Backtracking

Algorithm definition:

This algorithm is a problem-solving technique based on trial and error. It systematically tries different choices and explores the solution step by step using DFS and Recursion. When a choice leads to a dead end, it backtracks to a previous decision point and tries another path.

This process avoids unnecessary exploration of invalid solutions and makes Backtracking very efficient for solving Sudoku.

Algorithm Steps:

Step 1: Problem Representation

1. **Variable Definition:** Represent each empty cell in the Sudoku grid as a variable.
2. **Domain Assignment:** Assign each variable a domain of potential values (initially 1-9).
3. **Constraint Identification:** Define the rules that must not be violated:
    - **Row Constraint:** No duplicate numbers in the same row.
    - **Column Constraint:** No duplicate numbers in the same column.
    - **Box Constraint:** No duplicate numbers in the same 3×3 subgrid.

Step 2: Recursive Search (The Core)

1. **Select Unassigned Variable:** Choose an empty cell (usually the next available one in row-major order).
2. **Value Assignment:** Select a number from the domain (1-9) that has not been tried yet for this specific cell.

3. **Validity Check:** Before assigning, verify if the number satisfies all constraints (Row, Column, and Box).
    - **If Valid:** Assign the value and move to the next empty cell.
    - **If Invalid:** Try the next available number in the domain.

Step 3: Failure and Backtracking

1. **Dead-end Detection:** If no number from the domain (1-9) can be placed in the current cell without violating constraints, a failure is detected.
2. **The Backtrack:**
    - Undo the current assignment (set the cell back to empty).
    - Return to the **previous** variable/cell.
    - Try the next possible value for that previous cell.

Step 4: Success and Termination

1. **Goal State:** The process continues until there are no more unassigned variables (all cells are filled).
2. **Solution Return:** Once the grid is full and all constraints are met, the algorithm terminates and returns the completed puzzle.

Advantages & Disadvantages:

Advantages	Disadvantages
Guaranteed Completeness: Unlike Hill Climbing or Simulated Annealing, which might get stuck in "local optima" (incorrect partial solutions), Backtracking is a complete algorithm. It is guaranteed to find the correct solution if one exists.	**Lack of Foresight (Blind Search):** Unlike **Forward Checking** and **Constraint Propagation**, pure Backtracking does not "look ahead." it only detects a violation after it happens, whereas other algorithms prune the search space by predicting failures before they occur.

Memory Efficiency: It is far superior to BFS (Breadth-First Search) and A* in terms of memory. While BFS stores all possible states at each level, Backtracking only stores the current path in the recursion stack, maintaining a space complexity of O(n).	**Exponential Time Complexity:** In very difficult puzzles, Backtracking suffers from the "Exponential Explosion." While **A*** uses heuristics to prioritize the best moves, Backtracking may spend hours exploring useless branches of the search tree.
Simplicity and Reliability: Compared to Constraint Propagation or Forward Checking, Backtracking is much easier to implement and debug. It doesn't require complex data structures to track moving domains, making it highly reliable for standard puzzles.	**Redundant Work:** Unlike **Constraint Propagation**, which uses logic to fill cells instantly, Backtracking might try the same failing combinations repeatedly in different branches because it lacks a global logical understanding of the constraints.

Complexity Analysis:

The efficiency of Backtracking depends on the depth of the search tree and the branching factor at each node:

- **Time Complexity:** Classified as **Exponential**. In the context of Sudoku, it is represented as $O(9^n)$, where n is the number of empty cells.
- **Space Complexity:** Represented as $O(n^2)$ (or O(Cells) to store the board state, plus the **Stack Space** required for the recursive calls.

The code analysis:

pp.py

D: > c++ > pp.py > solveSudokuRec

```python
# Function to check if it is safe to place num at mat[row][col]
def isSafe(mat, row, col, num):

    # Check if num exists in the row
    for x in range(9):
        if mat[row][x] == num:
            return False

    # Check if num exists in the col
    for x in range(9):
        if mat[x][col] == num:
            return False

    # Check if num exists in the 3x3 sub-matrix
    startRow = row - (row % 3)
    startCol = col - (col % 3)

    for i in range(3):
        for j in range(3):
            if mat[i + startRow][j + startCol] == num:
                return False

    return True
```

Function (Safety Check)

This function is the **core logical component** of the Sudoku solver. Its primary responsibility is to ensure that placing a number in a specific cell does not violate the fundamental rules of Sudoku.

pp.py

D: > c++ > pp.py > solveSudokuRec

```python

# Function to solve the Sudoku problem
def solveSudokuRec(mat, row, col):
    # base case: Reached nth column of the last row
    if row == 8 and col == 9:
        return True

    # If last column of the row go to the next row
    if col == 9:
        row += 1
        col = 0

    # If cell is already occupied then move forward
    if mat[row][col] != 0:
        return solveSudokuRec(mat, row, col + 1)

    for num in range(1, 10):

        # If it is safe to place num at current position
        if isSafe(mat, row, col, num):
            mat[row][col] = num
            if solveSudokuRec(mat, row, col + 1):
                return True
            mat[row][col] = 0

    return False
```

The Solver Engine: Recursive Backtracking

This function is the **"Main Engine"** of the algorithm. It is responsible for executing the backtracking process through **Self-Recursion** (the function calling itself).

This part do:

1. **Decision Maker:** It acts as the brain that decides which number to place and where to go next.
2. **Navigation & Flow:** It automatically moves from one cell to the next and handles the transition between rows seamlessly.
3. **Clue Protection:** It identifies pre-filled numbers (the original puzzle clues) and ensures they are never changed or overwritten.
4. **The "Undo" Mechanism:** This is the most critical part; if the algorithm hits a dead-end, it has the ability to "undo" its last move (resetting the cell to 0) and try a different path.

pp.py

D: > c++ > pp.py > ...

```python
def solveSudokuRec(mat, row, col):

    return False

def solveSudoku(mat):
    solveSudokuRec(mat, 0, 0)

if __name__ == "__main__":
    mat = [
        [5, 6, 0, 0, 7, 0, 0, 0, 0],
        [6, 0, 0, 1, 9, 5, 0, 0, 0],
        [0, 9, 8, 0, 0, 0, 0, 6, 0],
        [8, 0, 0, 0, 6, 0, 0, 0, 3],
        [4, 0, 0, 8, 0, 3, 0, 0, 1],
        [7, 0, 0, 0, 2, 0, 0, 0, 6],
        [0, 6, 0, 0, 0, 0, 2, 8, 0],
        [0, 0, 0, 4, 1, 9, 0, 0, 5],
        [0, 0, 0, 0, 8, 0, 0, 7, 9]
    ]

    solveSudoku(mat)

    for row in mat:
        print(" ".join(map(str, row)))
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
5 6 0 0 7 0 0 0 0
6 0 0 1 9 5 0 0 0
0 9 8 0 0 0 0 6 0
8 0 0 0 6 0 0 0 3
4 0 0 8 0 3 0 0 1
7 0 0 0 2 0 0 0 6
0 6 0 0 0 0 2 8 0
0 0 0 4 1 9 0 0 5
0 0 0 0 8 0 0 7 9
```

The Entry Point & Final Output

This final part of the code serves as the Program Entry Point, where the data is initialized and the results are displayed.

1. **Wrapper Function: Simplifies the process by starting the solver at the first cell (row 0, column 0) automatically.**
2. **Puzzle Setup: Defines the Sudoku board as a 2D matrix, using 0 to represent unknown values that need to be solved.**
3. **Execution: Triggers the Backtracking engine to fill the entire grid.**
4. **Result Formatting: Iterates through the solved matrix and prints it in a clean, readable format for the user.**

Genetic Algorithm

Algorithm definition:

Genetic algorithms are inspired by the process of natural selection and are commonly used to generate high-quality solutions to optimization and search problems by relying on bio-inspired operators such as mutation, crossover and selection.

How is it fit for sudoku?

A genetic algorithm might find the solution quicker than trying all possible solutions but it does not mean than it will find the solution.
With a large enough population size or number of generations, it will/should most of
the time solve the sudokus. But it happens sometimes not being able to reach a correct solution.
Solving Sudoku puzzles is known to be NP-complete. it is not easy and sometimes it is difficult to solve them with just logic and you need to bet on some answers to move on. That's why genetic algorithm is a good choice as it generates an initial population randomly.

how is it better from other algorithms?

First, basic brute force is not an option; number of essentially different solutions was shown to be just **5,472,730,538**. That would take ages to run. For instance, Using backtracking works fine for small grid sizes but becomes computationally unaffordable for larger grid sizes.

Also Pencil mark can help to fill some cells but it might stop because at some point there is no more cell with single value to safely assign.

Algorithm steps :

1.read the puzzle from the input file.

2.Defining genes and chromosomes:

A gene represents a row of the Sudoku puzzle and is a permutation of the set {1,2,3,4,5,6,7,8,9}. A chromosome consists of 9 genes, each gene representing a row of the actual Sudoku puzzle. The make_gene function creates a gene, while the make_chromosome function creates a chromosome.

3.Making the first generation:

build a population of potential solutions of the problem

The make_population function creates a population of a specified size. Each member of the population is a Sudoku puzzle represented as a chromosome.

