Assignment A1: Team 41

Sven Steens   Riccardo Torlaini   Rafał Polak

21/11/2022

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**

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.

All the main ideas and ways the agent operates should be understandable from the report and the well-commented code, but in case the reviewer would like to get more explanation, refer to Section 5 (Code overview).

2 Agent Description

The main functionality of the agent is - obviously - the minimax algorithm to evaluate the (current and possible) board states. The other notable technique employed by this agent is the *alpha-beta pruning* that significantly sped up the algorithm in our tests (around 6 times faster computations for the random-2x3 board). Since the filled number does not matter for the score, it does not matter for the minimax. As a result, the algorithm works in two steps. Firstly we use a custom sudoku solver to determine a feasible number for each square. Then we find the optimal square to fill in. This optimal square is searched with respect to some depth on the minimax search (typically starting with depth=1, i.e. considering the game states only 1 move ahead of the current board state) and after such depth was thoroughly explored, move to a higher depth. This procedure carries on until some maximum depth for the current iteration. After each finalized such exploration, the best move (according to the current depth) is reported, with a new one overwriting the previous "best-declared move" after the deeper search has been conducted. Afterwards, increase the maximum depth and re-iterate. The maximum depth parameter is in practice unbounded - given infinite time, the entire game tree will be traversed. The traversal uses a depth-first search at each iteration. Contrary to suspicion, saving the current game tree when moving to a higher depth, did not prove significant in speeding up the algorithm, given the amount of time needed to save and load the time, as opposed to how quick our implementation naturally is.

2.1 Heuristics

We chose not to use any user-defined heuristics for evaluating the positions, other than the best score (assuming perfect play from the opponent) resulting in playing such a move (up to some depth). We noticed how the agent restrains from filling up squares that have an even number of empty squares in their corresponding groups. That makes intuitive sense - after simultaneous updates to the same groups, the opposing player will put the last number in and receive the points for these groups, so that is not our dream scenario.

2.2 "Solve_sudoku" function

Our sudoku solver finds a solution for a given Sudoku board in a reasonably "human" fashion: first, it attempts to iteratively fill in all the empty squares that can be simply determined to have a correct solution - for they are the only empty squares in their corresponding group. After any such square is filled, the procedure is restarted. Afterwards, the function makes a random guess, filling in some other empty square. Then, it proceeds with the previous step. These steps are invoked iteratively, until either the sudoku is solved, or turns out to be unsolvable. In the latter case, The most recent guess has to be undone and another one is made. As soon as no other possible guesses at this board state remain, the algorithm goes another step back and undoes the guess before. This can theoretically keep happening until the very first guess of the board was made. Further optimizations can be made to that function, however as it is only being called once per game, its running time is not such impactful of a bottleneck.

2.3 Alpha-beta Pruning

Of course, we can not explain the alpha-beta pruning better than this video does: https://youtu.be/l-hh51ncgDI. But in case you prefer to read instead consider the following example where a positive score favours Player1 ("White") and a negative score favours Player2 ("Black"): Imagine at some point in the game tree, White can make a move that leads to an evaluation score of 1.0 with Black's perfect play. The other move (we assume 2 moves only) leads to Black's choice between a move evaluated at -1.5 and another move branch. Then, this branch does not have to be explored at all - Black will choose a move that gives them at least 1.5 point advantage, so regardless of specifics of this branch, White should not choose that move; hence, we can "*prune*" that branch and terminate this part of minimax, evaluating the position as 1.0 at White-to-move turn.

2.4 Technical Notes

A few techniques were used to reduce the running time of each minimax iteration. Instead of recalculating the possible moves (`open_squares` in the code) at each new board state, the possible moves are calculated at the start. Whenever a move is done, it is then removed from this list before it is passed on to the child. To add to this, we also do not fully recalculate how many points a move can earn at each state. Instead, we keep track of the amount of empty (zero) squares in each row, column and region (`empty_squares` in the code). Thus we know that if one of the three groups of a square has exactly 1 still empty square, placing that square will complete that group, thus earning points. After a move is done we can then decrease the number of empty squares in the groups of that move by one. Lastly, we avoid copying lists and dictionaries as much as possible. Copying these structures is costly if we were to do it in every node. Instead, we change these variables before going to a lower depth and change them back after we are done. This is especially more time efficient in the case of the `open_squares` variable, where we remove and then read one value, as opposed to copying a whole list.

3 Agent Analysis

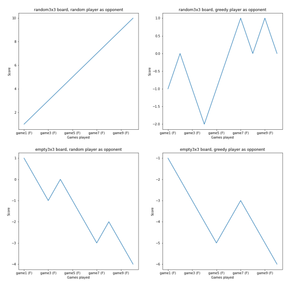
3.1 Formal results on 3x3 boards

The **Results** are formatted in a way that 'w' means a victory (win) of our agent, 'l' its loss and 'd' a draw. See Table 1 below for the detailed scores.

