A Comparative Study of Backtracking and Bruteforce Algorithms for Kakuro Puzzle Solving

Tabitha Permalla - 13521111
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
13521111@std.stei.itb.ac.id

Abstract—**Kakuro puzzles are crossword-like puzzles. But, instead of letters, the board is filled with numbers ranging from 1 to 9. Solving these puzzles efficiently has been a challenge for puzzle enthusiasts as well as computer-based puzzle solvers. This paper presents a comparative study of two commonly used algorithms, namely Backtracking and Brute Force, for solving Kakuro puzzles. The Backtracking algorithm is a systematic search algorithm that explores the solution space by making decisions and backtracking when necessary. On the other hand, the Brute Force algorithm exhaustively searches through all possible combinations of cell assignments until a valid solution is found. The objective of this study is to evaluate and compare the performance of these two algorithms in terms of solution quality and computational efficiency. A set of Kakuro puzzles with varying complexities is used as the benchmark for the evaluation.**

Keywords—Kakuro puzzles; Backtracking algorithm; Bruteforce algorithm; Computational efficiency

I. INTRODUCTION

Kakuro puzzles are crossword-like puzzles where the objective is to fill the cells of the puzzle with numbers according to specific rules. Similar to Sudoku, the cells in Kakuro puzzles can be filled with numbers ranging from 1 to 9. The rules of the puzzle require that numbers in the same row or column section must be distinct, and the sum of numbers in each section must match the given clues.

Solving Kakuro puzzles efficiently poses a challenge for both puzzle enthusiasts and computer-based solvers. While the focus of this study is on the comparative analysis of the Backtracking and Brute Force algorithms, it is important to acknowledge that there are other methods available for solving Kakuro puzzles.

One alternative approach is the Divide and Conquer strategy. In this method, the Kakuro puzzle is divided into smaller sub-puzzles, which are then solved independently. The solutions to the sub-puzzles are combined to obtain the solution for the entire puzzle. This approach can exploit the structure of the puzzle and reduce the search space by decomposing the problem into more manageable parts.

Additionally, techniques from Artificial Intelligence, such as heuristic search algorithms, can be applied to solve Kakuro puzzles. Heuristic search algorithms, like A* (A-star), use heuristic functions to guide the search towards more promising paths and avoid exploring less likely solutions. These algorithms can help improve the efficiency and speed of finding solutions by intelligently selecting which cell assignments to explore.

Moreover, evolutionary algorithms, such as genetic algorithms or particle swarm optimization, can also be employed for solving Kakuro puzzles. These algorithms mimic biological evolution or swarm behavior to iteratively generate and evaluate potential solutions. By using principles of selection, crossover, and mutation, they explore the solution space and gradually improve the quality of solutions over successive generations.

It is worth mentioning that different methods may have varying performance characteristics depending on the specific puzzle instance and its complexity. Therefore, evaluating and comparing the effectiveness and efficiency of various methods can provide valuable insights for solving Kakuro puzzles.

In this study, the focus is primarily on the comparative analysis of the Backtracking and Brute Force algorithms. Nevertheless, acknowledging the existence of alternative methods highlights the diverse range of approaches that can be used to solve Kakuro puzzles. Each method offers unique advantages and trade-offs in terms of computational efficiency, solution quality, and complexity.

II. THEORETICAL BASIS

A. *Brute Force Algorithm*

The Brute Force Algorithm is a straightforward and exhaustive problem-solving technique that explores all possible solutions or approaches for a given problem. It does not rely on complex optimizations or advanced methods to improve efficiency. Instead, Brute Force Algorithms rely on sheer computational power and consider every potential solution to find the desired outcome.

Using the Brute Force algorithm offers several advantages. Firstly, it provides a guaranteed way to find the correct solution by systematically listing all possible candidate solutions for the problem. This comprehensive exploration ensures that no potential solution is overlooked. Additionally, the Brute Force approach is not limited to any specific problem domain. It can

be applied to a wide range of problems in computing, making it a versatile method. Moreover, the simplicity of the Brute Force algorithm makes it particularly suitable for solving small and simpler problems. Its straightforward nature simplifies implementation and facilitates understanding. Lastly, the Brute Force algorithm is often used as a benchmark for comparison purposes, allowing for the evaluation of other design techniques against its straightforward approach.

