Penggunaan Algoritma Runut-Balik untuk Menyelesaikan Teka-Teki Sudoku

Muhammad Abdullah Munir 13516074
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
mabdullahmunir@s.itb.ac.id

Abstrak—Sudoku adalah sebuah *puzzle* berbentuk matrix berukuran 9x9 yang harus di isi dengan angka 1-9 dengan batasan-batasan tertentu dalam puzzle tersebut. Dalam makalah ini akan dibahas mengenai cara menyelesaikan *puzzle* Sudoku dengan menggunakan algoritma DFS-Backtracking dengan ditunjukkan waktu yang diperlukan serta berapa banyak status yang dibuat dalam proses menyelesaikan *puzzle* ini.

Kata Kunci—Sudoku, DFS, *Backtracking*, Pemenuhan Konstrain.

I. PENDAHULUAN

Pada zaman yang modern ini telah banyak sekali permainan dengan *genre puzzle* atau teka-teki yang telah meluas ke seluruh dunia. Permainan teka-teki ini terkadang memerlukan waktu yang lama untuk menyelesaikannya. Namun, ada pula beberapa puzzle yang ternyata dapat diselesaikan dikarenakan ada sebuah teori tertentu mengenai permainan tersebut.

Dalam pengertiannya teka-teki dapat diartikan sebagai sebuah permainan yang dibuat untuk mengasah pikiran dari pemainnya. Semakin tinggi tingkat kerumitan dari teka-teki yang dibuat, maka akan semakin terlatih pula kemampuan nalar dari para pemain teka-teki tersebut.

Pada makalah ini akan membahas tentang permainan teka-teki sudoku. Menyelesaikan teka-teki sudoku bisa dikatakan cukup mudah. Sehingga, saya mencoba untuk membuat program untuk menyelesaikan teka-teki sudoku. Pemilihan algoritma yang akan digunakan untuk menyelesaikan teka-teki ini sangat penting. Jika kita menggunakan algoritma *bruteforce* akan diperlukan waktu yang sangat lama.

Sehingga pada makalah ini, saya mencoba menyelesaikan teka-teki sudoku dengan algoritma *backtracking*.

II. SUDOKU

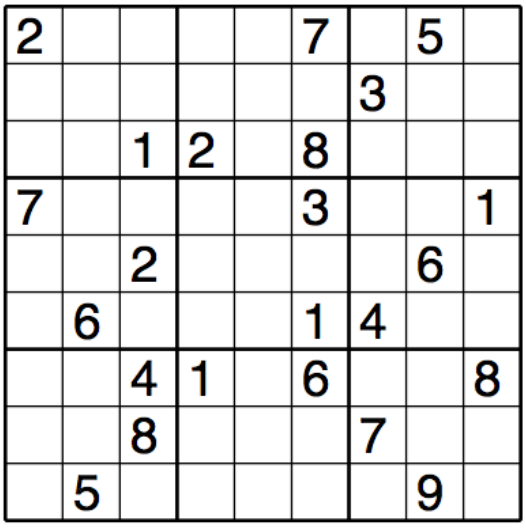
Seperti yang telah disebutkan di abstrak. Sudoku adalah sebuah *puzzle* yang mengharuskan kita untuk mengisi tabel 9x9. Pada awalnya aka ada beberapa kotak yang telah di isi oleh angka. Akan tetapi, di Sudoku terdapat beberapa batasan untuk menyelesaikan puzzle ini yaitu tidak boleh ada angka yang sama dalam satu baris, satu kolom, atau satu blok.

Satu blok dalam teka-teki sudoku ini adalah tabel berukuran 3x3 yang ada dalam tabel 9x9. Sehingga dalam tabel sudoku terdapat 9 blok kecil. Dalam satu blok ini juga tidak boleh ada angka yang sama.

Sejarah munculnya teka-teki ini ialah ketika pertama kali muncul di surat kabar pada akhir abad ke-19 di majalah *Dell Magazines*. Permainan ini didesain oleh Howard Grans dan mulai popular di Jepang pada tahun 1984 ketika dimuat di majalah bulanan *Nikoli*. Pada tahun 1986 Nikoli menambahkan aturan baru dalam sudoku. Pertama, angka yang dimasukkan dalam soal tidak boleh melebihi 32 buah angka. Serta, soal harus simetris.

