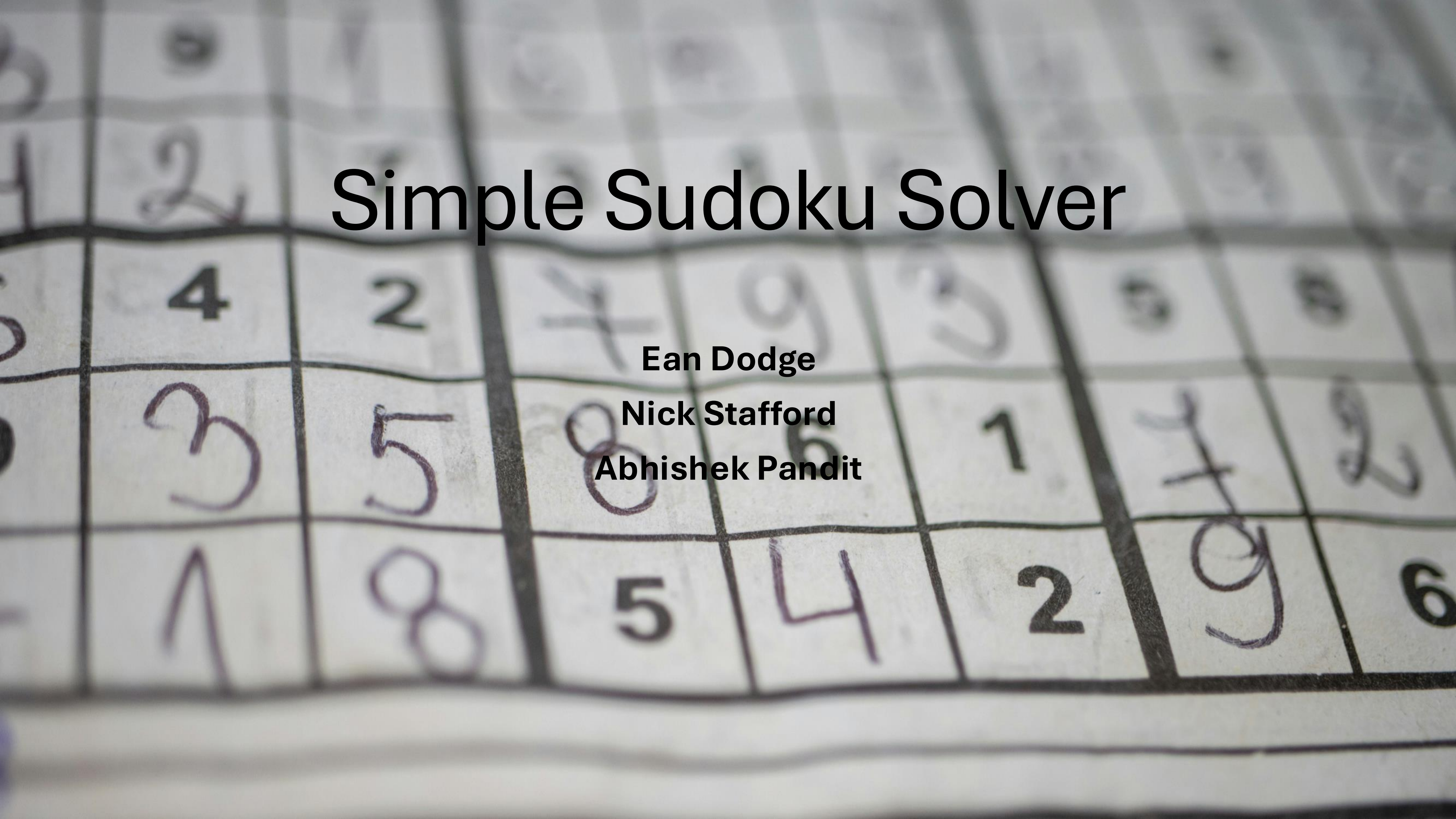


Simple Sudoku Solver

Ean Dodge

Nick Stafford

Abhishek Pandit

Creating the Sudoku Puzzle

- Can accept a custom grid from the user, or automatically generate one
- Partially completed solutions auto generated with py-sudoku

```
def get_puzzle():
    if (
        input(
            "Do you want to enter custom sudoku board or automatically generated board?\n (press 'c' for custom): "
        )
        == "c"
    ):
        custom_grid = _get_custom_puzzle()
        return Sudoku(3, board=custom_grid)
    else:
        # Auto generate a solvable soduku
        difficulty = _get_difficulty()
        return Sudoku(3).difficulty(difficulty)
```

