Assignment A2: Team 33

Lucas van Osenbruggen Martin Miksik Luca Mainardi

1 Agent Description

We implement an agent for the game *Competitive Sudoku*, in which two players must complete a Sudoku puzzle one turn at a time and receive points for completing regions (Krak [2023]). In this game our agent must propose its next move based on the current state of the game, which can be defined by a score for both players and the current values on the board. The state does not depend on other factors such as the move history and we always know that we are the player that has the turn. Our agent then has some indeterminate amount of time to decide on a move before the turn ends. If it does not propose a move or propose a move that is illegal according to the rules of Sudoku it loses. Additionally, the agent may not propose *taboo moves*, which are moves that would make the puzzle unsolvable.

The agent uses a search strategy based on the *Minimax* algorithm (Polak [1989]). This is implemented in the function `minimax`. This strategy tries to maximise the score of the player, taking the potential moves of the opponent into account by searching a *game tree* of all subsequent moves. Since the turn time is indeterminate we cannot reliably search to any depth. For a board of size $N \times N$, there are $\mathcal{O}(N^3)$ initial moves possible for the first turn, and $\mathcal{O}(N^3!)$ possible games. To deal with this time constraint, our agent will always start by proposing a random legal move. After this, agent searches the game tree until some interatively increasing depth – a strategy which is called *iterative deepening* (Korf [1985]). This is implemented in the function `compute_best_move`. Searching until a limited depth requires an evaluation function of an incompleted game. To this end, we define the following evaluation score:

$$S(s_{max}, s_{min}) = s_{max} - s_{min} \tag{1}$$

Where s_{max} and s_{min} represent the game scores at the maximum search depth of the maximising and minimising player respectively. This evaluation score is optimal, when searching the entire search tree it will find the optimal moves that result in a maximal score for the player. This iterative deepening strategy ensures that the agent will propose the best move it can within some time limit. Furthermore, given more time, the agent will find better moves.

The above approach is still limited by the size of the game tree. Note that when the agent is unable to search the game tree until depth 1 it plays randomly from the legal moves. When it is only able to search until depth 1, which often happens at the beginning of games, it is playing with a greedy strategy. In order to search deeper in the search tree, we employ multiple heuristics:

Alpha beta pruning is a strategy in which branches of the search tree that cannot lead to a better score than found so far are not searched (Knuth and Moore [1975]). This pruning strategy is not strictly a heuristic, since the optimal score is still found, but does allow for searching fewer branches.

Fewer filled heuristic is used to evaluate the state at the maximum search depth of the recursion. It is implemented in the function `evaluate_state`. The heuristic adds to the state evaluation defined in Equation 1 and is defined as:

$$h(R, s_{max}, s_{min}, p_{current}, p_{max}) = S(s_{max}, s_{min}) + 0.1 \cdot e(R, p_{current}, p_{max}) \tag{2}$$

$$e(R, p_{current}, p_{max}) = \begin{cases} \frac{1}{|R|} \sum_{r \in R} \frac{1}{|r|} \cdot |\{v \in r : v = 0\}| & \text{if } p_{current} = p_{max} \\ 0 & \text{else} \end{cases}$$

Where R represents the set of regions, i.e., rows, columns and blocks. Each region is defined as a matrix of positions with values $v \in \mathbb{N} \cap [0, N]$. Additionally, the value $v = 0$ indicates that a certain position is empty. Finally, $p_{current}$ and p_{max} indicate the current player in the recursion and the maximising player in the search respectively.

For this heuristic, the evaluation score will be higher when the board is more empty. This is thus a search strategy which prefers more empty boards. The intuition behind this is that we cannot look far ahead at the early game stage and we do not want to set up regions for the other player to complete. It may be observed that the value of e is at max 1 for a completely empty board. However, the contribution of the factor e is at max 0.1 and thus will never dominate the factor S. In other words, this heuristic will always prioritise the scoring of points in the game. As a result the new evaluation score is still optimal in unlimited time but may also perform better in limited time. Finally, we only apply the factor e from the perspective of the maximising player since we do not want to assume the strategy of the opponent.

Avoid two-free heuristic is used to change the order in which moves are evaluated by the agent. The heuristic is implemented in the function `find_initial_moves_heuristics`. This approach orders the moves ascendingly by a weight which is defined as:

$$w(R, T, m) = \begin{cases} \infty & \text{if } |l(R, T, m_{row}, m_{column})| = 2 \\ |l(R, T, m_{row}, m_{column})| & \text{else} \end{cases} \tag{3}$$

$$\begin{aligned} l(R, T, row, column) = \{&(row, column, v) : v \in \mathbb{N} \cap [1, N]\} \\ &- \{(row', col', v') : v' \in r \wedge row', col', r \in R_{row,col}\} \\ &- T \end{aligned}$$

Where m represents a legal move, which is a tuple of the form $(row, column, value) \in (\mathbb{N} \cap [0, N-1])^2 \times (\mathbb{N} \cap [1, N])$. Additionally, the function l represents the set of legal moves that can be performed at some position, taking into account the values in the adjacent regions $R_{m_{row}, m_{column}}$ and list of taboo moves T.

This ordering has two heuristic features. First, it prioritises searching positions with fewer possible moves. This limits the search space, allowing for a deeper search. Additionally, it will first look at moves that are more likely to result in completed regions. In our implementation we update the best move so far after each evaluation and thus we may find a better move in the limited time. This approach is still optimal in unlimited time since all moves are still considered.