4.Fitness function:

The fitness function calculates how "fit" a chromosome (puzzle) is based on the following criteria:

-For each column: subtract (number of times a number is seen) - 1 from the fitness for that number

-For each 3x3 square: subtract (number of times a number is seen) - 1 from the fitness for that number

The higher the fitness, the closer the puzzle is to being solved
so the fitness is calculated and the mating pool is selected.

5.Crossover and mutation:

then the offspring is generated from the mating pool using:

The crossover function takes two chromosomes as input and makes two new chromosomes by combining them. This crossover function decides the parent of each gene separately, so the result is independent of the location of the genes.

The mutation function applies a random change to a chromosome with a specified probability. In this case, the mutation function randomly changes a gene in a chromosome.

6.new generation:

replace the current population with the offsprings.

7.Repeat:

steps 4-6 for a specified number of generations.

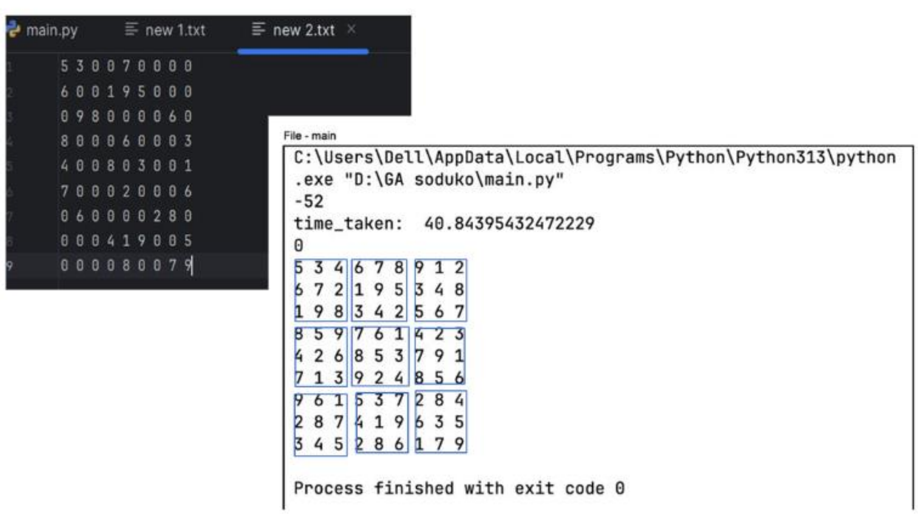
Advantages	disadvantages
GAs use a population-based approach, which allows for the exploration of the solution space by multiple potential solutions simultaneously (parallel search), a capability that can be leveraged for performance	depending on the difficulty level that the program gets stuck in local maximum , and the desired solution cannot be achieved. Due to a lack of diversity
Sudoku puzzles have a vast search space, and GAs are designed to navigate such complex landscapes where traditional gradient-based methods might struggle.	the *"best"* solution is only the best in comparison to other proposals. For optimization problems, we just have a solution which is the best we have found so far, not necessarily the actual optimal solution.
With a large enough population size or number of generations, it will/should Slove the sudoku	it does not mean than it will necessarily find the solution

Can be enhanced with increasing the initial mutation rate, population size, number of iterations. Choosing better selection methods such as tournament selection, or simply running the program more than once cab sometimes help.	since the algorithm's nature involves randomness, it is possible to encounter proplems (such as local minima or number of iterations ending without reaching a solution) no matter what
Trying some initial guesses and building on them is better for NP-problems than brute forcing or other algorithms.	Optimal performance often depends on careful tuning of various parameters such as population size, crossover rate, and mutation rate, which can be a complex task in itself.
might find the solution quicker than trying all possible solutions	

N= population size

	Time complexity	Space complexity
Worst	**O(nlogn)**	**O(n)**
Average	**O(nlogn)**	**O(n)**
Best	**O(nlogn)**	**O(n)**

Sudoku problem and solution:



---

execution time ~= 40s

Solving Sudoku Solver problem with algorithm BFS

Problem definition:

Sudoku is a logic-**based puzzle consisting of a 9×9 grid** divided into nine **3×3 sub grids**. Some cells are pre-filled with numbers **from 1 to 9**, while the remaining cells are empty.

The objective of the problem is to fill all empty cells such that:

- Each row **contains all digits from 1 to 9** exactly once.
- Each column contains all digits from 1 to 9 exactly once.
- **Each 3×3 sub grid** contains all digits from 1 to 9 exactly once.

Sudoku can be modeled as a Constraint Satisfaction Problem (CSP) where the goal is to find a valid assignment of values that satisfies all constraints.

Algorithm Definition:

Breadth-First Search **(BFS) is an uninformed search algorithm that explores the state space level by level.**
In the context of Sudoku:

- Each state represents a partially filled Sudoku board.
- Each action corresponds to **placing a valid number in an empty cell.**
- BFS systematically explores all possible board configurations **until a valid complete solution is found**

State Space Representation:

- Initial State: The original Sudoku board with **empty cells**.
- Goal State: A fully **filled board** that satisfies all Sudoku constraints.

- State Transition (Successor Function):
  Generate new states by placing **a valid number (1–9) into an empty cell without violating any constraints.**

BFS Steps for Solving Sudoku:

1. Insert the initial state into a queue.
2. While the **queue is not empty:**
   1. **Dequeue** the first state.
   2. If the state is a goal state, return it as the solution.
   3. Select the first empty cell in the board.
   4. Try all numbers from 1 to 9:
      - If a number is valid, create a new state by placing it in the selected cell.
      - Enqueue the new state.
3. **If the queue becomes empty without finding a solution, return failure.**

Example and Step-by-Step Execution:

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Step 1: Initial State

The given Sudoku grid is considered the initial state.
This state is inserted into the BFS queue **as the starting point of the search.**

Step 2: Dequeue Initial State

The algorithm removes the **first board** from the **queue** and checks whether it is a **goal state.**
Since the board still contains **empty cells**, the search continues.

Step 3: Select the First Empty Cell

BFS selects the first empty cell using a fixed scanning order (row-wise from left to right).

- First empty cell location: **Row 0, Column 2**

Step 4: Determine Valid Numbers

For the **selected cell (0,2),** the algorithm **checks which numbers from 1 to 9 can be placed** without violating Sudoku constraints.

- Numbers already in **row 0: {5, 3, 7}**
- Numbers already in **column 2: {8}**
- Numbers already in the **3×3 sub grid: {5, 3, 6, 9, 8}**

Valid **candidates: {1, 2, 4}**

Step 5: Generate Successor States

The algorithm creates three new states by placing each valid number in the selected cell

State A

5	3	**1**		7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

State B

5	3	**2**		7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

State C

5	3	**4**		7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

step 6: Breadth-Level Expansion

The BFS algorithm **continues by expanding all states** at the current depth before moving to deeper levels.
For each state:

- The next empty cell is selected
- Valid numbers are tested
- New successor states are generated and enqueued

Step 7: Pruning Invalid States

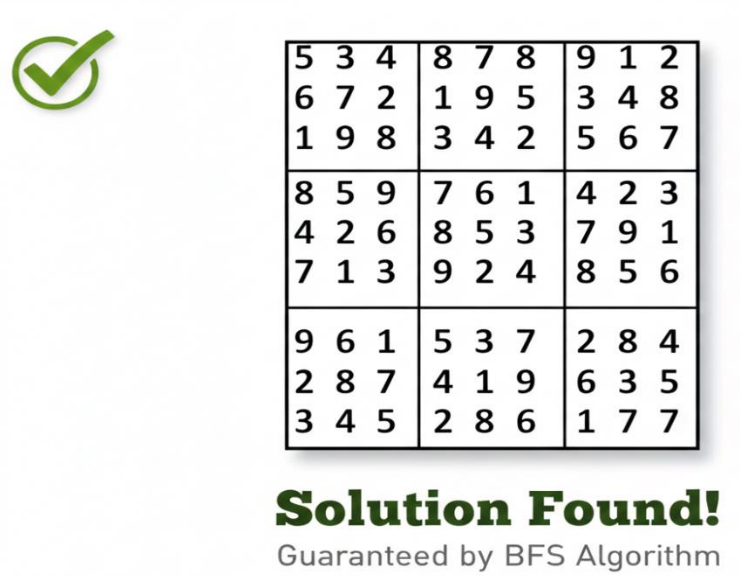
During the search, some states lead to **constraint violations or dead ends.** These states are **discarded and not expanded further.**

Step 8: Reaching the Goal State

Eventually, BFS dequeues a state in which:

- All cells are filled
- All Sudoku constraints are satisfied

This state is identified as the **goal state**.



Time Complexity:

Time and Space Complexity of BFS for Sudoku

Complexity	Aspect
$O(1)$	Constraint Checking
Up to 9	Branching Factor
$O(9^n)$	Search (Worst Case)
Exponential	Overall Time Complexity
$O(9^n)$	Space Complexity

Table X: Complexity analysis of time and space requirements for Breadth-First Search (BFS) in solving the Sudoku problem.

