Visualisasi Algoritma *Backtracking* dalam Permainan Sudoku dengan Aturan Tambahan (Excalibur Sudoku)

Arsa Izdihar Islam - 13521101
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13521101@std.stei.itb.ac.id

***Abstract*—Algoritma *backtracking* merupakan salah satu algoritma yang banyak digunakan dalam penyelesaian *puzzle* termasuk Sudoku yaitu permainan yang cukup populer dengan aturan utama yaitu mengisi papan terbagi dalam 9 baris dan 9 kolom dengan total 81 sel. Pada makalah ini, Sudoku yang akan diselesaikan merupakan Sudoku dengan aturan tambahan khusus, yang disebut sebagai Excalibur Sudoku atau Arrow Sudoku, memiliki aturan sama dengan Sudoku pada umumnya dengan beberapa panah dengan jumlah angka pada garis panah harus sama dengan angka pada ujung panah. Langkah-langkah dalam pencarian solusi dari *puzzle* akan divisulisasikan secara rinci. Perancangan algoritma dalam pencarian solusi Excalibur Sudoku ini diharapkan dapat memberikan wawasan yang mendalam dalam algoritma *backtracking*.**

***Keywords*—backtracking; Sudoku; puzzle; algorithm; Excalibur Sudoku; visualization.**

I. PENDAHULUAN

Sudoku merupakan salah satu *puzzle* yang cukup sederhana dan banyak diketahui. Sudoku mulai populer di Jepang pada tahun 1986 dan telah mendunia pada tahun 2005. Hal yang membuat permainan ini cukup diminati adalah aturannya yang sederhana, tetapi membutuhkan pemikiran yang cukup kompehensif.

Pada permainan ini, pemain diharuskan untuk mengisi angka 1 hingga 9 ke dalam kotak-kotak pada papan permainan yang terbagi menjadi 9 bagian kecil. Aturan utama pada permainan ini adalah setiap angka pada baris, kolom, dan kotak kecil pada papan tidak boleh berulang.

Dengan aturan yang cukup sederhana tersebut, Sudoku cukup banyak diteliti dalam bidang komputasi dan *Artificial Intelligence.* Akan tetapi, banyak muncul variasi-variasi baru dalam permainan ini dengan aturan-aturan yang cukup unik. Salah satunya adalah yang disebut Excalibur Sudoku atau Arrow Sudoku. Variasi ini dibuat oleh pengembang dengan nama Logic Wiz. Permainan Sudoku dengan berbagai variasi yang dibuat oleh Logic Wiz tersebar di aplikasi pada berbagai *platform* dengan nama aplikasi “Sudoku & Variants by Logic Wiz”.

Excalibur Sudoku memiliki aturan dasar yang sama dengan Sudoku pada umumnya. Perbedaannya adalah pada variasi ini ditambahkan panah-panah yang masing-masing dari panah tersebut memiliki dua bagian. Jumlah angka pada bagian garis panah harus sama dengan angka pada ujung panah.

Program untuk menemukan solusi dari permainan Sudoku dapat diimplementasikan dengan berbagai algoritma yang salah satunya adalah algoritma *backtracking*. Penulis akan mengimplementasikan pencarian solusi Excalibur Sudoku dengan algoritma *backtracking* yang tidak naif dan cukup mangkus sebab pada algoritma yang dibuat, akan diaplikasikan beberapa optimasi heuristik untuk membuat algoritma ini lebih cepat mengarah ke solusi.

II. DASAR TEORI

A. *Algoritma*

Algoritma merupakan urutan langkah-langkah dalam memecahkan suatu masalah komputasional untuk mencapai suatu tujuan. Dalam algoritma, selain parameter utama yaitu keberhasilan suatu algoritma dalam mencapai masalah yang terkait, terdapat beberapa parameter lain yang menunjukkan seberapa bagus suatu algoritma. Salah satu parameter lain tersebut adalah seberapa efisien algoritma tersebut dalam mendapatkan solusi yang diinginkan dan juga banyak *resource* yang dibutuhkan untuk menjalankan algoritma tersebut. Maka dari itu, perlu analisis algoritma sebagai upaya untuk mengukur parameter-parameter tersebut. Secara umum, parameter untuk kemangkusan algoritma dibagi menjadi dua, yaitu:

1. Kompleksitas waktu, $T(n)$ yaitu jumlah tahapan komputasi (operasi) yang dibutuhkan dalam menjalankan algoritma sebagai fungsi dari ukuran masukan n.
2. Kompleksitas ruang, $S(n)$ yaitu ruang memori yang dibutuhkan algoritma sebagai fungsi dari ukuran n.

