Strategi *Brute Force* dan *Backtracking* untuk Menyelesaikan Permainan Sudoku Berukuran 4x4 dan 9x9

Aditya Prawira Nugroho - 13520049
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13520049@std.stei.itb.ac.id

***Abstract*—Sudoku adalah permainan logika yang mengharuskan kita mengisi matriks n^2 x n^2 dengan digit 1 hingga n yang hanya muncul tepat satu kali pada submatriks n x n, baris yang sama, dan kolom yang sama. Makalah ini bertujuan untuk menyelesaikan permainan sudoku menggunakan beberapa strategi algoritma seperti *brute force* dan *backtracking*. Selain itu, makalah ini bertujuan untuk melihat strategi algoritma yang paling efektif dan efisien di antara keduanya. Untuk menentukan algoritma paling efisien dan efektif, kedua algoritma akan diuji dengan cara menyelesaikan sudoku dengan variasi ukuran 4x4 dan 9x9 serta variasi tingkat kesulitan sudoku menggunakan algoritma *brute force* dan *backtracking*. Penulis akan menganalisis kompleksitas waktu dan ruang masing-masing strategi algoritma dengan hasil eksperimen sebagai dasar analisis.**

***Keywords*—sudoku; *brute force*; strategi algoritma; *backtracking*;**

I. Pendahuluan

Sudoku merupakan permainan puzzle dari Jepang yang berbasis logika dan merupakan puzzle kombinatorial penempatan angka. Pada umumnya, ukuran sudoku adalah 9 x 9, yang berarti di dalam permainan sudoku tersebut ada 9 submatriks berukuran 3x3 yang diisi dengan angka 1 hingga 9. Puzzle dan permainan logika dari sudoku berada pada cara penempatan angka 1 hingga 9 tersebut tanpa ada angka yang sama di baris, kolom, dan submatriks yang sama. Tidak ada urutan pengisian angka sehingga pemain bebas mulai mengisi dari tempat yang kosong manapun.

Sudoku pada umumnya diselesaikan tanpa adanya metode atau algoritma khusus. Orang yang bemain sudoku pada umumnya menyelesaikan dengan memilih sebuah angka, kemudian mengecek apakah angka tersebut dapat dimasukkan ke tempat yang kosong. Dalam tulisan ini, penulis akan menyelesaikan sudoku dengan sebuah program menggunakan strategi algoritma *brute force* dan *backtracking*. Penulis akan memberikan analisis kompleksitas waktu dan ruang serta hasil data percobaan untuk kedua strategi algoritma tersebut. Selain itu, penulis akan membahas alternatif strategi algoritma untuk menyelesaikan sudoku.

II. Landasan Teori

A. *Brute force*

Salah satu pendekatan/strategi untuk menyelesaikan permasalahan komputasi adalah *brute force*. Jika diartikan secara harfiah, *brute* berarti murni secara fisik, tidak cerdas. Sedangkan *force* berarti kekuatan, gaya. Jadi, dapat disimpulkan *brute force* berarti kekuatan yang murni fisik tanpa kecerdasan. Sehingga, dalam dunia pemrograman, penyelesaian secara *brute force* berarti memecahkan persoalan dengan pendekatan yang lempang (*straightforward)* untuk memecahkan suatu persoalan [1]. Pada umumnya, *brute force* didasarkan pada inti masalah dan konsep dari permasalahan yang dipecahkan.

Pendekatan yang lempang dari *brute force* mengimplikasikan bahwa solusi yang dibuat tidak memperhitungkan apapun, melainkan langsung diterapkan. Karena itu, pendekatan *brute force* adalah pendekatan termudah untuk diaplikasikan/diterapkan.

Meskipun pendekatan *brute force* bukan pendekatan yang cerdas atau efisien, *brute force* merupakan pendekatan yang dapat diterapkan pada variasi permasalahan yang sangat luas. Bahkan, lebih sulit untuk menemukan permasalahan yang tidak dapat diselesaikan oleh algoritma *brute force*. Selain itu, beberapa kegunaan dan kelebihan pendekatan *brute force* adalah pertama, beberapa masalah penting, seperti perkalian matriks, pencarian, dan pengurutan, dapat diselesaikan dengan pendekatan *brute force* dengan kompleksitas yang cukup baik untuk ukuran berapa pun. Kedua, perumusan solusi menggunakan pendekatan *brute force* jauh lebih murah dibandingkan dengan perumusan solusi dengan algoritma lainnya sehingga jika hanya ada beberapa masalah dan *brute force* mampu menyelesaikannya dengan waktu yang cukup baik maka *brute force* adalah solusi yang dapat diterima. Terakhir, algoritma *brute force* dapat dijadikan tolak ukur dan melihat algoritma mana yang lebih efisien dalam memecahkan suatu persoalan.

