Implementing the Backtracking Algorithm and the Brute Force Algorithm for Solving KenKen Puzzles

Muhammad Neo Cicero Koda - 13522108
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): mneocicerok@gmail.com

***Abstract*—This paper presents the development and evaluation of the backtracking algorithm and the brute force algorithm for solving KenKen puzzles. KenKen is a Sudoku-like puzzle game where a player must fill an n x n grid with numbers ranging from 1 to n while adhering to certain constraints. The brute force algorithm tries to go through all possible combinations of numbers in the grid. On the other hand, the backtracking algorithm only continues to expand on the list of possible solutions that still adhere to KenKen's constraints and prunes the rest. The findings of this paper show that the backtracking algorithm solves the puzzle in a considerably faster amount of time compared to the brute force algorithm due to how it selectively expands on promising solutions.**

Keywords—KenKen; backtracking; brute force; puzzle solving;

I. Introduction

KenKen, also known as KenDoku or Square Wisdom, is a Sudoku-like puzzle game where a player must fill an n x n grid using numbers ranging from 1 to n. The player must also adhere to the constraints given by the puzzle. These constraints include not having duplicate numbers in a row, not having duplicate numbers in a column, and requiring numbers that are grouped together in a cluster to result to a certain target number based on that cluster's operation. These operations include addition, subtraction, multiplication, and division.

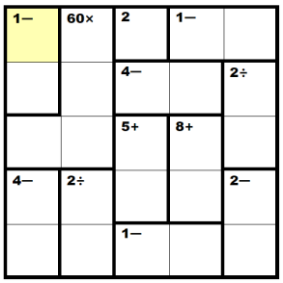

Figure 1. An example of a layout for a 5x5 KenKen puzzle. (Source: https://www.kenkenpuzzle.com/game)

Figure 1 shows an example of an empty 5x5 KenKen puzzle. For ease of reference, columns in the grid will be labeled alphabetically from left to right (starting from A) and rows will be labeled numerically from top to bottom (starting from 1). As an example, the square on the first row and third column will be labeled as C01. Clusters refer to groupings in the grid where numbers inside the cluster must result to that cluster's target number by a certain operation. For example, the cluster formed by squares A03, B01, B02, and B03 must be filled with numbers that add to eleven when summed together (noted by the 11+ label on square B02).

In more detail, if the target number on a cluster is labeled as n, the list of possible operations done on a cluster include:

- Addition (+): All numbers in the cluster must result to n when added together. There can be one or more squares in the cluster.
- Subtraction (-): Exactly two numbers are in the cluster. The two numbers must have a difference of n.
- Multiplication (x): All numbers in the cluster must result to n when multiplied together. There can be one or more squares in the cluster.
- Division (÷): Exactly two numbers are in the cluster. The value of n must be equal to the larger number divided by the smaller number.

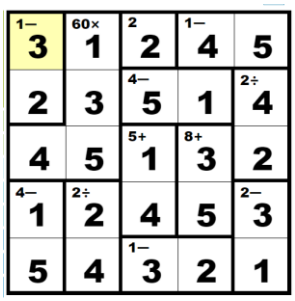

Figure 2. The solution for the puzzle on Figure 1. (Source: https://www.kenkenpuzzle.com/game)

Figure 2 shows the solution for the puzzle provided in Figure 1. As seen in the figure, each row and each column contain no duplicate values and every number in each cluster adheres to the constraints given by the cluster. For example, the cluster formed by squares A03, B01, B02, and B03 result to the number 60 when multiplied together.

This paper develops and evaluates two possible approaches for solving KenKen puzzles, one using brute force and one using backtracking. The two algorithms will be compared against each other in terms of their respective speed in solving the puzzle.

II. Basic Theory

A. Brute Force

The brute force algorithm is a straightforward approach to solve a given problem. The brute force algorithm is usually based on a problem statement and the concepts involved in that problem. It solves a problem using a simple, direct, and obvious way, which usually involves trying out all possible combinations for a problem and determining which one is the solution.

