Assignment A2: Team 41

Sven Steens Riccardo Torlaini Rafał Polak

8-12-2022

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 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

Our assignment A2 agent operates in two steps. Firstly, it generates a solved version of the current board (note how a board might have multiple solutions - the AI only finds one). This way, for each square it knows which numbers can be put in there. Afterwards, it uses a version of the minimax algorithm to determine the optimal square to fill with a number by iteratively running deeper maximum depths until the algorithm's "thinking time" has finished. For this, the minimax uses a relatively heuristic, concerned with gaining the largest positive score difference as well as a newly added desire to make rows even (more in section 2.4. This minimax is further enhanced using alpha-beta pruning, as well as small optimizations reducing the number of needed calculations, mostly by taking away the need to recalculate board-states or copy over variables (see the uses of *open_squares*, *empty_squares* and *numbers_left* in the code).

We built the A2 algorithm on the A1 algorithm in three main ways. Firstly, we gave the algorithm access to a new play aside from filling in a square: skipping a turn. Secondly, we improved the speed of our Sudoku Solver to give our minimax more time to run, and thus potentially reach deeper depths. Thirdly, we updated the used heuristics to implement a hint of strategy aside from simply maximizing points. Aside from this, we also made a few miscellaneous improvements which will get their own section.

2.1 Heuristics Update

Originally (Assignment A1), our agent made use of no advanced heuristics. The only metric on which the board evaluation was calculated, was the possible score to achieve from the current board state within some number of moves (dictated by the depth of the current run of minimax). Given unlimited time, the entire game tree could be traversed and an optimal move would be returned by this procedure alone. Knowing, however, how sparse the game tree is in the early- and mid-game, and given the timer constraints, such an approach was doomed to fail. The evaluation function needed to be tweaked. We enhanced it with a simple metric - a "desire" to have groups contain an even number of digits after our move (i.e. incentivize our agent to fill in empty squares belonging to groups with an odd number of empty squares).

Consider an empty group. As a rational player, one wants to be the one that puts in the last number in that group. Therefore, one does not want to be the one that puts in the second-to-last number, as this gives their opponent the opportunity to put in the last one. This reasoning can be extended to any group of arbitrary length and generalized to a wanting to put numbers in groups with odd numbers.

In our A1 agent, every point gained by a move is translated to exactly one point in the evaluation function deployed by our minimax algorithm. Our A2 agent extends that. To quantify the "desire" presented above, whenever an agent places a digit in a square, it gains an additional 0.01 "point" for each group containing an even number of empty squares after the move, that the particular square belongs to. 0.01 points is a rather low value. This makes sure that this metric never overshadows the most important factor, points gained, but simply acts as a decider between similar moves. In addition, this metric is very efficient to calculate in our framework, costing almost no additional time. It does, however, increase running times by increasing granularity in the scores, which causes the alpha-beta pruning to trigger less often/quickly, doubling (or worse) running times on deeper depths (>5 depth).

2.2 Move Skipping

The main strategy whose implementation was attempted for this assignment was 'move skipping' - attempting to have the agent propose a move that would lead to it effectively not inputting any numbers (a *bomb move*).

This may at first seem like a disadvantageous strategy - not playing a move would, in theory, guarantee no score and as such should incur minimal reward. However, the specific rules of this game may at times deem it the best 'move' to make. Specifically, there are situations where playing a move would guarantee that the opponent could complete a column, row, or region in the next turn (for instance, if only two squares remain open in all columns/rows/regions). As such, without move skipping, a player would be forced to "concede" points. By intentionally playing a *bomb move* and skipping the turn, this situation is effectively reversed, and it is now the opponent who has no choice but to "concede" points to us (if the opponent has no turn skipping implemented or cannot find a unique *bomb move*).

To be clear, **turn skipping has not yet been fully implemented as of Assignment A2.** The reason is simple: as will be explained later, testing revealed that the running time of this solution was far too high and made it unusable. We set the goal of creating a fast and reliable turn-skipping mechanism as the main point of improvement for assignment A3.

