Solving KenKen as Constraint Satisfaction Problem

Samuel Breider

May 2025

1 Abstract

A player can intuitively identify a set of baseline patterns when solving KenKen puzzles. This report aims to identify these patterns, namely Single Square, Naked N Tuple, X-Wing, Evil Twin, and Hidden Single, and codify them into custom heuristics. These heuristics are then compared to determine which has the greatest effect on search space size and solve time. When solving this puzzle, KenKen boards are framed as a standard constraint satisfaction problem (CSP). Each tile on the board has its own domain of number values, and each cage has its own domain of tile permutations. The overall search space is quantified by the number of possible cage permutations on the board, rather than the number of possible values.

The approach was to solve five KenKen puzzles of various dimensions. For each, assorted orderings and combinations of pruning heuristics were tested to measure their effectiveness in reducing the search space. Through this, it was found that Naked N Tuple and Single Square had a significantly large effect on search space size, whereas instances of the other patterns were far more sparse. All sparsely used or unused heuristics shared a key characteristic: they depended on partial pruning already performed by the player during puzzle solving. This suggests that the initial pruning done by a program is extensive enough to render these patterns less relevant in practice.

2 Problem Description

KenKen puzzles are a single-player game with a fixed solution; only one set of numbers can satisfy the constraints of the board. The organization of a puzzle's solutions is based on the Latin square, each number appearing once in each row and column. There is the additional constraint of the arithmetic cages. Each number in a cage must reach the product of the number listed in the corner, using only the accompanying operation.

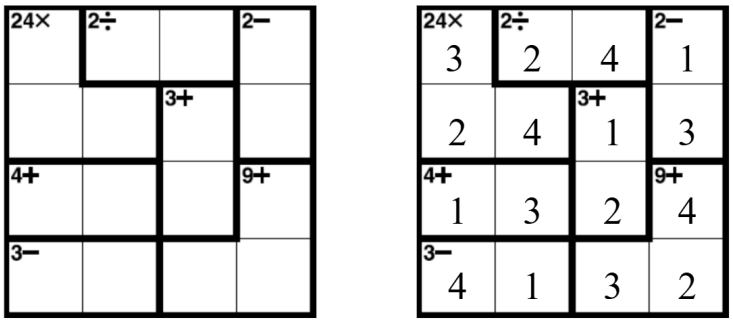

Figure 1: Example KenKen puzzle (left) and its solution (right). Each cage is outlined in bold and has an arithmetic goal in the upper-left corner.

Due to the inherent constraints of this puzzle, simpler boards can be solved with a basic brute-force style algorithm. However, such a strategy is unrealistic for a real player. As a human player gets more familiar with KenKen puzzles, a limited set of patterns emerges that can be exploited to solve boards much more easily. These patterns are detailed further in a later section of this paper. Through identifying these patterns, the goal of this research was to formulate a solving agent that approaches a KenKen puzzle similar to how a human player would. Each identified pattern was assigned a custom heuristic, and these pruning heuristics were executed in the order that a player could typically pick up on such patterns. In doing this, each heuristic can be compared for effectiveness in an arrangement of orders. Heuristics are also examined as standalone pruning methods; the result of these tests suggests an optimal order for how these patterns should be identified and exploited.

Each KenKen puzzle can be formalized as a Constraint Satisfaction Problem (CSP), where the variables correspond to individual cells, the domain of a tile is the set of integers from 1 to n (for an n x n puzzle), and the constraints include standard Latin square rules (row and column uniqueness) as well as arithmetic constraints imposed by the cages. A Minimum Remaining Value (MRV) heuristic is used on the count of possible permutations remaining in a cage to decide which values to explore first.

3 Problem Background

Many of today's puzzles and strategy games can be represented as constraint satisfaction problems (CSPs) and solved using standard algorithms and heuristics. These algorithms often follow similar structures when applied across different CSPs.

In developing a time-efficient and memory-conscious solving agent for KenKen puzzles, existing approaches to solving Ken-Ken, crossword puzzles, Sudoku, and the 8-puzzle, were examined. The primary focus of this review was to identify and compare heuristics, strategies and patterns that may prove to

be useful in constructing such a Ken-Ken solving agent.

