Solving Binary Puzzles Using Brute Force Algorithm and Backtracking Algorithm

Andjani Kiranadewi/13518109
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: 13518109@std.stei.itb.ac.id

***Abstract*—Binary Puzzle is a logic puzzle where the objective is to fill the empty cells with either 0 or 1 while still conforming to the rules where each row and column are unique, has equal numbers of 0s and 1s, and there are no more than two same numbers adjacent horizontally nor vertically to each other.**

Keywords—backtracking algorithm; binary puzzle; brute force algorithm;

I. INTRODUCTION

A puzzle is a type of a game that tests a person's knowledge by challenging them into solving it. People often need to put all the hints provided together in a logical manner to solve a puzzle. It is often used as a form of entertainment, something to waste time on.

There are multiple categories of puzzles. For example, jigsaw puzzle, which is probably the most well-known out of all, where all of the pieces of a jigsaw puzzle need to be put in a certain way to get the complete picture. There is also mathematical puzzle, where to find the solution, mathematical skills are required. Another popular type of puzzle is logical puzzle, where to solve the puzzle, logical thinking is needed.

This paper will discuss a type of logic puzzle, which is binary puzzle, and a way to solve it using two types of algorithm: brute force algorithm and backtracking algorithm.

II. BASE THEORIES

A. *Binary Puzzle*

1			0		
		0	0		1
	0	0			1
0	0		1		
	1			0	0

Figure 1. An example of a binary puzzle (Source: http://www.binarypuzzle.com/)

Binary puzzle, also known as Binairo, Takuzu, Tohu-wa-Vohu[1], or Binaire Puzzels[2] is a type of logic puzzle where the objective is to fill the grid, usually square-shaped with even amount of cells each row and column, with two symbols, often 0s and 1s, hence the name "binary puzzle".

At the start of the puzzle, the grid will be partially filled to serve as a hint of the puzzle. From the hint provided, the player must complete the grid with one of the two symbols, while obeying the rules of the puzzle.

Binary puzzle has several rules:

- More than two equal numbers can't be adjacent to each other, vertically nor horizontally.
- Each row and column must be unique
- Each row and column must contain equal numbers of both symbols [3]

B. *Brute Force Algorithm*

The brute force algorithm is an algorithm that "brute forces" its way to find the solution. It is an obvious way to solve a problem.

There are several characteristics of a brute force algorithm:

- It is not a 'smart' and effective solution since solving using the brute force algorithm usually requires a lot of computation and time.
- It is better used for small scale problems.
- Almost every problem can be solved using the brute force algorithm [4]

Example usage of the brute force algorithm is array sorting using bubble sort. This method of sorting simply swaps adjacent elements if it is in the wrong order. Such a method has the time complexity of $O(n^2)$, which can be improved by using several tricks, which will not be discussed in this paper.

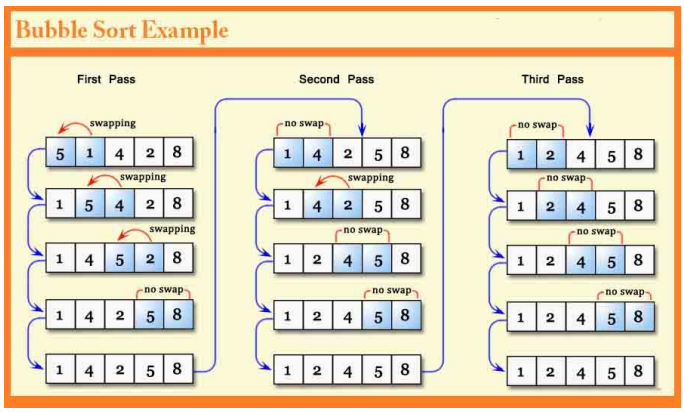

Figure 2. Visualization of bubble sort (Source: https://gubuktekno.com/program-bubble-sort-bahasa-c/)

C. *Backtracking Algorithm*

The backtracking algorithm is an optimization for exhaustive search. Unlike exhaustive search, which explores all of the possible solution available, backtracking algorithm builds up the solution one step at a time using a recursive method, while abandoning (backtracks from) any candidate solution that impossible to create the final, valid solution from. The term 'backtrack' itself was coined by D.H. Lehmer in the 1950s [5].

