Assignment A2 - Description

Lessons from the previous assignment

In your previous implementation, you may have noticed that expanding the minimax tree using all *legal* moves has several issues. In particular:

- Especially during the early and middle stages of the game, there are typically a lot of legal moves. This causes the minimax tree of legal moves to have a very high *branching factor*, and consequently it is difficult to search the tree to any significant depth. This also suggests that choosing a move during the beginning, middle, or the end of a game is not entirely the same problem.
- While there are many moves that are *legal*, it is also fairly common–especially during the middle stage of the game–that ostensibly good moves actually lead to an unsolvable position. This causes these moves to be rejected by the oracle, effectively wasting turns.

The current assignment — Basic idea

The purpose of this assignment is to give you an opportunity to demonstrate that you can find (e.g. in the literature) or invent AI methods that might be used to solve a complex problem; to demonstrate that you can design experiments to measure and compare the efficacy of such methods; and that you can reflect critically on the results of these experiments to choose the method best suited to solve the given problem.

Concretely, you are asked to find, develop, or invent one or more new features that can be added to your AI agent. You should then use your previous efforts (i.e. your A1 agent) together with the new functionality, to construct two or more agents. At least one of these agents should contain (a) new feature(s); another may for example simply be your A1 agent. You are tasked with designing and performing experiments to determine which of these agents performs best.

What follows are some suggestions that you may explore for this assignment; you are free to choose one or several of them, and/or to use something else entirely.

Potential Approach — Heuristics on top of Alpha-Beta Pruning

One potential approach to solve the above issues, is by using *heuristics* to (attempt to) focus the search on only a subset of legal moves. You are relatively free in choosing the concrete heuristics and how you implement this, but the main idea would be to (i) focus on exploring moves that are guaranteed to be part of the sudoku puzzle's final solution, and/or (ii) focus on moves that you have a heuristic reason for making *outside* of the normal minimax evaluation (or which may be difficult to encode in the evaluation function directly).

Note that you do *not* need to solve the entire sudoku puzzle exhaustively in order to find such moves (although you could, indeed, do that). For instance, you could search for empty cells such that there is only a single value that does not violate the basic constraints (value not already in block, column, or row) and that has not been declared taboo for this cell; because such values are the *only* legal move for their associated cells, they are necessarily part of the final puzzle's solution. Generally, you can implement any rule-based (pattern-matching) strategies and heuristics that human players use when solving normal sudoku puzzles. A lot of guides on this topic can be found online; we suggest e.g., https://www.sudokudragon.com/sudokustrategy.htm , for an overview of different strategies.

Apart from focusing on solving sudoku, in our problem there are also opportunities for strategic play, so using heuristics to take advantage of the competitive nature of the game. This might take the form of e.g. trying to play in certain cells with the intent to "trap" the opposing player, and de-prioritizing short-term points-scored in doing so. To achieve this, you might consider developing

some rule-based logic to determine, at any given point in the game, whether to follow the minimax-evaluation, or some (say) positional logic derived from e.g. boundary-detection heuristics that seek to classify the opponent's current "territory".

To be explicit about this, there are essentially two aspects of the game that this assignment lets you explore using heuristics and/or rule-based methods. On the one hand, you can seek to reduce the branching factor of the minimax tree, so as to improve the effective search-depth you can achieve. To this end, you may implement heuristics and/or rule-based methods to focus on cells and values that you know won't be declared taboo by the oracle; this prunes away a (potentially) large number of "useless" nodes from the minimax tree. On the other hand, you may develop rule-based methods and heuristics that focus on different strategic elements, such as the capture of territory. Such approaches do **not** need to be incorporated in your minimax algorithm itself; you can implement high-level decision rules that switch between using these strategic heuristics and minimax search, to determine the next move. (And, if you "like" several moves for strategic reasons, you can still try to use minimax evaluation to choose between just those moves!)

Especially for the first point, it is still important to explore moves which are not guaranteed to be part of the final solution; for instance, in certain cases it may not be possible to find *any* moves that are guaranteed to be (ultimately) correct. You should implement some kind of heuristic to balance this; for instance, you could randomly sample several generically legal moves that you explore, in addition to those moves that you know to be correct. Of course, you are free to develop more sophisticated strategies for this as well.

Moreover, you may notice that after having implemented several heuristics for generating moves that are guaranteed to be ultimately correct, that even only expanding those moves (again) leads to a high branching factor. As such, you may want to implement additional heuristics to prioritize between different such moves.

Potential Approach — Game Abstraction Levels

As a generic suggestion, the game of competitive sudoku has several axes along which one might simplify or abstract the game, to reduce the computational/combinatorial burden. This might take the form of focusing on particular aspects such as purely positional play, partial region completion, player turn order, move reduction/constraint introduction, or others; and then only re-introducing the full game complexity after decisions have been made based on these higher-level ideas.