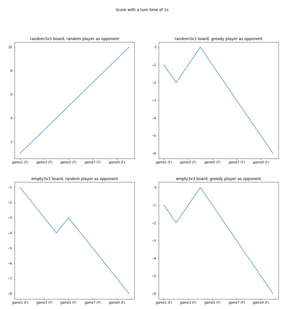
Table 1: Summary of the scores when testing the agent against the provided players.

Opponent	Board	Timer	Results	Winrate
random_player	random-3x3	0.5s	wwwwwwwwww	100%
random_player	random-3x3	1s	wwwwwwwwww	100%
random_player	random-3x3	5s	wwwwwwwwww	100%
greedy_player	random-3x3	0.5s	llwlwwwlwl	50%
greedy_player	random-3x3	1s	llwwllllll	20%
greedy_player	random-3x3	5s	wwlwllwwwl	60%
random_player	empty-3x3	0.5s	wllwlllwll	30%
random_player	empty-3x3	1s	llllwlllll	10%
random_player	empty-3x3	5s	wlwlwwlwwl	60%
greedy_player	empty-3x3	0.5s	lllllwwlll	20%
greedy_player	empty-3x3	1s	llwwllllll	20%
greedy_player	empty-3x3	5s	wwwwwlwlww	80%

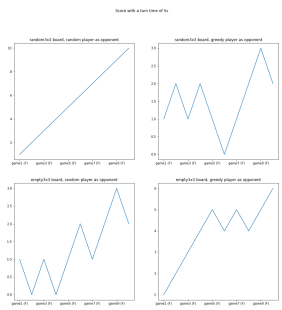
To better visualize these results and give more insight into their significance, we have also plotted the scores for each player and board combination at all three timer settings:



(a)



(b)



(c)

Figure 1: (a) Timer = 0.5s (b) Timer = 1s (c) Timer = 5s

3.2 For smaller boards

For the empty-2x2 board, a simple pattern emerged. At any timer and against any opponent, the player moving 2nd, showcased a much higher win-rate than any agent who took the first move. It makes intuitive sense! With an even number of squares to be filled in, the player that makes the 2nd move will also be the one making the last move of the game - and *this* is truly "overpowered"! Making the last move is *guaranteed* to bring the player additional 7 points, for the last move is forced to complete all 3 groups - a column, a row and a region. For a game on a large board, these guaranteed 7 points will still play a significant role, but it is less of a critical factor. For all the smaller boards than 3x3 - the impact of the last move will be tremendous. Notice how this problem also is applicable to boards with an even number of squares to be filled - then, it will be the "going first" player who gets the advantage.

4 Reflection

By far, the strongest point of our agent is how exhaustive the search is. Given a long timer per turn, it would theoretically be able to fully explore the game tree and "solve" the current competitive Sudoku board. Furthermore, the included alpha-beta pruning as well as optimizations in how the minimax is run result in searches over 15x faster than a 'naive' implementation of the algorithm.

There are however some limitations to the algorithm. First of all, the elephant in the room, the performance is sub-optimal. Due to the theoretical groundedness of the algorithm and the positive results of previous practical tests we believe this is due to an uncaught bug in the implementation. At the time of writing it is however unfortunately too late to fix this. See this as an exercise for the reader. On more theoretical notes, due to the fact that the amount of 'thinking' time is unknown, the algorithm has it essentially has to do many different runs of minimax, instead of a single deep one, costing valuable time. Additionally, while not allowed by the rules, an optimal version of this algorithm would send information between moves. Specifically, it would store the results of a tree of depth n, so that during its next move it would not have to calculate minimax till depth n-2, but could use the already established calculations instead. Further optimizations can of course also be implemented, such as the inclusion of transposition tables and improvements to our sub-optimal Sudoku solver. Lastly, our algorithm is missing the tactical ability to intentionally skip moves. As per the rules, the AI could intentionally pick a legal, non-taboo move that makes the Sudoku unsolvable in order to skip their turn without filling in any squares. This proved too difficult to implement into our framework but will put the AI at a disadvantage against AIs that are capable of this.

5 Code overview

Below, we would like to give a very top-level overview of the code appended, so that the reviewers can understand it more easily.

Lines 21-23: 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 26: We solve the sudoku using the approach given in 2.2.

Lines 29-32: 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 34-48: We simply update the "proposed move" using this function.

Lines 88-91: Calculating how many point would the given move contribute to the overall result (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 93-97: 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.

Lines 99-100: 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 102-111: Since in this place we finished inspecting the "previous" branch of the game tree, we restore the values of numbers 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 113-117: See Section 2.3 (Alpha-beta pruning).

Lines 121-133: A function to retrieve all the empty squares (where inputting a number is possible) from the current game state.

