Assignment 3: Team 41

Sven Steens Riccardo Torlaini Rafał Polak

9-1-2023

1 Introduction

Note that due to ambiguity within the assignment/rules description, we will be slightly altering the naming. We will be calling:

- A vertical set of numbers - a **column**
- A horizontal set of numbers - a **row**
- A n x m rectangular set of numbers - a **region**
- Any of the above generically - a **group**
- A "cell" which can be filled with precisely one number - a **square**
- A move that is legal, not a taboo move, but makes the board unsolvable (so that a next move like this one would be a taboo move) - a **bomb move**

Such naming is used both by this report and inside the code. Additionally, some functions in the code may use the term *amount* to refer to cardinalities of countable sets, but it is not to be confusing with the term *number* which we frequently use to refer to the digit value being placed on a Sudoku grid.

An attempt was made for the information in the report, as well as the provided comments, to be enough to sufficiently understand the code. In the case that some aspects of it are still unclear, one may refer to Section 5 (Code overview) for some additional block-by-block descriptions.

2 Agent Description

The main point of the A3 changes was the attempt to employ a Monte Carlo Tree Search (MCTS) as the main move-proposing algorithm, instead of Minimax. Besides it, we implemented time checking which can make the agent conclude the time alloted for a move in that game and reworked the Sudoku Solver completely. Additional feature of using a neural-like solution was attempted, but due to poor results, we abandoned it.

2.1 NEAT

Our first attempt on replacing the minimax algorithm was made using the NeuroEvolution of Augmenting Topologies (NEAT) algorithm. This is a genetic algorithm which attempts to create a neural network via a process similar to evolution. Our goal was to create a network for each board size between 2x2 and 4x4. We would then store them and call up the relevant network to decide upon a move. In theory, this could provide great performance on a constant, low running time.

2.1.1 Explaining NEAT

For our attempts we used the default version of NEAT, however, the NEAT algorithm is very mailable. As such not everything explained here will apply to every version of the algorithm (just most of it).

A NEAT network consists of Nodes and Connections between those nodes. Each node consists of a set of variables that convert the inputs of the node into an output. In the default version, these include bias, response, an activation function and an aggregation function. Between the nodes exist connections, which carry the output of some node to the input of another. The connections might also have variables to manipulate the value such as a bias. There will always be specific nodes which will contain the output of the algorithm, as well as special connections that feed the input into the system.

The algorithm itself works in generations. The first generation will be generated according to a set template you can design. Afterwards, in each generation the best-performing networks, according to some custom-programmed evaluation function, will be chosen. The networks will then be used to create the next generation via random mutation of the network. This mutation includes adding and removing nodes or connections, as well as changing the different variables they contain. How often and by how much these variables change can, and has to, be carefully curated via NEAT's configurations. To make this work properly aspects such as different species and stagnation also exist, but these are not required to understand the core of the algorithm.

2.1.2 Our Attempt

For our implementation we used the neat-python library [1]. Our network format has an input for each square, denoted with either 0 for no value or 1 for having a value. The network also has an output for each square, where the output with the highest value will be chosen to be played.

The main aspect to get right for NEAT to work is a proper evaluation function. Our evaluation consists of two phases. In the first phase, the network is presented with 6 boards: an empty board, a full board and two boards that are filled around 33% and 66% each. The actual percentages are varied via normal distribution to add more uncertainty. For each board, we ask it to return 0.1 or more for the squares that can be filled in and -0.1 or less for squares that can't. For any answers wrong the network's fitness is penalized. Only if this first phase is performed adequately is it allowed to go to the second phase. This ensures the network learns which moves are and aren't possible before we test their actual abilities.

For the second phase, we test how well each network plays the game. We do this via a set of rounds. Each round the networks that are left are put into groups of 4. 3 boards will be generated with 0, 33% and 66% filled (again using variation in the percentages). Each network in a group plays two games on every board against every other player, both starting and second. The network which performs the best in each group will be kept for the next round and rewarded with an increase in fitness. This continues until only one network is left.

