Assignment A3: Team 09

Petros Demetrakopoulos   Stefanos Karamperas   Dimitar Glavinkov

1 Agent Description

The goal of assignment A3 was to further develop and improve the performance of the agent we implemented for assignment A2 using advanced techniques and methods. To this end, three agents were used for this assignment:

- **Agent A2 (baseline)**: the agent we developed during assignment A2 and the basis of our efforts for the development of an improved agent in this assignment.
- **Agent A2_skip**: an improved version of `agent_A2` that was developed as part of this assignment. It incorporates a "smart" skip move logic (discussed in subsection 1.5), which led to better overall performance in terms of scoring game points. This is the best agent we developed in terms of performance and the one we finally submitted to Momotor.
- **Agent A3**: This was our attempt to further improve `agent_A2` using "**Monte Carlo Search Tree"**, an approximation algorithm that replaces Minimax and alpha-beta pruning. We also decided to reuse the "smart skip move" logic from `agent A2_skip` to `agent_A3`, after observing the significant performance gains it provided in the case of the former.

1.1 Baseline Agent_A2 Overview

Our agent from assignment A2 (baseline), as already mentioned, utilized the **Minimax** algorithm to explore and exploit game moves. Additionally, it made use of the following optimizations and heuristic techniques

- **Alpha-beta pruning**
- **Iterative deepening**
- A **custom pruning technique** of nodes representing moves we are certain would not lead to any score increase (referred to as "Known No-Reward Cell Pruning" in the report of assignment A1).
- **Penalizing our agent** (assigning negative score) for moves leading up to states that set up the board in a way where the opponent can easily score points in the next move.
- A **greedy algorithm** used to propose an initial move before Minimax starts its iterative tree search.

1.2 Newly implemented agent improvements

During the development of `agent_A3`, our main point of interest was replacing `Minimax/alpha-beta` pruning with an approximate (thus, potentially more efficient) algorithm for the proposal of optimal moves. To this end, we considered **Monte Carlo Tree Search (MCTS)** to be a good candidate since its logic is based on performing a number of iterative simulations over a given game state (root node). This is very convenient considering how "Competitive Sudoku" works, as MCTS proposes increasingly better moves without rebuilding the whole "search tree" from scratch. Instead, it gradually explores new game states of increasing complexity (expands the search tree) and updates the existing ones between consecutive iterations while heavily depending on running simulations for the rest of the "unexplored" game states (child nodes not yet generated). We found this **"trade-off"** between exploration and exploitation rather promising, given the strict time constraints of 0.1 and 0.5 seconds that our agent must be able to satisfy while playing Competitive Sudoku.

For the technical part of MCTS's implementation, we followed the tutorial by [1], adapting it for use in Competitive Sudoku. We also modified MCTS's **"roll-out policy"**, that is, the logic behind how the roll-out moves are chosen during the MCTS's simulation step. We empirically determined that the "standard" MCTS policy of choosing the roll-out moves randomly led to `agent_A3` performing significantly worse compared to `agent_A2`, which used Minimax, due to meaningless moves being chosen frequently at random. In order to mitigate this, our proposed roll-out move selection policy consists of **evaluating the score potential of all candidate roll-out moves**, **ordering them** in descending order based on that and **randomly picking one of the top k moves** for the roll-out (simulation). Obviously, this policy introduces a classic **"exploration vs exploitation"** trade-off. When the selected value for parameter k is too low, we observe that `Agent_A3` misses meaningful moves that would be beneficial in the long run. On the contrary, using very high values for k leads to more low-scoring moves being considered, therefore making the random selection of a meaningless move more likely. We empirically determined that an optimal value for k is 5 (further discussed in subsection 1.6). To better understand how this logic works, we briefly present the roll-out policy function *select_rollout_move*() in pseudocode snippet 1:

Algorithm 1 Select roll-out move function

```
procedure SELECT_ROLLOUT_MOVE(possible_moves, game_state)
    scored_moves ← []
    for each move ∈ possible_moves do
        move_score ← evaluate_move_score_increase(move, game_state)
        scored_moves[move] ← (move, move_score)
    end for
    scored_moves ← scored_moves.sort_by_move_score(descending = True)
    k ← 5
    if scored_moves.length() ≥ k then
        scored_moves ← scored_moves.first_k_elements(k)
    end if
    return scored_moves.random_choice()
end procedure
```

1.3 Time management strategy

The time management strategy used by both agents implemented for this assignment (`agent_A2_skip`, `agent_A3`) is the same as the one used in assignment A2. Specifically, the agents start by choosing a **random move** from all available legal moves. Then, they proceed to select a **greedy move** from all available legal moves, i.e., the move that awards the highest score in a given game state. Finally, the agents proceed to find and propose an optimal move based on the "advanced" method they implement, namely "Minimax"

(with the addition of the “smart” skip move algorithm) in the case of `agent_A2_skip` and “Monte Carlo Tree Search” in the case of `agent_A3`.
By starting with the low-complexity proposal of a random move, we ensure that our agent cannot be disqualified from the game due to not proposing a move on time. Secondly, taking into consideration the high computational complexity of “advanced” methods (Minimax, MCTS), it is expected that in large boards where the number of available moves is very high and/or when the move time limit is very strict (e.g., 0.1, 0.5 seconds), our agent will fail to reach tree nodes representing meaningful moves on time. Therefore, we “bridge” this performance gap by introducing the proposal of greedy moves, whose calculation may have a higher computational complexity compared to random moves but still offers a much lower overhead compared to advanced methods.

1.4 Scoring function

For the implementation of `agent_A3`, we kept the same scoring function used in `agent_A2`. For each node (game state) in the recursion tree, the score assigned to it is the **difference between the score accumulated by the maximizing player up to this state (S_M) and the score accumulated by the minimizing player (S_m)**. Mathematically, this can be expressed as:

$$\text{game\_state\_value} = S_M - S_m$$

We also used the same *evaluate_move_score_increase*() function as in assignment A2. The purpose of this scoring function is to **return the amount by which a player’s score would change after playing a specified move**. It also includes a penalization logic that penalizes moves (assigns a negative score) which would enable our opponent to score points immediately after our turn. The logic of the scoring function was explained in detail in assignment A2’s report. The move score increase function can be expressed as follows in mathematical terms:

$$\text{score\_increase} = \begin{cases} 1, & \text{if 1/3 scoring elements is completed} \\ 3, & \text{if 2/3 scoring elements are completed} \\ 7, & \text{if all (3/3) scoring elements are completed} \\ \mathit{score\_increase} - \max(\mathit{opp\_points}), & \text{if at least 1 scoring element can be completed in the next turn by the opponent} \\ 0, & \text{Otherwise} \end{cases}$$

where: *opp_points* is a list containing all possible values of score increase that the opponent can get in the next move (**as a result of our move**) and *score_increase* indicates the points that would be awarded to our agent after the move (thus $\mathit{score\_increase} \in \{0, 1, 3, 7\}$).

1.5 Smart Move skip

For the new versions of our agent (`agent_A2_skip`, `agent_A3`), we also decided to implement a functionality that would help us exploit the fact that the last move always awards 7 points due to the nature of the game. However, implementing such a strategy turned out to be more complex than anticipated. Initially, the idea was to wait until the last few turns to infer whether we are on track to make the last move or not and try to play a move that essentially "skips" our turn. If proposed, moves that make the Sudoku unsolvable but are not yet marked as taboo moves would allow us to intentionally skip our move. The reasoning behind the logic of the skipping moves originates from the fact that if there is an even number of empty cells left (and assuming each player fills one cell per move), our agent will not be the one to play the last move; therefore, this would yield the last 7 points

to the opponent. While trying to develop this strategy, we realized that making such moves when there are only a few empty cells left is almost impossible, as at that point, there is little to no ambiguity when it comes to which numbers should go in which cells. This is reflected in the legal moves which are available to our agent.
This led us to the realization that we need to proactively keep the game in a state where our agent is on track to make the last move. Thus, at the beginning of each turn, our agent checks whether the number of empty cells left is even. In such cases, the agent is not on track to make the last move and skipping a turn as soon as possible in order to invert the order of play is warranted. The procedure for finding such moves is quite simple. For each row and column, we need to try to find a cell for which there is only 1 legal move left (i.e., one unique number that can be placed in the cell). Consequently, we can conclude that the unique number, which must be placed in that cell, cannot appear anywhere else in the row or column since it would make the Sudoku board unsolvable. From there, in order to intentionally skip our agent's move, we just need to find another cell in the same row or column which does have the unique number in its set of legal moves and try to place it there. As previously described, this would make the Sudoku unsolvable, which results in no number being actually placed on the board after our move, inverting the order of play.