The second heuristic feature of this ordering is that it evaluates positions with two possible moves last. The idea behind this is that making a move here will allow the opponent to complete the region. This is particularly useful in the early game where the search depth is too limited for the agent to model the opponent's moves.

Using the above heuristics prioritises which parts of the game tree to search without pruning any branching. This allows the agent to make better use of the limited time while still playing optimally given unlimited time.

2 Agent Analysis

For the analysis we collected data by playing four versions of our agent. The first is essentially an optimization of the agent presented in assignment 1, where the minimax algorithm is properly functioning and a new data structure is used to save and update the list of valid moves. The second agent implements the heuristic that orders the moves before calling the minimax algorithm, to optimize the search. The third version implements the heuristic that favors large empty regions in the initial phase of the game. Both of the mentioned heuristics are described in detail in the Description section. Finally, the third agent tested uses both heuristics simultaneously. Initially all four agents were made to play against the greedy player on the empty 3x3 board and on all the non-empty boards (easy, hard and random). For each board and time limit, a match of 20 games was carried out, in half of which the tested agent was the starting player. The purpose of this test was to verify whether the use of heuristics to guide the search and evaluate the state of the game actually leads to improvements in the agent's performance. Furthermore, we were interested in analyzing the simultaneous operation of the two designed heuristics, to detect any contrasts between them.

The results can be seen in Figure 2. All four agents are in most cases able to beat the greedy player in over 50% of games with limit time greater than 0.1, except when playing on the easy 2x2 board. In this case, in fact, even if you always play the best moves, it is impossible to win if you start the game first. A draw is therefore the best result that can be achieved as it involves having won all the games in which you are the second player.

It has also been observed that on average the application of the heuristics individually does not bring significant advantages compared to using only the minimax algorithm with alpha-beta pruning and iterative deepening. In particular, applying both heuristics seems to lead to a decrease in performance, especially when the time limit is less than 5 seconds.

As directly observable from Figure 2, performance is very poor when the time available is very limited (0.1s). This issue is analyzed in detail in Section 3.

Subsequently, all the agents except the one that does not implement any heuristics were made to play against the agent we proposed in Assignment 1, which implements a minimax algorithm with alpha-beta pruning and iterative deepening. The agent without heuristics was excluded as it is simply an optimization of the agent developed for Assignment 1 and would only have added redundancy to the data obtained. Again 20 games were played for each configuration, alternating the starting player each time. In this case, our goal was to verify the consistency of the improvements obtained compared to the previous version of the agent. The results can be seen in figure 2.

Also in this case our agent, in all its versions, won the majority of the games, except with the easy 2x2 board and with a time limit of 0.1s. The average win rate observed in this case is even higher than that against the greedy player. This behavior is explained by the fact that the agent we developed for assignment 1 had a bug in the minimax implementation and therefore generally had worse performance than the greedy player. From Figure 2 it is even more evident than before how applying both heuristics simultaneously performs worse than applying them individually. The reason for this behavior could be a conflict between the two heuristics and will need to be studied better in the future. As regards the two agents that use a single heuristic, a higher win rate was measured when the board was large (3x4 and 4x4). This demonstrates the effectiveness of heuristics, as they aim to improve the choices of moves especially in the early game, and on large boards this phase lasts longer.

3 Reflection

First we will reflect on our implementation of the Minimax algorithm and then on each of the several extensions: a) Alpha-beta Pruning, b) Iterative Deepening, c) Avoid two-free heuristic, d) Fewer filled heuristic.

3.1 Minimax

In general the agent without heuristics appears to perform best, reliably defeating the agent from assignment 1. This is partially a result of bug-fixes of our initial implementation. Namely: not using the `min` function for the minimising player and incorrectly calculating legal moves.

We also reliably beat the greedy player when the turn time is 0.5 seconds or more. However, when the board is not filled, our implementation does not have enough time to explore depths higher than 1 and, thus, operates much like a greedy player who only looks 1 step ahead. For example, our agent typically only explores depths greater than 1 when the number of legal moves decreases to approximately 170.

As an optimisation, the new agent uses a multidimensional array that stores for each move whether it is legal. This allows us to locally update the set of legal moves, e.g. only in the same regions. Our testing results are inconclusive whatever this improves agents performance on its own, however the introduces data structure is useful for other heuristics. Importing `numpy` costs around 0.003 second[1], however there must be some other hidden overhead because with `numpy` import at the top level we would often fail to propose a move with time ≤ 1.

Conclusion:

+ Our agent is able to reliably beat greedy player.
∼ Benefits of caching legal moves alone are unclear.
- `Numpy` import overhead hurts the performance.

3.2 Alpha-beta pruning

The implementation of Alpha-beta pruning significantly reduces the number of search tree branches to be examined, thus reducing the computation time. Our testing show that on a 2x2 board the Alpha-beta pruning removed about 30 and 71 120 branches on the `easy-2x2` and easy-3x3 boards respectively[2]. Moreover, it is relatively simple to implement.

Conclusion: + High positive impact on performance

3.3 Iterative Deepening

The iterative deepening strategy partially addresses the problem of the unknown time limit, allowing our agent to evaluate multiple moves without having to compute the entire game tree at once. However, for each subsequently depth level the whole game tree has to be recomputed from the *initial game state*. Thus, our agents is repeating work. To illustrate,

[1] As measured using `timeit` for 10 000 runs.

[2] The number of pruned branches, of course, can vary widely per game.

on the `easy-3x3` board the iteration over depth 1 finished 45 times, over depth 2 finished 43 times, over depth 3 finished 17 times and over depth higher than 3 finished 15 times. It is important to stress that to get to dept 4 you have to performer the same calculation for depth 1, 2, and 3.

