KenKen Puzzle Solver using Backtracking Algorithm

Asanilta Fahda 13513079
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
asanilta@students.itb.ac.id

KenKen, also known as Mathdoku or Calcudoku, is a number puzzle game that requires a combination of basic arithmetic skills and logic to solve. It can be considered a more challenging variation of Sudoku or Kakuro. Each row and column of the grid must not contain the same digit, and each cage within the grid must fulfill its target number using the given type of operation (addition, subtraction, multiplication, or division). This paper presents an algorithm which can solve KenKen puzzles (tested up to 8×8 in size) using backtracking.

***Keywords*—puzzle, kenken, mathdoku, solver, backtracking**

I. Introduction to KenKen

A. Gameplay

KenKen, presently also known as Mathdoku or Calculdoku, was created in 2004 by a Japanese teacher named Tetsuya Miyamoto to fulfill his goal of training the math and logic skills of his students in a fun way. This brainteaser game quickly spread throughout Japan and the United States, replacing crossword puzzles in many newspapers. It then turned into a worldwide sensation after the appearance of online and mobile versions, especially appealing to lovers of number games such as Sudoku and Kakuro.[1]

In the game, the player is given a grid of size $n \times n$, with n commonly being an integer from 4 to 9. This grid must then be filled with the digits 1 through n in such a way that each row contains exactly one of each digit, each column contains exactly one of each digit, and each cage (a group of cells with a bold outline containing a target number and an operator) must fulfill the target result when the digits inside the cage are combined with the given operator. There are five different possible operators:

1. +, an n-nary operator indicating addition
2. -, a binary operator indicating subtraction
3. ×, an n-nary operator indicating multiplication
4. ÷, a binary operator indicating division
5. = (symbol usually omitted), a unary operator indicating equality

In some variations of the game, only the target number is given, and the player must guess the operators of each cage to solve the puzzle.

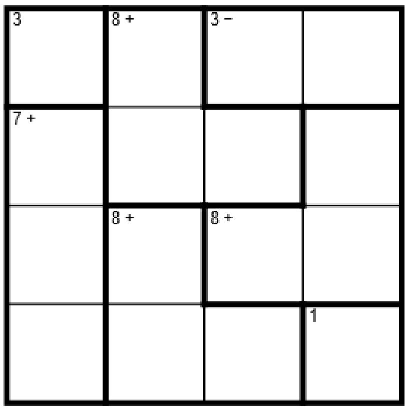

Fig 1.1. Example of an unsolved KenKen game [2]

B. Techniques and Strategies

In order to solve a KenKen puzzle, the player must first figure out two major problems: which numbers to put in a cage, and which order to put them in.

Like most other number games, the easiest manner to solve the puzzle is by means of elimination plus trial and error. The obvious start would be to find cages with only one square, because they do not produce any question of “which number” nor “which order”. For example, in the puzzle from Fig 1.1, the square at the left top corner and the one at the right bottom corner can be instantly filled in with their respective target number. We can then continue to find cages with only one possible combination of numbers (thus solving our “which numbers” problem), such as the cage in the top right corner with the rule stating “3-“. It can be easily concluded that the only pair of numbers from the set {1, 2, 3, 4} that will produce a result of 3 when one number is subtracted from the other is {1, 4}. Now, we can move on to the question of order. In this case, we are lucky because the right bottom corner has already been filled with the number 1, so any number in the same column cannot also be 1. Therefore, by elimination, we know that the top right corner must be 4 and the square on the left of it is 1. This, in turn, gives us the solution to the second square from the left in the top row, i.e. 2, because it is the only number not yet present in the entire row. We continue this process until all the cells in the grid are filled and produce a solution such as given below.

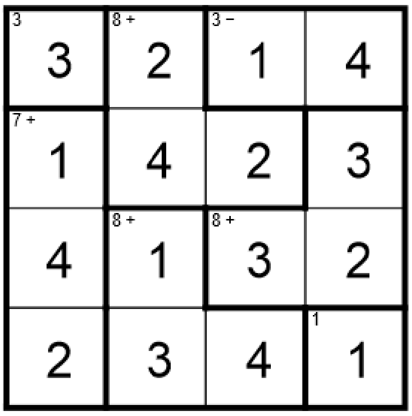

Fig 1.2. Solution for the KenKen game given in Fig 1.1 [2]

However, as the difficulty level progresses, the next move does not always appear so conspicuously. At times, the player is forced to make a guess and then later see whether or not the move ends up leading to a solution. If not, the player will have to "backtrack" and start over from the point of uncertainty.

