax(
        self,
        *,
```

```python
        game_state: GameState,
        move: Move,
        state_matrix: StateMatrixT,
        current_player: int,
        maximizing_player: int,
        depth: int,
        alpha: float = float("-inf"),
        beta: float = float("inf"),
    ) -> float:
        """
        Returns the score of a given move.

        Minimax search that considers the perspectives of two players.
        Searches until some depth before returning the score.
        Uses alpha beta pruning to avoid searching branches which cannot lead to better results.

        :param game_state: The current game state. Describes the board, scores, unsolvable moves and move history.
        :param move: The move to be evaluated
        :param state_matrix: State matrix containing the initial set of moves and whether they are still legal
        :param current_player: The player who's turn it is. Will be the same throughout the turn. (0 or 1 for first or second player)
        :param maximizing_player: The player who's score is to be maximised. (0 or 1 for first or second player)
        :param depth: The maximum depth to search before returning the score.
        :param alpha: The highest so far value for alpha beta pruning. (initially -inf)
        :param beta: The lowest so far value for alpha beta pruning. (initially inf)
        :return: The score of the move (higher is better for maximizing player, lower is better for minimizing player)
        """

        # Apply move, update resulting scores
        # Update legal moves and count newly invalidated moves
        game_state.board.put(move.i, move.j, move.value)
        _score_achieved = calculate_move_score(game_state, move)
        game_state.scores[current_player] += _score_achieved
        _moves_invalidated = self.update_legal(game_state, move, state_matrix)

        # Switches perspective to other player
        current_player = next_player(current_player)

        # Search until game is finished or maximum depth is reached
        if depth == 0 or state_matrix["legal_count"] == 0:
            # evaluate the current board
            best_value = self.evaluate_state(
                maximizing_player,
                game_state,
```

```
                )
        else:
            if maximizing_player == current_player:  # maximising player
                best_value = float("-inf")
                for try_move in state_matrix[" initial "]:
                    if not state_matrix[" legal "][try_move.i][try_move.j][try_move.value]:
                        continue
                    # Recurse and find value up to some depth
                    value = self.minimax(
                        game_state=game_state,
                        move=try_move,
                        state_matrix=state_matrix,
                        current_player=current_player,
                        maximizing_player=maximizing_player,
                        depth=depth - 1,
                        alpha=alpha,
                        beta=beta,
                    )
                    best_value = max(best_value, value)
                    alpha = max(alpha, best_value)
                    if beta < alpha:
                        break
            else:  # minimising player
                best_value = float(" inf ")
                for try_move in state_matrix[" initial "]:
                    if not state_matrix[" legal "][try_move.i][try_move.j][try_move.value]:
                        continue
                    # Recurse and find value up to some depth
                    value = self.minimax(
                        game_state=game_state,
                        move=try_move,
                        state_matrix=state_matrix,
                        current_player=current_player,
                        maximizing_player=maximizing_player,
                        depth=depth - 1,
                        alpha=alpha,
                        beta=beta,
                    )
                    best_value = min(best_value, value)
                    beta = min(beta, best_value)
                    if beta < alpha:
                        break
        # Undo move and its effects
        current_player = next_player(current_player)
        state_matrix["legal_count"] += len(_moves_invalidated)
        for inv_move in _moves_invalidated:
            state_matrix[" legal "][inv_move.i][inv_move.j][inv_move.value] = True
        game_state.scores[current_player] -= _score_achieved
        game_state.board.put(move.i, move.j, 0)
```

```
        # Recursion result
        return best_value

    def evaluate_state(self, maximizing_player: int, game_state: GameState) -> float:
        """
        Calculates the score of a given game state.

        :param maximizing_player: The player whose score is to be maximised. (0 or 1 for first or second player)
        :param game_state: The current game state.
        :return: The score of the game state. (higher is better for maximizing player, and vice versa)
        """
        return game_state.scores[maximizing_player] - game_state.scores[next_player(maximizing_player)]

    def update_legal(self, game_state: GameState, move: Move, moves: StateMatrixT) -> list:
        """
        Update which moves are legal in the recursive minimax search.

