

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

```

1: procedure SELECT_ROLLOUT_MOVE(possible_moves, game_state)
2:   scored_moves  $\leftarrow$  []
3:   for each move  $\in$  possible_moves do
4:     move_score  $\leftarrow$  evaluate_move_score_increase(move, game_state)
5:     scored_moves[move]  $\leftarrow$  (move, move_score)
6:   end for
7:   scored_moves  $\leftarrow$  scored_moves.sort_by_move_score(descending = True)
8:    $k \leftarrow 5$ 
9:   if scored_moves.length()  $\geq k$  then
10:    scored_moves  $\leftarrow$  scored_moves.first_k_elements( $k$ )
11:   end if
12:   return scored_moves.random_choice()
13: 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:

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

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 *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.

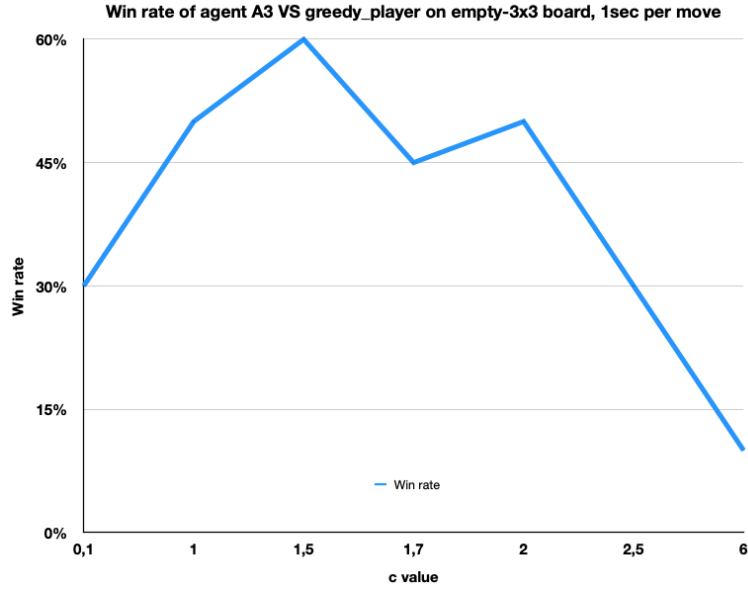


Figure 1: Win rate of `agent_A3` against `greedy_player` with regards to the c parameter value, on an empty 3×3 board given a 1 second time limit per move

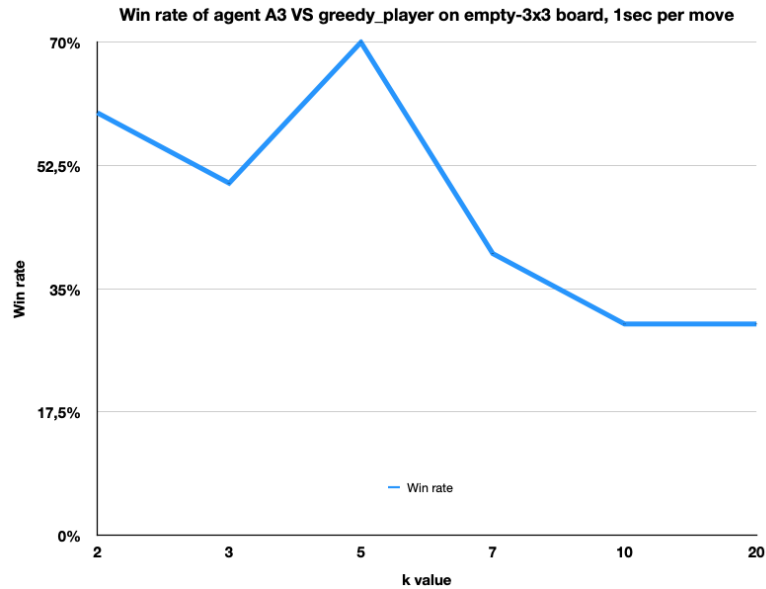


Figure 2: Win rate of `agent_A3` against `greedy_player` with regards to the k parameter value, on an empty 3×3 board given a 1 second time limit per move

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.

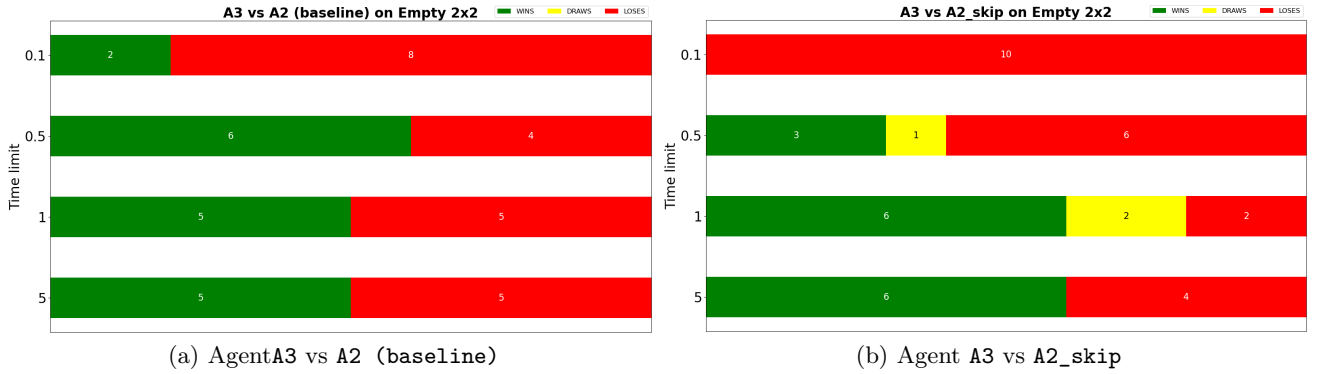


Figure 3: Win rates of Agent A3 against A2 (baseline) and A2_skip on Empty- 2×2 board

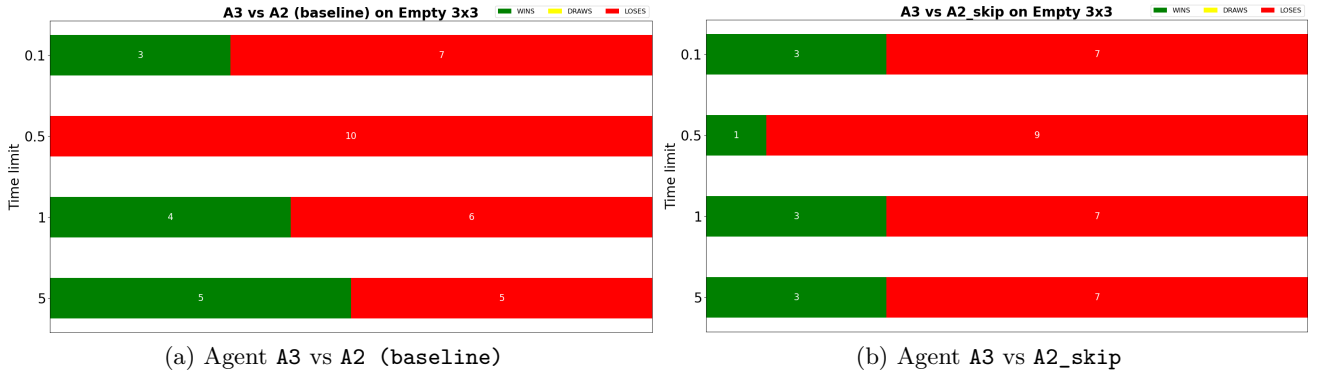


Figure 4: Win rates of Agent A3 against A2 (baseline) and A2_skip on Empty- 3×3 board

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.