Advantages & disadvantages:

Advantages and Disadvantages of BFS for Solving Sudoku

Aspect	Advantages	Disadvantages
Completeness	Guarantees finding a solution if one exists	May take a very long time to reach the solution
Optimality	Finds the shallowest solution in the search tree	Optimality is not critical for Sudoku problems
Time Complexity	Systematic and level-by-level exploration of states	Exponential time complexity $O(9^n)$
Space Complexity	Explores all possible states	Requires very large memory (high space usage)
Implementation	Easy to understand and implement	Not efficient for complex puzzles
Educational Value	Excellent for demonstrating state-space search concepts	Not suitable for real-world Sudoku solvers
Scalability	Works well for small or simple Sudoku puzzles	Does not scale well with many empty cells

Figure X: Comparison of the advantages and disadvantages of using Breadth-First Search (BFS) for solving the Sudoku problem.

Code Analysis:

Rawda.py > ...

```python
from collections import deque
import copy

# Sudoku input from the image
initial_board = [
    [5,3,0,0,7,0,0,0,0],
    [6,0,0,1,9,5,0,0,0],
    [0,9,8,0,0,0,0,6,0],
    [8,0,0,0,6,0,0,0,3],
    [4,0,0,8,0,3,0,0,1],
    [7,0,0,0,2,0,0,0,6],
    [0,6,0,0,0,0,2,8,0],
    [0,0,0,4,1,9,0,0,5],
    [0,0,0,0,8,0,0,7,9]
]

def is_valid(board, row, col, num):
    # Check row
    if num in board[row]:
        return False

    # Check column
    for i in range(9):
        if board[i][col] == num:
            return False

    # Check 3x3 box
    start_row = row - row % 3
    start_col = col - col % 3
    for i in range(3):
        for j in range(3):
            if board[start_row + i][start_col + j] == num:
                return False

    return True

def find_empty(board):
    for i in range(9):
        for j in range(9):
            if board[i][j] == 0:
                return i, j
    return None

def bfs_sudoku_solver(board):
    queue = deque()
    queue.append(board)

    while queue:
```

This part of the code represents the initial setup of the Sudoku solver. It defines the Sudoku grid as a 9×9 matrix, checks whether a number placement is valid according to row, column, and 3×3 sub grid constraints, identifies empty cells in the board, and prepares a queue structure to apply the Breadth-First Search algorithm for exploring all possible valid board configurations until a solution is found.

Rawda.py > ...

```python
def bfs_sudoku_solver(board):
    queue = deque()
    queue.append(board)

    while queue:
        current = queue.popleft()
        empty = find_empty(current)

        if not empty:
            return current  # Solution found

        row, col = empty

        for num in range(1, 10):
            if is_valid(current, row, col, num):
                new_board = copy.deepcopy(current)
                new_board[row][col] = num
                queue.append(new_board)

    return None

# Solve the Sudoku
solution = bfs_sudoku_solver(initial_board)

# Print solution
for row in solution:
    print(row)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER 5

```
D:\track\Ai_Project(Constraint_propagation)>"C:/Users/maryam elnwehy/AppD
"
[5, 3, 4, 6, 7, 8, 9, 1, 2]
[6, 7, 2, 1, 9, 5, 3, 4, 8]
[1, 9, 8, 3, 4, 2, 5, 6, 7]
[8, 5, 9, 7, 6, 1, 4, 2, 3]
[4, 2, 6, 8, 5, 3, 7, 9, 1]
[7, 1, 3, 9, 2, 4, 8, 5, 6]
[9, 6, 1, 5, 3, 7, 2, 8, 4]
[2, 8, 7, 4, 1, 9, 6, 3, 5]
[3, 4, 5, 2, 8, 6, 1, 7, 9]

D:\track\Ai_Project(Constraint_propagation)>
```

This part of the code implements the core Breadth-First Search (BFS) logic for solving the Sudoku puzzle. It uses a queue to store and explore board states level by level, selects the next empty cell, tries all valid numbers from 1 to 9, and generates new board configurations using deep copies. When a board with no empty cells is reached, the algorithm identifies it as the solution and prints the final solved Sudoku grid.

Constraint Propagation

Algorithm definition:

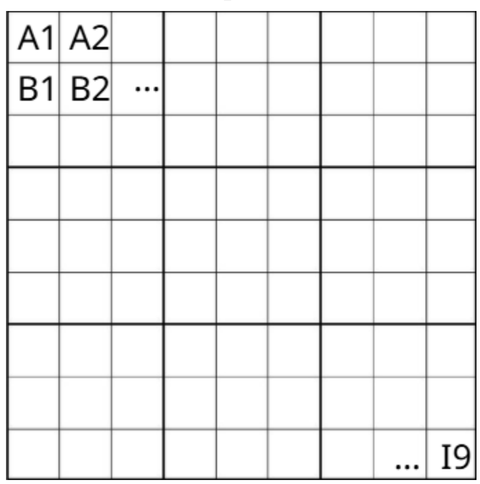
This algorithm is a problem-solving technique used in **Constraint Satisfaction problems (have domain, variables and constraints) CSPs,** and **our problem fits** these types of problems. The main idea of it to **reduce the variable's domain,** this information is **propagated** to related **variables**, potentially causing further reductions.

This technique combines the **constraint propagation** with the **depth-first search (backtracking)** to efficiently solve the sudoku puzzle.

This approach mimics human reasoning by reducing possibilities first, and only guessing when necessary (by using backtracking)

Why does our problem fit this algorithm?

- **Domain**: {1,2,3,4,5,6,7,8,9} and the total variables: **81.**
- **Variables**: ['A1', 'A2', ... ,'I9']



- **Constraints**:
    1. Each row must contain digits from 1-9 exactly once
    2. And each column must contain digits from 1-9 exactly once
    3. And each 3x3 box must contain digits from 1-9 exactly once.

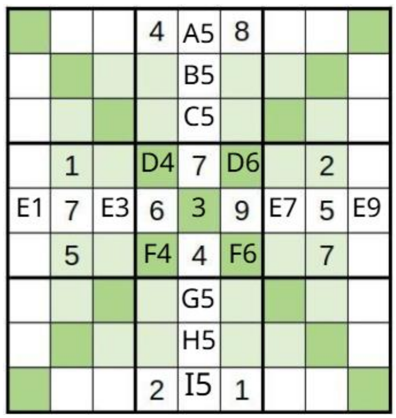
Algorithm steps

Step 1: problem representation

1. Represent each cell as a variable.
2. Assign each variable a domain of possible values (1-9).
3. Define **constraints** based on **rows**, **columns** and **3x3 box**.

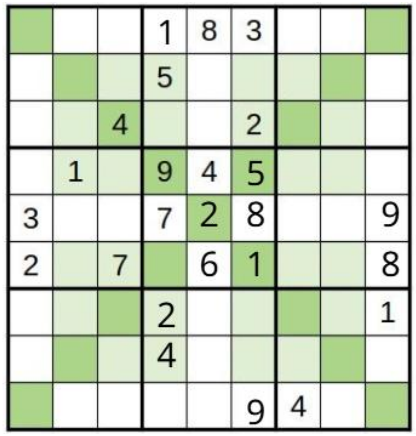
Step 2: Constraints propagation

1. **Elimination Rule**: If a variable is **assigned a single value**, **this value is removed from the domain** of its **peer** variable (the same row, column and 3x3 box)



Example:
As shown in the previous picture, we will **remove the number 3** from the domain of the variables **from A5 to I5** and **from E1 to E9**.

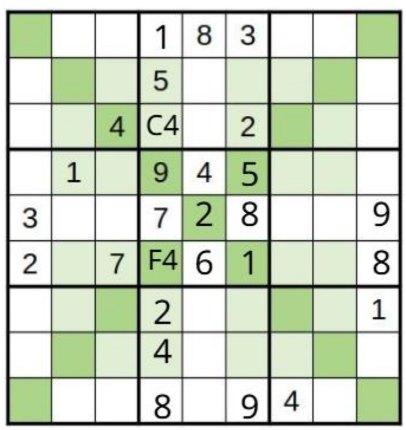
2. **Only choice Rule:** If a number **can go in only one cell inside a group**, then that **cell must** take that **number.**
A group can be:
   - A row
   - A column
   - A 3x3 box



Example:

As shown in the previous picture, **the center 3x3 box has all the values** except for one **value 3** so the last **cell must have 3.**

3. **Naked Twins Rule:** If **two variables within** the same **unit have identical domains containing exactly two variables**, these two values should be assigned to those two variables.
   Therefore, the **two values** can be **eliminated from the domain** of all other variables in the same group.



Example: The **domain of the six groups of C4 and F4** will **eliminate the values (3,6)** as these are the expected values for C4 and F4.

Step 3: propagation

- Any domain reduction **triggers further constraints on affected variables**.
- This process **continues iteratively** until no domains change.

Step 4: failure detection

The constraint propagation process continues iteratively until:

- **No further domain reduction** possible or there **is no change**.
- A contradiction is detected (a variable's domain becomes empty).

Step 5: Termination

- If all variables have a single value, a solution is reached.
- Otherwise, the reduced state is returned for further processing (a search algorithm "backtracking").

Step 6: Search (Backtracking)

If the puzzle is not solved using the constraint propagation:

1. Find the **first empty cell**.
2. Choose a number **from 1 to 9.** Before placing it, **check** that it is not already used in the same group.
3. If the number **violates** any rule, **reject it and try another number**.
4. If the number is **valid,** place it **temporarily** and move to the next empty.
5. Repeat the same steps until the **puzzle is completely filled.**
6. If a cell has **no valid number**, it means the **previous choice was wrong**. **Remove the number** and **backtrack** to the previous cell to try another number.
7. Stop when **no empty cells remain**, meaning the puzzle is fully solved correctly.

Advantages	Disadvantages
Efficient reduction of the search space: Constraint propagation significantly ***reduces the number*** of possible values before any guessing occur.	**Worst-case Exponential time complexity:** The search phase***(backtracking)*** may explore an ***exponential number*** of ***possibilities*** in the worst case, especially for very difficult puzzles.

Mimics Human Problem-solving:	Memory overhead:
The algorithm ***follows the logic reasoning*** used by humans to solve sudoku puzzles.	Backtracking requires ***copying the current puzzle state*** at each ***recursive*** call.
Completeness: When ***combines*** with DFS ***backtracking,*** the algorithm is complete.	**Constraint propagation Alone is not always sufficient:** Some sudoku puzzles ***cannot be solved using logical rules alone*** and require guessing and backtracking.
Generalizable to other CSPs: This approach can be applied to ***other CSP problems*** like scheduling, graph coloring and logic puzzles.	**Implementation complexity:** Compared to ***brute-force*** solutions, this algorithm is more ***complex to implement and understand.***

Time complexity

Aspect	Time complexity
Constraint Propagation	O(1) (for standard Sudoku)
Search (Worst Case)	$O(9^n)$
Overall Time Complexity	Exponential
Space Complexity	$O(n^2)$

The code analysis

SUdoku_CSP0.py