1.6 Hyper-parameter tuning

After we completed the basic implementation of `agent_A3`, we quickly determined that some hyper-parameters could be fine-tuned to optimize the results. The two basic hyper-parameters that we fine-tuned by empirically testing various values are the following:

1. The **exploration parameter** c in the UCT (Upper confidence bound applied to Trees) formula of Monte Carlo Tree Search. In mathematical terms, the UCT formula we used is this:
$$UCT = \frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$
Where (as shown in the relevant [2] page):
    - w_i is the number of wins for the node considered after the i-th move
    - n_i is the number of simulations for the node considered after the i-th move
    - N_i is the total number of simulations after the i-th move run by the parent node of the one considered
    - c is the exploration parameter
2. The k parameter (discussed in subsection 1.2) is used for the selection of the k best moves (offering the highest score increase) at every iteration of the MCTS simulation step.

Note that both these hyper-parameters are related to the **"exploration vs exploitation"** trade-off we discussed in section 1.2. In order to find the optimal numerical values for these parameters, we **performed various additional experiments**. For reference, we are providing an illustration of `agent_A3`'s performance (win rate) against `greedy_player` on the **empty** 3×3 board, given a move **time limit of 1 second**. The scenario was executed for a total of 10 times. Based on figure 1, it can easily be deduced that the optimal value for parameter c is 1.5. With the c parameter tuned, we then proceeded to find an optimal value for parameter k. Figure 2 clearly illustrates that the optimal value for parameter k is 5, as this is the value where we observe the maximum win rate.

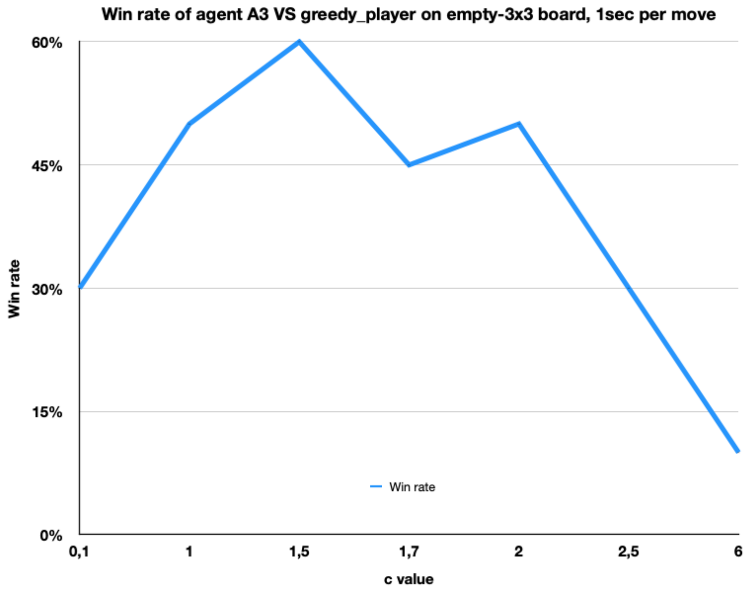


Figure 1: Win rate of `agent_A3` against `greedy_player` with regards to the c parameter value, on an empty 3×3 board given a 1 second time limit per move

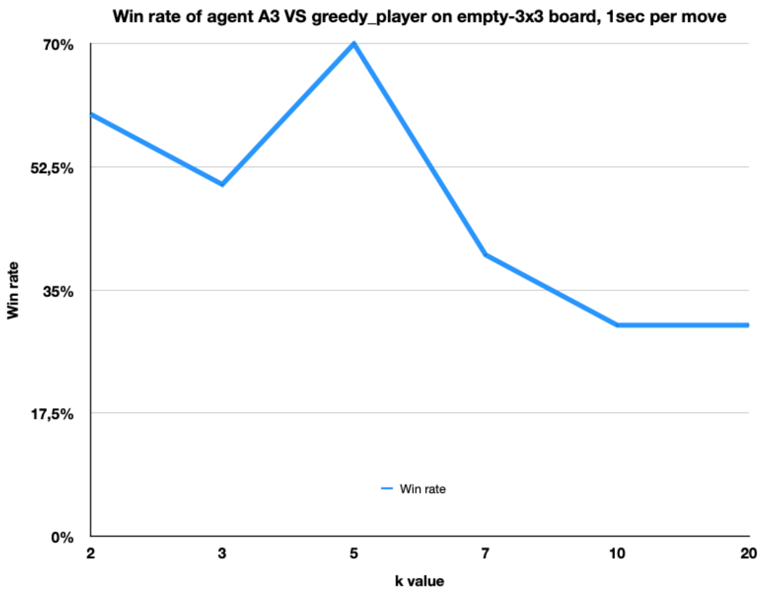
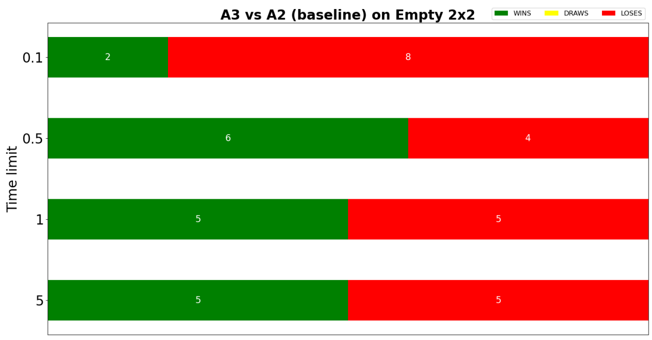


Figure 2: Win rate of `agent_A3` against `greedy_player` with regards to the k parameter value, on an empty 3×3 board given a 1 second time limit per move

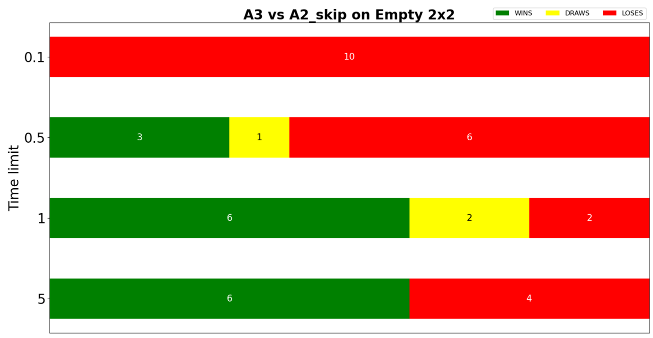
2 Agent Analysis

For the analysis of `agent_A3`, we performed tests **using all 12 given boards** and all move time limit values from previous assignments (0.1, 0.5, 1 and 5 seconds). We considered it meaningful to run our tests against both agent `agent_A2 (baseline)` we submitted for assignment A2 and `agent_A2_skip` (the improved version of `agent_A2` we developed as part of this assignment). So, in total, we executed $2 \times 12 \times 4 = 96$ (2 opponents, 12 boards, 4 different time values) tests in total. Apart from that, we performed **13 additional tests** (7 for the c parameter and 6 for the k parameter) to fine-tune the hyper-parameters of `agent_A3` (as explained in subsection 1.6). For all the tests performed, we used our custom **test execution framework**, which is able to run many tests in parallel by taking advantage of multiple threads to parallelize the execution of individual game rounds between players. The testing framework also takes care of interchanging the order of the agent playing first in order to have 50% of the iterations of each experiment executed with our agent playing first and 50% playing second.

As a general conclusion from our agent analysis, it appears that `agent_A3` **consistently performs worse** in terms of winning rate, both against `agent_A2` and `agent_A2_skip`. A possible explanation for these results is that the method we chose (MCTS) involves a significant amount of randomness, which was not that present in the previous Minimax-based agent implementations. We will now proceed to present the win rates of `agent_A3` against `agent_A2 (baseline)` and `agent_A2_skip` in two boards (a small 2×2 empty board and a large 3×3 empty board). We focus our visualization here on these two specific scenarios, as they sufficiently capture and summarize our agents' performances in smaller and larger boards. The results for these boards are illustrated in Figure 3 and Figure 4 respectively.