Selain itu, terdapat tiga notasi asimptotik kebutuhan waktu algoritma yaitu:

1. $O(f(n))$: batas lebih atas kebutuhan waktu algoritma
2. $\Omega(g(n))$: batas lebih bawah kebutuhan waktu algoritma
3. $\Theta(h(n))$: jika dan hanya jika $O(h(n)) = \Omega(h(n))$

Beberapa contoh bentuk strategi algoritma antara lain yaitu:

1. Algoritma *Brute-Force*
2. Algoritma *Greedy*
3. Algoritma *Divide and Conquer*
4. Algoritma *Decrease and Conquer*
5. Algoritma *Backtracking*
6. Algoritma *Branch and Bound*
7. *Dynamic Programming*

B. Algoritma Backtracking (Runut Balik)

Algoritma *backtracking* merupakan algoritma rekursif dalam mencari satu atau banyak solusi dari suatu permasalahan. Algoritma ini diperkenalkan oleh D. H. Lehmer pada tahun 1950. Cara kerjanya adalah dengan secara progresif membangun kandidat solusi dan mengabaikan kandidat yang dianggap tidak menuju kepada solusi. Algoritma ini dapat dipandang sebagai sebuah fase dalam algoritma traversal DFS (*Depth First Search)* yaitu algoritma yang bercabang. Algoritma ini disebut *backtracking* karena jika suatu cabang rekursif tidak mengarah kepada solusi yang valid, maka algoritma akan melakukan "*backtrack"* atau mundur ke langkah sebelumnya untuk mencoba opsi lain.

Algoritma ini merupakan bentuk perbaikan dari algoritma *brute-force* yaitu *exhaustive search.* Alih-alih mengekplorasi keseluruhan kemungkinan solusi yang ada, pada *backtracking* hanya pilihan yang mengarah ke solusi yang akan dieksplorasi.

Backtracking memiliki beberapa properti umum yaitu:

1. Solusi persoalan

   Solusi persoalan dapat dinyatakan sebagai vektor dengan *n-tuple*: $X = (x_1, x_2, \ldots, x_n)$ dengan $x_i \in S_i$. Pada umumnya, $S_1 = S_2 = \ldots = S_n$.

2. Fungsi pembangkit nilai x_k

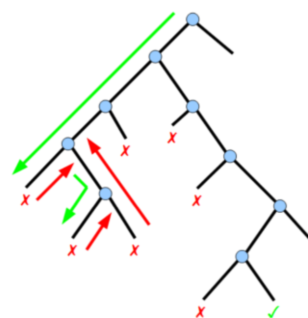
   Fungsi pembangkit ini dapat dinyatakan dengan predikat T() dengan $T(x_1, x_2, \ldots, x_{k-1})$ membangkitkan nilai untuk x_k, yang merupakan komponen vektor solusi

3. Fungsi pembatas (*bounding function)*

   Properti ini dinyatakan sebagai predikat $B(x_1, x_2, \ldots, x_k)$ dengan B akan bernilai *true* apabila $(x_1, x_2, \ldots, x_k)$ mengarah ke solusi permasalahan. Jika nilai dari B bernilai *false*, maka cabang rekursif tersebut akan dibuang dan akan runut balik (*backtrack*).

Terdapat beberapa prinsip dalam pencarian solusi menggunakan algoritma *backtracking* yaitu:

1. Solusi dicari dengan membuat simpul-simpul status yang akan menghasilkan lintasan hingga ke daun.
2. Aturan pembangkitan simpul akan mengikuti aturan DFS (*depth first search*).
3. Simpul-simpul yang sudah dibangkitkan dapat disebut sebagai simpul hidup (*live node).*
4. Simpul yang sedang diperluas dapat disebut sebagai simpul-E (*expand node*).
5. Setiap simpul-E diperluas, lintasan yang dibangun akan semakin panjang.
6. Jika lintasan yang dibentuk tidak mengarah ke solusi, maka simpul-E yang bersangkutan dapat "dimatikan" sehingga menjadi simpul mati (*dead node).*
7. Jika pembentukan lintasan berakhir pada simpul mati, maka proses dari pencarian solusi akan *backtrack* ke simpul *parent*-nya.
8. Pencarian akan dihentikan apabila telah ditemukan *goal node* (apabila hanya satu solusi yang dicari) atau hingga tidak ada lagi simpul-E yang masih hidup (apabila dicari semua solusi yang mungkin).



