OLAMPI Terry (to4008)
GARIMELLA Thribhuvan
Edinburgh CW 1 42

F29AI Coursework 1

Part 1: Solving and Analyzing Sudoku with Search Algorithms

1A :

Standard 9x9 Sudoku. We can establish 81 variables (one per cell), each within a domain of size 9, and empty (0) initially. The constraint enforces that every row, column and 3x3 subgrid contains all digits exactly once. Formally :

Let A[i][j] represent the value in the given sudoku cell at row i and column j.

Row Constraints: $\forall i \in \{1,2,\ldots,9\}, \{A[i][j] \mid j = 1, 2,\ldots, 9\} = \{1, 2,\ldots, 9\}$.
Column Constraints: $\forall j \in \{1,2,\ldots,9\}, \{A[i][j] \mid i = 1, 2,\ldots, 9\} = \{1, 2,\ldots, 9\}$.
Subgrid Constraints: Each subgrid (3×3) checks repeating numbers. Hence we define the variables indexed by (k, l) where k and l range from 0 to 2 (3×3 subgrids). We start at (3k+1, 3l+1) and end at (3k+3, 3l+3) so that all must contain each and every digit from 1 to 9.

Thus, for all $k,l \in \{0,1,2\}, \{A[3k+m][3l+n] \mid m,n \in \{1,2,3\}\} = \{1, 2,\ldots, 9\}$

This means that the blank (0) cells are variables, and enforce a kind of “all different” constraint on rows, columns and sub-blocks.

Comparison of brute-force search vs. backtracking in Sudoku :

Brute-force search simply means trying all possible combinations systematically until the correct solution is found.
The brute-force approach for solving Sudoku would involve trying all possible numbers for each empty cell, one by one, and checking if the current configuration is valid (i.e., no rule violations). If a number doesn't work, the algorithm backtracks to the previous decision point and tries a different number. This is essentially trying every permutation of the Sudoku grid.
Time Complexity: In the worst case, it would try all possible number combinations for each cell, leading to an exponential time complexity. A standard Sudoku puzzle consists of a 9×9

grid, which has 81 cells. Each cell can contain one of 9 possible values (from 1 to 9).
The time complexity would thus be O(9^81). This is extremely large and clearly impractical for even small grids like standard Sudoku.

Backtracking is a more efficient form of brute force. It is a search technique that systematically searches for a solution while pruning out large portions of the search space that cannot possibly lead to a valid solution.

The idea is to fill in numbers one by one, and each time a number is placed, the algorithm checks if the current state of the grid satisfies the Sudoku constraints (i.e., no repeated numbers in rows, columns, or subgrids). If a conflict occurs, it backtracks by removing the last placed number and trying a different one.
While backtracking is still an exhaustive search algorithm, it typically performs better than brute-force because it prunes infeasible paths early. The exact time complexity depends on how much the algorithm is able to prune, but in the worst case, it's still exponential.
In the worst case, backtracking might have to try all possible assignments for each empty cell, just like brute-force.
Worst-case Complexity: O(9^81), similar to brute-force in the worst-case scenario.
Average-case Complexity: The backtracking algorithm performs much better in practice because it explores fewer paths. This is due to early termination when a conflict is detected, making the algorithm much faster than brute-force.

So, while both are brute-force methods at their core, backtracking tends to be far more efficient in practice due to pruning of infeasible solutions and use of heuristics to explore the search space intelligently.