However, despite the seemingly good setup and the theoretical groundedness of the NEAT algorithm, we were not able to get any networks to pass the first phase after a full day of training even after attempting different configurations. The training configurations of the NEAT algorithm (as discussed in section 2.1.1) are notoriously hard to get right, so we suspect this might lie at the root of our problem. Optimizing the configuration would however take quite a long time since not only are there many of them (around 50 different settings to change) but seeing the performance of a new setup can easily take half an hour or more to see the results. Because of this, we decided against continuing with the NEAT

algorithm, as it seemed unlikely to be finished in a reasonable time.

2.2 Monte Carlo Tree Search

As explained in the lectures, this is a relatively fast way to assign evaluation scores to each position on the board that uses a deterministic part alongside a random part. Similar to Minimax, the MCTS portrays each game state as a node in the vast game tree of all possible states. What the Minimax does is to try to traverse this entire tree by evaluating each possible state stemming from a given state. Monte Carlo Tree Search explores this tree only partially and from a certain point on, samples the rest of the tree by playing a fully random game - hence the name Monte Carlo. This sampling is really fast, as for each state that the game takes, a move is played without evaluating all the other options, so the agent considers a linear number of moves, not an exponential one. Moreover, in sudoku, playing a move always limits the number of moves left to be played and never introduces any (contrary to e.g. chess, where playing *1. Ke2* is impossible, but after *1. e4*, the move *2. Ke2* is possible). After reaching a terminal state of such a random game (*terminal state* meaning obtaining a full board), the outcome of this game is passed upwards through each node's ancestor until the root and added to the score at each of these nodes. Of course, each node (a state in the game tree) has its score initialized to be 0. Lastly, the *score* can be defined in two main ways - the final point evaluation of the game (e.g. "-14" would mean that the second-starting player won with a 14 points margin) or the ternary indicator of the result ("-1", "0" or "1" to determine the winner). We are using the latter approach, sending the ternary result and not the score.

Remember that the algorithm determines the branch to be visited based on some value per node that we will call UCT (Upper Confidence Bound). It incentivises the procedure to visit any nodes that are not being visited frequently and also those nodes that on average return good scores for our agent. This way, we get more information about the promising moves. At the end, when proposing the final move to be played, the algorithm may choose the path that returned the highest score on average (we use a flag *global_selection = "max"* for that one) or the one that has been visited the most (*global_selection = "robust"* in that case). The advantage of using *"max"* is obviously maximizing the score, but it is entirely possible that the branch returning the highest score was just visited very few times and the MCTS was "lucky" with the random game played with this starting point. Using *"robust"* is a way to play a decent move, yet one we are very certain about.

2.3 Better Sudoku Solver

To reach a better Sudoku solver, we have decided to attempt an implementation of Knuth's Algorithm X in Python, with some modifications to specifically apply it to Sudoku.
Firstly, we should discuss what exactly this Algorithm X is.

Algorithm X is a recursive and non-deterministic algorithm to solve the exact cover problem [2]. An 'exact cover' is formally defined as '*a collection of subsets of a set X such that each element of X is contained in exactly one of those subsets*'. Sudoku can be intuitively seen as such a problem, as it consists of attempting to find (for each cell) a collection of moves such that each number only appears once in each cell.

We detail here a more generic pseudocode version of Algorithm X to give some theoretical backing:

Algorithm_X
Input: A matrix A consisting of 0s and 1s

Output: A reduced matrix A where 1 appears in all the columns exactly once.

1. IF A has no columns:
   (a) RETURN A
2. **c** <- A column in A chosen by some function
3. **r** <- A row in A chosen arbitrarily s.t. $A_{r,c} = 1$. Note that A_r is included in the partial solution.
4. FOR j with $A_{r,j} = 1$:
   (a) FOR i with $A_{i,j} = 1$:
      i. DELETE A_i from A
   (b) DELETE A_j from A
5. **Algorithm_X**(A)

