Perbandingan Algoritma *Brute Force* dan Algoritma *Backtracking* dalam Menyelesaikan Permainan Sudoku

Yan Arie Motinggo 13518129
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail):

Abstract*—Sudoku adalah sebuah game tradisional yang dimainkan dengan memenuhi angka pada kotak yang disediakan. Pada masanya sudoku adalah game yang sangat popular di dunia, yang dapat diakses dari berbagai platform mulai dari internet, gawai, koran, bahkan buku. Banyak dari platform tersebut yang melakukan kompetisi untuk menyelesaikan sudoku. Pada makalah ini penulis akan membahas tentang perbandingan antara algoritma *Brute Force* dan algoritma backtrackting (runut balik) dalam kecepatan dan keefisienan kedua algoritma tersebut dalam menyelesaikan permasalahan sudoku, Algoritma *Brute Force* adalah algoritma yang membangkitkan seluruh kemungkinan yang mungkin dengan cara menempatkan angka pada papan kemudian dari seluruh kemungkinan penempatan angka itu akan dicari himpunan-himpunan penempatan angka mana yang akan memenuhi *constraint* permainan sudoku sedangkan, Algoritma backtrackting adalah algoritma berbasis DFS (*depth search first*) untuk menyelesaikan permsalahan secara efisien dibandingkan *Brute Force*, secara sistematis menemukan solusi persoalan diantara kemungkinan solusi yang ada, hanya saya algoritma ini hanya melakukan pencarian yang mengarahkan solusi yang sedang dipertimbangkan saja. Makalah ini bertujuan untuk menjelaskan perbedaan antara kedua algoritma dalam menyelesaikan permasalahan soduko, dalam performa, Langkah pengerjaan, kerangka berpikir, hingga keefisienan algoritma. Dari hasil tersebut algoritma *Backtracking* merupakan algoritma yang lebih baik dibandinkan algoritma a*Brute Force* baik dalam performa, kecepatan penyelesaian, dan keefisienan.

Keywords—Sudoku, Backtracking, Brute Force, DFS.

I. PENDAHULUAN

Sudoku adalah puzzle logic yang berdasar pada konsep kombinatorial, yang tujuannnya adalah untuk mengisikan angka-angka dari 1 sampai 9 ke dalam jaring-jaring 9×9 yang terdiri dari 9 kotak 3×3 tanpa ada angka yang berulang di satu baris, kolom atau kotak. Pertama kali diterbitkan di sebuah surat kabar Prancis pada 1895 dan mungkin dipengaruhi oleh matematikawan Swiss Leonhard Euler, yang membuat terkenal Latin square.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Gambar 2; Sudoku 9x9 dengan daerah 3x3

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Gambar 1; Sudoku 9x9 dengan daerah 3x3 (solved)

Sudoku berasal dari bahasa jepang "*suuji wa dokushin ni kagiru*" yang artinya "angka-angkanya harus tetap tunggal", meskipun sudoku 9x9 dengan daerah 3x3 adalah sudoku yang paling umum digunakan terdapat beberapa variasi lain yang tersedia seperti sudoku 4x4 dengan darah 2x2, terdapat juga sudoku dengan 5x5 dengan daerah *pentomino* (polygon datar yang terbuat dari 5 persegi dengan ukuran yang sama yang

saling berhubungan sisi ke sisi) dan masih banyak jenis yang lain.

Gambar 3; Sudoku dengan daerah nonomino

Permasalahan puzzle Sudoku sulit untuk dipecahkan karena masuk dalam permasalahan NP-complete, sehingga tidak bisa diselesaikan dalam waktu yang sama. Hingga saat ini banyak programmer yang mencari algoritma yang tepat untuk menyelesaikan puzzle ini. Cara yang paling gampang adalah algoritma *Brute Force* yaitu dengan cara mengenumerasikan semua kemungkinan isi sel dengan angka 1 sampai 9(1) . Tetapi cara ini tentu saja tidak tepat karena kemungkinannya akan sangat banyak sekali. Karena itu algoritma ini diperbaiki dengan menambahkan batasan (constraints), yaitu tidak boleh ada angka yang sama dalam satu baris, kolom atau subgrid. Cara ini bisa mereduksi jumlah kemungkinan secara signifikan sehingga algoritma menjadi lebih tepat.berikut merupakan peraturan dalam permainan sudoku (gambar 1);