It is not so easy to find a proper move that leads to a turn skip. Here is an overview of the techniques tried so far in this field:

The simplest idea, proposing an "impossible" move (e.g. "out-of-bounds" move, such as putting a 5 at a square [10, 10] for a 3x3 board), would not work, as the rules of the game penalize such a proposition with an instant loss. Alternatively, one might obtain all possible *bomb moves* by finding all possible solutions to a Sudoku board given its current state (using a brute-force approach), but that has a disastrous running time.

The next solution to be investigated, and the one that has so far seemed most sensible, was to assume in the minimax that any turn can be skipped and only attempt to do it when the situation calls for it. In that case, compute the solved board from that position and in the process, find moves which can be played (are legal) but will not lead to said solved board. The code developed for that is included is the appendix (*note here that some variables are defined earlier in the code, and the minimax function is changed to take as input the entire board object*).

In short, first it is checked whether turn skipping should be considered, and if so the board is copied and solved. Then, the numbers that can still be played per row, column and region are computed via the *get_open_numbers_and_empty* function. The possible illegal moves are then computed, and if any are found, it will attempt to play them (and store the fact this move has been played to a global list, due to the fact playing an illegal move twice leads to losing the game). **The score is then given to this skip based on what the opponent has been rendered unable to do.**

2.3 Better Sudoku Solver

The *solve_sudoku* function mimics the human approach to solving a sudoku board, but this time it features one crucial additional step which we will highlight in green. The procedure is as follows:

1. Fill in all the trivially-solvable groups, e.g. on a 3x3 board, any column that has 8 numbers and 1 blank square in it. Keep recursing this step until no change to the board can be made in this way.

2. Now, proceed to move in a region-by-region fashion. For each of them, list all numbers not yet present in it. Step 1 will have filled any squares with only one number possible, but will have missed whenever multiple numbers can be placed in a square, yet only one square will be available to some number. For clarity, imagine a 3x3 region with numbers (1, 2, 3, 4, 5) already there and four empty squares left. If the "available" numbers for each square are (6, 7), (6, 7, 8), (7, 9) and (7, 9), then we know for sure that the second square has to be filled with an 8, as it can't belong anywhere else. Moreover, after filling in this square, we see how the number '6' also becomes unique in this "set of sets", only being possible to be put in square 1. That is why this step also has to recurse itself until no changes to the board can be made, just like step 1 of this Sudoku Solver.

3. Choose a random empty square on the board and fill it in with a random "fitting" number (not one against the rules of the game). Go back to step 1. Either we will solve the Sudoku correctly (which means this random inputted number move was not a taboo move) or we will find the Sudoku unsolvable (so this random inputted number move was indeed a taboo move). In that case, return to the state of the board from before trying this random move and try another one. Notice that any such move which makes the Sudoku Solver restore the board is a *bomb move* in the current

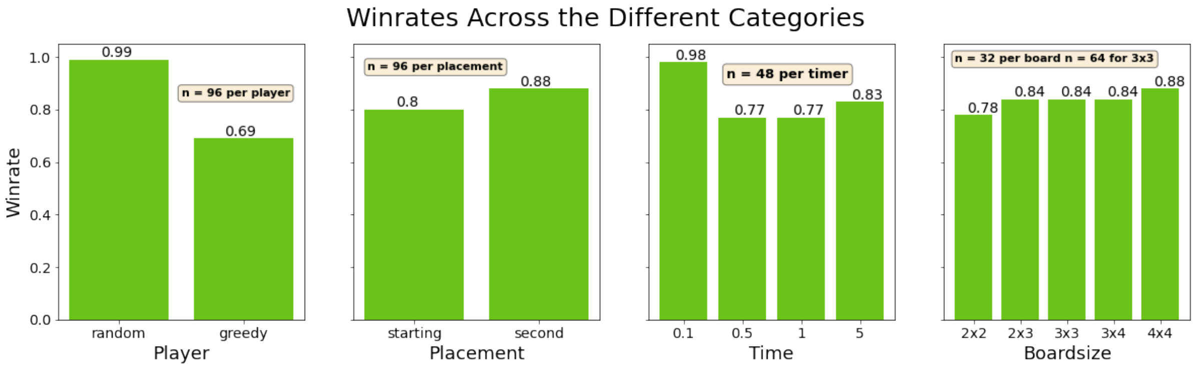

Figure 1: The results of the A2 AI tests. Draws are accounted as losses

situation, so it might be useful for implementing the turn skipping functionality (see: Section 3.3).

