SAPIENZA
UNIVERSITÀ DI ROMA

Artificial Intelligence and Robotics

Artificial Intelligence

AI homework - SUDOKU

Student:
Giovanni Zara
1929181

Academic Year 2025/2026

Contents

1 Introduction

In this project, I address the classic Sudoku puzzle as a search and constraint satisfaction problem. Sudoku is a well-known combinatorial puzzle where the objective is to fill a 9×9 grid with digits 1–9 such that each row, column, and 3×3 sub-box contains all digits only once. The puzzle provides a challenging problem for AI search algorithms due to its exponential search space and constrained nature.

I implemented and compared two distinct approaches for solving Sudoku:

1. **A* Search Algorithm**: I modeled Sudoku as a state-space search problem, where each state represents a partially filled grid and transitions correspond to placing a valid digit in an empty cell. A* uses heuristics to guide the search toward the goal state. I implemented two different heuristics for testing reasons: a simple one counting empty cells, and a more advanced one that also considers the constraint space of each cell.
2. **Constraint Satisfaction Problem (CSP) Reduction**: I reduced the Sudoku problem to a CSP formulation with 81 variables (one per cell), domains of $\{1, \ldots, 9\}$ for empty cells, and 27 different constraints (9 for rows, 9 for columns, and 9 for boxes). I then employed the python-constraint library as the CSP solver.

The solution is structured into four main modules:

- `sudoku.py`: Core problem representation and modelling.
- `a_star.py`: A* implementation with a problem-specific wrapper and custom heuristics.
- `sudoku_csp.py`: CSP formulation and solver interface.
- `benchmark.py`: Experimental framework for comparing both approaches across puzzles of varying difficulty.

The experimental results show that the CSP solver generally outperforms A* on easier puzzles due to efficient constraint propagation, while performance varies on harder puzzles. The A* approach with the advanced heuristic can sometimes reduce the number of nodes expanded compared to the simple heuristic, (but with increased computational overhead per node). The branching factor remains low (typically between 1.0 and 1.35) thanks to the MRV heuristic used in successor generation.

This work was inspired by and based on famous Peter Norvig essay on sudoku.
Cool Norvig's quote:

> "Why did I do this? As computer security expert Ben Laurie has stated, Sudoku is "a denial of service attack on human intellect". Several people I know (including my wife) were infected by the virus, and I thought maybe this would demonstrate that they didn't need to spend any more time on Sudoku. It didn't work for my friends (although my wife has since independently kicked the habit without my help), but at least one stranger wrote and said this page worked for him, so I've made the world more productive. And perhaps along the way I've taught something about Python, constraint propagation, and search."[Norvig, 2016a]

2 Task 1: Problem

2.1 Problem Choice

I selected the **Sudoku puzzle** as problem domain. Sudoku is a constraint-heavy combinatorial puzzle with a well-defined goal state, making it suitable for both heuristic search and constraint satisfaction approaches.

2.2 Problem Definition

A Sudoku puzzle consists of a 9×9 grid, partially filled with digits from 1 to 9. The goal is to complete the grid such that:

- Each **row** contains all digits 1–9 exactly once.
- Each **column** contains all digits 1–9 exactly once.
- Each of the nine 3×3 **sub-boxes** contains all digits 1–9 exactly once.

2.3 State-Space Formulation

For the A* search approach, we model Sudoku as a state-space search problem:

- **State**: A 9×9 grid represented as a NumPy array, where 0 indicates an empty cell. The state is encapsulated in the `SudokuState` class.
- **Initial State**: The given puzzle with pre-filled cells.
- **Actions**: Place a valid digit (1–9) in an empty cell. An action is valid if it does not violate the row, column, or box constraints.
- **Transition Model**: Given a state and an action (row, col, num), the successor state is a copy of the current grid with the digit num placed at position (row, col).
- **Goal Test**: The grid is complete (no zeros) and satisfies all Sudoku constraints.
- **Path Cost**: Each action has a uniform cost of 1, so the total cost equals the number of cells filled.

2.4 Test Instances

I prepared a benchmark suite with 8 puzzles of increasing difficulty, categorized as:

- **Easy**: 49–51 empty cells
- **Medium**: 51–53 empty cells
- **Hard**: 60–64 empty cells
- **Expert**: 58–64 empty cells

Difficulty correlates with the number of empty cells and so with the ambiguity in the puzzle, which affects the branching factor and search depth.

3 Task 2.1: Implementation of A*

3.1 Algorithm Overview

The A* algorithm was implemented following the standard formulation with duplicate detection and path cost optimization. The implementation uses the evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ represents the cost from the initial state to node n, and $h(n)$ is the heuristic estimate to the goal.