1. Sebuah angka hanya boleh muncul satu kali dalam setiap baris
2. Setiap angka hanya boleh muncul satu kali setiap kolom
3. Setiap angka hanya boleh muncul sati kali pada daerah 3x3 dari sudoku

Terdapat beberapa algoritma yang dapat menyelesaikan permasalahan ini, Bahasa yang berbeda dan konsep desain yang berbeda juga. Pada makalah hanya akan dibahas dua algoritma saja yaitu; algoritma *Brute Force* dan algoritma *Backtracking*, Sudoku merupakan puzzle yang dapat dikatakan mudah untuk diselesikan tetapi terkadang memiliki algoritma yang kompleks, tujuan utama makalah ini adalah melihat perbedaan dari performa kedua algoritma.

II. DASAR TEORI

2.1 Algoritma Brute Force

Algoritma *Brute Force* merupakan algoritma yang melakukan pemeceahan masalah dengan menggunakna cara yang *straightforward*. Secara umum cara kerja algoritma ini adalah dengan cara membangkitkan semua permutasi dari komponen yang memungkinkan, kemudian setelah dibangkitkan maka akan diperiksa apakah dari himpinan tersebut yang merupakan solusi dari suatu permasalahan sesuai dengan *constraint* dari permasalahan.

Penerapan algortima *Brute Force* dalam permainan sudoku adalah dengan mencoba semua kemungkinan isi kotak elemen dari matriks. Dengan sudoku 9x9 maka algoritma perlu membangkitkan $1.99x10^{77}$ kemungkinan

Pseudocode dari algoritma *Brute Force* merupakan loop berkalang dengan 81 buah dimana setiap loop menumerasi semua kemungkinan angka yang dapat dimiliki setiap elemen matriks sudoku

```
x1,x2,x3,x4,x5,x6,x7,x8,x9,… : integer
for (x1= 1 to 9) {
   for(x2=1 to 9) {
      for(x3=1 to 9){
         for(x4=1 to 9){
            for(x5=1 to 9) {
               for(x81=1 to 9)
               {
               if (Solusi(x1,x2,x3,x4,x5,x6,x7,x8,x9) ){
                     return (x1,x2,…,x9)}} } } }
```

Desain berfikir seperti ini memiliki kompleksitas $O(n^n)$ yang termasuk kategori *non polynomial complexity*, sehingga penggunaan algoritma ini tidak menguntungkan dan akan semakin besar dengan ukuran dari papan sudoku, yang akan memakan waktu yang sangat besar, algoritma ini tentunya dapat diperbaharui dengan penambahan beberapa batasan untuk tidak melanjutkan pencarian seperti jika sudah terdapat suatu angka tertentu pada suatu baris, kolom, maupun grid 3x3, maka tidak perlu dilanjutkan lebih jauh enumerasi pada sesi loop tersebut yang tentunya dapat mempercepat proses penyelesaian.

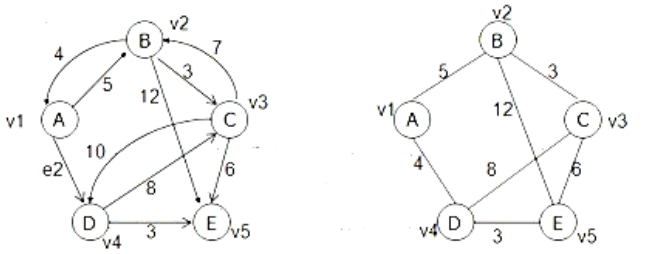
2.2 Graph

Sebuah graf merupakan representasi dari sebuah objek diskrit dan koneksinya. Sebuah graf terdiri dari node (*vertices*) dan busur (*edges*). Node direpresentasikan dengan objek diskrit dan busur direpresentasikan dengan garis yang menghubungkan node.