```
# Sudoku Solver (CSP)
# Constraint Propagation + Search(Backtracking)

rows = 'ABCDEFGHI'
cols = '123456789'
digits = '123456789'

#Creates combinations of row labels and column labels Ex: cross("A", "123") → ["A1", "A2", "A3"]
def cross(a, b):
    return [x + y for x in a for y in b]

#Creates combinations of row labels and column labels Ex: ["A1", "A2", ..., "I9"] (81 boxes)
boxes = cross(rows, cols)
row_units = [cross(r, cols) for r in rows]
column_units = [cross(rows, c) for c in cols]
square_units = [
    cross(rs, cs)
    for rs in ('ABC', 'DEF', 'GHI')
    for cs in ('123', '456', '789')
]

# Stores all units in one list
unitlist = row_units + column_units + square_units
"""
units["A1"] →
[
  Row A,
  Column 1,
  Top-left square
]

peers["A1"] →
(all boxes in same row, column, or square except A1)

"""
units = {box: [u for u in unitlist if box in u] for box in boxes}
peers = {box: set(sum(units[box], [])) - {box} for box in boxes}
```

This part of the code creates the combination **of the rows, columns** by the function **cross** and combination of the **3x3 box** by using cross in the variable box. Defining all **constraint groups** where all digits must **be unique. Storing all units in one list**.

SUdoku_CSP0.py

```python
(all boxes in same row, column, or square except A1)

"""
units = {box: [u for u in unitlist if box in u] for box in boxes}
peers = {box: set(sum(units[box], [])) - {box} for box in boxes}

# Convert grid string into a dictionary of possible values
def grid2values(grid):
    values = {}
    for box, char in zip(boxes, grid):
        if char in digits:
            values[box] = char
        else:
            values[box] = digits
    return values


# Display Sudoku grid
def display(values):
    width = 1 + max(len(values[b]) for b in boxes)
    line = '+'.join(['-' * (width * 3)] * 3)

    for r in rows:
        print(
            ''.join(values[r + c].center(width) + ('|' if c in '36' else '')
                    for c in cols)
        )
        if r in 'CF':
            print(line)
    print()

# Constraint Strategies
# Elimination Rule
def eliminate(values):
    solved_boxes = [b for b in boxes if len(values[b]) == 1]

    for box in solved_boxes:
        digit = values[box]
```

This part of the code **converts the grid string into a dictionary of possible values** by putting every **empty** cell for the domain from **1 to 9** and displays the sudoku grid.

The three rules

SUdoku_CSP0.py

```python
# Constraint Strategies
# Elimination Rule
def eliminate(values):
    solved_boxes = [b for b in boxes if len(values[b]) == 1]

    for box in solved_boxes:
        digit = values[box]
        for peer in peers[box]:
            values[peer] = values[peer].replace(digit, '')
    return values

# Only Choice Rule
def only_choice(values):
    for unit in unitlist:
        for digit in digits:
            places = [box for box in unit if digit in values[box]]
            if len(places) == 1:
                values[places[0]] = digit
    return values

# Naked Twins Rule
def naked_twins(values):
    for unit in unitlist:
        twins = [box for box in unit if len(values[box]) == 2]

        seen = {}
        for box in twins:
            val = values[box]
            seen.setdefault(val, []).append(box)

        for val, boxes_with_val in seen.items():
            if len(boxes_with_val) == 2:
                for box in unit:
                    if box not in boxes_with_val:
                        for digit in val:
                            values[box] = values[box].replace(digit, '')
    return values
```

This part is the **most important** part which **distinguishes** the constraint propagation using the **three rules** for reducing the domain.

Termination and backtracking

```python
# Puzzle Reduction: Apply constraint propagation repeatedly
def reduce_puzzle(values):
    stalled = False

    while not stalled:
        solved_before = len([b for b in boxes if len(values[b]) == 1])

        values = eliminate(values)
        values = only_choice(values)
        values = naked_twins(values)

        solved_after = len([b for b in boxes if len(values[b]) == 1])

        stalled = solved_before == solved_after

        if any(len(values[b]) == 0 for b in boxes):
            return False

    return values

# Search (Backtracking)
def search(values):
    """Solve Sudoku using DFS + Constraint Propagation"""
    values = reduce_puzzle(values)
    if values is False:
        return False

    if all(len(values[b]) == 1 for b in boxes):
        return values

    _, box = min((len(values[b]), b) for b in boxes if len(values[b]) > 1)

    for digit in values[box]:
        new_values = values.copy()
        new_values[box] = digit

        attempt = search(new_values)
        if attempt:
            return attempt

    return False
```

In this part we **repeatedly** apply the constraint propagation to reduce the domain as much as possible if we **couldn't solve** it using constraint propagation as we **face a termination condition**, we **complete** it using the **backtracking** search.

Example (run code)

```python
def search(values):
    return False

# Solver
def solve(grid):
    """Solve a Sudoku puzzle"""
    values = grid2values(grid)
    return search(values)

# Example
if __name__ == "__main__":
    puzzle = (
        "53..7...."
        "6..195..."
        ".98....6."
        "8...6...3"
        "4..8.3..1"
        "7...2...6"
        ".6....28."
        "...419..5"
        "....8..79"
    )

    print("Original Sudoku:")
    display(grid2values(puzzle))

    solution = solve(puzzle)

    print("Solved Sudoku:")
    display(solution)
```

```
Original Sudoku:
    5         3     123456789 |123456789     7     123456789 |123456789 123456789 123456789
    6     123456789 123456789 |    1         9         5     |123456789 123456789 123456789
123456789     9         8     |123456789 123456789 123456789 |123456789     6     123456789
------------------------------+------------------------------+------------------------------
    8     123456789 123456789 |123456789     6     123456789 |123456789 123456789     3
    4     123456789 123456789 |    8     123456789     3     |123456789 123456789     1
    7     123456789 123456789 |123456789     2     123456789 |123456789 123456789     6
------------------------------+------------------------------+------------------------------
123456789     6     123456789 |123456789 123456789 123456789 |    2         8     123456789
123456789 123456789 123456789 |    4         1         9     |123456789 123456789     5
123456789 123456789 123456789 |123456789     8     123456789 |123456789     7         9

Solved Sudoku:
5 3 4 |6 7 8 |9 1 2
6 7 2 |1 9 5 |3 4 8
1 9 8 |3 4 2 |5 6 7
------+------+------
8 5 9 |7 6 1 |4 2 3
4 2 6 |8 5 3 |7 9 1
7 1 3 |9 2 4 |8 5 6
------+------+------
9 6 1 |5 3 7 |2 8 4
2 8 7 |4 1 9 |6 3 5
3 4 5 |2 8 6 |1 7 9
4 2 6 |8 5 3 |7 9 1
7 1 3 |9 2 4 |8 5 6
------+------+------
9 6 1 |5 3 7 |2 8 4
2 8 7 |4 1 9 |6 3 5
3 4 5 |2 8 6 |1 7 9
```

References: geeksforgeeks , codinghelmet , ameblo(for the examples)

A* Search Algorithm for Solving Sudoku

Algorithm Definition

A* is an informed search algorithm that finds the **optimal path** from an **initial state** to a **goal state** using a heuristic function.

The evaluation function is defined as: *f(n)*=g(n)+h(n)

Where:

- **g(n)**: the cost from the start node to the current node
- **h(n)**: heuristic estimate of the cost from the current node to the goal
- **f(n)**: total estimated cost
  A* always expands the node with the **smallest f(n)** value

Modeling Sudoku as a Search Problem

Step1: State Representation

Each state represents a complete configuration of the Sudoku board.

- Filled cells contain numbers from 1 to 9
- Empty cells are represented by 0

Modeling Sudoku as a Search Problem

Step1: State Representation

Each state represents a complete configuration of the Sudoku board.

- Filled cells contain numbers from 1 to 9
- Empty cells are represented by 0

Step2: Initial State

The initial state is the given Sudoku puzzle with some cells already filled.

Step3: Goal State

A goal state is reached when:

- All cells are filled
- Every row, column, and 3×3 sub-grid contains each number from 1 to 9 exactly once

Step4: Actions

An action consists of:

- Selecting an empty cell
- Assigning a valid number (1–9) that does not violate Sudoku constraints

Each valid assignment generates a new successor state.

Cost Function and Heuristic

1. Cost Function g(n)

Each assignment of a number to a cell has a uniform cost of 1.
Therefore, **g(n)** represents the number of filled cells added so far.

2. Heuristic Function h(n)

The heuristic used is :

h(n)=number of empty cells

This heuristic:

- Never overestimates the remaining cost
- Is **admissible**
- Guarantees optimality of A*

3. Evaluation Function

The evaluation function combines both costs:

$f(n)=g(n)+h(n)$

The algorithm prioritizes states that are closer to completion.

A* Algorithm for Sudoku (Workflow)

1. Insert the initial Sudoku board into the open list
2. Select the state with the smallest **f(n)**
3. Check if the state is a goal state
4. If not, expand the state by filling a valid number in an empty cell
5. Add all valid successor states to the open list
6. Repeat until a solution is found or no states remain

Code Analysis:

Heuristic Function:

This part of the code defines the heuristic function that estimates the remaining distance to the goal by counting the number of empty cells.

- Each empty cell represents unfinished work
- The heuristic is admissible and safe for A*