Creating the Sudoku Puzzle

This is the puzzle that is created when automatically generated with a difficulty of .6

```
Do you want to enter custom sudoku board or automatically generated board?
 (press 'c' for custom):
What difficulty would you like the puzzle to be?(0-1): .6
Puzzle has multiple solutions
+-------+-------+-------+
|   1   |   7   | 2     |
| 4     |   1   | 3   7 |
|     7 |       |   5 6 |
+-------+-------+-------+
|   2 1 |       |   7   |
| 7 5 4 |       |     3 |
| 8     |       | 4 6   |
+-------+-------+-------+
| 3 4   |   5 7 | 6   8 |
|       |   4   |       |
| 9 8 5 | 2   1 |     4 |
+-------+-------+-------+
```

Naive solution with backtracking

- Brute force
- Going through every cell to put a value and make sure its valid.
- Recursively calls itself for backtracking.
- Recursive ends if there are no more 0's

```python
def brute_force(board: list[list[int]]) -> bool:
    """
    A brute-force backtracking solution to solve Sudoku.
    """
    for row in range(9):
        for col in range(9):
            if board[row][col] == 0:
                # Try every number from 1 to 9
                for num in range(1, 10):
                    if solver.is_valid(board, row, col, num):
                        board[row][col] = num
                        # print(f"Placing {num} at ({row},{col})...")
                        if brute_force(board):
                            return True
                        board[row][col] = 0
                return False
    return True
```

Heuristics: most constrained

- Finding the most constrained variables
- Solving cells that are most constrained
- Tied cells returned

```
What difficulty would you like the puzzle to be?(0-1): 0.3
Puzzle has exactly one solution
+-------+-------+-------+
| 5 6 7 |   8 1 | 4 2   |
| 8 1   | 2   4 |   3   |
| 2 4 3 | 5     | 6 1   |
+-------+-------+-------+
|       | 8 4 7 | 1 9 6 |
|     4 | 6 1 3 |     2 |
| 1 8 6 | 9 2   | 3 4   |
+-------+-------+-------+
| 4 5 2 | 1 9 6 |   7 3 |
| 9   1 | 7   8 | 2     |
| 6 7 8 |   3 2 | 9 5 1 |
+-------+-------+-------+

this code is working
[(0, 3), (0, 8), (1, 2), (2, 4), (2, 5), (3, 0), (3, 2), (4, 0), (4, 1), (4, 7), (5, 5), (6, 6), (7,
 1), (7, 4), (7, 7), (7, 8), (8, 3)] 1
```

```python
def most_constrained_variables(board: list[list[int]]) -> list[tuple[int, int]]:
    tied_cells: list[tuple[int, int]] = []
    min_valid_values = 10  # Start with a value larger than the max (9)

    for r in range(9):
        for c in range(9):
            if board[r][c] == 0:
                valid_values = [
                    num for num in range(1, 10) if is_valid(board, r, c, num)
                ]
                if len(valid_values) == min_valid_values:
                    # ties are allowed, if two cells are equally constrained
                    tied_cells.append((r, c))
                elif len(valid_values) < min_valid_values:
                    min_valid_values = len(valid_values)
                    # remove old cells, since this cell is more constrined.
                    tied_cells = [(r, c)]

    return tied_cells
```

Heuristics: most constraining

- Find the most constraining variables
- Return the most constraining cell

```python
def most_constraining_variable(
    board: list[list[int]], tied_cells: list[tuple[int, int]]
) -> tuple[int, int]:
    """From a list of tied cells, find the cell that imposes the most constraints on its
    neighbors. if there is a tie, return random of the maxs"""
    max_constraints = -1
    most_constraining_cell = tied_cells[0]
    for r, c in tied_cells:
        constraints = 0

        # counting unassigned cells in the same row and column
        for i in range(9):
            if board[r][i] == 0:
                constraints += 1
            if board[i][c] == 0:
                constraints += 1

        # counting unassigned cells in the subgrid
        start_row, start_col = 3 * (r // 3), 3 * (c // 3)
        for i in range(start_row, start_row + 3):
            for j in range(start_col, start_col + 3):
                if board[i][j] == 0:
                    constraints += 1

        # updating the most constraining variable
        if constraints > max_constraints:
            max_constraints = constraints
            most_constraining_cell = (r, c)

    return most_constraining_cell
```