Graf dapat dibedakan menjadi dua tipe berdasarkan orientasi arah pada busur;

1. Undirected Graph
   Graf, yang busurnya tidak memiliki orientasi arah.

2. Directed Graph
   Graf yang busurnya memiliki orientasi arah.



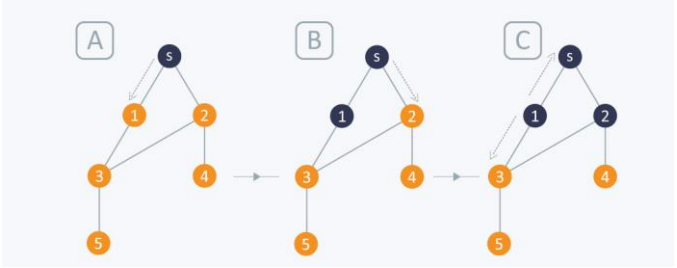
Gambar 4; Graf berarah (kiri) dan Graf tidak berarah (kanan)

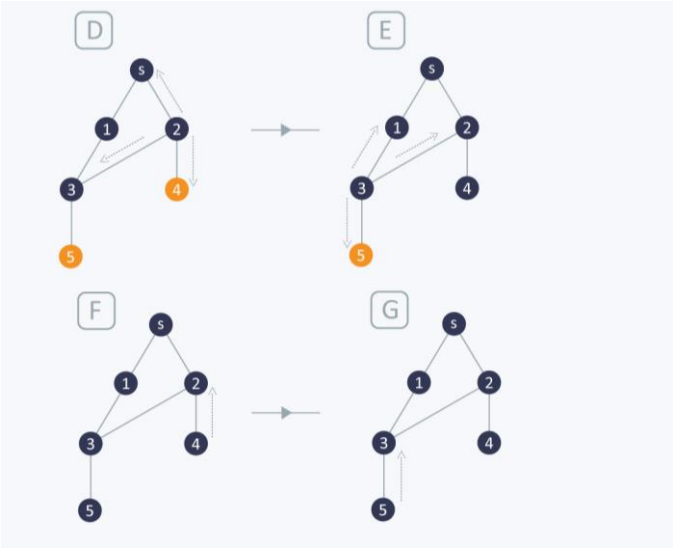
2.3 Breadth-First Search Algorithm

Breadth-First Search (BFS) merupakan algoritma yang melakukan pencarian secara melebar yang mengunjungi simpul secara preorder yaitu mengunjungi suatu simpul kemudian mengunjungi semua simpul yang bertetangga dengan simpul tersebut terlebih dahulu. Selanjutnya, simpul yang belum dikunjungi dan bertetangga dengan simpulsimpul yang tadi dikunjungi , demikian seterusnya. Jika graf berbentuk pohon berakar, maka semua simpul pada aras d dikunjungi lebih dahulu sebelum simpul-simpul pad aras d+1. Algoritma ini memerlukan sebuah antrian q untuk menyimpan simpul yang telah dikunjungi. Simpulsimpul ini diperlukan sebagai acuan untuk mengunjungi simpul-simpul yang bertetanggaan dengannya. Tiap simpul yang telah dikunjungu masuk ke dalam antrian hanya satu kali. Algoritma ini juga membutuhkan table Boolean untuk menyimpan simpul yang te lah dikunjungi sehingga tidak ada simpul yang dikunjungi lebih dari satu kali.

alam algoritma BFS, simpul anak yang telah dikunjungi disimpan dalam suatu antrian. Antrian ini digunakan untuk mengacu simpul-simpul yang bertetangga dengannya yang akan dikunjungi kemudian sesuai urutan pengantrian. Untuk memperjelas cara kerja algoritma BFS beserta antrian yang digunakannya, berikut langkah-langkah algoritma BFS:

1. Masukkan simpul ujing ke dalam antrian
2. Ambil simpul dari awal antrian, lalu cek apakah simpul merupakan solusi
3. Juka simpul merupakan solusi maka pencarian diberhentikan
4. Jika simpul bukan solusi maka masukkan simpul yang bertetangga dengan simpul tersebut kedalam antiran
5. Jika antiran kosong dan setiap simpul sudah dicek, maka pencarian selesai mengembalikan hasil solusi tidak ditemukan
6. Ulangi Langkah kedua