Most of the time, the brute force algorithm isn't the most efficient way to solve a problem because of its' high time and space complexity. It's more suitable for problems that have low input volume (n).

Even though it isn't the most efficient way of solving a problem, the brute force algorithm still has its' merits. It's often the most simple and straightforward approach to solving a problem. Besides that, almost all problems can be solved using the brute force algorithm. There are even problems that can only be solved by using brute force, such as finding the largest element in an unordered array.

Some examples of where the brute force algorithm is being used are in sequential search, counting factorials, string matching, and matrix multiplication.

```
   NOBODY NOTICED HIM
1  NOT
2   NOT
3    NOT
4     NOT
5      NOT
6       NOT
7        NOT
8         NOT
```

Figure 3. An example of brute force being used in string matching (Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Brute-Force-(2022)-Bag1.pdf)

Exhaustive search is a solution finding technique based on the brute force algorithm for problems related to combinatorics. The steps involved in exhaustive search are:

1. Enumerate all possible solutions in a systematic way.
2. Evaluate all possible solutions one by one and find the best solution found so far.
3. After the search ends, the best solution found so far will be the best solution.

In theory, exhaustive search will always result in a solution, but the time and space spent on finding the solution tend to be quite expensive.

B. Backtracking

Backtracking is an effective, structured, and systematic problem-solving method used for both optimization and non-optimization problems. It is a revision of exhaustive search. In exhaustive search, all possible solutions are explored and evaluated one by one. In backtracking, only options that lead to the solution are explored. Other options that don't lead to the solution are ignored/pruned.

The set of all possible solutions for a problem is called a solution space. For example, for The 0/1 Knapsack Problem where n = 3, the solution is expressed by $X = (x_1, x_2, x_3)$ where $x_i \in \{0, 1\}$ and the solution space is {(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)}.

The solution space can be organized into a tree-like structure where each node represents the state of the problem and each branch is labeled by values of x_i. The path from the tree's root to a leaf node represents a possible solution and the list of all possible solutions form a solution space. The organization of a solution space tree is referred to as a state space tree.

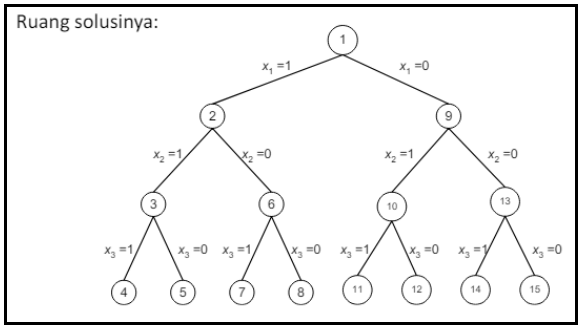

Figure 4. The solution space for The 0/1 Knapsack Problem, n = 3 (Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-backtracking-2021-Bagian1.pdf)

Finding a solution using backtracking involves generating nodes so that a path from the root to a leaf node is created. The generation of nodes follows a depth-first-search order. Generated nodes are called live nodes and live nodes that are currently being expanded are called expand nodes. Each time an expand node is expanded, the path that is being created by it becomes longer. If said path doesn't lead to the solution, the expand node is killed and becomes a dead node. The function used to kill the E-node is called a bounding function. When a node dies, its child nodes are implicitly pruned as well. If the creation of a path results in a dead node, the search continues by backtracking to the node in the previous level and continuing to generate one of its' other child nodes. That node becomes the new expand node. The search stops when a certain goal node is reached.

III. Implementation

A. Technical Details

The solver for KenKen puzzles using backtracking and brute force is implemented in C++. The program consists of two classes, the Cell class and the KenKen class.