Heuristics: least constraining

- Find the least constraining variable

```
def least_constraining_values(board: list[list[int]], row: int, col: int) -> list[int]:
    """Get the possible values of a cell, ordered by their least constraining effect"""
    candidates: list[tuple[int, int]] = []
    for num in range(1, 10):
        if is_valid(board, row, col, num):
            # counting how many other cells this value would restrict
            constraint_count = 0
            for r in range(9):
                if board[r][col] == 0 and is_valid(board, r, col, num):
                    constraint_count += 1
            for c in range(9):
                if board[row][c] == 0 and is_valid(board, row, c, num):
                    constraint_count += 1
            candidates.append((num, constraint_count))

    # Sort by the number of constraints (ascending)
    candidates.sort(key=lambda x: x[1])
    return [x[0] for x in candidates]
```

Testing

- Ensure updates don't break code.
- Parameters used to run different tests.

```
(aigroupproject-py3.12) →  aigroupproject git:(master) × python tests.py -h
usage: tests.py [-h] [-lt] [-bf] [-v] [-p] [epochs]

Run multiple Sudoku tests.

positional arguments:
  epochs               Number of test boards to run (default is 1000)

options:
  -h, --help           show this help message and exit
  -lt, --lookup_table  Run the tests with a lookup table
  -bf, --brute_force   Run the tests with the brute force algorithm
  -v, --verbose        Print extra information to the console.
  -p, --parallel       Run the tests in parallel
(aigroupproject-py3.12) →  aigroupproject git:(master) × python tests.py
```

Testing

- Tests algorithm speed through many tests
- Brute force < CSP < CSP with lookup table

```
(aigroupproject-py3.12) ➜  aigroupproject git:(master) ✗ python tests.py --brute_force
Testing 1000 test batch
Successfully passed 1000 tests.
Heuristic solving time: 13.571531 seconds.
(aigroupproject-py3.12) ➜  aigroupproject git:(master) ✗ python tests.py
Testing 1000 test batch
Successfully passed 1000 tests.
Heuristic solving time: 4.515533 seconds.
(aigroupproject-py3.12) ➜  aigroupproject git:(master) ✗ python tests.py --lookup_table
Testing 1000 test batch
Successfully passed 1000 tests.
Heuristic solving time: 2.770192 seconds.
```

Solving backtracking

- Only make moves if they are legal.
- Check how each state change affects other cells before commiting to new state
- Implemented with either a lookup table, or by checking all affected cells first.