Kelemahan dari *brute force* adalah algoritma *brute force* jarang menghasilkan algoritma yang mangkus. Kedua, algoritma *brute force* umumnya lambat untuk masukan yang berukuran besar sehingga tidak dapat digunakan. Ketiga, tidak sekonstruktif/sekreatif strategi lainnya.

Contoh dari penerapan algoritma *brute force* yang sering kita aplikasikan tanpa sadar adalah pencarian sekuensial dan pencarian nilai maksimum/minimum. Algoritma *brute force* untuk pencarian sekuensial untuk sebuah masukan larik berukuran N adalah

1. Mulai dari elemen pertama/terakhir larik
2. Cek apakah elemen tersebut sama dengan yang dicari
3. Jika iya, pencarian berhenti
4. Jika tidak, pencarian dilanjutkan ke elemen selanjutnya
5. Ulangi hingga elemen ke-N / ditemukan elemen yang dicari

Skenario terburuk dari algoritma tersebut adalah dilakukan perbandingan sebanyak N kali dan skenario terbaik adalah dilakukan perbandingan sebanyak satu kali. Dengan begitu, kompleksitas waktu untuk pencarian sekuensial menggunakan *brute force* adalah O(n). Dapat dilakukan sebuah optimasi dengan mengurutkan elemen dalam larik terlebih dahulu sehingga ketika ditemukan elemen yang lebih besar/kecil atau sama, pencarian akan langsung berhenti.

Algoritma pencarian nilai maksimum/minimum menggunakan *brute force* hampir sama dengan pencarian sekuensial. Perbedaanya adalah diperlukan ruang tambahan untuk menyimpan nilai maksimum/minimum yang ditemukan sejauh ini.

B. Backtracking

Algoritma *backtracking* adalah algoritma yang memperbaiki algoritma sebelumnya, yaitu *exhaustive search*. Algoritma ini pertama kali diperkenalkan oleh D. H. Lehmer tahun 1950. Pada *exhaustive search*, akan dibuat dan dicari semua kemungkinan solusi kemudian diindentifikasi solusi terbaik dari semua kemungkinan solusi. Namun, *backtracking* sedikit lebih cerdas daripada itu, ide utama dalam *backtracking* adalah membangun solusi satu per satu. Jika solusi yang sedang dibangun tidak melanggar aturan dari permasalahan, maka solusi tersebut ditelusuri lebih lanjut. Jika tidak terdapat solusi pada jalan yang ditempuh, maka algoritma ini akan mundur dan mengubah solusi yang sedang dibangun dan melihat kemungkinan lainnya. Sehingga, algoritma *backtracking* akan memangkas kemungkinan yang tidak mengarah ke solusi [2]. Algoritma ini dapat dipandang sebagai salah satu dari dua hal berikut:

1. Sebagai sebuah tahapan/fase di dalam algoritma traversal Depth-First Search (DFS)
2. Sebagai sebuah metode pemecahan masalah yang mangkus, terstruktur, dan sistematis, baik untuk persoalan optimasi maupun non-optimasi

Algoritma *backtracking* memiliki beberapa properti yang menjadi karakteristik dari pendekatan runut-balik ini. Properti umum tersebut adalah

1. Solusi Persoalan

   Solusi dari persoalan dinyatakan sebagai vektor dengan *n-tuple*, yaitu sebuah *tuple* yang memiliki n buah elemen. Misalkan X adalah himpunan solusi yang terbentuk dan S adalah himpunan kandidat solusi yang mungkin, maka notasi penulisan solusi menjadi $X = (x_1, x_2, \ldots, x_n)$ dengan $x_i \in S_i$. Umumnya $S_1 = S_2 = ... = S_n$.

   Contohnya pada persoalan $1/0$ *knapsack* $S_i = \{0,1\}$ dengan $x_i = 0$ atau 1.

2. Fungsi pembangkit nilai x_k

   Dinyatakan sebagai predikat T(). Contoh, T(x[1], x[2], …, x[k-1]) membangkitkan nilai untuk x_k, yang merupakan komponen vektor solusi.

3. Fungsi pembatas (*bounding function)*

   Dinyatakan sebagai predikat $B(x_1, x_2, \ldots, x_k)$. Nilai dari B *true* jika $(x_1, x_2, \ldots, x_k)$ mengarah ke solusi, artinya tidak melanggar aturan dan *constraints* dari persoalan. Jika nilai dari B benar, maka pembangkitan nilai untuk x_{k+1} dilanjutkan, tetapi akan dibuang jika nilainya *false*.