However, such an implementation would spend an inordinate amount of time searching for 1's. When selecting a column, the entire matrix would have to be searched for 1's. When selecting a row, an entire column would have to be searched for 1's. After selecting a row, that row and a number of columns would have to be searched for 1's.
Hence, Knuth proposed using a technique known as 'dancing links' (DLX), particularly useful for implementing a backtracking algorithm such as, of course, Algorithm_X. It uses circular doubly-linked lists, and is based around the observation that in a circular doubly-linked list, $x.right.left = x$ and $x.left.right = x$. Further, each column is also given a header (the headers will form a special row consisting of all the columns which still exist in the matrix). In this way, when an elimination occurs, all columns for which the selected row contains a 1 are removed, along with all rows (including the selected row) that contain a 1 in any of the removed columns. The columns are removed because they have been filled, and the rows are removed because they conflict with the selected row.

Now, it must be noted our implementation is actually based on the one by Computer Science graduate Ali Assaf, as provided on his website[1]. Thus, we do not use doubly-linked lists like in Knuth's DLX, but sets, in order to improve performance in Python. Some other heuristics are documented Section 5.

2.4 Time checking

For this stage of the assignment, we also implemented quite a "cheaty" trick. Despite the time format not being announced to the agents at the start of the game, we can make our agent essentially "skip" the first move (by only proposing a random valid move, without checking "how good" it is) to find it out. After proposing this random move, we update the time elapsed in an endless loop and use the save functionality to save the time elapsed, until - of course - the game simulation breaks the loop by a timeout. This timer recognition will technically miss some tiny details (the time used for imports or agent's initialization, as only those trigger before), but the outcomes of this trick proved to be quite precise. This timer knowledge can then be used by the move proposal functionality. For example, with Minimax, when one knows how long an iteration takes given the board size, number of empty squares and depth, the knowledge of the timer may help select the best maximum depth at an instant. One could e.g. skip the iterations with max depth equal to 1, 2, 3 and

[1]https://www.cs.mcgill.ca/ aassaf9/python/algorithm_x.html

4 and immediately proceed with the run on max depth equal 5. There are three major drawbacks of this functionality, though:

Firstly, the first move may be poor. That should not matter too much though, as the first move usually yields no points.

Moreover, this functionality only works for Minimax. With MCTS, there is no max depth assumed with each iteration, so this simply serves no purpose large in that case. What we do with this functionality on the MCTS case is avoid calling *self.propose_move* any time we update the current best move and only call it after 95% of our time per turn has passed. This might help run just a few more iterations of the MCTS algorithm in the time frame of our turn.

Lastly, the time saved by avoiding running low max depth runs of the Minimax, is less than any useful amount. Since the number of moves in a game tree rises approximately exponentially for large boards, the time saved can never account for more than one additional run with a larger max depth. We will only "gain" additional depth if the agent would have been very close to finishing up a run of that depth anyway. Analysis of this is a bit tricky due to heuristics, but nonetheless the gain due to that time checking turned out to be marginal. We left it in the A3 agent though, as the task was to get fancy and that certainly looks like a fancy feature!

3 Agent Analysis

Before discussing our main analysis we would like to note two weird aspects of the algorithm's performance. Firstly the time to set up the AI (i.e. importing and loading in the code) is much longer when simulating the game as opposed to running the AI separately, from roughly 0.03 to 0.11 seconds. This might be caused by a quirk of the game simulation code (or python) that we are yet to discover. Importantly however this means that the AI will never make a 0.1 seconds deadline. The second point is regarding its stability since on several occasions the model caused python to stop responding at all during our running time tests. No other algorithm that we tested caused this, so we are unsure why it happens. It does however seem to be running time dependant as it has only occurred on the empty 4x4 board.

3.1 Monte Carlo Variables

Our main points to test were the optimal values for each of the three variables described in Section (2.2). These values are C, whether to use the scores or results to update q and whether to use max or robust selection. For this, we had the player play against the random player on different settings and measured the resulting win rates. The C was varied between [0, 2, 5, 10], while the backtracking varied between "*totals*" and "*scores*" and the selection was either "*max*" or "*robust*". For each combination of the settings, our agent played two games on each board, once as the starting player and once as the second player. The results of these tests are summarized in figure 3.1.