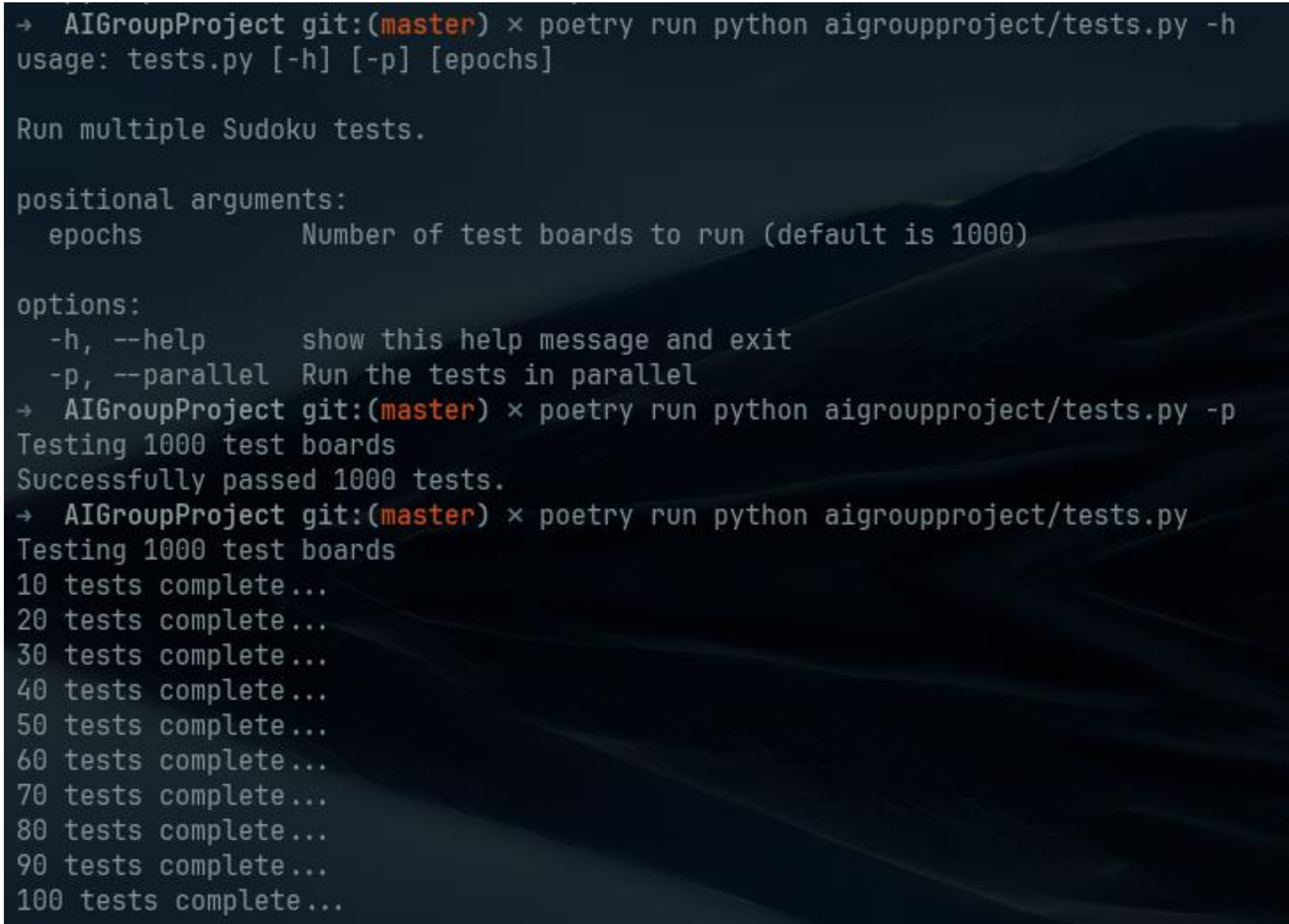
sudoku_solver.py

```python
            # counting how many other cells this value w
            constraint_count = 0
            for r in range(9):
                if board[r][col] == 0 and is_valid(board, r, col, num):
                    constraint_count += 1
            for c in range(9):
                if board[row][c] == 0 and is_valid(board, row, c, num):
                    constraint_count += 1
            candidates.append((num, constraint_count))

    # Sort by the number of constraints (ascending)
    candidates.sort(key=lambda x: x[1])
    return [x[0] for x in candidates]


def solve_heuristics(board: list[list[int]], depth: int = 0) -> bool:
    tab = " " * depth
    print(f"{tab}Depth={depth}")
    # Finding the most constrained variable(s)
    tied_cells: list[tuple[int, int]] = most_constrained_variables(board)
    # print(f"{tab}{len(tied_cells)} most constrained")

    # If the board is solved
    if not tied_cells:
        return True

    # If there's a tie, use Most Constraining Variable to break it
    if len(tied_cells) > 1:
        row, col = most_constraining_variable(board, tied_cells)
    else:
        row, col = tied_cells[0]

    # trying the least constraining values for the selected cell
    for num in least_constraining_values(board, row, col):
        board[row][col] = num
        # print(f"Trying {num} at ({row}, {col})")
        if solve_heuristics(board, depth + 1):
            print(f"{tab}Backtracking Success")
            return True
        print(f"{tab}Failed")
        board[row][col] = 0
        print(f"Backtracking at ({row}, {col})")

    print(f"{tab}Backtracking Fail")
    return False
```

```
Depth=3
Depth=4
Depth=5
Depth=6
Depth=7
Depth=8
Depth=9
Depth=10
Depth=11
Depth=12
Depth=13
Depth=14
Depth=15
Depth=16
Depth=17
Depth=18
Depth=19
Depth=20
Depth=21
Depth=22
Depth=23
Depth=24
Depth=25
Depth=26
Depth=27
Depth=28
Depth=29
Depth=30
Depth=31
Depth=32
Depth=33
Depth=34
Depth=35
Depth=36
Depth=37
Depth=38
Depth=39
Depth=40
Backtracking Success
Backtracking Success
Backtracking Success
Backtracking Success
Backtracking Success
Backtracking Success
Backtracking Success
Backtracking Success
Backtracking Success
Backtracking Success
Backtracking Success
Backtracking Success
Backtracking Success
Backtracking Success
Backtracking Success
Backtracking Success
Backtracking Success
Backtracking Success
Backtracking Success
Backtracking Success
Backtracking Success
Backtracking Success
Backtracking Success
Backtracking Success
Backtracking Success
Backtracking Success
Backtracking Success
```

