Solving Sudoku:
A Study of Heuristic Search and Probabilistic Methods

Vidyut Ramanan[1], Jonathan Sudarpo[2]

[1]Northeastern University
[2]Northeastern University
ramanan.v@northeastern.edu, sudarpo.j@northeastern.edu

Abstract

This project explores solving Sudoku puzzles using artificial intelligence, focusing on how algorithmic performance varies with puzzle difficulty. Difficulty is quantified based on the number of initial clues provided in each puzzle, with fewer clues generally indicating a higher level of challenge. We implemented two solving techniques: a Constraint Satisfaction Problem (CSP) solver utilizing Minimum Remaining Value (MRV) and Least Constraining Value (LCV) heuristics, and a Monte Carlo Tree Search (MCTS) based solver. The CSP approach efficiently narrows the search space and consistently solves puzzles across all difficulty levels with high accuracy. In contrast, the MCTS method performed worse due to its reliance on random simulations, which struggle to make meaningful progress in the highly constrained and deterministic nature of Sudoku. Our evaluation demonstrates that heuristic-driven CSP solvers are significantly more effective for this domain, while MCTS requires substantial adaptation to handle such constraint-heavy environments.

Code—github.com/vidyut-ramanan/CS4100-Final-Project
Datasets —kaggle.com/datasets/rohanrao/sudoku/data

Introduction

Sudoku is a widely recognized logic-based puzzle that has become a popular test case in artificial intelligence research due to its structured rules and computational complexity. Formally, Sudoku is a Constraint Satisfaction Problem (CSP) in which the objective is to fill a 9×9 grid so that each row, each column, and each 3×3 subgrid contains all digits from 1 to 9 exactly once. From a theoretical standpoint, solving a general Sudoku puzzle is known to be NP-complete, making it a compelling challenge for evaluating search algorithms, constraint reasoning, and probabilistic inference techniques, especially as the puzzle size increases.

In this project, we aim to develop and compare two AI-based approaches to solving Sudoku puzzles of varying difficulty levels. The input to our system is a partially-filled 9×9 Sudoku grid, and the output is a completed, valid grid that satisfies all Sudoku constraints. We use the publicly available Kaggle Sudoku dataset (Vopani, 2019), which contains 9 million puzzles along with their solutions. Since the dataset does not include difficulty labels, we introduce a heuristic-based metric to quantify puzzle difficulty: the number of given clues. We define the difficulty levels as follows:

- Easy: puzzles with 64 or more given clues.
- Medium: puzzles with between 50.0 and 63 given clues.
- Hard: puzzles with between 37 and 49 given clues
- Expert: puzzles with 36 or fewer given clues

This classification provides a way to evaluate solver performance under increasing levels of constraint.

To address the Sudoku-solving task, we explore two contrasting algorithms: a deterministic CSP-based solver using backtracking with constraint propagation, and a probabilistic solver based on Monte Carlo Tree Search (MCTS). The backtracking solver models each cell as a variable and uses heuristics such as Minimum Remaining Values (MRV) and Least Constraining Value (LCV) to efficiently search the solution space. This approach is well-suited to structured problems like Sudoku, where logical deduction significantly narrows down possibilities.

On the other hand, the MCTS-based solver, typically used in domains like game-playing, approaches the problem with randomized simulations and iterative improvement. It selects partial solutions based on their estimated likelihood of leading to valid completions, using the classic four-step MCTS loop: selection, expansion, simulation, and backpropagation. While MCTS offers flexibility in high-dimensional search spaces and does not require handcrafted heuristics, Sudoku's strict constraints and sparse valid solution space present a challenge for its exploratory strategy.

By comparing a rule-based, CSP method with a probabilistic, simulation-driven approach like MCTS, we investigate how different AI methods perform across puzzles of increasing difficulty. Our work not only contributes a comparative study on solver efficiency and effectiveness but also provides insights into how algorithmic design choices align with the structural properties of highly constrained problems like Sudoku.

