SNIoE
CSD311 (2023) Course Project

Note: For problems in either domain (single-agent or adversarial):

1) You can not use libraries that directly address the project problem, that is, while numpy is allowed for matrix operations or matplotlib is allowed for graphing data, you can not for example use emulator libraries for the game you are solving. State generation and so on should be hand-crafted.

2) Keep in mind that plagiarism-checkers index github and other public code hosts. All blatant plagiarism will be heavily penalized.

3) A minimal graphical representation of the game state is expected. Doesn't need to be fancy, just has to make it clear the game is being played correctly.

Evaluation will be the average of
- Report
- Visualization
- -Plagiarism
- Ability to explain methods used
- Solution Performance (Details vary between single-agent and adversarial; refer to the following pages)

Note:
- Competition simulation code for single- and double-agent problems will be posted and updated on github.com/amancapy/csd311-material

- For relative performance of teams (see following), to avoid premeditated equal performance of teams' agents, evaluation will be based on "% above 50%" of win-rate.

- You will be informed of any changes to the writeup.

Two-agent (Adversarial) Problems

Agents (solvers) written by teams that are assigned the same project problem will compete so that their relative performance can be assessed. A large number of games will be simulated between the agents to make this assessment, so there is some standardization required of the teams:

Teams will collaborate on programming the state-generation function that will be common to all teams with the same project. This function, as in your 8-tile assignment, takes a state s, and returns the list of possible immediate future states that one can arrive at from this state. For example, 123456780 would return [123456708, 123450786] in 8-tile. Or the starting position of chess would return all 16 possible pawn moves, and 4 possible knight moves. Each `state` object also typically needs to store additional information about the state. For example, castling rights, en-passant, etc. in chess. This function must also return whose turn it is next, and also adhere to the official rules of the game and/or those laid out here.

Both teams have a fixed amount of time with which to play the entire game. Timeout is a loss (barring certain conditions that are game-specific), as is standard in human play.

The two teams between whom games are simulated, team_A and team_B, will produce one function each, agent_A and agent_B, which will consume the list of future states; the output of the common *get_possible_states(...)* function, and produce an index into the list, for the state they wish to pursue i.e. the move they wish to play.

Pseudocode is laid out in the next page.

One game-round will look something like this:

```
time_A = Timer()
time_B = Timer()

start_state = ...
(future_states, A's_turn) = get_possible_states(start_state, ...)
# `A's_turn` is a boolean signifying whether it is A's turn to play
# the next move.

end_condition = False
while not end_condition:
    if A's_turn:
        from team_A import agent_A

        time_A.resume()
        choice_index = agent_A.choose_from(future_states, ...)
        chosen_state = future_states[choice_index]
        time_A.pause()
        del agent_A

        if check_end_conditions(time_A, time_B, chosen_state, ...):
            end_condition = True
            # update score
            break

        (future_states, A's_turn) = get_possible_states(chosen_state, ...)


    else:
        from team_B import agent_B

        time_B.resume()
        choice_index = agent_B.choose_from(future_states, ...)
        chosen_state = future_states[choice_index]
        time_B.pause()
        del agent_B

        if check_end_conditions(time_A, time_B, chosen_state, ...):
            end_condition = True
            # update score
            break

        (future_states, A's_turn) = get_possible_states(chosen_state, ...)
```

Note that the teams will also collaborate on producing the *check_end_conditions(...)* function along with *get_possible_states(...)* for the sake of consistency, as they will use both in their search functions anyway.

Also notice that in this flow, one agent isn't allowed to think during the other agent's turn. This is an oversimplification of real AI competitions, as allowing for this would require teams to understand parallelism, which would be a steep learning curve for those new to programming.

All computation by agent_A ends during agent_B's turn and vice-versa. Essentially every turn your agent wakes up fresh.

Additional notes:

- If any of these games is deemed too trivial, we might increase complexity in terms of board size, etc. later. You will be informed in advance.

- These problems essentially boil down to the minimax algorithm, and alpha-beta pruning will be crucial in making it tractable. Both are straightforward, so the success of your work will come down to your evaluation functions, and the efficiency of your computations.

- Teams can make special requests for changing the pseudo to meet their game's requirements and dynamics. In general the details in the pseudo are subject to change. For instance, the *A's_turn* logic is only relevant to dots-and-boxes and othello, since players can make multiple moves in a turn there. This flow can be removed for single-move games like the rest of them.

Minichess 6x6

Standard chess is a difficult problem to address. This is a simplified version where the game is played on a 6x6 board instead.

Random-Chess66 differs from standard chess in a few ways:

- No castling
- No en-passant
- No pawn two-step from home square

To make the simulated games less repetitive, the home-rank will be randomized every game, with 5 of {1Q, 2R, 2B, 2N} chosen and placed randomly, along with 1K. The placement will be symmetrical for both players.

The conditions for win/loss/draw are identical to those in standard chess:

- win/loss: checkmate/timeout
- draw: stalemate/both sides out of pieces that can deliver checkmate/timeout when opponent has no pieces that can deliver checkmate/3x-repetition of a position.

The backrank-randomization rule is a generalization of Los Alamos Chess, the first chess variant ever for which an AI agent was developed.