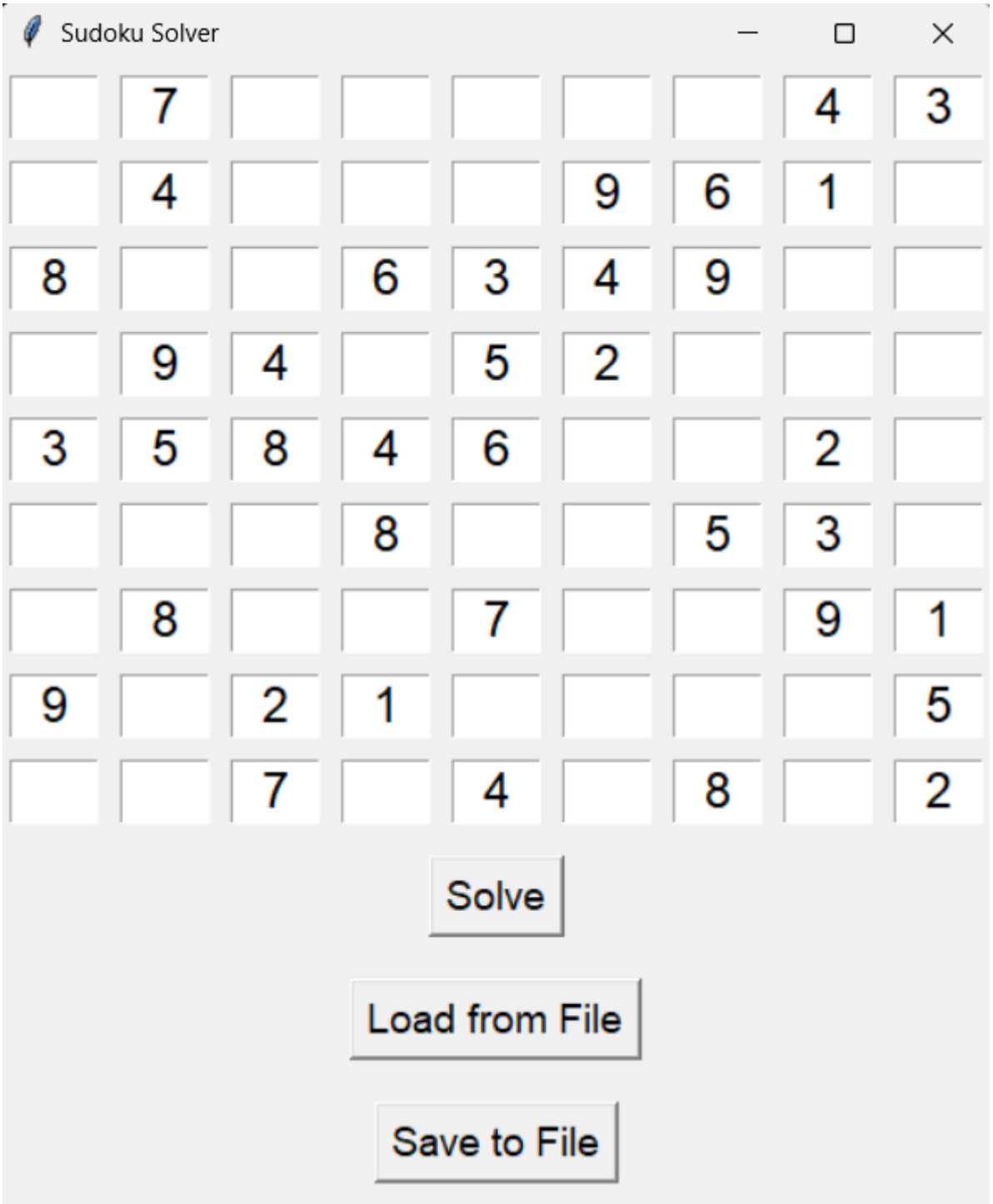
1B :

We used python for every coding-related question.