3.1 Setup and Filtering Heuristics

One key aspect of Ken-Ken is that certain cage layouts can produce unique combinations of values that are not immediately apparent without analyzing the relationships between two or more cages. These specific patterns, commonly recognized within the Ken-Ken community, are single square, naked pair, naked triple, evil twin, hidden single, and X-wing. [4] These patterns raise the need for custom heuristics not common to other puzzle games.

The single square pattern describes a case where a tile can only take on one possible value. The corresponding heuristic removes any cage permutation that would place this same value elsewhere in the same row or column, thereby reducing the search space.

The naked pair occurs when two tiles in the same row or column share an identical set of exactly two possible values. In this case, any other cage permutation that would assign either of these two values to a different tile in the same row or column is eliminated from consideration. This concept generalizes to the naked n-tuple. If n tiles in a row or column share an identical set of n possible values, all permutations assigning any of these values to other tiles in the unit can be removed from the search space.

The evil twin pattern arises when only one specific value can satisfy the arithmetic constraint of a cage, given the known values in the other tiles. The heuristic in this case involves eliminating any permutation from the set of possible cage assignments that does not place the required evil twin value in the appropriate tile.

The hidden single refers to a situation where a value can only appear in one tile within a row or column, even if multiple candidates exist in that tile. The heuristic here removes any cage permutation that does not place the necessary value in that unique tile.

The X-wing is a more complex pattern. It occurs when a number is restricted to exactly two cells in each of two different rows, and those cells form the corners of a rectangle in two columns. In this scenario, the heuristic removes all permutations assigning that number to any other cells in the affected columns, as its position is implicitly constrained by the X-wing pattern.

3.2 Search Algorithm Implementation

Two popular and effective heuristics for puzzle solving and search space exploration are Minimum Domain Size (MiD) and Maximum Domain Size (MaD). These are commonly used in alphanumeric puzzles, sudoku, etc. and can be effectively applied to Ken-Ken.

Additional heuristic strategies were considered during the design of this study, including a modified Hamming distance-inspired heuristic. This heuristic has seen successful implementations in 8-Puzzle and Chess solvers. A related

application involving distance-based metrics has been evaluated for its effectiveness in puzzle solving. [3] However, this approach was ultimately abandoned in favor of heuristics that prioritize minimizing the size of the search space, something far more relevant in the context of Ken-Ken solving.

The benefits of these heuristics for large-scale puzzle solving are demonstrated effectively in past studies. [1] Heuristics such as Maximum of Domain (MaD) and Minimum of Domain (MiD) may be particularly advantageous in the context of Ken-Ken solving due to their compatibility with initialization strategies that generate all possible cage permutations prior to search. With the solving strategy adopted in this study, pre-forming all valid cage permutations, domain-focused heuristics like MaD and MiD offer a more intuitive fit than decision-ordering heuristics such as Minimum Remaining Value (MRV) or Least Constraining Value (LCV), which are more reliant on costly partial exploration of the search space.

MiD and MaD heuristics can be effectively leveraged to construct a backtracking tree search algorithm for solving Ken-Ken puzzles. An example of such an implementation utilizes a depth-first search (DFS) of a tree representing the Ken-Ken search space. [2] While their solver operates without the assistance of heuristics, Fahda notes in the concluding section of their work that performance could be significantly enhanced through the integration of heuristic strategies. This study aims to explore these potential improvements by incorporating MiD and MaD heuristics into a similar tree-based search approach.

3.3 Challenges In Implementation

A notable limitation of this study is restricted access to computational resources. All search processes will be executed without the aid of GPUs or other high-performance computing units. Recent advancements in CSP and puzzle-solving increasingly rely on the computational power of GPUs and software platforms like CUDA. A prominent example of this is employing neural networks to perform CSP solving. [9] Their models are developed using PyTorch, trained on 12.5 million instances, and executed on GPU-accelerated systems. Similarly, other studies approaches CSP problems by constructing a graph neural network. [7] It is of note that both of these papers were published within the last 3 years. This points to a trend of predictive modeling centric CSP solvers. While such methods show significant promise, especially given their ability to quantify patterns that skilled human solvers often identify intuitively, the computational demands make them impractical for the current study. Consequently, this research will focus on more lightweight, domain-pruning approaches.