3.2 Data Structures

3.2.1 Node Representation

Each search node is represented using Python's `dataclass` decorator with the following fields:

- `f_cost`: The total estimated cost ($g + h$), used for priority queue ordering
- `state`: The `SudokuState` object representing the current grid configuration
- `g_cost`: The path cost from the initial state
- `h_cost`: The heuristic value
- `parent`: Reference to the parent node for solution reconstruction
- `action`: The action (row, column, value) that led to this state

The `@dataclass(order=True)` decorator enables automatic comparison based on `f_cost`, which is essential for the priority queue ordering.

3.2.2 Frontier

The frontier (open list) is implemented using Python's `heapq` module, which provides a min-heap priority queue. This ensures $O(\log n)$ insertion and $O(\log n)$ extraction of the minimum element. Additionally, a dictionary (`frontier_dict`) maps states to their corresponding nodes, enabling $O(1)$ lookup to check if a state is already in the frontier and to retrieve its current cost.

3.2.3 Explored Set

The explored set (closed list) is implemented as a Python `set`, leveraging the `__hash__` method of `SudokuState` (which hashes the grid's byte representation).

3.3 Implementation Details

3.3.1 Duplicate Detection

The implementation performs duplicate detection at two levels:

1. **Explored set check**: States already expanded are never reconsidered (no reopening policy)
2. **Frontier check**: When a successor is already in the frontier, I compare path costs and keep only the better path

When a better path to a frontier node is found, the old node is removed from the heap, the heap is re-heapified, and the new node with lower cost is inserted.

3.3.2 Successor Generation with MRV

The successor generation in `SudokuState.get_successors()` incorporates the Minimum Remaining Values (MRV) heuristic. Rather than selecting an arbitrary empty cell, I choose the cell with the fewest possible valid values: This significantly reduces the branching factor by prioritizing constrained cells.

3.3.3 Heuristics

Two heuristic functions were implemented:

Empty Cells Heuristic:

$$h_1(s) = |\{(r, c) : grid[r][c] = 0\}| \tag{1}$$

This is the easy heuristic, as it only counts the number of empty cells remaining.

Advanced Heuristic:

$$h_2(s) = h_1(s) + 0.1 \times \sum_{(r,c)\in \text{empty}} penalty(r, c) \tag{2}$$

where $penalty(r, c)$ adds 0.5 for cells with exactly 2 possible values, 1.0 for cells with 3+ possibilities, and returns ∞ for cells with no valid options (dead-end detection).

3.3.4 Uniform Step Cost

All actions have a uniform cost of 1.0, making the solution length equal to the number of cells filled.

3.4 Statistics Tracking

The implementation tracks several metrics for benchmarking:

- Nodes expanded and generated
- Maximum frontier size (peak memory for open list)
- Maximum total memory (frontier + explored)
- Branching factor statistics (min, avg, max)
- Solution path length and total time elapsed

4 Task 2.2: Implementation of CSP

4.1 CSP Formulation

The Sudoku puzzle is naturally expressible as a Constraint Satisfaction Problem. The reduction follows the standard CSP framework with variables, domains, and constraints.

4.1.1 Variables

We define 81 variables, one for each cell in the 9×9 grid:

$$\mathcal{V} = \{cell_{r,c} : r \in \{0, ..., 8\}, c \in \{0, ..., 8\}\} \tag{3}$$

4.1.2 Domains

The domain of each variable depends on whether the cell is pre-filled:

$$D(cell_{r,c}) = \begin{cases} \{grid[r][c]\} & \text{if } grid[r][c] \neq 0 \\ \{1, 2, ..., 9\} & \text{otherwise} \end{cases} \tag{4}$$

Pre-filled cells have singleton domains, effectively fixing their values.

4.1.3 Constraints

The constraints encode the Sudoku rules using 27 ($9 \times 9 \times 9$) `AllDifferent` constraints:

Row Constraints (9):

$$\forall r \in \{0, ..., 8\} : AllDiff(\{cell_{r,c} : c \in \{0, ..., 8\}\}) \tag{5}$$

Column Constraints (9):

$$\forall c \in \{0, ..., 8\} : AllDiff(\{cell_{r,c} : r \in \{0, ..., 8\}\}) \tag{6}$$

Box Constraints (9):

$$\forall b_r, b_c \in \{0, 1, 2\} : AllDiff(\{cell_{3b_r+i,3b_c+j} : i, j \in \{0, 1, 2\}\}) \tag{7}$$

4.2 Implementation Details

4.2.1 CSP Solver Library

The implementation uses the `python-constraint` library, which provides:

- A `Problem` class for defining CSP instances
- Built-in `AllDifferentConstraint` for expressing uniqueness
- A backtracking solver with constraint propagation

4.2.2 Reduction Process

The method `_create_csp_from_sudoku()` performs the reduction in two steps:

Step 1 - Variable Creation:

```
for row in range(9):
    for col in range(9):
        var_name = f"cell_{row}_{col}"
        if grid[row, col] != 0:
            problem.addVariable(var_name, [int(grid[row, col])])
        else:
            problem.addVariable(var_name, list(range(1, 10)))
```

Step 2 - Constraint Addition: Row, column, and box constraints are added using `AllDifferentConstraint`, which internally adds the constraint that all variables must have different values.

