The Use of Backtracking Algorithms in Attempt to Solve Picross Puzzle

Vincent Hendryanto Halim / 13515089
Program Studi Teknik Informaitka
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13515089@stei.std.itb.ac.id
vincenthendrha@gmail.com

***Abstract*—This writing defines ways to solve picross puzzle using exhaustive search and the backtracking algorithm. Exhaustive search algorithm are done using combinatorics on the solution set, while backtracking algorithm selects the feasible solution on each row to create the solution for the whole picross puzzle. The result of backtracking and exhaustive search algorithms are then compared with the result that backtracking has the best solve time than exhaustive search because of the number of iteration needed.**

***Keywords*—puzzle; picross; backtracking; exhaustive search**

I. Introduction

Puzzle games are a kind of games that require one's thinking skill to solve the game. During the age of technology, many puzzle games evolved into the form of video games to fit so that puzzle games can still exist in this age. There are many kinds of puzzle games, such as Tetris, Minesweeper, and the one that'll be discussed, Picross.

Picross is a puzzle game originated from japan that utilizes the form of grids in the form of the game. Picross requires player to mark specific tiles in the grid using the rule defined to form a picture from the marked tiles so that the marked tiles doesn't violate the constrain on each row and column. Picross has many variations, for example, the multi-colored Picross, the one-colored Picross, or the 3D Picross. The variance between Picross puzzles affects the rules of the picross puzzle iself. Due to those variations, the Picross that will be used in this writing to utilize backtracking is the one-colored Picross.

Picross puzzles are constraint satisfaction problems as defined by the row and column constraint on each row and column. Moreover Picross puzzles have almost the same characteristics with the n-queen problem in terms of placing without violating the constrain. Therefore, the writer chooses backtracking algorithm to solve the puzzle.

Through this writing, the writer will attempt to explain the use of backtracking and exhaustive search to solve Picross puzzles, by forming the backtracking algorithm and exhaustive search algorithm used using pseudocodes and the implementation of the backtracking algorithm.

II. Theories on Backtracking

Backtracking algorithm is an algorithm that returns to the previous step. If the current path of step followed doesn't lead to a solution, the algorithm will return to the previous step to try another solution and fails the step. This algorithm is basically the same as the Exhaustive Search algorithm, except Backtracking has pruning function that helps optimize the algorithm.

In backtracking, there are some primary components in order to create the algorithm :

1. Solution

The solution is expressed by vector of n-tuple: $X = (x_1, x_2, \ldots, x_n)$, $x_i \in S_i$.

2. Generation Function of X_k

Generation Function is expressed by the expression T(k). T(k) generates values for the solution.

3. Bounding Function

Bounding Function is expressed by the expression $B(x_1, x_2, \ldots, x_k)$. Bounding function is the function that bounds the path if the path doesn't lead to the solution. Due to that fact, Bounding function would return a true value if the current path checked leads to a solution

The generated solution from bactracking algorithm is expressed into a state space tree. A tree structure is a collection of nodes and vertex that is connected, and doesn't have a circuit. Thus, space state tree is a tree structure that contains all the state possible of a problem as nodes.

Later on, the algorithm will try to explore the state space tree in a depth-first-search-like algorithm. Then the bounding function will check if the state will lead to a solution. If the current state doesn't lead to a solution, the state will be cut-off by the bounding function, preventing it to be explored further.

Backtracking algorithm can be defined as a recursive algorithm. The pseudocode for a recursive backtracking algorithm is defined in below

```
function backtrack(move taken) → boolean

Variables:
r : array of array of integer

Algorithm:
if(move is solution){
   → true
}
else{
   for (every possible move)
      if(backtrack(move))
         → true
      endif
   endfor
   → false
endif
```

In the recursive part of the backtrack algorithm, it tries every single possible move, and if the move isn't possible, it returns a false value. Thus backtracking to the caller.

III. Theories on Brute Force

Brute force algorithm is a method to solve a problem based on the problem statement and the concept definition.

This algorithm is easy to comprehend and will basicly solve any problem, however in some cases, brute force algorithm is a slow algorithm, thus is not acceptable.

A. *Exhaustive Search*

Exhaustive search is a solution-searching technique using bruteforce by generating a solution and evaluates it using the solution's rule.

This method usually involves the problem that uses combinatorics or subset to solve their problems with a clear rule on how to evaluate the answer.

Exhaustive search can be divided into 3 parts in general :

1. Enumeration (list) of solution

The solution is listed by either using combinatorics or listing the subset of a set

2. Evaluation of solution's feasibility

The selected solution is then evaluated by using the evaluation function defined by the problem or by the constraint that is set by the problem

3. Output of the best solution

Generally the pseudocode for exhaustive search is defined below :

```
function exhaustiveSearch()→solution
   while(!solution) do
      //Generate Solution
      //Evaluate the Solution
   →solution
```

IV. The Picross Puzzle

A. *Rules of The Game*

Picross is a puzzle game that uses the form of grids in the gameplay of the game. Tiles in the grids has to be colored following clues in order to form a picture in the grid.