```
def heuristic(board):
    # number of empty cells
    return sum(row.count(0) for row in board)
```

Validity Check Function:

This part of the code checks whether placing a number in a specific cell satisfies all Sudoku constraints.

Prevents duplicates in:

- Rows
- Columns
- 3×3 sub-grids

```python
def is_valid(board, row, col, num):
    # check row
    if num in board[row]:
        return False

    # check column
    for i in range(9):
        if board[i][col] == num:
            return False

    # check 3x3 box
    box_row = (row // 3) * 3
    box_col = (col // 3) * 3
    for i in range(box_row, box_row + 3):
        for j in range(box_col, box_col + 3):
            if board[i][j] == num:
                return False

    return True
```

Finding an Empty Cell:

This part of the code searches for the next empty cell in the Sudoku grid.

- Returns the position of the first empty cell
- Returns None if the board is complete

```python
def find_empty(board):
    for i in range(9):
        for j in range(9):
            if board[i][j] == 0:
                return i, j
    return None
```

Initializing the A* Search:

This part of the code initializes the A* search by inserting the initial Sudoku board into the priority queue.

- g represents the path cost so far
- f is the evaluation function

The main loop:

```python
def a_star_sudoku(start_board):
    pq = []

    g = 0
    f = g + heuristic(start_board)

    heapq.heappush(pq, (f, g, start_board))
```

The main loop:

The while pq: loop is the main A* search cycle.
It repeatedly selects the most promising Sudoku board from the priority queue, checks if it's solved, and if not, expands it by filling an empty cell with valid numbers.
Each new board is added back to the queue with updated costs, and the process continues until a solution is found or no states remain.

```python
while pq:
    _, g, board = heapq.heappop(pq)

    empty = find_empty(board)
    if not empty:
        return board  # solved

    row, col = empty

    for num in range(1, 10):
        if is_valid(board, row, col, num):
            new_board = [r[:] for r in board]
            new_board[row][col] = num

            new_g = g + 1
            new_f = new_g + heuristic(new_board)

            heapq.heappush(pq, (new_f, new_g, new_board))

return None
```

Example of code after running:

```
Solved Sudoku:

5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9
PS C:\Users\Sama\OneDrive\Desktop\project_AI>
```

Forward Checking

Algorithm definition:

Forward Checking is a constraint propagation technique that enhances backtracking search by maintaining information about which values are still allowed for each variable (cell) given the current partial assignment.

Instead of blindly trying all possible values, forward checking keeps track of **remaining legal values** for every unassigned cell and prevents assignments that would immediately lead to a conflict.

Representation of Domains

In Sudoku, each cell can take a value from (1 to 9). Forward checking maintains a **domain** for every cell that represents the set of values that are still possible.

This information can be efficiently stored using a **[9 × 9 × 9] Boolean array**, where:

1. The first two dimensions represent the row and column of the cell.
2. The third dimension represents the possible values (1–9).

A value of true indicates that the number is still allowed.

A value of false indicates that the number is not allowed.

(if the third Boolean value of a cell is set to false, then the number **3** cannot be placed in that cell)

Mechanism

Whenever a value **x** is assigned to a specific cell:

I. The algorithm examines all other cells in the **same row, same column**, and **same 3×3 subgrid.**

II. The value **x** is removed (marked as false) from the domains of these **neighboring cells**.

III. This ensures that no future assignment will violate Sudoku constraints.

Forward checking is used in two important ways:

- **Conflict Prevention**: It ensures that no value is assigned that directly conflicts with an already assigned value.
- **Early Failure Detection**: If the domain of any unassigned cell becomes empty (i.e., all values are marked false), the current assignment is invalid, and the algorithm immediately backtracks.

This early detection significantly reduces unnecessary search.

5	6			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1				5
				8			7	9

Step-by-Step Solution

Initial State (Partial Row)

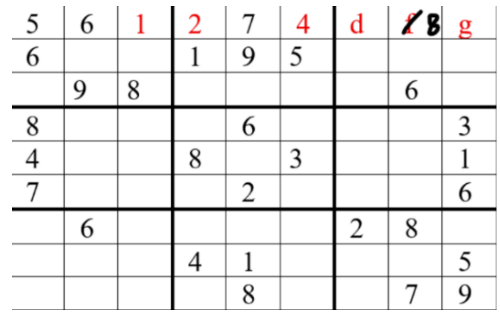
Cell	Domain
a	{1, 2, 3,4}
b	{2, 3}
c	{2,4,8}
d	{1,3,4,8,9}

5	6	a	b	7	c	d	f	g
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1				5
				8			7	9

5	6	a	b	7	c	d	f	g
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1				5
				8			7	9

Cell	Domain
f	{1,2,3,4,9}
g	{2,4,8}

Step 1: Assign a Value

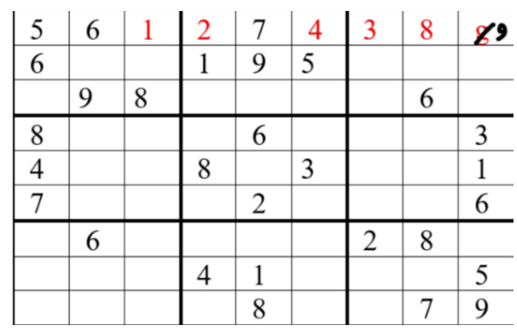


a = 1

Step 2: Forward Checking

Remove 1 from the domains of d and f:

d→ {3,4,8,9}, f→ {2,3,4,9}



b = 2

Remove 2 from the domains of c, f and g:

c→ {4,8}, f→ {3,4,9}, g→ {4,8}

c = 4

Remove 4 from the domains of c, f and g:

d→ {3,8,9}, f→ {3,9}, g→ {8}

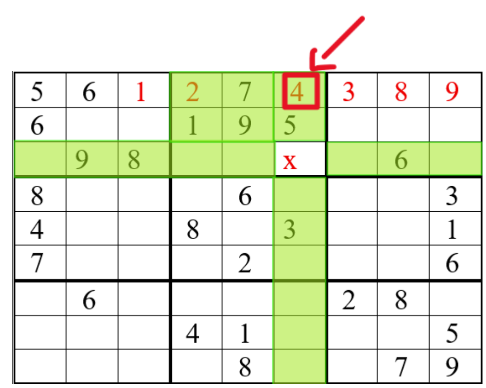
Step 3: Forced Assignment

g = 8 Remove 8 from d: d → {3,9}

d = 3 Remove 3 from the domains of f: f→ {9} then f = 9 (forced)

Final Result

The row is solved without exploring invalid assignments.



After that we again check the next raws

and columns and if there any empty domain

for a cell **like x with { }** here we get in

Step 4: Backtracking

change the previous assignments c to [8] and again forward checking to the next cells

Time Complexity Analysis

Sudoku has 81 variables, each with a maximum domain size of 9.

I. - In the **worst case**, forward checking still has **exponential time complexity**:

$$O(9^{81})$$

II. -However, forward checking significantly reduces the effective branching factor by eliminating inconsistent values early.

III. Compared to pure backtracking, forward checking explores far fewer nodes in the search tree.

Advantages	Disadvantages
Early Detection of Inconsistencies detects constraint If any unassigned variable loses all possible values, the immediately backtracks, preventing unnecessary exploration of invalid path	**Additional Memory Overhead** it requires storing and maintaining domains for each variable, which increases memory usage compared to simple backtracking.
Reduced Search Space: removing inconsistent values from the domains of neighboring variables, reduc the number of candidate assignments, resulting in a smaller and more efficient search tree.	**Limited Look-Ahead Capability** it only considers immediate neighbors an assigned variable. It does not detec deeper inconsistencies that may arise later, unlike Arc Consistency (AC-3)
Improved Performance over Simple Backtracking: Compared to plain backtracking, forwa checking avoids many futile recursive c by pruning inconsistent assignments ea making it faster in practice for problems such as Sudoku.	**Domain Restoration Overhead** During backtracking, domains must be restored to their previous states, which introduces additional computational overhead.

Guarantees Correctness When Combined with Backtracking: When integrated with backtracking sear it is complete and sound. never remove values that could be part of a valid solut and guarantees finding a solution if one exists.	**Still Exponential in the Worst Case** Despite its practical efficiency, the theoretical time complexity remains exponential, and forward checking may still take a long time for highly constrai or poorly structured problems.
Simple and Efficient to Implement: easy to implement and does not require complex constraint propagation algorithms, making it suitable for educational and research purposes	**Not a Standalone Solving Method** Forward checking alone is incomplete and cannot guarantee finding a solutio without being combined with backtracking or another complete sear strategy.

backtracking search, the algorithm becomes **complete.** This means that if a solution exists, the algorithm is **guaranteed to find** it, and if no solution exists, it will **terminate** after exploring all possible assignments. Forward checking only removes values that directly violate current constraints and therefore never eliminates values that could belong to a valid solution.

Code Analysis

