Menyelesaikan *Puzzle* Sudoku dengan Algoritma *Brute-Force* dan Runut-Balik

Ahmad Naufal Hakim 13517055
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
ahmadnaufalhakim@gmail.com

Abstrak—Sudoku adalah permainan teka-teki dengan tujuan menyusun angka dari satu sampai sembilan pada *grid* yang terdiri atas 9x9 kotak. Aturan dari permainan ini adalah penyusunan angka-angka pada board tidak boleh ada yang sama dalam setiap baris, kolom, dan 3x3 kotak yang berada di dalam *grid*. Pada awal permainan, sudah terdapat beberapa angka yang berguna sebagai *clue* bagi pemain agar bisa mengisi kotak-kotak selanjutnya. Tingkat kesulitan permainan ini ditentukan oleh banyaknya *clue* yang ada pada awal permainan. Semakin sedikit *clue* yang tersedia, maka Sudoku akan semakin susah untuk diselesaikan. Pada makalah ini, akan dibahas terkait cara menyelesaikan Sudoku dengan menggunakan algoritma Backtracking, serta waktu penyelesaian yang dibutuhkan untuk menyelesaikan Sudoku.

Kata kunci—Sudoku, *Brute-force*, DFS, *Backtracking*.

I. PENDAHULUAN

Sudoku adalah sebuah permainan teka-teki yang terdiri atas sebuah *grid* yang berisi 81 kotak yang tersusun dalam 9 baris dan 9 kolom. Pada *grid* Sudoku itu sendiri, terdapat *subgrid* yang berukuran 3x3. Dalam permainan Sudoku, pemain diminta untuk mengisi setiap kotak di dalamnya dengan satu angka, yaitu angka 1 sampai 9, sehingga pada setiap kotak memiliki angka yang unik untuk masing-masing baris, kolom, dan *subgrid* pada *board*. Tingkat kesulitan permainan Sudoku diukur dari banyaknya *clue* yang terdapat pada *grid* pada awal permainan. Semakin sedikit *clue* yang ada pada *grid*, maka Sudoku akan semakin susah untuk diselesaikan.

Permainan Sudoku diciptakan pada tahun 1970-an oleh Howard Garns. Permainan ini dipopulerkan oleh majalah yang diterbitkan di Jepang pada tahun 1984. Kemudian pada tahun 1997, sudah banyak *puzzle* Sudoku yang ditemukan pada toko buku Jepang. Beberapa tahun kemudian, Wayne Gould, seorang pensiunan hakim dari Hong Kong yang berasal dari Selandia Baru mengembangkan permainan Sudoku menjadi program komputer. Permainan ini sangat populer di Jepang dan meluas hingga ke seluruh dunia.

Menyelesaikan Sudoku adalah salah satu contoh dari masalah NP-*Complete* yang ada. Artinya belum ada algoritma yang mampu menyelesaikan permasalahan tersebut dengan waktu yang konsisten. Secara keseluruhan, ada sebanyak 6.670.903.752.021.072.936.960 kemungkinan *grid* Sudoku berukuran 9x9 yang dapat dibuat. Apabila menggunakan pendekatan dengan algoritma *brute-force* saja tentu tidak efisien, karena algoritma *brute-force* akan mengenumerasi segala kemungkinan *grid* yang ada dan mengecek seluruh kemungkinan *grid* tersebut, sehingga algoritma tersebut tidak mangkus karena akan memakan waktu yang sangat banyak. Meskipun begitu, untuk menyelesaikan Sudoku bisa menggunakan implementasi algoritma runut-balik, dan disertai bantuan algoritma *brute-force*.

II. DASAR TEORI

2.1 Algoritma *Brute-Force*

Algoritma *brute-force* adalah sebuah pendekatan yang lempang (*straightforward*) untuk memecahkan suatu masalah, biasanya didasarkan pada suatu pernyataan masalah (*problem statement*) dan definisi konsep yang dilibatkan. Algoritma *brute-force* memecahkan masalah dengan sangat sederhana, langsung dan dengan cara yang jelas (*obvious way*).