```
1  # (C) Copyright Wieger Wesselink 2021. Distributed under the GPL-3.0-or-later
2  # Software License, (See accompanying file LICENSE or copy at
3  # https://www.gnu.org/licenses/gpl-3.0.txt)
4  import copy
5  import random
6  import numpy as np

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

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

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

38     return

39 def get_q_value(self):
40     """
41     Getter method used to return the accumulated "q" (score) value of a node
42     @return: an integer representing the accumulated "q" value
43     """
44     p1_wins = self.win_count[0]
```



```

37         p1_loses = self.win_count[1]

39         return p1_wins - p1_loses

41     def get_n_value(self):
42         """
43         Getter method used to return the accumulated "n" (number of visits) value of a
44         node
45         @return: an integer representing the "n" value
46         """
47         return self.n_value

48     def get_parent_move(self):
49         """
50         Getter method used to get the stored parent move of the current node.
51         @return: a Move object representing this node's parent move.
52         """
53         return self.parent_move

55     def expand_tree(self):
56         """
57         Method used to perform the "tree expansion" step of Monte Carlo Tree Search
58         @return: the child node of the current node that was created as a result of
59                 the tree expansion
60
61         # Pick the next move from the unevaluated moves list
62         new_move = self.unevaluated_moves.pop() # Choose a random unevaluated
63                                                    move to evaluate
64
65         # Calculate the new move's score
66         new_move_score = evaluate_move_score_increase(new_move, self.game_state)
67
68         if self.is_our_turn:
69             self.game_state.scores[0] += new_move_score
70         else:
71             self.game_state.scores[1] += new_move_score

72
73         new_game_state = copy.deepcopy(self.game_state) # Create a copy of the
74                                                           current game state
75         new_game_state.board.put(new_move.i, new_move.j, new_move.value) # Make
76                                                           the new move on the new game state
77
78         updated_empty_cells = get_empty_cells(new_game_state)
79
80         # Find any legal moves we can make at the current game_state
81         updated_candidate_moves = legal_moves_after_pruning(new_game_state,
82                                                             updated_empty_cells)
83
84         child_node = TreeNode(new_game_state, self, new_move,
85                               updated_candidate_moves, self,
86                               num_empty_cells - 1,
87                               not self.is_our_turn)

```

```

81         self.children_nodes.append(child_node)

82     return child_node

83
84
85     def select_rollout_move( self , possible_moves, game_state):
86         """
87         Method used to select a rollout move from a list of possible moves and for a
88         given game state, every time it
89         is called . This method is called during every iteration of the simulation "
90         step" of Monte Carlo Search Tree
91         @param possible_moves: a list of "Move" objects representing possible moves
92         @param game_state: a GameState object representing the current state of the
93         Competitive Sudoku game
94         @return: a "Move" object representing the selected rollout move
95         """
96         # Sort moves based on their score increase (moves leading to higher score
97         first )
98         scores = [evaluate_move_score_increase(move, game_state) for move in
99                   possible_moves]
100         moves = [x[1] for x in sorted(zip(scores , possible_moves), key=lambda tup:
101                                     tup[0])]
102         moves.reverse()
103
104         # Pick 1 move out of the best k moves
105         k = 5
106         if len(moves) >= k:
107             moves = moves[:k]
108
109         return moves[np.random.randint(len(moves))]
110
111
112     def is_terminal_node( self ):
113         """
114         Helper method used to determine if the current node is a terminal (leaf) node
115         @return: a boolean value (True if the current node is a leaf , False otherwise)
116         """
117         return len(get_empty_cells( self .game_state)) == 0
118
119
120     def rollout ( self ):
121         """
122         Method used to perform the roll –out (simulation) step of Monte Carlo Search
123         Tree
124         @return: the accumulated scores of both players from the executed roll –out (
125         simulation)
126         """
127         # The simulation ( rollout ) starts one level below the current node (next move)
128         , therefore
129         # we change the player flag to indicate that it is the next player 's turn
130         is_our_turn = not self.is_our_turn
131         rollout_game_state = copy.deepcopy(self.game_state)
132         available_moves = self.candidate_moves

```

```

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

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

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

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

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

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

146     return rollout_game_state.scores

148     def backpropagate(self, result):
149         """
150         Method used to perform the back-propagation step of Monte Carlo Search Tree
151         @param result: a list containing the accumulated scores (game points) of each
152                        player after
153                        the execution of the roll-out (simulation) step.
154         """
155         self.n_value += 1.