The Cell class represents a square in the puzzle's grid. It contains information such as the current value being set in that cell, the target value and the operation being done in that cell's

cluster, and the coordinates of each member of that cell's cluster.

```
class Cell {
    private:
        int value;
        int target;
        char operation;
        vector<tuple<int, int>> cluster;
```

Figure 5. The Cell class and its' attributes

The KenKen class is where the main bulk of the program resides. The attributes stored in the KenKen class consist of the puzzle's grid size, the grid, and solutionFound, a boolean value that is true when a solution for the puzzle is found, false otherwise.

```
class KenKen {
    private:
        int gridSize;
        vector<vector<Cell>> grid;
        bool solutionFound;
```

Figure 6. The KenKen class and its' attributes

The constructor of the KenKen class takes in a file path as an argument and loads the contents of the file into its attributes. The puzzle is stored in a txt file that stores the puzzle's grid size and each cluster that makes up the puzzle.

```
3
3 / A01 B01
1 - A02 A03
12 * C01 B02 C02
2 - B03
```

Figure 6. An example of a txt file for a 3x3 puzzle.

An example of a puzzle's txt file is shown on Figure 6. The first line represents the grid size of the puzzle. The lines following that contain information about a cluster. The first token on each line represents the target value for that cluster. The second token represents the operation being done on that cluster. The tokens after that represent the coordinates that make up a cluster.

B. Implementing the Brute Force Algorithm

The approach used in implementing the brute force algorithm for solving KenKen puzzles is by exhaustive search. One way of doing this is by firstly generating all possible permutations of numbers ranging from 1 to n for a single row. For example, for an n x n puzzle where n = 5, there will be 5! = 120 possible permutations for a single row. After those permutations are generated, the algorithm evaluates every possible solution for the entire grid. In the previous case, where there are 120 possible permutations for a single row and 5 available rows, the algorithm will evaluate 120^5 possible solutions for the grid.

```
vector<vector<int>> permuteNumbers(vector<int>& numbers) {
    vector<vector<int>> permutations;
    do {
        permutations.push_back(numbers);
    } while (next_permutation(numbers.begin(), numbers.end()));
    return permutations;
}
```

Figure 8. The permuteNumbers function, used for generating every possible permutation of a set of numbers from 1 to n.

```
void findSolutionBruteForce
(vector<vector<Cell>>& gr, vector<vector<int>>& permutations, int i) {
    if (solutionFound) {
        return;
    }

    if (i == gridSize) {
        if (isSolution(gr)) {
            grid = copyGrid(gr);
            solutionFound = true;
        }
        return;
    }

    for (vector<int> permutation : permutations) {
        if (solutionFound) {
            return;
        }

        vector<int> temp;
        for (int j = 0; j < gridSize; j++) {
            temp.push_back(gr[i][j].getValue());
            gr[i][j].setValue(permutation[j]);
        }

        findSolutionBruteForce(gr, permutations, i + 1);

        for (int j = 0; j < gridSize; j++) {
            gr[i][j].setValue(temp[j]);
        }
    }
}
```

Figure 9. The findSolutionBruteForce function

Figure 9 shows the implementation of the brute force algorithm for solving KenKen puzzles. It utilizes the permutations generated from Figure 8 to go through possible solutions for a puzzle. Note that in this function, the possible solutions generated after finding a solution will not be evaluated. It uses the solutionFound attribute to terminate as soon as a solution is found. This modification was made due to time efficiency considerations.

The brute force algorithm iterates through every possible permutation. The function's i parameter determines the index of the row to be set by one of the permutations. For every permutation, it sets the values on the i-th row of the grid to a possible permutation. Then, it sets the values of the next row by recursively calling itself and incrementing i by one. It then reverts the grid's state to before the values of the i-th row were set. When the value of i is equal to the grid's size (when all the rows are assigned to a permutation), the function checks whether the possible solution is an actual solution. If it is, it copies the grid to the grid attribute of the class and tries to terminate itself.

```
vector<int> numberList;
for (int i = 1; i <= gridSize; i++) {
    numberList.push_back(i);
}
vector<vector<int>> permutations = permuteNumbers(numberList);
findSolutionBruteForce(gridCopy, permutations, 0);
```

Figure 10. The initial call to the brute force algorithm

C. *Implementing the Backtracking Algorithm*

Before implementing the backtracking algorithm, elements of the puzzle are first mapped onto elements of the backtracking algorithm.