Pada algoritma *backtracking*, semua kemungkinan solusi diorganisasi ddan disebut sebagai ruang solusi (*solution space*). Misalkan pada permasalahan urutan pengambilan pekerjaan dengan jumlah pekerjaan sebanyak 3, solusi persoalannya akan dinyatakan sebagai $X = (x_1, x_2, x_3)$ dimana $x_i \in \{1,2,3\}$ dan i adalah nomor pekerjaan. Maka akan terbentuk ruang solusi $\{(1,2,3), (3,2,1), (2,1,3), (1,3,2), (3,1,2), (2,3,1)\}$.

Ruang solusi diorganisasikan ke dalam bentuk struktur pohon berakar. Tiap-tiap simpul (*node*) pohon menyatakan status persoalan, sedangkan sisi dari pohon dilabeli dengan nilai-nilai x_i. Solusi yang mungkin akan ditunjukkan dengan lintasan dari akar ke daun. Seluruh pohon akan menjadi ruang solusi. Pengorganisasian pohon ruang solusi disebut sebagai pohon ruang status.

Algoritma *backtracking* memiliki prinsip untuk mencari solusi. Prinsip tersebut adalah

- Solusi dcari dengan membangkitkan simpul-simpul status sehingga terbentuk lintasan dari akar ke daun.
- Aturan urutan pembangkitan simpul mengikuti DFS (*depth-first search*).
- Simpul-simpul yang sudah dibangkitkan disebut simpul hidup.
- Simpul yang sedang diperluas dinamakan simpul ekspan.
- Jika simpul ekspan tidak mengarah ke solusi, simpul tersebut "dimatikan" sehingga menjadi simpul mati.
- Fungsi yang mematikan simpul ekspan disebut dengan fungsi pembatas (*bounding function*).
- Ketika simpul ekspan dimatikan, berarti kita telah memangkas simpul-simpul anaknya. Kemudian, pencarian akan *backtrack* ke simpul atasnya. Pembangkitan simpul diteruskan ke simpul anak yang lain dan simpul tersebut menjadi simpul ekspan yang terbaru.
- Pencarian berhenti ketika sudah mencapai *goal node*.

Berikut adalah skema umum algoritma *backtracking* versi rekursif dalam notasi algoritmik

```
procedure backtrack(input k: integer)
{I.S. x[1], x[2], …, x[k-1] sudah ditentukan
nilainya
F.S. x[1], x[2], …, x[k-1] merupakan solusi}
while k != 0 do
   if x[k] belum diekspan dan x[k] anggota
   T(x[1], x[2], …, x[k-1]) dan B(x[1], x[2],
   …, x[k]) = true then
       if (x[1], x[2], …, x[k]) adalah lintasan
       dari akar ke simpul solusi then
              write(x[1], x[2], …, x[k])
       k <- k + 1
   else
       k <- k + 1
```

Setiap simpul dalam pohon ruang status berasosiasi dengan sebuah pemanggilan rekursif. Jika jumlah simpul dalam pohon ruang status adalah 2^n atau n!, maka pada *worst case scenario*, algoritma *backtracking* membutuhkan waktu dalam $O(p(n)2^n)$ atau O(q(n)n!) dengan p(n) dan q(n) adalah polinom derajat n yang menyatakan waktu komputasi tiap simpul.

C. *Sudoku*

Sudoku adalah sebuah puzzle yang berasal dari Jepang. Sudoku merupakan bahasa Jepang yang berarti satu digit. Aslinya, permainan sudoku disebut sebagai **Number Place** [3]. Permainan sudoku adalah sebuah permainan puzzle yang berbasis logika dan kombinatorial penempatan angka. Pada permainan sudoku yang klasik, tujuan dari permainan adalah untuk mengisi sebuah matriks 9 x 9 dengan digit sehingga setiap kolom, baris, dan sembilan submatriks berukuran 3 x 3 yang membentuk matriks 9 x 9 memiliki semua digit 1 hingga 9. Penyedia puzzle akan mengisi sebagian kotak yang kosong sedemikian sehingga puzzle hanya memiliki satu solusi.

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 1 Sebuah sudoku berukuran 9x9 (Sumber: 1200px-Sudoku_Puzzle_by_L2G-20050714_standardized_layout.svg.png (1200×1200) (wikimedia.org))

Sudoku memiliki banyak sekali varian setelah puzzle ini menyebar ke seluruh dunia. Pertama, sudoku divariasikan berdasarkan ukuran matriks atau bentuk submatriks. Salah satu variasi sudoku berdasarkan ukuran matriks adalah sudoku berukuran 4 x 4 dengan submatriks berukuran 2 x 2. Selain itu, terdapat pula variasi sudoku pada bentuk submatriksnya, misal sebuah sudoku berukuran 5 x 5 dengan bentuk submatriks pentomino.

