Solving KenKen puzzle using Brute Force

Kahfi Soobhan Zulkifli 13519012
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13519012@std.stei.itb.ac.id

***Abstract*—KenKen, also known as Mathdoku or Calcudoku, is a number puzzle game that is like sudoku but has additional challenges to it. The player must also use its arithmetic skills when solving this puzzle, different to sudoku where it does not require arithmetic skills. Like Sudoku, each row and column of the grid can only have one unique digit, and each cage within the grid must fulfill its its given rule, defined by an operator (addition, subtraction, multiplication, and division) and a final number. This paper presents a brute force algorithmic approach.**

***Keywords*—kenken; algorithm; brute; force; strategy**

I. KENKEN PUZZLE

KenKen (Mathdoku or Calculdoku) was created in 2004 by Tetsuya Miyamoto, a Japanese teacher who wanted to train the math and logic skills of his students in an interactive and challenging method. This brainteaser game quickly spread throughout the world, currently one of the most famous online puzzle ever played on the Internet. 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].

The game consists of a grid with size nxn, n ranging from 3 to 9 and cages which consist of many grids that has a specified rule. The puzzle is completed when each row and each column consists all of the digits ranging from 1 to n and the numbers in each cage satisfies the rule applied to it. The rule consists of an operator and a target number. Mathdoku allows five different operators:

1. +, addition (n-ary)

2. -, subtraction (binary)

3. ×, multiplication (n-ary)

4. ÷, division (binary)

5. =, equality (unary, symbol usually not shown, just the number given)

To solve a KenKen puzzle, there are two challenges in which must be thought carefully by the player: which numbers to put in a cage, and which order to put them in since it affects other rows and columns. The most obvious strategy is to use trial and error with some added intuition that is subjective to the player. The easiest start would be to fill in cages that consist of one grid because it is obvious the number required by the grid since there is only one operator that can be used in a one-grid cage which is the '='unary operator. For example, in the puzzle from Fig 1.1, the number 3 is the target number required at the square at the first row since the rule written is (=)3. After that, we can continue to solve more cages, starting by small cages progressing to bigger ones, or by using trial and error. For example, shown by the cage with the rule '24x', we can clearly state that the only numbers that satisfy this rule is 4 and 6 from the set of numbers {1, 2, 3, 4, 5, 6}. The question is, in what order should we put these two numbers inside the cage.[5]

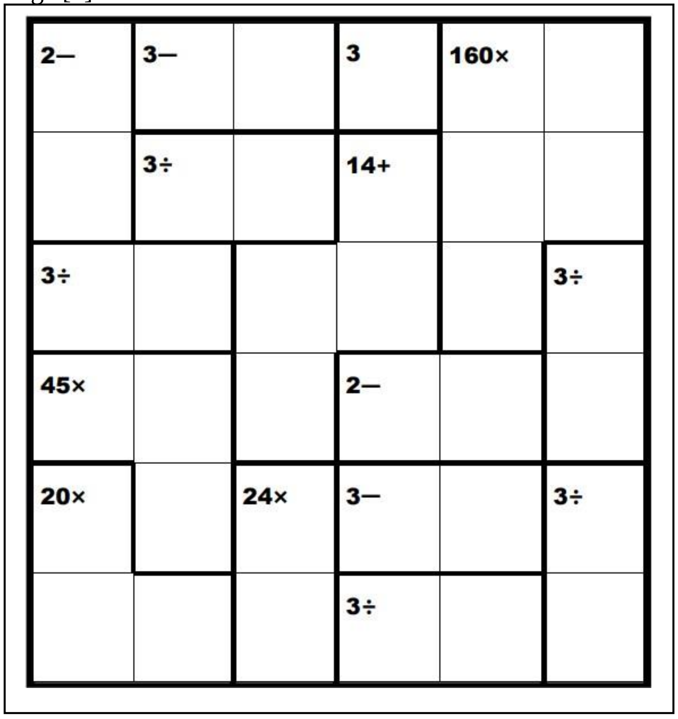


Fig. 1. Kenken Puzzle

From this, we can start using trial and error with some added knowledge gained from the previous findings, and this process repeats until all the grids are filled in with numbers. As the difficulty level progresses, the next move cannot be predicted easily. Sometimes the player is forced to guess a number if the cage can be filled with any combination of numbers given in the set, and then 'backtrack' to the point where it was uncertain. Other strategies could be by using additional heuritics such as size of cage or number of possible 'answers' in a cage.