Gambar 5; cara kerja algoritma BFS

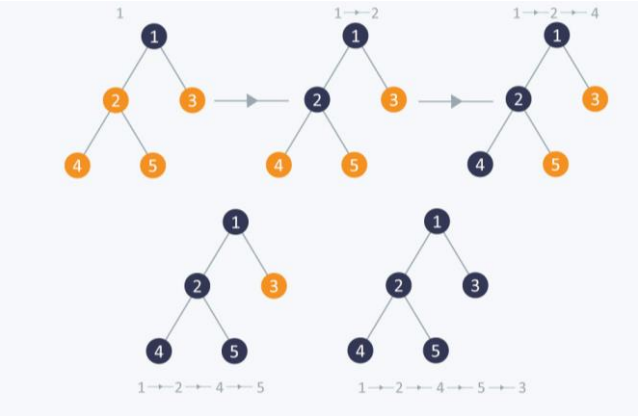
2.4 Depth-First Search Algorithm

Depth-First Search (DFS) adalah salah satu algoritma penelusuran struktur graf / pohon berdasarkan kedalaman. Simpul ditelusuri dari root kemudian ke salah satu simpul anaknya (misalnya prioritas penelusuran berdasarkan anak pertama [simpul sebelah kiri]), maka penelusuran dilakukan terus melalui simpul anak pertama dari simpul anak pertama level sebelumnya hingga mencapai level terdalam.

Setelah sampai di level terdalam, penelusuran akan kembali ke 1 level sebelumnya untuk menelusuri simpul anak kedua pada pohon biner [simpul sebelah kanan] lalu kembali ke langkah sebelumnya dengan menelusuri simpul anak pertama lagi sampai level terdalam dan seterusnya.

Dalam implementasinya DFS dapat diselesaikan dengan cara rekursif atau dengan bantuan struktur data stack. Kita akan membahas dengan cara yang menggunakan stack. Stack yang digunakan adalah stack yang isi elemennya adalah simpul pohon / tree. Bagaimana cara kerjanya ? Berikut ini adalah urutan algoritmanya :

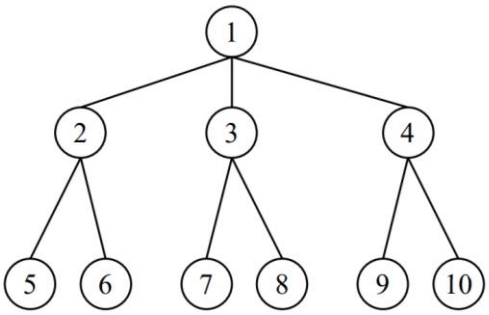
1. Masukkan simpul root ke dalam stack.
2. Ambil dan simpan isi elemen dari stack
3. Periksa apakah simpul merupakan solusi
4. Jika ya hentikan pencarian
5. Jika tidak periksa apakah simpul memiliki anak simpul
6. Jika ya, push semua anak simpul yang dibangkitkan ke dalam stack
7. Jika stack kosong, maka pencarian diberhentikan, dan pencarian selesai



Gambar 6: cara kerja Algoritma DFS

2.2 Algoritma Backtracking

Algoritma *Backtracking* mempunyai prinsip dasar yang sama seperti brute-force yaitu mencoba segala kemungkinan solusi. Perbedaan utamanya adalah pada ide dasarnya, semua solusi dibuat dalam bentuk pohon solusi (pohon ini tentunya berbentuk abstrak) dan algoritma akan menelusuri pohon tersebut secara DFS (depth field search) sampai ditemukan solusi yang layak. Nama backtrack didapatkan dari sifat algoritma ini yang memanfaat karakteristik himpunan solusinya yang sudah disusun menjadi suatu pohon solusi. Agar lebih jelas dapat dilihat contoh berikut:



Misalkan pohon diatas menggambarkan solusi dari suatu permasalahan. Untuk mecapai solusi (5), maka jalan yang ditempuh adalah (1,2,5), demikian juga dengan solusi-solusi yang lain. Algoritma backtrack akan memeriksa mulai dari solusi yang pertama yaitu solusi (5). Jika ternyata solusi (5) bukan solusi yang layak maka algoritma akan melanjutkan ke solusi (6). Jalan yang ditempuh ke solusi (5) adalah (1,2,5) dan jalan untuk ke solusi (6) adalah (1,2,6). Kedua solusi ini memiliki jalan awal yang sama yaitu (1,2). Jadi daripada memeriksa ulang dari (1) kemudian (2) maka hasil (1,2) disimpan dan langsung memeriksa solusi (6). Pada pohon yang lebih rumit, cara ini akan jauh lebih efisien daripada brute-force.