	1	3	
2			
			3
	2	1	

Gambar 2 Variasi sudoku berukuran 4 x 4 (Sumber: sudoku-kids-4x4-02.png (225×225) (sudokuweb.org))

Kedua, terdapat varian sudoku dimana diberikan tambahan *constraint*, yaitu angka pada diagonal juga harus unik. Sering kali tambahan *constraint* tersebut dalam bentuk dimensi ekstra.

Varian sudoku lainnya adalah *alphabetical* sudoku, terkadang disebut sebagai Wordoku. Aturannya hampir sama dengan sudoku klasik, hanya saja digit 1 hingga 9 bisa diganti menjadi 9 buah alfabet sembarang, misal BHDJUILQP. Selain itu, terdapat varian sudoku dimana ukurannya sama dengan sudoku klasik, tetapi terdapat tambahan 4 submatriks berukuran 3 x 3 dimana digit 1 hingga 9 harus muncul tepat 1 kali. Varian tersebut dinamakan Hyper Sudoku/Windoku.

Sudoku memiliki banyak properti dan sifat matematis yang menarik karena basisnya yang kombinatorial dan logis. Pembentukan solusi sudoku berukuran n^2 x n^2 dengan submatriks berukuran n x n diketahui masuk ke dalam kategori NP-Complete. Meskipun banyak algoritma penyelesaian, seperti *backtracking* dan *dancing links*, yang mampu menyelesaikan sudoku klasik secara efisien, tetap saja kombinatorial akan meningkat pesat ketika n meningkat. Oleh karena itu, sudoku yang dapat dibuat, dianalisis, dan diselesaikan terbatas ketika n meningkat.

Pada umumnya, tingkat kesulitan sudoku hanya dibagi menjadi mudah, sedang, dan sulit. Dalam penentuan tingkat kesulitan sudoku, tidak ada konvensi dan cara yang benar. Meskipun begitu, jumlah petunjuk minimal atau kotak yang terisi dalam sudoku klasik agar sudoku dapat diselesaikan adalah 17 [4]. Dalam makalah ini, klasifikasi tingkat kesulitan sudoku akan didasarkan pada jumlah petunjuk yang dimiliki dan teknik serta logika matematik yang harus digunakan.

III. Studi Kasus Sudoku

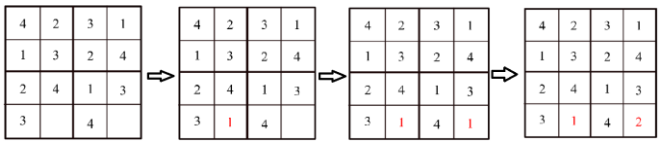
A. *Penyelesaian Sudoku dengan Strategi Brute Force*

Salah satu strategi algoritma yang paling intuitif dan paling mudah diterapkan untuk menyelesaikan puzzle sudoku adalah *brute force*. Algoritma *brute force* yang digunakan untuk menyelesaikan puzzle sudoku kali ini tidak memiliki heuristik, sehingga akan dicoba hampir semua kemungkinan yang bisa dan berhenti ketika solusi sudah ditemukan. Tahapan algoritma *brute force* rekursif untuk menyelesaikan sudoku berukuran n x n adalah sebagai berikut

1. Basis dari rekursi adalah ketika semua kotak sudah terisi penuh.
2. Cek apakah dia basis.
3. Jika iya, cek apakah dia solusi.
4. Jika bukan, lakukan pengulangan sebanyak n kali.
5. Isi kotak yang kosong dengan pengulangan ke berapa.
6. Panggil rekursi.
7. Jika rekursi menghasilkan false, isi kotak tersebut dengan 0.

Jika dilihat sekilas dari tahapannya, strategi tersebut hampir sama dengan strategi *exhaustive search*. Namun, pada *exhaustive search*, langkah yang ditempuh seharusnya mencari semua kemungkinan terlebih dahulu, kemudian dievaluasi satu-satu untuk mencari solusinya. Sedangkan pada algoritma tersebut, tidak dicari semua kemungkinan terlebih dahulu, melainkan langsung dilakukan pengecekan solusi ketika semua kotak sudah diisi.