        :param game_state: The current game state. Describes the board, scores, taboo moves and move history.
        :param move: The move to be evaluated
        :param moves: State matrix containing the initial set of moves and whether they are still legal
        :return: A list of moves that were invalidated by the move.
        """

        _moves_invalidated = []
        for row in range(game_state.board.board_height()):
            # move invalidates another move if not already illegal, avoid double counting
            if moves["legal"][row][move.j][move.value]:
                _moves_invalidated.append(Move(row, move.j, move.value))
                moves["legal"][row][move.j][move.value] = False  # Set legal status

        for column in range(game_state.board.board_width()):
            if moves["legal"][move.i][column][move.value]:
                _moves_invalidated.append(Move(move.i, column, move.value))
                moves["legal"][move.i][column][move.value] = False

        for row, col in block_range(row=move.i, col=move.j, board=game_state.board):
            if moves["legal"][row][col][move.value]:
                _moves_invalidated.append(Move(row, col, move.value))
                moves["legal"][row][col][move.value] = False
        moves["legal_count"] -= len(_moves_invalidated)
        return _moves_invalidated

    def find_initial_moves(self, game_state: GameState) -> list[Move]:
```

```
        """
        Find all possible moves for a given state.

        :param game_state: The current game state.
        :return: The list of legal initial moves.
        """
        board_size = game_state.board.board_width()

        # Generate possible moves
        initial_moves = [
            Move(i, j, value)
            for i in range(board_size)
            for j in range(board_size)
            for value in range(1, board_size + 1)
            if not is_illegal(move=Move(i, j, value), state=game_state)
        ]

        # Shuffle moves to be less predictable
        shuffle(initial_moves)

        return initial_moves

    def calculate_free_cells(self, game_state: GameState) -> dict:
        """
        Calculate how many free cells there are in each region of the sudoku (row, column, block), given a game state.

        It is used to assign each move a priority level.

        :return: A dictionary containing the number of free cells, for each region.
            row: list containing a value for each row of the sudoku, corresponding to the number of free cells in the row
            col: list containing a value for each column of the sudoku, corresponding to the number of free cells in the column
            block: bidimensional list containing a value for each block of the sudoku, corresponding to the number of free cells in the block
        """
        board_size = game_state.board.board_height()

        free_cells = {
            "row": [],
            "col": [],
            "block": [],
        }
        # Calculate free cells in rows
        for y in range(board_size):
            empty_count = sum(
                1
                for x in range(board_size)
                if game_state.board.get(y, x) == SudokuBoard.empty
```

```
            )

            free_cells["row"].append(empty_count)

        # Calculate free cells in columns
        for x in range(board_size):
            empty_count = sum(
                1
                for y in range(board_size)
                if game_state.board.get(y, x) == SudokuBoard.empty
            )

            free_cells["col"].append(empty_count)

        # Calculate free cells in blocks
        for block_i in range(board_size // game_state.board.region_height()):
            empty_list = []
            for block_j in range(board_size // game_state.board.region_width()):
                empty_count = 0
                for row, col in block_range(
                    row=block_i * game_state.board.region_height(),
                    col=block_j * game_state.board.region_width(),
                    board=game_state.board,
                ):
                    if game_state.board.get(row, col) == SudokuBoard.empty:
                        empty_count += 1
                empty_list.append(empty_count)
            free_cells["block"].append(empty_list)

        return free_cells

    def order_moves(
        self, board: SudokuBoard, moves_list: list[Move], free_cells: dict
    ) -> tuple[list[Move], list[Move]]:
        """
        Sort a list of moves by their priority. Priority is based on the number of regions the move allows to complete.
        Moves that complete three regions have top priority and are therefore placed at the top of the list,
        followed by those that complete two regions and those that complete one.
        Moves that do not complete any region have no priority, and are returned in a separate list.

        :param board: The current board.
        :param moves_list: List of moves that have to be sorted.
        :param free_cells: dictionary containing the number of free cells for each region. It's used to assign priority to moves.

        :return: Two lists, one with the moves with priority, sorted, the other with the moves without priority.
```

```python
        """

        # Assign priority to initial moves based on how many regions each move completes
        priority = {
            0: [],  # None priority
            1: [],
            2: [],
            3: [],
        }

        for move in moves_list:
            row = move.i
            col = move.j
            block_i, block_j = block_index(row, col, board)

