Team 30: Taron Moe-Stull (taronmoe@icloud.com)

Sudoku Solver

System Requirements

For Project 1, our job is to build a Sudoku solver in Python, delivered as a Jupyter notebook. I will drive everything from a single Parameters cell at the top so the user can swap puzzles algorithms without touching our code. Our exact parameters will be as follows:

```
GROUP_ID = "Group30"
ALGORITHM = "bt" one of:  bt, fc, ac3, sa, ga
PUZZLE_TYPE = "easy" one of:  easy, medium, hard, extreme
PUZZLE_PATH = "puzzles/easy/puzzle01.txt"
```

Input format: I will read a plain .txt file from PUZZLE_PATH that represents a 9×9 Sudoku in row major, comma delimited format. Unknowns are ?. I will validate that it's exactly 9×9 comma separated entries and each token is either ? or 1 through 9.

Output format: After solving, I will write the completed grid in the same comma delimited shape (no ? left). The filename will follow the required pattern and be saved in the notebook's working directory:

`[GROUP_ID]_[ALGORITHM]_[PUZZLE_TYPE]_[PUZZLE_NUMBER].txt`

Example: `Group30_fc_easy_puzzle01.txt`

Algorithms I will implement: Our solver framework will support five variants and switch between them using ALGORITHM:

- bt – simple backtracking (run only on easy and medium)
- fc – backtracking with Forward Checking
- ac3 – backtracking with Arc Consistency (AC-3)
- sa – Simulated Annealing
- ga – Genetic Algorithm

No hardcoding: the solver will read PUZZLE_PATH and ALGORITHM from the parameters and write exactly one output file, .txt only.

How I will measure performance: Instead of timing, I will count the main decisions each algorithm makes (this is our performance metric). For BT, FC, AC-3 I will track things like assignment attempts and backtracks (and, where relevant, arc revisions). For SA I will track neighbor evaluations (and moves accepted). For GA I will track fitness evaluations (and can report secondary stats if it seems informative while completing). Note: I will not record CPU or wall clock time.

Why the strict I/O and parameters: The user will run our notebook on held out puzzles I haven't seen. Standardizing the input, output, and parameters guarantees they can plug in any puzzle path or algorithm and our solver will run cleanly.

Design Document: The Design Document is a document I will use as a reference throughout the project that includes the project requirements, system architecture, system flow, test strategy, as

well as task assignments and schedule of Project 1. The Design Document may be used as a reference for groups as the project progresses to check plan of attack, due dates, workload distribution, and project structure.

Paper: Following the project I will summarize the results of our experiment and explain the experimental setup, any hyperparameter tuning and the final parameters for each algorithm. The paper includes: Title and author names, a brief, one paragraph abstract summarizing the results of the experiments, problem statement, including hypothesis, description of algorithms implemented, description of your experimental approach, presentation of the results of your experiments, a discussion of the behavior of your algorithms, combined with any conclusions you can draw, summary, and references of any external sources used in the completion of the project or paper.