II. Brute Force Algorithm

Brute force algorithm is a straightforward approach in solving a problem. This is usually based on the problem statement itself and the concepts that are involved within the problem. This algorithm solves a problem in the most obvious way and easy to be implemented.[2]

This algorithm is not intelligent and inefficient, since it requires a lot of steps in solving a problem. Sometimes this algorithm is also called the naive algorithm since it does not rely on the "brain", more on the "power." [3]

In cryptography, attackers normally use brute force attacks where they try every combination possible in guessing a user's password. This may take years depending on the strength of the password and how intelligent the the attacks are, usually by guessing whole dictionary words then moving on to random generated passwords. [4]

Brute force algorithms are used for small-scale problems since these problems are considered simple and easy to be solved. They are also used as a reference for more efficient algorithms.

Despite the algorithm's inefficiency, this algorithm can solve any problem since it searches for the solution. There are problems where it is impossible to use algorithms other than brute force, for instance finding the biggest number in an array. [6]

Here is the pseudocode for the global brute force algorithm.

```
Brute force:
c: candidate solution
P: set of candidates
sol: solution candidate
c ← first(P)
while c ≠ sol do
  if isValid(c) then
    output(c)
  c ← next(P)
```

III. Implementation of Brute Force Algorithm in Kenken Puzzle

For this problem, we start off by considering the board of the kenken puzzle as a matrix with size of NxN where N is the number of the grids. This means that the candidate solution should be in the form of a matrix, and the candidates are the result of a permutation generator. The permutation generator starts off by creating a base tuple starting from 1 to the size of the permutations. After that, we need to count the maximum number of iterations required in creating the list of candidates. Next, using an array, we start off by looping through the range from the size given until the last number. During this loop, we continuously update the indices by swapping the i-th one with the j-th one from the right from the cycles array (in the program, it is called 'perulangan'). If one of the elements in this array is 0, then it is handled differently, by placing the i-th item at the end, which shifts all the other elements to the front of the array. From here we get the candidates and the order in which these candidates must be placed into the board.

This algorithm guarantees that each row is already unique, the real task is making sure that each column in the board has unique numbers as well. To do this, we use a checkUnique function where for every column, the elements are stored in a set so that there are no duplicate values, and then the lenght of the set is checked so that it is the same as the length of the column.

Next, the algorithm ensures that the solution already satisfies the rules given by the player. This starts off by the player creating the rules and the algorithm stores it in a dictionary where the key is the number of the cage and the value of the key is the given rule. The locations of the cages are also stored in a cagePos array, so that it is easier to check whether the solution provided already satisfies the rules.

The checkCage algorithm starts off by looping through all of the cages and its respective operators. From here, the algorithm branches off according to the operator it detects.

"+": Sums all of the numbers provided in the board and ensures whether it is the same or different to the target number.

"-": Computes the difference between two numbers and checks whether it is equal to the target number.

"x": All the numbers in the cells given are multiplied and the result is compared to the target number.

"/": Divides the two numbers provided in the board and the result is compared to the target number.

Regarding the input of the program, the user types in the size of the board, the number of cages and the locations of the cages in the board. After that, the user types in the rules of the board.