            # Count how many regions the move completes
            # sum counts the number of True values in the list
            regions_to_be_completed = sum([
                free_cells["row"][row] == 1,
                free_cells["col"][col] == 1,
                free_cells["block"][block_i][block_j] == 1
            ])

            priority[regions_to_be_completed].append(move)

        return priority[3] + priority[2] + priority[1], priority[0]

    def initialize_stored_data(self, lower_limit: float, upper_limit: float) -> SavedDataT:
        """
        Initialize stored data.
        :param lower_limit: lower limit of the range of filling rate
                            values in which the list of moves that make Sudoku unsolvable is calculated.
        :param upper_limit: upper limit of the range of filling rate
        :return: initial values for the dictionary containing the stored data.
        """
        return {
            "unsolvable_moves": [],
            "check_unsolvable_range": (
                lower_limit,
                upper_limit,
            ),  # Check if a move makes the sudoku unsolvable only after
                # the board is 20% complete and before is 85% complete
            "unsolvable_list_building_started": False,
            "unsolvable_list_building_finished": False,
        }

    def calculate_unsolvable_moves(
        self, game_state: GameState, moves: list[Move], data: SavedDataT
```

```python
    ) -> None:
        """
        Calculate the list of unsolvable moves, simulating each move in the list moves and checking
        if the new gamestate represents a Sudoku impossible to solve.

        :moves: list of possible unsolvable moves.
        :data: SavedDataT dictionary containing the list of unsolvable moves.
        """
        for move in moves:
            game_state.board.put(move.i, move.j, move.value)
            if is_unsolvable(game_state.board):
                data["unsolvable_moves"].append(move)
            game_state.board.put(move.i, move.j, SudokuBoard.empty)

    def update_unsolvable_moves(self, data: dict, moves: StateMatrixT) -> None:
        """
        Update the list of unsolvable moves,
        checking if they are still present in the list of legal moves
        calculated at the beginning of each turn.

        :data: dictionary containing the list of unsolvable moves
        :moves: list of legal moves
        """
        updated_moves = []
        for move in data["unsolvable_moves"]:
            if move in moves["initial"]:
                updated_moves.append(move)
                moves["initial"].remove(move)

        data["unsolvable_moves"] = updated_moves

    def compute_best_move(self, game_state: GameState) -> None:
        """
        Computes the best move for the agent and proposes it.

        It initially proposes a random move, then sorts the moves based on their priority and proposes the one with the highest priority.
        It then searches for the best move with minimax search.

        Since the turn time is not known it proposes the best move found so far by iteratively deepening the search, starting with depth 2.
        There is no need to search with depth 1, as moves are already sorted by priority.

        If the best move found so far leads to a negative result (calculated by the evaluation function),
        it proposes a move that makes the sudoku impossible to solve, in order to pass the turn without entering new values.
```

```
        The list of moves that make Sudoku impossible is calculated when the board is 20% full,
        and is saved by the game engine so that it is also available in the next rounds.
        At each turn the list must be updated, to ensure that all the moves on it are still legal.
        If the time limit of the turn does not allow the list of moves that invalidate the sudoku to be calculated,
        the agent tries to calculate it again when the board is 30% full.
        If even then the time available is not enough, it waits for the board to be 40 % full, and so on.
        When the list is empty (all moves have been used or are no longer legal), it is recalculated

        :param game_state: The current game state. Describes the board, scores and move history.
        """
        board_size = game_state.board.board_height()

        # Generate possible moves
        initial_moves = self.find_initial_moves(game_state)

        # Propose a random move (initial, avoid timeout)
        self.propose_move(initial_moves[0])

        # Order initial moves by priority
        free_cells = self.calculate_free_cells(game_state)
        priority_moves, non_priority_moves = self.order_moves(game_state.board, initial_moves, free_cells)

        # Initial moves are sorted by priority | set(initial_moves) == set(priority_moves + non_priority_moves)
        initial_moves = priority_moves + non_priority_moves

        # Propose move with the highest priority (initial, avoid timeout)
        self.propose_move(initial_moves[0])

        # Initialize legal moves matrix with all moves being illegal
        legal_moves = [
            [[False for v in range(board_size + 1)] for col in range(board_size)]
            for row in range(board_size)
        ]

        # Mark initial moves as legal in the matrix
        for move in initial_moves:
            legal_moves[move.i][move.j][move.value] = True

        # Initialize state matrix
        state_matrix: StateMatrixT = {
            "initial": initial_moves,
```

```python
            "legal_count": len(initial_moves),
            "legal": legal_moves,
        }

        data: SavedDataT = self.load()
        if data is None:  # if no data is saved, initialize it with default values
            data = self.initialize_stored_data(0.2, 0.85)
            self.save(data)