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

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

167     def is_fully_expanded(self):

```

```

168     """
169     Helper method used to check if the current node is fully expanded (has no
170     unevaluated moves left)
171     @return: a boolean value (True if the node has no unevaluated moves left,
172     False otherwise)
173     """
174     return len( self .unevaluated_moves) == 0
175
176 def get_best_child( self , c_param=2):
177     """
178     Method implementing the UCT formula to select and return the best child of the
179     current node
180     @param c_param: a hyper-parameter used to tweak the formula's sensitivity /
181     performance
182     @return: the best child node of the current node selected using the UCT
183     formula
184     """
185     choices_weights = [
186         (c.get_q_value() / c.get_n_value()) + c_param * np.sqrt((2 * np.log( self .
187             get_n_value()) / c.get_n_value()))
188         for c in self .children_nodes]
189     return self .children_nodes[np.argmax(choices_weights)]
190
191 def select_rollout_node( self ):
192     """
193     Method used to select the roll-out node i.e., the node where the simulation
194     step will commence from.
195     @return: the selected roll-out node.
196     """
197     current_node = self
198
199     while not current_node.is_terminal_node():
200
201         if not current_node.is_fully_expanded():
202             return current_node.expand_tree()
203         else:
204             current_node = current_node.get_best_child(c_param=2)
205
206     return current_node
207
208 class SudokuAI(competitive_sudoku.sudokuai.SudokuAI):
209     """
210     Sudoku AI that computes a move for a given sudoku configuration .
211     """
212     verbose = False # a flag to print useful debug logs after each turn
213
214     def __init__(self):
215         super().__init__()
216         self .move_skipped = False

```

```

212 def get_greedy_move(self, game_state: GameState, legal_moves):
213     """
214     Returns the move that awards the most points at the current game_state.
215     If there are no moves awarding points, the first move of the list is returned
216         arbitrarily .
217     @param game_state: The GameState object that describes the current game_state
218         of the game in progress
219     @param legal_moves: All legal moves that can be performed given the current
220         game_state of the game
221     @return: A Move object representing the move awarding the mpst points
222     """
223     max_move = legal_moves[0]
224     max_score = -1
225     for move in legal_moves:
226         max_eval = evaluate_move_score_increase(move, game_state)
227         if max_eval > max_score:
228             max_move = move
229             max_score = max_eval
230     return max_move

231 def monte_carlo_tree_search(self, game_state: GameState, candidate_moves,
232                             num_simulations):
233     """
234     Helper method used to trigger the Monte Carlo Search Tree algorithm
235     @param game_state: the current state of the Competitive Sudoku game
236     @param candidate_moves: a list of "Move" objects containing all possible
237         moves that should be processed
238     @param num_simulations: The number of simulation we want the Monte Carlo
239         Search Tree algorithm to perform
240     """
241     num_empty_cells = len(get_empty_cells(game_state))
242     root_node = TreeNode(game_state, None, None, candidate_moves,
243                           num_empty_cells)

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

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

250 def get_skip_move(self, legal_moves: list, game_state: GameState):
251     """
252     Get a move that would make the Sudoku unsolvable, to use it to intentionally
253         skip the agent's move
254     @param game_state: The GameState object that describes the current state of
255         the game in progress
256     @param legal_moves: List of all legal moves that are available to the agent at
257         the current state
258     @return: A Move object representing a move that makes the sudoku unsolvable.

```

```

253         """
254         If such moves does not exists , return None
255
256         # iterate rows to find potential move than can force the agent to "skip" the
257         move
258         for i in range(game_state.board.N):
259             available_moves_in_row = [move for move in legal_moves if move.i == i]
260             available_cells_in_row = list(set([move.j for move in
261                                             available_moves_in_row]))
262             moves_per_cell = {col_index : [] for col_index in available_cells_in_row }
263             for move in available_moves_in_row:
264                 moves_per_cell[move.j].append(move.value)
265             non_ambiguous_value = None
266             for col_index, moves_list in moves_per_cell.items():
267                 if len(moves_list) == 1:
268                     non_ambiguous_value = moves_list[0]
269
270             if non_ambiguous_value is not None:
271                 # If a cell in the row can contain only a single value
272                 # but another cell from the row can also have it
273                 # then it means that putting the value in the latter one will result
274                 in an unsolvable sudoku
275                 # we want to propose that move in order to "skip" the turn
276                 for col_index, moves_list in moves_per_cell.items():
277                     if len(moves_list) > 1 and non_ambiguous_value in moves_list:
278                         return Move(i, col_index, non_ambiguous_value)
279
280         for j in range(0,game_state.board.N):
281             available_moves_in_col = [move for move in legal_moves if move.j == j]
282             available_cells_in_col = list(set([move.i for move in
283                                             available_moves_in_col]))
284             moves_per_cell = {row_index : [] for row_index in available_cells_in_col }
285             for move in available_moves_in_col:
286                 moves_per_cell[move.i].append(move.value)
287             non_ambiguous_value = None
288             for row_index, moves_list in moves_per_cell.items():
289                 if len(moves_list) == 1:
290                     non_ambiguous_value = moves_list[0]
291
292             if non_ambiguous_value is not None:
293                 # If a cell in the row can contain only a single value
294                 # but another cell from the row can also have it
295                 # then it means that putting the value in the latter one will result
296                 in an unsolvable sudoku
297                 # we want to propose that move in order to "skip" the turn
298                 for row_index, moves_list in moves_per_cell.items():
299                     if len(moves_list) > 1 and non_ambiguous_value in moves_list:
300                         return Move(row_index, j, non_ambiguous_value)
301
302         return None
303
304     def compute_best_move(self, game_state: GameState) -> None:

```

```

298     # Initialize GameState.scores with 0 for both players
299     game_state.scores = [0, 0]

301     # Filter out illegal moves AND taboo moves
302     range_N = range(game_state.board.N)
303     range_N_plus_1 = range(1, game_state.board.N + 1)
304     legal_moves = []
305     for i in range_N:
306         for j in range_N:
307             for value in range_N_plus_1:
308                 if is_possible(i, j, value, game_state) and value not in
309                     get_illegal_moves(i, j, game_state):
310                     legal_moves.append(Move(i, j, value))

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

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

320     empty_cells_count = len(get_empty_cells(game_state))

321     # Check whether skipping a move is to the best of our interest here using our
322     "skip" move logic