There are three properties of a backtracking algorithm:

1. Solution of the problem

Solution is represented as a n-tuple: X = (x1 , x2 , …, xn), xi ∈ Si. It is possible that $S_1 = S_2 = \ldots = S_n$.

2. Generating function

The generating function for x_i is represented as a predicate T(i), which will generate x_i for the solution component.

3. Bounding function

The bounding function is represented Predicate $B(x_1,x_2,\ldots,x_n)$. B is true if x_i leads towards a valid solution and false if it is not.

Backtracking algorithm's basic method of finding a solution is as follows:

1. Start with an empty solution set S. S = {}
2. Add to S the first move that is still left (All possible moves are added to S one by one). This now creates a new sub-tree s in the search tree of the algorithm.
3. Check if S+s satisfies each of the constraints in C.
4. If Yes, then the sub-tree s is "eligible" to add more "children".
5. Else, the entire sub-tree s is useless, so recurs back to step 1 using argument S.
6. In the event of "eligibility" of the newly formed sub-tree s, recurs back to step 1, using argument S+s.
7. If the check for S+s returns that it is a solution for the entire data D. Output and terminate the program.
8. If not, then return that no solution is possible with the current s and hence discard it [6]

III. Solution Using Brute Force Algorithm

The usage of brute force algorithm for solving binary puzzles is straight-forward—all possible combination of filled grid of same size are checked if it is a solution. The combination is a solution if:

- Abide by the rules stated in section II.A.
- Possible to reach from the original grid.

Because all of the possible combinations of the grid is checked, the method is highly ineffective. As binary puzzle can be classified as subset problem, the time complexity for binary puzzle with the size of nxn is $O(2^{n^2})$, which means that a 6x6 binary puzzles can have 2^{36} or 68,719,476,736 combinations to be checked.

There are two basic steps in finding the solution: subset generation, and validation.

A. *Subset generation*

Binary puzzle, as its name implies, is a problem involving an element being in two states. Thus, a binary puzzle can be represented using a binary digit. Consider the 2x2 binary puzzle grid of Figure 3.

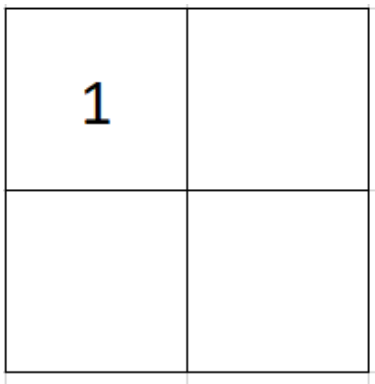

Figure 3. An example of a 2x2 binary puzzle (Source: Author's personal document)

In each cell in the grid, there are only two state: 0 or 1. All of the possible combination for the grid can be, from the top left cell to the bottom right cell, ranging from all cells filled with 0s to all cells filled with 1s.

To generate a subset using a code, the grid can be 'flattened' into a 1-dimensional array, instead of 2-dimensional. From there, a subset can be generated from number 0 (binary digit 0000) to number 15 (binary digit 1111). In another words, an nxn binary puzzle can be represented using binary digits with the length of n^2.

```
public int[][] generateSubset(int size, int n){
  int[][] sub = new int[size][size];
  for (int i = 0; i < size; i++){
    //i-th element of the flatten grid has value as 1
    if ((n & (1 << i)) > 0){
      sub[i/size][i%size] = 1;
    } else {
      sub[i/size][i%size] = 0;
    }
  }
  return sub;
}
```

B. Validation

The combination acquired from the subset generator need to be validated for it to be deemed as a solution. As stated in section IIA, there are three rules for a solution to be counted as valid.