        # If the previous attempt to build unsolvable moves list failed,
        # then increase the lower limit of the filling range by 10% (this number was chosen arbitrarily)
        if (
            data["unsolvable_list_building_started"] is True
            and data["unsolvable_list_building_finished"] is False
        ):
            data["check_unsolvable_range"] = (
                data["check_unsolvable_range"][0] + 0.1,
                data["check_unsolvable_range"][1],
            )
            data["unsolvable_list_building_started"] = False

            self.save(data)

        filling_rate = calculate_filling_rate(free_cells)
        # First turn or unsolvable move list is empty (all unsolvable moves have been used or are not legal anymore)
        if (
            len(data["unsolvable_moves"]) == 0
            and filling_rate > data["check_unsolvable_range"][0]
            and filling_rate < data["check_unsolvable_range"][1]
        ):
            data["unsolvable_list_building_started"] = True
            self.save(data)

            # Only moves that not complete any regions can invalidate the game
            self.calculate_unsolvable_moves(game_state, non_priority_moves, data)

            data["unsolvable_list_building_finished"] = True
            self.save(data)

        # Check which unsolvable moves can still be used and remove them from moves
        if len(data["unsolvable_moves"]) > 0:
            self.update_unsolvable_moves(data, state_matrix)
            self.save(data)

        # Iteratively increase the search depth of minimax
        best_move: tuple[Move, float] | None = None
        for depth_limit in range(2, len(state_matrix["initial"]), 1):
            # Evaluate different moves based on minimax
            for move in state_matrix["initial"]:
```

```
                # Player_index is 0 or 1 (first or second player)
                player_index = game_state.current_player() - 1

                value = self.minimax(
                    game_state=game_state,
                    move=move,
                    state_matrix=state_matrix,
                    current_player=player_index,  # Current player is self
                    maximizing_player=player_index,  # Maximising own score
                    depth=depth_limit,
                )

                if best_move is None or value > best_move[1]:
                    best_move = (move, value)
                    # If every explored move leads to a negative score, propose a
                        unsolvable move
                    if best_move[1] < 0 and len(data["unsolvable_moves"]) > 0:
                        self.propose_move(data["unsolvable_moves"][0])
                    else:
                        self.propose_move(best_move[0])
```

Code Listing 3: MCTS agent, the second agent from Assignment A3. `team33_A3_MCTS/sudokuai.py`.

```
"""Competitive Sudoku AI.

Adapted from /naive_player/sudokuai.py

A1: iterative deepening minimax search with alpha beta pruning.
A2: heuristic search.
A3: different improvements and a second strategy.
"""

from typing import Literal, TypedDict, TypeVar, Type, Union

from competitive_sudoku.sudoku import GameState, Move, SudokuBoard
from competitive_sudoku.sudokuai import SudokuAI
from random import shuffle, choice
from copy import deepcopy
from math import log

DEBUG = False

if DEBUG:
    from graphviz import Digraph
    import uuid


# Game specific logic not related to strategy stored in utils.py
from .utils import (
    block_range,
    calculate_move_score,
```

```python
    is_illegal,
    next_player,
    PlayerID,
)

# Hyperparameters
MCTS_EXPLORATION = 2


def build_graph(
    node: "MCGTNode", max_player: PlayerID, graph=None, node_name=None, depth_limit=None
):
    """ Visualise the MCTS game tree using graphviz"""
    if node_name is None:
        node_name = uuid.uuid4().hex

    if graph is None:
        graph = Digraph()
        graph.node(name=node_name, label=str(node).replace(";", "\n"))

    for child in node.children:
        child_name = uuid.uuid4().hex

        graph.node(
            name=child_name,
            label=str(child).replace(";", "\n"),
            color="green" if child.player == max_player else "black",
        )
        graph.edge(tail_name=node_name, head_name=child_name)

        if depth_limit is None or child.depth < depth_limit:
            build_graph(child, max_player, graph, child_name, depth_limit)

    return graph


class StateMatrixT(TypedDict):
    """Type of state matrix used in MCTS"""

    initial: list[Move]
    legal: list[list[list[bool]]]
    legal_moves_count: int


class MCGTNode:
    """Monte Carlo Game Tree Node"""

    def __init__(
        self,
        parent: Union["MCGTNode", None],
```

```python
        move: Move,
        depth: int,
        player: PlayerID,
        visited=0,
        score=0,
    ):
        """MC Game tree node
        :param parent: parent node, allows backtracking
        :param move: move that was made to reach this node
        :param depth: depth of this node in the tree
        :param visited: number of times this node has been visited
        :param score: score of the game at this node
        """
        self.move = move
        self.visited = visited
        self.score = score
        self.parent = parent
        self.children: list[MCGTNode] = []  # Children are added in expansion phase
        self.depth = depth
        self.player: PlayerID = player