Visualisasi tahapan algoritma tersebut kira-kira sebagai berikut



Gambar 3 Penyelesaian sudoku 4x4 dengan strategi brute force (Sumber: koleksi pribadi)

B. *Penyelesaian Sudoku dengan Strategi Backtracking*

Strategi algoritma yang bisa menyelesaikan sudoku secara efektif dan efisien melebihi *brute force* adalah *backtracking*. Pada dasarnya, algoritma ini merupakan perbaikan dari algoritma *brute force*. Pemetaan persoalan ke dalam algoritma *backtracking* adalah sebagai berikut

1. Solusi dari persoalan sudoku dinyatakan dalam bentuk matriks berukuran n x n.
2. Ruang solusi dari permasalahan adalah semua kemungkinan kotak yang diisi dengan angka 1 hingga n.
3. Simpul dari pohon ruang status adalah puzzle sudoku dengan kotak yang sudah diisi.
4. Sisi dari pohon ruang status adalah kotak kosong yang diisi dengan salah satu angka 1 hingga n.
5. Fungsi pembangkit dari permasalahan adalah fungsi yang mengisi kotak dengan salah satu angka 1 hingga n.
6. Fungsi pembatas atau *bounding function* permasalahan adalah fungsi yang mengecek tiap kotak puzzle agar tidak melanggar aturan, yaitu angka tersebut muncul tepat satu kali dalam baris, kolom, dan submatriks yang sama.

Berdasarkan hasil pemetaan persoalan, algoritma *backtracking* lebih baik dibuat secara rekursif karena struktur ruang solusi dari persoalan berbentuk pohon dan dasar urutan pembangkitan adalah DFS. Tahapan algoritma *backtracking* untuk pencarian solusi sudoku adalah sebagai berikut

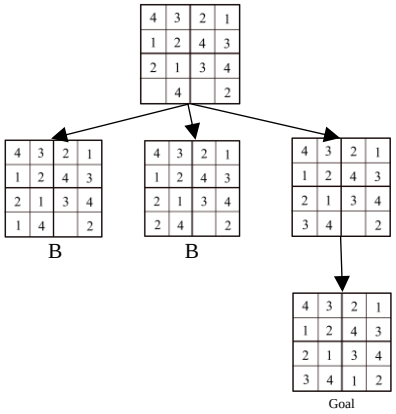
1. Basis dari rekursi adalah ketika semua kotak terisi penuh.
2. Jika belum penuh, cari indeks kotak yang kosong.
3. Sebelum mengisi kotak dengan angka i, panggil fungsi pembatas untuk mengecek apakah i menuju solusi.
4. Jika tidak, mundur ke simpul atasnya dan bangkitkan anak yang lain.
5. Jika iya, lakukan rekursi.
6. Lakukan langkah ke-2 hingga ke-5 sebanyak n-kali.

Supaya lebih jelas, berikut adalah potongan kode pencarian solusi menggunakan *backtracking* dalam bahasa C++

```
bool solveBacktracking(){
    int i , j ;
    // Basis ketika sudah tidak ada yang kosong
    if (!isThereEmptySpace(i, j)){
        return true;
    } else {
        // Jika masih ada slot kosong, akan diisi
dengan angka 1 sampai n
        for (int k = 1; k <= n; k++){
            // Cek apakah penempatan valid, jika
iya, lakukan rekursi
            if (isValidBacktrack(i, j, k)){
                set(i, j, k);
                *simpulDibangkitkan += 1;
                if
(solveBacktracking(simpulDibangkitkan)){
                    return true;
                } else {
                    // Kembalikan lagi ke 0 karena
itu berarti tidak mengarah ke solusi
                    set(i, j, 0);
                    subMatrixFilled[getSubmatrix(i,
j)][k - 1] = false;
                }
            }
        }
    }
    return false;
}
```

Gambar 4 Potongan kode algoritma backtracking dalam bahasa C++ (Sumber: Koleksi pribadi)

Dari potongan kode di atas, fungsi pembatas adalah isValidBacktrack(int i, int j, int k) yang akan mengecek apakah angka k bila diletakkan pada kotak indeks baris i kolom j tidak melanggar *constraint*. Berikut adalah visualisasi algoritma *backtracking* dalam penyelesaian sudoku 4 x 4 dengan tingkat kesulitan mudah