Potential Approach — Alternative Search Strategies

Another angle you can explore---instead of, or in addition to, the heuristic suggestions above---would be to use a different search method than (alpha-beta pruning) minimax. Concrete suggestions along these lines are:

- A learned evaluation function, for instance implemented using a (small-ish) neural network, taking as input the current board state and/or some heuristically derived properties that you think might be useful. This can be combined with a shallow search-depth minimax search to find good moves, while reducing issues with the high branching factor. This neural net might for instance be trained using TD-Learning (discussed in Reading Assignment 2).
- As a completely different search strategy, you could explore Monte Carlo Tree Search (discussed in lecture L9) instead of, or in combination with, any of the above methods.
- Something else entirely that you think is interesting and might be a good idea.

Changes as compared to Assignment 1

- *You may now use the provided functionality in the code bundle to persistently save and load a data structure across turns.* Take into account that this functionality also costs time;

however, we have ensured that the save/load actions will complete, even if time runs out while you're saving or loading (if time does run out, the turn will end immediately upon finishing the save/load, so you won't then have the opportunity to use the data during that turn). This means that you do not need to worry about dealing with files in which only part of the data from the previous turn was saved (i.e. these are thread-safe implementations). The provided **random_save_player** agent displays the use of this functionality in its *compute_best_move function* (in short, use the new *self.save* and *self.load* functions of the SudokuAI base-class).

- You now also have more freedom to use packages beyond standard Python: namely, Numpy and generally any package in the description of the Anaconda installation on Momotor is an option. However, do not use functionality not normally needed for the assignment! Specifically, any IO code apart from the one provided in the bundle (see previous point) may not be used. (Further restrictions up to our discretion.)
  Be careful with this, loading modules costs time and needs to be re-done each time your agent is called.

Experimental Comparison of your Agents

You should design and execute a systematic comparison of your two (or more) agents, to determine which is the stronger player. Things to keep in mind during this:

- Games are typically played on the "3x3" variant boards (so actually, 9x9 total grid size), starting from an empty position, with "think time" between 1-10 seconds per turn. Which player gets to start is likely relevant for game outcome.
- You should compare the performance of your agents both against each other, as well as against the provided Greedy player.
- The provided **play_match.py** script can be used as a starting point for your implementation of these experiments.
- In our experience, variance in performance can be relatively high between games, so ensure that you run enough tests to be confident in your conclusions. For this reason, performing some type of statistical testing is highly encouraged (see below on assessment).

Submission and Assessment of the Assignment

As previously discussed, your work on this assignment will be assessed in two separate ways. What follows are some details on this.

As you finalize your work on the development and evaluation of your agents, you need to make a choice about which agent you submit; our overall suggestion is to submit the strongest player that you have managed to develop, unless you have good reasons to do otherwise (e.g., compute limitations in the automated assessment part). Regardless, you should be able to articulate your motivation for this decision.

Note that it is possible that, despite all your efforts, you may not have been able to improve on your A1 agent; in this case you are allowed to use your A1 agent as your final submission.

The first assessment is on the *basic correct functionality* of your agent. For this part, you submit your (final) agent design in a similar way as in A0 and A1, and this will get assessed on Momotor using the following scheme:

- After a zip file check and a check whether your code returns valid moves (on arbitrary board sizes, not just 3x3!), your agent will again be tested using Momotor against three opponent agents. For each of these, points are awarded based on the total number of games won, identically to the scoring in Assignment 1. The tests are the following (*for technical reasons, this is subject to later change on the last two opponents):
    1. Play six games against the greedy player.
    2. Play six games against a basic player of our design.
    3. Play six games against a stronger player of our design.
- You receive feedback on the performance of your agent "immediately", that is, once Momotor has finished running the games. Both due to the actual wall-clock time involved in playing these games, as well as the high number of potential concurrent submissions, **we recommend submitting well in advance of the deadline.**
- You can resubmit an "unlimited" number of times, until the deadline closes at **Friday 9 Jan 2026 21:00h.** This deadline is *hard*, so again, please make sure your actually submit well in advance. If locally your agent performs consistently better than it did in your submission, it's possible that you hit a streak of bad variance in the evaluation; in that case you can resubmit the same agent as before to force another evaluation. Your (team's) *last* submission is the one that counts, so make sure you coordinate within your team so that a working submission does not get "overwritten".
- The pass/fail assessment that counts for 10% of your final grade, only requires you to **score 5 or more points against the greedy player**. Normally, a correct alpha-beta pruning minimax implementation (like you made in A1) should be able to do this. Performance against the other agents only serves to give you a baseline sense of the strength of your agent.
- Your submitted agent is the one that is used in the (non-graded) end-of-course tournament against all other teams' submissions. As a point of reference, in previous years the well-performing student teams in the tournament were able to consistently beat both of the other agents used in the Momotor tests.