To do this, every row and column of the grid need to be checked whether each row is unique towards each other and each column likewise.

```
  public boolean checkRows(int[][] grid){
    int trueCount, falseCount;
    HashSet<Integer> values = new HashSet<Integer>();
    int completed = 0;
    StringBuilder current;

    int last;
    int count;
    for (int[] ints : grid) {
        trueCount = 0;
        falseCount = 0;
        current = new StringBuilder();
        count = 0;
        last = -1;

        for (int j = 0; j < grid.length; j++) {
            if (ints[j] != -1) {

current.append(Integer.toString(ints[j]));
                if (ints[j] == 1) {
                    trueCount++;
                } else {
                    falseCount++;
                }

                if (last == b || last == ints[j]) {
                    count++;
                } else {
                    count = 1;
                }
                last = ints[j];

                if (count == 3) {
                    return false;
                }
            } else {
                last = -1;
                count = 0;
            }
        }

        if (current.length() == grid.length) {
            if (trueCount != falseCount) return false;

values.add(Integer.parseInt(current.toString(), 2));
            completed++;
        }
    }
    return values.size() == completed;
}
```

```
  public boolean checkCols(int[][] grid){
    int trueCount, falseCount;
    HashSet<Integer> values = new HashSet<Integer>();
    StringBuilder current;
    int completed = 0;

    int last;
    int count;

    for (int i = 0; i < grid.length; i++){
        trueCount = 0;
        falseCount = 0;
        current = new StringBuilder();
        count = 0;
        last = -1;

        for (int[] ints : grid) {
            if (ints[i] != -1) {

current.append(Integer.toString(ints[i]));
                if (ints[i] == 1) {
                    trueCount++;
                } else {
                    falseCount++;
                }
                //System.out.println(last+" "+grid[i][j]);
                if (last == b || last == ints[i]) {
                    count++;
                } else {
                    count = 1;
                }
                last = ints[i];

                if (count == 3) {
                    return false;
                }
            } else {
                last = -1;
                count = 0;
            }
```

```
        }

        if (current.length() == grid.length) {
            if (trueCount != falseCount) return
false;

values.add(Integer.parseInt(current.toString(),
2));
            completed++;
        }
    }
    return values.size() == completed;
}
```

C. Experiment

The experiment towards brute force approach towards binary puzzle solving, unfortunately, can only be done, at most, on a 4x4 puzzle. This is caused by exponential nature of the amount of combination possible, which is $2^{n\wedge 2}$. The 6x6 binary puzzle has been tested, but it took nearly an hour to get the results.

1. Experiment 1

   Input:

```
public static void main(String[] args) {
    //b = blank space
    int[][] grid = {{1,b,b,0},
                    {b,b,0,0},
                    {b,0,0,b},
                    {b,b,b,b}};
    Binairo b = new Binairo(grid);
    b.solve();
}
```

Figure 4. Input of 4x4 grid for Experiment 1 of Brute Force Algorithm

   Output:

```
Binairo
"C:\Program Files\Java\jdk-12.0.2\bin\java.exe"
Time elapsed: 29ms
1 0 1 0
1 1 0 0
0 0 0 0
0 1 0 0

Process finished with exit code 0
```

Figure 5. Output of completed puzzle in Figure 4

2. Experiment 2

   Input:

```
public static void main(String[] args) {
    //b = blank space
    int[][] grid = {{b,b,b,0},
                    {b,b,b,b},
                    {b,1,b,b},
                    {0,1,b,1}};
    Binairo b = new Binairo(grid);
    b.solve();
```

Figure 6. Input of 4x4 grid for Experiment 2 of Brute Force Algorithm

   Output:

```
Binairo
"C:\Program Files\Java\jdk-12.0.2\bin\java.exe"
Time elapsed: 343ms
1 0 1 0
1 0 0 1
0 1 0 0
0 1 0 1

Process finished with exit code 0
```

Figure 7. Output of completed puzzle in Figure 6

3. Experiment 3

   Input:

```
public static void main(String[] args) {
    //b = blank space
    int[][] grid = {{b,b,b,1},
                    {b,b,0,b},
                    {0,b,b,b},
                    {b,1,1,0}};
    Binairo b = new Binairo(grid);
    b.solve();
}
```

Figure 8. Input of 4x4 grid for Experiment 3 of Brute Force Algorithm

   Output:

```
Binairo
Time elapsed: 61ms
1 0 0 1
1 1 0 0
0 0 0 0
0 1 1 0

Process finished with exit code 0
```

Figure 7. Output of completed puzzle in Figure 8

IV. SOLUTION USING BACKTRACKING ALGORITHM

A. Basic properties

1) Solution of the problem

2) Generating function

The generating function T(i,j) generates an integer of either 0 or 1 for x_{ij} if x_{ij} is empty (no hint for x_{ij}).

3) Bounding function

The bounding function performs three kinds of operation:

- Adjacency checking

  This function returns true if there are no more than two of the same digits adjacent with each other, vertically or horizontally, otherwise returns false.