Penerapan algoritma *Backtracking* pada permainan sudoku adalah sebagai berikut :

1. Dimulai dari baris pertama dan kolom pertama
2. Periksa seluruh kemungkinan angka yang dapat dimiliki oleh baris tersebut.
3. Jika terdapat angka yang valid dengan constraint permainan sudoku maka lanjutkan pemeriksaan ke elemen berikutnya dari matriks sudoku
4. Jika tidak terdapat angka valid maka backtrack ke elemen matriks sebelumnya
5. Algoritma ini berhenti saat sudah ditemukan suatu solusi atau jika terdapat kemungkinan adanya solusi

Berikut merupakan pseudocode algoritma *Backtracking* untuk menyelesaikan permsalahan sudoku:

```
Procedure SolveSudoku(input
array:matriks)
Deklarasi
solusi:boolean
i,j:integer
Algoritma
solusi = false;
i=1; j=1;
while (!solusi && i > 0) {
    array[i][j].SetValue(array[i][j].GetValue() + 1);}
while(array[i][j].GetValue()<=9&& !Tempat(i, j)) {
     array[i][j].SetValue(array[i][j].GetValue() + 1);
}
     if (array[i][j].GetValue() <=maks){
        if (i == 9 && j == 9){
            solusi = true;
        } else {
            j = j + 1;
           if (j > 9) {
              i = i + 1; j = 1;}}
```

III. IMPLEMENTASI DAN ANALISIS

3.1 Algortma Brute Force

Dalam implementasi penyelesaian sudoku dengan algoritma *Brute Force*, dilakuakan enumari setiap kemungkinan yang ada, berikut merupakan implementasi algoritma *Brute Force* dengan Bahasa python

```
import time
def solve(s):

   try:
      i  = s.index(0)
   except ValueError:

    return s

   c = [s[j] for j in range(81)
       if not ((i-j)%9 * (i//9^j//9) * (i//27^j//27 | ( i%9//3^j%9//
3 ) ) ) ]

   for v in range(1, 10):
      if v not in c:
         r = solve(s[:i]+[v]+s[i+1:])
         if r is not None:
            return r

if __name__ == '__main__':
   class Sudoku(list):
      '''Sudokus with nicer IO'''
      def __init__(self, content):
         list.__init__(self, [int(i) for i in content.split()]
                  if isinstance(content, str) else content)
      def __str__(self):
         return '\n'.join(
            ' '.join([(str(j) if j != 0 else '-')
                  for j in self[i*9:(i+1)*9]]) for i in range(9))

   problem = Sudoku('''
   0 9 0 2 0 0 0 0 0
   0 0 0 1 0 0 0 9 0
   1 0 2 0 7 0 0 0 0
   3 0 0 0 6 0 0 1 0
   4 0 0 0 3 0 0 6 0
   0 0 6 0 8 0 0 0 3
   6 5 0 0 9 0 8 0 0
   0 2 0 0 0 0 0 0 7
   0 0 0 0 0 7 0 0 6
      ''')

   start_time = time.process_time()
   result = Sudoku(solve(problem))

   print('==== Problem ====\n{0}\n\n=== Solution
====\n{1}'.format(
         problem, result))

   print (time.process_time() - start_time, "detik")
```

3.2 Algoritma Backtracking

Dalam implementasinya , proses pencarian pada algoritma *Backtracking* sebenarnya identik dengan penggunaan algoritma exhaustive-search namun dengan proses yang lebih sistematis. Secara garis besar langkah – langkah yang dilakukan oleh program dalam pencarian solusi adalah sebagai berikut :