Gambar 1. Gambaran Umum Proses Pencarian Solusi pada Algoritma *Backtracking*

(Sumber: https://www.w3.org/2011/Talks/01-14-steven-phenotype/)

C. Sudoku

Dalam permainan Sudoku, umumnya terdapat 81 sel (9 baris dan 9 kolom) dan terbagi menjadi 9 kotak yang lebih kecil yang masing-masingnya mengandung 9 sel yang dapat disebut sebagai *subgrids.* Permainan ini diawali dengan beberapa sel sudah terisi dan tidak dapat diubah. Pemain harus mengisi seluruh sel yang masih kosong sehingga tidak ada angka yang muncul lebih dari sekali pada baris, kolom, atau *subgrid* yang sama. Setiap *puzzle* pada Sudoku memiliki satu solusi yang unik.

Banyak matematikawan yang tertarik dengan Sudoku dan mempertanyakan berbagai hal seperti berapa banyak "solusi" yang dapat dibangun. Pada saat ini, logika dan komputer memungkinkan kita untuk mengestimasi banyaknya solusi valid dari Sudoku yaitu sebanyak 6.670.903.752.021.072.936.960.

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

Gambar 2. Contoh *Puzzle* Sudoku
(Sumber: https://sudoku-puzzles.net/sudoku-easy/)

Program dalam mencari solusi unik Sudoku dapat dicapai dengan berbagai metode, salah satunya adalah *backtracking*. Algoritma *backtracking* dasar dapat dilakukan dengan cara sebagai berikut. Program akan mengisi sel pertama yang kosong dengan angka satu. Apabila angka satu tersebut memenuhi *constraint*, maka akan lanjut ke sel kosong berikutnya yang juga akan dimulai dengan diisi angka satu. Ketika terdapat konflik pada saat mengisi suatu sel kosong, maka angka terakhir yang diisi akan di-*increment* (dari satu ke dua) hingga angka sembilan. Apabila seluruh angka yang diisi tidak valid, maka program akan *backtrack* dan kembali ke sel sebelumnya untuk *increment* sel tersebut juga. Pencarian ini akan terus dilakukan hingga menemukan satu solusi yaitu berhasil mengisi seluruh sel kosong tanpa melanggar *constraint.*

Selain secara algoritma, terdapat suatu cara untuk memecahkan permainan Sudoku yaitu dengan cara membuat *"notes"* atau sebuah catatan kecil pada setiap sel kosong. Catatan ini dapat berisikan angka-angka yang mungkin yang dapat diisikan pada sel tersebut tanpa melanggar aturan Sudoku. Dengan melakukan hal ini, Dapat terlihat sel-sel kosong yang memiliki solusi yang sudah jelas (contoh paling sederhananya adalah pada suatu sel kosong hanya mungkin diisikan satu angka saja, maka angka tersebut pasti merupakan solusi untuk sel tersebut). Banyak pola-pola yang umum dalam menggunakan catatan ini supaya bisa mendeduksi solusi yang tepat tanpa mencoba-coba angka seperti *Swordfish.*

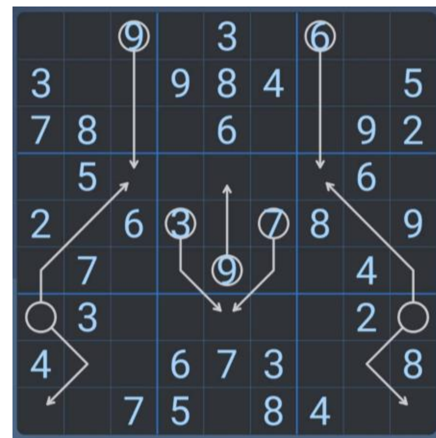
5		9	2				1	
4			1				8	5
		1	8		5		9	
7	5	6						1
		3		1	6	8		
			3	7		5		6
	7				4	2		8
3	6			5				
	4			8		3	5	

Gambar 3. Memecahkan Sudoku dengan Menggunakan *Notes*
(Sumber: https://www.learn-sudoku.com/pencil-marks.html)

D. Excalibur Sudoku

Excalibur Sudoku merupakan salah satu variasi tambahan dari permainan Sudoku. Variasi ini dikembangkan oleh Logic Wiz pada aplikasi "Sudoku & Variants by Logic Wiz". Dalam aplikasi ini, terdapat berbagai macam variasi Sudoku yang salah satunya adalah Excalibur Sudoku.