The clues for each row and column are written on the side of the rows and on top of columns specifying how many unbroken lines of colored squares there are in the row or column.

Another rule is every unbroken line has to have at least an empty tile separating it from the other unbroken line. Thus prevents the same meaning between, for example the "1 2" rule and "3" rule.

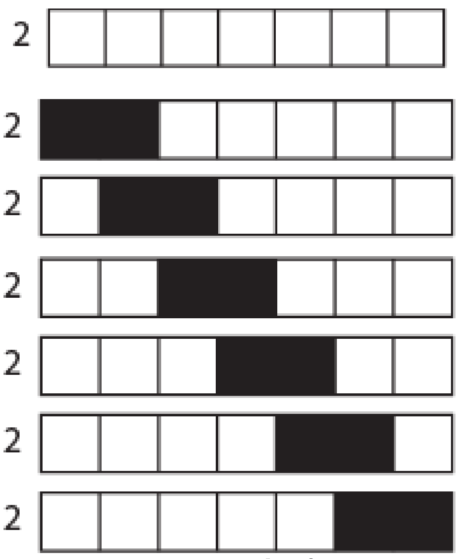


Figure 1. Example of a row

If there is one number for the clue with the number *n*, it means that the row/column will only have one unbroken line with the length of *n* tiles. From the example in Figure 1, the number 2 means that there is one unbreakable line with length of two tiles, and thus can be placed like the rest 6 column. Consequently the number 0 on a column or row, means that the row/column is empty.

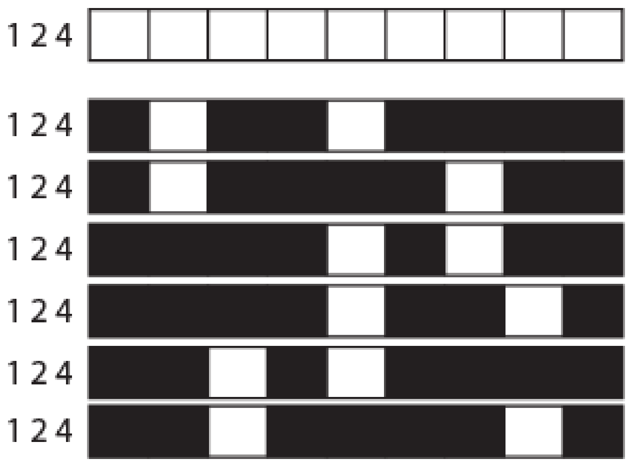


Figure 2. Example of a multi-line row

If there are more than one number on a row/column, it means that the row/column is going to have a number of unbroken lines in the column. For example, in Figure 2 above , the number "1 2 4" in the side of the column, means that there will be 3 unbroken lines in the respective column or row, each consisting of one, two, and four tiles, with at least one space between each line.

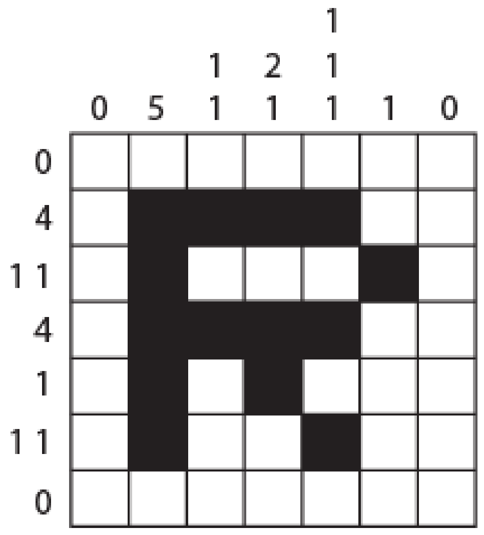


Figure 3. Example of a completed Picross Puzzle

The figure above is an example of a complete picross puzzle for each columns and rows follows the clues on the respective columns and rows. (i.e. the second column has one unbroken lines, consisting of 5 tiles; the fifth column has 3 unbroken lines, each consisting of one tile).

V. Exhaustive Search Algorithm in Solving Picross

The bruteforce algorithm or exhaustive search for solving picross only utilizes the properties of combinatorics in order to solve the puzzle and evaluation of solution after generating. In general the pseudocode is as defined below

A. Enumeration of Solution

Enumeration of Solutions is done by using permutation to generate the solution for each row by first, inserting the blank lines as lines with the length of 1 so that the total length of the row is the same as the width. After that, the permutation is invoked.