3.4 Distantly Related Solving Agents

While numeric puzzles like Sudoku share the most direct similarities with Ken-Ken, valuable problem-solving strategies can also be drawn from research on a broader range of CSP-based puzzle solvers. It is useful to investigate CSP techniques applied to the domain of crossword puzzles. [8] A key observation

from this work is that crossword solving algorithms are heavily constrained by the difficulty of integrating natural language understanding. To mitigate this, Steinthal proposes several heuristics, such as prioritizing search order based on whether a candidate word is likely to be a noun or verb, and leveraging clues provided within the puzzle itself to guide search decisions. These findings suggest a potential direction for future Ken-Ken solving approaches: designing heuristics that take advantage of structural or logical patterns within a specific puzzle layout, beyond simple domain-based ordering. Such strategies could allow for more human-like exploration patterns that are not easily captured by traditional CSP heuristics.

Additional insight for this study can be drawn from research in two-player games, particularly in the application of heuristic evaluation functions, where many strategies have been developed for improving game-play. [6] While game-tree search techniques such as Minimax and Alpha-Beta pruning are mostly inapplicable to Ken-Ken solving agents, the study considers an applicable static evaluation heuristic: assigning a fixed weight to each board position based solely on location, independent of the opponent's response. In the context of Othello, this encourages players to aim for moves that control strategically valuable areas of the board. This idea can be transferred to Ken-Ken by assigning static priorities to certain cage types based on structural characteristics rather than their current domain constraints. For example, cages with higher tile counts or more permissive operations (e.g., addition and multiplication) could be either prioritized or deferred during the solving process. This positional evaluation strategy introduces an additional layer of search guidance, potentially complementing more traditional domain-pruning heuristics.

4 Approach

Each KenKen board in this experiment (5 in total) was represented using an object, made up of cages, which referenced all possible tile permutations that could make up that cage. The implementation of this in Python is shown in Figure 2. The initialization of the board, setup of the permutations, and self-explanatory helper functions are omitted from the following figures, but can be found in the repository for this paper. Each permutation was formed by finding all possible combinations of numbers 1 through n that could reach the goal total using the cage operation. The total of all these permutations serves as the baseline for assessing heuristic pruning performance.

```
class Board:
    def __init__(self, cages, k):
        self.cages = cages
        self.k = k

class Cage:
    def __init__(self, territory, operation, goal):
        self.territory = territory
        self.operation = operation
        self.goal = goal
        self.permutations = []

    # MRV
    def __lt__(self, other):
        return len(self.permutations) < len(other.permutations)

class Permutation:
    def __init__(self, tiles):
        self.tiles = tiles

class Tile:
    def __init__(self, y, x, value):
        self.y = y
        self.x = x
        self.value = value
```

Figure 2: Class definitions for KenKen board representation.

The less than operator override provides implicit MRV functionality when sorting cages. This guides which cages to explore next in solving.

```
def sort_cages(cages):
    q = []
    for cage in cages:
        heapq.heappush(q, cage)
    for i in range(len(cages)):
        cages[i] = heapq.heappop(q)
```

Figure 3: Heap sort algorithm using Minimum Remaining Value (MRV).

Each common pattern was given an accompanying heuristic to handle instances. In many cases, there is overlap between 2 patterns, such as X-Wing and Naked 2 Tuple or Single Square and Hidden Single. Despite this, it is beneficial to isolate each situation with a single handler for comparative analysis

```
single_square(possible):
    k = len(possible)
    for row in range(k):
        for col in range(k):
            if len(possible[row][col]) == 1:
                val = list(possible[row][col])[0]
                for c in range(k):
                    if c != col:
                        possible[row][c].discard(val)
                for r in range(k):
                    if r != row:
                        possible[r][col].discard(val)
```

Figure 4: Algorithm removing tiles from the possible values set by identifying Single Square patterns.