Pada Excalibur Sudoku, terdapat satu aturan tambahan dari Sudoku biasa yaitu terdapat beberapa panah pada papan permainan. Masing-masing panah ini terdiri dari dua bagian. Angka pada bagian ujung panah nilainya harus sama dengan jumlah dari angka-angka pada bagian garis panah. Dengan adanya aturan tambahan ini, tentunya pengisian sel kosong pada Excalibur Sudoku lebih terestriksi dibanding Sudoku biasanya.



Gambar 4. Contoh *Puzzle* Excalibur Sudoku
(Sumber: Aplikasi "Sudoku & Variants by Logic Wiz" di Android)

Pada gambar 4 terlihat contoh persoalan dari Excalibur Sudoku. Pada *puzzle* tersebut, dapat terlihat baris, kolom, dan *subgrid* yang sama seperti Sudoku pada umumnya. Akan tetapi, terdapat perbedaan yaitu ada beberapa lingkaran dengan panah yang berasal dari lingkaran tersebut yang melewati beberapa sel. Lingkaran tersebut menunjukkan aturan yaitu angka pada lingkaran haruslah merupakan hasil penjumlahan dari garis panah pada sel-sel yang dilewati.

III. Visualisasi Algoritma *Backtracking* dalam Permainan Sudoku dengan Aturan Tambahan (Excalibur Sudoku)

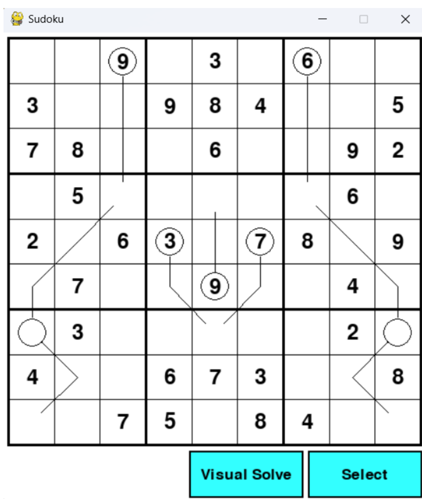
A. Pembuatan Generator dan Tampilan Excalibur Sudoku

Penulis membuat algoritma untuk memecahkan Excalibur Sudoku menggunakan Python. Pada program, dibuat kelas utama yaitu ExcaliburSudoku yang berfungsi untuk menyimpan semua *logic* dari *puzzle.* Kelas ini dapat membaca file konfigurasi dalam bentuk json untuk diterjemahkan dalam atribut-atribut dari kelas, yaitu atribut utama *board,* matriks 9x9 yang berisi angka tiap sel dan atribut *arrows* yang berisi panah-panah yang terdapat pada *puzzle.* Maka dari itu, file json yang dapat dibaca harus berupa *object* dengan dua *key. Key*

board berisi array dua dimensi berukuran 9x9 yang berisi angka 0 hingga 9. Angka 0 artinya sel yang berkorespondensi dengan elemen array tersebut merupakan sel kosong. *Key* kedua, yaitu *arrows,* berisi array dari *arrow.* Masing-masing arrow merupakan sebuah tuple. Elemen pertama tuple merupakan koordinat dari ujung panah (lingkaran) sedangkan elemen kedua adalah array yang berisi koordinat-koordinat dari garis panah. Sebagai contoh, panah pertama di *subgrid* terkiri dan teratas di gambar 3 akan memiliki konfigurasi pada json yaitu [[0, 2], [[1, 2], [2, 2], [3, 2]]].

Penulis akan mengambil contoh-contoh *puzzle* Excalibur Sudoku dari aplikasi sesuai dengan gambar 3. Maka dari itu, perlu untuk membuat *generator* untuk konfigurasi file json guna mempermudah proses *input* dari persoalan pada aplikasi. *Generator* ini akan meminta input berupa 9 baris dengan masing masing baris berupa 9 angka 0 hingga 9. Kemudian, akan diminta juga input berupa seluruh panah-panah pada *puzzle.* Setelah itu, *puzzle* yang telah di-input dapat di-*export* pada suatu file json yang diminta.