2.4 Miscellaneous Improvements

For our previous agent, we noticed that on particularly big and empty boards, it was not always able to propose any move under shorter timers. This used to result in an immediate loss. To prevent this, we added functionality to send a move before the Sudoku Solver or the minimax are run. This is an arbitrary move, which is only checked to be neither illegal nor a taboo move (though it still might be a *bomb move*. Either of those two would still lose the game if inputted, yet checking for them is almost instant in practice.

Another change we made was condensing a set of functions together. Previously we calculated *open_squares*, *empty_squares* and *numbers_left* separately, but these are rewritten into the *get_open_numbers_and_empty* function. This change reduced the amount of double for loops from 7 to 1 and halved the amount of code required for these calculations.

Lastly, two other small included language changes were made in the code to make the function of each variable clear in a vacuum and a bugfix that makes the Sudoku Solver use the implemented tactics instead of just random guessing.

3 Agent Analysis

3.1 Methods

For our analysis, we made our AI play games on 12 different boards against the two provided opponents. This was done using timers of 0.1, 0.5, 1 and 5 seconds. In each combination of board, opponent and timer two games were played, one with our AI as the starting player and one with the opponent as the starting player. This has resulted in 192 different games being played. The win rates across these experiments are summarized in Figure 3.1. For comparison, we have also included the summary of the same experiment done on the A1 AI in Figure 3.1. We also clashed the A1 and A2 AI against each other for each board/time/starting player combination, as can be seen in Figure **??**.

3.2 Results

One foremost conclusion of the A2 test runs is the lack of games lost due to moves not being proposed at all, compared to the results of the A1 agent. While not visualized in the graph, roughly 55% of A1's games were lost due to the AI being too slow to propose any moves. For A2, this has dropped to a single game, likely due to the implementation of a