From these results, it seems that the performance is hardly dependent of the chosen settings, as none of the values significantly affect the win rate when changed. Additional investigation into the specific combinations also revealed none that had win rates too high or low to not be attributed to random chance. Furthermore, while not the goal of the test the algorithm finished the tests with a % win rate. Considering that the opponent of choice was the random player, this would heavily suggest the algorithm in its current state is making random moves too. Lastly, we can also observe that the algorithm performs particularly poorly on smaller boards, though we have no strong theories on why this might be.

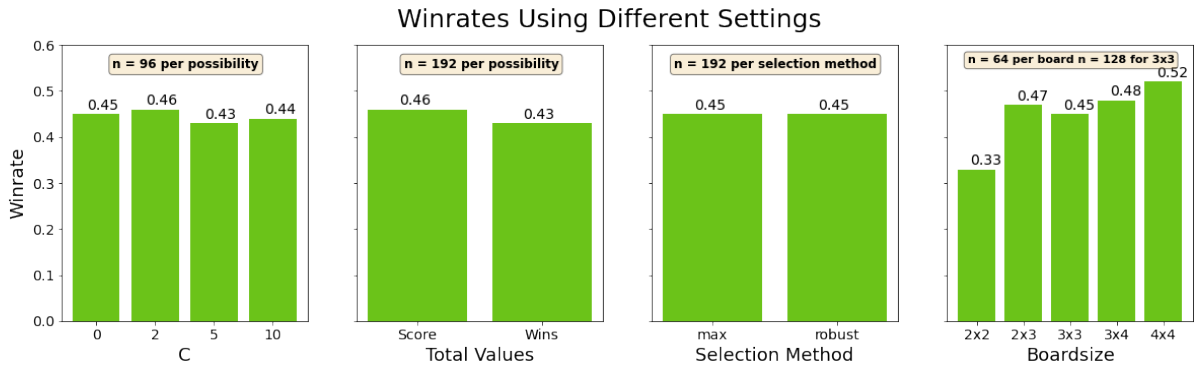

Figure 1: The results of the different settings test for the algorithm. Draws are accounted as losses

3.2 Agent performance

As seen in the previous section, the agent performs with effectively random results. As a result, we theorize that full performance testing of our agent in a different setting would not be more interesting than making the random player play. Preliminary testing, utilizing two different boards four times and two players showed that, although not statistically proven, this theory seems to be the case, as all resulted in a 50% win-rate or worse. Because of this, no full test was performed in lieu of spending time on potentially improving the agent.

4 Reflection

The final decision regarding this agent is straightforward - not send it to the tournament. The MCTS is certainly a nice method to dig into, but it seems like our implementation did not succeed in the time frame given. Instead of focusing on improving this Monte Carlo AI, we shifted a great deal of our efforts to improving our Minimax AI, with the intention of great performance during the tournament. What exactly did go wrong with the Monte Carlo Tree Search? One hypothesis was that the lack of transposition tables essentially leads to an untrue game tree, making it artificially sparser than it is in reality.

Further, it is sad to admit that the turn skipping was likely unfeasible to implement even in the best of cases. At least from our research and reflections, it became clear that in order to implement such a function with the problem restraints given we would need to be able to compute a solved board at any iteration where we may have wished to skip; but if we *could* implement that, that would mean having a Sudoku solver that works reliably and is much faster than what we currently have, which is exactly what we are trying to develop! Hence, the turn skipping was dropped entirely for the finished product.

5 Code overview

Lines 11 - 13: Setting some parameters. The "C" parameter is the trade-off parameter between visiting unexplored vs. promising nodes during the MCTS Selection step. The "*total*" parameter determines whether in the score update part, we update the score of each node with the score of the game or the ternary variable determining who won. "*Selection*" parameter is used in determining the move to finally be played - check Section 2.2 for details.

Line 41: *possible_moves* is a dictionary that stores all squares of the board with no number assigned yet. *numbers_left* is a dictionary that for each group, list the numbers not

yet present in this group (placing them may still result in bomb moves). *empty_squares* is used for bookkeeping the number of possible moves (also called "empty squares" per each group). It is used solely for speed purposes.