Here is the code of the program.

```
##Main.py
from engine import *
from mathdoku import *
from math import *
from itertools import *



grid, rules_dict, board, cagePos, N, NCages = start()
brutes = list(perms(N)) #engine
order = list(permutations(range(len(brutes)), N)) #itertools


solved = False


import time
start_program_time = time.time()


index = 0
while not solved and index < len(order):
    ordering = order[index]
    i = 0
    for rowindex in range(N):
        board[rowindex] = brutes[ordering[i]]
        i = i+1


    solved = checkUnique(N, board) and checkCage(NCages, board, rules_dict, cagePos)
    index += 1


for row in board:
    print(row)


print(str(time.time() - start_program_time)+" seconds")
```

```
##Mathdoku.py
def start():
    N = int(input("NxN: "))
    NCages = int(input("Number of cages: "))

    grid = []
    for i in range(N):
        nums = input().split(' ')
        nums = [int(x) for x in nums]
        grid.append(nums)

    rules = input().split(' ')
    if (len(rules) == NCages):
        rules_dict = {}
        i = 1
        for rule in rules:
            rules_dict.update({i : rule})
            i += 1

    cagePos = [[] for i in range (NCages)]
    for i in range(N):
        for j in range(N):
            index = grid[i][j]
            cagePos[index-1].append((i,j))

    board = [[0 for j in range(N)] for i in range(N)]

    return grid, rules_dict, board, cagePos, N, NCages

def checkUnique(N, board):

    for col in range(N):
        a = []
        for row in range(N):
            a.append(board[row][col])
        col_length = len(set(a))
        if col_length != N:
            return False
```

```
##Engine.py
def perms(size):
  dasar = tuple(range(1, size+1))
  length = size

  maxnumberofiter = 0
  for i in range(1,size+1):
    maxnumberofiter *= i

  arrayofnum = []
  for i in range(length):
    arrayofnum.append(i)

  perulangan = []
  for i in range(length, length-size, -1):
    perulangan.append(i)

  yield list(dasar[i] for i in arrayofnum[:size])

  count = 0
  while True or count <= maxnumberofiter:
    for i in range(size-1, -1, -1):
      perulangan[i] -= 1
      count += 1
      if perulangan[i] == 0:
  arrayofnum[i:]=arrayofnum[i+1:]+arrayofnum[i:i+1]
        perulangan[i] = length - i
      else:
        j = perulangan[i]
        swap(arrayofnum, i, -j)
        yield list(dasar[i] for i in arrayofnum[:size])
        break
    else:
```

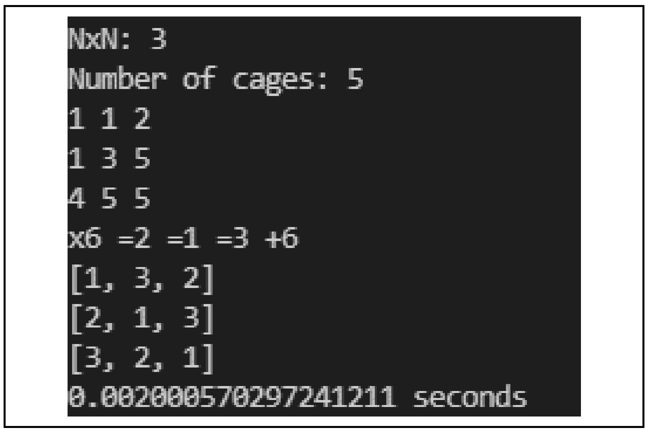
.

```
##Mathdoku.py
def checkCage(NCages: int, board, rules_dict: dict,
cagePos)
    for cageNum in range(NCages):
      if (rules_dict[cageNum+1][0] == "+"):
        total = 0
        for loc in cagePos[cageNum]:
          total += board[loc[0]][loc[1]]
        if (total != int(rules_dict[cageNum+1][1:])):
          return False
          break
      elif (rules_dict[cageNum+1][0]=="-"):
        loc1 = cagePos[cageNum][0]
        loc2 = cagePos[cageNum][1]
        num1 = board[loc1[0]][loc1[1]]
        num2 = board[loc2[0]][loc2[1]]
if(abs(num1-num2)!=int(rules_dict[cageNum+1][1:])):
          return False
          break
      elif (rules_dict[cageNum+1][0]=="x"):
        total = 1
        for loc in cagePos[cageNum]:
          total *= board[loc[0]][loc[1]]
        if total != int(rules_dict[cageNum+1][1:]):
          return False
          break
      elif (rules_dict[cageNum+1][0]=="/"):
        loc1 = cagePos[cageNum][0]
        loc2 = cagePos[cageNum][1]
        num1 = board[loc1[0]][loc1[1]]
        num2 = board[loc2[0]][loc2[1]]
        if (num1 > num2):
     if(num1/num2!=int(rules_dict[cageNum+1][1:]):
            return False
            break
        else:
     if num2/num1!=int(rules_dict[cageNum+1][1:]):
            return False
            break
```

```
elif (rules_dict[cageNum+1][0]=="="):
            loc = cagePos[cageNum][0]
            if board[loc[0]][loc[1]] !=
int(rules_dict[cageNum+1][1:]):
               return False
               break
      return True
```

