Combination of BFS and Brute Force Algorithm Implementation in Futoshiki Puzzle Game

Hendy Sutanto - 13507011

Informatics Engineering
School of Electrical and Informatics Engineering
Institut Teknologi Bandung
Jalan Ganeca 10
e-mail: hendy_lau8@yahoo.com

ABSTRACT

Brute-force is straightforward algorithm to solve problem in simple, to-the-point and obvious way.
BFS is a graph search algorithm that begins at the root node and explores all neighboring nodes.
Futoshiki is one of the puzzle games originated from Japan. Type of game is very similar to Sudoku. What makes it special are the "greater than" and "less than" signs among numbers.
This paper presents how to solve Futoshiki, one of paper based puzzle games, could be solved with combination of those two algorithms.
We use brute force to scan all square in futoshiki puzzle game, and BFS to update all possible numbers which some other square trigger.

Keywords: Futoshiki, Brute force, BFS, combination.

1. INTRODUCTION

Game is one kind of entertainment, it can also be challenging as well as to hone the player brain's ability. There are various types of games such as console games, board games, skill games, puzzle games, and others.

Lately, a lot of puzzle games that popular in community, one of them is the type of paper-based puzzle game. Paper-based puzzle game is a puzzle game can be played just with paper and pencil. As an easy and unboring game to play in recent years, the game Sudoku has been booming in the market, triggering the emergence of many other puzzle games that are not less interesting, like Futoshiki.

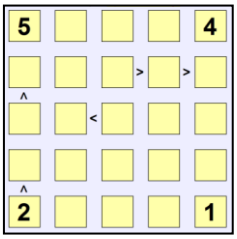


Figure 1. Futoshiki Puzzle Game

2. ALGORITHMS

2.1 Breadth First Search Algorithm

In graph theory, breadth-first search (BFS) is a graph-search algorithm that begins at the root node and explores all neighboring nodes. Then, for each of those nearest nodes, it explores their unexplored neighboring nodes, and so on, until it finds the goal or stucked (no more unexplored neighbours) without having found the goal.

It's basic scheme:

a. Enqueue the root node.
b. Dequeue a node and examine it.
   i. If the element sought is found in this node, quit the search and return a result.
   ii. Otherwise enqueue any successors (the direct neighboring nodes) that have not yet been discovered.
c. If the queue is empty, every node on the graph has been examined – quit the search and return "not found".
d. Repeat from Step b.

2.2 Brute Force Algorithm

Brute-force algorithm is one of a very simple algorithm, but no less powerful than any other algorithms. Although complexity is often relatively bigger than the others, it still becomes the favorite of many programmers in problem solving because it can solve all sorts of problems.

3. FUTOSHIKI

3.1 Introduction to Futoshiki

Futoshiki is one of the puzzle game originated from Japan. This game is very similar to Sudoku. What makes it special are the "greater than" and "less than" signs among numbers.

3.2 Futoshiki Rules

The aim of Futoshiki is to place the numbers 1 to 5 (or higher, if the puzzle is larger) into each row and column of the puzzle so that no number is repeated in a row or column and so that all of the inequality signs (< and >) are obeyed.

In order to finish a Futoshiki puzzle game, we need to know all possible numbers with respect to each square the same line, same column, and the inequality signs.

Generally, we write down the possible numbers for each square with pencil. Write possible numbers for each square with following order:

1) Square around filled square
2) Square that corresponds inequality signs
3) Other Squares

If exists a square that has only one possible number, then write down that number with pen (it means we already believe that the number is correct). After that, we remove that number from possible numbers from squares sharing same row or column.

4. IMPLEMENTATION

In Futoshiki Puzzle Game, one could use some kind of combinations from brute-force algorithm and BFS into the following algorithm:

```
procedure BFScan(output S : Square)

VARIABLE
x     : integer          {axis of square)
y     : integer          {ordinat of square)
size  : integer          {size of futoshiki board}
Found : boolean

ALGORITHM
x = 1
y = 1
_S = Square
Found = false

repeat
   ScanSquare(_S,x,y)
   if (IsSingleNum(_S))
      Found <- true
      S <- _S
   else if (IsSingleCand(_S,_S2))
      Found <- true
      S <- _S2
   x <- x + 1
   if (x > size)
      y <- y + 1
      x <- 1
until ((y > size) or Found)
```

```
procedure BFS(input S : Square)

VARIABLE
x : integer              {axis of square S)
y : integer              {ordinat of square S)
i : integer              {iterator for axis}
j : integer              {iterator for ordinat}
Q : queue of Square

ALGORITHM
x = Axis(S)
y = Ordinate(S)
i = 0
j = 0

repeat
   ScanSquare(_S,x,j)
   j <- j + 1
   if (PN(_S) == Info(S))
      Enqueue(_S,Q)
      UpdatePN(_S)
until (j > size)

repeat
   ScanSquare(_S,i,y)
   i <- i + 1
   if (PN(_S) == Info(S))
      Enqueue(_S,Q)
      UpdatePN(_S)
until (i > size)

repeat
   Dequeue(_S2,Q)
   if (IsSingleNum(_S2))
      WriteNumber(_S2)
      BFS(_S2)
   else if (IsSingleCand(_S2,_S3))
      WriteNumber(_S3)
      BFS(_S3)
until (IsEmpty(Q))
```

```
MAIN

VARIABLE
FS : Square {filled square}
IS : Square {square that correspond inequality sign}
ES : Square {empty squares}

ALGORITHM
generate possible numbers around FS
generate possible numbers in IS
generate possible numbers in ES
while (game is not finish)
   BFScan(ChangedSquare)
   WriteNumber(_S)
   BFS (_S)
   UpdatePN in IS
```