Line 44: A call to the functionality to propose any valid move instantly, without running Minimax or MCTS. Implemented to prevent losing on blitz timers.

Lines 47 - 51: The time checking functionality described in Section 2.4.

Lines 53 - 54: The call to the brand new Sudoku Solver described in Section 2.3.

Lines 60 - 61: Creating the data structure of the root of the game tree for the MCTS algorithm.

Lines 63 - 66: Within almost all the given time per turn, iterate the MCTS algorithm via this function that runs it once.

Lines 68 - 81: After our time per turn has almost run out, determine the best move to play. Use the *"max"* or *"robust"* flag explained in Section 2.2.

Lines 89 - 104: The functionality to propose any valid move instantly, without running Minimax or MCTS.

Lines 106 - 128: Top-level view of the MCTS algorithm. First runs the Selection step iteratively, then the Expansion step and the Simulation step. The steps are implemented separately. Lastly, pushes the result of the game up the tree.

Lines 154 - 158: Selection: always follow the branch with the highest UCT at the branch's root.

Lines 160 - 203: Expansion: once the MCTS stumbled upon a node that does not branch any further (has no children) but is not a leaf of the game tree (filled board), the Expansion step starts. As explained in the lecture, the algorithm then creates entries (nodes) for all possible children (possible moves from that state). These nodes are initialized with proper values, such as the points earned by playing this move (with the proper sign flip in line 178.) or a list of open squares of the board after this move. The UCT for each child is initialized as infinity.

Lines 205 - 216: Run once a Simulate step was completed. It propagates the score received through a simulation up the game tree, updating the nodes' scores and their UCTs as well.

Lines 218 - 249: Simulate step. Runs a simulation of the random game from a given state. Notice how in sudoku, playing a move can never introduce new possible moves, so to account for randomness, we could just use shuffle(current possible moves) and then iterate over them, updating the points per player (by updating the values of the data structure bookkeeping the amounts [numbers] of empty squares per group and checking if they become a 0) in an alternating fashion. At the end, the final score of the game is returned (the conversion to the ternary winner-determining score is done inside *one_mc_loop*), to be propagated up the tree.

Lines 259 - 305: The function creating the data structures described in the description of line 41.

Lines 308 - 402: The Sudoku Solver remastered. It uses Donald Knuth's Algorithm X to solve the Sudoku in the order of thousands times faster than our previous algorithm (it is insanely fast!). For an explanation of it, check Section 2.3

References

[1] Alan McIntyre et al. *neat-python*.

[2] *Exact cover problem Wikipedia page*. `https://en.wikipedia.org/wiki/Exact_cover`. Accessed: 2023-01-11.