Parallel Programming

- Multithreading performance increases scale with number of cpu cores
- enabled with the "-p" parameter on cli.



Look up table

```
+-----------------------------------+
| 25679      25679      3           |
| 245679     8          1           |
| 245679     245679     245679      |
+-----------------------------------+
| 123456789  12345679   2456789     |
| 123456789  12345679   2456789     |
| 123456789  12345679   2456789     |
+-----------------------------------+
| 123456789  12345679   2456789     |
| 123456789  12345679   2456789     |
| 123456789  12345679   2456789     |
+-----------------------------------+
```

```
+--------------------------------+--------------------------------+--------------------------------+
| 25679     25679     3          | 256789    4         256789     | 256789    1         256789     |
| 245679    8         1          | 235679    235679    235679     | 2345679   2345679   2345679    |
| 245679    245679    245679     | 12356789  12356789  12356789   | 23456789  23456789  23456789   |
+--------------------------------+--------------------------------+--------------------------------+
| 123456789 12345679  2456789    | 123456789 12356789  123456789  | 123456789 23456789  123456789  |
| 123456789 12345679  2456789    | 123456789 12356789  123456789  | 123456789 23456789  123456789  |
| 123456789 12345679  2456789    | 123456789 12356789  123456789  | 123456789 23456789  123456789  |
+--------------------------------+--------------------------------+--------------------------------+
| 123456789 12345679  2456789    | 123456789 12356789  123456789  | 123456789 23456789  123456789  |
| 123456789 12345679  2456789    | 123456789 12356789  123456789  | 123456789 23456789  123456789  |
| 123456789 12345679  2456789    | 123456789 12356789  123456789  | 123456789 23456789  123456789  |
+--------------------------------+--------------------------------+--------------------------------+

+--------------------------------+--------------------------------+--------------------------------+
| 25679     25679     3          | 26789     4         26789      | 256789    1         256789     |
| 24679     8         1          | 5         23679     23679      | 234679    234679    234679     |
| 245679    245679    245679     | 1236789   1236789   1236789    | 23456789  23456789  23456789   |
+--------------------------------+--------------------------------+--------------------------------+
| 123456789 12345679  2456789    | 12346789  12356789  123456789  | 123456789 23456789  123456789  |
| 123456789 12345679  2456789    | 12346789  12356789  123456789  | 123456789 23456789  123456789  |
| 123456789 12345679  2456789    | 12346789  12356789  123456789  | 123456789 23456789  123456789  |
+--------------------------------+--------------------------------+--------------------------------+
| 123456789 12345679  2456789    | 12346789  12356789  123456789  | 123456789 23456789  123456789  |
| 123456789 12345679  2456789    | 12346789  12356789  123456789  | 123456789 23456789  123456789  |
| 123456789 12345679  2456789    | 12346789  12356789  123456789  | 123456789 23456789  123456789  |
+--------------------------------+--------------------------------+--------------------------------+

+--------------------------------+--------------------------------+--------------------------------+
| 25679     25679     3          | 2789      4         2789       | 256789    1         256789     |
| 2479      8         1          | 5         2379      6          | 23479     23479     23479      |
| 245679    245679    245679     | 123789    123789    123789     | 23456789  23456789  23456789   |
+--------------------------------+--------------------------------+--------------------------------+
| 123456789 12345679  2456789    | 12346789  12356789  12345789   | 123456789 23456789  123456789  |
| 123456789 12345679  2456789    | 12346789  12356789  12345789   | 123456789 23456789  123456789  |
| 123456789 12345679  2456789    | 12346789  12356789  12345789   | 123456789 23456789  123456789  |
+--------------------------------+--------------------------------+--------------------------------+
| 123456789 12345679  2456789    | 12346789  12356789  12345789   | 123456789 23456789  123456789  |
| 123456789 12345679  2456789    | 12346789  12356789  12345789   | 123456789 23456789  123456789  |
| 123456789 12345679  2456789    | 12346789  12356789  12345789   | 123456789 23456789  123456789  |
+--------------------------------+--------------------------------+--------------------------------+

+--------------------------------+--------------------------------+--------------------------------+
| 25679     25679     3          | 2789      4         2789       | 256789    1         256789     |
| 2479      8         1          | 5         279       6          | 3         2479      2479       |
| 245679    245679    245679     | 123789    123789    123789     | 2456789   2456789   2456789    |
+--------------------------------+--------------------------------+--------------------------------+
| 123456789 12345679  2456789    | 12346789  12356789  12345789   | 12456789  23456789  123456789  |
| 123456789 12345679  2456789    | 12346789  12356789  12345789   | 12456789  23456789  123456789  |
| 123456789 12345679  2456789    | 12346789  12356789  12345789   | 12456789  23456789  123456789  |
+--------------------------------+--------------------------------+--------------------------------+
| 123456789 12345679  2456789    | 12346789  12356789  12345789   | 12456789  23456789  123456789  |
| 123456789 12345679  2456789    | 12346789  12356789  12345789   | 12456789  23456789  123456789  |
| 123456789 12345679  2456789    | 12346789  12356789  12345789   | 12456789  23456789  123456789  |
+--------------------------------+--------------------------------+--------------------------------+
```