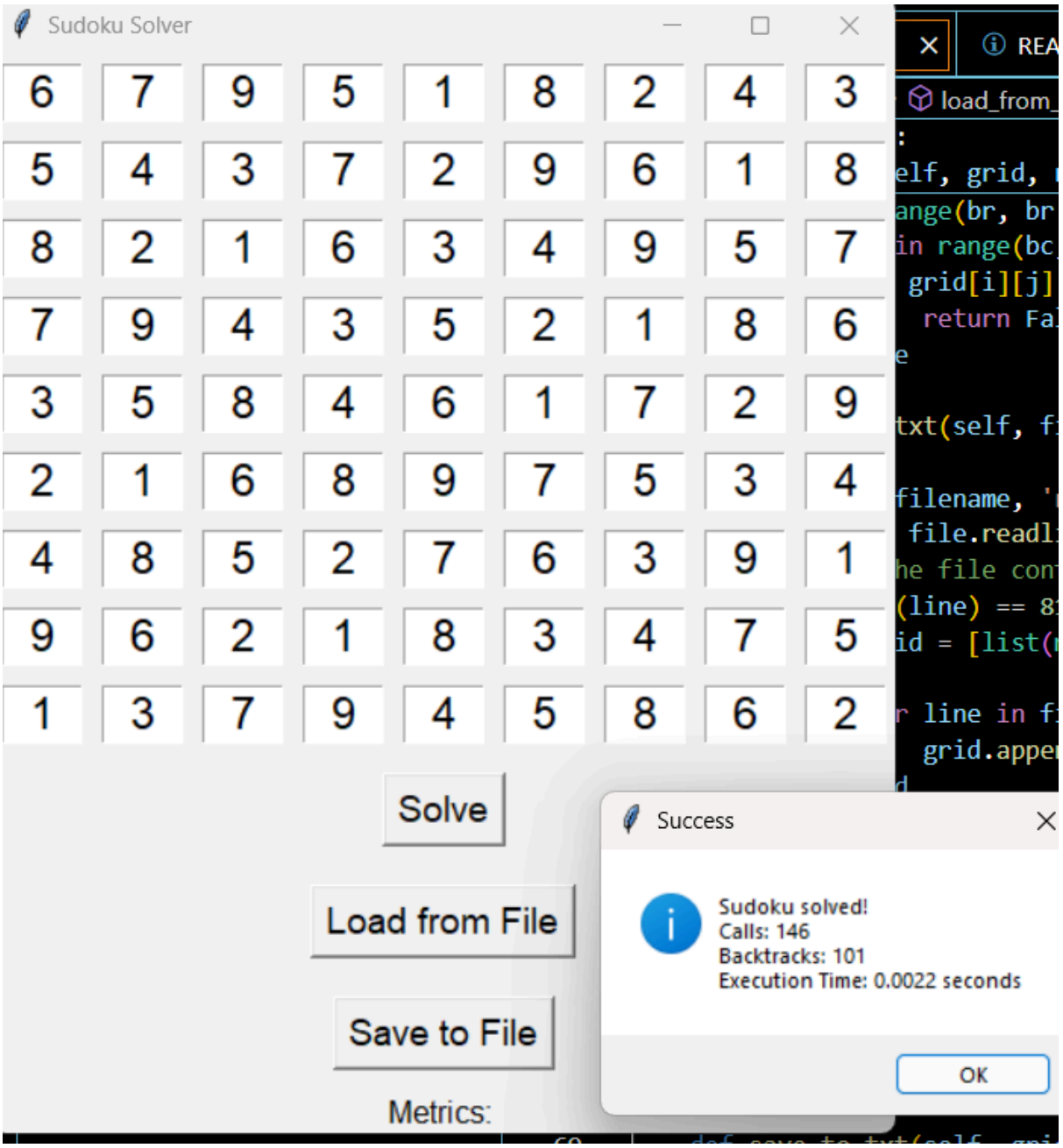
We implemented a depth-first backtracking search, with CSP heuristics. At each step we select an unfilled cell and try values 1–9 that are consistent with current constraints. If no value works, we backtrack (undo the assignment).

The solver reads an input grid (e.g. from a .txt or .csv where blanks are 0) into a 9×9 list of lists and calls solve().
Metrics like total recursive calls and backtracks are reported.

Before solving :



After being solved :



Our Sudoku solver utilizes a backtracking algorithm that exhibits exponential time complexity in the worst-case scenario. Specifically, in the extreme case, the solver attempts all possible assignments for m blanks, leading to O(9^m) possibilities, with n = 81 for an empty grid. In general, the backtracking approach has a time complexity of O(n^m), where n is the number of choices per cell (9), and m is the number of unfilled cells.

However, the algorithm's practical performance is significantly improved through constraint propagation. For instance, in the implementation, the is_safe() method ensures that a number is placed only if it does not violate the row, column, or 3x3 block constraints, pruning infeasible branches early on. Additionally, by using a depth-first search (DFS) approach, the solver efficiently explores only promising assignments, drastically reducing the search space in typical 9x9 puzzles.

Even with constraint propagation, backtracking still faces an exponential worst-case time complexity. But in practice, most Sudoku puzzles are solved in milliseconds due to early pruning, especially when the search space is reduced by forward checking and using heuristics like the Most Restrictive Value (MRV), which is not implemented here but could further optimize performance.

In contrast, an A* search algorithm applied to Sudoku is generally ineffective without a powerful heuristic. If we define a state as the "current grid" and actions as "fill one blank," we might consider a simple heuristic h(n)=number of remaining blanks. However, this heuristic is perfectly accurate but ultimately unhelpful. Since each action reduces the number of remaining blanks by exactly one, the search essentially becomes breadth-first search or uniform-cost search. This is akin to Dijkstra’s algorithm where all paths to the solution have the same cost, which means the heuristic offers no guidance to prioritize certain branches over others. Thus, the A* search degenerates to an exhaustive exploration of all possible solutions.

This inefficiency is reflected in our Sudoku solver implementation. In our backtracking solution, the algorithm prunes invalid branches as soon as a constraint is violated, such as when a number already exists in the same row, column, or block. This pruning allows for much faster solutions compared to the A* approach, which would spend time exploring equivalent states without a meaningful heuristic to direct the search.