Teka-teki sudoku ini menarik dikarenakan pemain hanya perlu mengisi tiap kotak dengan angka 1 sampai 9. Dengan batasan tidak boleh ada angka berulang dalam baris dan kolom yang sama, serta dalam blok tersebut. Sehingga untuk menyelesaikannya tidak diperlukan perhitungan apapun.

Sedikit hal lagi yang menarik dari teka-teki sudoku ini ialah, ada 6.670.903.752.021.072.936.960 teka-teki sudoku yang dapat dibuat dari tabel yang berukuran 9x9. Sehingga terdapat banyak sekali kombinasi yang dapat dibuat dan tidak akan kehabisan teka-teki sudoku.



Gambar 1 Hard Sudoku Puzzle
(*http://apollon.issp.u-tokyo.ac.jp/~watanabe/sample/sudoku/index.html*)

III. BACKTRACKING

Backtracking atau runut-balik adalah algoritma umum yang sering digunakan untuk mencari solusi dari masalah komputasi dengan diketahui batasan-batasannya, sehingga bisa melewati beberapa kondisi jika diketahui tidak akan bisa mencapai solusi akhir. Algoritma ini pertama kali dikenalkan oleh D. H. Lehmer pada tahun 1950. Kemudian R.J Walker, Golomb, dan Baumert memberikan penjelasan mengenai algoritma runut-balik ini.

Algoritma *backtracking* ini merupakan pengembangan dari algoritma DFS. Pada algoritma DFS, program akan melakukan pencarian dengan mengutamakan kedalaman terlebih dahulu. Jika dengan algoritma DFS yang biasa, pencarian akan sangat lama. Dikarenakan solusi belum tentu akan didapatkan di cabang yang sekarang sedang di proses.

Sedangkan pada Algoritma *backtracking* jika pada suatu cabang dilakukan pengecekan dan cabang tersebut melanggar batasan yang ditetapkan. Program akan mundur ke kondisi sebelumnya, dimana belum terjadi proses yang menyebabkan batasan tersebut dilanggar, dan melewati proses yang barusan. Hal tersebut dilakukan berulang-ulang sampai masalah diselesaikan.

Algoritma *backtracking* atau runut-balik ini memiliki 3 properti umum, yaitu :

A. Solusi Permasalahan.

   Solusi permasalahan direpresentasikan sebagai sebuah vektor dengan *n-tuple* : $X = (x_1,x_2,\ldots,x_n)$, $x_i \in S_i$. x_i berupa langkah yang diambil ketika membuat solusi dari permasalahan. Selain itu, dapat dimungkinkan S_i sama dengan S_n. $(S_1 = S_2 = \ldots = S_n)$

B. Fungsi Pembangkit.

   Fungsi pembangkit ini dinyatakan sebagai T(k). Fungsi T(k) ini akan menghasilkan sebuah nilai atau langkah yang dapat dilakukan. Hasil dari fungsi ini bisa juga disebut dengan x_k, yang merupakan komponen dari vektor solusi.

C. Fungsi Pembatas.

   Fungsi pembatas ini dinyatakan sebagai $B(x_1,x_2,\ldots,x_n)$. Fungsi pembatas ini akan melakukan pengecekan apakah $B(x_1,x_2,\ldots,x_n)$ dapat mengarah ke solusi. Hasil dari fungsi ini akan bernilai *true* jika dapat mencapai solusi, sehingga pembangkitan nilai kondisi selanjutnya akan dilakukan. Akan tetapi, jika false maka $(x_1,x_2,\ldots,x_n)$ akan dibuang karena tidak bisa mencapai kondisi solusi jika menggunakan langkah ini.

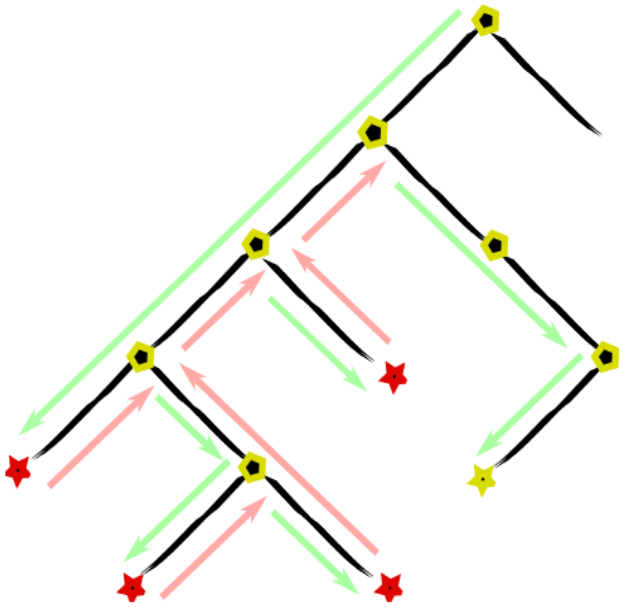
Semua kemungkinan solusi yang didapatkan dari algoritma ini disebut dengan ruang solusi. Sebagai contoh, untuk kasus *knapsack* 0/1 dengan n = 4. Solusi dari permasalahan dapat dituliskan dengan sebuah vektor (x_1,x_2, x_3,x_4) dengan $x_i \in \{0, 1\}$. Dalam kasus ini, ruang solusinya sebanyak 2^n dengan $n = 4$.