Conclusion:

+ In early game iterative deepening helps to mitigate the combinatory explosion
- In end game it introduces overhead. Future improvement could be starting with higher initial depth once the number if initial legal moves is sufficiently low.

3.4 Avoid two-free heuristic

This heuristic runs once when the `compute_best_move` is invoked. It examines the list of all legal moves and aims to prioritise those likely to bring immediate rewards while avoiding moves that benefit the opponent. The resulting priority order is not guaranteed to be correct, since the number of options for a cell is influenced by combination of row, column and region. On the other hand, the sorting has negligible time-wise impact of < 0.0001 seconds[3] and results show that it has negligible impact on the performance on average.

Conclusion: ∼ Mixed impact on performance

3.5 Fewer filled heuristic

This heuristics increases evaluation score of moves in empty regions. The rationale for this, similarly to the one of *Avoid two-free heuristic*, is that in the early stages of the game, we cannot anticipate moves too far in advance, and we aim to avoid creating opportunities for the opponent. The heuristics has negligible time-wise impact of 0.003 seconds[4]. The results for this heuristics is fluctuating and it cannot be conclude that it performs better on average. In very limited time scenarios it performs better than *Avoid two-free heuristic*, but with time ≥ 1 second it slightly under-performs it. The combination of both heuristics is detrimental to the effectiveness against our old (*and incorrect minimax*) agent.

Conclusion:

∼ Mixed impact on performance

- The combination with *Avoid two-free* heuristic decreases the performance

3.6 Summary

Our agent adheres to the baseline established in lectures post-assignment 1, consistently outperforming the greedy player with a time constraint of ≥ 1. Alpha-beta pruning is arguably *the best* extensions because of its favourable cost-to-performance ratio, given its straightforward implementation and substantial improvement in agent performance. Looking forward, the next assignment should focus on refining the iterative deepening strategy by predicting initial depth to mitigate unnecessary computations. Moreover, the performance impact of combining of both heuristics should be explored. Lastly, we must remove dependency on `numpy` as importing it has more negatives than positives.

[3]As measured using `timeit` for 10 000 runs.
[4]As measured using `timeit` for 10 000 runs.