The single square pattern refers to the case where a tile can only take on a single value. The primary pruning that occurs here is when that number is filtered out of the entire row and column that tile resides in. This is typically the first action carried out by a human KenKen player.

```
def tuple_on_unit(sets, tiles, n):
    counts = Counter([s for s in sets if len(s) == n])
    for valset, count in counts.items():
        if count == n:
            for i, s in enumerate(sets):
                if s != valset:
                    tiles[i].difference_update(valset)

def naked_n_tuple(possible, n):
    k = len(possible)
    for i in range(k):
        row_sets = [frozenset(possible[i][j]) for j in range(k)]
        col_sets = [frozenset(possible[j][i]) for j in range(k)]
        tuple_on_unit(row_sets, possible[i], n)
        tuple_on_unit(col_sets, [possible[j][i] for j in range(k)], n)
```

Figure 5: Algorithm removing tiles from the possible values set by identifying Naked N Tuple patterns.

The Naked N Tuple, which almost always takes the form of a Naked 2 Tuple or Naked 3 Tuple, identifies spaces in which N spots in the same row or column can only be N of the same values. In this case, all other tiles in that row or column can not take on those values and they are removed from possibility.

```
def x_wing(possible):
    k = len(possible)
    for val in range(1, k+1):
        positions = []
        for row in range(k):
            cols_with_val = [col for col in range(k) if val in possible[row][col]]
            if len(cols_with_val) == 2:
                positions.append((row, tuple(cols_with_val)))
        for i in range(len(positions)):
            for j in range(i+1, len(positions)):
                r1, (c1a, c1b) = positions[i]
                r2, (c2a, c2b) = positions[j]
                if {c1a, c1b} == {c2a, c2b}:
                    for r in range(k):
                        if r != r1 and r != r2:
                            possible[r][c1a].discard(val)
                            possible[r][c1b].discard(val)
```

Figure 6: Algorithm removing tiles from the possible values set by identifying X-Wing patterns.

X-Wing is a far less common pattern, in which 4 tiles that form a rectangle with one another share a single value in their possible set, which must be of size 2. In this case, the shared value must occupy a space in each row and column on that rectangle, so it can be removed from the possible set of all other tiles in those rows and columns. This is also a common Sudoku pattern.

```
def hidden_single(possible):
    k = len(possible)
    changed = True

    while changed:
        changed = False
        for row in range(k):
            count = [0 for _ in range(k + 1)]
            position = [None for _ in range(k + 1)]
            for col in range(k):
                for val in possible[row][col]:
                    count[val] += 1
                    position[val] = (row, col)
            for val in range(1, k + 1):
                if count[val] == 1:
                    r, c = position[val]
                    if possible[r][c] != {val}:
                        possible[r][c] = {val}
                        changed = True
                        for r2 in range(k):
                            if r2 != r and val in possible[r2][c]:
                                possible[r2][c].discard(val)
                        for c2 in range(k):
                            if c2 != c and val in possible[r][c2]:
                                possible[r][c2].discard(val)

        for col in range(k):
            count = [0 for _ in range(k + 1)]
            position = [None for _ in range(k + 1)]
            for row in range(k):
                for val in possible[row][col]:
                    count[val] += 1
                    position[val] = (row, col)
            for val in range(1, k + 1):
                if count[val] == 1:
                    r, c = position[val]
                    if possible[r][c] != {val}:
                        possible[r][c] = {val}
                        changed = True
                        for r2 in range(k):
                            if r2 != r and val in possible[r2][c]:
                                possible[r2][c].discard(val)
                        for c2 in range(k):
                            if c2 != c and val in possible[r][c2]:
                                possible[r][c2].discard(val)
```

Figure 7: Algorithm removing tiles from the possible values set by identifying Hidden Single patterns.

The Hidden Single is a simple pattern, and one of the most used by human KenKen players, but sees less usage in this solver due to its time and space complexity. With the memory and computing power of an automated solver, Naked N Tuples and Single Square often clear out any instances of this pattern before they can be identified. Nonetheless, this heuristic handles instances of this pattern.

```
def evil_twin(cages):
    for cage in cages:
        for perm in cage.permutations[:]:
            required_values = [tile.value for tile in perm.tiles]
            for i, _ in enumerate(perm.tiles):
                others = required_values[:i] + required_values[i+1:]
                if cage.operation == "+" and sum(others) > cage.goal:
                    cage.permutations.remove(perm)
                    break
                if cage.operation == "*" and product(others) > cage.goal:
                    cage.permutations.remove(perm)
                    break
```

Figure 8: Algorithm removing tiles from the possible values set by identifying Evil Twin patterns.