Testing and Performance Analysis

To evaluate the performance of our backtracking solver, we tested it using several Sudoku grids, either created on a whim or downloaded from websites such as sudoku.com. We measured the number of recursive calls made and the number of backtracks encountered during the solving process, as well as the total time taken. The testing also involved solving puzzles of varying difficulty, from easy to hard, in order to assess how the solver's performance scales with different input sizes and complexities.

- For simple puzzles, the solver quickly completed in milliseconds, making only a few hundred calls and backtracks.

- For more complex puzzles, the solver still performed efficiently, solving “extremely difficult” 9x9 grids in under a second.

In summary, while both the backtracking approach and A* search share exponential worst-case time complexity, our backtracking solver outperforms A* search in the context of Sudoku. This is primarily due to its constraint-based pruning, which eliminates invalid solutions early, whereas A* search without domain-specific heuristics would explore many equivalent and redundant branches, leading to unnecessary computations and inefficiencies.

Part 2: Automated Planning

2A

We can define a domain, that we will here call *lunar*, using a few main types. We chose to use : lander, rover, astronaut (for Mission 3), and location (for surface waypoints and internal areas). We include separate logical locations for each waypoint and also (for Mission 3) the docking bay and control room of each lander (modeled as distinct location objects).

Key predicates include:

(at ?r - rover ?l - location): rover ?r is at location ?l.

(at-lander ?ld - lander ?l - location): lander ?ld is at location ?l (landers are stationary once landed).

(path ?l1 ?l2 - location): true if surface locations ?l1 and ?l2 are connected (bidirectional).

(assigned ?r - rover ?ld - lander): rover ?r belongs to lander ?ld (fixed in each problem).

(image-taken ?r - rover ?l - location): rover ?r has taken an image at location ?l.

(scan-done ?r - rover ?l - location): rover ?r has performed a subsurface scan at location ?l.

(has-sample ?r - rover): rover ?r is currently carrying a sample.

(sample-stored ?ld - lander): lander ?ld has a collected sample stored.

(memory-free ?r - rover): rover ?r's memory is free (no image/scan stored).

For Mission 3 we add:
(astronaut-at ?a - astronaut ?loc - location): astronaut ?a is at location ?loc (docking or control room).

Possibly predicates to distinguish docking vs control area if needed.

Each action has :precondition and :effect clauses.
We define the following actions :

- move, to move from one place to another. (parameters (?r - rover ?from - location ?to - location) precondition (and (at ?r ?from) (path ?from ?to)) effect (and (at ?r ?to) (not (at ?r ?from))))
- take-image and take-scan (requiring (at ?r ?loc) and (memory-free ?r) with effects (image-taken ?r ?loc) or (scan-done ?r ?loc) and (not (memory-free ?r))).
- transmit-data (effect: free memory, maybe add data flag to lander)
- deploy-rover (for undeployed rover: precondition that an astronaut is at the docking bay of its lander (in Mission 3) or simply co-located in earlier missions; effect: rover is now on surface)
- collect-sample (precondition: (at ?r ?loc) where sample exists; effect: (has-sample ?r) and remove sample from ground)
- store-sample at the lander (pre: rover at lander and (has-sample ?r); effect: (sample-stored ?ld), (not (has-sample ?r)))
- For Mission 3, deploy/collect actions require (astronaut-at ?astronaut docking-bay-of-lander), and transmit-data requires (astronaut-at ?astronaut control-room-of-lander) (as per extension rules)

2B and 2C :

The PDDL work comprises two domains and three problem instances: a base domain lunar-domain.pddl for Missions 1 and 2 and an extended domain lunar-extended.pddl for Mission 3 (which adds astronaut-related features). I modelled four primary object types : rover, lander, location and, in the extension, astronaut, and used a compact set of predicates to represent world state: rover-at, lander-at, path, assigned, deployed, memory-free, image-saved, scan-saved, sample-at, has-sample, and sample-stored in the base domain, with astronaut-at, docking and control added in the extended domain. Actions were kept simple and STRIPS-compatible: move, deploy (plus deploy-with-astronaut in the extension), take-image, take-scan, transmit (and transmit-with-astronaut), collect-sample, store-sample, and move-astronaut (extension), each with clear preconditions/effects that enforce the mission rules such as one-piece rover memory and sample storage at landers. The problem files included are lunar-mission1.pddl (one lander/one rover), lunar-mission2.pddl (two landers/two rovers, rover1 pre-deployed), and lunar-mission3.pddl (the extension with Alice and Bob, docking/control areas, and astronaut-dependent deployment/transmission).