Algoritma *brute-force* umumnya tidak "cerdas" dan tidak mangkus, karena membutuhkan jumlah langkah yang besar dalam penyelesaiannya, terutama bila masalah yang dipecahkan berukuran besar. Kadang-kadang algoritma *brute-force* disebut juga algoritma **naif** (*naive algorithm*).

Algoritma *brute-force* seringkali merupakan pilihan yang kurang disukai karena ketidakmangkusannya, tetapi dengan mencari pola-pola yang mendasar, keteraturan, atau trik-trik khusus, biasanya akan membantu kita menemukan algoritma yang lebih cerdas dan lebih mangkus.

Untuk masalah yang ukurannya kecil, kesederhanaan *brute-force* biasanya lebih diperhitungkan daripada kemangkusannya. Algoritma *brute-force* lebih sering digunakan sebagai basis bila membandingkan beberapa alternatif algoritma yang mangkus. Misalnya pada perhitungan a^n, dengan algoritma *brute-force* mudah diperlihatkan bahwa kompleksitas waktunya adalah $O(n)$. Bila perhitungan tersebut diselesaikan dengan metode *Divide and Conquer*, maka kompleksitas waktunya jauh lebih baik daripada *brute-force*, yaitu $O(\log n)$.

Meskipun *brute-force* bukan merupakan teknik pemecahan masalah yang mangkus, namun teknik *brute-force* dapat diterapkan pada sebagian besar masalah. Agak sukar menunjukkan masalah yang tidak dapat dipecahkan dengan teknik *brute-force*. Bahkan ada masalah yang hanya dapat dipecahkan secara *brute-force*. Beberapa pekerjaan mendasar di dalam komputer dilakukan secara *brute-force*, seperti menghitung jumlah dari *n* buah bilangan, mencari elemen terbesar di dalam tabel, dan sebagainya.

Selain itu, algoritma *brute-force* seringkali lebih mudah diimplementasikan daripada algoritma yang lebih canggih, dan karena kesederhanaannya, kadang-kadang algoritma *brute-force* dapat lebih mangkus (ditinjau dari segi implementasi).

2.2 Algoritma Runut-balik

Algoritma runut-balik (backtracking) adalah algoritma yang berbasis pada DFS untuk mencari solusi persoalan secara lebih mangkus. Runut-balik, yang merupakan perbaikan dari algoritma *brute-force*, secara sistematis mencari solusi persoalan di antara semua kemungkinan solusi yang ada. Dengan metode ini, kita tidak perlu memeriksa semua kemungkinan solusi yang ada. Hanya pencarian yang mengarah ke solusi saja yang selalu dipertimbangkan. Akibatnya, waktu pencarian dapat dihemat. Runut-balik lebih alami dinyatakan secara rekursif. Kadang-kadang disebutkan pula bahwa runut-balik merupakan bentuk tipikal dari algoritma rekursif.

Istilah runut-balik pertama kali diperkenalkan oleh D.H. Lehmer pada tahun 1950. Selanjutnya, R.J. Walker, Golomb, dan Baumert menyajikan uraian umum tentang runut-balik dan penerapannya pada berbagai persoalan. Saat ini algoritma runut-balik banyak diterapkan untuk program *games* (seperti menemukan jalan keluar dalam labirin, catur, *tic-tac-toe*, dll) dan permasalahan pada bidang *artificial intelligence*).

2.2.1 Properti Umum Algoritma Runut-balik

a. Solusi persoalan

Solusi dinyatakan dalam bentuk vektor dengan tupel $X = (x_1, x_2, \ldots, x_n)$, $x_i \in S_i$. Mungkin saja $S_1 = S_2 = \ldots = S_n$.