Evil Twin is a pattern in which, given that another tile is solved, only one tile value can meet the requirements of the arithmetic equation. This handler could be omitted in our final solver because only possible permutations are generated, which implicitly handles this rule. However, it is included to provide further emulation of human solving patterns, as humans do not have the capabilities to immediately only consider possible permutations, especially at problems of scale.

```
def filter_cages(cages, k, heuristic_to_use):
    possible = [[set() for _ in range(k)] for _ in range(k)]

    if "naked_n_tuple" in heuristic_to_use:
        for count in range(2,4):
            naked_n_tuple(possible, count)
    if "x_wing" in heuristic_to_use:
        x_wing(possible)
    if "single_square" in heuristic_to_use:
        single_square(possible)
    if "evil_twin" in heuristic_to_use:
        evil_twin(cages)
    if "hidden_single" in heuristic_to_use:
        hidden_single(possible)

    for cage in cages:
        good_permutations = []
        for permutation in cage.permutations:
            if all(tile.value in possible[tile.y][tile.x] for tile in permutation.tiles):
                good_permutations.append(permutation)
        cage.permutations = good_permutations

def solve(puzzle, heuristic_to_use):
    cages = puzzle.cages
    k = puzzle.k
    filter_cages(cages, k, heuristic_to_use)
    sort_cages(cages)
    def backtrack(tiles, board, i):
        if i == len(cages):
            for tile in tiles:
                board[tile.y][tile.x] = tile.value
            if validate_board(board):
                visualize(board)
                return True
        else:
            for perm in cages[i].permutations:
                new_tiles = tiles + perm.tiles
                if validate_search(new_tiles, k):
                    if backtrack(new_tiles, copy.deepcopy(board), i+1):
                        return True
        return False
    board = [[-1 for _ in range(k)] for _ in range(k)]
    backtrack([], board, 0)
```

Figure 9: Board solving and filtering algorithm.

The board-solving and filtering algorithms serve as the performance metric for this experiment. The filtering algorithm reduces the number of searchable permutations by using the heuristic functions for pruning. The reduction in permutation count can be calculated with a small modification to the algorithms in Figure 9. The clock time of the solving algorithm in Figure 9 serves as the time metric, a reduction in the running time of this algorithm means that a correct solution to the board was found more quickly. The search and board validation algorithms are omitted from this figure, but can be found in the repository for this paper.

5 Experiment Design and Results

This experiment examines 5 different KenKen boards of size 4 x 4, 5 x 5, 6 x 6, 7 x 7, and 9 x 9. A 9 x 9 board was selected as the largest size due to the hardware restrictions imposed by the lack of a GPU or other high-powered processing unit. Each heuristic will be run individually on each board, and then all will be run in unison. Pruning heuristics are recorded using 2 metrics: time complexity in reaching a board solution and the number of cage permutations removed before searching the solution space. When collecting these statistics, each board is searched 3 times and the average output for each metric is recorded as the final output.

Heuristic	**Permutations Pruned**	**Runtime (s)**
Single Square	6 / 31 (19.4%)	5.18×10^{-4}
Naked N Tuple	6 / 31 (19.4%)	5.53×10^{-4}
X-Wing	0 / 31 (0.0%)	5.19×10^{-4}
Hidden Single	10 / 31 (32.3%)	5.24×10^{-4}
Evil Twin	0 / 31 (0.0%)	5.19×10^{-4}
All Combined	**17 / 31 (54.8%)**	5.21×10^{-4}

Table 1: Heuristic pruning effectiveness and runtime (Puzzle 1, 4x4 board).

Heuristic	**Permutations Pruned**	**Runtime (s)**
Single Square	0 / 82 (0.0%)	2.00×10^{-3}
Naked N Tuple	7 / 82 (8.5%)	2.00×10^{-3}
Hidden Single	39 / 82 (47.6%)	1.00×10^{-3}
X-Wing	0 / 82 (0.0%)	2.01×10^{-3}
Evil Twin	0 / 82 (0.0%)	9.88×10^{-3}
All Combined	**43 / 82 (52.4%)**	2.00×10^{-3}

Table 2: Heuristic pruning effectiveness and runtime (Puzzle 2, 5x5 board).