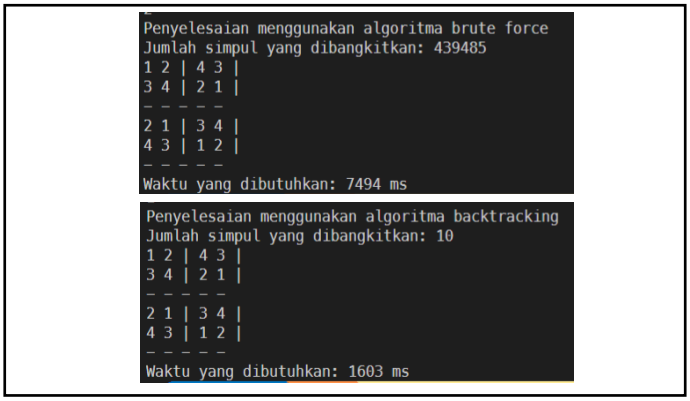


Gambar 5 Ilustrasi algoritma backtracking untuk menyelesaikan sudoku 4x4 (Sumber: koleksi pribadi)

C. *Hasil Pengujian*

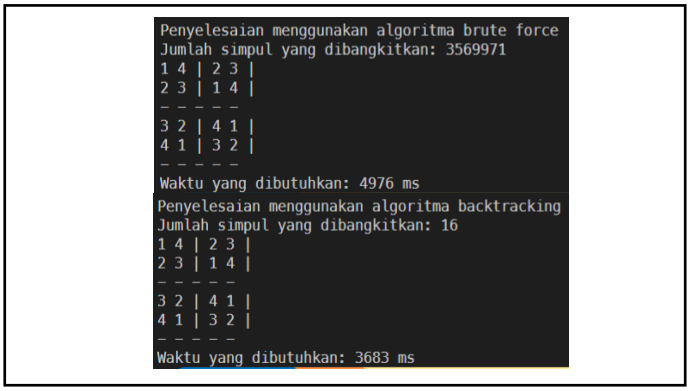
Algoritma *backtracking* dan *brute force* akan diuji untuk menyelesaikan sudoku dengan beragam variasi, yaitu ukuran 9x9, 4x4, serta tingkat kesulitan mudah dan tinggi.

Variasi pertama adalah sudoku ukuran 4x4 dengan tingkat kesulitan mudah



Gambar 6 Hasil pengujian algoritma dalam menyelesaikan sudoku 4x4 mudah (Sumber: koleksi pribadi)

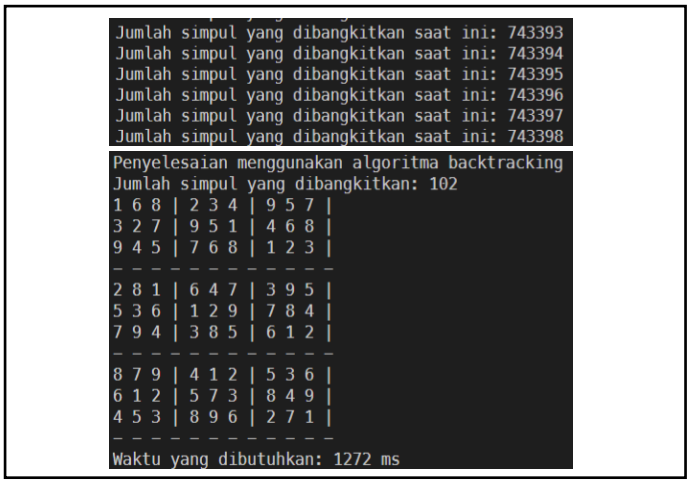
Dari hasil terlihat bahwa algoritma *backtracking* jauh lebih unggul dalam segi waktu dan ruang dibandingkan *brute force* untuk sudoku ukuran 4x4 tingkat mudah. Namun, bagaimana dengan tingkat yang sulit?



Gambar 7 Hasil pengujian algoritma untuk sudoku 4x4 sulit (Sumber: koleksi pribadi)

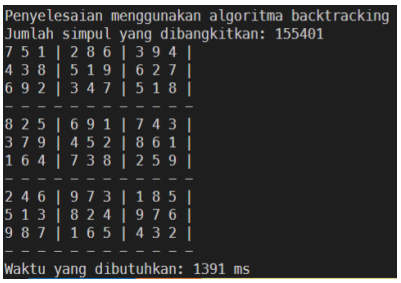
Ternyata perbandingan yang dihasilkan hampir sama dengan hasil pengujian untuk sudoku 4x4 tingkat mudah. Akan tetapi, perbandingan waktu antara kedua algoritma kali ini cukup dekat. Hal tersebut bisa memiliki beberapa makna, pertama, bisa saja kondisi perangkat keras pada komputer sedang tidak optimal, misalnya dalam mode hemat baterai. Kedua, untuk tingkat sulit, algoritma *backtracking* memang membutuhkan waktu yang lebih lama untuk membangkitkan simpul. Meskipun begitu, algoritma *backtracking* tetap jauh lebih unggul.