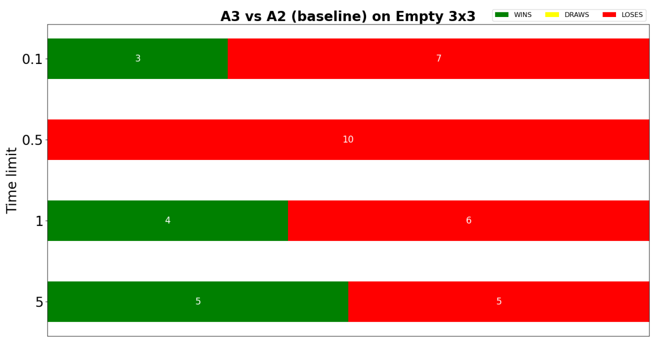


(a) Agent`A3` vs `A2 (baseline)`

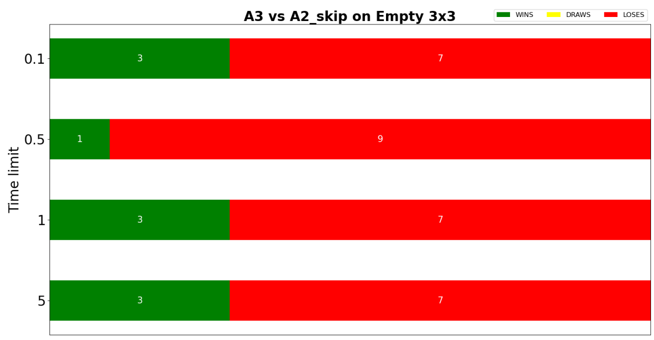


(b) Agent `A3` vs `A2_skip`

Figure 3: Win rates of Agent `A3` against `A2 (baseline)` and `A2_skip` on Empty-2×2 board



(a) Agent `A3` vs `A2 (baseline)`



(b) Agent `A3` vs `A2_skip`

Figure 4: Win rates of Agent `A3` against `A2 (baseline)` and `A2_skip` on Empty-3×3 board

The 3 main conclusions we derive from the analysis and the experiments we executed are the following:

1. `agent_A3` appears to perform better (winning more games) in **smaller boards** than in larger ones
2. `agent_A3` appears to perform slightly better against `agent_A2 (baseline)` than against `agent_A2_skip`

3. It appears that agent_A3 performs better **when more time is given to it**; however, this correlation is not consistently strong across different boards.

Due to limitations concerning the length of the report, we cannot provide all the results and plots we generated during our agent's empirical analysis. However, in case the reader wants to further explore our findings, the raw data of our tests are available on **this Google sheet**.

3 Motivation & Reflection

As far as our newly introduced (agent_A2_skip, agent_A3) agents' strong points are concerned, the new "skip" move strategy (discussed in subsection 1.5) is definitely worth mentioning. The ability to strategically influence the future development of the game, despite the fact that it may lead to short-term disadvantages, has proven to be one of the best improvements introduced to the agents we have implemented so far. Our decision to add it not only to the latest agent (agent_A3) but also to the agent we previously implemented for assignment A2 and include the resulting improved agent_A2_skip agent in our tests was crucial in assessing the individual effect of this new "skip" move strategy. Running separate tests for the agent_A2_skip and (agent_A3) (as presented in section 2) allowed us to better assess the impact of the "skip" move functionality. Given that, we consider our experiment design to be a strong point of our report.

Another strong point of agent_A2_skip and agent_A3 is that neither of them showed a significant difference in the outcome of their games in connection with the order of playing (i.e., whether these agents played as the first or second player in a given game). Both implementations behaved in a consistent way, independently of whether our agent was the first or the second to play.

Regarding the weak points, as we observed during our empirical analysis (section 2), agent_A3 performs worse than agent_A2 (implemented for assignment A2), as well as against agent_A2_skip that was developed as part of this assignment. This is the case because the chosen method, despite being advanced, includes the **element of randomness**, which seems to have a great impact on the outcome compared to the Minimax-based solutions we proposed in previous assignments. It is precisely this effect of randomness which we have identified as one of the inherent weaknesses of the MCTS approach, as it reduces our agents "transparency" and, consequently, significantly limits our ability, as observers, to confidently interpret the worse results obtained by agent_A3.

References

[1] Bosonic-Studios. Monte carlo tree search (mcts) algorithm tutorial and it's explanation with python code. *https://ai-boson.github.io*. URL https://ai-boson.github.io/mcts/. [Online; accessed 13-December-2022].

[2] Wkipedia. Monte carlo tree search. *Wikipedia*. URL https://en.wikipedia.org/wiki/Monte_Carlo_tree_search. [Online; accessed 10-January-2023].

Python files

We included the code of both agent `A3` and agent `A2_skip` that were developed during Assignment 3.