B. Evaluation of Solution's feasibility

The solution is checked by matching the solution with the column constraint, since the solution is generated based on the row constraint. The column constraint is checked by enumerating all the unbroken lines on a row with their length. If the number of unbroken line and their length matches the column constraint, then the column constraint on that column is fulfilled. Furthermore, if all the column of the solution matches the column constraint, then the solution is correct.

C. Pseudocode Implementation

The implementation for the exhaustive search algorithm is defined in the pseudocode below

```
function solvePicross(P: Picross) → Solution
//Declaration
S : solution

//Algorithm
while not(solution_correct(S))
   for each row
      S ← generate_row_solution()
      //Generates a solution that satisfies the
constraint
   endfor

   check_solution
   if (solution) then
      solution_correct ← true
   endif
endwhile

→ S
```

The solution will be defined as a list of list of integer data type. On every row the solution is a list of integer where the unbroken line starts. For example, the solution set of the after generation is {0, 3, 5}. After generating the solution, then the solution is represented into an array for each row forming a matrix, such as in Figure 4.

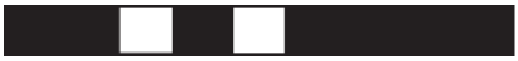


Figure 4. Example of a row

VI. Backtracking Algorithm in Solving Picross

This backtracking algorithm focuses heavily on creating a permutation based on the constraint created. To define the backtracking algorithm used in Picross Solver, the basic components of backtracking algorithm had to be defined.

A. Solution Set

To find the solution set to picross puzzles, let the solution for each row be a set of numbers where an unbroken line starts starting from 0 just like the solution for the exhaustive search method to help in creating the permutation. After generating the permutation, the solution is then represented in matrix (array of array of integer) to ease the solution checking.

B. Generation Function

The generation function's goal in this problem is to generate a sequence for each row, so that the constraint of the row will still be satisfied. The generation of sequence can be done using combinatorics or using another backtrack algorithm. The generation function is easier done using the feature of next permutation.

C. Bounding Function

Since the generation is done per row, the bounding function used in the solver is whether the current solution satisfies the column constraint by checking the content of the column. If the content of the column is in the column constraint, then the column satisfies the constraint.

The column constraint is checked by using the steps below

1. Count the length of unbroken lines in a column until it reached the end of the column or the last generated row
2. If the unbroken line counted is incomplete, ignore the unbroken line; else add the length to an array
3. Check whether the array (filled with the unbroken line length) is a subset of the respective column's column constraint. If it is, then the soluton satisfies the column constraint.

There is also a row constraint that must be checked, that is the spacing between lines. It can be achieved by the same method or just by checking the generated row at the start.

D. Pseudocode Implementation

Like the exhaustive search implementation, the class for the picross should be defined as in the pseudocode below

```
class Picross
  constraintRow : array of list of integer
  constraintCol : array of list of integer
  width : integer
  height : integer
  viewTable : array of array of integer
```

The backtracking algorithm for picross solver is implemented as a method in the Picross class and can be seen in the pseudocode below

```
function Solve(i:integer) → boolean

Variables:
r : array of array of integer

Algorithm:
if(i == height){
   → true
}
else{
   for (all permutation of i-row constraint)
      if(rowOK(c)andcolumnOK(c)andSolve(i+1))
         → true
      endif
   endfor
   clearRow(i)
   → false
endif
```

In the algorithm above, the recursive definition of the picross solver was written. The recursive function checks all the permutations of the row constraints. After generating a permutation, then the algorithm would check the row constraints and the column constraints.

The row constraint is checked by checking the space between unbroken lines. If there are more than one space or one space between all the unbroken lines then the row constraint is satisfied.

The column constraint is checked by checking all the column's unbroken lines. If the contents of the current column is the subset of the column constraint, then the column constraint is satisfied.

If all of the constraint is satisfied, then the next row would be checked, and invoked by using the recursive Solve(i+1) that indicates checking for the next row.

As for the basis, the recursive function would return true if i equals the value of height, indicating that all the row has been checked. Since the value of i increases in each recursive calls on a successful row check, i having the same value as height would mean that all the row have been checked.

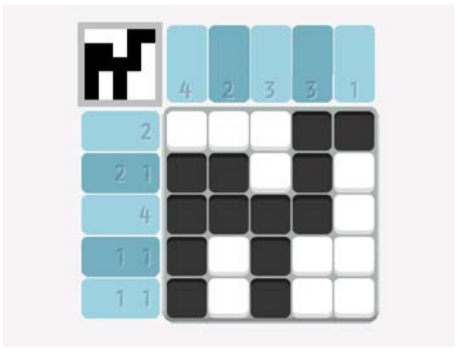
VII. Analysis of Algorithm

A. Difference between Backtrack and Exhaustive Search

The backtrack algorithm doesn't differ much from the exhaustive search, since backtrack is the optimized version of exhaustive search. However, in backtrack, the solutions are checked and accepted/rejected after generating the solutions for each row. Because of that, the backtrack algorithm gives a faster solve time than exhaustive search.