Ruang Solusi ini digambarkan dalam bentuk pohon, dengan tiap simpul pohon menyatakan status dari permasalahan dan sisi pohon berupa langkah yang diambil atau x_i. Untuk mendapatkan vektor solusi dari permasalahan ini, yaitu dengan cara mengikuti lintasan dari akar menuju ke daun yang memenuhi kondisi dari permasalahan. Bukan daun yang gagal mencapai solusi.

Algoritma ini membentuk solusi dengan membuat pohon ruang status. Pohon ini dibuat dengan mengikuti aturan dari algoritma *depth-first order* (DFS). Simpul yang sudah dibuat ketika proses penyelesaian disebut dengan simpul hidup. Sedangkan simpul hidup yang sedang diperluas disebut dengan simpul-E.

Setiap simpul-E diperluas, akan terbentuk lintasan-lintasan baru. Tiap lintasan baru ini selanjutnya dicek, jika lintasan yang dibentuk ini tidak menuju ke solusi permasalahan, maka simpul-E tersebut akan dibunuh atau tidak dilanjutkan perluasannya. Yang kemudian disebut dengan simpul mati. Fungsi yang digunakan untuk membunuh simpul-E ini adalah fungsi pembatas.

Jika ketika proses pembentukan lintasan solusi mencapai sebuah simpul mati, maka program akan kembali ke simpul sebelumnya (*parent*). Proses ini yang disebut dengan *backtracking*. Setelah kembali ke simpul sebelumnya dilakukan lagi perluasan yang lainnya.



Gambar 2 Pohon status *backtracking* (*https://homepages.cwi.nl/~steven/Talks/2013/08-02-evolution/*)

Pencarian akan dihentikan ketika simpul yang sekarang merupakan solusi akhir dari permasalahan (*goal state*). Solusi akhir dari permasalahan ini ialah simpul terakhir, atau jika diperlukan proses untuk mencapainya dapat dicari dengan mengikuti dari akar sampai ke goal state. Dengan mencatat sisi dari pohon yang berupa langkah yang diambil tiap state.

```
boolean solve(Node n) {
    if n is a leaf node {
        if the leaf is a goal node, return true
        else return false
    } else {
        for each child c of n {
            if solve(c) succeeds, return true
        }
        return false
    }

}
```

Gambar 3 Pseudocode algoritma *backtracking* (*https://www.cis.upenn.edu/~matuszek/cit594-2002/Pages/backtracking.html*)

IV. Penyelesaian Teka-Teki Sudoku dengan Algoritma Runut-Balik

Teka-teki sudoku sebenarnya dapat diselesaikan dengan banyak cara, beberapa diantaranya *bruteforce* dan *backtrack*. Pada makalah ini saya memilih menggunakan algoritma runut-balik dikarenakan jika menggunakan algortima *bruteforce* akan memerlukan waktu yang sangat lama.

Sebagai contoh, jika ada petak sudoku dengan ukuran 9x9. Kemudian telah terisi 30 angka. Jika menggunakan algoritma *bruteforce*, program akan langsung mengisi semua kotak yang kosong. Jumlah kotak yang kosong ada 81 – 30 = 51 kotak. Selain itu, tiap kotak dapat diisi dengan angka antara 1 sampai 9.

Jika digunakan algoritma *bruteforce* program akan mencoba membuat petak sebanyak 9^{51} = 4.638.397.686.588.101.979.328.150.167.890.591.454.318.967 .698.009 kombinasi petak sudoku. Hal ini dikarenakan, program hanya akan mengecek ketika petak sudah diisi, tanpa mempedulikan jika ada bilangan yang sama di setiap pengisian kotak.