1. Solution

   The solution for this problem is expressed as an (n*n)-tuple: X = (x_1, x_2, …, x_n), where $x_i \in S_i$ and S_i = {1, 2, 3, 4, 5, 6, 7, 8, 9}.

2. Generating function

   The generating function T() returns the same values for all x_i. It will return numbers ranging from 1 to n.

3. Bounding function

```
bool placeable(vector<vector<Cell>>& gr, int row, int col, int val) {
    vector<vector<Cell>> gridCopy = copyGrid(gr);
    gridCopy[row][col].setValue(val);

    if (!isRowValid(gridCopy, row)) {
        return false;
    }

    if (!isColValid(gridCopy, col)) {
        return false;
    }

    if (!isClusterValid(gridCopy, row, col, false)) {
        return false;
    }

    return true;
}
```

Figure 11. The placeable function

The bounding function B() evaluates to true when (x_1, x_2, x_3, …, x_k) doesn't break any constraints. In this case, the bounding function is the placeable function, which checks whether the row, column, and cluster of the cell that is being set a value is still valid.

```
void findSolutionBacktrack(vector<vector<Cell>>& gr, int k) {
    int i = k / gridSize;
    int j = k % gridSize;
    for (int val = 1; val <= gridSize && !solutionFound; val++) {
        if (placeable(gr, i, j, val)) {
            vector<vector<Cell>> gridCopy = copyGrid(gr);
            gridCopy[i][j].setValue(val);

            if (k == (gridSize * gridSize - 1) && isSolution(gridCopy)) {
                grid = copyGrid(gridCopy);
                solutionFound = true;
            }

            if (k < (gridSize * gridSize  - 1)) {
                findSolutionBacktrack(gridCopy, k + 1);
            }
        }
    }
}
```

Figure 12. The findSolutionBacktrack function

The findSolutionBacktrack function takes two arguments, the first argument is the current state of the grid and the second, k, is the index of the current value being set. The value of k ranges from 0 to n * n – 1, where n is the size of the grid. The function first translates the value of k to a row (i) and column (j) index. It then iterates through numbers ranging from 1 to the size of the grid. If the number is placeable in the cell at row i and column j, it sets the value of that cell to that number. It then checks whether the values in the grid form a solution. If it does, the grid will be copied into KenKen's grid attribute and solutionFound will be marked as true. If the grid still has missing values, the function will recursively call itself with an incremented k value to set the next value in the grid.

```
findSolutionBacktrack(gridCopy, 0);
```

Figure 13. The initial call to the backtracking algorithm

IV. Testing and Analysis

A. *Testing*

The two algorithms will be tested in terms of their accuracy and speed in generating the right solution. Tests will be done using puzzles with varying grid sizes.

1. Test Case 1: 3 x 3 Puzzle

```
3
3 / A01 B01
1 - A02 A03
12 * C01 B02 C02
2 - B03
```

Figure 13. Test case 1

```
Solution using backtracking:
3 1 2
1 2 3
2 3 1

Time taken: 442 microseconds.

Solution using brute force:
3 1 2
1 2 3
2 3 1

Time taken: 296 microseconds.
```

Figure 14. Results for test case 1

2. Test case 2: 4 x 4 Puzzle

```
4
2  - A01 B01
3  - A02 A03
1  - A04 B04
3  + B02 C02
48 * B03 C03 C04
2  + C01
1  - D01 D02
2  / D03 D04
```