Background

Solving Sudoku puzzles effectively requires a strong understanding of both constraint-based reasoning and probabilistic decision-making. At its core, Sudoku is a classic example of a Constraint Satisfaction Problem (CSP)—a well-established framework in artificial intelligence where the objective is to assign values to a set of variables within defined constraints. In Sudoku, each of the 81 cells in the 9×9 grid represents a variable. The domain of each variable is the digits 1 through 9, and the constraints are that no digit may repeat in any row, column, or 3×3 subgrid. These are binary constraints because they define relationships between pairs of variables.

To solve CSPs like Sudoku efficiently, AI algorithms typically employ a technique known as backtracking search, where the system assigns values to variables incrementally, backtracking whenever a constraint is violated. To avoid wasting time exploring invalid branches of the search space, constraint propagation techniques such as forward checking and arc consistency are used. These techniques proactively eliminate values from the domains of unassigned variables whenever a new assignment is made, thereby pruning the search space early. Further performance improvements are achieved through heuristics. The Minimum Remaining Values (MRV) heuristic helps prioritize the most constrained variables—those with the fewest legal values left—while the Least Constraining Value (LCV) heuristic guides the solver to select values that minimize constraints on other variables, preserving flexibility for future assignments.

While backtracking and CSPs represent a deterministic and structured approach, we also explore a probabilistic method known as Monte Carlo Tree Search (MCTS). MCTS is widely used in AI for problems with large and uncertain search spaces, particularly in games such as Go and Chess. Unlike backtracking, which systematically explores the search space, MCTS relies on simulation. It builds a search tree over time by selecting promising moves, expanding them, simulating random outcomes, and backpropagating the results to inform future decisions. Each iteration helps the algorithm learn which actions are more likely to lead to successful outcomes, balancing exploration of new paths with exploitation of known good strategies.

However, applying MCTS to Sudoku presents unique challenges. Unlike many game scenarios where intermediate rewards are available, Sudoku only has a single, all-or-nothing success condition: the entire grid must be filled with no constraint violations. This makes it difficult for MCTS to learn efficiently from simulations, since most random completions of the puzzle are invalid. As a result, MCTS may struggle to converge on valid solutions without additional guidance or hybridization with rule-based logic.

By grounding one approach in structured logical inference (CSP with backtracking) and the other in exploratory decision-making (MCTS), our project investigates how different paradigms in AI perform when applied to the same problem. This contrast allows us to study the strengths and limitations of each method—especially as puzzle difficulty increases—and provides insight into how deterministic and probabilistic solvers behave under different computational constraints.

Related Work

Sudoku has been a popular benchmark problem in artificial intelligence, with a wide range of solving techniques explored in both academic research and practical applications. One of the earliest and most straightforward approaches is brute-force search, where every possible number is tried in every cell until a valid configuration is found. While this method guarantees a solution if one exists, it is computationally expensive for harder puzzles due to the size of the search space.

More sophisticated deterministic methods frame Sudoku as a Constraint Satisfaction Problem (CSP) and apply backtracking search augmented with heuristics and constraint propagation. Peter Norvig’s classic solver (2006) used this approach by using depth-first search. This method forms the basis of our deterministic approach, which we

enhance with heuristics such as Minimum Remaining Values (MRV) and Least Constraining Value (LCV). These improvements aim to reduce unnecessary branching and make the solver scalable to harder puzzles.

On the probabilistic side, Monte Carlo Tree Search (MCTS) has been a dominant algorithm in AI game-playing, particularly after its successful use in games like Go and Chess. Cazenave (2009) proposed Nested Monte Carlo Search as a method for guiding search in environments with large state spaces and no effective heuristics, using layers of random simulations to evaluate and select promising moves. MCTS has also been adapted for single-player planning and puzzle-solving tasks. However, Sudoku poses unique difficulties for MCTS because there are no intermediate rewards or partial correctness—either the puzzle is fully solved or it is not. This lack of range in the reward makes it harder for MCTS to converge without incorporating sudoku specific guidance. Nevertheless, we explore its application here to test its capability in a highly constrained, single-solution space.