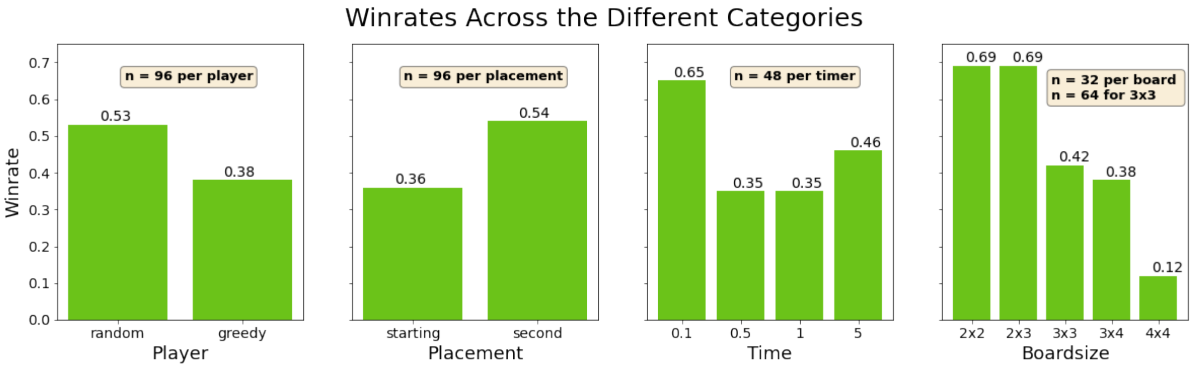

Figure 2: The results of the A1 AI tests. Draws are accounted as losses

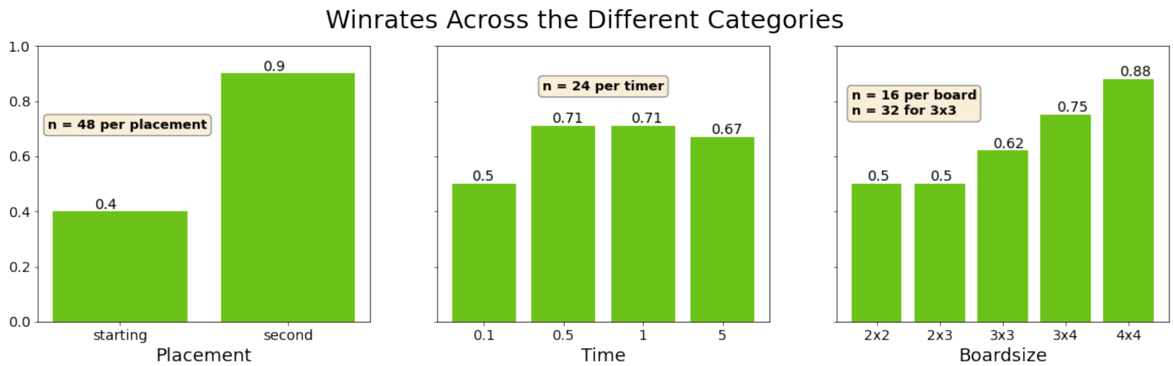

Figure 3: The results of the A2 vs A1 test. Draws are accounted as losses

quick move proposal and the substantial improvement of the Sudoku Solver. Additionally, the overall win rate has increased from 45% to 84%.

3.3 Turn Skipping

While this feature was not yet able to move to formal testing and as such no proper results exist yet, preliminary testing consistently shows it to be inefficient in its current form, to the point that games with a timer of fewer than 5 seconds are lost whenever this function comes into play by time-out.

4 Reflection

The agent in its current state is showing good general performance. An almost 100% win rate against the random player hints that employing the heuristics has a positive effect on the outcome. Additionally, the increased performance at higher times shows that it does indeed properly use its time (although perhaps still not enough) Lastly, the better results at larger board sizes show the benefits of the added heuristics.

There are, however, still issues to be overcome in the final part of the assignment. The main one is most likely the large and yet increasing overhead that new features add to the algorithm. While the recently-added features managed to improve the level of our player on average, one could also observe scenarios in which the additions resulted in slower processing. Even changes that run asymptotically in $O(1)$ are expected to inevitably delay the processing of our algorithm under low timers.

Take the "*odd groups*" (new) heuristic, for example. While this gives the AI more agency in the early game, it also causes it to slow down, hampering the late game where the "*odd* *groups*" heuristic is less important. Therefore, we theorize that the logical next step is to include some measure to differentiate the early-, middle- and late-game. This would allow us to implement new features, without as big of a risk of slowing down in board states where it is no longer relevant.

Such a distinction might also bring other benefits. It may make the implementation of move skipping easier, if we can make an overall guess if/how many skips might be available without proper calculations.

Naturally, turn skipping is also a feature that will be further developed for the next iteration of this algorithm, as its potential benefits and the strategic advantage it would bring cannot be understated. Being able to 'get out', potentially, of any situation that would lead to a certain loss of points is certainly a powerful move that we feel our agent would greatly benefit from having access to. It is regrettable that it has not been developed to an extent that it could be implemented, but we chose to add a section on it in this report all the same as we feel it is an important step even if work in progress.

5 Code overview

Note that this overview deals with the second appended file.

Line 21: At initialization, we retrieve the info on the open (available) squares on the board, how many open squares are there in each group and what numbers are missing in each group.

Line 24: Quickly proposes an arbitrary valid, legal and non-taboo move to avoid a scenario where we have committed nothing

Line 27: We create a solved version of the current board. We will later use this to determine what number to fill in for a square.

Lines 31-33: Creating a dictionary to help us incentivize odd-number-left moves. See Section 2.1 for further info. The final dictionary will have the following form: $1 : 0, 2 : 0, 3 : 1, 4 : 0, 5 : 1, 6 : 0, 7 :$

Lines 36-39: We run the minimax algorithm on the current game state to retrieve (and propose) the best move in the current situation. The minimax will be running recursively, each time starting with depth=1 and increasing it until max_depth, where max_depth will increase by 1 at every iteration.