Sebelum membuat fungsi untuk menyelesaikan teka-teki sudoku yang memanfaatkan algoritma runut-balik ini, harus didefinisikan terlebih dahulu properti umum dari algoritma runut-balik. Berupa solusi permasalahan, fungsi pembangkit, fungsi pembatas.

Yang pertama, solusi permasalahan dari teka-teki sudoku ini berupa vektor *n-tuple* X = (x_{00}, x_{01}, .., x_{88}), dengan batasan $x_{ij} \in \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, $0 \leq i \leq 8$, $0 \leq j \leq 8$. Berdasarkan vektor solusi di atas, vektor merupakan petak sudoku yang telah terisi dengan angka yang sesuai.

Fungsi Pembangkit, fungsi ini menerima masukan berupa kondisi petak sudoku saat ini. Kemudian akan dicari kotak yang masih kosong. Setelah ditemukan kotak yang kosong, akan diisi dengan suatu angka antara 1 sampai 9. Sehingga akan terbentuk sebuah kondisi yang baru.

Untuk fungsi yang terakhir ialah fungsi pembatas. Fungsi ini dibuat untuk melakukan pengecekan apakah status saat ini belum melanggar batasan yang ditetapkan. Dalam hal ini, pada sudoku terdapat 3 batasan yang pasti. Pertama, dalam satu baris tidak boleh ada angka yang sama. Kedua, dalam satu kolom tidak boleh ada angka yang sama. Dan yang terakhir, dalam satu blok tidak boleh ada angka yang sama.

Jika fungsi pembatas mengembalikan *false* maka simpul tersebut akan dimatikan, dan program akan *backtrack* ke status yang sebelumnya. Hal ini dilakukan secara terus menerus sampai petak teka-teki sudoku terisi semua.

Pada gambar di bawah, terdapat potongan kode *python* yang berisi fungsi *solver* yang digunakan untuk menyelesaikan teka-teki sudoku. Fungsi ini dilakukan secara rekursif, dengan pada awalnya melakukan pengecekan apakah kondisi saat ini telah mencapai goal state. Jika sudah, maka proses rekursif akan berhenti.

Pengecekan kondisi pada fungsi sebelumnya sekaligus dengan mencari kotak yang masih belum diisi atau masih bernilai 0. Kemudian sebelum dimasukkan angka, program akan melakukan pengecekan terlebih dahulu, apakah angka yang akan dimasukkan memenuhi batasan dari teka-teki sudoku.

Pengecekan angka ini menggunakan fungsi isSafe. Fungsi ini menerima parameter berupa angka yang ingin dicek, posisi baris, serta posisi kolom. Fungsi ini akan mengembalikan *True* jika angka tersebut tidak ada yang sama dalam satu baris, kolom, serta blok berdasarkan kondisi saat ini.

```
def isSafe(self, num, posx, posy):
        #check row
        for j in range(9):
                if (self.table[posx][j] == num)
                and (j != posy):
                        return False
        #check column
        for i in range(9):
                if (self.table[i][posy] == num)
                and (i != posx):
                        return False
        #check block
        locx = posx / 3
        locy = posy / 3
        for i in range(3):
                for j in range(3):
                        if (self.table[locx*3+i]
                        [locy*3+j] == num)
                        and (locx*3+i != posx)
                        and (locy*3+j != posy):
                                return False
        return True

def solver(self):
        solved, pos = self.findEmpty()
        if (solved):
                return True
        self.cNode += 1
        for i in range(1,10):
                if (self.isSafe(i, pos[0], pos[1])):
                        self.table[pos[0]][pos[1]] = i
                        solved = self.solver()
                        if (solved):
                                return True
                        self.table[pos[0]][pos[1]] = 0
        return False
```

Gambar 4 Potongan kode python yang digunakan untuk menyelesaikan teka-teki sudoku