Othello

This game has to be played with standard rules on the standard 8x8 board. Refer to the official rules and conditions.

Scrabble

Scrabble too will be standard.
- The game is played on a 15x15 board.
- Each team is provided with a rack of 7 letter tiles drawn from a pool of English alphabet tiles (A-Z).
- The objective is to create valid words on the board using the tiles from your rack.
- Words can be placed either horizontally or vertically on the board, with the first word placed on the center square. New words can branch off only from existing letters, and all new strings that form during a player's turn must be valid words.

Go 13x13

13x13 Go is the medium standard board size, between 9x9 and 19x19. Two players, black and white, take turns placing one 'stone' each on the board. Whenever a contiguous chunk of stones of the same color is surrounded by the other color's stones, including the edge of the board, the whole chunk is captured, and as many points as stones captured are awarded to the capturer.

Refer to the official rules about Go; they apply identically to all known board sizes.

All rules are as stipulated officially.

Dots-and-Boxes 10x10

The game begins with a grid of dots laid evenly. Players take turns connecting any horizontal or vertical adjacent pairs of dots, forming lines. When all 4 sides of a square of dots are formed, the player that formed it claims the square, and must play an additional turn. This aspect of having to play an additional turn is the source of difficulty in this game. The game ends when all possible lines are already formed, and the player with more claimed squares wins.

This game too is well-documented.

Connect-N

This is a generalization of the classic Connect-4. However, connect-4 is a solved problem and somewhat trivial, so your program will have to generalize to not only connect-N, but also varying board sizes. Since the branching factor of this game depends only on the #columns of the board, your program is expected to work for fairly large parameters of N, width, and height.

Single-Agent Problems

Like in the two-agent problems, teams here too are required to standardize their playing field. This amounts to collaborating on producing the state-generator function, which given a current state of the game/board (along with any necessary additional information), produces the list of all possible future states, just as how in 8-tile, 123456780 gives you [123456708, 123450786].

The teams will also need to communicate about standardizing the rules of the game, and random start-state initialization like in 8-tile. For some games, it might be required to use existing datasets for standard start states.
All else has to be separate work.

The evaluation of your work will be based on your agent's performance on a large number of simulations from different start states, relative to the agents of the other teams working on the same project. This will include solution time, solution brevity, and other factors if available.

As in two-agent, the rules here are subject to change if the teams make reasonable requests. There might also be changes that are game-specific.

Sudoku 25x25

Sudoku is a logic-based, combinatorial number-placement puzzle. The objective is to fill a 25x25 grid with letters so that each column, each row, and each of the 5x5 subgrids that compose the grid contains all the letters from A to Y. This is just a larger version of the classic 9x9 Sudoku.

Minesweeper 16x30

Minesweeper is generally played on personal computers. The game features a grid of clickable tiles, with hidden "mines" (depicted as naval mines in the original game) scattered throughout the board. The objective is to clear the board without detonating any mines, with help from clues about the number of neighboring mines in each field. When you click a square, three things can happen. You can click a mine, resulting in an instant loss. You can click on a square next (diagonal included) to at least one mine, resulting in a number being shown under the square.
Or you can click on a square that isn't a mine and isn't next to a mine. When this happens, all 8 surrounding squares are revealed. If any of those squares are also not next to any mines, all of the surrounding squares that are not yet revealed become revealed.

Rubik's Cube NxN

This is entirely standard. N will be a reasonable number during evaluation.

Snake 15x15

This is the standard classic, where a "snake" and its body move around a 2D grid, and the snake gets one cell longer every time it eats a food cell. The objective is to make the snake as long as possible in a minimal number of moves.

65536 4x4

65536 is a single-player sliding tile puzzle game which is a variant of 2048. It is played on a plain 4×4 grid, with numbered tiles that slide when a player moves them using the four arrow keys. The game begins with two tiles already in the grid, having a value of either 2 or 4, and another such tile appears in a random empty space after each turn. Tiles slide as far as possible in the chosen direction until they are stopped by either another tile or the edge of the grid. If two tiles of the same number collide while moving, they will merge into a tile with the total value of the two tiles that collide. The resulting tile cannot merge with another tile again in the same move.

The objective, as in 2048, is to primarily try and make a 65536 tile, or secondarily to at least maximize the largest tile.

Kakuro 30x30

Kakuro is like a crossword puzzle with numbers. It is played in a grid of filled and barred cells, "black" and "white" respectively. Apart from the top row and leftmost column, which are entirely black, the grid is divided into "entries"—lines of white cells—by the black cells. The black cells contain a diagonal slash from upper-left to lower-right and a number in one or both halves, such that each horizontal entry has a number in the black half-cell to its immediate left and each vertical entry has a number in the black half-cell immediately above it. These numbers, borrowing crossword terminology, are commonly called "clues". The objective of the puzzle is to insert a digit from 1 to 9 inclusive into each white cell so that the sum of the numbers in each entry matches the clue associated with it and that no digit is duplicated in any entry. There is another rule for making Kakuro puzzles that each clue must have at least two numbers that add up to it since including only one number is mathematically trivial when solving Kakuro puzzles.