Lines 41-55: We add the "proposed move" function here to avoid unnecessary imports.

Lines 57-68: The functionality to propose an instant move, as described in Section 2.4.

Lines 72-84: Quality of Life functions.

Lines 120-126: Calculating how many points would the given move contribute to the overall result by the heuristics described in the report (bearing in mind that this will be a negative value if it is the turn of the minimizing player, as determined in Line 81).

Lines 131-134: Since this current move is being considered, the numbers of empty groups have to be updated accordingly, if we decide to play it, so that is what we do here.

Line 137: Recurse minimax with a larger ("going more down the tree") depth. This way, the changes to the numbers of empty groups and game score, altered by considering the current move, stay in place and have an impact on this entire "branch" of the game tree that we will be traversing by recursing minimax in this place.

Lines 131-134: Since in this place we finished inspecting the "previous" branch of the game tree, we restore the values of the number of empty squares per group. We also save the score of the previous branch in case this was the best one seen so far (so that we might want to choose the move leading to this branch in the game).

Lines 146-154: This implements Alpha-beta pruning.

Lines 159-200: A function to calculate the three sets of variables invoked at line 21.

Lines 202-318: The Sudoku Solver described in this report

Python files

Code Listing 1: `Code to implement turn skipping.`

```python
if all( map( lambda x: x == 2, curr_empty['region'] )):
        solved_board_test = solve_sudoku(deepcopy(board), deepcopy(open_squares), board.get_numbers_left())
        numleft = get_open_numbers_and_empty(board)[1]
        for square in open_squares:
            realmove = solved_board_test.get(square[0], square[1])

            row_left = numleft['row'][square[0]] #the numbers left in that row
            col_left = numleft['column'][square[1]] #the numbers left in that column
            regio_left = numleft['region'][square2region(square[0], square[1])]
            nums_left = [num for num in row_left if num in col_left and num in regio_left]
            nums_left.pop(nums_left.index(realmove)) if realmove in nums_left else 0

            if len(nums_left):
                if nums_left not in taboos:
                    taboo_move = nums_left[0]
                    taboos.append(taboo_move)
                    best_score = 4 #it's better than putting in a move and letting the
                            #opponent take a region worth 3 points


    elif all( map( lambda x: x == 2, curr_empty['row'])) or all( map(lambda x: x == 2, curr_empty['column'] )):
        solved_board_test = solve_sudoku(deepcopy(board), deepcopy(open_squares), board.get_numbers_left())
        numleft = get_open_numbers_and_empty(board)[1]
        for square in open_squares:
            realmove = solved_board_test.get(square[0], square[1])

            row_left = numleft['row'][square[0]] #the numbers left in that row
            col_left = numleft['column'][square[1]] #the numbers left in that column
            regio_left = numleft['region'][square2region((square[0], square[1]), board.m, board.n)]
            nums_left = [num for num in row_left if num in col_left and num in regio_left]
            nums_left.pop(nums_left.index(realmove)) if realmove in nums_left else 0

            if len(nums_left):
                if nums_left not in taboos:
                    taboo_move = nums_left[0]
                    taboos.append(taboo_move)
                    best_score = 2.5 #it's better than putting in a move and letting the
                        #opponent take a row/column or potential row/column pair
                        #worth 1 or 2 points, BUT if we get a region it's a net gain
                        #hence 2.5 as it is <2 but >3
```