Setelah *generator* dan *loader* konfigurasi, penulis juga membuat suatu tampilan GUI untuk memperjelas proses algoritma. Tampilan ini dibuat menggunakan PyGame dengan mengambil sumber dari GitHub yang telah membuat tampilan Sudoku sederhana pada PyGame. Program tersebut dapat dikembangkan untuk menambahkan tampilan dari panah-panah pada papan permainan. Selain papan Sudoku, ditambahkan juga dua tombol yaitu tombol untuk visualisasi (akan dijelaskan berikutnya) dan tombol *select* untuk memilih file konfigurasi json.



Gambar 5. Hasil Tampilan GUI yang Dibuat Sesuai dengan Konfigurasi pada Gambar 4

B. Pembuatan Algoritma Backtracking

Seperti yang disebutkan sebelumnya, algoritma *backtracking* untuk memecahkan Sudoku dapat dibuat secara cukup naif yaitu dengan mencoba seluruh kemungkinan hingga ditemukan solusi atau melanggar aturan. Apabila hanya dibuat algoritma seperti itu, maka hanya perlu menambahkan pengecekan pada fungsi pembatas untuk mengecek apakah sel yang diisi melanggar panah-panah yang ada atau tidak. Akan tetapi, penulis akan membuat algoritma yang lebih sangkil dengan menerapkan metode *notes* untuk menentukan angka mana yang lebih tepat untuk diisikan pada langkah berikutnya (bukan urut dari kiri atas ke kanan bawah). Algoritma ini akan memiliki properti umum sebagai berikut:

1. Solusi persoalan

   Solusi akan berupa tuple (x_1, x_2, ..., x_n) dengan n merupakan jumlah sel kosong pada awal algoritma. Masing-masing dari x merupakan sebuah tuple dengan elemen pertama merupakan koordinat sel yang diisi dan elemen kedua adalah angka yang diisikan pada sel tersebut. Dengan solusi persoalan seperti ini, langkah pada algoritma *backtracking* tidak harus urut dari kiri atas ke kanan bawah. Pada aplikasinya, solusi persoalan akan langsung dimasukkan pada matriks *board* yang ada pada *object* sudoku sehingga apabila telah ditemukan solusi untuk seluruh sel, angka-angka pada matriks *board* akan bernilai 1 hingga 9 seluruhnya. Misalkan langkah pertama mengisi angka 5 pada baris indeks 4 dan kolom indeks 2, maka x_1 akan bernilai ((4, 2), 5).

2. Fungsi pembangkit nilai x_k

   Bagian ini merupakan bagian penting dalam optimasi algoritma *backtracking.* Pada setiap sel yang kosong akan dicatat semua angka yang mungkin diisikan yang tidak menyalahi *constraint* dari Sudoku. Kemudian, akan dicari koordinat sel dengan jumlah kemungkinan terkecil. Selain itu, algoritma akan mencari apakah suatu angka hanya mungkin diletakkan pada satu sel tertentu pada suatu baris, kolom, atau *subgrid.* Akibatnya, jika pada suatu sel hanya memiliki satu kemungkinan nilai, sel tersebut akan dipilih sebagai elemen pertama pada tuple di x_k dan pemilihan tersebut sudah relatif benar untuk tahap itu (tidak menambah cabang baru) yang akan membuat algoritma lebih sangkil. Selain itu, dengan memilih bagian dengan jumlah cabang (*expand node* baru) paling sedikit (kemungkinan angka yang diisi paling sedikit), maka nantinya jumlah langkah keseluruhan yang dibutuhkan akan jauh lebih sedikit. Bagian yang lumayan rumit adalah untuk membuat "catatan" pada tiap sel yang diperbarui tiap langkahnya terutama dengan adanya tambahan aturan Excalibur Sudoku.