Contoh: $S_i = \{0,1\}$, $x_i = 0$ atau $x_i = 1$

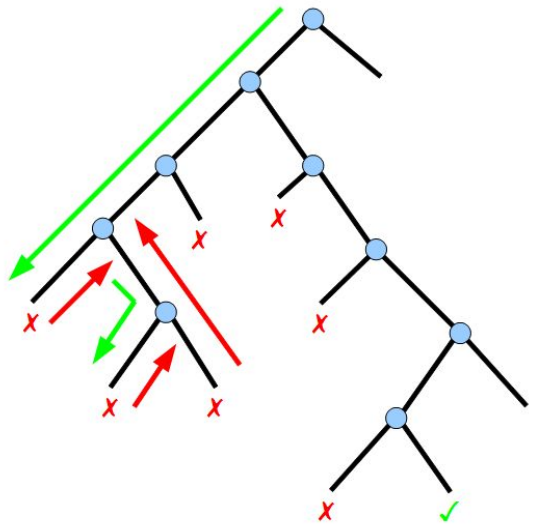
b. Fungsi pembangkit nilai x_k

Fungsi pembangkit dinyatakan sebagai *T(k)* di mana *T(k)* membangkitkan nilai untuk x_k yang merupakan komponen vektor solusi.

c. Fungsi pembatas

Dinyatakan dalam predikat $B(x_1, x_2, ..., x_k)$. *B* bernilai benar jika $(x_1, x_2, ..., x_k)$ mengarah ke solusi. Jika benar, maka pembangkitan nilai untuk x_{k+1} dilanjutkan, tetapi jika tidak, maka $(x_1, x_2, ..., x_k)$ dibuang.

2.2.2 Prinsip Pencarian Solusi dengan Algoritma Runut-balik



Gambar 1. Ilustrasi algoritma runut-balik
(sumber: dreamincode.net)

a. Solusi dicari dengan membentuk lintasan dari akar ke daun. Aturan pembentukan yang dipakai mengikuti aturan *depth-first order* (DFS).
b. Simpul-simpul yang sudah dilahirkan dinamakan **simpul hidup**.
c. Simpul-simpul yang sedang diperluas dinamakan **simpul ekspan**.
d. Tiap kali simpul ekspan diperluas, lintasan yang dibangun olehnya bertambah panjang.
e. Jika lintasan yang sedang dibentuk tidak mengarah ke solusi, maka simpul ekspan tersebut akan "dibunuh" sehingga menjadi **simpul mati**.
f. Fungsi yang digunakan untuk membunuh simpul ekspan menerapkan **fungsi pembatas**. Simpul yang mati tidak akan pernah diperluas lagi.

2.2.3 Skema Umum Algoritma Runut-balik

Untuk menyelesaikan Sudoku, ada dua cara dalam menyatakan algoritma runut-balik, yaitu secara rekursif dan iteratif. Pada makalah ini akan dibahas cara menyelesaikan Sudoku dengan menggunakan algoritma runut-balik secara rekursif, karena algoritma runut-balik lebih alami dinyatakan dalam bentuk rekursi. Berikut adalah skema umum dari algoritma tersebut:

```
function solve(input g : Node)
Algoritma:
if (mencapai solusi) then
  return True
else
  for semua node yang belum dicoba
    kunjungi node
    if (solve(g)) then
      return True
    else
      backtrack
    endif
  endfor
  return False
endif
```

III. PENGGUNAAN ALGORITMA RUNUT BALIK UNTUK MENYELESAIKAN SUDOKU

Untuk bisa mengaplikasikan algoritma runut-balik, pertama kita perlu membuat sebuah struktur data untuk merepresentasikan sebuah *grid* Sudoku. Pada makalah ini, penulis menggunakan bahasa pemrograman *Python* dan memanfaatkan *array* untuk merepresentasikan sebuah *grid* Sudoku. Nilai angka pada kotak-kotak dalam *grid* akan dibaca melalui file eksternal. Berikut ini adalah contoh implementasinya:

```
def readFile(grid, fileInp):
  with open(fileInp+".txt",'r') as f:
    for line in f :
      line = line.rstrip()
        row = []
        for value in line :
          if (value=='.'):
            row.append(0)
          else:
            row.append((int)(value))
        grid.append(row)
```