Other alternative methods that could be applied to this problem include genetic algorithms, which use evolutionary principles like mutation and crossover to evolve potential solutions over time. While genetic algorithms can occasionally stumble upon valid solutions, they often struggle with constraint-heavy environments like Sudoku where most random configurations are invalid. Similarly, simulated annealing and other stochastic hill-climbing methods offer another class of optimization-based solvers, but these also suffer from the same issue: they lack a structured way to enforce Sudoku's strict rules throughout the search.

More recently, deep learning approaches have been explored for solving Sudoku, but these models often struggle to generalize beyond their training distribution. Neural networks can learn to solve specific puzzle configurations, but they typically fail to apply learned strategies to out-of-distribution examples without significant architectural or training modifications (Abdool 2023). This limits their reliability and interpretability compared to symbolic methods like CSP or MCTS.

Ultimately, we chose to focus on backtracking with constraint propagation and MCTS because they represent two fundamentally different philosophies in AI: structured logical inference versus statistical decision-making. By directly comparing these approaches on the same problem, our project aims to shed light on when and why one might outperform the other, and what this tells us about AI strategies more broadly. Our work builds upon and extends prior research, while intentionally avoiding methods like pure neural networks or genetic algorithms in order to maintain interpretability and control over the solving process.

Project Description

1. CSP with MRV + LCV

```
## 1. CSP Solver (with MRV + LCV Heuristics)

Function solve_csp(grid):
    If no empty cells:
        Return True  // Puzzle solved

    Select the empty cell with the Minimum Remaining Values (MRV)

    For each value in the Least Constraining Value (LCV) order for that cell:
        If placing the value is valid:
            Place the value in the cell
            If solve_csp(grid) returns True:
                Return True  // Solved
            Else:
                Undo the placement (backtrack)

    Return False  // No valid assignment found
```

Figure 1: Pseudocode for CSP using Minimum Remaining Value (MRV) and Least Constraining Value (LCV)

Algorithm Overview

The heuristic CSP solver is based on a classic backtracking approach, enhanced by two domain-specific heuristics: Minimum Remaining Values (MRV) and Least Constraining Value (LCV).

MRV helps the solver choose the "most constrained" variable first — in other words, the empty cell that has the fewest possible valid numbers remaining. By tackling the hardest parts of the puzzle early, MRV reduces the chances of running into contradictions deep into the search, thus saving time by failing early when necessary.

LCV orders the candidate numbers for a given cell. Rather than trying numbers in sequential order, LCV prefers numbers that impose the least restriction on neighboring empty cells. This helps maximize future options and reduces the likelihood of getting stuck later in the search.

By combining MRV for variable selection and LCV for value ordering, this solver makes much more informed decisions compared to naive backtracking. It systematically navigates the search space, focusing effort where it matters most. As a result, the MRV + LCV solver tends to solve even harder Sudoku puzzles relatively efficiently, with significantly fewer backtracks than an uninformed search. This method still guarantees completeness (it will find a solution if one exists) and correctness (only valid moves are made), but greatly improves practical performance through smarter search ordering.

Pseudocode Description

The pseudocode for the CSP MRV+LCV solver follows a recursive backtracking structure with heuristics guiding the search:

- First, the solver checks if the puzzle is already solved by seeing if there are any remaining empty cells.
- If not solved, it selects the next empty cell using Minimum Remaining Values (MRV) — the cell with the fewest valid candidate numbers.
- It then generates a list of valid numbers for that cell, ordered by Least Constraining Value (LCV) — preferring values that least restrict future moves.
- For each candidate number:
  - It attempts to place the number in the cell.
  - Recursively calls itself to attempt solving the updated board.
  - If successful, the recursion unwinds and the solution is returned.
  - If unsuccessful, the solver backtracks by removing the number and trying the next candidate.
- If no valid moves are found at any point, the function returns False to trigger backtracking.