Figure 15. Test case 2

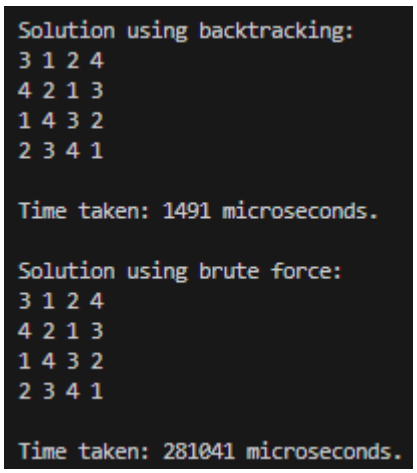

Figure 16. Results for test case 2

3. Test case 3: 5 x 5 Puzzle

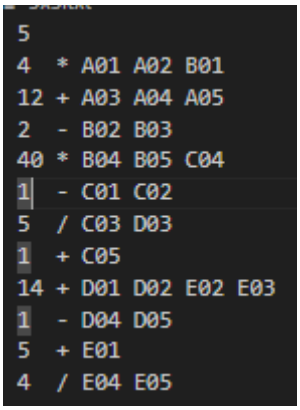

Figure 17. Test case 3

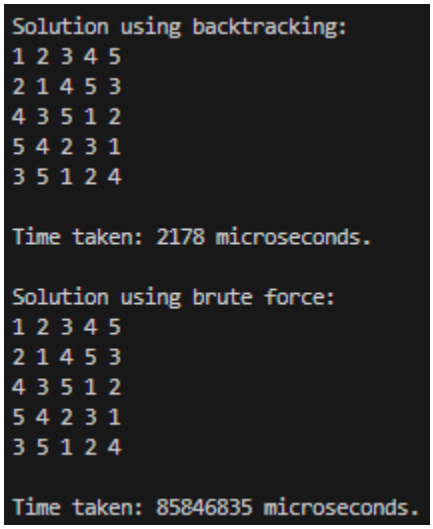

Figure 18. Results for test case 3

4. Test case 4: 6 x 6 puzzle

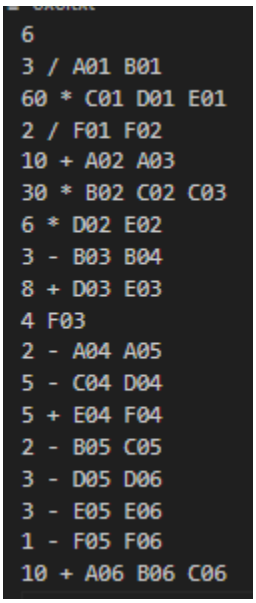

Figure 19. Test case 4

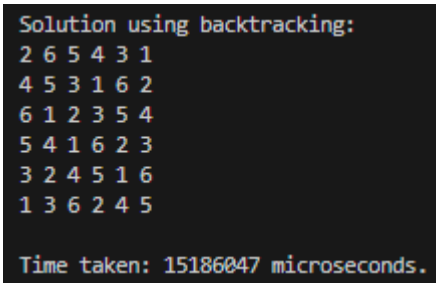

Figure 20. Results for test case 4

Note that for puzzles with grid size 6 and upwards, the brute force algorithm fails to find the solution in a reasonable amount of time (less than 1 hour).