Lookup Table

- Same function, custom data structure.
- Use a set in each cell which has possible legal values
- Reference to see what values are possible.
- Update after each cell is placed.

```python
class State:
    table: list[list[set[int]]]
    is_set: set[tuple[int, int]]
    is_not_set: set[tuple[int, int]]
    verbose: bool = False

    def __init__(self, grid: list[list[int]], verbose: bool = False) → None:
        self.table = []
        self.is_set = set()
        self.is_not_set = set()
        self.verbose = verbose
        for r in range(9):
            row: list[set[int]] = []
            for c in range(9):
                row.append(set(range(1, 10)))
                self.is_not_set.add((r, c))
            self.table.append(row)

        self.init_table(grid)

    def is_valid(self, row: int, col: int, num: int) → bool:
        return (row, col) in self.is_not_set and num in self.table[row][col]

    def init_table(self, grid: list[list[int]]):
        for row in range(9):
            for col in range(9):
                cell = grid[row][col]
                if cell ≠ 0:
                    self.constrain(row, col, cell)
```

```python
    def constrain(self, row: int, col: int, val: int, verbose: bool = False):
        for i in range(9):
            self.table[row][i].discard(val)
            self.table[i][col].discard(val)

        start_row, start_col = row - row % 3, col - col % 3
        for i in range(3):
            for j in range(3):
                self.table[i + start_row][j + start_col].discard(val)
        self.table[row][col] = set([val])
        self.is_set.add((row, col))
        self.is_not_set.remove((row, col))
        if verbose or self.verbose:
            print(self.format_as_string(row, col))

    def constrain_trivial_cells(self) → bool:
        did_update = False
        to_update = deepcopy(self.is_not_set)
        for r, c in to_update:
            if len(self.table[r][c]) == 1:
                to_update.add((r, c))
                did_update = True
                val = next(iter(self.table[r][c]))
                self.constrain(r, c, val)
        return did_update
```

Trivial State Changes

- Optional addition for algorithm optimization
- Faster implementation of most constrained.
- If cell has only one possible value, commit that change. Repeat until no more trivial changes exist.

```python
def constrain_trivial_cells(self) → bool:
    did_update = False
    to_update = deepcopy(self.is_not_set)
    for r, c in to_update:
        if len(self.table[r][c]) == 1:
            to_update.add((r, c))
            did_update = True
            val = next(iter(self.table[r][c]))
            self.constrain(r, c, val)
    return did_update

def is_finished(self):
    return all(
        len(self.table[r][c]) ≤ 1
        for r, c in self.is_not_set
    )
```

```python
while state.constrain_trivial_cells():
    if state.is_finished():
        return state
```