323     if empty_cells_count % 2 == 0:
324         skip_move = self.get_skip_move(legal_moves, game_state)
325         if skip_move is not None:
326             self.propose_move(skip_move)
327             self.move_skipped = True

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

332

333     ##### Start of helper functions #####
334     def get_filled_row_values(row_index: int, game_state: GameState):
335         """
336         Returns the non-empty values of the row specified by a given row index.
337         @param row_index: The row index
338         @param game_state: The GameState object that describes the game in progress
339         @return: A list containing the integer values of the specified row's non-empty
340                 cells
341         """
342         # returns non-empty values in row with index row_index

```

```

344     filled_values = []
345     for i in range(game_state.board.N):
346         cur_cell = game_state.board.get(row_index, i)
347         if cur_cell != SudokuBoard.empty:
348             filled_values.append(cur_cell)
349     return filled_values

352 def get_filled_column_values(column_index: int, game_state: GameState):
353     """
354     Returns the non-empty values of the column specified by a given column index.
355     @param column_index: The column index
356     @param game_state: The GameState object that describes the game in progress
357     @return: A list containing the integer values of the specified column's non-
358                empty cells.
359     """
360     filled_values = []
361     for i in range(game_state.board.N):
362         cur_cell = game_state.board.get(i, column_index)
363         if cur_cell != SudokuBoard.empty:
364             filled_values.append(cur_cell)
365     return filled_values

367 def get_filled_block_values(row_index: int, column_index: int, game_state: GameState)
368     :
369     """
370     Returns the non-empty values of the (rectangular) block that the cell specified
371     by the given row and column indices belongs to.
372     @param row_index: The row index
373     @param column_index: The column index
374     @param game_state: The GameState object that describes the game in progress
375     @return: A list containing the integer values of the specified block's non-empty
376                cells
377     """
378     first_row = (row_index // game_state.board.m) * game_state.board.m
379     # A smart way to determine the first row of the rectangular block where the cell
380     # belongs to,
381     # is to get the integer part of the (row / m) fraction (floor division) and then
382     # multiply it by m.
383     # The same logic is applied to determine the first column of the rectangular block
384     # in question.
385     first_column = (column_index // game_state.board.n) * \
386         game_state.board.n
387     filled_values = []
388     # If first_row is the index of the first row of the block, then the index of the
389     # last row should be
390     # first_row + game_state.board.m - 1
391     for r in range(first_row, first_row + game_state.board.m):
392         # If first_column is the index of the first column of the block, then the
393         # index of the last column

```



```

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

395 def is_possible(row_index, column_index, proposed_value, game_state: GameState):
396     """
397     Determines whether a proposed game move is possible by examining whether the
398     target cell is empty
399     and the proposed move in non-taboo.
400     @param row_index: The empty cell's row index
401     @param col_index: The empty cell's column index
402     @param proposed_value: The proposed value to be placed in the specified cell
403     @param game_state: The GameState object that describes the game in progress
404     @return: True if the proposed move is possible, False otherwise.
405     """
406     return game_state.board.get(row_index, column_index) == SudokuBoard.empty and \
407            TabooMove(row_index, column_index,
408                      proposed_value) in game_state.taboo_moves

410 def get_illegal_moves(row_index: int, col_index: int, game_state: GameState):
411     """
412     Returns a list of numbers that already exist in the specified cell's row, column
413     or block. These numbers
414     are illegal values and CANNOT be put on the given empty cell.
415     @param row_index: The empty cell's row index
416     @param col_index: The empty cell's column index
417     @param game_state: The GameState object that describes the game in progress
418     @return: A list of integers representing the illegal values of the specified
419     empty cell.
420     """
421     illegal = get_filled_row_values(row_index, game_state) + get_filled_column_values(
422         col_index, game_state) +
423         get_filled_block_values(row_index, col_index,
424                                game_state)
425     return set(illegal) # Easy way to remove duplicates

426 def legal_moves_after_pruning(game_state: GameState, empty_cells):
427     """
428     Filters the provided legal moves using the defined pruning rules.
429     @param game_state: The GameState object that describes the game in progress
430     @param empty_cells: A list of integer tuples (i, j) representing the coordinates
431     of empty cells
432     @return: A list of Move objects representing the result of the legal move
433     filtering process.

```

```

429     """
430     # Prune any cell that we have no information about (the block, row and column
431         containing it are empty). The
432         # reasoning behind this pruning is that it is a bit naive to fill in cells for
433         which we have no information
434         # and most probably there will be better moves available. This technique
435         significantly reduces the minimax
436         # tree size and offers performance advantages.
437         # contains all empty cells except the ones for which we have no information
438     known_no_reward_cells = []
439     if not game_state.board.empty:
440         for (row_index, col_index) in empty_cells:
441             if not (len(get_filled_row_values(row_index, game_state)) == 0 and
442                     len(get_filled_column_values(col_index, game_state)) == 0 and
443                     len(get_filled_block_values(row_index, col_index, game_state)) == 0):
444                 known_no_reward_cells.append((row_index, col_index))
445     else:
446         # In the case of an empty board, we assign empty_cells to
447         known_no_reward_cells to avoid pruning all
448         cells
449     known_no_reward_cells = empty_cells
450
451     # Filter out illegal moves AND taboo moves from the known_no_reward_cells.
452     # The resulting list contains all moves which are both possible and LEGAL
453     legal_moves = []
454     for coords in known_no_reward_cells:
455         for value in range(game_state.board.N + 1):
456             if is_possible(coords[0], coords[1], value, game_state) and value not in
457                 get_illegal_moves(
458                     coords[0], coords[1], game_state):
459                 legal_moves.append(Move(coords[0], coords[1], value))
460     return legal_moves
461
462 def get_empty_cells(game_state: GameState):
463     """
464     Returns the empty cells of the sudoku board at a specified game game_state
465     @param game_state: The GameState object that describes the current game_state of
466         the game in progress
467     @return: A list of integer tuples (i, j) representing the coordinates of the
468         empty cells
469         present in the Sudoku board at its current game game_state
470     """
471     # Compute empty cells coordinates
472     # These are the cells that the agent can probably fill
473     board_size = game_state.board.N
474     empty_cells = [(i, j) for i in range(board_size) for j in range(board_size) if
475                     game_state.board.get(i, j) == SudokuBoard.empty]
476     return empty_cells

```