4.2.3 Solution Parsing

The CSP solver returns a dictionary mapping variable names to values. The method `_parse_solution()` converts this back to a 9×9 NumPy array:

```
for var_name, value in csp_solution.items():
    parts = var_name.split('_')
    row, col = int(parts[1]), int(parts[2])
    grid[row, col] = value
```

4.3 Solving Process

The complete solving workflow is:

1. **Modeling Phase**: Convert `SudokuState` to CSP (`_create_csp_from_sudoku`)
2. **Solving Phase**: Call `problem.getSolution()` which invokes the backtracking solver
3. **Parsing Phase**: Convert the solution dictionary back to `SudokuState`

4.4 Statistics Tracking

The implementation separately measures:

- **Modeling time**: Time to create variables and constraints
- **Solving time**: Time spent in the CSP solver
- **Total time**: End-to-end solving time
- **Problem size**: Number of variables (81) and constraints (27)

5 Task 3: Experimental Results

5.1 Experimental Setup

As said before, the benchmarks were conducted on puzzles of varying difficulty, categorized as `easy`, `medium`, `hard`, and `expert`. Difficulty scales with the number of empty cells: more empty cells generally result in a larger search space. Each algorithm configuration was run 3 times per puzzle, and results were averaged to reduce variance.
All experiments were run on a Windows machine using Python.

5.2 Metrics Description

The following metrics were collected during the experiments (standard literature metrics + web search):

5.2.1 A* Algorithm Metrics

- **Nodes Expanded**: The number of nodes removed from the frontier and processed. This measures the effective search effort.
- **Nodes Generated**: The total number of successor nodes created during search. This is always $\geq$ nodes expanded.
- **Max Frontier Size**: The max number of nodes stored in the open list (priority queue). This represents the memory required for the frontier.
- **Max Memory Nodes**: The maximum of (frontier size + explored set size) at any point. This approximates total memory usage in terms of nodes stored.
- **Solution Length**: The number of actions (cell placements) in the solution path, equal to the number of empty cells filled.
- **Branching Factor**: Statistics on the number of successors per expanded node:
  - *Min BF*: Minimum successors from any node (often 1-2 due to MRV)
  - *Avg BF*: Average branching factor across all expansions
  - *Max BF*: Maximum successors from any single node
- **Time Elapsed**: Time from search start to solution (in ms).

5.2.2 CSP Solver Metrics

- **Variables**: Number of CSP variables (always 81 for standard Sudoku).
- **Constraints**: Number of `AllDifferent` constraints (always 27: 9 rows + 9 columns + 9 boxes).
- **Modeling Time**: Time to construct the CSP model (create variables and constraints).
- **Solving Time**: Time spent by the CSP solver's backtracking algorithm.
- **Total Time**: Tot time including modelling and solving.

5.3 Analysis

5.3.1 A* Heuristic Comparison

The advanced heuristic, which penalizes cells with multiple possibilities, generally results in fewer nodes expanded compared to the simple empty-cells heuristic.

5.3.2 A* vs. CSP

The CSP solver typically outperforms A* on harder puzzles due to:

- Built-in constraint propagation that prunes the search space more aggressively
- Optimized implementation of the `AllDifferent` constraint (from constraint python lib)
- Arc consistency algorithms that detect failures earlier

However, for easier puzzles, the overhead of CSP modeling may make A* competitive.

5.4 Results

5.4.1 Execution Time Comparison

Table 1 shows the execution time for each algorithm across all puzzles.

Table 1: Execution time (ms) comparison across puzzle difficulties

Puzzle	Empty Cells	A* (simple)	A* (advanced)	CSP
easy_2	49	9.27	17.91	**1.79**
easy_1	51	9.24	18.62	**4.12**
medium_1	51	18.82	24.56	**6.32**
medium_2	53	29.83	47.14	**3.95**
expert_1	58	245.61	195.83	**44.27**
hard_1	60	2634.75	3634.34	**301.46**
hard_2	64	75.36	**31.43**	2.97
expert_2	64	1326.86	3233.41	2393.44

5.4.2 A* Search Effort

Table 2 presents the search effort metrics for A* with both heuristics.