Python files

Code Listing 1: `sudokuai.py`.

```
from copy import deepcopy
from math import log, sqrt
from random import choice, shuffle
from time import time
from itertools import product

from competitive_sudoku.sudoku import Move
import competitive_sudoku.sudokuai


global_C = 2
global_total = False
global_selection = "max" #max or robust


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

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

    def compute_best_move(self, game_state) -> None:
        """
        The AI calculates the best move for the given game state.
        It continuously updates the best_move value until forced to stop.
        Firstly, it creates a solved version of the sudoku such that it has a valid number for every square.
        Then it repeatedly uses minimax to determine the best square, at increasing depths.
        @param game_state: The starting game state.
        """
        start_time = time()

        # Create some global variables so we do not have to pass these around
        global global_m, global_n, global_N
        global_m = game_state.board.m
        global_n = game_state.board.n
        global_N = global_m*global_n

        # Calculates the starting variable we will be using
        possible_moves, numbers_left, empty_squares = get_possible_numbers_and_empty(game_state.board)

        # Quickly propose a valid move to have something to present
        self.quick_propose_valid_move(game_state, possible_moves, numbers_left)

        # If this is the first turn we are in control, we figure out and save the alloted time
        available_time = self.load()
        if available_time == None:
            while True:
                cur_time = time()-start_time
                self.save(cur_time)

        # Gives a solution of the board
        solved_board = solve_sudoku(deepcopy(game_state.board))

        # Set the starting score of the MC_tree
        if not global_total: score = game_state.scores[0] - game_state.scores[1]
        else: score = 0

        # The root of the MC_tree
        root = Node(None, 1, possible_moves, empty_squares, score, 0)

        stop_time = available_time*0.95 - 0.001
        # We will loop this until we get low on time
        while time()-start_time < stop_time:
            one_mc_loop(root)

        # Determine the best child at this point
        max_value = -float('inf')
        move_index = None
        if global_selection == "max":
            for child in root.children:
                if child.q/child.n > max_value:
                    max_value = child.q/child.n
                    move_index = child.index

        elif global_selection == "robust":
            for child in root.children:
                if child.n > max_value:
                    max_value = child.n
                    move_index = child.index

        # Propose the current best move
        move = root.possible_moves[move_index]
        number_to_use = solved_board.get(move[0], move[1])
        self.propose_move(Move(move[0], move[1], number_to_use))


    def quick_propose_valid_move(self, game_state, possible_moves: list, numbers_left: dict) -> None:
        """
        Proposes a random move which is neither illegal, nor a taboo move, though it might be a bomb move.
        @param game_state: The game state for which this is happening.
        @param possible_moves: A list of coordinates for all empty squares (result of the get_possible_moves function).
```

```
94            @param numbers_left: A dictionary with for each group which number are not in that group (result of the get_numbers_left
                  function).
95            """

97            move = choice(possible_moves)
98            numbers = set(numbers_left["rows"][move[0]] & numbers_left["columns"][move[1]] & numbers_left["regions"][int(move[0] \
99                / game_state.board.m)*game_state.board.m + int(move[1] / game_state.board.n)])
100           moves = [Move(move[0], move[1], number) for number in numbers]
101           for move in moves:
102               if move not in game_state.taboo_moves:
103                   self.propose_move(move)
104                   break

106   def one_mc_loop(root) -> None:
107       """
108       Do one single loop of the the MC_tree algorithm.
109       @param root: The root node of the tree to work on.
110       """

112       # Selection Step
113       cur_node = root
114       while cur_node.children:
115           cur_node = cur_node.selection_step()

117       # Expansion Step
118       if cur_node.n == 1:
119           cur_node = cur_node.add_children()

121       # Simulation Step
122       result = cur_node.simulate()

124       if not global_total:
125           if result > 0:
126               result = 1
127           elif result < 0:
128               result = -1

130       # Backpropagation Step
131       cur_node.send_added_q(result)

133   class Node():
134       def __init__(self, parent, starting: int, possible_moves: list, empty_squares: list, score: int, index: int):
135           self.n = 0 # Amount of times this node was visited
136           self.q = 0 # Total score at this node
137           self.UCT = float('inf') # Upper Confidence Bound for Trees

139           self.parent = parent

141           self.starting = starting
142           self.possible_moves = possible_moves
143           self.empty_squares = empty_squares
144           self.score = score