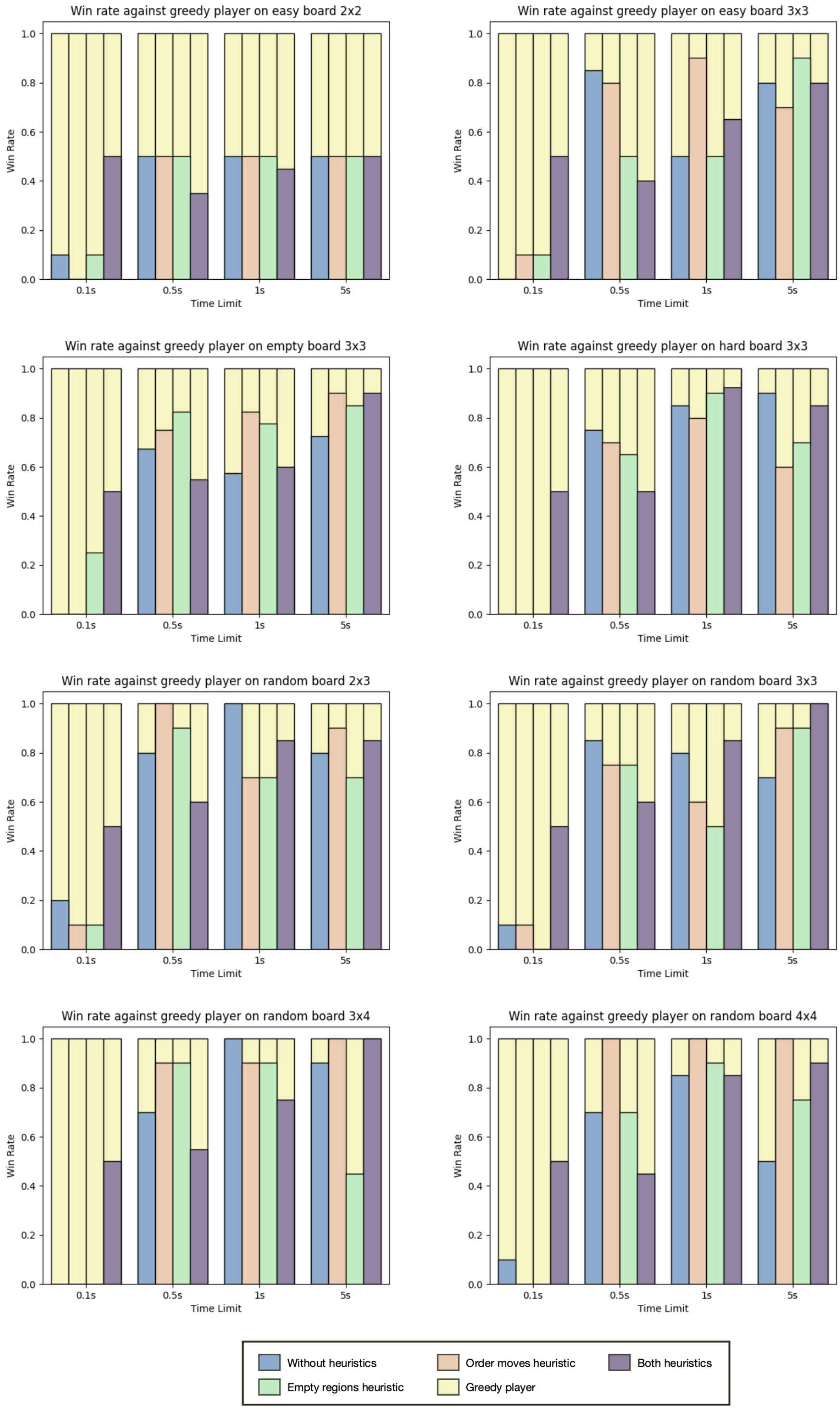

Figure 1: Results against Greedy Player

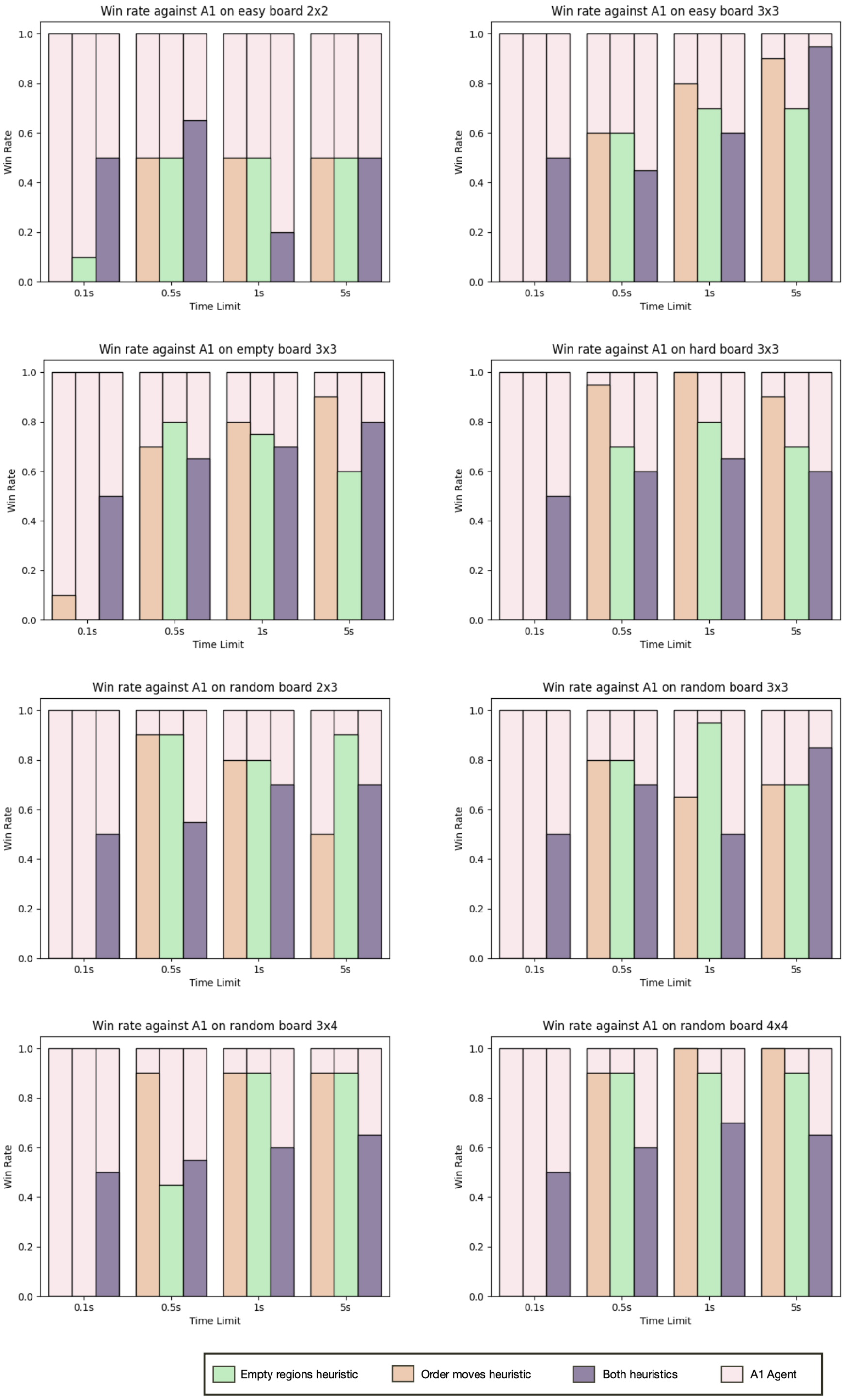

Figure 2: Results against A1 Player

References

Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975. ISSN 0004-3702. doi:https://doi.org/10.1016/0004-3702(75)90019-3. URL https://www.sciencedirect.com/science/article/pii/0004370275900193.

Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985. ISSN 0004-3702. doi:https://doi.org/10.1016/0004-3702(85)90084-0. URL https://www.sciencedirect.com/science/article/pii/0004370285900840.

Thomas Krak. Competitive sudoku: Rules, 2023. URL https://canvas.tue.nl/courses/25392/files/5120538?module_item_id=522484. Accessed: 2023-11-29.

E. Polak. *Basics of Minimax Algorithms*, pages 343–369. Springer US, Boston, MA, 1989. ISBN 978-1-4757-6019-4. doi:10.1007/978-1-4757-6019-4_20. URL https://doi.org/10.1007/978-1-4757-6019-4_20.