- Uniqueness checking

  This function returns true if all rows and columns are unique (no sequence are present in more than one row or column), otherwise returns false.

- Equality checking

  This function returns true if all rows and columns have the same amount of 0s and 1s. This can be checked whether the amount of 0 = 1 = n/2, with n being the number of rows. Otherwise, returns false.

B. Visualization of the Solution Space

To demonstrate the use of backtracking algorithm to solve a binary puzzle, Figure 1 will be used as an example.

The root of the solution space, or the 1st node, is the starting point—the initial state of the puzzle. As x_{11} is already filled to serve as a hint, the generating function will generate a value for x_{12}, starting from 0, which will become Node 2.

1	0		0		
		0	0		1
	0	0			1
0	0		1		
	1			0	0

Figure 8. Node 2 of the Solution Space

The placement of number 0 is completely valid, as it does not violate any of the rules. Thus, the generating function will continue by generating a value for x_{13}, starting from 0, which will become Node 3.

1	0	0	0		
		0	0		1
	0	0			1
0	0		1		
	1			0	0

Figure 9. Node 3 of the Solution Space

This node violates the rule of the puzzle: no more than two of the same digits adjacent to each other. So, the node is deemed as a dead node, as it won't lead to any valid solution. Since Node 3 is dead, we return (backtrack) to Node 2, and generate another value for x_{13}, which is 1, which will become Node 4.

1	0	1	0		
		0	0		1
	0	0			1
0	0		1		
	1			0	0

Figure 10. Node 3 of the Solution Space

Now, the solution space is as follows:

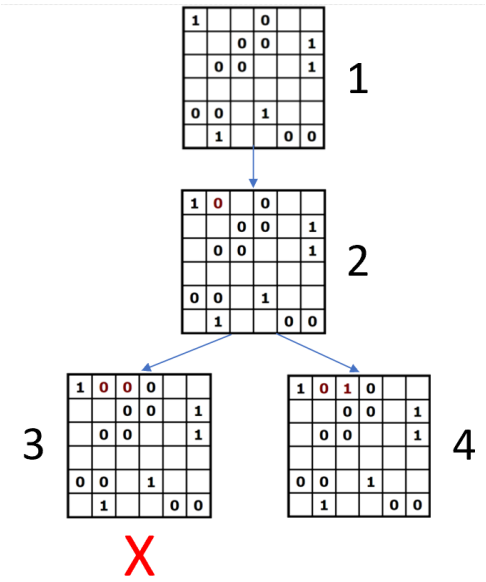

Figure 11. Solution Space up to Node 4

Similar method can be used until a valid solution is found—generate the next value if the bounding function is satisfied, backtrack to previous node if the bounding function is not satisfied. Below is the finished solution space:

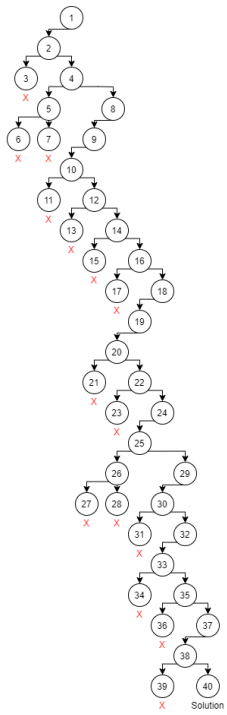

Figure 12. Finished solution space

1	0	1	0	1	0
0	1	0	0	1	1
1	0	0	1	0	1
0	1	1	0	1	0
0	0	1	1	0	1
1	1	0	1	0	0

Figure 13. Node 40 of Solution Space (Puzzle solution)

C. Code Implementation

The Java code implementation of backtracking algorithm to solve a binary puzzle are as follows:

```
   public void backtrack(int[][] grid, int i,
int j){

    //i = row, j = column
    int[][] temp = cloneGrid(grid);


    //if all of the cells are filled
    if (i == grid.length) {
        if (checkCols(grid) && checkRows(grid))
{
            printGrid(grid);
            System.exit(0);
        }
    } else {

        //Generating function
        for (int k = 0; k <= 1; k++) {
            if (this.grid[i][j] == b) {
                temp[i][j] = k;
            }

             //if no rules are violated
            if (checkRows(temp) &&
checkCols(temp)) {

                //proceed to the next cell
                if (j == grid.length - 1)
                    backtrack(temp, i + 1, 0);
                else
                    backtrack(temp, i, j + 1);
            }
        }
    }
}
```

The bounding function checkRows() and checkCols() are the ones in section IIIB.