II. Theory of Backtracking Algorithm

Backtracking is a general algorithm which finds a solution by trying one of several choices; if the choice proves incorrect, the computation restarts at the point of choice and tries another choice. [2] In order to "trace back our steps", it is necessary that we either: 1. explicitly keep track of all the steps taken, or 2. use recursion. The latter is used because it is by far easier, which is why backtracking is almost always DFS-based.[3]

The backtracking algorithm was first introduced by D.H. Lehmer in 1950 as an improvement to the brute-force algorithm. It was then developed further by R.J. Walker, Golomb, and Baumert. The algorithm proved to be effective for solving many logic games (e.g. tic-tac-toe, maze, chess, etc.) because it is notably useful for solving constraint satisfaction problems, in which a set of objects must satisfy a number of constraints or limitations.

The implementation of backtracking has the following common properties:

1. Solution space
   The solution of the problem is stated as a vector with n-tuple:

$$X = (x_1, x_2, ..., x_n), x_i \in S_i$$

   where it is possible that

$$S1 = S2 = ... = Sn$$

2. Generating function of x_k
   The generating function of x_k is stated as

$$T(k)$$

   where $T(k)$ generates the values of x_k, which are the components of the solution vector.
3. Bounding function
   The bounding function is stated as

$$B(x_1, x_2, ..., x_k)$$

   in which B is true if $(x_1, x_2, ..., x_k)$ leads to the solution.
   If B is true, the values of x_{k+1} continue on being generated, otherwise $(x_1, x_2, ..., x_k)$ is discarded.

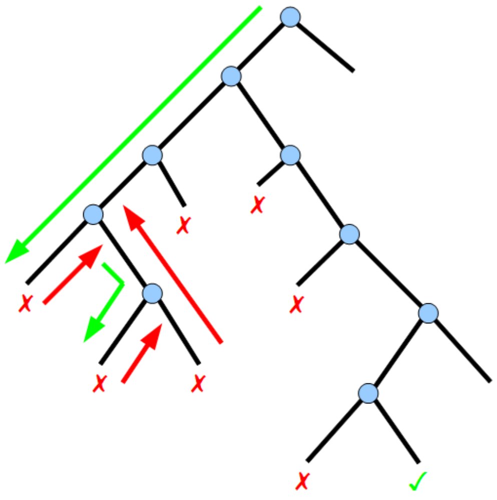

Fig 2.1. Illustration of state space tree used in backtracking algorithm [3]

The solution space for backtracking is organized in a tree structure, where each node represents the state of the problem and an edge is labeled as x_i. The path from the root to leaf represents a possible solution, and all the paths collected together form the solution space. This tree structure is called a state space tree.

The steps to using a state space tree for finding the solution are as following:

1. The solution is searched by building a path from the root to a leaf using depth-first order (DFS)
2. The nodes that are created are called live nodes
3. The nodes that are currently being expanded are called expand-nodes or E-nodes
4. Each time an E-node is being expanded, the path it is building becomes longer
5. If the path currently being built does not lead to the solution, the E-node is "killed" and becomes a dead node
6. The function used to kill off E-nodes is an implementation of the bounding function
7. Dead nodes will not be expanded
8. If the path being built ends with a dead node, the process backtracks to the node before it
9. It then continues to generate other child nodes, which in turn becomes the new E-nodes
10. The search ends when the goal node is achieved

Each node in the state space tree is associated with a recursive call. If the number of nodes in the tree is 2^n or $n!$, then the worst case for the backtracking algorithm is a time complexity of $O(p(n)2^n$ or $O(q(n)n!)$, with $p(n)$ and $q(n)$ as n-degree polynomials stating the computation time of each node.