```

472 def evaluate_move_score_increase(move: Move, game_state: GameState, allow_recursion=
                                         True):
473     """
474     Calculates the score increase achieved after the proposed move is made.
475     @param move: A Move object that describes the proposed move
476     @param game_state: The GameState object that describes the game in progress
477     @return: The calculated score increase achieved by the proposed move
478     """
479     filled_row = get_filled_row_values(move.i, game_state)
480     filled_col = get_filled_row_values(move.j, game_state)
481     filled_block = get_filled_block_values(move.i, move.j, game_state)
482
483     full_len = game_state.board.N - 1
484     score = 0
485     # Case where a row, a column and a block are completed after the proposed move is
486     # made
487     if len(filled_row) == full_len and len(filled_col) == full_len and len(
488         filled_block) == full_len:
489         score = 7
490     # Case where a row and a column are completed after the proposed move is made
491     elif len(filled_row) == full_len and len(filled_col) == full_len:
492         score = 3
493     # Case where a row and a block are completed after the proposed move is made
494     elif len(filled_row) == full_len and len(filled_block) == full_len:
495         score = 3
496     # Case where a col and a block are completed after the proposed move is made
497     elif len(filled_col) == full_len and len(filled_block) == full_len:
498         score = 3
499     # Case where only 1 among column, row and block are completed after the proposed
500     # move is made
501     elif len(filled_row) == full_len or len(filled_col) == full_len or len(
502         filled_block) == full_len:
503         score = 1
504
505     # Case where either a row, a column, a block or a combination of them can be
506     # immediately filled during the
507     # next game turn, thus easily providing points to the opponent. Our intention is
508     # to introduce an artificial
509     # "penalty" (not reflected in the final score of the game) for the proposal of
510     # such moves. This will force
511     # the agent to avoid such moves, as they allow the opponent to immediately score
512     # points afterwards.
513     is_row_almost_filled = len(filled_row) == full_len - 1
514     is_col_almost_filled = len(filled_col) == full_len - 1
515     is_block_almost_filled = len(filled_block) == full_len - 1
516     # The "allow_recursion" parameter is used to avoid getting stuck in an infinite
517     # loop.
518     # This check of the allow_recursion parameter is needed because when we evaluate
519     # our own moves
520     # and want to reason about the points the opponent can score with the next move

```

```

511     # we want to keep the heuristic out of the calculation and get the actual score
512         the opponent can get
513     if allow_recursion :
514         full_len_range = range(1, full_len + 2)
515         empty_cells = get_empty_cells(game_state)
516         if is_row_almost_filled :
517             # Calculate which value is missing from the row under examination
518             # We do so by finding the difference between the sets containing all the
519             nxm values that must be present in a
520             complete row
521             # and the set containing the values that are currently filled in the row
522             # we follow the same reasoning for columns and blocks in the following if
523             statements
524             missing_value = list(set(full_len_range) - set(filled_row))[0]
525             empty_cell_index = [
526                 x for x in empty_cells if x[0] == move.i][0]
527             # Place move that is immediately available for point scoring
528             game_state.board.put(
529                 empty_cell_index[0], empty_cell_index[1], missing_value)
530             # Evaluate that move to check how many points it awards
531             move_score = evaluate_move_score_increase(
532                 Move(empty_cell_index[0], empty_cell_index[1], missing_value),
533                 game_state, False)
534             potential_row_move_points_lost = move_score
535             # Remove move from board to return to original game_state
536             game_state.board.put(
537                 empty_cell_index[0], empty_cell_index[1], SudokuBoard.empty)
538         else :
539             potential_row_move_points_lost = 0
540
541     if is_col_almost_filled :
542         # Calculate which value is missing from the column under examination
543         missing_value = list(set(full_len_range) - set(filled_col))[0]
544         empty_cell_index = [
545             x for x in empty_cells if x[1] == move.j][0]
546         # Place move that is immediately available for point scoring
547         game_state.board.put(
548             empty_cell_index[0], empty_cell_index[1], missing_value)
549         # Evaluate that move to check how many points it awards
550         move_score = evaluate_move_score_increase(
551             Move(empty_cell_index[0], empty_cell_index[1], missing_value),
552             game_state, False)
553         potential_col_move_points_lost = move_score
554         # Remove move from board to return to original game_state
555         game_state.board.put(
556             empty_cell_index[0], empty_cell_index[1], SudokuBoard.empty)
557     else :
558         potential_col_move_points_lost = 0
559
560     if is_block_almost_filled :
561         # Calculate which value is missing from the column under examination

```

```

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

559         first_row = (move.i // game_state.board.m) * game_state.board.m
560         first_column = (move.j // game_state.board.n) * \
561             game_state.board.n
562         empty_cell_index = [x for x in empty_cells if
563             x[0] in range(first_row, first_row + game_state.board.
564                             m) and x[1] in range(
565                             first_column, first_column + game_state.board.n)][
566             0]
567         # Place move that is immediately available for point scoring
568         game_state.board.put(
569             empty_cell_index[0], empty_cell_index[1], missing_value)
570         # Evaluate that move to check how many points it awards
571         move_score = evaluate_move_score_increase(
572             Move(empty_cell_index[0], empty_cell_index[1], missing_value),
573             game_state, False)
574         potential_block_move_points_lost = move_score
575         # Remove move from board to return to original game_state
576         game_state.board.put(
577             empty_cell_index[0], empty_cell_index[1], SudokuBoard.empty)
578     else:
579         potential_block_move_points_lost = 0
580         score = score - max(potential_row_move_points_lost,
581                             potential_col_move_points_lost,
582                             potential_block_move_points_lost)

580     return score

```

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

```

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

5  import random
6  import math
7  import time
8  from competitive_sudoku.sudoku import GameState, Move, SudokuBoard, TabooMove
9  import competitive_sudoku.sudokuai

12 class SudokuAI(competitive_sudoku.sudokuai.SudokuAI):
13     """
14     Sudoku AI that computes a move for a given sudoku configuration.
15     """
16     verbose = False # a flag to print useful debug logs after each turn

