Assignment A1: Team 33

Lucas van Osenbruggen Martin Miksik Luca Mainardi

1 Agent Description

We have implemented an agent for the game *Competitive Sudoku* (Unknown [2023]). In this game our agent is prompted for their 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 history and we always know that we are the player that has the turn. The state can thus be said to be *Markovian*. Our agent then has some indeterminate amount of time to decide on a move before the turn ends.

We implement a search strategy based on the *MiniMax* algorithm (Polak [1989]). We first find the set of legal moves, which are positions that are empty on the board, have not been declared to be taboo and do not violate the rules of the game. We then iterate over the set of legal moves, update the state (i.e., the board and scores), remove moves from the set that have become illegal and recursively repeat this. This approach is often represented as a *game tree*, where leaves represent finished games, branches decisions and paths from the root to a leave playouts of the game.

This basic approach suffers from computational explosion. 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 playouts. Enumerating all these playouts is infeasible for larger board sizes, especially within the time that a turn lasts.

In order to deal with the limited time, the agent will always begin by proposing a random legal move. This avoids the situation where the agent loses instantly, except for extremely short turns where there is too little time to find the set of legal moves.

After this, the agent will use an *iterative deepening* approach to searching the game tree (Korf [1985]). On the first iteration, i.e., depth limit is 1, all moves for the maximising player are attempted. In the next iteration, it will also consider the moves of the minimising player. It continues searching to an increasing number of turns until the time limit is reached and execution stopped. Searching until a limited depth requires an evaluation function of an incompleted game. To this end, we define the following heuristic:

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

Where s_{max} and s_{min} represent the scores of the maximising and minimising player respectively. This heuristic has the property of being high when the maximising player has a high score compared to the minimising player. In an end state of the game, the player with the highest score wins. Therefore the heuristic is *admissible* and an exhaustive search gives an optimal result. As a result of this strategy, the agent will always propose the best move after evaluating the move until some depth. This will avoid the problem of a timeout during a search rendering the search useless. Additionally, given more time the agent will find better moves.

In order to search a larger part of the tree, we use *alpha-beta pruning* (Knuth and Moore [1975]). This strategy does not recurse on branches which cannot lead to a better result than the global maximum or minimum.

2 Agent Analysis

We compared our agent against the random and greedy player on *empty* and *easy* boards of size 3×3 with varying *turn duration* $\in \{0.1, 0.5, 1, 5\}$ in seconds. We repeated each configuration five times and then counted number of wins. Our agent performs as well or better if the win count is ≥ 2.5

Data from the experiments is reported in Figure 2. Our agent wins against against the random player in all but one experiment. That is where $turn\ duration = 0.1s$ and $board = Empty_{3x3}$. This may be explained by the short turn duration, which may not allow our agent to search the game tree to any depth. If this is the case, our player would essentially be playing random since it always starts by proposing a random legal move. The fact that the random player has more wins can be attributed to chance.

Our agent always wins from the greedy player on the empty board. On the other hand, our agent loses against the greedy player on the easy board except for $turn\ duration = 0.1s$. For this short turn duration our agent is likely playing randomly, so one explanation for its performance is that the greedy player is sometimes unable to provide a move within this time span. However, this would not explain why the greedy player still wins on the empty board. In general, we conclude that our agent does not perform better than the greedy player.