1. Menguji sembarang nilai (0 < nilai < n+1) dengan fungsi pembatas yang ada.
2. a. Jika nilai tersebut memenuhi fungsi pembatas maka kotak akan terisi dengan nilai uji tersebut kemudian proses akan berpindah ke kotak berikutnya.
   b. Jika nilai tersebut tidak memenuhi fungsi pembatas, maka program akan mengambil nilai lain yang belum dipakai.
3. Jika semua opsi nilai pada domain nilai telah dicoba dan tidak ada satupun yang memenuhi fungsi pembatas, maka terjadi *Backtracking* ke kotak sebelumnya.
4. Pada kotak hasil *Backtracking* ini, program akan menguji fungsi pembatas dengan nilai baru.
5. a. Jika nilai baru tersebut dapat memenuhi fungsi pembatas, maka nilai tersebut akan diisikan ke dalam kotak. Kemudian proses pengisian akan berpindah ke kotak berikutnya.
   b. Jika semua opsi nilai baru yang ada pada domain tidak dapat memenuhi fungsi pembatas maka akan terjadi lagi *Backtracking* ke kotak sebelumnya.
6. Program akan melakukan proses di atas secara rekursif sampai ditemukan penyelesaian atau tidak terdapat penyelesaian.
   a. Terdapat penyelesaian : program berhasil mengisi semua kotak kosong yang tersedia dengan nilai yang memenuhi fungsi pembatas.
   b. Tidak terdapat penyelesaian : setelah beberapa saat mencari solusi, ternyata program melakukan *Backtracking* hingga kembali ke kotak pertama dan saat nilai indeks baris = -1 menandakan bahwa program tidak berhasil menemukan solusi.

Berikut merupakan implementasinya dalam Bahasa *python*:

```
import time

SIZE = 9

matrix = [
    [0,9,0,2,0,0,0,0,0],
   [0,0,0,1,0,0,0,9,0],
   [1,0,2,0,7,0,0,0,0],
   [3,0,0,0,6,0,0,1,0],
   [4,0,0,0,3,0,0,6,0],
   [0,0,6,0,8,0,0,0,3],
   [6,5,0,0,9,0,8,0,0],
   [0,2,0,0,0,0,0,0,7],
   [0,0,0,0,0,7,0,0,6]]


def print_sudoku():
```

```
    for i in matrix:
        print (i)


def number_unassigned(row, col):
    num_unassign = 0
    for i in range(0,SIZE):
        for j in range (0,SIZE):

            if matrix[i][j] == 0:
                row = i
                col = j
                num_unassign = 1
                a = [row, col, num_unassign]
                return a
    a = [-1, -1, num_unassign]
    return a

def is_safe(n, r, c):

    for i in range(0,SIZE):

        if matrix[r][i] == n:
            return False

    for i in range(0,SIZE):

        if matrix[i][c] == n:
            return False
    row_start = (r//3)*3
    col_start = (c//3)*3;

    for i in range(row_start,row_start+3):
        for j in range(col_start,col_start+3):
            if matrix[i][j]==n:
                return False
    return True


def solve_sudoku():
    row = 0
    col = 0

    a = number_unassigned(row, col)
    if a[2] == 0:
        return True
    row = a[0]
    col = a[1]

    for i in range(1,10):

        if is_safe(i, row, col):
            matrix[row][col] = i

            if solve_sudoku():
                return True
            matrix[row][col]=0
    return False

start_time = time.process_time()
if solve_sudoku():
    print_sudoku()
else:
    print("Tidak Ada Solusi")

print (time.process_time() - start_time, "detik")
```

IV. DOKUMENTASI

1. Percobaan I

```
0 9 0 2 0 0 0 0 0
0 0 0 1 0 0 0 9 0
1 0 2 0 7 0 0 0 0
3 0 0 0 6 0 0 1 0
4 0 0 0 3 0 0 6 0
0 0 6 0 8 0 0 0 3
6 5 0 0 9 0 8 0 0
0 2 0 0 0 0 0 0 7
0 0 0 0 0 7 0 0 6
```