5. Test case 5: 7 x 7 puzzle

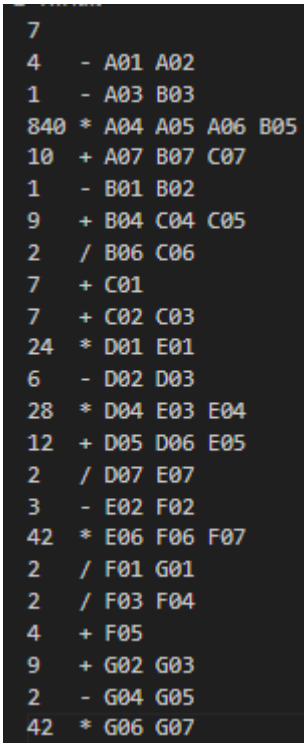

Figure 21. Test case 5

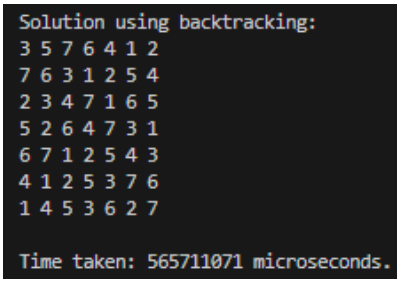

Figure 22. Results for test case 5

B. Analysis

Test cases	Time taken (μs)	
	Brute force	**Backtracking**
1	296	442
2	281041	1491
3	85846835	2178
4	-	15186047
5	-	565711071

Table 1. Test case results

While both algorithms succeed at finding an accurate solution for each test case that it finished, overall, the backtracking algorithm is considerably faster than the brute force algorithm. This showcases the effectiveness of the pruning done by the backtracking algorithm for skipping unnecessary steps.

The data in Table 1 shows an exponential growth in the time needed for the brute force algorithm to complete a single puzzle. When performing a search on an n x n puzzle, the brute force algorithm makes n! permutations for a row. Because there are n rows, the algorithm goes through a maximum of $(n!)^n$ possible solutions for the puzzle, making it have a time complexity of $O(n!)^n$, which is incredibly slow.

Test case 1 shows an anomaly in the speed of the two algorithms. In this test case, the brute force algorithm is faster than the backtracking algorithm. This might be due to the smaller number of possibilities being checked by the brute force algorithm when n is small combined with the fact that the brute force algorithm generates solutions row by row instead of cell by cell.

V. Conclusion

In this paper, two approaches for solving KenKen puzzles are developed, one using brute force and one using backtracking. The two algorithms also are compared against each other in terms of their speed at solving the puzzle. The test results show that the backtracking algorithm is considerably faster than the brute force algorithm due to its nature of pruning unpromising solutions.

Video Link at Youtube

https://youtu.be/ay1tKjsyQxk

Acknowledgment

The author would like to express their gratitude to the following individuals:

1. God Almighty for the blessings and grace that provided the author strength in writing and completing this paper.
2. The author's parents for their support and encouragement throughout the writing process.
3. Dr. Ir. Rinaldi Munir, M.T., Dr. Nur Ulfa Maulidevi, and Dr. Ir. Rila Mandala, lecturers of the Algorithm Strategies course in the even semester of 2023/2024, for providing knowledge that proved to be essential in writing this paper.

The author would also like to express their gratitude to all references utilized in this paper and would like to apologize if there are any errors present in this paper.

References

[1] Munir, Rinaldi. 2021. "Algoritma Runut-balik (Backtracking) (Bagian 1)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-backtracking-2021-Bagian1.pdf, accessed on June 1st, 2024.

[2] Munir, Rinaldi. 2021. "Algoritma Runut-balik (Backtracking) (Bagian 2)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-backtracking-2021-Bagian2.pdf, accessed on June 1st, 2024.

[3] Munir, Rinaldi. 2022. "Algoritma Brute Force (Bagian 1)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Brute-Force-(2022)-Bag1.pdf, accessed on June 1st, 2024.

[4] Munir, Rinaldi. 2022. “Algoritma Brute Force (Bagian 2)”. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Brute-Force-(2022)-Bag2.pdf, accessed on June 1st, 2024.

[5] KenKen Puzzle Official Website. n.d. "KenKen Puzzle". https://www.kenkenpuzzle.com/game, accessed on June 1st, 2024.

APPENDIX

The source code implemented in this paper can be found on this GitHub repository:
https://github.com/neokoda/KenKen-Puzzle-Solver\

DECLARATION

I hereby declare that this paper that I have written is my own work, not an adaptation or translation of someone else’s paper, and not a result of plagiarism.

Bandung, 12 June 2024

Muhammad Neo Cicero Koda 13522108