NxN solutions

```
(aigroupproject-py3.12) → aigroupproject git:(master) ✗ python nxn_sudoku.py 5
Running Sudoku solver for a 25 grid
Verbose mode: False
+-----------------------+-----------------------+-----------------------+-----------------------+-----------------------+
| 4   16  10  5   13    | 2   9   20  19  23    | 17  14  3   12  7     | 25  11  18  22  24    | 21  6   1   15  8     |
| 22  11  17  3   2     | 1   16  12  21  7     | 8   23  19  10  25    | 6   9   15  14  13    | 18  24  5   20  4     |
| 25  7   14  9   6     | 5   17  8   18  15    | 4   24  13  21  22    | 1   12  10  3   20    | 23  16  11  19  2     |
| 21  24  20  23  15    | 25  3   6   13  10    | 11  18  1   5   2     | 4   16  7   19  8     | 22  17  12  9   14    |
| 8   1   18  19  12    | 22  4   14  24  11    | 9   6   15  16  20    | 21  2   5   23  17    | 13  3   10  25  7     |
+-----------------------+-----------------------+-----------------------+-----------------------+-----------------------+
| 5   3   24  16  10    | 19  7   9   17  20    | 6   2   11  23  18    | 22  14  8   1   25    | 4   12  15  21  13    |
| 12  2   25  22  4     | 10  13  18  23  6     | 15  20  17  8   19    | 11  21  9   16  7     | 1   14  24  5   3     |
| 14  20  6   15  17    | 24  11  1   25  21    | 13  7   10  22  12    | 18  5   3   4   19    | 9   2   23  8   16    |
| 1   23  9   11  18    | 16  12  15  8   22    | 14  4   21  3   5     | 24  13  17  2   10    | 20  7   19  6   25    |
| 13  8   19  7   21    | 14  5   3   4   2     | 16  25  24  9   1     | 12  23  6   20  15    | 11  22  17  10  18    |
+-----------------------+-----------------------+-----------------------+-----------------------+-----------------------+
| 17  13  22  12  3     | 7   20  24  16  19    | 5   9   18  14  6     | 15  10  2   21  1     | 25  4   8   11  23    |
| 9   25  15  18  16    | 8   2   21  1   5     | 22  17  23  13  11    | 7   20  4   12  14    | 3   19  6   24  10    |
| 19  5   23  20  1     | 15  14  4   9   13    | 25  16  12  24  10    | 17  6   11  8   3     | 2   18  22  7   21    |
| 7   21  8   10  11    | 3   25  23  6   18    | 20  1   4   2   15    | 9   22  19  24  16    | 12  13  14  17  5     |
| 24  4   2   6   14    | 11  22  17  10  12    | 3   8   7   19  21    | 5   18  25  13  23    | 16  15  20  1   9     |
+-----------------------+-----------------------+-----------------------+-----------------------+-----------------------+
| 15  9   4   24  19    | 13  6   2   22  17    | 10  5   25  20  14    | 8   3   23  18  12    | 7   1   21  16  11    |
| 16  14  13  8   22    | 23  15  11  5   25    | 7   3   2   1   9     | 19  17  24  6   21    | 10  20  4   18  12    |
| 23  10  7   1   25    | 21  19  16  14  4     | 12  22  8   18  17    | 20  15  13  11  5     | 6   9   3   2   24    |
| 3   17  11  2   5     | 20  18  10  12  24    | 21  13  6   15  4     | 16  7   1   9   22    | 14  8   25  23  19    |
| 6   18  12  21  20    | 9   8   7   3   1     | 23  19  16  11  24    | 2   25  14  10  4     | 15  5   13  22  17    |
+-----------------------+-----------------------+-----------------------+-----------------------+-----------------------+
| 2   15  5   13  7     | 18  1   19  11  3     | 24  10  14  25  16    | 23  4   22  17  6     | 8   21  9   12  20    |
| 20  22  1   17  9     | 4   21  13  2   8     | 18  15  5   7   23    | 3   19  12  25  11    | 24  10  16  14  6     |
| 11  19  21  14  24    | 6   10  25  7   16    | 2   12  22  4   8     | 13  1   20  5   9     | 17  23  18  3   15    |
| 10  6   16  4   8     | 12  23  5   15  9     | 1   11  20  17  3     | 14  24  21  7   18    | 19  25  2   13  22    |
| 18  12  3   25  23    | 17  24  22  20  14    | 19  21  9   6   13    | 10  8   16  15  2     | 5   11  7   4   1     |
+-----------------------+-----------------------+-----------------------+-----------------------+-----------------------+

Heuristic solving time: 1.340499 seconds.
Solution is solved and legal.
```