III. Implementation of Backtracking Algorithm in KenKen Solver

A. Common Properties

1. Solution space

   The solution space of a KenKen puzzle of size $n \times n$ is:

   $X = (x_1, x_2, ..., x_m), x_i \in \{1, 2, ..., n\}$

   with $m = n^2$

2. Generating function

   The generating function generates an integer sequentially from 1 to n as x_k

3. Bounding function

   The bounding function combines three different constraint checking functions:

   - Column checking

     This function returns true if x_k is not yet present in the column or false otherwise

   - Row checking

     This function returns true if x_k is not yet present in the row or false otherwise

   - Grid checking

     This function checks the operator of the grid and performs a check accordingly:

Operator	Function
+	returns true if the sum of all the values present in the grid plus x_k is less than or equal to the target value; returns false otherwise
-	returns true if either both cells in the grid are empty or if there is one empty cell and the result of x_k subtracted by the value in the other cell, or vice versa, equals the target value; returns false
×	returns true if the multiplication result of all the values present in the grid times x_k is less than or equal to the target value; returns false otherwise
÷	returns true if either both cells in the grid are empty or if there is one empty cell and the result of x_k divided by the value in the other cell, or vice versa, equals the target value; returns false otherwise
=	returns true if x_k equals the target value; returns false otherwise

Table 1. Grid checking

B. State Space Tree

To illustrate the building of a dynamic state space tree during the process of solving a KenKen puzzle, we will use this following example of a 3×3 grid.

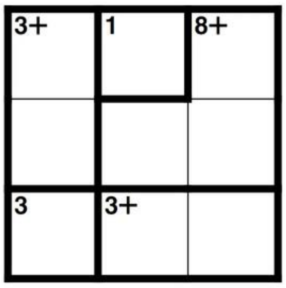

Fig 3.1. Example of a 3×3 KenKen puzzle [1]

We start by building state 1 which represents an empty grid. The generating function will then firstly generate the number 1 as x_1, meaning that it is placed in the first empty cell (state 2). The bounding function will check if this is a valid move, which it is. For the next empty cell, the number 1 is once again generated (state 3). However, it fails the row check in the bounding function and therefore becomes a dead node. We then try with number 2 (state 4), which fails the grid check in the bounding function because it does not equal the target value, i.e. 1. Number 3 (state 5) suffers the same fate.

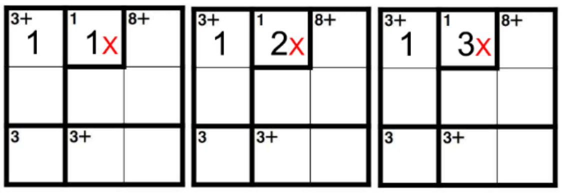

Fig 3.2 Illustration of state 3, 4, and 5 on a KenKen grid

Since there are no possible solutions this way, we backtrack to state number 1 and generate a new number as x_1. Number 2 is valid as x_1 (state 6) and hence we can continue to x_2. We try number 1 (state 7) and it satisfies the bounding function, meaning we can now continue to x_3. Both 1 (state 8) and 2 (state 9) fail the row check which means that we are left with number 3 (state 10). Because state 10 is valid, we can then create a new state with number 1 placed in the first column in the second row in the grid (state 11). This happens to fulfill the bounding check already, so we can continue to the cell beside it. Using number 1 (state 12) obviously doesn't work because it fails both the column check and the row check. Number 2 (state 13) seems to work, but none of its children (state 14, 15, 16) do so well afterwards (1 and 2 fail the row check while 3 fails the column check). We backtrack once again to state 11 and create a new state (state 17). Number 3 fits just fine as x_5, so we try number 1 (state 18) as x_6, which fails the row check, and continue with number 2 (state 19). This produces an almost-finished result.

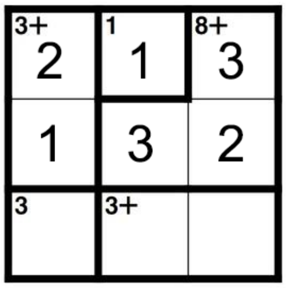

Fig 3.3 Illustration of state 19 on a KenKen grid

The last row can be deduced in a similar manner, and the result is achieved in state 25, as can be seen in the tree structure below.