```
C: > Users > NVIDIA > sudoku code.py > print_grid
N = 9
VALUES = set(range(1, 10))

def print_grid(grid):
    for i in range(N):
        if i % 3 == 0 and i != 0:
            print("-" * 21)
        for j in range(N):
            if j % 3 == 0 and j != 0:
                print("|", end=" ")
            print(grid[i][j], end=" ")
        print()
    print()


def is_complete(grid):
    return all(all(cell != 0 for cell in row) for row in grid)


def neighbors(r, c):
    nbrs = set()
    for i in range(N):
        nbrs.add((r, i))
        nbrs.add((i, c))
    br, bc = (r // 3) * 3, (c // 3) * 3
    for i in range(br, br + 3):
        for j in range(bc, bc + 3):
            nbrs.add((i, j))
    nbrs.remove((r, c))
    return nbrs


def prune(domains, r, c, val):
    for (nr, nc) in neighbors(r, c):
        domains[(nr, nc)].discard(val)

```

```
def init_domains(grid):
    domains = {}
    for r in range(N):
        for c in range(N):
            if grid[r][c] == 0:
                domains[(r, c)] = VALUES.copy()
            else:
                domains[(r, c)] = {grid[r][c]}

    # Initial constraint propagation
    for r in range(N):
        for c in range(N):
            if grid[r][c] != 0:
                prune(domains, r, c, grid[r][c])

    return domains


def forward_check(domains, r, c, val):
    removed = []
    for (nr, nc) in neighbors(r, c):
        if val in domains[(nr, nc)]:
            domains[(nr, nc)].remove(val)
            removed.append((nr, nc, val))
            if len(domains[(nr, nc)]) == 0:
                return False, removed
    return True, removed


def restore(domains, removed):
    for r, c, v in removed:
        domains[(r, c)].add(v)

```

```python
def assign_single_domains(grid, domains):
    """
    Assign all cells that have exactly one remaining value.
    """
    changed = True
    while changed:
        changed = False
        for (r, c), dom in domains.items():
            if grid[r][c] == 0 and len(dom) == 1:
                val = next(iter(dom))
                grid[r][c] = val
                prune(domains, r, c, val)
                changed = True


def select_mrv(domains, grid):
    unassigned = [
        (cell, len(domains[cell]))
        for cell in domains
        if grid[cell[0]][cell[1]] == 0
    ]

    if not unassigned:
        return None

    return min(unassigned, key=lambda x: x[1])[0]


def solve(grid, domains):
    assign_single_domains(grid, domains)

    if is_complete(grid):
        return True

    cell = select_mrv(domains, grid)
    if cell is None:
        return False

    r, c = cell
```

```python
    for val in sorted(domains[(r, c)]):
        grid[r][c] = val
        old_domain = domains[(r, c)]
        domains[(r, c)] = {val}

        ok, removed = forward_check(domains, r, c, val)
        if ok and solve(grid, domains):
            return True

        # Backtrack
        grid[r][c] = 0
        domains[(r, c)] = old_domain
        restore(domains, removed)

    return False

if __name__ == "__main__":

    sudoku = [
        [5, 3, 0, 0, 7, 0, 0, 0, 0],
        [6, 0, 0, 1, 9, 5, 0, 0, 0],
        [0, 9, 8, 0, 0, 0, 0, 6, 0],
        [8, 0, 0, 0, 6, 0, 0, 0, 3],
        [4, 0, 0, 8, 0, 3, 0, 0, 1],
        [7, 0, 0, 0, 2, 0, 0, 0, 6],
        [0, 6, 0, 0, 0, 0, 2, 8, 0],
        [0, 0, 0, 4, 1, 9, 0, 0, 5],
        [0, 0, 0, 0, 8, 0, 0, 7, 9]
    ]

    print("Initial Sudoku:")
    print_grid(sudoku)

    domains = init_domains(sudoku)

    if solve(sudoku, domains):
        print("Solved Sudoku:")
        print_grid(sudoku)
    else:
        print("No solution found.")
```

```
Initial Sudoku:
5 3 0 | 0 7 0 | 0 0 0
6 0 0 | 1 9 5 | 0 0 0
0 9 8 | 0 0 0 | 0 6 0
---------------------
8 0 0 | 0 6 0 | 0 0 3
4 0 0 | 8 0 3 | 0 0 1
7 0 0 | 0 2 0 | 0 0 6
---------------------
0 6 0 | 0 0 0 | 2 8 0
0 0 0 | 4 1 9 | 0 0 5
0 0 0 | 0 8 0 | 0 7 9

Solved Sudoku:
5 3 4 | 6 7 8 | 9 1 2
6 7 2 | 1 9 5 | 3 4 8
1 9 8 | 3 4 2 | 5 6 7
---------------------
8 5 9 | 7 6 1 | 4 2 3
4 2 6 | 8 5 3 | 7 9 1
7 1 3 | 9 2 4 | 8 5 6
---------------------
9 6 1 | 5 3 7 | 2 8 4
2 8 7 | 4 1 9 | 6 3 5
3 4 5 | 2 8 6 | 1 7 9
```

Simulated Annealing

1. Overview

This project demonstrates how **Simulated Annealing (SA)**, a probabilistic optimization algorithm, can be used to solve a **Sudoku puzzle**.

Sudoku is normally a **constraint satisfaction problem**, but here it is converted into an **optimization problem**:

- A *good* solution has **few rule violations**
- A *perfect* solution has **zero violations**

The algorithm starts with a **random valid filling** (respecting 3×3 blocks) and gradually improves the solution by **random swaps**, sometimes accepting worse states to escape local min.

---

2. What is Simulated Annealing (Intuition)

Simulated annealing is inspired by **cooling metal**:

- At high temperature → atoms move freely (more randomness)
- At low temperature → system becomes stable (less randomness)

In optimization:

- Start with a **high temperature** → accept bad moves sometimes
- Gradually **cool down** → become greedier

Acceptance probability:

$P = \exp(-\Delta cost / temperature)$

Where:

- $\Delta cost < 0$ → always accept (better solution)
- $\Delta cost > 0$ → sometimes accept (worse solution)

---

3. Sudoku Representation

Input Sudoku

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

---

4. Fixed vs Free Cells

- **Fixed cells**: Given by the puzzle (cannot change)
- **Free cells**: Filled randomly and swapped during optimization

```
def FixSudokuValues(fixed_sudoku):  1 usage
    for i in range(0, 9):
        for j in range(0, 9):
            if fixed_sudoku[i, j] != 0:
                fixed_sudoku[i, j] = 1

    return (fixed_sudoku)
```

FixSudokuValues

Creates a binary matrix:

- 1 → fixed cell
- 0 → free cell

This ensures the algorithm never modifies given numbers.

5. Cost Function (Core Idea)

Goal

Minimize the number of **rule violations**.

Rules Checked

- Duplicate numbers in **rows**
- Duplicate numbers in **columns**

(3×3 blocks are always valid by construction)

```python
# Cost Function
def CalculateNumberOfErrors(sudoku):
    numberOfErrors = 0
    for i in range(0, 9):
        numberOfErrors += CalculateNumberOfErrorsRowColumn(i, i, sudoku)
    return (numberOfErrors)


def CalculateNumberOfErrorsRowColumn(row, column, sudoku):
    numberOfErrors = (9 - len(np.unique(sudoku[:, column]))) + (9 - len(np.unique(sudoku[row, :])))
    return (numberOfErrors)
```

CalculateNumberOfErrors

Counts total duplicates across all rows and columns.

If **cost = 0** → Sudoku is solved.

6. 3×3 Block Handling

```python
def CreateList3x3Blocks():
    finalListOfBlocks = []
    for r in range(0, 9):
        tmpList = []
        block1 = [i + 3 * ((r) % 3) for i in range(0, 3)]
        block2 = [i + 3 * math.trunc((r) / 3) for i in range(0, 3)]
        for x in block1:
            for y in block2:
                tmpList.append([x, y])
        finalListOfBlocks.append(tmpList)
    return (finalListOfBlocks)
```

CreateList3x3Blocks

Creates a list of all 9 blocks, where each block contains the coordinates of its 9 cells.

Why blocks matter

- Random filling ensures **each block contains digits 1–9 exactly once**
- Moves are restricted **inside a block**, preserving block validity

---

7. Initial Random State

RandomlyFill3x3Blocks

For every 3×3 block:

- Fill empty cells with numbers 1–9
- Ensure no duplicates *inside the block*

After this step:

- Blocks are valid
- Rows and columns may contain duplicates

This is the **starting state** for simulated annealing.