Code Listing 1: A3 agent: `A3/sudokuai.py`.

```
# (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 numpy as np

from competitive_sudoku.sudoku import GameState, Move, SudokuBoard, TabooMove
import competitive_sudoku.sudokuai


# Implementation of MCTS is based on https://ai-boson.github.io/mcts/
class TreeNode:
    # Class representing the nodes of Monte Carlo Tree Search
    def __init__(self, game_state: GameState, parent_node, parent_move,
                                     candidate_moves, num_empty_cells,
                                     is_our_turn=True):
        self.game_state = game_state               # The current game state object
        self.parent_node = parent_node             # The TreeNode object representing this node's parent node
        self.parent_move = parent_move             # The move selected by the parent node of this node
        self.children_nodes = []                   # The children nodes of this node
        self.candidate_moves = candidate_moves     # All available moves provided as potential choices to this node
        self.num_empty_cells = num_empty_cells     # The number of empty cells in the current game state
        self.is_our_turn = is_our_turn             # Flag that indicates whether it is our agent's turn to play

        self.n_value = 0                           # Number of times the node has been visited
        self.win_count = [0, 0]                    # The win counts of both players
                                                   # (index 0: Our agent, index 1: Opponent)
        self.unevaluated_moves = candidate_moves   # A collection of moves that have not been evaluated
                                                   # in terms of potential score yet
        return

    def get_q_value(self):
        """
        Getter method used to return the accumulated "q" (score) value of a node
        @return: an integer representing the accumulated "q" value
        """
        p1_wins = self.win_count[0]
```

```python
        p1_loses = self.win_count[1]

        return p1_wins - p1_loses

    def get_n_value(self):
        """
        Getter method used to return the accumulated "n" (number of visits) value of a node
        @return: an integer representing the "n" value
        """
        return self.n_value

    def get_parent_move(self):
        """
        Getter method used to get the stored parent move of the current node.
        @return: a Move object representing this node's parent move.
        """
        return self.parent_move

    def expand_tree(self):
        """
        Method used to perform the "tree expansion" step of Monte Carlo Tree Search
        @return: the child node of the current node that was created as a result of the tree expansion
        """
        # Pick the next move from the unevaluated moves list
        new_move = self.unevaluated_moves.pop()  # Choose a random unevaluated move to evaluate
        # Calculate the new move's score
        new_move_score = evaluate_move_score_increase(new_move, self.game_state)

        if self.is_our_turn:
            self.game_state.scores[0] += new_move_score
        else:
            self.game_state.scores[1] += new_move_score

        new_game_state = copy.deepcopy(self.game_state)  # Create a copy of the current game state
        new_game_state.board.put(new_move.i, new_move.j, new_move.value)  # Make the new move on the new game state

        updated_empty_cells = get_empty_cells(new_game_state)

        # Find any legal moves we can make at the current game_state
        updated_candidate_moves = legal_moves_after_pruning(new_game_state,
                                                            updated_empty_cells)

        child_node = TreeNode(new_game_state, self, new_move,
                              updated_candidate_moves, self.num_empty_cells - 1,
                              not self.is_our_turn)
```

```
        self.children_nodes.append(child_node)

        return child_node

    def select_rollout_move(self, possible_moves, game_state):
        """
        Method used to select a rollout move from a list of possible moves and for a given game state, every time it
        is called. This method is called during every iteration of the simulation "step" of Monte Carlo Search Tree
        @param possible_moves: a list of "Move" objects representing possible moves
        @param game_state: a GameState object representing the current state of the Competitive Sudoku game
        @return: a "Move" object representing the selected rollout move
        """
        # Sort moves based on their score increase (moves leading to higher score first)
        scores = [evaluate_move_score_increase(move, game_state) for move in possible_moves]
        moves = [x[1] for x in sorted(zip(scores, possible_moves), key=lambda tup: tup[0])]
        moves.reverse()

        # Pick 1 move out of the best k moves
        k = 5
        if len(moves) >= k:
            moves = moves[:k]

        return moves[np.random.randint(len(moves))]

    def is_terminal_node(self):
        """
        Helper method used to determine if the current node is a terminal (leaf) node
        @return: a boolean value (True if the current node is a leaf, False otherwise)
        """
        return len(get_empty_cells(self.game_state)) == 0

    def rollout(self):
        """
        Method used to perform the roll-out (simulation) step of Monte Carlo Search Tree
        @return: the accumulated scores of both players from the executed roll-out (simulation)
        """
        # The simulation (rollout) starts one level below the current node (next move), therefore
        # we change the player flag to indicate that it is the next player's turn
        is_our_turn = not self.is_our_turn
        rollout_game_state = copy.deepcopy(self.game_state)
        available_moves = self.candidate_moves
```

```python
        while available_moves:
            # Select a random move from the available moves
            selected_move = self.select_rollout_move(available_moves, game_state=rollout_game_state)

            # Calculate the selected move's score
            selected_move_score = evaluate_move_score_increase(selected_move, rollout_game_state)

            # "Play" the move on the board
            rollout_game_state.board.put(selected_move.i, selected_move.j, selected_move.value)

            # Update the saved score for the player that is currently playing
            if is_our_turn:
                rollout_game_state.scores[0] += selected_move_score
            else:
                rollout_game_state.scores[1] += selected_move_score

            # Change the player order flag to the next player
            is_our_turn = not is_our_turn

            # Update the available moves
            empty_cells = get_empty_cells(rollout_game_state)
            available_moves = legal_moves_after_pruning(rollout_game_state, empty_cells)

        return rollout_game_state.scores

    def backpropagate(self, result):
        """
        Method used to perform the back-propagation step of Monte Carlo Search Tree
        @param result: a list containing the accumulated scores (game points) of each player after
        the execution of the roll-out (simulation) step.
        """
        self.n_value += 1.

        # Check which player has the most turn wins based on the provided result
        if result[0] > result[1]:
            # Player1 has the most wins
            self.win_count[0] += 1
        elif result[1] > result[0]:
            self.win_count[1] += 1

        # If a parent node exists (current node is NOT root), backpropagate the result
        if self.parent_node:
            self.parent_node.backpropagate(result)

    def is_fully_expanded(self):
```

```python
        """
        Helper method used to check if the current node is fully expanded (has no unevaluated moves left)
        @return: a boolean value (True if the node has no unevaluated moves left, False otherwise)
        """
        return len(self.unevaluated_moves) == 0

    def get_best_child(self, c_param=2):
        """
        Method implementing the UCT formula to select and return the best child of the current node
        @param c_param: a hyper-parameter used to tweak the formula's sensitivity / performance
        @return: the best child node of the current node selected using the UCT formula
        """
        choices_weights = [
            (c.get_q_value() / c.get_n_value()) + c_param * np.sqrt((2 * np.log(self.get_n_value()) / c.get_n_value()))
            for c in self.children_nodes]
        return self.children_nodes[np.argmax(choices_weights)]

    def select_rollout_node(self):
        """
        Method used to select the roll-out node i.e., the node where the simulation step will commence from.
        @return: the selected roll-out node.
        """
        current_node = self

        while not current_node.is_terminal_node():

            if not current_node.is_fully_expanded():
                return current_node.expand_tree()
            else:
                current_node = current_node.get_best_child(c_param=2)

        return current_node


class SudokuAI(competitive_sudoku.sudokuai.SudokuAI):
    """
    Sudoku AI that computes a move for a given sudoku configuration.
    """
    verbose = False  # a flag to print useful debug logs after each turn

    def __init__(self):
        super().__init__()
        self.move_skipped = False
```

```
    def get_greedy_move(self, game_state: GameState, legal_moves):
        """
        Returns the move that awards the most points at the current game_state.
        If there are no moves awarding points, the first move of the list is returned
            arbitrarily .
        @param game_state: The GameState object that describes the current game_state
            of the game in progress
        @param legal_moves: All legal moves that can be performed given the current
            game_state of the game
        @return: A Move object representing the move awarding the mpst points