    @property
    def is_leaf(self):
        return not self.children

    @property
    def average_score(self):
        return self.score / self.visited if self.visited > 0 else 0

    @property
    def ucb(self):
        """Upper Confidence Bound of this node"""
        if self.visited == 0:
            return float("inf")

        return self.average_score + MCTS_EXPLORATION * (
            (log(self.parent.visited) / self.visited) ** 0.5
        )

    def __str__(self):
        return f"{str(self.move)}; n = {self.visited}; q = {self.score}"


class MCGameTree:
    def __init__(
        self,
        root: MCGTNode,
        game_state: GameState,
        state_matrix: StateMatrixT,
        max_player: PlayerID,
    ):
```

```python
        """Monte Carlo Game Tree

        :param root: root node of the tree
        :param game_state: current game state
        :param state_matrix: state matrix used to track legal moves
        :param max_player: player to maximize score for
        """
        self.root = root
        self.maximizing_player = max_player
        self.game_state = game_state
        self.state_matrix = state_matrix

    def selection(self, node: MCGTNode) -> MCGTNode:
        """Select a leaf node to expand.

        Recursively select the child with the highest UCB until a leaf node is reached.

        :param node: node to start selection from.
        :return: leaf node.
        """
        if node.is_leaf:
            return node

        # Remove illegal moves (can occur because of reloading the tree)
        node.children = [
            child for child in node.children
            if self.state_matrix["legal"][child.move.i][child.move.j][child.move.value]
        ]

        # Select child with highest UCB
        return self.selection(max(node.children, key=lambda child: child.ucb))

    def expansion(self, node: MCGTNode) -> MCGTNode:
        """Expand a leaf node.

        A node is expanded when it is selected twice.
        Expansion adds all legal moves in the current game state as children of the node.

        :param node: node to expand.
        :return: node to simulate from.
        """
        # Apply all moves from node to the root (excluding)
        self.apply_move_on_node(node)

        # If v is a terminal state of the game, move to Backpropagation phase
        if self.state_matrix["legal_moves_count"] == 0:
            return node
```

```python
        # If n(v) = 0, move to Simulation phase with node v
        if node.visited == 0:
            return node

        # If n(v) > 0, add new states reached from legal moves in v
        # TODO use array of legal moves instead of matrix and inital moves?
        for move in self.state_matrix["initial"]:
            if self.state_matrix["legal"][move.i][move.j][move.value]:
                node.children.append(
                    MCGTNode(node, move, node.depth + 1, next_player(node.player))
                )

        return choice(node.children)

    def simulation(self, node: MCGTNode) -> int:
        """Simulate a game from a node.

        Simulate a game from a node by applying random moves until the game ends.

        :param node: node to simulate from.
        :return: score of the game. 1 when the maximizing player wins, 0 otherwise.
        """

        self.apply_move_on_node(node)
        player = next_player(node.player)
        while self.state_matrix["legal_moves_count"] > 0:  # until game ends
            # Select a random move
            # TODO use array of legal moves instead of matrix and inital moves?
            move = None
            while move is None:
                move = choice(self.state_matrix["initial"])
                if not self.state_matrix["legal"][move.i][move.j][move.value]:
                    move = None
            # Apply move
            self.apply_move(player, move)
            player = next_player(player)

        return (
            1
            if self.game_state.scores[self.maximizing_player]
            > self.game_state.scores[next_player(self.maximizing_player)]
            else 0
        )

    def backpropagation(self, node: MCGTNode | None, score: int) -> None:
        """Backpropagation the score from a node to the root

        :param node: node to start backpropagation from
        :param score: score to backpropagation
        """
        if node is None:
```

```
            return

        node.visited += 1
        node.score += score if node.player == self.maximizing_player else -score
        self.backpropagation(node.parent, score)

    def find_best_child(self) -> Move:
        """Find the best child of the root node

        Is based on the best average score of the children of the root node.

        :return: best move found so far
        """
        best_child = None
        best_avg_score = float("-inf")
        for child in self.root.children:
            if child.average_score > best_avg_score:
                best_avg_score = child.average_score
                best_child = child

        if best_child is None: # Happens in the first iteration
            return None

        return best_child.move

    def iterate(self) -> Move:
        """Perform one iteration of MCTS.

        Runs all phases of MCTS once.
        Selection, expansion, simulation and backpropagation.