This structure ensures that the solver always prioritizes the most constrained parts of the puzzle first, and attempts moves that are least disruptive to future choices, significantly improving search efficiency.

2. Naive CSP without MRV + LCV

```
## 2. CSP Solver (Naive without MRV + LCV)

Function solve_csp_naive(grid):
    For each row from 0 to 8:
        For each column from 0 to 8:
            If the cell is empty:
                For each number from 1 to 9:
                    If placing the number is valid:
                        Place the number in the cell
                        If solve_csp_naive(grid) returns True:
                            Return True  // Solved
                        Else:
                            Undo the placement (backtrack)
                Return False  // No valid number found, trigger backtracking

    Return True  // All cells filled successfully
```

Figure 2: Pseudocode for Naive CSP

Algorithm Overview

The naive CSP solver implements pure backtracking without any form of heuristics or lookahead. It simply scans the Sudoku grid from top-left to bottom-right, and upon encountering an empty cell, it tries filling it with every number from 1 to 9 sequentially. If none of the numbers are valid at a certain cell, the solver backtracks to the previous decision point and tries a different number.

Because it lacks any prioritization or strategic selection, the naive solver may waste time trying doomed paths or making poor early choices that lead to deep failure much later. Its search tree is often enormous compared to the heuristic version.

However, this method is still guaranteed to find a solution if one exists (because it explores all possibilities exhaustively) and is easy to implement. It is often effective for easy puzzles where most numbers have only a few options, but becomes very slow and inefficient for hard or expert-level puzzles where there are many possible combinations.

In practice, this solver serves as a useful baseline: it demonstrates the fundamental principles of recursive search and highlights how important smart heuristics (like MRV/LCV) are for scaling to more complex problem instances.

Pseudocode Description

The pseudocode for the CSP Naive solver follows a simple recursive backtracking structure without applying any heuristics:

- First, the solver scans the Sudoku grid from the top-left to the bottom-right to find the first empty cell.
- When an empty cell is found, it tries placing each number from 1 to 9 sequentially.
- For each candidate number:
  - It checks whether placing that number is valid according to Sudoku rules.
  - If valid, it places the number into the cell.
  - It then recursively calls itself to attempt solving the updated grid.
  - If successful, the recursion unwinds and the solution is returned.
  - If the placement leads to a dead-end later, the solver backtracks by removing the number and tries the next candidate.
- If none of the numbers 1–9 lead to a solution from that cell, the function returns False to trigger backtracking.

This approach explores all possibilities without prioritization, making it effective for simple puzzles but very slow for harder puzzles due to the large search space.

3. MCTS

```
## 3. MCTS Solver with Tree Search
Function solve_mcts(puzzle, max_iterations):
    Initialize root node with the initial puzzle state

    For i from 1 to max_iterations:
        node = tree_policy(root)
        result = rollout(node.state)
        backpropagate(node, result)

        If the root state is terminal (puzzle solved):
            Return the solved board and the number of iterations

    Return the best board found after max_iterations


Function selection(node):
    While the current node's state is not terminal:
        If the node is not fully expanded:
            expand(node)
            If node has children:
                Return a random child
            Else:
                Return the current node
        Else:
            node = best_child(node)
    Return the node


Function expand(node):
    Find an empty cell using Minimum Remaining Values (MRV)
    For each valid number at that cell:
        Create a new board state by placing that number
        If the new board is solvable:
            Add a new child node for this move


Function best_child(node, c_param):
    For each child:
        Calculate UCT score = (wins / visits) + c_param * sqrt(log(parent_visits) / child_visits)
    Return the child with the highest UCT score


Function rollout(state):
    While the board is incomplete:
        Get possible moves (domain values)
        Randomly pick one and fill it
    Return True if the resulting board is solved, else False


Function backpropagate(node, result):
    While node is not None:
        Increment node.visits
        Add result to node.wins
        Move to node.parent
```

Figure 3: Monte Carlo Tree Search Pseudocode for Sudoku solver

Algorithm Overview

The MCTS (Monte Carlo Tree Search) solver takes a fundamentally different approach compared to traditional CSP backtracking. Instead of systematically enumerating possibilities, MCTS builds a search tree dynamically through simulation and probabilistic exploration.