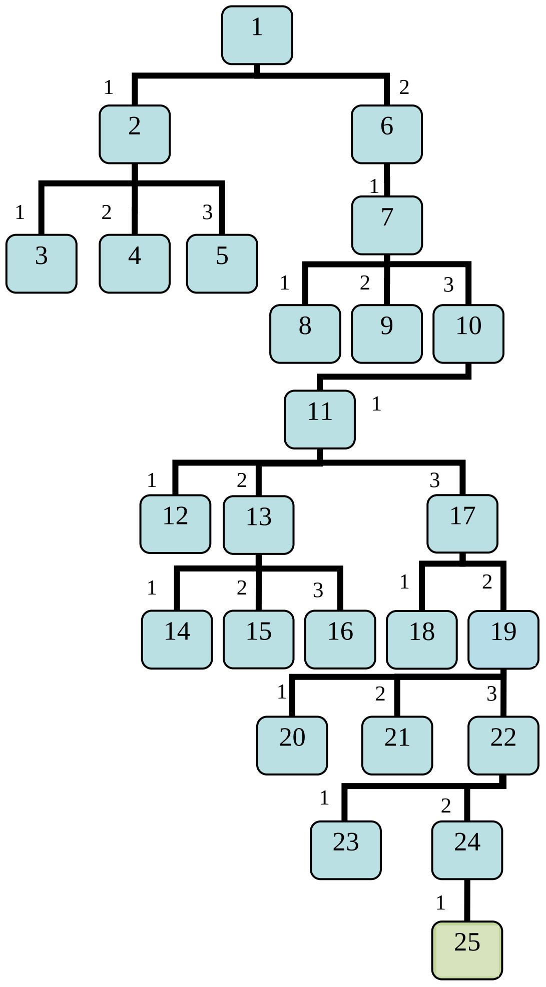
Fig 3.2. The state space tree built in the process of solving the puzzle in Fig 3.1.

The space tree above has reached its goal node, state 25, with a path of 2-1-3-1-3-2-3-2-1.

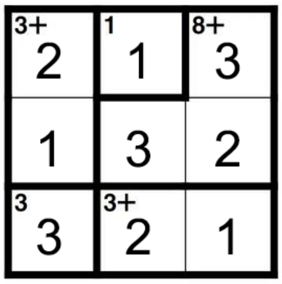

Fig 3.1. State 25 (the goal node) as the achieved result

The height of a tree built to solve a puzzle of size $n \times n$ should end up being of height n^2+1 once it reaches its goal node, with the path from the root node to the goal node representing all of the numbers used to fill in the grid from the top left cell to the bottom right cell.

C. Pseudocode of Implementation

The basic steps of the implementation can be described like this:

1. Find the first/next empty cell in the grid
2. Place a number starting from 1 to N in the cell until a valid number is found or until the number has exceeded N
3. If the number for the cell is valid, repeat step 1 and 2
4. If the number has exceeded N and no number from 1 to N is valid for the cell, backtrack to the previous cell and try the next possible valid number for that cell
5. If there are no more empty cells, the solution has been found

The solver is implemented as a class named KenKenSolver with the following methods and attributes:

```
class KenKenSolver

ATTRIBUTES
   class location
      x: int
      y: int

   class cage
      goal: int
      opr: char
      squares: list of location

   N: int
   NCage: int
   Board: array[1..N][1..N] of int
   CageBoard: array[1..N][1..N] of int
   Cages: array [1..NCage] of cage

METHODS
   initialize()
   readCageRules()
   printResult()
   solve(): boolean
   isValid(int, int, int): boolean
   checkRow(int, int): boolean
   checkCol(int, int): boolean
   checkCage(int, int): boolean
```

Pseudocode 1. KenKenSolver class

The `initialize` procedure is used to read an input with the following format:

```
3 5
1 2 3
1 3 3
4 5 5
3+ 1= 8+ 3= 3+
```

The first two integers on the top row indicate the size of the grid (N) and the number of cages (NCage) respectively. The following N rows (and N columns) indicate the cage numbers of each cell. The cage number for each cell is saved in the CageBoard matrix, and the locations for each cage number are added to the list of the corresponding cage number in the Cages array. The rules for each cage are given in the last row and are saved in the Cages array.

```
initialize()

KAMUS LOKAL
i: int
j: int
x: int

ALGORITMA
   input(N)
   input(NCages)
   i traversal [1..N]
      j traversal [1..N]
         input(x)
         Board[i][j] = 0
         CageBoard[i][j] = x
         location cell(i,j)
         Cages[x].squares.add(cell)

   readCageRules()
```