        :return: best move found so far as defined in `find_best_child`.
        """
        # TODO optimise
        selected_leaf = self.selection(self.root)
        selected_leaf = self.expansion(selected_leaf)

        score = self.simulation(selected_leaf)
        self.backpropagation(selected_leaf, score)

        return self.find_best_child()

    def apply_move_on_node(self, node: MCGTNode) -> None:
        """Recursively apply moves to the game state"""
        # If node is root, no moves to apply (root is a dummy node)
        if node is self.root:
            return

        # If node parent is not a root, apply parent's moves first
        if node.parent is not None:
            self.apply_move_on_node(node.parent)
```

```python
        self.apply_move(node.player, node.move)

    def apply_move(self, player: PlayerID, move: Move):
        self.game_state.board.put(move.i, move.j, move.value)
        self._apply_move_update_state_matrix(move)
        self.game_state.scores[player] += calculate_move_score(self.game_state, move)

    def _apply_move_update_state_matrix(self, move: Move):
        """Update the state matrix when a move is made"""
        # Update which moves are legal
        changes = 0
        for idx in range(self.game_state.board.N):
            # move invalidates another move if not already illegal, avoids double counting
            if self.state_matrix["legal"][idx][move.j][move.value]:
                self.state_matrix["legal"][idx][move.j][move.value] = False
                changes += 1

            if self.state_matrix["legal"][move.i][idx][move.value]:
                self.state_matrix["legal"][move.i][idx][move.value] = False
                changes += 1

        for row, col in block_range(
            row=move.i, col=move.j, board=self.game_state.board
        ):
            if self.state_matrix["legal"][row][col][move.value]:
                self.state_matrix["legal"][row][col][move.value] = False
                changes += 1

        self.state_matrix["legal_moves_count"] -= changes


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

    def compute_best_move(self, game_state: GameState) -> None:
        """Computes the best move for the agent and proposes it.

        Will initially propose a random move, then evaluate different moves based on minimax search.
        Since the turn time is not known it will propose the best move found so far by iteratively deepening the search.

        :param game_state: The current game state. Describes the board, scores and move history.
        """
        board_size = game_state.board.board_height()
        num_blocks = board_size // game_state.board.region_height()
```

```
        # TODO also save game state matrix?
        # player_index is 0 or 1 ( first or second player )
        player_index = game_state.current_player() - 1

        # List of initially legal moves, used as subset to avoid iterating over all
                                    moves
        initial_moves = [
            Move(i, j, value)
            for i in range(board_size)
            for j in range(board_size)
            for value in range(1, board_size + 1)
            if not is_illegal (move=Move(i, j, value), state=game_state)
        ]
        # Propose a random move (initial, avoid timeout)
        self.propose_move(choice(initial_moves))

        # Avoid repeated regeneration of legal moves by tracking their status
        legal_moves = [
            [[False for _ in range(board_size + 1)] for _ in range(board_size)]
            for _ in range(board_size)
        ]
        for move in initial_moves:
            legal_moves[move.i][move.j][move.value] = True

        state_matrix: StateMatrixT = {
            "initial": initial_moves,
            "legal": legal_moves,
            "legal_moves_count": len(initial_moves),
        }

        cache = self.load()
        if cache is None: # First turn
            # Create dummy root node for MCTS, the move "belongs" to the opponent
            dummy_root = MCGTNode(
                None, Move(-1, -1, -1), 0, next_player(player_index)
            )
            game_tree = MCGameTree(
                dummy_root,
                game_state,
                state_matrix,
                player_index,   # Maximizing player
            )
            cache = {}
            cache["tree"] = game_tree
            # cache["state_matrix"] = state_matrix
            self.save(cache)
        else: # Return with cached state
            game_tree = cache["tree"]
            # Go to new position in tree
            moves = game_state.moves[-2:] # Moves made since last turn (from argument
```

```
)
            node = game_tree.root
            for move in moves:
                if move not in game_state.taboo_moves: # This move was indeed made since it is not taboo
                    # game_tree.apply_move(player_index, move) # Update state and state matrix
                    for child in node.children: # Move to child
                        if child.move == move:
                            node = child
                            node.parent = None
                            break
                    else: # Child not found
                        node = MCGTNode(None, move, node.depth + 1, next_player(node.player))
            game_tree.root = node
            self.save(cache)

        # Remove data not used in MCTS
        del game_state.taboo_moves
        del game_state.moves
        del game_state.initial_board

        # Keep improving the estimate of best move using MCTS until timeout
        while True:
            game_tree.state_matrix = deepcopy(state_matrix)
            game_tree.game_state = deepcopy(game_state)
            best_move = game_tree.iterate()

            if best_move is not None:
                self.propose_move(best_move)
                # Store game tree to reuse
                # TODO save only if timeout is close
                if game_tree.root.visited % 10 == 0:
                    self.save(cache)

            if DEBUG and game_tree.root.visited == 25:
                graph = build_graph(game_tree.root, player_index, depth_limit=5)
                graph.view()
                print("Graph built")
                exit()
```

Code Listing 4: Different utility functions for game-specific logic. `team33_A3/utils.py` and `team33_MCTS/utils.py`.