However, there are several drawbacks to consider when using the Brute Force approach. Firstly, while it may be effective, it is not an efficient method. For real-time problems, the time complexity of the Brute Force algorithm can grow exponentially, often reaching orders such as O(n!). As a result, programs implementing the Brute Force algorithm can become slow and impractical for large-scale or time-sensitive problems. Furthermore, the Brute Force method relies more on the computational power of the system rather than on the design of an optimized algorithm to solve a problem. This can lead to suboptimal solutions and a heavy reliance on computational resources. Lastly, Brute Force algorithms are not inherently constructive or creative compared to other problem-solving techniques. They lack the ability to exploit problem-specific insights or employ specialized optimizations.

In conclusion, the Brute Force algorithm is a straightforward and exhaustive method that guarantees solutions by considering all possible options. It is versatile, simple to implement, and serves as a benchmark for comparison. However, it suffers from efficiency issues, a reliance on computational power, and a lack of creativity compared to other techniques. Careful consideration should be given to the specific problem and its requirements when deciding whether to use the Brute Force approach. As of the use of the Brute Force algorithm in this study is specifically intended to provide a comparison with the Backtracking algorithm.

B. *Backtracking Algorithm*

Backtracking is a highly effective method for problem-solving that builds upon the concept of exhaustive search, in other words the brute force algorithm. While exhaustive search explores and evaluates all possible solutions, backtracking takes a more focused approach. It selectively explores choices that lead to potential solutions while disregarding options that do not contribute to the desired outcome. This is achieved through the process of pruning, which eliminates nodes that do not lead to a solution. The concept of backtracking was first introduced by D.H. Lehmer in 1950, and subsequent contributions from R.J. Walker, Golomb, and Baumert provided a comprehensive understanding of this algorithmic approach.

There are several basic terminologies used in the Backtracking approach.

- Solution vector: The intended solution X for a given problem instance P with an input size of n is defined as a vector consisting of candidate solutions chosen from a finite set of possible solutions S. Consequently, a solution can be represented as an n-tuple (X1, X2, ..., Xn), and a partial solution is denoted by (X1, X2, ..., Xi), where i<n.
- Solution space: The solution space S of a problem instance P consists of all candidate solutions xi that satisfy the explicit constraints. Within a state space tree, the solution space is described by all paths from the root node to a leaf node.
- Generator function: The function for generating the value xk is expressed as the predicate T(). T(x[1], x[2], ..., x[k - 1]) generates the value for x_k, which is a component of the solution vector.
- Constraints: Constraints are the guidelines that restrict the solution vector (X1, X2, ..., Xa). They define the allowable values for candidate solutions and specify the relationships between them.
- Bounding function: The function is alternatively referred to as a "validity function," "criterion function," or "promising function." For a given problem instance P, it involves optimizing the function B(x1, x2, ..., Xa), which is intended to be either maximized or minimized. By optimizing the search within the solution space S of problem instance P, it aids in finding a solution vector (X1, X2, ..., Xn). This function is helpful in eliminating candidate solutions that do not lead to the desired solution for the problem. It effectively prunes live nodes by not exploring their children if the constraints are not satisfied.

III. Problem Mapping

When solving a Kakuro puzzle using backtracking, several key elements come into play. These elements help us navigate the puzzle-solving process effectively and efficiently. By understanding and utilizing these elements, we can devise a systematic approach to explore the solution space, generate candidate solutions, enforce constraints, and update the solution vector. This strategic mapping of elements forms the foundation for successfully solving Kakuro puzzles using the backtracking algorithm. In the context of solving a Kakuro puzzle using backtracking, we can map the following elements:

A. *Solution Vector*

The solution vector represents the current state of the puzzle grid, including the numbers assigned to the cells. It is a representation of the partial solution being constructed during the backtracking process.

B. *Solution Space*

The solution space represents all possible combinations of numbers that can fill the empty cells in the Kakuro puzzle grid. Where each cell can be assigned a number from 1 to 9, subject to the constraints of the puzzle.

Identify applicable sponsor/s here. If no sponsors, delete this text box (*sponsors).*

C. Generator Function

The generator function in the context of backtracking generates the next candidate solution for a particular position in the puzzle grid. It determines the possible numbers that can be placed in an empty cell based on the current state of the grid and the existing numbers in the row and column.

D. Constraints

The constraints of the Kakuro puzzle are as follows:

- Each number of the same row section must be unique.
- Each number of the same column section must be unique.
- Each number of a certain row section must be less than the clue provided.
- Each number of a certain column section must be less than the clue provided.
- The sum of the numbers in a certain row or column should be less than the clue provided.