Variasi selanjutnya adalah sudoku ukuran 9x9 dengan tingkat kesulitan mudah



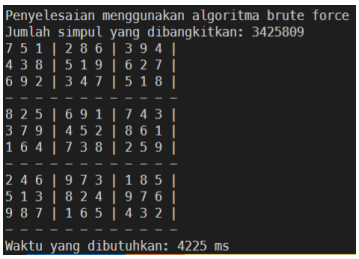
Gambar 8 Hasil pengujian algoritma brute force (atas) dan backtracking (bawah) untuk sudoku 9x9 tingkat mudah (Sumber: koleksi pribadi)

Untuk sudoku ukuran 9x9, algoritma *brute force* berjalan dengan sangat buruk. Setelah dijalankan lebih dari 5 menit, solusi masih belum ditemukan sehingga penulis menghentikan program secara paksa. Sepertinya, penggunaan rekursi dengan strategi *brute force* sangat memakan waktu. Penyebab lain yang mungkin adalah algoritma *brute force* melakukan pengecekan validitas semua kotak dalam puzzle, hal tersebut menambah operasi yang cukup signifikan dan menyebabkan redundansi karena kotak petunjuk seharusnya tidak perlu dicek. Meskipun demikian, algoritma *backtracking* masih berjalan dengan sangat baik. Bahkan, waktu yang dibutuhkan kurang lebih sama dengan puzzle sudoku ukuran 4x4. Namun, bagaimana dengan tingkat yang lebih sulit?



Gambar 9 Hasil pengujian sudoku ukuran 9x9 tingkat sulit dengan algoritma backtracking (Sumber: koleksi pribadi)

Ternyata, algoritma *backtracking* masih berjalan dengan sangat baik. Bukti tingkat kesulitan sudoku dapat dilihat dari jumlah simpul yang dibangkitkan yang jauh lebih banyak dibandingkan dengan tingkat yang mudah. Meskipun demikian, waktu yang dibutuhkan kurang lebih masih sama. Untuk algoritma *bruteforce*, masih berjalan dengan sangat buruk dan hasilnya masih tidak kunjung keluar sehingga gambar tidak ditunjukkan. Kemudian, penulis mencoba menguji algoritma *bruteforce* untuk menyelesaikan sudoku 9x9 dengan jumlah petunjuk 70 sehingga kotak yang kosong hanyalah 11 buah dan ternyata hasilnya masih tetap sama. Setelah menambah petunjuk menjadi 74, algoritma *bruteforce* berhasil menemukan solusi.



Gambar 10 Hasil pengujian algoritma brute force dengan jumlah petunjuk 74 (Sumber: koleksi pribadi)

Sepertinya, penambahan ukuran sudoku dan jumlah petunjuk yang ada sangat memengaruhi performa algoritma *brute force*. Hal ini dibuktikan oleh hasil algoritma *brute force* yang cukup baik dalam menyelesaikan sudoku ukuran 4x4 dengan tingkat kesulitan yang beragam, tetapi hasil yang buruk untuk sudoku berukuran 9x9. Oleh karena itu, mari kita coba lihat kompleksitas waktu dan ruang masing-masing algoritma.

D. Analisis Kompleksitas Waktu dan Ruang

Dalam menganalisis kompleksitas waktu, kita menghitung berapa banyak operasi yang dilakukan oleh algoritma. Sedangkan dalam analisis kompleksitas ruang, kita menghitung seberapa besar ruang yang diperlukan oleh algoritma.

Untuk algoritma *brute force*, dilakukan pengisian kotak sebanyak n kali untuk m kotak kosong. Selain itu, dilakukan pengecekan apakah angka pada masing-masing kotak muncul tepat satu kali dalam baris, kolom, dan submatriks yang sama sehingga pengecekan dilakukan sebanyak n^3 untuk n^2 elemen. Kemudian, dalam mencari kotak kosong, program akan mencari sebanyak n^2 kali karena dilakukan iterasi satu-satu. Sehingga

kompleksitas waktu dalam notasi Big-O untuk algoritma *brute force* adalah

$$O(n) = O(nm) + O(n^3 \text{ x } n^2) + O(n^2) = O(nm) + O(n^5)$$

Pada skenario terburuk, nilai dari m adalah n^2 dan untuk skenario terbaik, nilai dari m adalah 1. Dari analisis, terlihat bahwa pengecekan memakan cukup banyak waktu, seharusnya, jika pengecekan hanya dilakukan untuk kotak yang kosong, kompleksitas waktu bisa menjadi $O(n^3m)$ dengan m adalah jumlah kotak yang kosong.