```
"""
Module containing helper functions for the sudoku game.

These functions are used for game specific logic, such as checking if a move is legal or calculating the score of a move.
They do not contain any strategic logic specific to the AI agent.
"""
```

```python
from typing import Iterator, TypedDict

from competitive_sudoku.sudoku import GameState, Move, SudokuBoard


class StateMatrixT(TypedDict):
    """
    Type definition for the state matrix.

    :param initial: List of moves that were initially on the board
    :param legal: Array of shape (board_size, board_size, board_size + 1)
                  where legal[i, j, k] is True if Move(i, j, k) is legal
    :param legal_count: count of legal moves
    """
    initial: list[Move]
    legal: list[list[list[bool]]]
    legal_count: int


class SavedDataT(TypedDict):
    """
    Type definition for the saved data.

    :param unsolvable_moves: List of moves that make the board unsolvable
    :param check_unsolvable_range: When to calculate the unsolvable moves
    :param unsolvable_list_building_started: True if the unsolvable list building has started
    :param unsolvable_list_building_finished: True if the unsolvable list building has finished
    """

    unsolvable_moves: list[Move]
    check_unsolvable_range: tuple[float, float]
    unsolvable_list_building_started: bool
    unsolvable_list_building_finished: bool


def next_player(current_player: int) -> int:
    """
    Returns the next player.

    :param current_player: the current player.
    :return: the next player.
    """
    return (current_player + 1) % 2


def block_index(row: int, col: int, board) -> tuple[int, int]:
    """
    Transform in which block a certain coordinate is.
```

```
    Enumerates blocks in the same way as coordinates, top left is (0, 0).

    :param row: a row index inside the block.
    :param col: a column index inside the block.
    :param board: the board to which the indices belong.
    :return: the indices of the block in the board (vertical, horizontal).
    """
    return (
        row // board.region_height(),  # floor division
        col // board.region_width(),
    )


def block_range(*, row: int, col: int, board: SudokuBoard) -> Iterator[tuple[int, int]]:
    """
    Return the range of indices in a block.

    :param row: a row index inside the block.
    :param col: a column index inside the block.
    :param board: the board to which the indices belong.
    :return: an iterator of indices in the block.
    """
    region_width, region_height = board.region_width(), board.region_height()
    block_indices = block_index(row, col, board)

    region_start = {
        "x": block_indices[1] * region_width,
        "y": block_indices[0] * region_height,
    }

    region_end = {
        "x": (block_indices[1] + 1) * region_width,
        "y": (block_indices[0] + 1) * region_height,
    }

    for row in range(region_start["y"], region_end["y"]):
        for col in range(region_start["x"], region_end["x"]):
            yield row, col


def is_illegal(*, move: Move, state: GameState) -> bool:
    """
    Returns whether a move is illegal.

    A move is illegal if it puts a duplicate value in a position, block, row or column.
    Additionally, moves should not be 'taboo', meaning that they make the board unsolvable.
    Illegal moves are not allowed and will result in a loss.
```

```python
    :param move: The move to be checked.
    :param state: The current state of the game.
    :return: Whether the move is illegal (True) or not (False).
    """

    # Check if square is empty
    if state.board.get(move.i, move.j) != SudokuBoard.empty:
        return True

    # Check if move is taboo
    if move in state.taboo_moves:
        return True

    # Check duplicate value in row
    if any(
            state.board.get(row, move.j) == move.value
            for row in range(state.board.board_height())
    ):
        return True

    # Check duplicate value in column
    if any(
            state.board.get(move.i, col) == move.value
            for col in range(state.board.board_width())
    ):
        return True

    # Lastly check for duplicate values in the region
    return any(
        state.board.get(row, col) == move.value
        for row, col in block_range(row=move.i, col=move.j, board=state.board)
    )


def calculate_move_score(game_state: GameState, move: Move) -> int:
    """
    Check if a move completes any regions and returns the score earned.

    Static method, uses less memoy sinds it does not need to be instantiated.