Selanjutnya, dibuat implementasi aturan yang berlaku pada permainan Sudoku, yaitu angka yang ada pada setiap kotak harus unik pada setiap baris, kolom, dan *subgrid*. Berikut adalah contoh implementasinya:

```
def isInRow(grid, row, n):
  for j in range(len(grid)):
    if (grid[row][j]==n):
      return True
  return False

def isInCol(grid,col,n):
  for i in range(len(grid)):
    if (grid[i][col]==n):
      return True
  return False

def isInSubGrid(grid, row, col, n):
  for i in [0,1,2]:
    for j in [0,1,2]:
      if(grid[((int)(row/3))*3+i]
      [((int)(col/3))*3+j]==n):
        return True
    return False

def isSafe(grid, row, col, n):
  return not isInSubGrid(grid, row,
  col, n) and not isInRow(grid, row,
  n) and not isInCol(grid, col, n)
```

Lalu, untuk membuat implementasi dari algoritma runut-balik, perlu dibuat suatu fungsi yang mengecek apakah semua kotak dalam *grid* Sudoku sudah terisi semuanya, serta sebuah fungsi yang akan mencari kotak pertama yang akan di-*assign* pertama kali pada saat pemanggilan algoritma runut-balik. Berikut ini adalah contoh implementasinya:

```
def isGridFull(grid):
  gridTemp = np.array(grid)
  row,col = gridTemp.shape
  for i in range(row):
    for j in range(col):
      if (isUnassigned(grid,i,j)):
        return False
  return True

def searchFirstUnassignedCell(grid,
cell):
  if (not isGridFull(grid)):
    gridTemp = np.array(grid)
    row,col = gridTemp.shape
    for i in range(row):
      for j in range(col):
          if (isUnassigned(grid,i,j)):
            cell[0] = i
            cell[1] = j
```

Pada makalah ini, penulis menyelesaikan permasalahan Sudoku dengan menggunakan pendekatan menggunakan algoritma runut-balik secara rekursif, sehingga program akan memanggil fungsi tersebut sehingga semua kotak dalam *grid* Sudoku terisi seluruhnya. Berikut ini adalah *pseudocode* dari algoritma runut-balik yang akan digunakan:

```
function solveGrid(input g : grid)
{Mengisi grid Sudoku dengan cara
backtracking, yaitu mengisi seluruh
kotak dalam grid Sudoku dengan semua
kemungkinan yang memenuhi aturan
permainan Sudoku}

Algoritma:
if (grid penuh) then
  return True
else
  for angka 1 sampai 9 yang memenuhi
  aturan Sudoku do
    isiCell(angka)
    if (solveGrid(g)) then
      return True
    else
      hapusIsiCell(angka)
    endif
  endfor
endif
```

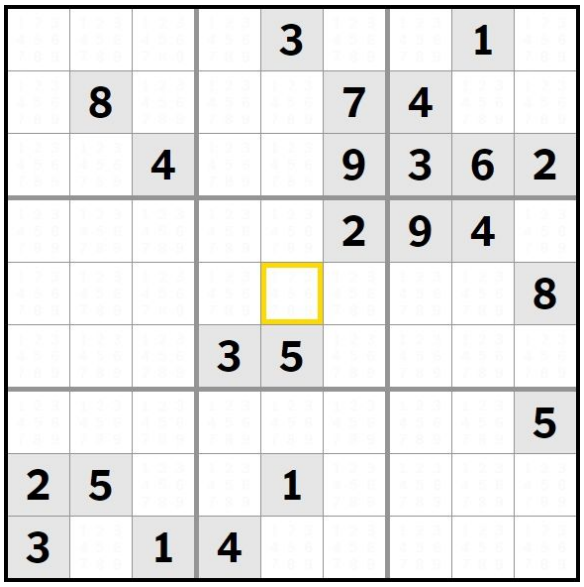
Dengan memanfaatkan fungsi dan prosedur pembantu yang telah didefinisikan di atas, maka algoritma runut-balik sudah bisa

diimplementasikan. Berikut ini adalah contoh implementasinya:

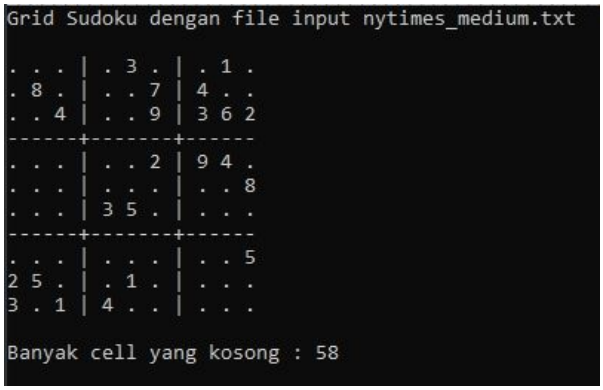
```
def solveSudokuBacktrack(grid):
  if (isGridFull(grid)):
    return True
  else:
    startCell = [0,0]
    searchFirstUnassignedCell(grid,
    startCell)

    row = startCell[0]
    col = startCell[1]

    for val in range(1,10):
      if (isSafe(grid, row, col,
      val)):
        grid[row][col] = val
          if
          (solveSudokuWithBacktrack
          (grid)):
            return True
          else:
            grid[row][col] = 0
    return False
```

Proses algoritma runut-balik dimulai dengan mengecek apakah *grid* sudah penuh atau belum. Jika *grid* sudah penuh, maka proses *backtracking* selesai. Jika tidak, maka program akan mencari kotak pertama yang belum diisi sebuah angka, lalu mengisi semua kemungkinan angka yang bisa diisi pada kotak tersebut. Kemudian program secara rekursif akan memanggil kembali fungsi tersebut. Algoritma akan terus berjalan hingga seluruh kotak pada *grid* terisi penuh. Jika terdapat kotak yang tidak bisa diisi oleh angka berapapun, maka program akan melakukan *backtrack* dengan cara menghapus isi dari kotak yang sebelumnya sudah terisi.

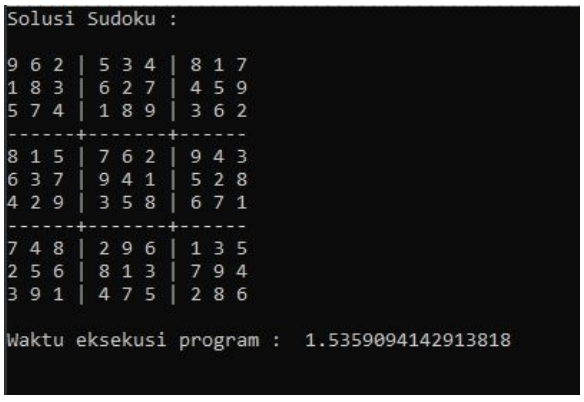
Berikut ini adalah contoh *puzzle* Sudoku yang ada di website https://www.nytimes.com/crosswords/game/sudoku/medium beserta program yang digunakan untuk menyelesaikan *puzzle* tersebut:



Gambar 2. *Puzzle* Sudoku New York Times Medium (sumber:www.nytimes.com/crosswords/game/sudoku/medium)



Gambar 3. *Grid* Sudoku New York Times Medium (sumber: pribadi)



Gambar 4. Solusi Sudoku New York Times Medium (sumber: pribadi)

IV. PENGGUNAAN ALGORITMA RUNUT BALIK DENGAN BANTUAN BRUTE-FORCE

Algoritma runut-balik bisa digunakan untuk menyelesaikan seluruh kemungkinan *puzzle* Sudoku yang ada, namun ada kalanya kasus di mana *puzzle* Sudoku yang hendak diselesaikan membutuhkan banyak proses *backtracking*, dikarenakan sedikitnya *clue* yang ada di awal permainan. Hal ini bisa membuat algoritma runut-balik menjadi tidak terlalu efisien.