E. Bounding Function

The bounding function in backtracking helps in pruning the search space by identifying early on if a particular partial solution cannot lead to a valid solution. In the context of Kakuro puzzles, the bounding function can check if the sum of numbers in a row or column exceeds the given clue or if a number violates the uniqueness constraint.

IV. Implementation

To evaluate the effectiveness of Brute Force and Backtracking algorithms in solving Kakuro puzzles, a Python program was developed. The program was designed to handle relatively straightforward Kakuro puzzles, ensuring ease of implementation and analysis. By implementing both algorithms in the program, their performance and efficiency can be compared and evaluated. This approach allows for a comprehensive assessment of the strengths and limitations of each algorithm when applied to simpler Kakuro puzzles. The details of the program are as follows:

A. Kakuro Puzzle Representation

In this program, the Kakuro puzzle is represented in a class containing the following attributes:

- count : the number of steps taken to solve the puzzle
- row_clues, col_clues : list of row and column clues which define constraints of the puzzle. Each row clue is of the format [row index, start column index, end column index, sum]. While each column clue is of the format [column index, start row index, end row index, sum].
- rows, cols : the number of rows and columns in the puzzle; the dimension of the puzzle
- board : a 2-dimensional list representing the state of the puzzle
- clue_grids : a list of coordinates of the board that contains clues
- empty_grids : a list of coordinates of the board that are not to be filled, or in other words are not part of the puzzle
- question : a 2-dimensional list representing the starting state of the puzzle

The implementation of the Kakuro Puzzle in python are as follows:

```
class KakuroPuzzle:
    def __init__(self, row_clues, col_clues):
        self.count = 0
        self.row_clues = row_clues
        self.col_clues = col_clues
        self.rows = max(row[0] for row in row_clues) + 1
        self.cols = max(col[0] for col in col_clues) + 1
        self.board = [[0 for _ in range(self.cols)] for _ in 
range(self.rows)]
        self.clue_grids = []
        self.empty_grids = [(i, j) for i in range(self.rows) for j in 
range(self.cols)]
        for row in row_clues:
            self.board[row[0]][row[1]-1] = row[3]
            self.clue_grids.append((row[0],row[1]-1))
            for j in range(row[1]-1,row[2]+1):
                if self.empty_grids.count((row[0],j)) > 0:
                    self.empty_grids.remove((row[0],j))
        for col in col_clues:
            self.board[col[1]-1][col[0]] = col[3]
            self.clue_grids.append((col[1]-1,col[0]))
            for i in range(col[1]-1,col[2]+1):
                if self.empty_grids.count((i,col[0])) > 0:
                    self.empty_grids.remove((i,col[0]))
        self.question = [[self.board[i][j] for j in range(self.cols)] for i 
in range(self.rows)]
```

B. Brute Force Algorithm

The implementation of the Brute Force algorithm to solve the Kakuro Puzzle are as follows:

```
def solve_brute_force(self):
    self.count = 0
    return self._solve_brute_force_helper(0, 0)

def _solve_brute_force_helper(self, row, col):
    self.count += 1
    self.draw_solution("./case"+str(self.case)+"/brute 
force/"+str(self.count))
    if row > self.rows - 1:
        if self.is_solution() and self.is_unique():
            return True
        return False

    next_row = row + 1 if col == self.cols - 1 else row
    next_col = 1 if col == self.cols - 1 else col + 1
```

```
    if (row,col) not in self.empty_grids:
        if self.board[row][col] != 0:
            return self._solve_brute_force_helper(next_row, next_col)

        for num in range(9,0,-1):
            self.board[row][col] = num

            if self._solve_brute_force_helper(next_row, next_col):
                return True

        self.board[row][col] = 0

    if (row,col) in self.empty_grids and
self._solve_brute_force_helper(next_row, next_col):
        return True

    return False
```

The program above iterates through every possible combination until a solution is found. It fills the grids of the puzzle with numbers 1 through 9 (for several reasons, the program is written to fill the grids with larger numbers first, starting from 9 through 1).