        """
        max_move = legal_moves[0]
        max_score = -1
        for move in legal_moves:
            max_eval = evaluate_move_score_increase(move, game_state)
            if max_eval > max_score:
                max_move = move
                max_score = max_eval
        return max_move

    def monte_carlo_tree_search(self, game_state: GameState, candidate_moves,
            num_simulations):
        """
        Helper method used to trigger the Monte Carlo Search Tree algorithm
        @param game_state: the current state of the Competitive Sudoku game
        @param candidate_moves: a list of "Move" objects containing all possible
            moves that should be processed
        @param num_simulations: The number of simulation we want the Monte Carlo
            Search Tree algorithm to perform
        """
        num_empty_cells = len(get_empty_cells(game_state))
        root_node = TreeNode(game_state, None, None, candidate_moves,
            num_empty_cells)

        for i in range(num_simulations):
            v = root_node.select_rollout_node()
            result = v. rollout ()
            v.backpropagate( result )

            best_move = root_node.get_best_child(c_param=2).get_parent_move()
            self .propose_move(best_move)

    def get_skip_move(self, legal_moves: list , game_state: GameState):
        """
        Get a move that would make the Sudoku unsolvable, to use it to intentionally
            skip the agent's move
        @param game_state: The GameState object that describes the current state of
            the game in progress
        @param legal_moves: List of all legal moves that are available to the agent at
            the current state
        @return: A Move object representing a move that makes the sudoku unsolvable.
```

```
                                        If such moves does not exists, return None
        """

        # iterate rows to find potential move than can force the agent to "skip" the move
        for i in range(game_state.board.N):
            available_moves_in_row = [move for move in legal_moves if move.i == i]
            available_cells_in_row = list(set([move.j for move in available_moves_in_row]))
            moves_per_cell = {col_index: [] for col_index in available_cells_in_row}
            for move in available_moves_in_row:
                moves_per_cell[move.j].append(move.value)
            non_ambiguous_value = None
            for col_index, moves_list in moves_per_cell.items():
                if len(moves_list) == 1:
                    non_ambiguous_value = moves_list[0]

            if non_ambiguous_value is not None:
                # If a cell in the row can contain only a single value
                # but another cell from the row can also have it
                # then it means that putting the value in the latter one will result in an unsolvable sudoku
                # we want to propose that move in order to "skip" the turn
                for col_index, moves_list in moves_per_cell.items():
                    if len(moves_list) > 1 and non_ambiguous_value in moves_list:
                        return Move(i, col_index, non_ambiguous_value)
        for j in range(0,game_state.board.N):
            available_moves_in_col = [move for move in legal_moves if move.j == j]
            available_cells_in_col = list(set([move.i for move in available_moves_in_col]))
            moves_per_cell = {row_index: [] for row_index in available_cells_in_col}
            for move in available_moves_in_col:
                moves_per_cell[move.i].append(move.value)
            non_ambiguous_value = None
            for row_index, moves_list in moves_per_cell.items():
                if len(moves_list) == 1:
                    non_ambiguous_value = moves_list[0]

            if non_ambiguous_value is not None:
                # If a cell in the row can contain only a single value
                # but another cell from the row can also have it
                # then it means that putting the value in the latter one will result in an unsolvable sudoku
                # we want to propose that move in order to "skip" the turn
                for row_index, moves_list in moves_per_cell.items():
                    if len(moves_list) > 1 and non_ambiguous_value in moves_list:
                        return Move(row_index, j, non_ambiguous_value)

        return None

    def compute_best_move(self, game_state: GameState) -> None:
```

```python
        # Initialize GameState.scores with 0 for both players
        game_state.scores = [0, 0]

        # Filter out illegal moves AND taboo moves
        range_N = range(game_state.board.N)
        range_N_plus_1 = range(1, game_state.board.N + 1)
        legal_moves = []
        for i in range_N:
            for j in range_N:
                for value in range_N_plus_1:
                    if is_possible(i, j, value, game_state) and value not in get_illegal_moves(i, j, game_state):
                        legal_moves.append(Move(i, j, value))

        # Propose a valid move arbitrarily at first (random choice from legal moves), to make sure at least "some" move
        # is proposed by our agent in the given time limit
        random_move = random.choice(legal_moves)
        self.propose_move(random_move)

        # Propose a greedy move (the highest-scoring move available at the current game state)
        move = self.get_greedy_move(game_state, legal_moves)
        self.propose_move(move)

        empty_cells_count = len(get_empty_cells(game_state))

        # Check whether skipping a move is to the best of our interest here using our "skip" move logic
        if empty_cells_count % 2 == 0:
            skip_move = self.get_skip_move(legal_moves, game_state)
            if skip_move is not None:
                self.propose_move(skip_move)
                self.move_skipped = True

        if not self.move_skipped:
            # Monte Carlo Search Tree
            num_simulations = 1000000
            self.monte_carlo_tree_search(game_state, legal_moves, num_simulations)


###### Start of helper functions ######
def get_filled_row_values(row_index: int, game_state: GameState):
    """
    Returns the non-empty values of the row specified by a given row index.
    @param row_index: The row index
    @param game_state: The GameState object that describes the game in progress
    @return: A list containing the integer values of the specified row's non-empty cells
    """
    # returns non-empty values in row with index row_index
```

```python
    filled_values = []
    for i in range(game_state.board.N):
        cur_cell = game_state.board.get(row_index, i)
        if cur_cell != SudokuBoard.empty:
            filled_values.append(cur_cell)
    return filled_values


def get_filled_column_values(column_index: int, game_state: GameState):
    """
    Returns the non-empty values of the column specified by a given column index.
    @param column_index: The column index
    @param game_state: The GameState object that describes the game in progress
    @return: A list containing the integer values of the specified column's non-empty cells.
    """
    filled_values = []
    for i in range(game_state.board.N):
        cur_cell = game_state.board.get(i, column_index)
        if cur_cell != SudokuBoard.empty:
            filled_values.append(cur_cell)
    return filled_values


def get_filled_block_values(row_index: int, column_index: int, game_state: GameState):
    """
    Returns the non-empty values of the (rectangular) block that the cell specified
    by the given row and column indices belongs to.
    @param row_index: The row index
    @param column_index: The column index
    @param game_state: The GameState object that describes the game in progress
    @return: A list containing the integer values of the specified block's non-empty cells
    """
    first_row = (row_index // game_state.board.m) * game_state.board.m
    # A smart way to determine the first row of the rectangular block where the cell belongs to,
    # is to get the integer part of the (row / m) fraction (floor division) and then multiply it by m.
    # The same logic is applied to determine the first column of the rectangular block in question.
    first_column = (column_index // game_state.board.n) * \
                   game_state.board.n
    filled_values = []
    # If first_row is the index of the first row of the block, then the index of the last row should be
    # first_row + game_state.board.m - 1
    for r in range(first_row, first_row + game_state.board.m):
        # If first_column is the index of the first column of the block, then the index of the last column
```

```
        # should be first_column + game_state.board.n - 1
        for c in range(first_column, first_column + game_state.board.n):
            crn_cell = game_state.board.get(r, c)
            if crn_cell != SudokuBoard.empty:
                filled_values.append(crn_cell)
    return filled_values


def is_possible(row_index, column_index, proposed_value, game_state: GameState):
    """
    Determines whether a proposed game move is possible by examining whether the target cell is empty
    and the proposed move in non-taboo.
    @param row_index: The empty cell's row index
    @param col_index: The empty cell's column index
    @param proposed_value: The proposed value to be placed in the specified cell
    @param game_state: The GameState object that describes the game in progress
    @return: True if the proposed move is possible, False otherwise.
    """
    return game_state.board.get(row_index, column_index) == SudokuBoard.empty and not \
        TabooMove(row_index, column_index,
                  proposed_value) in game_state.taboo_moves


def get_illegal_moves(row_index: int, col_index: int, game_state: GameState):
    """
    Returns a list of numbers that already exist in the specified cell's row, column or block. These numbers
    are illegal values and CANNOT be put on the given empty cell.
