Competitive Sudoku: Assignment A1

1 General Outline

The goal of Assignment 1 is to implement a game-playing AI for the Competitive Sudoku game outlined in the `rules_full.pdf` document. Your implementation should use the α-β **pruning variant** of the `minimax` search algorithm to determine its moves. If you have not yet read the `rules_full.pdf` document explaining the rules of this game, then please start there.

In Assignment 2, you will be asked to iterate on this initial design, and to perform experiments which compare the different variations of your implementation, as well as conduct some analysis of, and reflect on, their performance.

2 Implementation Details

You are **not** expected to implement the full game by yourself; a code framework is provided by us, and you will only need to implement functionality that is specific to your agent. In particular, you need (and should) not worry about such details as keeping track of scores, functionality of the oracle, or loading files. All of this, and more, is already handled by the framework. Moreover, functionality is provided to perform experiments in which you can let your agent compete against other AI agents.

2.1 Getting Started

To get started, first ensure that you have already completed Assignment 0; if you have not done so, please go to Canvas and complete this first.

If you have completed Assignment 0, then at this point you should already have a working Python environment, as well as a folder called `team00_A0` in the root folder of your project. Please copy this folder, and name it "`team00_A1`", obviously replacing the '00' with your own team number.

To ensure that everything works correctly, you can try running

```
simulate_game.py --first=team00_A1
```

This command should invoke a game of competitive sudoku and this should run without errors.

2.2 Implementing your Agent

In the "`team00_A1/sudokuai.py`" file, write an implementation of the `SudokuAI` class's method

```
def compute_best_move(self, game_state:  GameState) -> None
```

The `game_state` argument provides all of the available information about the state of the game at the start of the AI player's turn; this is an instance of the `GameState` class, and you can find details in "`competitive_sudoku/sudoku.py`". In your implementation, you should use the `self.propose_move` method of the `SudokuAI` class to propose the move that you want your agent to play in the current turn.

In order to implement your agent, you must solve a number of (related) problems:

- You must implement a way to generate *all* the *legal* moves from a given game state / board position—that is, those moves that do not immediately cause the player to lose the game; see the `rules_full.pdf`. Note that lists of *taboo* moves and *allowed* cells are provided through the `game_state` argument.
- You must implement an evaluation function that assigns a numerical score to any game state. For instance, you might use the difference in points between the players, or you may develop more sophisticated methods.
- You must implement the α-β pruning variant of the `minimax` tree search algorithm, as the core logic used in the `compute_best_move` method. As possible extensions of this algorithm, you may also implement things like iterative deepening, quiescence search, and/or transposition tables.
- The `compute_best_move` method should be an *anytime* implementation; that is, the AI is not aware of how much time it has to execute, and it should be able to provide its current best move essentially at any time after the `compute_best_move` method was called. However, you do not really need to worry about how this is implemented; simply call the `self.propose_move` method of the `SudokuAI` class whenever you want to update your current assessment of the best move to make—you can do this as often as you like, and only the move proposed on the most recent call to this function will be used after the time for your current turn ends.

For strategic considerations as well as some technical notes on the implementation, see Sections 4 and 5 below.

3 Submission Details

Assignment 1 has **no mandatory submission**. However, if you want (and this is encouraged), you may submit your agent to already see how it performs against some benchmark agents.

To submit your agent, first do the following:

- Create a `zip` archive of the folder `team00_A1` that contains your implementation of the AI agent; as in Assignment 0, make sure you zip the *folder* and not only its contents.

The team member that is doing the submission should perform the following:

- Go to Canvas→Assignments→M1 and upload the `zip` archive containing your implementation to the Momotor tool.

The (mandatory) submission for Assignment 2 will follow the same workflow, and hence also submitting Assignment 1 is useful to get familiarity with the tool's interface, and to already catch any unforeseen issues with your code.

4 Strategic Remarks

The following are some strategic remarks that you may want to keep in mind when designing your AI agent:

- You may assume that the main program's logic provides sufficient time to, at least, find *some* legal moves for the current board position, and to call the `propose_move` method with an initial guess. However, you should not assume that you have enough time to search the game tree to any specific depth. Note that, if *no* move is proposed before the time for your current turn runs out, then the AI player automatically loses.
- You should *not* (yet) attempt to solve the sudoku puzzle to completion, and to then *only* expand the `minimax` search tree with moves that you are certain are ultimately part of the solved sudoku puzzle. The purpose of Assignment 1 is explicitly to search the tree of *all legal* moves; again, this will change in later assignments.
- You should *not* assume that the sudoku puzzle has a unique solution; in particular, starting the game from a completely empty grid is an anticipated scenario.

5 Technical Remarks

The following are some technical remarks that you should keep in mind when implementing your agent:

- For this assignment you can only use modules that are included in the Python standard library, as well as Numpy (version 1.22.3).
  - Be careful with this, loading modules costs time and needs to be re-done each time your agent is called.

- The provided code necessarily includes some functionality that is similar to what you will need to implement yourself for your agent; e.g., the implementation of the oracle requires checking whether a proposed move invalidates the puzzle. We have black-boxed this functionality as much as possible in the `bin/solve_sudoku` executable. Some enterprising students may also discover that the code for the provided "random" and "greedy" AI players invokes this executable. We urge you not to try and reverse-engineer it; we black-boxed it for a reason. In any case, your agent implementation is **not** allowed to invoke this executable; the logic of your agent should be your own implementation and self-contained in the python module that you need to submit.

- Your code should be generic, in that it should work for any sudoku grid with parameters $n, m > 1$; see the `rules_full.pdf` document for details. These parameters are provided through the `board` field of the `game_state` argument of the `compute_best_move` method.

- You should *not* attempt to store information between turns. Effectively, at every turn the main program's logic will create a new instance of the `SudokuAI` class, and the *only* information that it is allowed to use for that turn is provided by the `game_state` argument of the `compute_best_move` method.

- Your implementation should be single-threaded; that is, you may not create worker threads to parallelize your implementation. Similarly, for technical reasons we exclude the possibility for GPU-acceleration in this or later assignments.

Finally, the code that you submit is expected to be run on machines owned by us—the organizers of the course—and/or your fellow students. It should really go without saying, but let us note that you have a responsibility to make a reasonable effort that your submitted code does not have malicious side-effects.

Having the opportunity to share executable code allows us to design assignments where students can develop, express, and display their mastery in a topic, but it requires good-faith participation from all parties to do this in a safe manner. Please act accordingly.