Python files

Code Listing 1: `utils.py`

```python
""" Module containing helper functions for the sudoku game.

These functions are used for game specific logic, such as checking if a move is legal or calculating the score of a move.
They do not contain any strategic logic specific to the AI agent.
"""

from typing import Iterator
from competitive_sudoku.sudoku import SudokuBoard, GameState, Move


def next_player(current_player: int) -> int:
    """Returns the next player.

    :param current_player: the current player.
    :return: the next player.
    """
    return (current_player + 1) % 2


def block_index(row: int, col: int, board) -> tuple[int, int]:
    """Transform in which block a certain coordinate is.

    Enumerates blocks in the same way as coordinates, top left is (0, 0).

    :param row: a row index inside the block.
    :param col: a column index inside the block.
    :param board: the board to which the indices belong.
    :return: the indices of the block in the board (vertical, horizontal).
    """
    return (
        row // board.region_height(),  # floor division
        col // board.region_width(),
    )


def block_range(*, row: int, col: int, board: SudokuBoard) -> Iterator[tuple[int, int]]:
    """Return the range of indices in a block.

    :param row: a row index inside the block.
    :param col: a column index inside the block.
    :param board: the board to which the indices belong.
    :return: an iterator of indices in the block.
    """
    region_width, region_height = board.region_width(), board.region_height()
    block_indices = block_index(row, col, board)
```

```
    region_start = {
        "x": block_indices[1] * region_width,
        "y": block_indices[0] * region_height,
    }

    region_end = {
        "x": (block_indices[1] + 1) * region_width,
        "y": (block_indices[0] + 1) * region_height,
    }

    for row in range(region_start["y"], region_end["y"]):
        for col in range(region_start["x"], region_end["x"]):
            yield row, col


def is_illegal(*, move: Move, state: GameState) -> bool:
    """
    Returns whether a move is illegal.

    A move is illegal if it puts a duplicate value in a position, block, row or column.
    Additionally, moves should not be 'taboo', meaning that they make the board unsolvable.
    Illegal moves are not allowed and will result in a loss.

    :param move: The move to be checked.
    :param state: The current state of the game.
    :return: Whether the move is illegal (True) or not (False).
    """

    # Check if square is empty
    if state.board.get(move.i, move.j) != SudokuBoard.empty:
        return True

    # Check if move is taboo
    if move in state.taboo_moves:
        return True

    # Check duplicate value in row
    if any(
            state.board.get(row, move.j) == move.value
            for row in range(state.board.board_height())
    ):
        return True

    # Check duplicate value in column
    if any(
            state.board.get(move.i, col) == move.value
            for col in range(state.board.board_width())
    ):
        return True
```

```
    # Lastly check for duplicate values in the region
    return any(
        state.board.get(row, col) == move.value
        for row, col in block_range(row=move.i, col=move.j, board=state.board)
    )


def calculate_move_score(game_state: GameState, move: Move) -> int:
    """Check if a move completes any regions and returns the score earned.

    Static method, uses less memoy sinds it does not need to be instantiated.

    :param game_state: The current game state. Describes the board, scores and move history.
    :param move: The move to be evaluated
    :return: The score earned by the move (0, 1, 3 or 7)
    """
    row_complete = col_complete = block_complete = True

    # Check if completed a row
    for col in range(game_state.board.board_width()):
        if game_state.board.get(move.i, col) == SudokuBoard.empty:
            row_complete = False
            break

    # Check if completed a column
    for row in range(game_state.board.board_height()):
        if game_state.board.get(row, move.j) == SudokuBoard.empty:
            col_complete = False
            break

    # Check if completed a block
    for row, col in block_range(row=move.i, col=move.j, board=game_state.board):
        if game_state.board.get(row, col) == SudokuBoard.empty:
            block_complete = False
            break

    # Return score by move
    regions_complete = int(row_complete) + int(col_complete) + int(block_complete)
    return {
        0: 0,
        1: 1,
        2: 3,
        3: 7,
    }[regions_complete]
```