    @param row_index: The empty cell's row index
    @param col_index: The empty cell's column index
    @param game_state: The GameState object that describes the game in progress
    @return: A list of integers representing the illegal values of the specified empty cell.
    """
    illegal = get_filled_row_values(row_index, game_state) + get_filled_column_values(col_index, game_state) + get_filled_block_values(row_index, col_index, game_state)
    return set(illegal)  # Easy way to remove duplicates


def legal_moves_after_pruning(game_state: GameState, empty_cells):
    """
    Filters the provided legal moves using the defined pruning rules.
    @param game_state: The GameState object that describes the game in progress
    @param empty_cells: A list of integer tuples (i, j) representing the coordinates of empty cells
    @return: A list of Move objects representing the result of the legal move filtering process.
```

```python
    """
    # Prune any cell that we have no information about (the block, row and column containing it are empty). The
    # reasoning behind this pruning is that it is a bit naive to fill in cells for which we have no information
    # and most probably there will be better moves available. This technique significantly reduces the minimax
    # tree size and offers performance advantages.
    # contains all empty cells except the ones for which we have no information
    known_no_reward_cells = []
    if not game_state.board.empty:
        for (row_index, col_index) in empty_cells:
            if not (len(get_filled_row_values(row_index, game_state)) == 0 and
                    len(get_filled_column_values(col_index, game_state)) == 0 and
                    len(get_filled_block_values(row_index, col_index, game_state)) == 0):
                known_no_reward_cells.append((row_index, col_index))
    else:
        # In the case of an empty board, we assign empty_cells to known_no_reward_cells to avoid pruning all cells
        known_no_reward_cells = empty_cells

    # Filter out illegal moves AND taboo moves from the known_no_reward_cells.
    # The resulting list contains all moves which are both possible and LEGAL
    legal_moves = []
    for coords in known_no_reward_cells:
        for value in range(game_state.board.N + 1):
            if is_possible(coords[0], coords[1], value, game_state) and value not in get_illegal_moves(
                    coords[0], coords[1], game_state):
                legal_moves.append(Move(coords[0], coords[1], value))
    return legal_moves


def get_empty_cells(game_state: GameState):
    """
    Returns the empty cells of the sudoku board at a specified game game_state
    @param game_state: The GameState object that describes the current game_state of the game in progress
    @return: A list of integer tuples (i, j) representing the coordinates of the empty cells
    present in the Sudoku board at its current game game_state
    """
    # Compute empty cells coordinates
    # These are the cells that the agent can probably fill
    board_size = game_state.board.N
    empty_cells = [(i, j) for i in range(board_size) for j in range(board_size) if
                   game_state.board.get(i, j) == SudokuBoard.empty]
    return empty_cells
```

```
def evaluate_move_score_increase(move: Move, game_state: GameState, allow_recusion=True):
    """
    Calculates the score increase achieved after the proposed move is made.
    @param move: A Move object that describes the proposed move
    @param game_state: The GameState object that describes the game in progress
    @return: The calculated score increase achieved by the proposed move
    """
    filled_row = get_filled_row_values(move.i, game_state)
    filled_col = get_filled_row_values(move.j, game_state)
    filled_block = get_filled_block_values(move.i, move.j, game_state)

    full_len = game_state.board.N - 1
    score = 0
    # Case where a row, a column and a block are completed after the proposed move is made
    if len(filled_row) == full_len and len( filled_col ) == full_len and len(filled_block) == full_len:
        score = 7
    # Case where a row and a column are completed after the proposed move is made
    elif len(filled_row) == full_len and len( filled_col ) == full_len:
        score = 3
    # Case where a row and a block are completed after the proposed move is made
    elif len(filled_row) == full_len and len( filled_block ) == full_len:
        score = 3
    # Case where a col and a block are completed after the proposed move is made
    elif len(filled_row) == full_len and len( filled_block ) == full_len:
        score = 3
    # Case where only 1 among column, row and block are completed after the proposed move is made
    elif len(filled_row) == full_len or len( filled_col ) == full_len or len(filled_block) == full_len:
        score = 1

    # Case where either a row, a column, a block or a combination of them can be immediately filled during the
    # next game turn, thus easily providing points to the opponent. Our intention is to introduce an artificial
    # "penalty" (not reflected in the final score of the game) for the proposal of such moves. This will force
    # the agent to avoid such moves, as they allow the opponent to immediately score points afterwards.
    is_row_almost_filled = len(filled_row) == full_len - 1
    is_col_almost_filled = len( filled_col ) == full_len - 1
    is_block_almost_filled = len( filled_block ) == full_len - 1
    # The "allow_recurstion" parameter is used to avoid getting stuck in an infinite loop.
    # This check of the allow recursion parameter is needed because when we evaluate our own moves
    # and want to reason about the points the opponent can score with the next move
```

```
    # we want to keep the heuristic out of the calculation and get the actual score the opponent can get
    if allow_recusion:
        full_len_range = range(1, full_len + 2)
        empty_cells = get_empty_cells(game_state)
        if is_row_almost_filled:
            # Calculate which value is missing from the row under examination
            # We do so by finding the difference between the sets containing all the nxm values that must be present in a complete row
            # and the set containing the values that are currently filled in the row
            # we follow the same reasoning for columns and blocks in the following if statements
            missing_value = list(set(full_len_range) - set(filled_row))[0]
            empty_cell_index = [
                x for x in empty_cells if x[0] == move.i][0]
            # Place move that is immediately available for point scoring
            game_state.board.put(
                empty_cell_index[0], empty_cell_index[1], missing_value)
            # Evaluate that move to check how many points it awards
            move_score = evaluate_move_score_increase(
                Move(empty_cell_index[0], empty_cell_index[1], missing_value), game_state, False)
            potential_row_move_points_lost = move_score
            # Remove move from board to return to original game_state
            game_state.board.put(
                empty_cell_index[0], empty_cell_index[1], SudokuBoard.empty)
        else:
            potential_row_move_points_lost = 0

        if is_col_almost_filled:
            # Calculate which value is missing from the collumn under examinationive
            missing_value = list(set(full_len_range) - set(filled_col))[0]
            empty_cell_index = [
                x for x in empty_cells if x[1] == move.j][0]
            # Place move that is immediately available for point scoring
            game_state.board.put(
                empty_cell_index[0], empty_cell_index[1], missing_value)
            # Evaluate that move to check how many points it awards
            move_score = evaluate_move_score_increase(
                Move(empty_cell_index[0], empty_cell_index[1], missing_value), game_state, False)
            potential_col_move_points_lost = move_score
            # Remove move from board to return to original game_state
            game_state.board.put(
                empty_cell_index[0], empty_cell_index[1], SudokuBoard.empty)
        else:
            potential_col_move_points_lost = 0

        if is_block_almost_filled:
            # Calculate which value is missing from the column under examination
```

```
                missing_value = list(
                    set(full_len_range) - set(filled_block))[0]

                first_row = (move.i // game_state.board.m) * game_state.board.m
                first_column = (move.j // game_state.board.n) * \
                               game_state.board.n
                empty_cell_index = [x for x in empty_cells if
                                    x[0] in range(first_row, first_row + game_state.board.
                                          m) and x[1] in range(
                                      first_column, first_column + game_state.board.n)][
                                          0]
                # Place move that is immediately available for point scoring
                game_state.board.put(
                    empty_cell_index[0], empty_cell_index[1], missing_value)
                # Evaluate that move to check how many points it awards
                move_score = evaluate_move_score_increase(
                    Move(empty_cell_index[0], empty_cell_index[1], missing_value),
                                          game_state, False)
                potential_block_move_points_lost = move_score
                # Remove move from board to return to original game_state
                game_state.board.put(
                    empty_cell_index[0], empty_cell_index[1], SudokuBoard.empty)
            else:
                potential_block_move_points_lost = 0
            score = score - max(potential_row_move_points_lost,
                                          potential_col_move_points_lost,
                                potential_block_move_points_lost)

        return score
```