Biasanya dalam menyelesaikan *puzzle* Sudoku, pemain pasti akan mengisi kotak-kotak pada *grid* Sudoku yang memiliki solusi angka yang sudah pasti hanya bisa diisi pada kotak tersebut, atau mengisikan solusi angka pada kotak yang hanya bisa diisi pada baris, kolom, atau *subgrid* yang bersesuaian dengan kotak tersebut. Maka dari itu, agar implementasi algoritma runut-balik bisa lebih mangkus dan efisien, akan diimplementasikan empat buah fungsi baru dengan algoritma *brute-force* yang bisa mengimplementasikan strategi di atas. Berikut ini adalah contoh implementasinya:

```
def solveObviousCell(grid,arrayCell):
  if (not isGridFull(grid)):
    gridTemp = np.array(grid)
    row,col = gridTemp.shape
    values = [1,2,3,4,5,6,7,8,9]
    for i in range(row):
      for j in range(col):
        candidateSolution = []
        for val in values:
          if (isUnassigned(grid,i,j
          and isSafe(grid,i,j,val)):
            candidateSolution.append
            (val)
        if(len(candidateSolution)==1):
          grid[i][j] =
          candidateSolution[0]
          arrayCell.append((i,j))
    arrayCell = list(set(arrayCell))
```

```
def solveObviousRow(grid,arrayCell):
  if (not isGridFull(grid)):
    gridTemp = np.array(grid)
    row,col = gridTemp.shape
    values = [1,2,3,4,5,6,7,8,9]
    for i in range(row):
      for val in values:
        candidateSolution = {val:[]}
        for j in range(col):
          if (isUnassigned(grid,i,j)
          and isSafe(grid,i,j,val)):
            candidateSolution[val]
            .append((i,j))
      if (len(candidateSolution[val])
      ==1):
        grid[candidateSolution[val]
        [0][0]][candidateSolution[val]
        [0][1]] = val
        arrayCell.append(
        (candidateSolution[val]
        [0][0],candidateSolution[val]
        [0][1]))
    arrayCell = list(set(arrayCell))

def solveObviousCol(grid,arrayCell):
  if (not isGridFull(grid)):
    gridTemp = np.array(grid)
    row,col = gridTemp.shape
    values = [1,2,3,4,5,6,7,8,9]
    for j in range(col):
      for val in values:
        candidateSolution = {val:[]}
        for i in range(row):
          if (isUnassigned(grid,i,j)
          and isSafe(grid,i,j,val)):
            candidateSolution[val]
            .append((i,j))
      if (len(candidateSolution[val])
      ==1):
        grid[candidateSolution[val]
        [0][0]][candidateSolution[val]
        [0][1]] = val
```

```
        arrayCell.append(
        (candidateSolution[val]
        [0][0],candidateSolution[val]
        [0][1]))
    arrayCell = list(set(arrayCell))

def solveObviousSubGrid(grid,
arrayCell):
  if (not isGridFull(grid)):
    values = [1,2,3,4,5,6,7,8,9]
    for iSubGrid in [0,1,2]:
      for jSubGrid in [0,1,2]:
        for val in values:
          candidateSolution = {val:[]}
          for offsetRow in [0,1,2]:
            for offsetCol in [0,1,2]:
              if (isUnassigned(grid,
              (iSubGrid*3+offsetRow),
              (jSubGrid*3+offsetCol))
              and isSafe(grid,
              (iSubGrid*3+offsetRow),
              (jSubGrid*3+offsetCol),
              val)):
                candidateSolution[val]
                .append((iSubGrid*3
                +offsetRow,jSubGrid*3
                +offsetCol))
          if (len(candidateSolution
          [val])==1):
            grid[candidateSolution
            [val][0][0]]
            [candidateSolution
            [val][0][1]] = val
            arrayCell.append(
            (candidateSolution[val]
            [0][0],candidateSolution
            [val][0][1]))
    arrayCell = list(set
    (arrayCell))
```

Keempat algoritma tersebut akan dijalankan di dalam algoritma runut-balik yang sudah dibuat sebelumnya, sehingga setiap kali fungsi dipanggil, keempat fungsi tersebut akan dipanggil terlebih dahulu untuk menyelesaikan kotak-kotak yang mudah untuk diselesaikan sebelum menyelesaikan *grid* Sudoku keseluruhan dengan algoritma runut-balik yang murni.