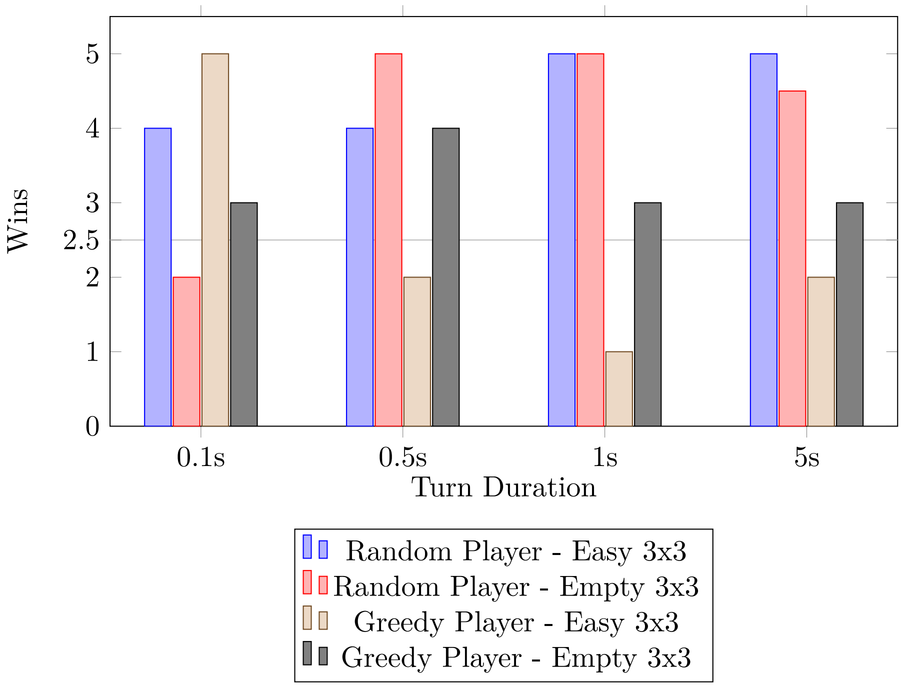

Figure 1: Wins plotted against the turn duration and opponent type. Results that are above the horizontal line at 2.5 imply winning the match while bellow represents loosing.

3 Reflection

The agent we implemented, based on the minimax algorithm with alpha-beta pruning and iterative deepening, has several strengths. First, the minimax algorithm ensures optimal search within the game space, choosing moves that maximize long-term gain or minimize losses. The implementation of alpha-beta pruning allows you to significantly reduce the number of search tree branches to be examined, thus reducing the computation time without compromising the quality of the chosen move. The iterative deepening strategy *partially* addresses the problem of the unknown time limit, allowing our agent to explore multiple moves without having to compute the entire game tree at once. Thus, we avoid the situation where our agent does not provide any move or misses an *easy* move (e.g. a move that produces win in a single turn). Finally, the agent uses knowledge of the game rules (row, column and block structure) to evaluate the best moves and earn points by completing regions.

However, the adopted approach also has some weaknesses. The major limitation is the computational complexity of Sudoku, especially for large boards. This significantly limits the reachable search depth in the game tree. For example, with a 3x3 board size, for a good part of the game the iterative deepening stops at depth 1, leading our agent to simply choose the move that causes a score increase, without evaluating the opponent's moves.

Furthermore, the *minimax* algorithm does not consider the opponent's strategy, always assuming that it makes the optimal choice according to the same criteria as our agent. The agent's effectiveness may decrease if it is faced with unexpected adversary strategies or if the adversary's strategy does not follow a predictable pattern.

In conclusion, the minimax approach with alpha-beta pruning and iterative deepening is a solid foundation for tackling the competitive Sudoku problem, but may have difficulty handling increasing computational complexity and predicting unconventional or unpredictable adversary strategies.

A possible solution that could improve the performance of the agent is the implementation of a transposition table. It would reduce the redundancy of the search, memorizing the positions visited during the search and avoiding the recalculation of the same positions in the future. Thanks to the computational time savings obtained in this way, the agent can further expand the search depth, leading to a more accurate evaluation of future moves.

However, it is necessary to take into account the memory footprint of the transposition table, which can be significant on very large grids. Furthermore, it is necessary to design an efficient hash function, which allows each game state to be uniquely represented, avoiding collisions, but at the same time which does not require too much time to be computed.

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.

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.

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

Python files

Code Listing 1: `sudokuai.py`.

```python
"""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]
```

```python
        # 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
        # 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
```

```python
        # 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)

        # 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 2: `utils.py`.

```
from typing import Iterator

from competitive_sudoku.sudoku import SudokuBoard, GameState, Move


def next_player(current_player: int) -> int:
    """Returns the next player."""
    return (current_player + 1) % 2


def region_range(
    *, row: int, col: int, board: SudokuBoard
) -> Iterator[tuple[int, int]]:
    """Return the range of cells in a region."""
    # TODO: Preprocess regions for faster lookup
    region_width, region_height = board.region_width(), board.region_height()
```

```python
    region_start = {
        "x": (col // region_width) * region_width,
        "y": (row // region_height) * region_height,
    }

    region_end = {
        "x": (col // region_width + 1) * region_width,
        "y": (row // region_height + 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, for_state: GameState) -> bool:
    """
    Returns whether a move is illegal.

    A move is illegal if it puts a duplicate value in a region, row or column.
    Illegal moves are not allowed and will result in a loss.
    """

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

    # Check if move is in taboo list
    # Idea: Hashmap would be faster (constant time lookup)
    if move in for_state.taboo_moves:
        return True

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

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

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

```
    )


def is_possible(*, move: Move, for_state: GameState):
    """Returns which moves are possible.

    All returned moves are on empty squares and not in the taboo list.
    Taboo moves are moves that would result in an unsolvable board and would thus be rejected.
    When making such a move, it will be added to the taboo list.
     Illegal moves are also removed.
    """
    return move not in for_state.taboo_moves and not is_illegal(
        for_state=for_state, move=move
    )
```