Algoritma *Brute Force*

```
7 9 3 2 5 6 1 8 4
5 6 8 1 4 3 7 9 2
1 4 2 8 7 9 6 3 5
3 7 9 4 6 2 5 1 8
4 8 5 7 3 1 2 6 9
2 1 6 9 8 5 4 7 3
6 5 7 3 9 4 8 2 1
9 2 4 6 1 8 3 5 7
8 3 1 5 2 7 9 4 6
6.03125 detik
```

Algoritma *Backtracking*

```
[7, 9, 3, 2, 5, 6, 1, 8, 4]
[5, 6, 8, 1, 4, 3, 7, 9, 2]
[1, 4, 2, 8, 7, 9, 6, 3, 5]
[3, 7, 9, 4, 6, 2, 5, 1, 8]
[4, 8, 5, 7, 3, 1, 2, 6, 9]
[2, 1, 6, 9, 8, 5, 4, 7, 3]
[6, 5, 7, 3, 9, 4, 8, 2, 1]
[9, 2, 4, 6, 1, 8, 3, 5, 7]
[8, 3, 1, 5, 2, 7, 9, 4, 6]
2.46875 detik
```

2. Percobaan II

```
0 0 0 0 0 5 0 0 8
2 0 5 0 9 0 0 0 6
0 0 0 2 3 0 0 0 0
0 0 0 0 0 0 0 0 0
3 0 0 0 1 4 9 0 0
1 0 7 0 0 0 0 0 2
0 7 0 0 0 9 6 8 0
0 8 0 6 0 0 0 0 7
0 9 0 0 0 0 1 0 3
```

Algoritma *Brute Force*

```
7 6 9 1 4 5 3 2 8
2 3 5 8 9 7 4 1 6
8 1 4 2 3 6 5 7 9
9 4 6 5 8 2 7 3 1
3 2 8 7 1 4 9 6 5
1 5 7 9 6 3 8 4 2
5 7 1 3 2 9 6 8 4
4 8 3 6 5 1 2 9 7
6 9 2 4 7 8 1 5 3
55.171875 detik
```

Algoritma *Backtracking*

```
[7, 6, 9, 1, 4, 5, 3, 2, 8]
[2, 3, 5, 8, 9, 7, 4, 1, 6]
[8, 1, 4, 2, 3, 6, 5, 7, 9]
[9, 4, 6, 5, 8, 2, 7, 3, 1]
[3, 2, 8, 7, 1, 4, 9, 6, 5]
[1, 5, 7, 9, 6, 3, 8, 4, 2]
[5, 7, 1, 3, 2, 9, 6, 8, 4]
[4, 8, 3, 6, 5, 1, 2, 9, 7]
[6, 9, 2, 4, 7, 8, 1, 5, 3]
15.109375 detik
```

Dari kedua percobaan dapat dilihat kalua algoritma *Backtracking* memiliki keunggulan lebih jauh baik dalam performa maupun keefisienan.

VIDEO LINK AT YOUTUBE *(Heading 5)*

https://www.youtube.com/watch?v=WgsYfVtSxAQ&rel=0

ACKNOWLEDGMENT *(Heading 5)*

Penulis mengucapkan terima kasih yang sebesar- besarnya kepada pihak-pihak yang telah mendukung penulis sehingga makalah ini dapat selesai dengan baik. Terima kasih kepada Tuhan YME yang telah memberkati penulis sehingga dapat menyelesaikan makalah ini. Terima kasih kepada kedua orang tua penulis atas segala dukungan baik lahir maupun batin.

Terima kasih kepada Dr. Ir.Rinaldi, M. T. yang mengajar di kelas penulis sehingga penulis dapat memahami materi-materi yang diberikan dan dapat menulis makalah ini. Terima kasih juga kepada para penulis dan pencipta dari semua referensi yang digunakan oleh penulis sebagai sumber data dari makalah ini.

REFERENCES

[1] https://en.wikipedia.org/wiki/Sudoku (diakses pada 2/5/2020)

[2] https://www.geeksforgeeks.org/bfs-vs-dfs-binary-tree/.

[3] https://www.geeksforgeeks.org/*Backtracking*-introduction/.

[4] https://en.wikipedia.org/wiki/*Backtracking* (diakses pada 2/5/2020)

[5] https://www.geeksforgeeks.org/sudoku-*Backtracking*-7/

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

Yan Arie Motinggo - 13518129