Table 2: A* nodes expanded, generated, and memory usage

Puzzle	Heuristic	Expanded	Generated	Max Frontier	Max Memory
easy_2	empty_cells	49	50	1	50
easy_2	advanced	49	50	1	50
easy_1	empty_cells	51	52	1	52
easy_1	advanced	51	52	1	52
medium_1	empty_cells	81	82	3	84
medium_1	advanced	51	54	3	54
medium_2	empty_cells	161	162	7	168
medium_2	advanced	125	137	12	137
expert_1	empty_cells	1304	1312	17	1321
expert_1	advanced	549	592	43	592
hard_1	empty_cells	13235	13248	40	13275
hard_1	advanced	8842	9737	895	9737
hard_2	empty_cells	437	579	142	579
hard_2	advanced	69	93	24	93
expert_2	empty_cells	6706	6727	45	6751
expert_2	advanced	8344	9488	1144	9488

6 How to Run

6.1 Prerequisites

The implementation requires Python 3.8 or higher.

6.1.1 Required Libraries

- **NumPy**: For efficient array operations in the Sudoku grid representation
- **python-constraint**: CSP solver library providing backtracking with constraint propagation

This implementation does not require external solvers such as MiniSAT, OR-Tools, or Fast Downward. The CSP solving is handled entirely by the `python-constraint` library.

6.2 Project Structure

```
homework/
|-- sudoku.py          # Sudoku state representation and utilities
|-- a_star.py          # A* algorithm implementation
|-- sudoku_csp.py      # CSP reduction and solver
|-- benchmark.py       # Benchmarking script
`-- benchmark_results.csv  # Output file (generated)
```

6.3 Running Individual Solvers

6.3.1 A* Solver

To run the A* solver on the default test puzzles:

```
python a_star.py
```

This will solve the EASY puzzle and display the solution along with statistics. To solve a custom puzzle, modify the code or import and use programmatically:

```
from sudoku import load_sudoku_from_string
from a_star import AStar, SudokuProblem, advanced_heuristic

puzzle_str = "530070000600195000..."  # 81 characters
puzzle = load_sudoku_from_string(puzzle_str)
problem = SudokuProblem(puzzle)

solver = AStar(heuristic_fn=advanced_heuristic)
solution, stats = solver.search(problem)
```

6.3.2 CSP Solver

To run the CSP solver on test puzzles:

```
python sudoku_csp.py
```

This will solve EASY, MEDIUM, and HARD puzzles and display results.
Programmatic usage:

```
from sudoku import load_sudoku_from_string
from sudoku_csp import SudokuCSPSolver

puzzle = load_sudoku_from_string("530070000600195000...")
solver = SudokuCSPSolver()
solution, stats = solver.solve(puzzle)
```

6.4 Running Benchmarks

To reproduce the experimental results:

```
python benchmark.py
```

This will:

1. Run A* with the `empty_cells` heuristic on all puzzles (3 runs each)
2. Run A* with the `advanced` heuristic on all puzzles (3 runs each)
3. Run the CSP solver on all puzzles (3 runs each)
4. Print a summary table to the console
5. Export detailed results to `benchmark_results.csv`

6.4.1 Customizing Benchmarks

To add new puzzles, modify the `PUZZLES` dictionary in `benchmark.py`:

```
PUZZLES = {
    "my_puzzle": "123456789...",  # 81 characters, 0 for empty
    # ... existing puzzles
}
```

To change the number of runs per configuration:

```
results = runner.run_all_benchmarks(PUZZLES, num_runs=5)
```

6.5 Choosing Algorithm and Heuristic

6.5.1 A* Heuristics

Two heuristics are available:

- `empty_cells_heuristic`: Counts remaining empty cells (admissible)
- `advanced_heuristic`: Adds penalties for constrained cells with dead-end detection

Select the heuristic when creating the solver:

```
from a_star import AStar, empty_cells_heuristic, advanced_heuristic

solver = AStar(heuristic_fn=empty_cells_heuristic)
# or
solver = AStar(heuristic_fn=advanced_heuristic)
```

Bibliography

Faizalkhan. Cracking sudoku with ai: Understanding constraint satisfaction problems (csps). *Medium*, 2025. URL https://medium.com/@faizalkhan1712/cracking-sudoku-with-ai-understanding-constraint-satisfaction-problems-csps-f7cf15610ce1.

Peter Norvig. Solving every sudoku puzzle. *Norvig.com*, 2016a.

Peter Norvig. Solving every sudoku puzzle quickly. *Norvig.com*, 2016b.

Yashkochar. Solving sudoku as a constraint satisfaction problem (csp). *Medium*, 2024. URL https://medium.com/@yashkochar01/solving-sudoku-as-a-constraint-satisfaction-problem-csp-54cb553c3cab.