Code Listing 2: `sudokuai.py`.

```python
from copy import deepcopy
from competitive_sudoku.sudoku import GameState, SudokuBoard, Move, TabooMove

class SudokuAI(object):
    """
    Sudoku AI that computes a move for a given sudoku configuration.
    """
    def __init__(self):
        self.best_move = [0, 0, 0]
        self.lock = None

    def compute_best_move(self, game_state: GameState) -> 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.
        """
        # Calculates the starting variable we will be using
        open_squares, numbers_left, empty_squares = get_open_numbers_and_empty(game_state.board)

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

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

        # This sets the dictionary odds, it returns 1 if the variable is odd and not 1 (only for this sudoku)
        global odds # A global variable so we do not have to pass it down (also, yes this is faster than x%2==0)
        odds = {1:0, 2:0}
        for i in range(3, game_state.board.N+1):
            odds[i] = int(i%2 == 1)

        # Calculate for every increasing depth
        for depth in range(1,9999):
            move = minimax(max_depth = depth, open_squares = open_squares, empty_squares = empty_squares, m = game_state.board.m, n = game_state.board.n)[1]
            number_to_use = solved_board.get(move[0], move[1])
            self.propose_move(Move(move[0], move[1], number_to_use))

    def propose_move(self, move: Move) -> None:
        """
        Note: This function is implemented here to save time with importing
        Updates the best move that has been found so far.
        N.B. DO NOT CHANGE THIS FUNCTION!
        @param move: A move.
        """
```

```
            i, j, value = move.i, move.j, move.value
            if self.lock:
                self.lock.acquire()
            self.best_move[0] = i
            self.best_move[1] = j
            self.best_move[2] = value
            if self.lock:
                self.lock.release()

        def quick_propose_valid_move(self, game_state: GameState, open_squares: list, numbers_left: dict) -> None:
            """
            Proposes a 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 open_squares: A list of coordinates for all empty squares ( result of the get_open_squares function )
            @param numbers_left: A dictionary with for each group which number are not in that group ( result of the get_numbers_left function ) """
            move = open_squares[0]
            numbers = set(numbers_left["rows"][move[0]] & numbers_left["columns"][move[1]] & numbers_left["regions"][int(move[0] \
                / game_state.board.m)*game_state.board.m + int(move[1] / game_state.board.n)])
            moves = [Move(move[0], move[1], number) for number in numbers]
            non_taboo_moves = [move for move in set(moves) if move not in set(game_state.taboo_moves)]
            self.propose_move(non_taboo_moves[0])


#The below two functions exist so that we can create certain references in the minimax function
def greater(i: int, j: int) -> int:
    return i > j


def smaller(i: int, j: int) -> int:
    return i < j


# From the coordinates of a square return the region the square is in
def square2region(square: tuple, m: int, n: int) -> int:
    region_number = square[0] - square[0] % m
    region_number += square[1]//n
    return(region_number)


def minimax(max_depth: int, open_squares: list, empty_squares: dict, m: int, n: int,
is_maximizing_player: bool = True, current_score: int = 0, alpha: int = float("−inf"), beta: int = float("inf")) -> set:
    """
    A version of the minimax algorithm implementing alpha−beta pruning.
    Every time we create a child , we calculate how many points the move associated with that child might get us.
    This calculation is done with empty_squares, while all potential moves are kept track of via open_squares.
    Variables with default values take those values during the first iteration .
    @param max_depth: The maximum depth the function is allowed to further search from its current depth.
    @param open_squares: A list containing all still open squares / possible moves.
    @param empty_squares: A dictionary containing the number of empty squares for each group.
    @param m: The number of rows per region for this board, used to calculate regions from coordinates .
    @param n: The number of columns per region for this board, used to calculate regions from coordinates .
    @param is_maximizing_player: Whether the current player is attempting to maximize or minimize the score .
    @param current_score: The score at the node we start this iteration of minimax on.
    @alpha: The alpha for alpha−beta pruning.
    @beta: The beta for alpha−beta pruning.
    @return: The score that will be reached from this node a maximum depth and the optimal next move to achieve that.
    """