Code Listing 2: `team33_A1/sudokuai.py`.

```
"""Competitive Sudoku AI.

Adapted from /naive_player/sudokuai.py
```

```
Changes:
A1: basic MiniMax implementation.
"""

import random
from competitive_sudoku.sudoku import GameState, Move, SudokuBoard
import competitive_sudoku.sudokuai
from team33_A1.utils import is_possible, is_illegal , region_range, next_player


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

    def __init__(self):
        super().__init__()
        self.transposition_table = []

    def minimax(
        self,
        game_state: GameState,
        move: Move,
        moves: list[Move],
        current_player: int,
        maximizing_player: int,
        depth: int,
        *,
        alpha: int = float("-inf"),
        beta: int = float("inf"),
    ) -> int:
        """Returns the score of the best move.

        The score is the number of empty squares on the board after the move.

        Searches until some depth.
        Uses alpha beta pruning to avoid searching branches which cannot lead to better results.
        """
        # TODO store intermittent state between iterative deeping steps (transposition table)

        old_score = game_state.scores[current_player]

        # Apply move and find new possible moves
        game_state.board.put(move.i, move.j, move.value)
        game_state.scores[current_player] += self.score_move(game_state, move)

        # TODO edit moves in place for efficiency, calculate if newly illegal instead of checking entire board
```

```python
        # TODO use different data structure for moves, e.g. binary matrix
        new_moves = [
            m for m in moves if not is_illegal(move=move, for_state=game_state)
        ]

        # switches current player (more efficient than storing all moves in game state)
        current_player = next_player(current_player)

        if (
            depth == 0 or len(new_moves) == 0
        ):  # Game is finished or maximum depth is reached
            # evaluate with high value when the maximising player is winning
            best_value = (
                game_state.scores[maximizing_player]
                - game_state.scores[next_player(maximizing_player)]
            )
        else:
            const_function = max if maximizing_player == current_player else min
            best_value = float("-inf")

            for try_move in new_moves:
                # Recurse and find value up to some depth
                value = self.minimax(
                    game_state,
                    try_move,
                    new_moves,
                    current_player,
                    maximizing_player,
                    depth - 1,
                    alpha=alpha,
                    beta=beta,
                )
                best_value = const_function(best_value, value)
                # Alpha beta pruning, do not search branches which cannot lead to better results
                alpha = const_function(alpha, best_value)
                if beta <= alpha:
                    break

        # Undo move
        game_state.board.put(move.i, move.j, 0)
        current_player = next_player(current_player)
        game_state.scores[current_player] = old_score

        # Recursion result
        return best_value

    @staticmethod
    def score_move(game_state: GameState, move: Move) -> int:
        """
```

```
        Check if a move completes any regions and returns the score earned
        """
        # TODO can make this faster with sums of rows, columns and regions
        row_complete = col_complete = block_complete = True

        # Check if completed a row
        for col in range(game_state.board.board_width()):
            if game_state.board.get(move.i, col) == SudokuBoard.empty:
                row_complete = False
                break

        # Check if completed a column
        for row in range(game_state.board.board_height()):
            if game_state.board.get(row, move.j) == SudokuBoard.empty:
                col_complete = False
                break

        # Check if completed a block
        for row, col in region_range(row=move.i, col=move.j, board=game_state.board):
            if game_state.board.get(row, col) == SudokuBoard.empty:
                block_complete = False
                break

        # Return score by move
        regions_complete = int(row_complete) + int(col_complete) + int(
                                                              block_complete)
        return {
            0: 0,
            1: 1,
            2: 3,
            3: 7,
        }[regions_complete]

    def compute_best_move(self, game_state: GameState) -> None:
        board_size = game_state.board.board_height()
        # TODO save state between moves, not allowed for A1
        # TODO create unitttests
        # TODO also actively avoid taboo moves (to avoid loss of move)?

        # Generate possible moves
        initial_moves = [
            Move(i, j, value)
            for i in range(board_size)
            for j in range(board_size)
            for value in range(1, board_size + 1)
            if is_possible(move=Move(i, j, value), for_state=game_state)
        ]

        # Shuffle moves to be less predictable
        random.shuffle(initial_moves)
```

```python
        # Propose a certain move (initial, avoid timeout)
        self.propose_move(initial_moves[0])

        # evaluate different moves based on minimax
        # TODO track lime limit
        best_move: tuple[Move, float] | None = None
        for depth_limit in range(1, len(initial_moves), 1):
            # evaluate different moves based on minimax
            for move in initial_moves:
                player_index = (
                    game_state.current_player() - 1
                )  # player_index is 0 or 1 (self or opponent)

                value = self.minimax(
                    game_state,
                    move,
                    initial_moves,
                    player_index,  # Current player is self
                    player_index,  # Maximising own score
                    depth_limit,
                )

                if best_move is None or value > best_move[1]:
                    best_move = (move, value)
                    # Update proposed move (best so far, avoid timeout while find a better move)
                    self.propose_move(best_move[0])
```