Jika angka tersebut bisa diletakkan di kotak tersebut, maka program akan melanjutkan penyelesaian dengan memanggil kembali fungsi *solver*. Akan tetapi, jika angka tidak dapat diletakkan di tempat tersebut maka akan dicoba angka yang lain. Jika pada akhirnya tidak ada angka yang bisa diletakkan pada kotak tersebut, maka program akan *backtrack* dan kembali ke kondisi sebelumnya.

```
PS E:\Kuliah\Semester 4\Stima> python .\sudoku.py
[0, 7, 0, 4, 0, 0, 0, 8, 1]
[0, 0, 6, 2, 5, 0, 4, 0, 0]
[0, 2, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 7, 9, 2]
[0, 0, 9, 1, 0, 2, 5, 0, 0]
[2, 8, 3, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 6, 0]
[0, 0, 7, 0, 3, 1, 9, 0, 0]
[1, 3, 0, 0, 0, 5, 0, 4, 0]

[9, 7, 5, 4, 6, 3, 2, 8, 1]
[3, 1, 6, 2, 5, 8, 4, 7, 9]
[8, 2, 4, 9, 1, 7, 3, 5, 6]
[4, 5, 1, 3, 8, 6, 7, 9, 2]
[7, 6, 9, 1, 4, 2, 5, 3, 8]
[2, 8, 3, 5, 7, 9, 6, 1, 4]
[5, 9, 8, 7, 2, 4, 1, 6, 3]
[6, 4, 7, 8, 3, 1, 9, 2, 5]
[1, 3, 2, 6, 9, 5, 8, 4, 7]
node visited : 3742
elapsed time : 0.0812066726073 second
```

Gambar 5 Tampilan program untuk menyelesaikan teka-teki Sudoku

V. Perbandingan Kecepatan Penyelesaian

Pada makalah ini, saya mencoba melakukan perbandingan mengenai kecepatan penyelesaian jika urutan pengisian dari kotak teka-teki sudoku ditentukan. Dalam hal ini, akan ada 3 cara dalam menentukan pengisian tabel.

Yang pertama, dengan mengisi kotak urut dari pojok kiri atas menuju pojok kanan bawah. Hal ini diatur dalam fungsi *findEmpty*, program hanya akan iterasi dari kiri atas ke kanan bawah. Ketika ditemukan tempat kosong, tempat itu akan dikembalikan oleh fungsi untuk diisi terlebih dahulu.

Yang Kedua, dengan mengisi kotak urut dari pojok kanan bawah menuju kiri atas. Cara ini berkebalikan dengan cara yang pertama. Pendekatannya sama seperti yang sebelumnya dengan cara iteratif.

Sedangkan yang terakhir, tabel akan diisi diurutkan berdasarkan kotak yang memiliki kandidat angka yang dapat di tempat tersebut paling sedikit. Hal ini diharapkan dapat mempercepat proses penyelesaian teka-teki sudoku.

Contohnya ketika ada kotak yang hanya dapat diisi dengan satu angka, pasti angka tersebut harus ada di tempat itu. Sehingga diharapkan dapat menghindari proses *backtracking* sampai ke kondisi awal.

Percobaan dilakukan dengan menjalankan *script python* untuk menyelesaikan teka-teki sudoku tertentu. Pada percobaan yang saya lakukan saya menggunakan 3 tabel sudoku yang diambil dari situs *https://www.websudoku.com/* . Dalam situs ini terdapat 4 tingkatan dari tabel sudoku, tingkatan yang digunakan dalah pengujian ini hanya tingkatan mudah, sedang, dan sulit.

Teka-teki yang digunakan untuk melakukan perbandingan ini memiliki tingkatan yang berbeda-beda. Agar dapat diketahui perbedaan waktu eksekusi dari tiap tingkatan. Tiap sample teka-teki sudoku akan dicoba diselesaikan dengan menggunakan 3 cara yang sebelumnya.

TABEL I HASIL EKSEKUSI SUDOKU SOLVER

Kesulitan sudoku / Metode	Waktu Eksekusi (detik)	Simpul dihasilkan
Mudah (kiri atas)	0.00850613946323	359
Mudah (kanan bawah)	0.017500490855	726
Mudah (jumlah kandidat)	0.00434746492275	208
Sedang (kiri atas)	0.0812066726073	3742
Sedang (kanan bawah)	0.0385932851037	1447
Sedang (jumlah kandidat)	0.124280331063	4898
Sulit (kiri atas)	0.244030070532	9667
Sulit (kanan bawah)	0.806982978659	20382
Sulit (jumlah kandidat)	0.00946859283201	428

Berdasarkan percobaan yang telah dilakukan, didapatkan hasil sesuai dengan tabel di atas. Setelah dianalisa, ternyata mengisi kotak yang memiliki kandidat paling sedikit belum tentu bisa mengurangi waktu untuk menyelesaikan teka-teki sudoku ini.

Setelah diteliti, ternyata hal ini terjadi dikarenakan angka yang dimasukkan ke kotak yang awal tidak tepat. Sehingga pada akhirnya program akan melakukan *backtrack* yang jauh, yaitu sampai kembali ke kondisi pertama.