4.1 Variable Explanation

a. FS = filled square
   Filled square means square that is already filled as the game begins.
b. IS = square that corresponds inequality sign
c. ES = empty squares

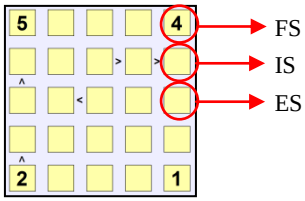


Figure 2. Variable Explanation

4.2 Function Explanation

a. IsSingleNum(S)
Boolean-returning function that checks if square S has only one possible number.

b. IsSingleCand(S)
Boolean-returning function that checks if there exist square having possible number that only appear in the same row or column with S (single candidate).

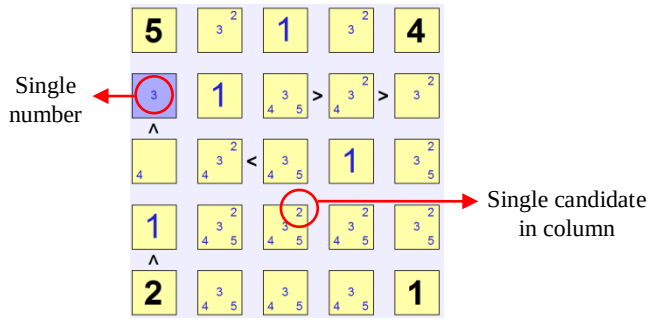


Figure 3. Function Explanation

c. IsEmpty(Q)
Boolean-returning function that checks if queue Q is empty.

d. Axis(S)
Get the x-axis from square S.

e. Ordinate(S)
Get the y-axis from square S.

4.3 Procedure Explanation

a. BFScan(S)
Scans all squares using bruteforce algorithm, sequentially scan every row and column to find single possible number or single candidate.

b. BFS (S)
Using BFS algorithm, find all squares that have been changed in same row or column with S, then check if there exist square having single possible number or single candidate triggered by S.

c. WriteNumber(S)
Writes number that we assuming is correct with pen instead just writing possible numbers with pencil.

d. UpdatePN(S)
Updates possible numbers.

e. ScanSquare(S,x,y)
Scans square S in point (x,y)

f. Enqueue(S,Q)
Inserts square S into queue of square Q

g. Dequeue(S,Q)
Moves head of queue of square Q into S and remove it from Q.

4.4 Example

a. generate possible numbers around S1
Write down all possible number in squares around filled square (up, right, down, left).

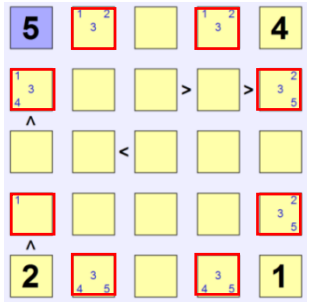


Figure 4. generate possible numbers around filled square

b. generate possible numbers in S2
Write down all possible numbers in squares that correspond inequality sign. Smallest possible number in "smaller" square should be not equal or greater than smallest possible number in the "bigger" square, as well as for the largest number.

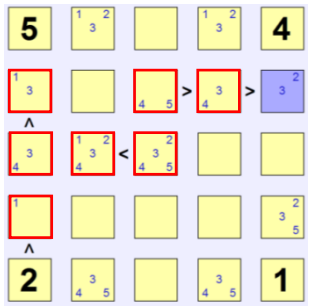


Figure 5. generate possible numbers in square that corresponds inequality sign

c. generate possible numbers in S3
Write all possible numbers in all remaining squares.

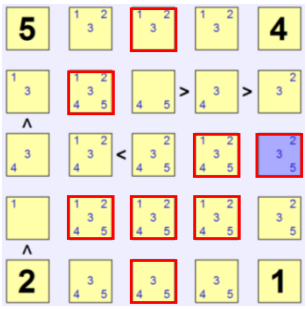


Figure 6. generate possible numbers in empty squares

d. while (game is not finish)
   While not all the squares are filled with the correct number, do something.
e. BFScan(ChangedSquare)
   Scans all squares thoroughly, starting from first row and first column sequentially to find single possible number or single candidate, and then outputs it to Changed-Square.