System Architecture

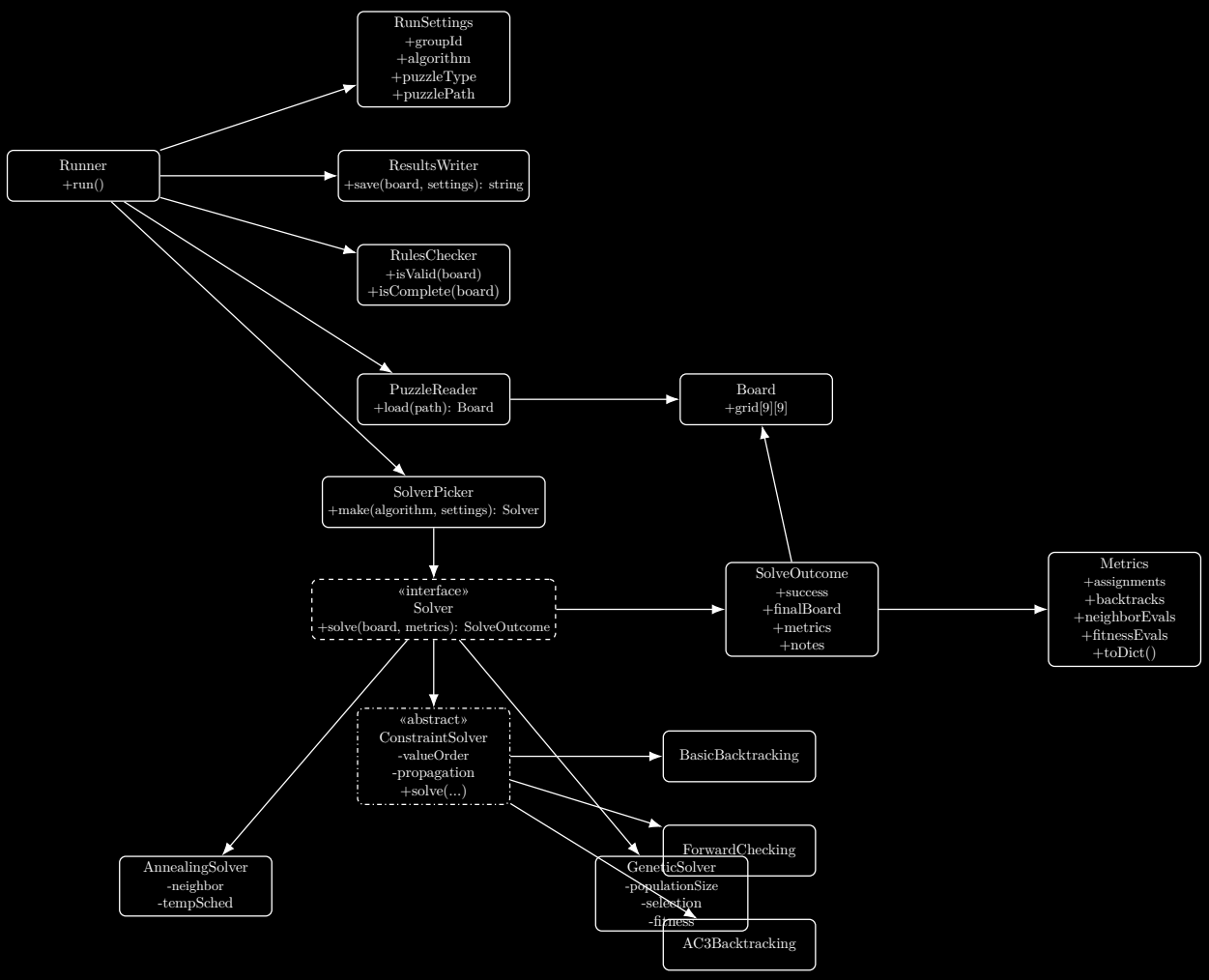

Figure 1: System architecture and solver hierarchy for the Sudoku solver framework.

Note: All subject to change as of right now

Runner: Runs the pipeline: read settings then load puzzle then pick solver then solve then validate then save then report metrics. Inputs: RunSettings, puzzle file. Output: SolveOutcome, saved .txt.
RunSettings: Holds groupId, algorithm, puzzleType, puzzlePath.
PuzzleReader: Parses and confirms the 9×9 comma delimited .txt with ? Output: Board
Board: In memory 9×9 grid
RulesChecker: Checks for interim and final boards. Outputs booleans.

Metrics: Decision counters (assignments, backtracks, neighborEvals, fitnessEvals). Sends to toDict()
SolverPicker: Maps algorithm and other settings.
Solver (our interface): solve(board, metrics) Outputs: SolveOutcome for any algorithm.
SolveOutcome: checks success status, finalBoard, metrics, and notes?
ConstraintSolver (abstract *just means no instances for class): Generic Backtracking solver that chooses between consistency modes: valueOrdering, consistency (None, Foreward Checking, AC-3). Implements solve.
BasicBacktracking: Plain Backtracking
ForwardChecking: Backtracking and filtering after each puzzle
AC3Backtracking: Backtracking and arc consistency using a queue?
AnnealingSolver: Search solver, starts from a complete board respecting constraints and makes small neighbor changes and accepts if it reduces conflicts with constraints.
GeneticSolver: Population solver fills boards consistent with the constraints and scores them by penalty until penalty is reduced and continues to try swaps until ideally no constraints are crossed.
ResultsWriter: Writes solved board as comma delimited .txt, filename: `[GROUP_ID]_[ALGORITHM]_[PUZZLE_TYPE]_` in the notebooks working directory.

System Flow

Ideally our notebook runs a single, repeatable pipeline: read parameters, load a puzzle, pick the requested solver, solve while counting decisions, validate the result, and save the output in the required format.

1. Start & Settings. The run begins in the Parameters cell. I load GROUP_ID, ALGORITHM (bt, fc, ac3, sa, or ga), PUZZLE_TYPE (easy, medium, hard, or extreme), and PUZZLE_PATH. These settings are the only source of configuration. Note: no hardcoding.

2. Read & Validate Input. System opens the comma delimited 9×9 .txt file at PUZZLE_PATH, validates shape and tokens (? or 1–9), and constructs an in memory Board. If the format is invalid, the run halts with some error message.

3. Checks & Prep I check that all givens are consistent (no row, column, box conflicts).

4. Pick a Solver. Based on algorithm selection, the pipeline selects one implementation:

   - Bt, fc, ac3
   - sa (simulated annealing, local search)
   - ga (genetic algorithm, population search)

5. Initialize Metrics. A shared Metrics object starts at zero to track the decision based performance counters (this could be assignments, backtracks, neighbor and fitness evaluations). Every algorithm increments these at the moment work happens.