3. Fungsi pembatas (*bounding function)*

   Pada algoritma *backtracking* sederhana untuk memecahkan sudoku, fungsi pembatas yang digunakan adalah mengecek apakah ada pelanggaran aturan pada angka-angka yang sudah dimasukkan ke dalam sel. Akan tetapi, berdasarkan fungsi pembangkit yang telah didefinisikan sebelumnya, seluruh langkah yang diambil adalah langkah yang *valid* secara aturan Excalibur Sudoku. Maka dari itu, akan diberlakukan fungsi pembatas yang lain. Fungsi pembatas yang akan digunakan adalah pengecekan terhadap "catatan" yang paling baru setelah dilakukannya langkah terakhir. Apabila langkah terakhir mengakibatkan ada sel

kosong yang tidak memiliki nilai yang mungkin untuk diisi sama sekali, maka fungsi pembatas akan langsung mengembalikan nilai *false.*

Setelah mendefinisikan ketiga properti umum tersebut, algoritma juga perlu dirancang langkah-langkah penyelesaiannya yaitu:

1. Setelah konfigurasi dari Excalibur Sudoku dimasukkan sebagai input, akan dilakukan proses awal yaitu melakukan iterasi pada seluruh elemen *board*. Iterasi ini akan menghitung total sel kosong dan menyimpan jumlahnya sebagai nilai n yaitu panjang dari solusi. Selain itu, akan dibuat suatu matriks 9x9 dengan elemen berupa set yang menyimpan nilai-nilai yang mungkin pada setiap sel yang disebut dengan *possible_vals*. Setiap sel yang kosong akan diinisasi dengan nilai set seluruh angka dari 1 sampai 9.
2. Iterasi dilakukan kembali pada setiap sel yang sudah terisi (sel soal). Untuk setiap sel tersebut, *possible_vals* akan diperbarui dengan nilai dan koordinat pada sel dengan menggunakan *method update_possible_vals*. *Method* ini akan menghapus seluruh angka yang ada pada sel dari set di *possible_vals* pada baris, kolom, atau *subgrid* yang sama. Akibatnya, jika terdapat angka 1 pada baris 0 kolom 0, maka setiap angka 1 pada set di *possible_vals* baris 0, kolom 0, dan *subgrid* 0 akan dihapus.
3. Proses awal terakhir yang dilakukan yaitu memperbarui *possible_vals* sesuai dengan aturan *arrows* yang diberikan yaitu dengan *method update_arrows*. Cara kerja *method* ini adalah sebagai berikut. Akan dilakukan iterasi untuk semua *arrow* yang ada. Pada masing-masing *arrow*, akan diambil nilai dari sel yang terdapat pada ujung panah. Jika sel tersebut telah terisi (memiliki nilai), maka nilai tersebut akan dimasukkan pada array *sum_possibilities* dengan menjadi satu-satunya nilai pada array tersebut. Sebaliknya, *sum_possibilities* akan diambil dari *possible_vals* sel tersebut. Setelah itu, setiap elemen pada *sum_possibilities* tersebut akan diiterasi dan dicarikan seluruh permutasi angka 1 sampai 9 yang mungkin sesuai dengan panjang panah yang apabila dijumlahkan bernilai sama. Sebagai contoh, apabila jumlahnya adalah 3 dengan panjang panah 2, maka angka yang mungkin adalah [1, 2] dengan [2, 1]. Kemudian, angka-angka ini akan dicek pada bagian seluruh sel pada garis panah. Jika permutasi tersebut mungkin untuk diberlakukan pada sel-sel panah, maka angka-angka yang bersangkutan akan tetap bertahan pada *possible_vals*. Sebaliknya, angka tersebut akan dihapus dari set pada sel. Setelah proses ini dilakukan, nilai pada *possible_vals* dengan sel yang terkait dengan suatu *arrow* akan lebih terestriksi.
4. Setelah proses awal telah selesai, barulah fungsi rekursif dari algoritma *backtracking* dapat dijalankan. Fungsi ini akan berakhir jika total sel yang dijawab sudah mencapai n (akan mengembalikan nilai *true*). Pertama, kondisi simpul akan dicek dengan fungsi pembatas, yaitu mengecek apakah terdapat sel kosong yang tidak memiliki *possible values.* Jika ada, maka akan langsung mengembalikan nilai *false.*
5. Setelah fungsi pembatas, akan dilakukan langkah pertama dalam fungsi pembangkit yaitu mencari angka yang hanya dapat diisikan hanya pada satu sel kosong pada suatu baris, kolom, atau *subgrid* tertentu. Apabila ditemukan, maka sel dan angka tersebut akan dipilih sebagai nilai x kemudian *possible_vals* akan diperbarui dengan memanggil *method update_possible_vals* dan *update_arrows* dan lanjut ke simpul berikutnya.
6. Jika tidak ada, maka lanjut ke langkah berikutnya yaitu mencari sel dengan kemungkinan angka paling sedikit. Apabila telah ditemukan, maka setiap kemungkinan pada sel tersebut akan diekspan untuk ditemukan solusi angka yang sesuai dengan solusi dan juga memperbarui *possible_vals* sama seperti sebelumnya. Langkah 4, 5, dan 6 akan terus diulang hingga ditemukan solusi atau jika tidak ada solusi pada Sudoku.