Code Listing 3: `team33_A2/sudokuai.py`.

```python
"""Competitive Sudoku AI.

Adapted from /naive_player/sudokuai.py

A1: iterative deepening minimax search with alpha beta pruning.
A2: heuristic search.
"""

import os
from random import shuffle

# from numpy import full

# Import types and libraries
from competitive_sudoku.sudoku import GameState, Move, SudokuBoard
from competitive_sudoku.sudokuai import SudokuAI

from .utils import block_range  # Game specific logic
from .utils import block_index, calculate_move_score, is_illegal, next_player


class SudokuAI(SudokuAI):
```

```python
    """
    Sudoku AI agent that computes a move for a given sudoku configuration .
    """

    def __init__(self):
        super().__init__()
        self . transposition_table = []

    def update_legal( self , game_state: GameState, move: Move, moves: dict) -> list:
        """Update which moves are legal in the recursive minimax search.

        :param move: The move to be evaluated
        :param moves: A dictionary containing the inital set of moves, whether they
                                                are still legal and other properties of the
                                                moves.
            initial : list of initally legal moves, used as subset to avoid iterating
                                                over all moves
            legal : numpy array of shape (board_size, board_size, board_size + 1) where
                                                legal[i, j, k] is True if Move(i, j, k) is
                                                legal
            count: Counter for legal moves, avoid repeated iteration
        : return : A list of moves that were invalidated by the move.
        """
        _moves_invalidated = []
        for row in range(game_state.board.board_height()):
            # move invalidates another move if not already illegal , avoid double
                                                counting
            if moves["legal"][row, move.j, move.value]:
                _moves_invalidated.append(Move(row, move.j, move.value))
                moves["legal"][row, move.j, move.value] = False  # Set legal status
        for column in range(game_state.board.board_width()):
            if moves["legal"][move.i, column, move.value]:
                _moves_invalidated.append(Move(move.i, column, move.value))
                moves["legal"][move.i, column, move.value] = False
        for row, col in block_range(row=move.i, col=move.j, board=game_state.board):
            if moves["legal"][row, col, move.value]:
                _moves_invalidated.append(Move(row, col, move.value))
                moves["legal"][row, col, move.value] = False
        moves["count"] -= len(_moves_invalidated)
        return _moves_invalidated

    def minimax(
            self ,
            game_state: GameState,
            move: Move,
            moves: dict,
            current_player: int,
            maximizing_player: int,
            depth: int,
            *,
            alpha: float = float("-inf"),
```

```python
            beta: float = float("inf"),
    ) -> float:
        """Returns the score of a given move.

        Minimax search that considers the perspectives of two players.
        Searches until some depth before returning the score.
        Uses alpha beta pruning to avoid searching branches which cannot lead to better results.

        :param game_state: The current game state. Describes the board, scores, taboo moves and move history.
        :param move: The move to be evaluated
        :param moves: A dictionary containing the inital set of moves, whether they are still legal and other properties of the moves.
            initial: list of initally legal moves, used as subset to avoid iterating over all moves
            legal: numpy array of shape (board_size, board_size, board_size + 1) where legal[i, j, k] is True if Move(i, j, k) is legal
            count: Counter for the number of legal moves.
            free: shows per region (row, col or block) what the number of free squares is, used in heuristics.
        :param current_player: The player who's turn it is. Will be the same throughout the turn. (0 or 1 for first or second player)
        :param maximizing_player: The player who's score is to be maximised. (0 or 1 for first or second player)
        :param depth: The maximum depth to search before returning the score.
        :param alpha: The highest so far value for alpha beta pruning. (initially -inf)
        :param beta: The lowest so far value for alpha beta pruning. (initially inf)
        :return: The score of the move (higher is better for maximizing player, lower is better for minimizing player)
        """
        block_indices = block_index(move.i, move.j, game_state.board)

        # Apply move, update resulting scores
        # Update legal moves and count newly invalidated moves
        game_state.board.put(move.i, move.j, move.value)
        _score_achieved = calculate_move_score(game_state, move)
        game_state.scores[current_player] += _score_achieved
        _moves_invalidated = self.update_legal(game_state, move, moves)
        # switches perspective to other player
        current_player = next_player(current_player)

        # Update properties used in heuristics
        moves["free"]["row"][move.i] -= 1
        moves["free"]["col"][move.j] -= 1
        moves["free"]["block"][block_indices[0]][block_indices[1]] -= 1
```

```
        # Search until game is finished or maximum depth is reached
        if depth == 0 or moves["count"] == 0:
            # evaluate the current board
            best_value = self.evaluate_state(
                maximizing_player, current_player, game_state, moves["free"]
            )
        else:
            if maximizing_player == current_player:  # maximising player
                best_value = float("-inf")
                for try_move in moves["initial"]:
                    if not moves["legal"][try_move.i, try_move.j, try_move.value]:
                        continue
                    # Recurse and find value up to some depth
                    value = self.minimax(
                        game_state,
                        try_move,
                        moves,
                        current_player,
                        maximizing_player,
                        depth - 1,
                        alpha=alpha,
                        beta=beta,
                        )
                    best_value = max(best_value, value)
                    alpha = max(alpha, best_value)
                    if beta < alpha:
                        break
            else:  # minimising player
                best_value = float("inf")
                for try_move in moves["initial"]:
                    if not moves["legal"][try_move.i, try_move.j, try_move.value]:
                        continue
                    # Recurse and find value up to some depth
                    value = self.minimax(
                        game_state,
                        try_move,
                        moves,
                        current_player,
                        maximizing_player,
                        depth - 1,
                        alpha=alpha,
                        beta=beta,
                        )
                    best_value = min(best_value, value)
                    beta = min(beta, best_value)
                    if beta < alpha:
                        break

        # Undo move and its effects
        current_player = next_player(current_player)
        moves["free"]["row"][move.i] += 1
```

```
        moves["free"]["col"][move.j] += 1
        moves["free"]["block"][block_indices[0]][block_indices[1]] += 1
        moves["count"] += len(_moves_invalidated)
        for inv_move in _moves_invalidated:
            moves["legal"][inv_move.i, inv_move.j, inv_move.value] = True
        game_state.scores[current_player] -= _score_achieved
        game_state.board.put(move.i, move.j, 0)

        # Recursion result
        return best_value

    def evaluate_state(
        self,
        maximizing_player: int,
        current_player: int,
        game_state: GameState,
        free: dict,
    ) -> float:
        """Heuristic evaluation of the current game state.

        Is used by minimax to evaluate the current game state which is most often not a complete game.
        Base score is the difference between the scores of the two players since maximizing this will result in a win.