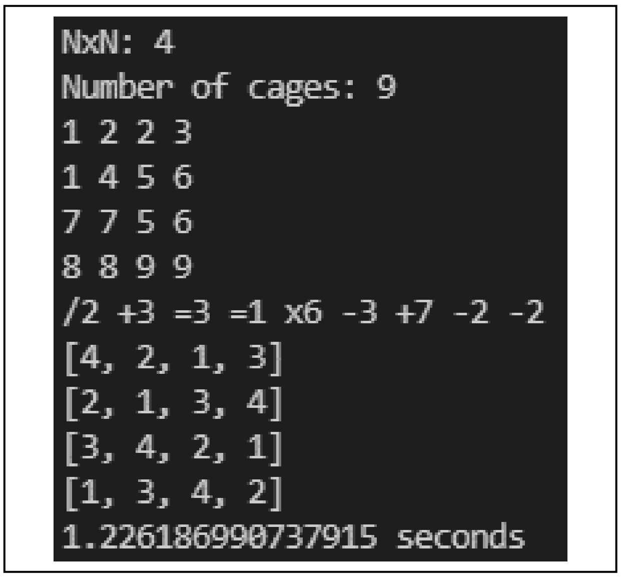
IV. Testing

We use two test cases, one with size 3x3 and the other size 4x4. Here are the results.



```
NxN: 3
Number of cages: 5
1 1 2
1 3 5
4 5 5
x6 =2 =1 =3 +6
[1, 3, 2]
[2, 1, 3]
[3, 2, 1]
0.002000570297241211 seconds
```

Fig. 2. Test Case 3x3



```
NxN: 4
Number of cages: 9
1 2 2 3
1 4 5 6
7 7 5 6
8 8 9 9
/2 +3 =3 =1 x6 -3 +7 -2 -2
[4, 2, 1, 3]
[2, 1, 3, 4]
[3, 4, 2, 1]
[1, 3, 4, 2]
1.226186990737915 seconds
```

Fig. 3. Test Case 4x4

V. Analysis

Based on the algorithm and testing, this algorithm cannot accept kenken puzzles with sizes of 5x5 or more due to the algorithm's inefficiency when handling large inputs. This is proven by the fact that the complexity of this algorithm is O(n!) since the length of the order list in main.py has length n!.

Furthermore, this is proven by the fact that the runtime of Fig. 3 is around 375% longer than the runtime of Fig 2. This indicates that a slight increase in the size of the grid causes a more significant increase in the time complexity, thus proving the fact that the algorithm has a time complexity of O(n!).

VI. Conclusion

In conclusion, the Kenken puzzle is a challenging problem that was created by a Japanese teacher and then became more popular around the world. This puzzle can be solved by many algorithms, one of which is brute force algorithm. However, the grid size of the puzzle cannot exceed 4x4. This is due to the inefficiency of the algorithm with a time complexity of O(n!) which is one of the most inefficient algorithms. We recommend using backtracking algorithm for larger grids, since it is clearly more efficient than brute force algorithm.[2]

Video Link At Youtube

https://youtu.be/hq3BnFQssDE

Acknowledgment

Special thanks to my lecturer Dr. Ir. Rila Mandala, M.Eng. for guiding me and his class, K1 during this semester. Also to the other lecturers such as Dr.Ir. Rinaldi Munir, S.T., Dr. Nur Ulfa Maulidevi, S.T., M.Sc. and Prof. Dwi Hendratmo W., Ph.D. for providing us the lecture materials in videos and in powerpoint slides. Also the laboratory assistants of IF2211 Algorithm Strategies who have challenged us with their small and big projects over the course of this semester. Finally to my fellow classmates who have struggled together during these hard times especially during this pandemic.

References

[1] KenKen Puzzle Offical Site. http://www.kenkenpuzzle.com/ [Accessed: May 11th 2021]

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

[3] FreeCodeCamp. https://www.freecodecamp.org/news/brute-force-algorithms-explained/ [Accessed: May 11th 2021]

[4] Blocking Brute Force Attacks https://web.archive.org/web/20161203020306/http://www.cs.virginia.edu/~csadmin/gen_support/brute_force.php [Accessed: May 11th 2021]

[5] Fahda. Asanilta, "KenKen Puzzle Solver using Backtracking Algorithm," Bandung: Teknik Informatika Institut Teknologi Bandung, 2015

[6] Levitin, Anathy. Introduction to The Design and Analysis of Algorithms 3rd Edition. United States of America: Pearson, 2012

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 11th 2021

[illegible]

Kahfi Soobhan Zulkifl