Akan tetapi, jika pada saat memasukkan angka di kotak awal sudah tepat. Hal ini bisa mempercepat proses penyelesaian. Seperti pada hasil eksekusi dengan tingkat kesulitan sulit. Penyelesaian dengan pendekatan mengisi terlebih dahulu kotak yang memiliki kandidat angka paling sedikit memiliki waktu eksekusi paling cepat.

Hal ini dikarenakan program tidak perlu lagi melakukan *backtracking* sampai ke kondisi yang awal. Sebab, angka yang dimasukkan sudah benar. Sehingga untuk menyesuaikan angka yang lain bisa dilakukan dengan lebih cepat.

Selain itu, hasil yang didapatkan dengan pendekatan melalui posisi kanan atas dengan pendekatan posisi kanan bawah memberikan hasil yang berbeda. Hal ini disebabkan karena angka yang telah diisi di teka-teki sudoku ini tidak merata. Sehingga terkadang bisa lebih cepat dengan cara kiri atas, pada kondisi yang lain bisa pula lebih cepat di kondisi kanan bawah.

TABEL II WAKTU EKSEKUSI RATA-RATA SUDOKU SOLVER

Kesulitan sudoku / Metode	Waktu Eksekusi rata-rata (detik)
Mudah (kiri atas)	0.00176526559031
Mudah (kanan bawah)	0.00176065910498
Mudah (jumlah kandidat)	0.000984881666188
Sedang (kiri atas)	0.135745301277
Sedang (kanan bawah)	0.203108136973
Sedang (jumlah kandidat)	0.12156277442
Sulit (kiri atas)	2.77171678422
Sulit (kanan bawah)	0.645049916177
Sulit (jumlah kandidat)	6.70483091933

Setelah dilakukan percobaan yang lain untuk menghitung rata-rata waktu eksekusi dari tiap metode. Didapatkan rata-rata waktu eksekusi terkecil pada kesulitan mudah dan sedang adalah ketika menggukan metode pendekatan jumlah kandidat angka yang dapat dimasukkan. Waktu eksekusi rata-rata ini didapatkan dengan menyelesaikan 5 buah sample acak yang diambil dari *http://sudokugarden.de/en/download-pdf/*. Sampel yang sama digunakan di ketiga metode.

Akan tetapi, pada saat tingkat kesulitan sudoku dinaikkan menjadi sulit. Penggunaan metode berdasarkan jumlah kandidat tidak lagi efektif. Hal ini dikarenakan rata-rata jumlah kandidat tiap kotak hampir sama. Sehingga hampir tidak ada bedanya dengan melakukan iteratif dari kiri atas maupun kanan bawah.

VI. KESIMPULAN

Teka-teki sudoku memiliki banyak sekali metode penyelesaiannya. Salah satunya dapat menggunakan algoritma *backtracking*. Dengan menggunakan algoritma runut-balik juga memiliki banyak sekali metode pendekatan untuk penyelesaiannya. Seperti yang ada pada makalah ini, digunakan 3 cara untuk penyelesaian teka-teki sudoku.

Berdasarkan hasil yang didapat dari percobaan, penyelesaian teka-teki sudoku dengan mengisi terlebih dahulu kotak yang memiliki kandidat angka paling sedikit dapat mempercepat proses penyelesaian teka-teki sudoku.

Selain dari 3 cara di atas sebenarnya masih ada banyak sekali pendekatan untuk menyelesaikan teka-teki sudoku ini. Seperti dengan mengisi terlebih dahulu petak yang memiliki banyak pilihan, atau bisa juga dengan mengisi berurutan dengan kotak yang bersangkutan dengan kotak sebelumnya.

VII. Ucapan Terimakasih

Puji Syukur kepada Allah SWT, atas izin-Nya penulis dapat menyelesaikan makalah ini. Terima kasih kepada Ibu Drs. Nur Ulfa Maulidevi Selaku dosen pengajar Strategi Algoritma yang telah memberikan materi selama satu semester ini.

Daftar Pustaka

[1] http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2017-2018/Algoritma-Runut-balik-(2018).pdf
diakses pada 10 Mei 2018

[2] https://www.cis.upenn.edu/~matuszek/cit594-2002/Pages/backtracking.html
diakses pada 12 Mei 2018

[3] http://www.sudoku-space.com/sudoku.php
diakses pada 12 Mei 2018

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, 13 Mei 2018

Muhammad Abdullah Munir
13516074