The process follows four main phases:

Selection:

Starting from the root node (the initial puzzle state), the solver navigates down the tree by selecting child nodes. Selection uses a balance of exploration and exploitation, calculated using the Upper Confidence Bound for Trees (UCT) formula: $UCT = \frac{w_i}{n_i} + c * \sqrt{\frac{ln(N)}{n_i}}$

where:

- w_i = number of successful outcomes (wins) for child i,
- n_i = number of visits to child i,
- N = total number of visits to the parent node,
- c = exploration constant (often 2).

This encourages picking moves that either have high success rates (exploitation) or have been rarely tried (exploration).

Expansion:

When reaching a node that is not fully expanded, the solver generates possible next moves by picking an empty cell (using MRV to prioritize) and trying all valid numbers. Each new move becomes a new child node.

Rollout (Simulation):

From the new node, the solver performs a random rollout — it fills the remaining empty cells randomly with valid numbers (without further deep search) until the board is complete or gets stuck. This quick simulation estimates how "good" a partial board might be.

Backpropagation:

The result of the rollout (success or failure) is propagated back up the tree. Each node along the path updates its visit count and success rate accordingly, strengthening successful paths and discouraging poor ones over time.

Over many iterations, MCTS gradually focuses its search on promising regions of the tree without ever exhaustively checking all possibilities.

Overall, MCTS offers a flexible and adaptive method for Sudoku solving, fundamentally different from deterministic backtracking approaches.

Pseudocode Description

The pseudocode for the MCTS solver organizes solving into an iterative, simulation-guided search using a dynamic tree structure:

- First, the solver initializes a root node that represents the initial Sudoku puzzle state.
- For each iteration up to a specified maximum:
  - It calls the Selection phase to navigate from the root down to a promising node.
- During Selection:
  - If the current node is not fully expanded, it calls Expand to generate new child nodes representing possible valid moves.
  - If the node is fully expanded, it selects the best child node based on the Upper Confidence Bound (UCT) formula, balancing exploration and exploitation.
- After selecting or expanding a node, the solver proceeds to the Rollout phase:

    - In Rollout, the solver simulates randomly filling the remaining empty cells with valid moves until the board is completed or a dead-end is reached.
- The result of the Rollout (success or failure) is passed into the Backpropagation phase:
    - In Backpropagation, the solver updates the visit and win statistics for each node along the path back to the root.
- If at any point the root node's puzzle state becomes a fully solved Sudoku board, the solver terminates early and returns the solution.
- Otherwise, the solver continues iterating, gradually refining its search tree to favor moves that led to more successful outcomes during previous simulations.

This structure allows the solver to efficiently guide exploration without exhaustive enumeration, balancing random exploration and learned experience through simulations.

Experiments

To evaluate the performance of our Sudoku solvers, we designed a set of experiments focused on two main aspects: solving speed and solving accuracy. We used a subset of puzzles from the publicly available Kaggle Sudoku dataset (Vopani, 2019) and categorized puzzle difficulty based on the number of given clues: puzzles with 64 or more clues were labeled "Easy," puzzles that had between 50 and 63 clues as "Medium," between 37 and 49 clues as "Hard," and puzzles with fewer than 37 clues as "Expert." Each solver—the CSP solver with MRV+LCV heuristics, the naive CSP solver, and the MCTS-based solver—was tested on the same set of puzzles across all difficulty levels. For each trial, we recorded the average time to solve a puzzle and the solver's success rate, defined as producing a fully correct, constraint-satisfying solution.

The results show clear performance differences between the methods. In terms of speed, both CSP solvers performed extremely efficiently, solving puzzles in a fraction of a second on average. In contrast, the MCTS solver required significantly more time, averaging over 1.4 seconds per puzzle. This disparity is illustrated in Figure 1, where the MCTS solver's average solve time is substantially higher due to the computational overhead of random simulations and tree-building operations.