6. Solve Step (by algorithm?).

   - Constraint search (bt, fc, ac3): The solver selects an empty cell, tries values, and checks Sudoku constraints. fc prunes for unsuccessful trees?; ac3 enforces arc consistency. On dead ends, I backtrack. Metrics record variable selections, assignment attempts or commits, arc revisions, and backtracks.
   - Simulated annealing (sa): The solver starts from a filled board consistent with givens, proposes small neighbor changes, and accepts moves by a standard score of inconsistencies with constraints. Metrics record neighbor evaluations (and maybe moves accepted or restarts).
   - Genetic algorithm (ga): The solver fills a puzzle respecting givens, evaluates fitness, and iterates through selection then "crossover" (choosing where to swap) then swaps. Metrics record fitness evaluations.

7. Result: Each solver returns a SolveOutcome containing a success bool, the final board (if solved), the metrics, and any notes (like resets or settings).

8. Validation. Before writing output, I verify the board is complete and valid (each row, column, box contains 1–9 with no duplicates). Failure here cancels the write and produces some error message; metrics are still available for debugging and for us.

9. Write Output. On success, I save the solution as a comma delimited 9×9 .txt using the exact filename pattern `[GROUP_ID]_[ALGORITHM]_[PUZZLE_TYPE]_[PUZZLE_NUMBER].txt` in the notebook's working directory.

10. Report Metrics. The run prints the decision counts so I can compare algorithms and difficulties later on.

11. Error Handling. If an algorithm fails to solve within its limits, the system reports failure cleanly (again some specific error message) and importantly still prints metrics.

Test Strategy

End to end (as simple as possible)
I start by setting the parameters at the top of the notebook and load in the puzzle file. If the file is wrong in any way, shape, characters, file path, the run stops and displays an error message so I don't have to traverse the entire program.
If the file is fine, the system picks the solver named in the parameters and starts it. From here, the solver does its work to fill in the puzzle. While it runs, I keep track of our metrics so I can compare and assess efficiency or change something where needed.
When the solver is finished I check the board against normal Sudoku rules like rows columns and each 3x3 box must have the numbers 1 through 9 no repeats. If the board passes the check, I save

the solution as our comma delimited text file using the exact required filename. If it doesn't pass I will again display an error message so I know exactly where we're messing up.
That's our whole flow with all the requirements in the big picture: parameters, load puzzle, run the solver based on selection, verify the result, then either save the solution or fail somewhere along the way. The metrics will be recorded along the way so I can compare runs and adjust where I need to.

Schedule overview

- Phase 0 — Design doc: finalize Requirements, Architecture, Flow, Test Strategy. (8/22-8/29)
- Phase 1 — Setup: Parameters, reader and writer, basic validator, solver picker. (9/2-9/4)
- Phase 2 — Constraint search: BT then FC then AC-3 (hook in metrics). (9/5-9/7)
- Phase 3 — Local search: SA and GA, each with metrics.(9/5-9/7)
- Phase 4 — Integration: run end to end on all provided puzzles and verify output format or naming and fix bugs. (9/7-9/9)
- Phase 5 — Experiments & Final Touches: compare methods, and prep notes for the paper. (9/10-9/11)

Reference

1. GeeksForGeeks "https://www.geeksforgeeks.org/dsa/sudoku-backtracking-7/" (8/22) Trying to understand bt
2. GeeksForGeeks "https://www.geeksforgeeks.org/artificial-intelligence/constraint-satisfaction-problems-csp-in-artificial-intelligence/" (8/22) Trying to understand constraint
3. GeeksForGeeks "https://www.geeksforgeeks.org/artificial-intelligence/constraint-propagation-in-ai/" (8/23) Trying to understand constraint and propagation
4. GeeksForGeeks "https://www.geeksforgeeks.org/artificial-intelligence/what-is-simulated-annealing/" (8/25) Trying to understand sa
5. GeeksForGeeks "https://www.geeksforgeeks.org/dsa/implement-simulated-annealing-in-python/" (8/25)
6. GeeksForGeeks "https://www.geeksforgeeks.org/dsa/genetic-algorithms/" (8/26) Trying to understand ga
7. GeeksForGeeks "https://www.geeksforgeeks.org/artificial-intelligence/introduction-to-optimization-with-genetic-algorithm/" (8/27) Trying to understand ga
8. GeeksForGeeks "https://www.geeksforgeeks.org/dsa/backtracking-algorithms/" (8/27) Trying to understand bt
9. MermaidLiveEditor "https://mermaid.live/edit/" (8/27) For the Class diagram