146           self.children = []
147           self.children_UCT = [] # This will keep track of the children's UCT values

149           self.index = index

151       def __str__(self) -> str:
152           return f'n:{self.n}, q:{self.q}, UCT:{self.UCT}, score:{self.score}'

154       def selection_step(self):
155           """
156           Returns the child with the highest UCT.
157           """
158           return self.children[max(range(len(self.children_UCT)), key=self.children_UCT.__getitem__)]

160       def add_children(self):
161           """
162           For each move possible in this node create a child.
163           @return: Returns a random child.
164           """
165           self.children = [None]*len(self.possible_moves)

167           # For each move a child
168           for new_move in self.possible_moves:

170               # Update the amount of points earned by this move
171               points = (self.empty_squares[new_move[0]] == 1) + \
172               (self.empty_squares[new_move[1]+global_N] == 1) + \
173               (self.empty_squares[square2region(new_move)+global_N*2] == 1)

175               new_score = self.score + self.starting*{0:0, 1:1, 2:3, 3:7}[points]

177               # Flip wether the node is the starting player
178               new_starting = -self.starting

180               # Make copies of possible_moves and empty_squares and update them
181               new_possible_moves = self.possible_moves.copy()
182               new_possible_moves.remove(new_move)

184               new_empty_squares = self.empty_squares.copy()
185               new_empty_squares[new_move[0]] -= 1
186               new_empty_squares[new_move[1]+global_N] -= 1
187               new_empty_squares[square2region(new_move)+global_N*2] -= 1

189               # Send the index of the child in this parent node for easy bookkeeping
190               new_index = self.possible_moves.index(new_move)
```

```
192                 # Create the child
193                 new_child = Node(self, new_starting, new_possible_moves, new_empty_squares, new_score, new_index)
194                 self.children[new_index] = new_child

196             # Setups the children UCT storage list
197             self.children_UCT = [float('inf')]*len(self.possible_moves)

199             # Return a random child
200             if self.children:
201                 return choice(self.children)
202             else:
203                 return self

205         def send_added_q(self, added_q: int) -> None:
206             """
207             Updates the n and q in this node and sends the function upwards the tree.
208             @param added_q: The amount that should be added to q (influenced by 'starting').
209             """
210             self.n += 1
211             self.q += self.starting*added_q

213             if self.parent != None: # Only do this if we are not at the root
214                 self.UCT = (self.q/self.n) + global_C*sqrt(log(self.parent.n+1)/self.n)
215                 self.parent.children_UCT[self.index] = self.UCT
216                 self.parent.send_added_q(added_q)

218         def simulate(self):
219             """
220             Simulates a game from this node by randomly picking moves.
221             """

223             # Create copies of all relevant values that we can change
224             cur_starting = self.starting
225             cur_score = self.score
226             cur_possible_moves = self.possible_moves.copy()
227             cur_empty_squares = self.empty_squares.copy()

229             # Loop over a shuffled list to simulate random moves
230             shuffle(cur_possible_moves)
231             while cur_possible_moves:
232                 move = cur_possible_moves.pop()

234                 # Get and update points
235                 points = (self.empty_squares[move[0]] == 1) + \
236                 (self.empty_squares[move[1]+global_N] == 1) + \
237                 (self.empty_squares[square2region(move)+global_N*2] == 1)

239                 cur_score = cur_score + cur_starting*{0:0, 1:1, 2:3, 3:7}[points]

241                 # Update empty_squares
242                 cur_empty_squares[move[0]] -= 1
243                 cur_empty_squares[move[1]+global_N] -= 1
244                 cur_empty_squares[square2region(move)+global_N*2] -= 1

246                 # Flip wether we are currently the starting player
247                 cur_starting = -cur_starting

249             return cur_score


252     def square2region(square: tuple) -> int:
253         """
254         From the coordinates of a square return the region the square is in.
255         @param square: the x and y coordinates of a square.
256         """
257         region_number = square[0] - square[0] % global_m
258         region_number += square[1] // global_n
259         return(region_number)


262     def get_possible_numbers_and_empty(board) -> set:
263         """
264         For the current board get the possible_moves, numbers_left and empty_squares.
265         Possible_moves: The coordinates of all square that are still empty.
266         Numbers_left: The numbers not yet in a group for each row/column/region
267         Empty_squares: The number of empty (zero) squares for each row/column/region.
268         Note: read 'possible, numbers and empty'.
269         @param board: The board this should be done on.