Heuristic	Permutations Pruned	Runtime (s)
Single Square	101 / 298 (33.9%)	4.51×10^{-2}
Naked N Tuple	24 / 298 (8.1%)	4.40×10^{-2}
Hidden Single	0 / 298 (0.0%)	8.42×10^{-2}
X-Wing	0 / 298 (0.0%)	1.23×10^{-1}
Evil Twin	0 / 298 (0.0%)	8.18×10^{-2}
All Combined	**117 / 298 (39.3%)**	9.51×10^{-2}

Table 3: Heuristic pruning effectiveness and runtime (Puzzle 3, 6x6 board).

Heuristic	Permutations Pruned	Runtime (s)
Single Square	19 / 211 (9.0%)	1.30×10^{-2}
Naked N Tuple	53 / 211 (25.1%)	1.20×10^{-2}
Hidden Single	50 / 211 (23.7%)	1.86×10^{-1}
X-Wing	0 / 211 (0.0%)	4.92×10^{-2}
Evil Twin	0 / 211 (0.0%)	3.62×10^{-2}
All Combined	**84 / 211 (39.8%)**	2.13×10^{-2}

Table 4: Heuristic pruning effectiveness and runtime (Puzzle 4, 7x7 board).

Heuristic	Permutations Pruned	Runtime (s)
Single Square	83 / 646 (12.8%)	16.99
Naked N Tuple	110 / 646 (17.0%)	2.28
Hidden Single	0 / 646 (0.0%)	21.13
X-Wing	0 / 646 (0.0%)	21.22
Evil Twin	0 / 646 (0.0%)	21.36
All Combined	**178 / 646 (27.6%)**	**3.47**

Table 5: Heuristic pruning effectiveness and runtime (Puzzle 5, 9x9 board).

6 Results Analysis

The experiment provides a picture of how different pruning heuristics affect both the number of permutations eliminated and the time required to reach a solution. Across all five puzzles, consistent patterns emerged, displaying the relative utility of each heuristic. These patterns also reveal the effect that problem scale has on these pruning heuristics.

6.1 Effectiveness of Individual Heuristics

The *Single Square* heuristic had strong performance on smaller boards (4x4 and 6x6), pruning a substantial portion of permutations with very fast runtime. On the 6x6 puzzle, it removed 33.9% of permutations with a runtime of just 0.0451

seconds. However, on larger boards like the 9x9, although it still removed a significant number (12.8%), its runtime increased dramatically to 16.99 seconds. It also did not affect the search space of the 5x5 board. This suggests the heuristic scales poorly with board size, likely due to increased overhead in examining each tile's domain individually. This logically tracks, as single square patterns have a proportionally larger effect when fewer numbers exist in a domain.

Naked N Tuple consistently pruned a moderate to high portion of the search space across all puzzles. It was particularly effective on the 7x7 and 9x9 puzzles, pruning 25.1% and 17.0% of permutations respectively, while maintaining reasonable runtimes (e.g., 2.28 seconds on the 9x9 board). This suggests that the pattern appears more frequently in larger grids, and the algorithm is computationally manageable.

Hidden Single showed mixed results. It was highly effective in the 5x5 and 7x7 puzzles, pruning 47.6% and 23.7% of permutations, respectively. However, it failed to prune any permutations on the 6x6 and 9x9 boards. Additionally, its runtime was unpredictable: while it was efficient in smaller boards, it was the slowest heuristic on the 7x7 board (0.186 seconds) and completely inefficient on the 9x9 board (21.13 seconds), where it provided no pruning benefit. This indicates that while powerful, the heuristic is not scalable and may be redundant if other filters are more aggressively applied earlier.

X-Wing and *Evil Twin* performed poorly across all puzzles. Neither heuristic consistently pruned any permutations in the majority of cases. Their runtimes, however, were non-negligible. X-Wing in particular was very slow on the 6x6, 7x7, and 9x9 puzzles (up to 21.22 seconds) despite yielding no benefits. The poor pruning efficiency combined with high runtimes makes both heuristics candidates for omission in a production solver unless their implementations can be substantially optimized.