        Additional heuristic contributions are made for early game where score is often 0.

        :param maximizing_player: The player who's score is to be maximised. (0 or 1 for first or second player)
        :param game_state: The current game state. Describes the board, scores, taboo moves and move history.
        :param free: The number of free squares per region.
        :return: The score of the game state (higher is better for maximizing player, and vice versa)
        """
        score = (
            game_state.scores[maximizing_player]
            - game_state.scores[next_player(maximizing_player)]
        )

        if not os.environ.get("not_prefer_more_empty"):
            # Scale to avoid this additional heuristic dominating the score
            # Will result in an early game strategy avoiding a filled field.
            # Positive contribution, our player will thus prefer less filled fields.
            if current_player == maximizing_player:
                score += 0.1 * self.prefer_empty_regions(game_state, free)

        return score

    def prefer_empty_regions(self, game_state: GameState, free: dict):
        """Heuristic for evaluating a state.
```

```
        Prefer less filled out regions by counting the number of free moves
        Normalising using the the maximum number of squares in a region

        :param maximizing_player: The player who's score is to be maximised. (0 or 1 for first or second player)
        :param game_state: The current game state. Describes the board, scores, taboo moves and move history.
        :param free: The number of free squares per region.
        :return: The heuristic score, will be 1 for a completely empty field.
        """
        board_size = game_state.board.board_height()
        num_blocks = board_size / game_state.board.region_height()
        score = sum([count / board_size for count in free["row"]]) / board_size / 3
        score += sum([count / board_size for count in free["col"]]) / board_size / 3
        score += (
                sum(
                    sum([count / board_size for count in block]) / num_blocks
                    for block in free["block"]
                )
                / num_blocks
                / 3
        )
        return score

    def find_initial_moves(self, game_state: GameState) -> list[Move]:
        """
        Find all possible moves for a given state. This is copy of method used in A1 used for benchmarking purposes.
        @param game_state: GameState
        @return: list of moves
        """
        board_size = game_state.board.board_width()

        # Generate possible moves
        initial_moves = [
            Move(i, j, value)
            for i in range(board_size)
            for j in range(board_size)
            for value in range(1, board_size + 1)
            if not is_illegal(move=Move(i, j, value), state=game_state)
        ]

        # Shuffle moves to be less predictable
        shuffle(initial_moves)

        return initial_moves

    def find_initial_moves_heuristics(self, state: GameState) -> list[Move]:
        """
        Find all possible moves for a given state and order them by priority. The
```

```
        priority is determined by the number of possible moves for a cell. Cells with fewer possible moves are prioritised because they are
        more likely to result in a completed row, column, or block.
        Any cell with 2 possible moves is *de*-prioritised, because that means the opponent could complete a row, column
        or block next turn.
        @param state: GameState
        @return: ordered list of moves
        """
        size = state.board.board_width()

        # Store moves in a dictionary with the number of possible moves for that cell as key
        priority_dict = dict([(key, []) for key in range(0, size + 1)])

        for i in range(size):
            for j in range(size):
                possible_moves_for_cell = []
                for value in range(1, size + 1):
                    move_candidate = Move(i, j, value)
                    if not is_illegal(move=move_candidate, state=state):
                        possible_moves_for_cell.append(move_candidate)

                priority_dict[len(possible_moves_for_cell)].extend(possible_moves_for_cell)

        key_order = sorted(priority_dict.keys())

        # Prioritise cells that have 2 possible moves,
        # because the opponent could complete a row, column or block
        if len(key_order) > 2:
            key_order.pop(2)
            key_order.append(2)

        # Return list of moves in order of search priority
        return [move for key in key_order for move in priority_dict[key]]

    def compute_best_move(self, game_state: GameState) -> None:
        """Computes the best move for the agent and proposes it.

        Will initially propose a random move, then evaluate different moves based on minimax search.
        Since the turn time is not known it will propose the best move found so far by iteratively deepening the search.

        :param game_state: The current game state. Describes the board, scores and move history.
        """
        board_size = game_state.board.board_height()
        num_blocks = board_size // game_state.board.region_height()
```

```python
        # print(f"Avoid 2 moves: {not os.environ.get('not_avoid_2_moves')}")
        # print(f"Prefer empty regions: {not os.environ.get('not_prefer_more_empty')}
            ")

        # Generate possible moves
        initial_moves = (
            self.find_initial_moves_heuristics(game_state)
            if not os.environ.get("not_avoid_2_moves")
            else self.find_initial_moves(game_state)
        )

        # Move cache
        moves = {}

        # List of initially legal moves, used as subset to avoid iterating over all
            moves
        moves["initial"] = initial_moves

        # Propose a random move (initial, avoid timeout)
        self.propose_move(moves["initial"][0])

        # Avoid repeated regeneration of legal moves by tracking their status
        from numpy import full
        moves["legal"] = full(
            shape=(board_size, board_size, board_size + 1),
            dtype=bool,
            fill_value=False,
        )
        moves["count"] = len(moves["initial"])
        for move in moves["initial"]:
            moves["legal"][move.i, move.j, move.value] = True

        # Track some properties of the game to be used in statistical search
        # Count how many free squares there are per region
        # TODO do this above when generating moves?
        moves["free"] = {
            "row": [
                len(set(((m.i, m.j) for m in moves["initial"] if m.i == y)))
                for y in range(board_size)
            ],
            "col": [
                len(set(((m.i, m.j) for m in moves["initial"] if m.j == x)))
                for x in range(board_size)
            ],
            "block": [
                [
                    len(
                        set(
                            (
                                (m.i, m.j)
```

```
                                for m in moves["initial"]
                                if block_index(m.i, m.j, game_state.board) == (i, j)
                            )
                        )
                    )
                    for j in range(num_blocks)
                ]
                for i in range(num_blocks)
            ],
        }

        # Iteratively increase the search depth of minimax
        best_move: tuple[Move, float] | None = None
        for depth_limit in range(1, len(moves["initial"]), 1):
            # evaluate different moves based on minimax
            for move in moves["initial"]:
                # player_index is 0 or 1 (first or second player)
                player_index = game_state.current_player() - 1

                value = self.minimax(
                    game_state,
                    move,
                    moves,
                    player_index,  # Current player is self
                    player_index,  # Maximising own score
                    depth_limit,
                )

                if best_move is None or value > best_move[1]:
                    best_move = (move, value)
                    # Update proposed move (best so far, avoid timeout while find a better move)
                    self.propose_move(best_move[0])
                    # print(move.i, move.j, value)
```