```
def RandomlyFill3x3Blocks(sudoku, listOfBlocks):  1 usage
    for block in listOfBlocks:
        for box in block:
            if sudoku[box[0], box[1]] == 0:
                currentBlock = sudoku[block[0][0]:(block[-1][0] + 1), block[0][1]:(block[-1][1] + 1)]
                sudoku[box[0], box[1]] = choice([i for i in range(1, 10) if i not in currentBlock])
    return sudoku
```

---

8. Generating New States (Moves)

Key Idea

Randomly swap **two non-fixed cells inside the same block**.

Functions Involved

TwoRandomBoxesWithinBlock

- Randomly selects two free cells in a block

```
def TwoRandomBoxesWithinBlock(fixedSudoku, block):  1 usage
    while (1):
        firstBox = random.choice(block)
        secondBox = choice([box for box in block if box is not firstBox])

        if fixedSudoku[firstBox[0], firstBox[1]] != 1 and fixedSudoku[secondBox[0], secondBox[1]] != 1:
            return ([firstBox, secondBox])
```

FlipBoxes

- Swaps their values

```
def FlipBoxes(sudoku, boxesToFlip):  1 usage
    proposedSudoku = np.copy(sudoku)
    placeHolder = proposedSudoku[boxesToFlip[0][0], boxesToFlip[0][1]]
    proposedSudoku[boxesToFlip[0][0], boxesToFlip[0][1]] = proposedSudoku[boxesToFlip[1][0], boxesToFlip[1][1]]
    proposedSudoku[boxesToFlip[1][0], boxesToFlip[1][1]] = placeHolder
    return (proposedSudoku)
```

ProposedState

- Chooses a random block
- Swaps two free cells
- Produces a *neighbor state*

This defines the **neighborhood structure** of the search.

```
def ProposedState(sudoku, fixedSudoku, listOfBlocks):  2 usages
    randomBlock = random.choice(listOfBlocks)

    if SumOfOneBlock(fixedSudoku, randomBlock) > 6:
        return (sudoku, 1, 1)
    boxesToFlip = TwoRandomBoxesWithinBlock(fixedSudoku, randomBlock)
    proposedSudoku = FlipBoxes(sudoku, boxesToFlip)
    return ([proposedSudoku, boxesToFlip])
```

---

9. Choosing Whether to Accept a Move

ChooseNewState

Steps:

1. Compute cost before move
2. Compute cost after move
3. Calculate cost difference Δcost
4. Accept move with probability:

exp(-Δcost / sigma)

Where sigma = temperature.

This allows:

- Always accepting better states
- Sometimes accepting worse states (escaping local minima)

```python
def ChooseNewState(currentSudoku, fixedSudoku, listOfBlocks, sigma):  1 usage
    proposal = ProposedState(currentSudoku, fixedSudoku, listOfBlocks)
    newSudoku = proposal[0]
    boxesToCheck = proposal[1]
    currentCost = CalculateNumberOfErrorsRowColumn(boxesToCheck[0][0], boxesToCheck[0][1],
                                                   currentSudoku) + CalculateNumberOfErrorsRowColumn(boxesToCheck[1][0],
                                                                                                     boxesToCheck[1][1],
                                                                                                     currentSudoku)
    newCost = CalculateNumberOfErrorsRowColumn(boxesToCheck[0][0], boxesToCheck[0][1],
                                               newSudoku) + CalculateNumberOfErrorsRowColumn(boxesToCheck[1][0],
                                                                                             boxesToCheck[1][1],
                                                                                             newSudoku)
    # currentCost = CalculateNumberOfErrors(currentSudoku)
    # newCost = CalculateNumberOfErrors(newSudoku)
    costDifference = newCost - currentCost
    rho = math.exp(-costDifference / sigma)
    if (np.random.uniform( low: 1,  high: 0,  size: 1) < rho):
        return ([newSudoku, costDifference])
    return ([currentSudoku, 0])
```

---

10. Temperature Initialization

CalculateInitialSigma

- Generates several random moves
- Computes standard deviation of costs
- Uses this as the **initial temperature**

This adapts SA to the difficulty of the puzzle.

```
def CalculateInitialSigma(sudoku, fixedSudoku, listOfBlocks):  1 usage
    listOfDifferences = []
    tmpSudoku = sudoku
    for i in range(1, 10):
        tmpSudoku = ProposedState(tmpSudoku, fixedSudoku, listOfBlocks)[0]
        listOfDifferences.append(CalculateNumberOfErrors(tmpSudoku))
    return (statistics.pstdev(listOfDifferences))
```

---

11. Number of Iterations per Temperature

ChooseNumberOfItterations

- Based on number of fixed cells
- More free cells → more exploration needed

This is called **homogeneous simulated annealing**.

```
def CalculateInitialSigma(sudoku, fixedSudoku, listOfBlocks):  1 usage
    listOfDifferences = []
    tmpSudoku = sudoku
    for i in range(1, 10):
        tmpSudoku = ProposedState(tmpSudoku, fixedSudoku, listOfBlocks)[0]
        listOfDifferences.append(CalculateNumberOfErrors(tmpSudoku))
    return (statistics.pstdev(listOfDifferences))
```

---

12. Cooling Schedule

- Cooling factor = 0.99
- After each temperature cycle:

sigma = sigma * 0.99

High temperature → exploratory
Low temperature → greedy

---

13. Stuck Detection and Reheating

If the score does not improve for many cycles:

- Increase temperature slightly

This avoids getting stuck in **local minima**.

---

14. Main Solver Loop

solveSudoku

Overall process:

1. Fix original cells
2. Randomly fill blocks
3. Initialize temperature
4. Repeat until solved:
    - Try many neighbor states
    - Accept or reject moves
    - Reduce temperature
    - Detect stagnation

Stops when:

CalculateNumberOfErrors == 0

```python
def solveSudoku(sudoku):  1 usage
    f = open("demofile2.txt", "a")
    solutionFound = 0
    while (solutionFound == 0):
        decreaseFactor = 0.99
        stuckCount = 0
        fixedSudoku = np.copy(sudoku)
        PrintSudoku(sudoku)
        FixSudokuValues(fixedSudoku)
        listOfBlocks = CreateList3x3Blocks()
        tmpSudoku = RandomlyFill3x3Blocks(sudoku, listOfBlocks)
        sigma = CalculateInitialSigma(sudoku, fixedSudoku, listOfBlocks)
        score = CalculateNumberOfErrors(tmpSudoku)
        itterations = ChooseNumberOfItterations(fixedSudoku)
        if score <= 0:
            solutionFound = 1
```

```python
        while solutionFound == 0:
            previousScore = score
            for i in range(0, itterations):
                newState = ChooseNewState(tmpSudoku, fixedSudoku, listOfBlocks, sigma)
                tmpSudoku = newState[0]
                scoreDiff = newState[1]
                score += scoreDiff
                print(score)
                f.write(str(score) + '\n')
                if score <= 0:
                    solutionFound = 1
                    break