6.2 Effectiveness of Combined Heuristics

When all heuristics were applied together, the solver achieved the best pruning results. Notably, the combined pruning consistently outperformed any individual heuristic across all puzzle sizes. For example, the 9x9 puzzle saw 27.6% of permutations eliminated with a runtime of 3.47 seconds, compared to just 17.0% and 2.28 seconds using Naked N Tuple alone. Similarly, in the 6x6 puzzle, combining heuristics pruned 39.3%, compared to 33.9% by Single Square alone. There were some outliers in terms of time complexity differences, but the consistent pruning ability suggests this is the best approach for larger problems.

This supports the intuition that heuristics are complementary. Some patterns may not appear until others have simplified the board, especially with domain-focused heuristics like Naked N Tuple or Single Square. However, the benefit of a combination must be weighed against runtime. For instance, the jump in runtime between applying just one heuristic and all of them was minor in smaller puzzles but became more significant in the largest ones. Still, even with all heuristics, the runtime on the 9x9 board remained under 4 seconds, which is acceptable for most use cases.

Within the data presented in this study, some relationships between pruning effectiveness and problem scale can be identified, but it would be relevant to explore performance over larger puzzles that are not possible with the hardware constraints in this study. It may also be useful to categorize puzzles before assessing heuristic effectiveness. Certain pruning actions are only useful under specific circumstances. Therefore, in this study, those heuristics with more general applications performed far better. With an increase in processing power and stronger categorization, it may be possible to draw stronger conclusions.

7 Summary and Conclusion

Overall, Naked N Tuple and Single Square provided the most consistent and scalable pruning benefits across puzzle sizes. Hidden Single, while effective on select boards, proved too computationally expensive and unreliable to always include. X-Wing and Evil Twin provided negligible benefits and increased runtime, suggesting they should only be selectively used when specific board patterns are detected.

Combining heuristics offers the most substantial pruning but introduces increased complexity. Developers implementing a heuristic-driven KenKen solver may benefit from an adaptive approach: starting with efficient, broad-scope heuristics like Single Square and Naked N Tuple, then conditionally applying costlier ones if needed. This balances pruning efficiency and runtime scalability.

References

[1] Broderick Crawford, Carlos Castro, and Eric Monfroy. *Cutting-Edge Research Topics on Multiple Criteria Decision Making*, volume 35, chapter Solving Sudoku with Constraint Programming, pages 345–348. 01 2009.

[2] Raden Yoga Fahda. Kenken puzzle solver using backtracking algorithm. PDF, May 2015. Accessed April 13, 2025.

[3] Anca Iordan. A comparative study of three heuristic functions used to solve the 8-puzzle. *British Journal of Mathematics Computer Science*, 16:1–18, 01 2016.

[4] Olivia Johanna, Kie Van, and Kie Van Ivanky Saputra. Solving and modeling ken-ken puzzle by using hybrid genetics algorithm. In *Proceedings of the International Conference on Engineering and Technology Development (ICETD)*, 06 2012.

[5] Malek Mouhoub and Bahareh Jashmi. Heuristic techniques for variable and value ordering in csps. In *13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011, Proceedings, Dublin, Ireland, July 12-16, 2011*, pages 457–464, 07 2011.

[6] Vaishnavi Sannidhanam and Muthukaruppan Annamalai. An analysis of heuristics in othello. Technical report, Department of Computer Science and Engineering, University of Washington, 2004. Final Project Report for CSE 573: Artificial Intelligence.

[7] Wen Song, Zhiguang Cao, Jie Zhang, Chi Xu, and Andrew Lim. Learning variable ordering heuristics for solving constraint satisfaction problems. *Engineering Applications of Artificial Intelligence*, 109:104603, 2022.

[8] Russell Steinthal. Crossword puzzles as constraint satisfaction problems. Technical report, Columbia University, 1999. Final Project Report for COMS W4721: Advanced Intelligent Systems (Prof. Siegel).

[9] Jan Tönshoff, Berke Kisin, Jakob Lindner, and Martin Grohe. One model, any csp: Graph neural networks as fast global search heuristics for constraint satisfaction, 08 2022.

Notes

I, Samuel Breider, was the sole contributor to this report. Full code can be found at the following repository: https://github.com/BREISAMU/kenken-solver.