B. Implementation and Testing

For testing purpose of the algorithm, three picross puzzle, a 3x3 5x5 and 10x10 in the figure below is used as a testcase to measure the quality of the code.



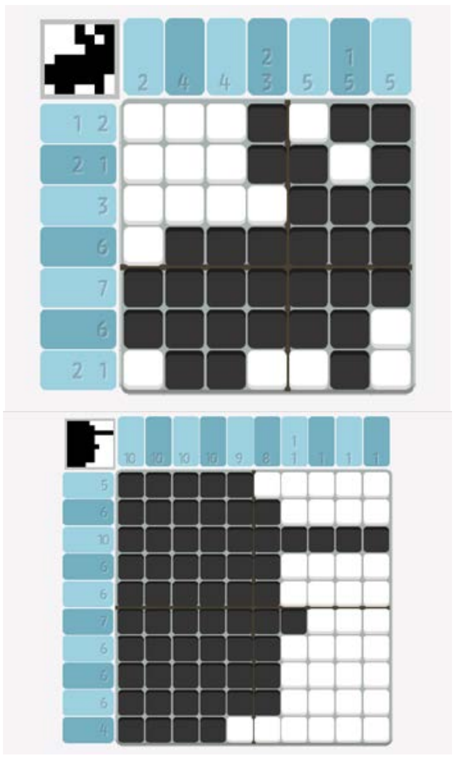


Figure 5. Sample Picross Puzzle
(Source :logicpicwalkthrough.blogspot.com)

Using the sample 5x5 and 7x7 and 10x10 puzzle above and the algorithm from the pseudocode defined in the previous section written in JAVA, the result below, which is the same as the solution is achieved.

```
   11
11 1
1111
1 1
1 1
Time taken :20ms
Branch Created :4384


   1 11
   11 1
    111
 111111
1111111
111111
 11  1
Time taken :2400ms
Branch Created :12718635
```

```
1111
111111
1111111111
111111
111111
1111111
111111
111111
111111
1111
Time taken :0ms
Branch Created :10
```

From the test cases above, it can be inferred that the amount of time used to solve the picross puzzle is still big. This happened because of the permutation used and the difficulty of the puzzle.

In general, the bigger the puzzle, the bigger the state space tree that will be generated. Solving 5x5 puzzle would generate a smaller state space tree than solving a 10x10 puzzle. Consequently, a larger puzzle would take a longer time to solve the puzzle. Note that the puzzle size is actually affected by the row size, since the permutation calculation only considers the row constraint.

The content of the row would also define the time taken to solve a picross puzzle. Rows with a lot of constraints would result in a higher number of permutations. Consequently, it increases the amount of time taken to solve the puzzle

Moreover, for puzzles whose ‘s permutations generated at the end would take a longer time to complete. Since the branch created would be a lot. This can be seen in the third example. Since the generation algorithm passes all the bounding function from the start, the branch passes all the test and thus resulting in a faster solve time despite the size of the puzzle. Failing all cases except the first, can result to the backtrack algorithm backtracks to the first state and makes waste of all the state that have been generated.

The time consumed can also be caused by the standart permutation algorithm in the library used that has a big complexity, thus contributes in the high amount of time to solve the puzzle.

Despite all of that, the backtracking algorithm is still usable in cases of small picross puzzles.

VIII. CONCLUSION

Backtracking is the optimized version of exhaustive search. However, there is still a slight possibility that the algorithm itself doesn’t differ much from exhaustive search.

In the example of this Picross Solver, it still takes a long time to produce the result despite having functions to reduce the generation of nodes because the algorithm depends on the final result of the puzzle. If the puzzle’s result is close to the starting permutation than the algorithm will perform well.

ACKNOWLEDGMENT

The author thanks to God for giving inspirations and time to make this writing, thus the author could finish this writing.

Other thanks are given to the author's friend for giving inspiration about the Picross game which we played during our times in high school.

REFERENCES

[1] Rinaldi Munir, Diktat Kuliah IF2211: Strategi Algoritma. Bandung: Program Studi Teknik Informatika Sekolah Teknik Elektro dan Inforrmatika Institut Teknologi Bandung. 2009

[2] http://stackoverflow.com/questions/2799078/permutation-algorithm-without-recursion-java (accessed at May, 18 2017)

[3] *logicpicwalkthrough.blogspot.com* (accessed at May, 18 2017)

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, 18 Mei 2017

[illegible]

Vincent Hendryanto Halim – 13515089