D. Experiments

Since backtracking algorithm prunes the nodes that don't lead to a valid solution, the effectiveness is greatly improved. At the worst case, the time complexity is still the same, which is $O(2^{n^2})$.

1. Experiment 1

Input:

```
public static void main(String[] args) {
    //b = blank space
    int[][] grid = {{1,b,b,0,b,b},
                    {b,b,0,0,b,1},
                    {b,0,0,b,b,1},
                    {b,b,b,b,b,b},
                    {0,0,b,1,b,b},
                    {b,1,b,b,0,0}};
    Binairo b = new Binairo(grid);
    b.backtrack(grid, i: 0, j: 0);
}
```

Figure 14. Input of 6x6 grid for Experiment 1 of Backtracking Algorithm

Output:

```
Binairo
"C:\Program Files\Java\jdk-12.0.2\bin\java.exe"
1 0 1 0 1 0
0 1 0 0 1 1
1 0 0 1 0 1
0 1 1 0 1 0
0 0 1 1 0 1
1 1 0 1 0 0

Time elapsed: 26ms

Process finished with exit code 0
```

Figure 15. Output of completed puzzle in Figure 14

2. Experiment 2

Input:

```
public static void main(String[] args) {
    //b = blank space
    int[][] grid = {{b,b,b,b,0,0},
                    {b,b,b,b,b,b},
                    {b,1,b,b,b,b},
                    {0,1,b,1,b,b},
                    {0,b,b,b,0,b},
                    {b,b,1,b,b,b}};
    Binairo b = new Binairo(grid);
    b.start = System.currentTimeMillis();
    b.backtrack(grid, i: 0, j: 0);
}
```

Figure 16. Input of 6x6 grid for Experiment 2 of Backtracking Algorithm

Output:

```
Binairo
"C:\Program Files\Java\jdk-12.0.2\bin\java.exe"
1 1 0 1 0 0
0 0 1 0 1 1
1 1 0 0 1 0
0 1 0 1 0 1
0 0 1 1 0 1
1 0 1 0 1 0

Time elapsed: 131ms

Process finished with exit code 0
```

Figure 17. Output of completed puzzle in Figure 16

3. Experiment 3

Input:

```
public static void main(String[] args) {
    //b = blank space
    int[][] grid = {{b,b,b,1,b,b},
                    {b,b,0,b,b,1},
                    {0,b,b,b,0,b},
                    {b,1,1,b,b,b},
                    {b,b,b,b,b,b},
                    {1,b,b,b,0,b}};
    Binairo b = new Binairo(grid);
    b.start = System.currentTimeMillis();
    b.backtrack(grid, i: 0, j: 0);
}
```

Figure 18. Input of 6x6 grid for Experiment 3 of Backtracking Algorithm

Output:

```
Binairo
"C:\Program Files\Java\jdk-12.0.2\bin\java.exe"
0 1 0 1 1 0
1 0 0 1 0 1
0 1 1 0 0 1
0 1 1 0 1 0
1 0 0 1 1 0
1 0 1 0 0 1

Time elapsed: 25ms

Process finished with exit code 0
```

Figure 19. Output of completed puzzle in Figure 18

VIDEO LINK AT YOUTUBE

https://youtu.be/OmXpMgNRTfU

ACKNOWLEDGMENT

The author would like to thank God for giving the author the ability to finish this paper in time. The author also expresses gratitude towards Strategi Algoritma professors at ITB for giving the author knowledge to create this paper.

REFERENCES

[1] https://www.janko.at/Raetsel/Tohu-Wa-Vohu/index.htm, accessed on May 1st 2020

[2] http://www.clarity-media.co.uk/binary-puzzles.php], accessed on May 2nd 2020

[3] http://www.binarypuzzle.com/rules.php, accessed on May 2nd 2020

[4] http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2017-2018/Algoritma-Brute-Force-(2016).pdf

[5] Rossi, Francesca; Beek, Peter Van; Walsh, Toby. "Handbook of Constraint Programming". Amsterdam: Elsevier. p. 14, August 2006.

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, 29 April 2020

Andjani Kiranadewi/13518109