Code Listing 2: A2_skip agent: `A2_skip/sudokuai.py`.

```
#  (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 random
import math
import time
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.
    """
    verbose = False  # a flag to print useful debug logs after each turn

    def __init__(self):
        super().__init__()
        self.move_skipped = False
```

```
        self.N = -1
        self.range_N = range(self.N)
        self.range_N_plus_1 = range(1, self.N + 1)

    def compute_best_move(self, game_state: GameState) -> None:
        N = game_state.board.N
        range_N = range(N)
        range_N_plus_1 = range(1,N+1)

        ################### Start of helper functions ###################
        def get_filled_row_values(row_index: int, state: GameState):
            """
            Returns the non-empty values of the row specified by a given row index.
            @param row_index: The row index
            @param state: The GameState object that describes the game in progress
            @return: A list containing the integer values of the specified row's non-empty cells
            """
            # returns non-empty values in row with index row_index
            filled_values = []
            start = time.time()
            for i in range_N:
                cur_cell = state.board.get(row_index, i)
                if cur_cell != SudokuBoard.empty:
                    filled_values.append(cur_cell)
            return filled_values

        def get_filled_column_values(column_index: int, state: GameState):
            """
            Returns the non-empty values of the column specified by a given column index.
            @param column_index: The column index
            @param state: The GameState object that describes the game in progress
            @return: A list containing the integer values of the specified column's non-empty cells.
            """
            filled_values = []
            for i in range_N:
                cur_cell = state.board.get(i, column_index)
                if cur_cell != SudokuBoard.empty:
                    filled_values.append(cur_cell)
            return filled_values

        def get_filled_block_values(row_index: int, column_index: int, state: GameState):
            """
            Returns the non-empty values of the (rectangular) block that the cell specified
            by the given row and column indices belongs to.
            @param row_index: The row index
```

```
        @param column_index: The column index
        @param state: The GameState object that describes the game in progress
        @return: A list containing the integer values of the specified block's non-empty cells
        """
        first_row = (row_index // state.board.m) * state.board.m
        # A smart way to determine the first row of the rectangular block where the cell belongs to,
        # is to get the integer part of the (row / m) fraction (floor division) and then multiply it by m.
        # The same logic is applied to determine the first column of the rectangular block in question.
        first_column = (column_index // state.board.n) * state.board.n
        filled_values = []
        # If first_row is the index of the first row of the block, then the index of the last row should be
        # first_row + state.board.m - 1
        for r in range(first_row, first_row + state.board.m):
            # If first_column is the index of the first column of the block, then the index of the last column
            # should be first_column + state.board.n - 1
            for c in range(first_column, first_column + state.board.n):
                crn_cell = state.board.get(r, c)
                if crn_cell != SudokuBoard.empty:
                    filled_values.append(crn_cell)
        return filled_values

    def get_illegal_moves(row_index: int, col_index: int, state: GameState):
        """
        Returns a list of numbers that already exist in the specified cell's row, column or block. These numbers
        are illegal values and CANNOT be put on the given empty cell.
        @param row_index: The empty cell's row index
        @param col_index: The empty cell's column index
        @param state: The GameState object that describes the game in progress
        @return: A list of integers representing the illegal values of the specified empty cell.
        """
        illegal = get_filled_row_values(row_index, state) + get_filled_column_values(col_index, state) + get_filled_block_values(row_index, col_index, state)
        return set( illegal )  # Easy way to remove duplicates

    def evaluate_move_score_increase(move: Move, state: GameState, allow_recusion=True):
        """
        Calculates the score increase achieved after the proposed move is made.
        @param move: A Move object that describes the proposed move
        @param state: The GameState object that describes the game in progress
        @return: The calculated score increase achieved by the proposed move
```

```python
    """
    filled_row = get_filled_row_values(move.i, state)
    filled_col = get_filled_column_values(move.j, state)
    filled_block = get_filled_block_values(move.i, move.j, state)

    full_len = N - 1
    score = 0
    # Case where a row, a column and a block are completed after the proposed move is made
    if len( filled_row ) == full_len and len( filled_col ) == full_len and len( filled_block ) == full_len:
        score = 7
    # Case where a row and a column are completed after the proposed move is made
    elif len( filled_row ) == full_len and len( filled_col ) == full_len:
        score = 3
    # Case where a row and a block are completed after the proposed move is made
    elif len( filled_row ) == full_len and len( filled_block ) == full_len:
        score = 3
    # Case where a col and a block are completed after the proposed move is made
    elif len( filled_row ) == full_len and len( filled_block ) == full_len:
        score = 3
    # Case where only 1 among column, row and block are completed after the proposed move is made
    elif len( filled_row ) == full_len or len( filled_col ) == full_len or len( filled_block ) == full_len:
        score = 1

    # Case where either a row, a column, a block or a combination of them can be immediately filled during the
    # next game turn, thus easily providing points to the opponent. Our intention is to introduce an artificial
    # "penalty" (not reflected in the final score of the game) for the proposal of such moves. This will force
    # the agent to avoid such moves, as they allow the opponent to immediately score points afterwards.
    is_row_almost_filled = len( filled_row ) == full_len-1
    is_col_almost_filled = len( filled_col ) == full_len-1
    is_block_almost_filled = len( filled_block ) == full_len-1

    # The allow recurstion parameter is needed so that we don't get stuck in an infinite loop.
    # Essentially what this says is to not consider these rules when this function is called with the
    # purpose of obtaining a score with which we decrease a score of the move we initially wanted to score
    if allow_recusion:
        if is_row_almost_filled:
            # Get missing value in row
```

```
        missing_value = list(set(range(1, full_len + 2)) - set(filled_row))[0]
        empty_cell_index = [x for x in get_empty_cells(state) if x[0] == move.i][0]
        # Place move that is immediately available for point scoring
        state.board.put(empty_cell_index[0], empty_cell_index[1], missing_value)
        # Evaluate that move to check how many points it awards
        move_score = evaluate_move_score_increase(Move(empty_cell_index[0], empty_cell_index[1], missing_value), state, False)
        potential_row_move_points_lost = move_score
        # Remove move from board to return to original state
        state.board.put(empty_cell_index[0], empty_cell_index[1], SudokuBoard.empty)
    else:
        potential_row_move_points_lost = 0

    if is_col_almost_filled:
        # Get missing value in column
        missing_value = list(set(range(1, full_len + 2)) - set(filled_col))[0]
        empty_cell_index = [x for x in get_empty_cells(state) if x[1] == move.j][0]
        # Place move that is immediately available for point scoring
        state.board.put(empty_cell_index[0], empty_cell_index[1], missing_value)
        # Evaluate that move to check how many points it awards
        move_score = evaluate_move_score_increase(Move(empty_cell_index[0], empty_cell_index[1], missing_value), state, False)
        potential_col_move_points_lost = move_score
        # Remove move from board to return to original state
        state.board.put(empty_cell_index[0], empty_cell_index[1], SudokuBoard.empty)
    else:
        potential_col_move_points_lost = 0

    if is_block_almost_filled:
        # Get missing value in block
        missing_value = list(set(range(1, full_len + 2)) - set(filled_block))[0]

        first_row = (move.i // state.board.m) * state.board.m
        first_column = (move.j // state.board.n) * state.board.n
        empty_cell_index = [x for x in get_empty_cells(state) if x[0] in range(first_row, first_row + state.board.m) and x[1] in range(first_column, first_column + state.board.n)][0]
        # Place move that is immediately available for point scoring
        state.board.put(empty_cell_index[0], empty_cell_index[1],
```

```
missing_value)
                # Evaluate that move to check how many points it awards
                move_score = evaluate_move_score_increase(Move(empty_cell_index[0], empty_cell_index[1], missing_value), state, False)
                potential_block_move_points_lost = move_score
                # Remove move from board to return to original state
                state.board.put(empty_cell_index[0], empty_cell_index[1], SudokuBoard.empty)
            else:
                potential_block_move_points_lost = 0

            score = score - max(potential_row_move_points_lost, potential_col_move_points_lost, potential_block_move_points_lost)

        return score

    def possible(row_index, column_index, proposed_value):
        return game_state.board.get(row_index, column_index) == SudokuBoard.empty \
               and not TabooMove(row_index, column_index, proposed_value) in game_state.taboo_moves

    def legal_moves_after_pruning(state, empty_cells):
        # prune any cell that we have no info about it (block, row and column containing it are empty)
        # the reasoning behind this pruning is that it is a bit naive to fill in cells for which we have no information and most probably there will be better options
        # this technique significantly reduces tree size and offers performance advantage
        # known_no_reward_cells list contains all empty cells except the ones that we have no info for
        known_no_reward_cells = []
        if not state.board.empty:
            for cell in empty_cells:
                row_index = cell[0]
                cell_index = cell[1]
                if not (len(get_filled_row_values(row_index, state)) == 0 and
                        len(get_filled_column_values(cell_index, state)) == 0 and
                        len(get_filled_block_values(row_index, cell_index, state)) == 0):
                    known_no_reward_cells.append(cell)
        else:
            # in case of an empty board, we assign empty_cells to known_no_reward_cells
            # otherwise the pruning would prune all cells
            known_no_reward_cells = empty_cells
```

```python
    # filter out illegal moves AND taboo moves from the known_no_reward_cells,
    # the resulting list contains all moves which are both possible and LEGAL
    legal_moves = []
    for coords in known_no_reward_cells:
        for value in range(1, N + 1):
            if possible(coords[0], coords[1], value) and value not in get_illegal_moves(coords[0], coords[1], state):
                legal_moves.append(Move(coords[0], coords[1], value))
    return legal_moves

def get_empty_cells(state):
    """
    Returns the empty cells of the sudoku board at a specified game state
    @param state: The GameState object that describes the current state of the game in progress
    @return: A list of integer tuples (i, j) representing the coordinates of the empty cells
    present in the Sudoku board at its current game state
    """
    # Compute empty cells coordinates
    # These are the cells that the agent can probably fill
    empty_cells = [(i,j) for i in range_N for j in range_N if state.board.get(i, j) == SudokuBoard.empty]
    return empty_cells

# the function that initially triggers the recursion
def find_optimal_move(state, max_depth):
    """
    Used as a helper function that triggers Minimax's recursive call
    @param state: The GameState object that describes the current state of the game in progress
    @param max_depth: The maximum depth to be reached by Minimax's tree
    @return: A Move object representing the best game move determined through Minimax's recursion
    """
    # Initialize max_score with the lowest possible supported value
    max_score = -math.inf
    # find all empty cells
    empty_cells = get_empty_cells(state)

    if len(empty_cells) == 0:
        # game end, all cells are filled , practically reached a leaf node