Untuk algoritma *backtracking*, dilakukan pengisian m kotak yang kosong sebanyak n kali. Kemudian, pengecekan hanya dilakukan untuk kotak yang kosong sebanyak n^3 x m kali. Selain itu, algoritma melakukan iterasi satu-satu untuk mencari kotak yang kosong sebanyak n^2. Maka, kompleksitas waktu dalam notasi Big-O untuk algoritma *backtracking* adalah

$$O(n) = O(nm) + O(n^3m) + O(n^2) = O(n^3m)$$

Didapatkan bahwa kompleksitas waktu algoritma *backtracking* lebih baik dibandingkan dengan algoritma *brute force* meskipun perbedaanya tidak terlalu jauh. Kompleksitas waktu kedua algoritma akan sama untuk kasus terburuk. Lalu, bagaimana dengan kompleksitas ruang kedua algoritma?

Kali ini, kedua algoritma menggunakan rekursif dan sebuah matriks berukuran n x n yang berisi integer. Karena itu, kompleksitas ruang kedua algoritma sama, yaitu O(n). Kompleksitas ruang O(n) didapatkan dari ukuran matriks n x n yang akan selalu konstan dan akan bertumbuh sebanyak n kali pemanggilan sehingga kompleksitas ruang yang didapatkan adalah

$$O(n) = O(1) \text{ x } O(n) = O(n)$$

IV. Kesimpulan

Puzzle sudoku merupakan puzzle yang membutuhkan kematangan konsep matematika dan logika yang kuat. Strategi algoritma yang lebih efektif dan efisien untuk menyelesaikan puzzle sudoku ukuran 4x4 dan 9x9 adalah algoritma *backtracking* dengan kompleksitas ruang O(n) dan kompleksitas waktu $O(n^3m)$. Algoritma *brute force* memiliki performa yang sangat buruk untuk sudoku ukuran 9x9, tetapi performanya cukup baik untuk sudoku berukuran 4x4 dan 9x9 jika jumlah petunjuk yang disediakan sebanyak 74 atau lebih. Selain itu, dengan sedikit modifikasi pada algoritma *brute force*, dapat dihasilkan algoritma yang jauh lebih baik. Kemudian, kompleksitas waktu yang lebih baik tidak berarti kompleksitas ruang yang dimiliki algoritma akan lebih baik juga, hal ini dibuktikan dengan kompleksitas ruang algoritma *backtracking* sama dengan algoritma *brute force* meskipun kompleksitas waktu *backtracking* lebih baik.

Link Video Youtube dan Github

Berikut link youtube untuk penjelasan makalah https://youtu.be/pdPyEnrdiMo. Berikut link github untuk *source code* https://github.com/Adityapnn811/Makalah-stima-sudoku-solver.

Ucapan Terima Kasih

Penulis bersyukur kepada Tuhan Yang Maha Esa yang telah melimpahkan rahmat dan rezeki-Nya sehingga penulis mampu menyelesaikan makalah strategi algoritma yang berjudul "Strategi Brute Force dan Backtracking untuk Menyelesaikan Permainan Sudoku Berukuran 4x4 dan 9x9". Penulis mengucapkan terima kasih kepada orang tua, sahabat, dan teman-teman HMIF yang telah memberikan dukungan dan bantuan terhadap penyelesaian tugas makalah. Penulis berterima kasih kepada dosen-dosen mata kuliah IF2211 Strategi Algoritma Semester II 2021/2022, yaitu Bapak Rinaldi Munir, Ibu Masayu Leylia Khodra, dan Ibu Nur Ulfa Maulidevi atas ilmu dan pengalaman yang sudah diberikan untuk penulis.

References

[1] Munir, Rinaldi. 2022. "Algoritma *Brute Force* (Bagian 1)". http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Brute-Force-(2022)-Bag1.pdf (Diakses pada 14 Mei 2022, Pukul 16.44 WIB).

[2] Munir, Rinaldi. 2022. "Algoritma Runut-balik (*Backtracking*) (Bagian 1)". http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-backtracking-2021-Bagian1.pdf (Diakses pada 14 Mei 2022, Pukul 16.44 WIB).

[3] Grossman, Lev. 2013. "The Answer Men". https://web.archive.org/web/20130301013815/http://www.time.com/time/magazine/article/0,9171,2137423,00.html (Diakses pada 20 Mei 2022, Pukul 10.58).

[4] G. McGuire, B. Tugemann, G. Civario. 2012. "There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem". [1201.0749] There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem (arxiv.org) (Diakses pada 20 Mei 2022, Pukul 11.53).

[5] Levitin, A. 2003. "Introduction to The Design & Analysis of Algorithms". New Jersey: Pearson Education Inc.

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, 1 Mei 2022

Aditya Prawira Nugroho 13520049