Namun, jika di tengah proses menyelesaikan *grid* Sudoku dengan algoritma runut-balik diperlukan untuk melakukan *backtracking*, maka kotak-kotak yang sebelumnya telah di-*assign* oleh fungsi dengan algoritma *brute-force* harus dihapus nilainya. Sehingga, harus dibuat suatu fungsi sederhana yang menghapus nilai dari semua kotak yang telah diisi oleh empat fungsi algoritma *brute-force* tersebut untuk proses *backtracking*. Berikut ini adalah contoh implementasinya:

```
def unsolveObvious(grid,arrayCell):
    for cell in arrayCell:
        grid[cell[0]][cell[1]] = 0
```

Sehingga algoritma runut-balik yang didapatkan adalah sebagai berikut:

```
def solveSudokuImprovedBacktrack
(grid):
  obvCells = []
  solveObviousSubGrid(grid,obvCells)
  solveObviousRow(grid,obvCells)
  solveObviousCol(grid,obvCells)
  solveObviousCell(grid,obvCells)
  if (isGridFull(grid)):
    return True
  else:
    startCell = [0,0]
    searchFirstUnassignedCell(grid,
    startCell)
      row = startCell[0]
      col = startCell[1]
```

```
    for val in range(1,10):
      if (isSafe(grid, row, col,
      val)):
        grid[row][col] = val
        if (solveSudokuImproved
        Backtrack(grid)):
          return True
        else:
          grid[row][col] = 0
  unsolveObvious(grid,obvCells)
    return False
```

Selanjutnya, penulis akan membandingkan kedua algoritma utama yang telah dijabarkan di atas, di mana algoritma pertama adalah algoritma runut-balik murni, dan algoritma kedua adalah algoritma runut-balik + *brute-force.*

Untuk membandingkan kedua algoritma tersebut, penulis menguji kedua algoritma dengan 5 buah *puzzle* Sudoku dengan tingkat kesulitan di atas *Hard* yang hanya memiliki 21 *clue* di awal permainan. Waktu eksekusi kedua algoritma untuk masing-masing *puzzle* yang tercatat adalah sebagai berikut:

No.	Waktu eksekusi algoritma 1 (s)	Waktu eksekusi algoritma 2 (s)
1.	3.150596	0.023975
2.	6.336058	0.028924
3.	12.011887	0.086748
4.	15.133148	0.028922
5.	5.854344	0.019967

Tabel 1. Perbandingan kecepatan algoritma pertama terhadap algoritma kedua
(sumber: pribadi)

Berdasarkan percobaan yang telah dilakukan, didapatkan hasil sesuai dengan tabel di atas. Rata-rata waktu yang dibutuhkan algoritma pertama (runut-balik murni) untuk menyelesaikan *puzzle* Sudoku adalah 8.497206 detik. Sedangkan rata-rata waktu yang dibutuhkan algoritma kedua (runut-balik + *brute-force*) adalah 0.037707 detik.

V. KESIMPULAN

Hal ini menunjukkan bahwa algoritma runut-balik bisa diimplementasikan bersama dengan algoritma *brute-force* untuk menyelesaikan *puzzle* Sudoku 225 kali lebih cepat bila dibandingkan dengan algoritma runut-balik yang murni.

VI. DAFTAR PUSTAKA

[1] Munir, Rinaldi, Diktat Kuliah IF2211 Strategi Algoritma. Program Studi Teknik Informatika ITB, 2015.

[2] https://www.kristanix.com/sudokuepic/sudoku-solving-techniques.php
Diakses pada 26 April 2019

[3] https://www.minimumsudoku.com/
Diakses pada 26 April 2019

[4] https://www.codeproject.com/Articles/865143/A-Sudoku-program-with-some-advanced-features
Diakses pada 26 April 2019

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, 26 April 2019

Ahmad Naufal Hakim
13517055