        return Move(-1, -1, -1)

    # initialize best_move to an invalid move
    best_move = Move(-1,-1,-1)
    # find all possible legal moves for the current game state
    legal_moves = legal_moves_after_pruning(state, empty_cells)
```

```
        for legal_move in legal_moves:
            # Calculate the amount by which the score of the maximizing player will be increased if it plays legal_move
            score_increase = evaluate_move_score_increase(legal_move, state)
            # Make the move
            state.board.put(legal_move.i, legal_move.j, legal_move.value)

            # Increase the score of the player at the current state.
            # The score of the maximizing player is saved at state.scores[0] and
            # the score of minimizing player is saved at state.scores[1]
            if state.scores:
                if state.scores[0]:
                    state.scores[0] += score_increase
                else:
                    state.scores[0] = score_increase
            else:
                state.scores = [0, score_increase]
            cur_max_score = minimax(state, max_depth, 0, -math.inf, math.inf, False)

            # Clear legal_move from the board to continue by checking other possible moves (recursion unrolling)
            state.board.put(legal_move.i, legal_move.j, SudokuBoard.empty)

            # Undo the score increase to continue by checking other possible moves (recursion unrolling)
            state.scores[0] -= score_increase

            if cur_max_score > max_score:
                best_move = Move(legal_move.i, legal_move.j, legal_move.value)
                max_score = cur_max_score

        return best_move

    def get_skip_move(legal_moves: list, game_state: GameState):
        """
        Get a move that would make the Sudoku unsolvable, to use it to intentionally skip the agent's move
        @param game_state: The GameState object that describes the current state of the game in progress
        @param legal_moves: List of all legal moves that are available to the agent at the current state
        @return: A Move object representing a move that makes the sudoku unsolvable. If such moves does not exists, return None
        """

        # iterate rows to find potential move than can force the agent to "skip" the move
        for i in range(game_state.board.N):
```

```
            available_moves_in_row = [move for move in legal_moves if move.i == i]
            available_cells_in_row = list(set([move.j for move in available_moves_in_row]))
            moves_per_cell = {col_index : [] for col_index in available_cells_in_row}
            for move in available_moves_in_row:
                moves_per_cell[move.j].append(move.value)
            non_ambiguous_value = None
            for col_index, moves_list in moves_per_cell.items():
                if len(moves_list) == 1:
                    non_ambiguous_value = moves_list[0]

            if non_ambiguous_value is not None:
                # If a cell in the row can contain only a single value
                # but another cell from the row can also have it
                # then it means that putting the value in the latter one will result in an unsolvable sudoku
                # we want to propose that move in order to "skip" the turn
                for col_index, moves_list in moves_per_cell.items():
                    if len(moves_list) > 1 and non_ambiguous_value in moves_list:
                        return Move(i, col_index, non_ambiguous_value)
        for j in range(0,game_state.board.N):
            available_moves_in_col = [move for move in legal_moves if move.j == j]
            available_cells_in_col = list(set([move.i for move in available_moves_in_col]))
            moves_per_cell = {row_index : [] for row_index in available_cells_in_col }
            for move in available_moves_in_col:
                moves_per_cell[move.i].append(move.value)
            non_ambiguous_value = None
            for row_index, moves_list in moves_per_cell.items():
                if len(moves_list) == 1:
                    non_ambiguous_value = moves_list[0]

            if non_ambiguous_value is not None:
                # If a cell in the row can contain only a single value
                # but another cell from the row can also have it
                # then it means that putting the value in the latter one will result in an unsolvable sudoku
                # we want to propose that move in order to "skip" the turn
                for row_index, moves_list in moves_per_cell.items():
                    if len(moves_list) > 1 and non_ambiguous_value in moves_list:
                        return Move(row_index, j, non_ambiguous_value)

        return None

    def get_greedy_move(state: GameState, legal_moves):
        max_move = legal_moves[0]
        max_score = -1
        for move in legal_moves:
```

```
            max_eval = evaluate_move_score_increase(move, state)
            if max_eval > max_score:
                max_move = move
                max_score = max_eval
        return max_move

    def minimax(state: GameState, max_depth: int, depth: int, alpha: float, beta: float, is_maximizing_player: bool):
        """
        Implementation of the Minimax algorithm that includes alpha-beta pruning
        @param state: The GameState object that describes the current state of the game in progress
        @param max_depth: The maximum depth to be reached by Minimax's tree
        @param depth: The current depth reached by Minimax's tree
        @param alpha: The alpha value (used for alpha-beta pruning)
        @param beta: The beta value (used for alpha-beta pruning)
        @param is_maximizing_player: A boolean flag indicating whether it is the maximizing player's turn to play
        @return: The maximum maximizer-minimizer score difference achieved by the Minimax Algorithm
        """
        empty_cells = get_empty_cells(state)
        # find out any legal moves we can do at the current game state
        legal_moves = legal_moves_after_pruning(state, empty_cells)

        if depth >= max_depth:
            # Max depth reached, returning the score of the node
            return state.scores[0] - state.scores[1]

        if len(legal_moves) == 0:
            # No legal moves left, practically a leaf node The evaluation function of a node is the difference
            # between the score of the maximizer at this state and the score of the minimizer at the same state.
            # This is the quantity that the minimax tries to maximize for the maximizing player and minimize for the
            # opponent
            return state.scores[0] - state.scores[1]

        if is_maximizing_player:
            # Maximizer's move
            # Initialize max_score with the lowest possible supported value
            max_score = -math.inf

            for legal_move in legal_moves:
                # Calculate the amount by which the score of the maximizing player will increase if it plays
                # legal_move
                maximizer_score_increase = evaluate_move_score_increase(legal_move, state)
```

```
                # Play the move (add the move to the sudoku board)
                state.board.put(legal_move.i, legal_move.j, legal_move.value)

                # In the "scores" property of the GameState object we can find the scores of the maximizing and
                # minimizing players. Since our logic is based on the difference of scores between the maximizing
                # player and minimizing player after a move is played, we need to temporarily reflect the move's
                # result on the player score before continuing the search
                if state.scores:
                    if state.scores[0]:
                        state.scores[0] += maximizer_score_increase
                    else:
                        state.scores[0] = maximizer_score_increase
                else:
                    state.scores = [maximizer_score_increase, 0]

                # Call minimax for the minimizing player
                max_score = max(max_score, minimax(state, max_depth, depth+1, alpha, beta, False))

                # Clear legal_move from the board to continue by checking other possible moves (recursion unrolling)
                state.board.put(legal_move.i, legal_move.j, SudokuBoard.empty)

                # Undo the score increase to continue by checking other possible moves (recursion unrolling)
                state.scores[0] -= maximizer_score_increase

                # Implementation of the alpha-beta pruning technique as demonstrated on
                # https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning
                alpha = max(alpha, max_score)
                if beta <= alpha:
                    break

            return max_score
        else:
            # minimizer's move
            # initialize min_score with the highest possible supported value
            min_score = math.inf

            for legal_move in legal_moves:
                # Calculate the amount by which the score of the minimizing player will increase if it plays
                # legal_move
                minimizer_score_increase = evaluate_move_score_increase(legal_move, state)
```

```
                    # Play the move (add the move to the sudoku board)
                    state.board.put(legal_move.i, legal_move.j, legal_move.value)

                    # Increase score for the maximizer in the current state
                    if state.scores:
                        if state.scores[1]:
                            state.scores[1] += minimizer_score_increase
                        else:
                            state.scores[1] = minimizer_score_increase
                    else:
                        state.scores = [0, minimizer_score_increase]

                    # Call minimax for the maximizing player
                    min_score = min(min_score, minimax(state, max_depth, depth+1, alpha, beta, True))

                    # Clear legal_move from the board to continue by checking other possible moves (recursion unrolling)
                    state.board.put(legal_move.i, legal_move.j, SudokuBoard.empty)

                    # Undo the score increase to continue by checking other possible moves (recursion unrolling)
                    state.scores[1] -= minimizer_score_increase

                    beta = min(beta, min_score)
                    if beta <= alpha:
                        break

                return min_score
        #################### End of helper functions ####################

        # Filter out illegal moves AND taboo moves
        legal_moves = [Move(i,j,value) for i in range_N for j in range_N for value in range_N_plus_1 if possible(i, j, value) and value not in get_illegal_moves(i, j, game_state)]
        # Propose a valid move arbitrarily at first (random choice from legal moves),
        # then keep finding optimal moves with minimax and propose them for as long as we are given the time to do so.
        rndm_move = random.choice(legal_moves)
        self.propose_move(rndm_move)
        move = get_greedy_move(game_state, legal_moves)
        self.propose_move(move)

        empty_cells_count = len(get_empty_cells(game_state))
        # If we are not on track to make the last move
        # attempt to skip the move by proposing a move that would make the sudoku unsolvable
        if empty_cells_count % 2 == 0:
            skip_move = get_skip_move(legal_moves, game_state)
```

```
            if skip_move is not None:
                self.propose_move(skip_move)
                self.move_skipped = True
        if not self.move_skipped:
            # Initial Minimax search depth
            max_depth = 0
            while True:
                # Iteratively increase Minimax's tree depth to discover more optimal moves.
                # On each iteration the move proposed should be slightly better than the move of the previous iteration.
                max_depth += 1
                best_move = find_optimal_move(game_state, max_depth)  # Initial call to the recursive minimax function
                if best_move != Move(-1, -1, -1):  # Failsafe mechanism to ensure we will never propose an invalid move
                    if self.verbose:
                        # Print statements for debug purposes
                        print("--------------")
                        print("Random move proposed: " + str(best_move))
                        print("Score for selected legal_move: " + str(evaluate_move_score_increase(best_move, game_state)))
                        print("Illegal moves for selected cell: " + str(get_illegal_moves(best_move.i, best_move.j, game_state)))
                        print("Block filled values for selected cell: " + str(get_filled_block_values(best_move.i, best_move.j, game_state)))
                        print("Row filled values for selected cell: " + str(get_filled_row_values(best_move.i, game_state)))
                        print("Column filled values for selected cell: " + str(get_filled_column_values(best_move.j, game_state)))
                        print("--------------")
                    self.propose_move(best_move)
```