18     def __init__(self):
19         super().__init__()
20         self.move_skipped = False

```

```

21     self .N = -1
22     self .range_N = range(self.N)
23     self .range_N_plus_1 = range(1, self.N + 1)

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

30         ##### Start of helper functions
31         #####
32         def get_filled_row_values(row_index: int, state: GameState):
33             """
34             Returns the non-empty values of the row specified by a given row index.
35             @param row_index: The row index
36             @param state: The GameState object that describes the game in progress
37             @return: A list containing the integer values of the specified row's non-
38                     empty cells
39             """
40             # returns non-empty values in row with index row_index
41             filled_values = []
42             start = time.time()
43             for i in range_N:
44                 cur_cell = state.board.get(row_index, i)
45                 if cur_cell != SudokuBoard.empty:
46                     filled_values .append(cur_cell)
47             return filled_values

48         def get_filled_column_values(column_index: int, state: GameState):
49             """
50             Returns the non-empty values of the column specified by a given column
51             index.
52             @param column_index: The column index
53             @param state: The GameState object that describes the game in progress
54             @return: A list containing the integer values of the specified column's
55                     non-empty cells.
56             """
57             filled_values = []
58             for i in range_N:
59                 cur_cell = state.board.get(i, column_index)
60                 if cur_cell != SudokuBoard.empty:
61                     filled_values .append(cur_cell)
62             return filled_values

63         def get_filled_block_values(row_index: int, column_index: int, state:
64                                     GameState):
65             """
66             Returns the non-empty values of the (rectangular) block that the cell
67             specified
68             by the given row and column indices belongs to.
69             @param row_index: The row index

```

```

66     @param column_index: The column index
67     @param state: The GameState object that describes the game in progress
68     @return: A list containing the integer values of the specified block's non
               -empty cells
69     """
70     first_row = (row_index // state.board.m) * state.board.m
71     # A smart way to determine the first row of the rectangular block where
               the cell belongs to,
72     # is to get the integer part of the (row / m) fraction (floor division)
               and then multiply it by m.
73     # The same logic is applied to determine the first column of the
               rectangular block in question.
74     first_column = (column_index // state.board.n) * state.board.n
75     filled_values = []
76     # If first_row is the index of the first row of the block, then the index
               of the last row should be
77     # first_row + state.board.m - 1
78     for r in range(first_row, first_row + state.board.m):
79         # If first_column is the index of the first column of the block, then
               the index of the last column
80         # should be first_column + state.board.n - 1
81         for c in range(first_column, first_column + state.board.n):
82             crn_cell = state.board.get(r, c)
83             if crn_cell != SudokuBoard.empty:
84                 filled_values.append(crn_cell)
85     return filled_values

87 def get_illegal_moves(row_index: int, col_index: int, state: GameState):
88     """
89     Returns a list of numbers that already exist in the specified cell's row,
               column or block. These numbers
90     are illegal values and CANNOT be put on the given empty cell.
91     @param row_index: The empty cell's row index
92     @param col_index: The empty cell's column index
93     @param state: The GameState object that describes the game in progress
94     @return: A list of integers representing the illegal values of the
               specified empty cell.
95     """
96     illegal = get_filled_row_values(row_index, state) +
               get_filled_column_values(col_index, state)
               + get_filled_block_values(row_index,
               col_index, state)
97     return set(illegal) # Easy way to remove duplicates

99 def evaluate_move_score_increase(move: Move, state: GameState, allow_recursion
               =True):
100     """
101     Calculates the score increase achieved after the proposed move is made.
102     @param move: A Move object that describes the proposed move
103     @param state: The GameState object that describes the game in progress
104     @return: The calculated score increase achieved by the proposed move

```

```

105      """
106      filled_row = get_filled_row_values(move.i, state)
107      filled_col = get_filled_column_values(move.j, state)
108      filled_block = get_filled_block_values(move.i, move.j, state)

110      full_len = N - 1
111      score = 0
112      # Case where a row, a column and a block are completed after the proposed
move is made
113      if len( filled_row ) == full_len and len( filled_col ) == full_len and len(
          filled_block ) == full_len:
114          score = 7
115      # Case where a row and a column are completed after the proposed move is
made
116      elif len( filled_row ) == full_len and len( filled_col ) == full_len:
117          score = 3
118      # Case where a row and a block are completed after the proposed move is
made
119      elif len( filled_row ) == full_len and len( filled_block ) == full_len:
120          score = 3
121      # Case where a col and a block are completed after the proposed move is
made
122      elif len( filled_row ) == full_len and len( filled_block ) == full_len:
123          score = 3
124      # Case where only 1 among column, row and block are completed after the
proposed move is made
125      elif len( filled_row ) == full_len or len( filled_col ) == full_len or len(
          filled_block ) == full_len:
126          score = 1

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

136      # The allow recursion parameter is needed so that we don't get stuck in
an infinite loop.
137      # Essentially what this says is to not consider these rules when this
function is called with the
138      # purpose of obtaining a score with which we decrease a score of the move
we initially wanted to score

139      if allow_recursion:
140          if is_row_almost_filled:
141              # Get missing value in row

```