            sigma *= decreaseFactor
            if score <= 0:
                solutionFound = 1
                break
            if score >= previousScore:
                stuckCount += 1
            else:
                stuckCount = 0
            if (stuckCount > 80):
                sigma += 2
            if (CalculateNumberOfErrors(tmpSudoku) == 0):
                PrintSudoku(tmpSudoku)
                break
```

---

15. Why This Works

- Randomness explores the search space

- Cost function guides improvement
- Temperature balances exploration vs exploitation

This is **not brute force**:

- Does not try all solutions
- Uses probability + structure

---

16. Limitations

- No guarantee of solution
- Slower than backtracking
- Requires parameter tuning

Used mainly for:

- Learning optimization
- Research

Hill Climbing

Algorithm Definition | What is Hill Climbing?

Hill Climbing is a **local search Algorithm**.
It does not operate on a complete search tree; instead, it only considers the **current state** and its **neighbouring states**.

The algorithm moves by comparing heuristic values and selecting a neighbouring state with a **better heuristic value**.
Depending on the problem definition, this process may lead to a **local minimum or a local maximum**, where the solution is better than the current one but not necessarily the best overall.

Hill Climbing does not always guarantee reaching the goal state because:

- It does not perform backtracking
- It does not explore the entire search space
- It follows a greedy approach

Why Does Our Problem Fit This Algorithm? (Sudoku)

Sudoku can be modelled as a Constraint Satisfaction Problem (CSP) where the objective is to minimize constraint violations.

Variables:
Each cell in the Sudoku grid represents a variable.
For a standard 9×9 Sudoku puzzle, there are 81 variables in total.

Domain:
The domain of each variable consists of the digits
{1, 2, 3, 4, 5, 6, 7, 8, 9}.
For pre-filled cells (clues), the domain is restricted to a single fixed value.

Constraints:
The Sudoku problem is governed by the following constraints:

- **Row constraints:**
  No number may appear more than once in any row.
- **Column constraints:**
  No number may appear more than once in any column.
- **3×3 sub grid constraints:**
  No number may be repeated within any 3×3 sub grid.

A solution is obtained when all variables are assigned values from their domains while satisfying all constraints.

Hill Climbing is suitable for Sudoku because it is an optimization-based approach that improves an already complete solution rather than building it incrementally.

This algorithm fits the Sudoku problem because:

- A Sudoku puzzle can be represented as a complete state
- A clear heuristic function can be defined based on the number of conflicts
- Neighbouring states can be generated using controlled swaps

---

Algorithm Steps | Solving Sudoku Using Hill Climbing

Hill Climbing does not solve Sudoku cell by cell like a human solver.
Instead, it starts with a **complete but imperfect solution** and improves it iteratively.

Step 1: Initial State

- The given numbers (clues) are fixed and cannot be changed.
- The Sudoku grid is filled **row by row**.
- Each row is filled randomly using the numbers 1 to 9 without repetition.

This guarantees that all rows are valid from the beginning.

Step 2: Fixing Rows

By fixing the rows, we start from a complete current state with one constraint already satisfied.
This allows the algorithm to focus only on optimizing **columns and 3×3 sub grids**.

At this stage, the Sudoku grid is complete but may contain conflicts.

Step 3: Heuristic Function

The heuristic value (h) is defined as:

- The number of conflicts in columns
- Plus the number of conflicts in 3×3 sub grids

The goal is to reach:

h = 0

Step 4: Generating Neighbor States

The only allowed operation is:

- Swapping two **non-fixed** values
- Within the **same row**

This preserves row validity.

Step 5: Hill Climbing Process

- Perform a swap
- Compute the new heuristic value
- If the heuristic value decreases, accept the new state
- If the heuristic value increases, reject the move and try another swap
- Repeat until h = 0 or no further improvement is possible

Advantages & Disadvantages of Using Hill Climbing for Sudoku:

Advantages	Disadvantages
Fast conflict reduction **Hill Climbing quickly reduces the number of conflicts by iteratively improving the current Sudoku state using a well-defined heuristic.**	**May get stuck in local optima** Hill Climbing may stop at a near-solution where no local swap reduces conflicts, even though a valid solution exists.
Works well with a clear heuristic function **Sudoku has a natural heuristic (number of conflicts in columns and 3×3 sub grids), which makes Hill Climbing effective for this problem.**	**Sensitive to the initial random filling** A poor initial configuration can prevent the algorithm from reaching a complete solution.

Maintains row validity throughout the search **By allowing swaps only within the same row, row constraints remain satisfied during the entire solving process.**	**No guarantee of finding a solution** Hill Climbing does not guarantee reaching a conflict-free Sudoku solution without additional techniques such as random restarts.

Time Complexity

Case	Time Complexity	Explanation
Best Case	O(1)	The algorithm reaches a solution immediately when the initial state has no conflicts.
Average Case	O(K)	The algorithm performs K improvement iterations, where each iteration evaluates a constant number of neighbours.
Worst Case	O(K)	The algorithm continues until it reaches a local optimum after K iterations.
Overall Time Complexity	O(K)	Since the Sudoku grid size is fixed, all operations per iteration run in constant time.
Space Complexity	O(1)	The algorithm stores only a fixed-size grid and its neighbouring states.

The value of K depends on the initial configuration and the presence of local optima.

Comparison between Sudoku Solving Algorithms

BFS, DFS, Backtracking, Forward Checking, Constraint Propagation, A*, Hill Climbing, Simulated Annealing

1. Execution Time

Algorithm	Execution Time
BFS	Very slow due to exploring all states level by level
Genetic	Could be faster than the backtracking depending on the conditions.
Backtracking	Faster because it abandons invalid assignments early
Forward Checking	Fast due to early elimination of inconsistent values
Constraint Propagation	Very fast; solves many cells logically before search
A*	Moderate to slow due to heuristic evaluation and queue operations
Hill Climbing	Fast per run but unreliable due to local optima
Simulated Annealing	Slow because of many iterations and cooling schedule

Ranking:

Constraint Propagation->Forward Checking→ ->backtracking-> Genetic Algorithm→ A* → Hill Climbing→ Simulated Annealing→ BFS

2. Space Complexity

Algorithm	Space Complexity
BFS	Very high memory usage due to storing all frontier states
Genetic algorithm	Stores entire population; moderate memory
Backtracking	Low to moderate memory usage
Forward Checking	Moderate memory usage due to domain tracking
Constraint Propagation	Moderate memory usage for domains and propagation
A*	Very high memory usage (open and closed lists)
Hill Climbing	Very low memory usage
Simulated Annealing	Low memory usage

Ranking (best memory efficiency):
Hill Climbing ≈ Simulated Annealing > genetic algorithm> Backtracking > Forward Checking ≈ Constraint Propagation > BFS ≈ A*

---

3. Pruning Power

Algorithm	Pruning Capability
BFS	No pruning capability
Genetic algorithm	Implicit pruning via fitness function.
Backtracking	Prunes when constraints are violated
Forward Checking	Strong pruning by eliminating future invalid values

Algorithm	Pruning Capability
Constraint Propagation	Very strong pruning using logical rules
A*	Moderate pruning through heuristic guidance
Hill Climbing	Weak pruning
Simulated Annealing	Weak pruning

Ranking:

Constraint Propagation > Forward Checking > Backtracking > A* > genetic algorithm> Simulated Annealing > Hill Climbing > BFS

---

4. Expanded Nodes

Algorithm	Expanded Nodes
BFS	Expands the largest number of nodes
	Explores population members.
Backtracking	Expands fewer nodes
Forward Checking	Expands very few nodes
Constraint Propagation	Expands the fewest nodes
A*	Expands many nodes depending on heuristic
Hill Climbing	Expands few nodes but lacks exploration
Simulated Annealing	Expands a very large number of states

Ranking (fewer is better):

Constraint Propagation < Forward Checking < genetic

algorithm<Backtracking < Hill Climbing < DFS < A* < BFS < Simulated Annealing

---

5. Constraint Handling

Algorithm	Constraint Handling
BFS	Checks constraints only after assignment
Genetic algorithm	Enforces constraints indirectly via fitness function or repair mechanisms.
Backtracking	Checks constraints during assignment
Forward Checking	Actively enforces constraints early
Constraint Propagation	Explicit and continuous constraint enforcement
A*	Enforces constraints during successor generation
Hill Climbing	Implicit (via cost function)
Simulated Annealing	Implicit (via cost function)

Ranking:

Constraint Propagation→ Forward Checking→ Hill Climbing→ Simulated Annealing

→ Genetic Algorithm→ Backtracking→ A*→ BFS

Constraint Propagation > Forward Checking > Backtracking > A* > Simulated Annealing ≈ Hill Climbing > DFS = BFS

---

6. Failure Detection Speed

Algorithm	Failure Detection
BFS	Very slow failure detection
Genetic algorithm	indirect failure detection through fitness evaluation
Backtracking	Faster due to early backtracking
Forward Checking	Fast failure detection
Constraint Propagation	Very fast (empty domain detection)
A*	Slow (after exhausting states)
Hill Climbing	Fast local failure detection
Simulated Annealing	Slow failure detection

Ranking:

Constraint Propagation→ Forward Checking→ Backtracking→ A*→ Genetic Algorithm→ Simulated Annealing→ Hill Climbing→ BFS

---

7. Scalability

Algorithm	Scalability
BFS	Poor scalability
Genetic algorithm	Population-based search can scale to large problems.
Backtracking	Good scalability
Forward Checking	Excellent scalability
Constraint Propagation	Excellent scalability

Algorithm	Scalability
A*	Poor scalability due to memory explosion
Hill Climbing	Poor scalability
Simulated Annealing	Moderate scalability

Ranking:

Constraint Propagation→ Forward Checking→ Genetic Algorithm→ Hill Climbing

→ Simulated Annealing→ Backtracking→ A*→ BFS-style Backtracking

Constraint Propagation > Forward Checking > Backtracking > Simulated Annealing > DFS > Hill Climbing > A* > BFS

---

8. Implementation Complexity

Algorithm	Implementation Difficulty
Genetic algorithm	moderate
BFS	Simple
Backtracking	Moderate
Forward Checking	More complex
Constraint Propagation	Complex
A*	Complex
Hill Climbing	Very simple
Simulated Annealing	Complex

Ranking (easiest to hardest):

Hill Climbing→ Simulated Annealing→ Backtracking→ BFS→ Genetic Algorithm

→ Forward Checking→ Constraint Propagation→ A*

---

9. Optimality

Algorithm	Optimal Solution	Practical Use
Genetic algorithm	No guarantee	Used in research/ experiments
DFS	No	Limited
Backtracking	Yes	Practical
Forward Checking	Yes	Best practical
Constraint Propagation	Yes	Best practical
A*	Yes (with admissible heuristic)	Limited
Hill Climbing	No	Not reliable
Simulated Annealing	No guarantee	Experimental

---

10. Completeness

Algorithm	Completeness
BFS	Complete
Genetic algorithm	incomplete
Backtracking	Complete

Algorithm	Completeness
Forward Checking	Complete
Constraint Propagation	Complete (with backtracking)
A*	Complete
Hill Climbing	Incomplete
Simulated Annealing	Incomplete

Final Conclusion

Constraint-based algorithms (Constraint Propagation and Forward Checking) are the most effective and practical for solving Sudoku. Search-based methods (BFS and A*) suffer from high time or memory costs.

Optimization techniques (Hill Climbing ,genetic algorithm and Simulated Annealing) demonstrate alternative approaches but lack completeness and reliability.