Pseudocode 2. initialize procedure

The bounding function is given by the `isValid` function, which in turn calls `checkRow`, `checkColumn`, and `checkGrid`.

```
function isValid(input row: int, input
col: int, input num: int) -> boolean

ALGORITMA
   -> (checkRow(row,num) and
      checkCol(col,num) and
      checkGrid(CageBoard[row][col],num))
```

Pseudocode 3. isValid function

```
function checkRow(input row: int, input
num: int) -> boolean

KAMUS LOKAL
col: int

ALGORITMA
   col traversal [1..N]
      if (Board[row][col]==num)
         -> false

   -> true
```

Pseudocode 4. checkRow function

```
function checkCol(input col: int, input
num: int) -> boolean

KAMUS LOKAL
row: int

ALGORITMA
   row traversal [1..N]
      if (Board[row][col]==num)
         -> false

   -> true
```

Pseudocode 5. checkCol function

```
function checkGrid(input c: int, input num:
int) -> boolean

KAMUS LOKAL
total: int

ALGORITMA
   if (Cages[c].opr==’+’)
      total = num
      for each loc in Cages[c].squares
         total += Board[loc.x][loc.y]
      if (total <= Cages[c].goal)
         -> true
   else if (Cages[c].opr==’-‘)
      loc1 = Cages[c].squares[0]
      loc2 = Cages[c].squares[1]
      num1 = Board[loc1.x][loc1.y]
      num2 = Board[loc2.x][loc2.y]
      if (num1==0)
         if (num2==0)
            -> true
         else if(num2-num==Cages[c].goal)
            -> true
         else if(num-num2==Cages[c].goal)
            -> true
      else if (num1-num==Cages[c].goal)
         -> true
      else if (num-num1==Cages[c].goal)
         -> true
   else if (Cages[c].opr==’x’)
      total = num
      for each loc in Cages[c].squares
         total *= Board[loc.x][loc.y]
      if (total <= Cages[c].goal)
         -> true
   else if (Cages[c].opr==’/’)
      loc1 = Cages[c].squares[0]
      loc2 = Cages[c].squares[1]
      num1 = Board[loc1.x][loc1.y]
      num2 = Board[loc2.x][loc2.y]
      if (num1==0)
         if (num2==0)
            -> true
         else if(num2/num==Cages[c].goal)
            -> true
         else if(num/num2==Cages[c].goal)
            -> true
      else if (num1/num==Cages[c].goal)
         -> true
      else if (num/num1==Cages[c].goal)
         -> true
   else if (Cages[c].opr==’=’)
      if (num==Cages[c].goal)
         -> true

   -> false
```

Pseudocode 6. checkGrid function

To check for the next empty square to fill (also to check if there are no more empty squares, meaning the puzzle is solved), we have a function called `findEmpty`.

```
function findEmpty() -> location

KAMUS LOKAL
i: integer
j: integer

ALGORITMA
   i traversal [1..N]
      j traversal [1..N]
         if (Board[i][j]==0)
            -> (i,j)

   -> null
```

Pseudocode 7. findEmpty function

The functions given above are used in the function `solve`, which is a recursive function that returns true if the current state is the solution, or if the current state can be expanded to achieve the solution.

```
function solve() -> boolean

KAMUS LOKAL
empty: location

ALGORITMA
   empty = findEmpty()
   if (empty != null)
      i traversal [1..N]
         if isValid(empty.x,empty.y,i)
            Board[empty.x][empty.y]=i
               if (solve())
                  -> true
            Board[empty.x][empty.y]=0

      -> false
   else
      -> true
```

Pseudocode 8. solve function

Finally, we can run the program with a simple driver:

```
main()
   K: KenKenSolver
   K.initialize()
   if (K.solve())
      K.printGrid()
   else
      output("Oops!")
```

Pseudocode 9. Program driver

IV. Results and Performance

a. 4×4 grid

Input:

```
4 9
1 2 3 3
1 4 4 5
6 7 7 5
8 8 9 9
7+ 2= 2- 3- 2/ 1= 6x 3+ 7+
```

Output:

```
4 2 3 1
3 4 1 2
1 3 2 4
2 1 4 3
```

Time: 68 ms

b. 5×5 grid

Input:

```
5 11
1  1  2  2  3
4  5  5  3  3
4  6  7  7  8
4  6  9  7  8
10 10 9  9  11
2- 2/ 9+ 24x 4- 2/ 48x 4- 75x 3-
```