```

142     missing_value = list (set (range(1, full_len + 2)) - set (filled_row )) [0
143     ]
144     empty_cell_index = [x for x in get_empty_cells(state) if x[0] ==
145         move.i][0]
146     # Place move that is immediately available for point scoring
147     state.board.put(empty_cell_index[0], empty_cell_index[1],
148         missing_value)
149     # Evaluate that move to check how many points it awards
150     move_score = evaluate_move_score_increased(Move(empty_cell_index
151         [0], empty_cell_index[1], missing_value),
152         state, False)
153     potential_row_move_points_lost = move_score
154     # Remove move from board to return to original state
155     state.board.put(empty_cell_index[0], empty_cell_index[1],
156         SudokuBoard.empty)
157
158     else:
159         potential_row_move_points_lost = 0
160
161     if is_col_almost_filled :
162         # Get missing value in column
163         missing_value = list (set (range(1, full_len + 2)) - set (filled_col )) [0
164         ]
165         empty_cell_index = [x for x in get_empty_cells(state) if x[1] ==
166             move.j][0]
167         # Place move that is immediately available for point scoring
168         state.board.put(empty_cell_index[0], empty_cell_index[1],
169             missing_value)
170         # Evaluate that move to check how many points it awards
171         move_score = evaluate_move_score_increased(Move(empty_cell_index
172             [0], empty_cell_index[1], missing_value),
173             state, False)
174         potential_col_move_points_lost = move_score
175         # Remove move from board to return to original state
176         state.board.put(empty_cell_index[0], empty_cell_index[1],
177             SudokuBoard.empty)
178
179     else:
180         potential_col_move_points_lost = 0
181
182     if is_block_almost_filled :
183         # Get missing value in block
184         missing_value = list (set (range(1, full_len + 2)) - set (filled_block ))
185         [0]
186
187         first_row = (move.i // state.board.m) * state.board.m
188         first_column = (move.j // state.board.n) * state.board.n
189         empty_cell_index = [x for x in get_empty_cells(state) if x[0] in
190             range(first_row, first_row + state.board.m
191                 ) and x[1] in range(first_column,
192                     first_column + state.board.n)][0]
193
194         # Place move that is immediately available for point scoring
195         state.board.put(empty_cell_index[0], empty_cell_index[1],

```

```

missing_value)
177     # Evaluate that move to check how many points it awards
178     move_score = evaluate_move_score_increase(Move(empty_cell_index
                                                    [0], empty_cell_index[1], missing_value),
                                                    state, False)
179     potential_block_move_points_lost = move_score
180     # Remove move from board to return to original state
181     state.board.put(empty_cell_index[0], empty_cell_index[1],
                     SudokuBoard.empty)
182     else:
183         potential_block_move_points_lost = 0
184
185     score = score - max(potential_row_move_points_lost,
                        potential_col_move_points_lost,
                        potential_block_move_points_lost)
186
187     return score
188
189 def possible(row_index, column_index, proposed_value):
190     return game_state.board.get(row_index, column_index) == SudokuBoard.
        empty \
191         and not TabooMove(row_index, column_index, proposed_value) in
        game_state.taboo_moves
192
193 def legal_moves_after_pruning(state, empty_cells):
194     # prune any cell that we have no info about it (block, row and column
        containing it are empty)
195     # 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
196     # this technique significantly reduces tree size and offers performance
        advantage
197     # known_no_reward_cells list contains all empty cells except the ones that
        we have no info for
198     known_no_reward_cells = []
199     if not state.board.empty:
200         for cell in empty_cells:
201             row_index = cell[0]
202             cell_index = cell[1]
203             if not (len(get_filled_row_values(row_index, state)) == 0 and
204                     len(get_filled_column_values(cell_index, state)) == 0
205                     and
206                     len(get_filled_block_values(row_index, cell_index, state))
207                         == 0):
208                 known_no_reward_cells.append(cell)
209     else:
210         # in case of an empty board, we assign empty_cells to
            known_no_reward_cells
209         # otherwise the pruning would prune all cells
210         known_no_reward_cells = empty_cells

```

```

212         # filter out illegal moves AND taboo moves from the
213         known_no_reward_cells,
214         # the resulting list contains all moves which are both possible and
215         LEGAL
216
217     legal_moves = []
218     for coords in known_no_reward_cells:
219         for value in range(1, N + 1):
220             if possible(coords[0], coords[1], value) and value not in
221                 get_illegal_moves(coords[0], coords[1],
222                 state):
223                 legal_moves.append(Move(coords[0], coords[1], value))
224
225     return legal_moves
226
227 def get_empty_cells(state):
228     """
229     Returns the empty cells of the sudoku board at a specified game state
230     @param state: The GameState object that describes the current state of the
231                   game in progress
232     @return: A list of integer tuples (i, j) representing the coordinates of
233              the empty cells
234              present in the Sudoku board at its current game state
235     """
236     # Compute empty cells coordinates
237     # These are the cells that the agent can probably fill
238     empty_cells = [(i, j) for i in range_N for j in range_N if state.board.get(
239         i, j) == SudokuBoard.empty]
240
241     return empty_cells
242
243 # the function that initially triggers the recursion
244 def find_optimal_move(state, max_depth):
245     """
246     Used as a helper function that triggers Minimax's recursive call
247     @param state: The GameState object that describes the current state of the
248                   game in progress
249
250     @param max_depth: The maximum depth to be reached by Minimax's tree
251     @return: A Move object representing the best game move determined through
252              Minimax's recursion
253     """
254     # Initialize max_score with the lowest possible supported value
255     max_score = -math.inf
256     # find all empty cells
257     empty_cells = get_empty_cells(state)
258
259     if len(empty_cells) == 0:
260         # game end, all cells are filled, practically reached a leaf node
261         return Move(-1, -1, -1)
262
263     # initialize best_move to an invalid move
264     best_move = Move(-1, -1, -1)
265     # find all possible legal moves for the current game state
266     legal_moves = legal_moves_after_pruning(state, empty_cells)

```

```

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

261         # Increase the score of the player at the current state.
262         # The score of the maximizing player is saved at state.scores[0] and
263         # the score of minimizing player is saved at state.scores[1]
264         if state.scores:
265             if state.scores[0]:
266                 state.scores[0] += score_increase
267             else:
268                 state.scores[0] = score_increase
269         else:
270             state.scores = [0, score_increase]
271         cur_max_score = minimax(state, max_depth, 0, -math.inf, math.inf,
                False)

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

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

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

283     return best_move

285 def get_skip_move(legal_moves: list, game_state: GameState):
286     """
287     Get a move that would make the Sudoku unsolvable, to use it to
                intentionally skip the agent's move
288     @param game_state: The GameState object that describes the current state
                of the game in progress
289     @param legal_moves: List of all legal moves that are available to the
                agent at the current state
290     @return: A Move object representing a move that makes the sudoku
                unsolvable. If such moves does not exists ,
                return None
291     """