    # If we have hit either the maximum depth or if there are no more moves left , we stop iteration
    if max_depth == 0 or not open_squares:
        return current_score, (-1,-1)

    # Switches values around depending on if the player is maximizing or not
    value, function, multiplier = (float('−inf'), greater, 1) if is_maximizing_player else (float('inf'), smaller, -1)

    # Initializes where we store the best move and its associated score of this node
    best_score = value
    best_move = open_squares[0]

    for move in open_squares[:]:

        # Calculates how the move would change the score
        row_amount = empty_squares["row"][move[0]]
        column_amount = empty_squares["column"][move[1]]
        region_amount = empty_squares["region"][square2region(move, m, n)]

        # First one is the desire to finish rows, second one is the desire to make rows even/not odd
        amount_finished = (row_amount == 1) + (column_amount == 1) + (region_amount == 1)
        amount_even = odds[row_amount] + odds[column_amount] + odds[region_amount]

        new_score = current_score + multiplier*({0:0, 1:1, 2:3, 3:7}[amount_finished] + amount_even*0.01)

        # Removes the move from open_squares and updates empty_squares to account for the move
        open_squares.remove(move)
        empty_squares["row"][move[0]] -= 1
        empty_squares["column"][move[1]] -= 1
        empty_squares["region"][square2region(move, m, n)] -= 1

        # Goes one layer of minimax deeper
        returned_score = minimax(max_depth-1, open_squares, empty_squares, m, n, not is_maximizing_player, new_score, alpha, beta)[0]

        # Changes open_squares and empty_squares back to the original state
        open_squares.append(move)
        empty_squares["row"][move[0]] += 1
        empty_squares["column"][move[1]] += 1
        empty_squares["region"][square2region(move, m, n)] += 1
```

```python
        # If the score of this move going deeper is better , this becomes the best move with the best score
        if function(returned_score, best_score):
            best_score = returned_score
            best_move = move

            # Does the alpha-beta pruning
            if is_maximizing_player: alpha = max(alpha, best_score)
            else: beta = min(beta, best_score)
            if alpha >= beta:
                break

    return best_score, best_move


def get_open_numbers_and_empty(board: SudokuBoard) -> set:
    """
    For the current board get the open_squares, numbers_left and empty_squares.
    Open_squares: The coordinates of all square that are still empty.
    Numbers_left: The numbers not yet in a group for each row/column/region
    Empty_squares: The number of empty (zero) squares for each row/column/region.
    @param board: The board this should be done on.
    @return: A set with the open_squares list, the numbers_left dictionary and the empty_squares dictionary """

    # The variables we will be adding to while looping
    open_squares = []
    numbers_present = {"rows": [[] for i in range(board.N)], "columns": [[] for i in range(board.N)], "regions": [[] for i in range(board.N)]}
    empty_squares = {"row": [0]*board.m*board.n, "column": [0]*board.m*board.n, "region": [0]*board.m*board.n,}

    # Loop over every square
    for row in range(board.N):
        for column in range(board.N):
            region = square2region((row,column), board.m, board.n)

            value = board.get(row, column)
            empty = (value == SudokuBoard.empty)

            if empty:
                open_squares.append((row,column))

            numbers_present["rows"][row].append(value)
            empty_squares["row"][row] += empty

            numbers_present["columns"][column].append(value)
            empty_squares["column"][column] += empty

            numbers_present["regions"][region].append(value)
            empty_squares["region"][region] += empty

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

    # Use the numbers that are present to calculate the numbers which are left
    for group in ["rows", "columns", "regions"]:
        for i in range(len(numbers_present["rows"])):
            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]))})

    return open_squares, numbers_left, empty_squares

def solve_sudoku(board: SudokuBoard, open_squares: list, numbers_left: dict) -> SudokuBoard:
    '''
    Iteratively gives a solution to the given sudoku.
    First, fills in any squares where only one number is possible, then randomly guesses.
    @param open_squares: A list containing all still open squares/possible moves.
    @param empty_squares: A dictionary containing the missing numbers for each group.
    @return: A filled board.
    '''

    move_possibilities = {} # For each move keep track of the possibilities of that move
    result = [] # All squares where only one number is possible
    for move in open_squares:
        possibilities = set(numbers_left["rows"][move[0]] & numbers_left["columns"][move[1]] & \