Output:

```
5 3 2 1 4
4 1 5 2 3
2 5 4 3 1
3 2 1 4 5
1 4 3 5 2
```

Time: 96ms

c. 6×6 grid

Input:

```
6 16
1  2  3  4  4  5
6  2  3  4  7  5
6  2  8  8  7  7
6  9  9  10 11 11
12 9  13 13 14 14
12 15 13 16 16 14
1= 11+ 11+ 12+ 3+ 11+ 10+ 6+ 10+
6= 1- 3- 8+ 13+ 1= 7+
```

Output:

```
1 3 6 5 4 2
4 2 5 3 6 1
5 6 2 4 1 3
2 5 1 6 3 4
6 4 3 1 2 5
3 1 4 2 5 6
```

Time: 171 ms

d. 7×7 grid

Input:

```
7 22
1  2  2  3  4  4  5
1  6  6  3  7  8  5
9  6  10 10 7  8  8
9  11 12 12 13 14 14
11 11 15 12 13 16 16
17 18 18 19 19 16 20
17 17 21 21 22 22 22
2- 13+ 3- 6x 2/ 14+ 6- 150x 6- 1-
10+ 36x 1- 1- 1= 16+ 24x 8+ 3- 6=
35x 8+
```

Output:

```
2 1 6 4 3 7 5
4 5 3 1 7 6 2
7 6 4 5 1 2 3
1 2 7 6 5 3 4
3 4 1 2 6 5 7
5 3 2 7 4 1 6
6 7 5 3 2 4 1
```

Time: 823 ms

e. 8×8 grid

Input:

```
8 26
1  2  2  3  4  4  5  5
1  6  6  7  8  9  5  5
10 10 6  7  7  9  9  11
12 13 14 14 15 16 11 11
12 13 17 17 15 16 18 18
19 19 20 21 21 22 22 18
23 20 20 21 22 22 24 24
23 25 25 21 26 26 24 24
7+ 7- 4= 2/ 630x 14+ 2x 7= 72x
15+ 60x 3/ 2- 2- 3- 1- 1- 14+ 35x
330x 19+ 224x 24x 18+ 3/ 11+
```

Output:

```
1 2 6 3 5 4 7 8
5 3 7 6 8 1 4 2
2 4 1 7 3 8 5 6
3 8 5 2 6 7 1 4
6 7 4 5 2 3 8 1
8 1 2 4 7 6 3 5
4 5 3 8 1 2 6 7
7 6 8 1 4 5 2 3
```

Time: 100314 ms

V. Analysis and Conclusion

In conclusion, backtracking is a versatile algorithm which can be used for solving many types of puzzle games, including KenKen, in a systematic and structured manner. The algorithm can compute the solution for small to medium-sized KenKen puzzles (up to 7×7 grid size) quickly within seconds. However, as the size increases, the execution time also experiences a great difference. 8×8 grids vary to a great extent in time depending on the puzzle itself (between ~1.5 minutes to ~20 minutes). This can be improved by implementing better heuristics that can eliminate more choices or by using a different type of algorithm altogether.

References

[1] KenKen Puzzle Official Site. http://www.kenkenpuzzle.com/ [Accessed: May 4th, 2015]

[2] Davis, Tom. "Kenken for Teachers". http://mathcircle.berkeley.edu/archivedocs/2009_2010/lectures/0910lecturespdf/kenken_tom_davis.pdf [Accessed: May 4th, 2015]

[3] Munir, Rinaldi. *Diktat Kuliah IF2211 Strategi Algoritma*. Bandung: Teknik Informatika Institut Teknologi Bandung, 2009.

[4] GeeksforGeeks "Backtracking | Set 7 (Sudoku)". http://www.geeksforgeeks.org/backtracking-set-7-suduku/ [Accessed: May 3rd, 2015]

Statement

I hereby declare that this paper is my own work and not a copy, translation, nor plagiarism of somebody else's work.

Bandung, May 5th 2015

[illegible]

Asanilta Fahda 13513079