C. Backtracking Algorithm

The implementation of the Backtracking algorithm to solve the Kakuro Puzzle are as follows:

```
def solve_backtrack(self):
    self.count = 0
    return self._solve_bactrack_helper(0, 0)

def _solve_bactrack_helper(self, row, col):
    self.count += 1

self.draw_solution("./case"+str(self.case)+"/backtrack/"+str(self.count))
    if row > self.rows - 1:
        return self.is_solution()

    next_row = row + 1 if col == self.cols - 1 else row
    next_col = 1 if col == self.cols - 1 else col + 1

    if self.board[row][col] != 0:
        return self._solve_bactrack_helper(next_row, next_col)

    if (row,col) not in self.empty_grids:
        for num in range(9,0,-1):
            if self.is_valid(row, col, num):
                self.board[row][col] = num

                if self._solve_bactrack_helper(next_row, next_col):
                    return True

                self.board[row][col] = 0

    if (row,col) in self.empty_grids and
self._solve_bactrack_helper(next_row, next_col):
        return True

    return False
```

It is very much similar to the Brute Force algorithm. However, on every step, constraint checks are taken. If any constraint is violated, the program backtracks. The constraint checks are done by calling the bounding function is_valid as follows:

```
def is_valid(self, row, col, num):
    # Check if the number is already present in the row
    i = 0
    while self.row_clues[i][0] != row:
        i += 1
    if num in self.board[row][self.row_clues[i][1]:self.row_clues[i][2]+1]:
        return False
    # Check if the number is greater than the row clue
    if num > self.row_clues[i][3]:
        return False

    # Check if the number is already present in the column
    j = 0
    while self.col_clues[j][0] != col:
        j += 1
    for i in range(self.col_clues[j][1], self.col_clues[j][2]+1):
        if num == self.board[i][col]:
            return False
    # Check if the number is greater than the column clue
    if num > self.col_clues[j][3]:
        return False


    # Check the row clues
    for clue in self.row_clues:
        if row == clue[0] and col >= clue[1] and col <= clue[2]:
            total = sum(self.board[row][clue[1] : clue[2] + 1])
            if total > clue[3]:
                return False
            if total == clue[3] and self.board[row][col] != 0:
                return False

    # Check the column clues
    for clue in self.col_clues:
        if col == clue[0] and row >= clue[1] and row <= clue[2]:
            total = sum(self.board[i][col] for i in range(clue[1], clue[2] +
1))
            if total > clue[3]:
                return False
            if total == clue[3] and self.board[row][col] != 0:
                return False

    return True
```

V. Testing and Analysis

In order to provide data for this study, 4 test cases are made. Meanwhile, the parameters that are going to be used for analysis are the solution, the number of steps taken until the solution is found, and the time taken until the solution is found.

A. Test Cases

The test cases used are as follows:

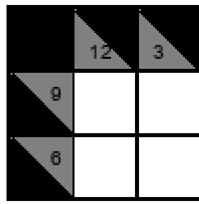

Fig 5.1.1 Test Case 1

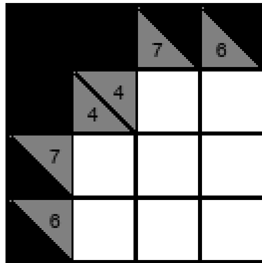

Fig 5.1.2 Test Case 2

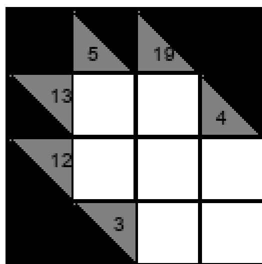

Fig 5.1.3 Test Case 3

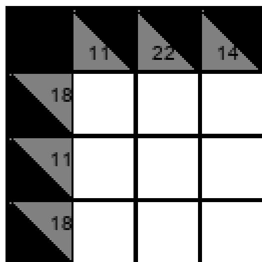

Fig 5.1.4 Test Case 4

B. Solutions

For all test cases, the solutions found through Backtracking and Brute Force are the same. Both algorithms result in the following solutions:

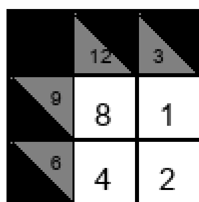

Fig 5.2.1 Result 1

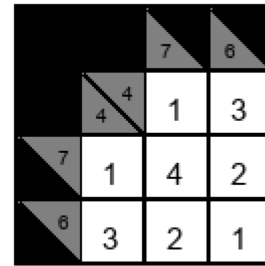

Fig 5.2.2 Result 2

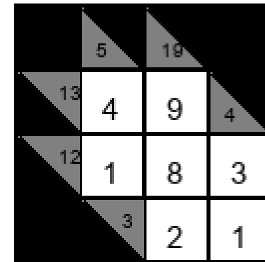

Fig 5.2.3 Result 3

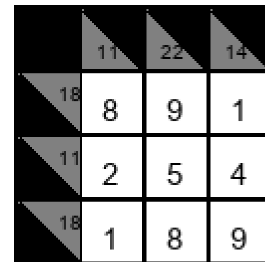

Fig 5.2.4 Result 4

C. Number of Steps

The number of steps taken for each test case until the solution is found using the Backtracking and Brute Force algorithm differs. The number of steps for each test case are as follows:

Table I: Number of steps taken until solution is found

Test Case	Backtracking	Brute Force
1	101 steps	1613 steps
2	8559 steps	47208669 steps
3	5476 steps	3083374 steps
4	1598824 steps	53709915 steps

D. Time

The amount of time taken for each test case until the solution is found using the Backtracking and Brute Force algorithm differs. The amount of time for each test case are as follows:

Table II: Amount of time taken until solution is found

Test Case	Backtracking	Brute Force
1	0.00265 seconds	0.00313 seconds
2	0.06795 seconds	48.72821 seconds
3	0.02860 seconds	4.09411 seconds
4	7.52062 seconds	62.45073 seconds

E. *Analysis and Conclusion*

From the provided data in Table I, it can be observed that the number of steps taken to find the solution using the Backtracking algorithm is generally lower compared to the Brute Force algorithm for all test cases. This indicates that the Backtracking algorithm is more efficient in terms of exploring the solution space and reaching the desired solution. Test Case 1 demonstrates a significant difference in the number of steps, with Backtracking requiring only 101 steps compared to Brute Force's 1613 steps. This trend continues in Test Cases 2, 3, and 4, further highlighting the advantage of Backtracking in terms of reducing the number of steps taken.

The time taken to find the solution using the Backtracking and Brute Force algorithms can be analysed based on the provided data in Table II. It can be observed that the Backtracking algorithm generally outperforms the Brute Force algorithm in terms of time efficiency. In Test Case 1, Backtracking took 0.00265 seconds, which is considerably shorter than the 0.00313 seconds taken by Brute Force. This trend continues in Test Cases 2, 3, and 4, with Backtracking consistently requiring less time compared to Brute Force. Notably, Test Case 2 demonstrates a substantial difference in time, with Backtracking taking 0.06795 seconds while Brute Force took 48.72821 seconds.

Overall, the analysis of the number of steps and time taken suggests that the Backtracking algorithm is more efficient and faster in finding the solution for the given Kakuro puzzles compared to the Brute Force algorithm. It demonstrates superior performance in terms of reducing the number of steps and completing the puzzles in a shorter amount of time. These findings highlight the benefits of employing the Backtracking algorithm for solving Kakuro puzzles, especially when dealing with relatively simple cases.

Repository Link at GitHub

https://github.com/Bitha17/simple-kakuro-solver

Video Link at YouTube

https://youtu.be/8sRUsxmhbuY

Acknowledgment

First and foremost, the author would like to express gratitude and thankfulness towards God. For only by His grace and kindness the author is able to complete this study. He has provided sufficient guidance, blessings, and strength throughout the completion of this project.

The author would also like to express gratitude and sincere appreciation towards Dr. Ir. Rinaldi, M.T. as the lecturer of IF2211 Strategi Algoritma. His invaluable knowledge, expertise, and dedication in teaching have greatly contributed to the author's understanding of algorithmic strategies and their applications. His continuous support and guidance have been instrumental in shaping the author's academic journey and enhancing the author's problem-solving skills.

References

[1] Munir, Rinaldi. (2023) "Algoritma Brute Force (Bagian 1)" accessed from https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Brute-Force-(2022)-Bag1.pdf on May 19, 2023

[2] Munir, Rinaldi. (2023) "Algoritma Runut-balik (Backtracking) (Bagian 1)" accessed from https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-backtracking-2021-Bagian1.pdf on May 21, 2023

[3] Geeks For Geeks. "Brute Force Approach and its pros and cons" accessed from https://www.geeksforgeeks.org/brute-force-approach-and-its-pros-and-cons/ on May 20, 2023

[4] Geeks For Geeks. "Introduction to Backtracking – Data Structure and Algorithm Tutorials" accessed from https://www.geeksforgeeks.org/introduction-to-backtracking-data-structure-and-algorithm-tutorials/ on May 21, 2023

[5] "Tips on solving (3): The mathematics of Kakuro" accessed from https://theory.tifr.res.in/~sgupta/kakuro/algo.html on May 18, 2023

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 22 Mei 2023

Tabitha Permalla 13521111