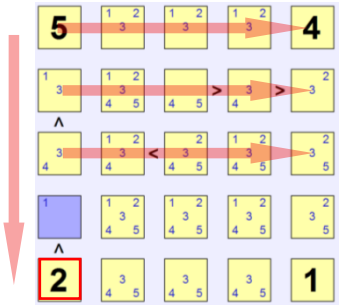


Figure 7. BFScan

f. WriteNumber(S)
   Writes correct number in square S.

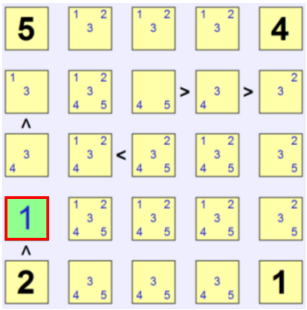
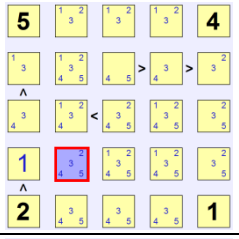
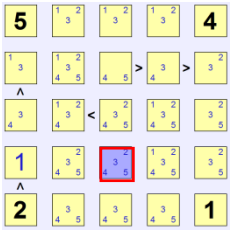
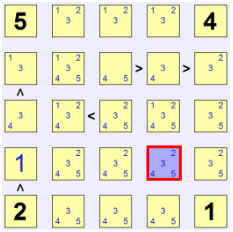
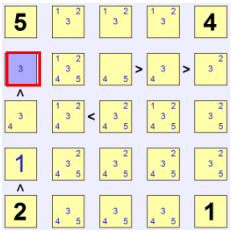


Figure 8. WriteNumber

g. BFS(_S)
   First, we scan squares sharing same row or column with square _S and check if its possible number equals to the correct number that we wrote before. If so, insert that square into Q then remove that possible number.

Table 1. BFS(_S)

Square	Queue
	4 2
	4 2 | 4 3
	4 2 | 4 3 | 4 4
	4 2 | 4 3 | 4 4 | 2 1

Next, we dequeue element in head of queue to check if it has single possible number or single candidate. If so, write that single possible number inside that square on that square, and do BFS on it again. This procedure is recursive.

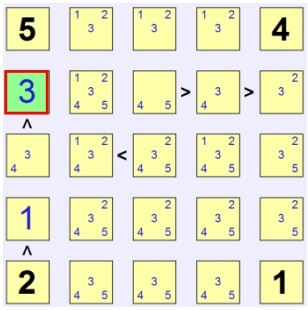
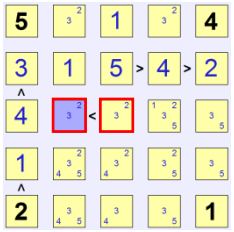
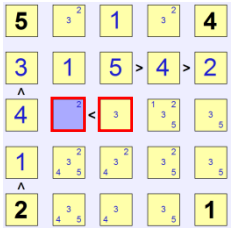


Figure 9. WriteNumber

h. UpdatePN in S2
   Updates possible number in square that corresponds inequality sign.

Table 2. UpdatePN in S2

Status	Square
Before updatePN	
After updatePN	

After all squares are filled correctly, the game is finished:

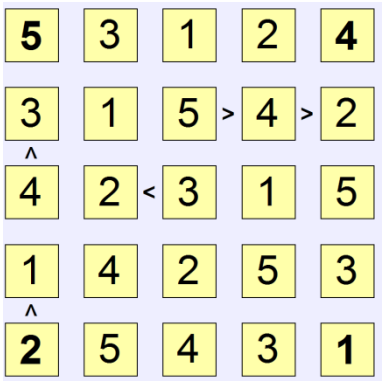


Figure 10. Finished Futoshiki

5. CONCLUSION

We already saw that we could use combination of brute force and BFS algorithm simultaneously in explanation above. This method is good for any size of futoshiki. As the size increases; the BFS algorithm will generate more nodes in state tree.

With brute force algorithm, we could solve any problems despite its large complexity. Also, we could try every single possible solution using BFS.

Futoshiki can be solved using elimination method of possible numbers in all squares. Starting with generating all possible numbers at the beginning of the game, updating them in the middle of game, and finally write it if it has only single possible number.

REFERENCE

[1] http://www.puzzlemix.com/rules-futoshiki.php?briefheader=1&JStoFront=1
Access time: December 30th, 2009 – 17:30

[2] http://www.puzzlemix.com/playgrid.php?id=5001&type=futoshiki
Access time: January 1st, 2010 – 1:39

[3] http://en.wikipedia.org/wiki/Breadth-first_search
Access time: January 1st, 2010 – 2:00

[4] http://en.wikipedia.org/wiki/Brute-force_search
Access time: January 1st, 2010 – 2:48

[5] Munir, Rinaldi, "Diktat Kuliah IF2251 Strategi Algoritmik", Program Studi Teknik Informatika, ITB, 2007.