Lines 135-170: The actual function to retrieve the numbers of empty squares per group, as invoked in Lines 21-23.

Lines 172-204: The actual function to retrieve the missing numbers per group, as invoked in Lines 21-23.

Lines 206-260: Very naïve sudoku solver, explained in Section 2.2 ("Solve_sudoku" function).

Python files

Code Listing 1: `sudokuai.py`.

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

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 minimax needs
        open_squares = game_state.board.get_open_squares()
        empty_squares = game_state.board.get_empty_squares()
        numbers_left = game_state.board.get_numbers_left()

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

        # 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:
        """
        Updates the best move that has been found so far.
        N.B. DO NOT CHANGE THIS FUNCTION!
        This function is implemented here to save time with importing
        @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()

#The below 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

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")):
    """
    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
        amount_finished = (empty_squares["row"][move[0]] == 1) + (empty_squares["column"][move[1]] == 1) + (empty_squares["region"][int(move[0] / m)*m + int(move[1] / n)] == 1)
        new_score = current_score + multiplier*{0:0, 1:1, 2:3, 3:7}[amount_finished]
```

```python
        # 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"][int(move[0] / m)*m + int(move[1] / 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"][int(move[0] / m)*m + int(move[1] / n)] += 1

        # 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_squares(board: SudokuBoard) -> list:
    """
    For the current board, gets all square that are still empty.
    @param board: The board this should be done on.
    @return: a list of all empty coordinates as sets.
    """
    open_squares = []
    for i in range(board.N):
        for j in range(board.N):
            if board.get(i,j) == SudokuBoard.empty:
                open_squares.append((i,j))

    return open_squares

def get_empty_squares(board: SudokuBoard) -> dict:
    """
    For the current board, gets the number of empty squares for each row/column/region.
    @param board: The board this should be done on.
    @return: A dictionary with keys: "row", "column" and "region"; and values being lists with the number of empty squares per
                                 group.
    """
    # Calculates the number of empty squares per row
    empty_row = []
    for i in range(board.N):
        current_empty_row = 0
        for j in range(board.N):
            current_empty_row += board.get(i,j) == SudokuBoard.empty

        empty_row.append(current_empty_row)

    # Calculates the number of empty squares per column
    empty_column = []
    for i in range(board.N):
        current_empty_column = 0
        for j in range(board.N):
            current_empty_column += (board.get(j,i) == SudokuBoard.empty)

        empty_column.append(current_empty_column)

    # Calculates the number of empty squares per region
    empty_region = []
    for i in range(board.N):
        current_empty_region = 0
        for j in range(board.N):
            row = int(i/board.m)*board.m + int(j/board.n)
            column = (i%board.m)*board.n + (j%board.n)
            current_empty_region += (board.get(row,column) == SudokuBoard.empty)

        empty_region.append(current_empty_region)

    return {"row": empty_row, "column": empty_column, "region": empty_region}

def get_numbers_left(board: SudokuBoard):
    '''
    For the current board, gets the numbers not yet in a group for each row/column/region.
    @param board: The board this should be done on.
    @return: A dictionary with keys: "row", "column" and "region"; and values being lists with the numbers unused per group.
    '''
    # Calculates the missing numbers for each row
    rows = []
    for row in range(board.N):
        this_row = []
        for column in range(board.N):
            this_row.append(board.get(row, column))
        rows.append({x for x in range(1,board.N+1) if x not in set(filter((0).__ne__, this_row))})

    # Calculates the missing numbers for each column
    columns = []
    for column in range(board.N):
```

```
        this_column = []
        for row in range(board.N):
            this_column.append(board.get(row, column))
        columns.append({x for x in range(1,board.N+1) if x not in set(filter((0).__ne__, this_column))})

    # Calculates the missing numbers for each region
    regions = []
    for region in range(board.N):
        this_region = []
        for value in range(board.N):
            row = int(region/board.m)*board.m + int(value/board.n)
            column = (region%board.m)*board.n + (value%board.n)
            this_region.append(board.get(row, column))
        regions.append({x for x in range(1,board.N+1) if x not in set(filter((0).__ne__, this_region))})

    return {"rows": rows, "columns": columns, "regions": regions}

def solve_sudoku(board, open_squares, numbers_left):
    '''
    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.
    '''
    # Finds all squares where only one number is possible
    result = []
    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 len( possibilities ) == 1:
            number = next(iter(possibilities))

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

    # 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])
            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])

        return solve_sudoku(board, open_squares, numbers_left)

    # If no squares can be filled in, keep making a guess until you hit a correct one
    elif 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

# Adds three function as methods of SudokuBoard for ease of use
SudokuBoard.get_open_squares = get_open_squares
SudokuBoard.get_empty_squares = get_empty_squares
SudokuBoard.get_numbers_left = get_numbers_left
```