Accuracy results further reinforce the advantages of the CSP-based approaches. As shown in Figure 2, both the CSP MRV+LCV solver and the naive CSP solver achieved nearly 100% correctness across all tested puzzles, while the MCTS solver lagged behind with an overall accuracy of approximately 85%. The accuracy breakdown by difficulty level, displayed in Figure 3, reveals that while all solvers performed comparably on easy and medium puzzles, the MCTS solver's accuracy dropped significantly on hard puzzles and even more drastically on expert-level puzzles, where its success rate fell to around 65%.

The performance trends observed can be explained by the fundamental differences between the algorithms. The CSP solvers succeed consistently because Sudoku is inherently a constraint-heavy, structured problem, which aligns perfectly with backtracking and constraint propagation techniques. The MRV heuristic helps target the most constrained cells first, and LCV minimizes the impact of choices on future possibilities, significantly improving efficiency compared to naive backtracking. By contrast, the MCTS solver struggles because Sudoku provides no intermediate rewards; random rollouts are unlikely to complete valid boards by chance, especially as puzzles become more constrained. Without partial feedback during simulation, MCTS finds it difficult to converge on successful strategies, leading to lower accuracy and longer solution times on difficult puzzles.

In summary, our experiments demonstrate that heuristic-driven CSP solvers are highly effective for solving Sudoku puzzles across all difficulty levels, both in terms of speed and correctness. In contrast, MCTS, while powerful in domains with rich intermediate feedback like Go or Chess, is not well-suited to domains like Sudoku without significant modification. These results highlight the importance of aligning solver strategies with the structural properties of the problem domain.

Conclusion

Through this project, we explored two fundamentally different approaches to solving Sudoku puzzles: a heuristic-driven Constraint Satisfaction Problem (CSP) solver and a probabilistic Monte Carlo Tree Search (MCTS) solver. Our experiments demonstrated that the CSP approach, especially when enhanced with Minimum Remaining Values (MRV) and Least Constraining Value (LCV) heuristics, was highly effective across all puzzle difficulty levels, achieving near-perfect accuracy and extremely fast solve times. In contrast, the MCTS solver, while flexible and powerful in other domains, struggled with the rigid constraint structure of Sudoku, resulting in significantly slower performance and lower success rates on harder puzzles. These findings reinforced the importance of choosing algorithms that align with the underlying structure of a problem. In particular, problems characterized by strict constraints and single-solution spaces, like Sudoku, benefit greatly from deterministic, logic-driven methods rather than exploratory, simulation-based ones. Completing this project deepened our understanding of both constraint reasoning and

probabilistic search, and highlighted the critical role that domain characteristics play in the success or failure of AI algorithms.

References

Conference Proceedings

[Cazenave, T. 2009.] Nested Monte-Carlo Search. *IJCAI 2009 Conference Proceedings*. https://www.ijcai.org/Proceedings/09/Papers/083.pdf

Conference Paper in Edited Volume

[Coulom, R. 2006.] Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *Computers and Games (CG 2006)*, LNCS 4630, 72–83. https://link.springer.com/chapter/10.1007/978-3-540-75538-8_7

Dataset

[Vopani. 2019.] 9 Million Sudoku Puzzles and Solutions. *Kaggle*. https://www.kaggle.com/datasets/rohanrao/sudoku/data

Web Resource

[Norvig, P. 2006.] Solving Every Sudoku Puzzle. http://norvig.com/sudoku.html

Journal Article

[Yato, T., and Seta, T. 2003.] Complexity and completeness of finding another solution and its application to puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 86(5): 1052–1060. https://academic.timwylie.com/17CSCI4341/sudoku.pdf

Preprint Server

[Abdool, M. 2023.] Continual Learning and Out of Distribution Generalization in a Systematic Reasoning Task. arXiv preprint. https://mathai2023.github.io/papers/54.pdf