            numbers_left["regions"][int(move[0] / board.m)*board.m + int(move[1] / board.n)])

        # If this is the case we mark this move to be filled in
        if len( possibilities ) == 1:
            number = next(iter(possibilities))
            result.append([move,number]) # Note to past self, do not forget to add this line or you'll brick A1

        # If this is the case, a previous guess was wrong
        elif len( possibilities ) == 0:
            return -1

        move_possibilities[move] = possibilities

    # If squares can be filled in, do so and start back at the beginning
    if result != []:
        for i in result:
            board.put(i[0][0], i[0][1], i[1])
            open_squares.remove(i[0])

            # If this try fails we have made an incorrect guess before this
            try:
                numbers_left["rows"][i[0][0]].remove(i[1])
                numbers_left["columns"][i[0][1]].remove(i[1])
                numbers_left["regions"][int(i[0][0] / board.m)*board.m + int(i[0][1] / board.n)].remove(i[1])
            except:
                return -1
```

```python
        return solve_sudoku(board, open_squares, numbers_left)

    # Inspect  for  each  square  if  for  one  of  its  groups  it  is  the  only  square  were  a  number can  go
    else:
        success  = False

        # For each row gets  a  list  of  all    possibilities    for  all  squares  combined
        row_possibilities  = {}
        for move in move_possibilities:
            if  move[0]  in  row_possibilities :
                row_possibilities [move[0]]  += list( possibilities )

            else:
                row_possibilities [move[0]]  =  list ( possibilities  )

        # For each column gets  a  list  of  all    possibilities    for  all  squares  combined
        column_possibilities  = {}
        for move in move_possibilities:
            if  move[1]  in  column_possibilities:
                column_possibilities[move[1]]  += list( possibilities )

            else:
                column_possibilities[move[1]]  =  list ( possibilities  )

        # For each  region  gets  a  list  of  all    possibilities    for  all  squares  combined
        region_possibilities   = {}
        for move in move_possibilities:
            if square2region(move, board.m, board.n) in region_possibilities:
                region_possibilities [square2region(move, board.m, board.n)] += list( possibilities )

            else:
                region_possibilities [square2region(move, board.m, board.n)] = list( possibilities  )

        # For each open square  check  for  each  number if  it  is  only  once  in  the    possibilities    for  all  squares
        for move in move_possibilities:
            for number in move_possibilities[move]:
                if  ( row_possibilities [move[0]] .count(number)  == 1)  or  (column_possibilities[move[1]].count(number)  == 1) \
                   or  ( region_possibilities  [square2region(move, board.m, board.n)].count(number)  == 1):
                   board.put(move[0], move[1], number)
                   open_squares.remove(move)

                   # If  this  try   fails   we  have  made an  incorrect   guess  before   this
                   try:
                       numbers_left["rows"][i [0] [0] ] .remove(i[1] )
                       numbers_left["columns"][i[0] [1] ] .remove(i[1] )
                       numbers_left["regions"] [int(i [0] [0]  /  board.m)*board.m + int(i[0][1]  /  board.n)].remove(i[1] )
                   except:
                       return -1

        # If  a  change  was  made we can  go  back  to  the   start
        if  success :
            return solve_sudoku(board, open_squares, numbers_left)

    # If  no  squares  can  be   filled   in ,  keep  making a  guesses   until  you  hit  a  correct  one
    if  board.empty in board.squares:
        iterator  = iter( possibilities )
        for number in iterator:
            new_board = deepcopy(board)
            new_board.put(move[0], move[1], number)

            new_open_squares = deepcopy(open_squares)
            new_open_squares.remove(move)

            new_numbers_left = deepcopy(numbers_left)
            new_numbers_left["rows"][move[0]].remove(number)
            new_numbers_left["columns"][move[1]].remove(number)
            new_numbers_left["regions"][int(move[0] / board.m)*board.m + int(move[1] / board.n)].remove(number)
            result  = solve_sudoku(new_board, new_open_squares, new_numbers_left)

            if  result  != -1:
                return result

        # If  no  possible  number worked ,  a  previous  guess  was  wrong
        return -1

    # If  the  board  is   full ,   return
    return board

# Makes moves hashable
def move_hash(self):
    return hash((self.i,  self .j ,  self .value))

Move.___hash___ = move_hash
TabooMove.___hash___ = move_hash
```