270         @return: A set with the possible_moves list, the numbers_left dictionary and the empty_squares dictionary.
271         """

273         # The variables we will be adding to while looping
274         possible_moves = []
275         numbers_present = {"rows": [[] for i in range(board.N)], "columns": [[] for i in range(board.N)], "regions": [[] for i in
                                                                                                                        range(board.N)]}
276         empty_squares = [0]*(3*board.N)

278         # Loop over every square
279         for row in range(board.N):
280             for column in range(board.N):
281                 region = square2region((row,column))

283                 value = board.get(row, column)
284                 empty = (value == 0)

286                 if empty:
287                     possible_moves.append((row,column))

289                 numbers_present["rows"][row].append(value)
```

```
290                 empty_squares[row] += empty

292                 numbers_present["columns"][column].append(value)
293                 empty_squares[column+board.N] += empty

295                 numbers_present["regions"][region].append(value)
296                 empty_squares[region+board.N*2] += empty

298         numbers_left = {"rows": [], "columns": [], "regions": []}

300         # Use the numbers that are present to calculate the numbers which are left
301         for group in ["rows", "columns", "regions"]:
302             for i in range(len(numbers_present["rows"])):
303                 numbers_left[group].append({x for x in range(1,board.N+1) if x not in set(filter((0).__ne__, numbers_present[
                                                                                                    group][i]))})

305         return possible_moves, numbers_left, empty_squares


308     def solve_sudoku(board):
309         """
310         A Sudoku solver using Knuth's Algorithm X to solve an Exact Cover Problem.
311         Credit goes out to Ali Assaf for inspiration from their version.
312         https://www.cs.mcgill.ca/~aassaf9/python/algorithm_x.html
313         @param board: The board to be solved
314         @return board: A solved board
315         """

317         # Set X for the Exact Cover Problem
318         N = board.m * board.n
319         X = ([("roco", roco) for roco in product(range(N), range(N))] +
320              [("ronu", ronu) for ronu in product(range(N), range(1, N + 1))] +
321              [("conu", conu) for conu in product(range(N), range(1, N + 1))] +
322              [("renu", renu) for renu in product(range(N), range(1, N + 1))])

324         # Subsets Y for the Exact Cover Problem
325         Y = {}
326         for row, column, number in product(range(N), range(N), range(1, N + 1)):
327             region = square2region((row, column))
328             Y[(row, column, number)] = [
329                 ("roco", (row, column)),
330                 ("ronu", (row, number)),
331                 ("conu", (column, number)),
332                 ("renu", (region, number))]

334         # Changes X such that we can use a dictionary instead linked lists
335         X = reformat_X(X, Y)
336         for row, column in product(range(N), range(N)):
337             number = board.get(row, column)
338             if number:
339                 solver_select(X, Y, (row, column, number))

341         # Grabs the first solution, puts in in a board and returns it
342         for solution in actual_solve(X, Y, []):
343             for (row, column, number) in solution:
344                 board.put(row, column, number)
345             return board

347     def reformat_X(X, Y):
348         """
349         Subfunction of sudoku solve
350         Reformats the X to be usable
351         """
352         X = {i: set() for i in X}
353         for key, value in Y.items():
354             for i in value:
355                 X[i].add(key)
356         return X


359     def actual_solve(X, Y, solution):
360         """
361         Subfunction of sudoku solve.
362         Does the actual solving algorithm X.
363         """
364         if not X:
365             yield list(solution)
366         else:
367             c = min(X, key=lambda c: len(X[c]))
368             for r in list(X[c]):
369                 solution.append(r)
370                 cols = solver_select(X, Y, r)
371                 for s in actual_solve(X, Y, solution):
372                     yield s
373                 solver_deselect(X, Y, r, cols)
374                 solution.pop()

376     # Selects group for Algorithm X
377     def solver_select(X, Y, key):
378         """
379         Subfunction of sudoku solve.
380         'Selects' a subset of X, which removes it.
381         """
382         cols = []
383         for i in Y[key]:
384             for j in X[i]:
385                 for k in Y[j]:
386                     if k != i:
387                         X[k].remove(j)
```

```
388             cols.append(X.pop(i))
389         return cols
390
391     # Deselects group for Algorithm X
392     def solver_deselect(X, Y, key, cols):
393         """
394         Subfunction of sudoku solve.
395         'Deselects' a subset of X, which adds it back.
396         """
397         for i in reversed(Y[key]):
398             X[i] = cols.pop()
399             for j in X[i]:
400                 for k in Y[j]:
401                     if k != i:
402                         X[k].add(i)
```