    :param game_state: The current game state. Describes the board, scores and move history.
    :param move: The move to be evaluated
    :return: The score earned by the move (0, 1, 3 or 7)
    """
    row_complete = col_complete = block_complete = True

    # Check if completed a row
    for col in range(game_state.board.board_width()):
        if game_state.board.get(move.i, col) == SudokuBoard.empty:
```

```python
            row_complete = False
            break

    # Check if completed a column
    for row in range(game_state.board.board_height()):
        if game_state.board.get(row, move.j) == SudokuBoard.empty:
            col_complete = False
            break

    # Check if completed a block
    for row, col in block_range(row=move.i, col=move.j, board=game_state.board):
        if game_state.board.get(row, col) == SudokuBoard.empty:
            block_complete = False
            break

    # Return score by move
    regions_complete = int(row_complete) + int(col_complete) + int(block_complete)
    return {
        0: 0,
        1: 1,
        2: 3,
        3: 7,
    }[regions_complete]


def calculate_filling_rate(free_cells: dict) -> float:
    """
    Calculate the percentage of occupied cells on the sudoku board.

    :param free_cells: A dictionary containing the number of free cells, for each region.
        row: list containing a value for each row of the sudoku, corresponding to the number of free cells in the row
        col: list containing a value for each column of the sudoku, corresponding to the number of free cells in the column
        block: bidimensional list containing a value for each block of the sudoku, corresponding to the number of free cells in the block
    """
    num_cells = len(free_cells["row"]) * len(free_cells["col"])
    num_empty_cells = 0
    for empty_in_row in free_cells["row"]:
        num_empty_cells += empty_in_row

    return (num_cells - num_empty_cells) / num_cells


def is_unsolvable(board: SudokuBoard) -> bool:
    """
    Determines whether a sudoku is impossible to solve, given a game state.
    It is used to create the unsolvable move list.
```

```python
    :return: True if the sudoku does not have any solutions, False otherwise.

    """
    board_size = board.board_height()
    num_blocks = board_size // board.region_width()

    # Consider rows of each block
    for row_index in range(board_size):
        for block_ind in range(num_blocks):
            # List containing the values not present in the condidered row of the block
            missing_values = [val for val in range(1, board_size + 1)]

            block_row_index = row_index // board.region_height()
            # List containing the indexes of the rows in the block that are not considered
            missing_row_indexes = [
                index
                for index in range(
                    block_row_index * board.region_height(),
                    block_row_index * board.region_height() + board.region_height(),
                )
                if index != row_index
            ]
            # List containing the indexes of the columns without values in the considered row of the block
            missing_col_indexes = [
                index
                for index in range(
                    block_ind * board.region_width(),
                    block_ind * board.region_width() + board.region_width(),
                )
            ]
            # Build the lists of missing values and missing indexes
            for col_index in range(
                    block_ind * board.region_width(),
                    block_ind * board.region_width() + board.region_width(),
            ):
                value = board.get(row_index, col_index)
                if value != SudokuBoard.empty:
                    missing_values.remove(value)
                    missing_col_indexes.remove(col_index)

            # If the row of the block is empty, then it can't invalidate the sudoku
            if len(missing_values) == board.region_height():
                continue

            # Check the impossibility conditions of sudoku
            for missing_value in missing_values:
                # If the value missing in the block row is present in the same block,
```

```python
                # then that value is not invalidating the sudoku
                found_in_block = False
                for row, col in block_range(
                        row=row_index,
                        col=block_ind * board.region_width(),
                        board=board,
                ):
                    if board.get(row, col) == missing_value:
                        found_in_block = True
                        break
                # Check the next missing value
                if found_in_block:
                    continue

                # Check invalidating values in rows (the missing value must be present in all missing rows)
                found_in_rows_count = 0
                for missing_row_index in missing_row_indexes:
                    for col in range(board_size):
                        if board.get(missing_row_index, col) == missing_value:
                            found_in_rows_count += 1
                # Check the next missing value
                if found_in_rows_count != len(missing_row_indexes):
                    continue

                # Check invalidating values in columns (the missing value must be present in all missing columns)
                found_in_cols_count = 0
                for missing_col_index in missing_col_indexes:
                    for row in range(board_size):
                        if board.get(row, missing_col_index) == missing_value:
                            found_in_cols_count += 1
                # Check the next missing value
                if found_in_cols_count != len(missing_col_indexes):
                    continue

                # If the value has been found in all rows and columns and is not in the block,
                # then the sudoku is impossible to solve
                return True

    # All the missing values have been checked and none of them is invalidating the sudoku
    return False
```