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

```

```

295     available_moves_in_row = [move for move in legal_moves if move.i == i
296                               ]
297     available_cells_in_row = list(set([move.j for move in
298                                       available_moves_in_row]))
299     moves_per_cell = {col_index : [] for col_index in
300                       available_cells_in_row }
301     for move in available_moves_in_row:
302         moves_per_cell[move.j].append(move.value)
303     non_ambiguous_value = None
304     for col_index, moves_list in moves_per_cell.items():
305         if len(moves_list) == 1:
306             non_ambiguous_value = moves_list[0]
307
308     if non_ambiguous_value is not None:
309         # If a cell in the row can contain only a single value
310         # but another cell from the row can also have it
311         # then it means that putting the value in the latter one will
312         result in an unsolvable sudoku
313         # we want to propose that move in order to "skip" the turn
314         for col_index, moves_list in moves_per_cell.items():
315             if len(moves_list) > 1 and non_ambiguous_value in moves_list:
316                 return Move(i, col_index, non_ambiguous_value)
317     for j in range(0, game_state.board.N):
318         available_moves_in_col = [move for move in legal_moves if move.j == j]
319         available_cells_in_col = list(set([move.i for move in
320                                           available_moves_in_col]))
321         moves_per_cell = {row_index : [] for row_index in
322                           available_cells_in_col }
323         for move in available_moves_in_col:
324             moves_per_cell[move.i].append(move.value)
325         non_ambiguous_value = None
326         for row_index, moves_list in moves_per_cell.items():
327             if len(moves_list) == 1:
328                 non_ambiguous_value = moves_list[0]
329
330     if non_ambiguous_value is not None:
331         # If a cell in the row can contain only a single value
332         # but another cell from the row can also have it
333         # then it means that putting the value in the latter one will
334         result in an unsolvable sudoku
335         # we want to propose that move in order to "skip" the turn
336         for row_index, moves_list in moves_per_cell.items():
337             if len(moves_list) > 1 and non_ambiguous_value in moves_list:
338                 return Move(row_index, j, non_ambiguous_value)
339
340     return None
341
342 def get_greedy_move(state: GameState, legal_moves):
343     max_move = legal_moves[0]
344     max_score = -1
345     for move in legal_moves:

```

```

339         max_eval = evaluate_move_score_increase(move, state)
340         if max_eval > max_score:
341             max_move = move
342             max_score = max_eval
343     return max_move

345 def minimax(state: GameState, max_depth: int, depth: int, alpha: float, beta:
                                     float, is_maximizing_player: bool):
346     """
347     Implementation of the Minimax algorithm that includes alpha–beta pruning
348     @param state: The GameState object that describes the current state of the
                                     game in progress
349     @param max_depth: The maximum depth to be reached by Minimax's tree
350     @param depth: The current depth reached by Minimax's tree
351     @param alpha: The alpha value (used for alpha–beta pruning)
352     @param beta: The beta value (used for alpha–beta pruning)
353     @param is_maximizing_player: A boolean flag indicating whether it is the
                                     maximizing player's turn to play
354     @return: The maximum maximizer–minimizer score difference achieved by the
                                     Minimax Algorithm
355     """
356     empty_cells = get_empty_cells(state)
357     # find out any legal moves we can do at the current game state
358     legal_moves = legal_moves_after_pruning(state, empty_cells)

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

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

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

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

```

```

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

384     # In the "scores" property of the GameState object we can find the
385     scores of the maximizing and
386     # minimizing players. Since our logic is based on the difference
387     of scores between the maximizing
388     # player and minimizing player after a move is played, we need to
389     temporarily reflect the move's
390     # result on the player score before continuing the search
391     if state.scores:
392         if state.scores[0]:
393             state.scores[0] += maximizer_score_increase
394         else:
395             state.scores[0] = maximizer_score_increase
396     else:
397         state.scores = [maximizer_score_increase, 0]

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

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

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

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

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

420     for legal_move in legal_moves:
421         # Calculate the amount by which the score of the minimizing player
422         will increase if it plays
423         # legal_move
424         minimizer_score_increase = evaluate_move_score_increase(
425             legal_move, state)

```

```

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

425         # Increase score for the maximizer in the current state
426         if state.scores:
427             if state.scores[1]:
428                 state.scores[1] += minimizer_score_increase
429             else:
430                 state.scores[1] = minimizer_score_increase
431         else:
432             state.scores = [0, minimizer_score_increase]

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

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

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

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

447         return min_score
448         ##### End of helper functions #####

450         # Filter out illegal moves AND taboo moves
451         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)]

452         # Propose a valid move arbitrarily at first (random choice from legal moves),
453         # then keep finding optimal moves with minimax and propose them for as long as
454         we are given the time to do so.
455         rndm_move = random.choice(legal_moves)
456         self.propose_move(rndm_move)
457         move = get_greedy_move(game_state, legal_moves)
458         self.propose_move(move)

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

```



```

464         if skip_move is not None:
465             self .propose_move(skip_move)
466             self .move_skipped = True
467     if not self .move_skipped:
468         # Initial Minimax search depth
469         max_depth = 0
470         while True:
471             # Iteratively increase Minimax's tree depth to discover more optimal
472             # On each iteration the move proposed should be slightly better than
473             the move of the previous iteration .
474             max_depth += 1
475             best_move = find_optimal_move(game_state, max_depth) # Initial call
476             to the recursive minimax function
477             if best_move != Move(-1, -1, -1): # Failsafe mechanism to ensure we
478                 will never propose an invalid move
479                 if self .verbose:
480                     # Print statements for debug purposes
481                     print("-----")
482                     print("Random move proposed: " + str(best_move))
483                     print("Score for selected legal_move: " + str(
484                         evaluate_move_score_increase(best_move,
485                                                         game_state)))
486                     print("Illegal moves for selected cell : " + str(
487                         get_illegal_moves(best_move.i, best_move.j,
488                                                         game_state)))
489                     print("Block filled values for selected cell : " + str(
490                         get_filled_block_values(best_move.i,
491                                                         best_move.j, game_state)))
492                     print("Row filled values for selected cell : " + str(
493                         get_filled_row_values(best_move.i,
494                                                         game_state)))
495                     print("Column filled values for selected cell : " + str(
496                         get_filled_column_values(best_move.j,
497                                                         game_state)))
498                     print("-----")
499                 self .propose_move(best_move)

```