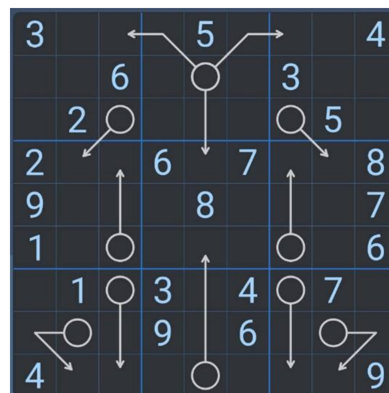
C. Visualisasi

Setelah algoritma *backtracking,* maka perlu dibuat implementasi untuk memungkinkan diberlakukannya visualisasi pada algoritma. Hal ini dapat dilakukan dengan sedikit menambahkan fungsionalitas pada tiap langkah rekursif, yaitu setelah mengisi suatu sel, GUI akan diperbarui sesuai dengan *board* terbaru dan diberikan delay dengan durasi tertentu untuk memperjelas visualisasi.

D. Percobaan

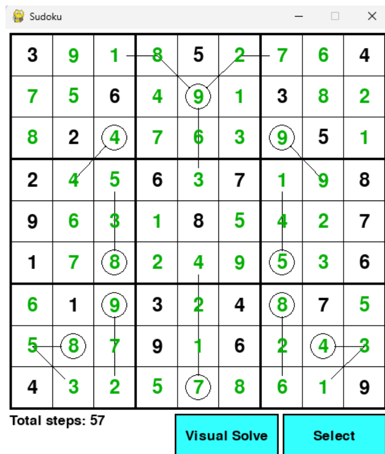
Percobaan dilakukan dengan berbagai tingkat kesulitan dari Excalibur Sudoku, yaitu *easy, medium, hard,* hingga *expert.* Pada bagian ini hanya akan diperlihatkan persoalan dan solusi dari tiap tingkat kesulitan. Untuk visualisasi lebih jelas, dapat dengan melihat video yang terlampir.

1. *Easy*



Gambar 6. Persoalan *Easy*

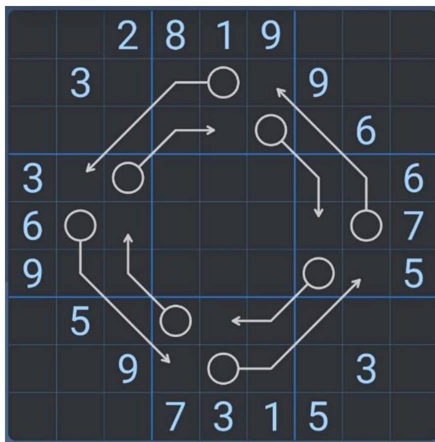
(Sumber: Aplikasi "Sudoku & Variants by Logic Wiz" di Android)



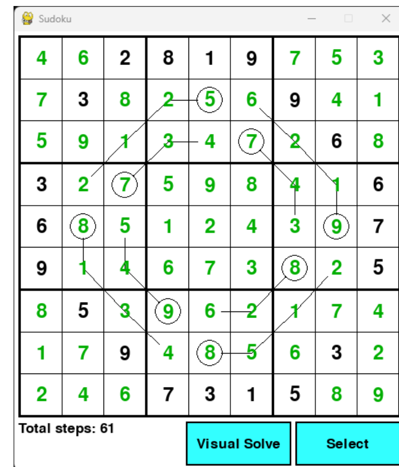
Gambar 7. Solusi dari Persoalan *Easy*

Pada solusi terlihat bahwa jumlah langkah atau simpul yang diproses sama dengan jumlah sel kosong pada soal. Hal ini membuktikan bahwa pada proses pencarian sama sekali tidak terjadi *backtrack* yang mengindikasikan bahwa algoritma yang dibuat sudah efisien. Kemudian, dari solusi juga dapat terlihat untuk semua panah yang ada, aturan yang diberikan sudah dipenuhi, yaitu jumlah seluruh angka pada garis panah sama dengan angka pada ujung panah (contohnya 9 = 1 + 8).

2. *Medium*



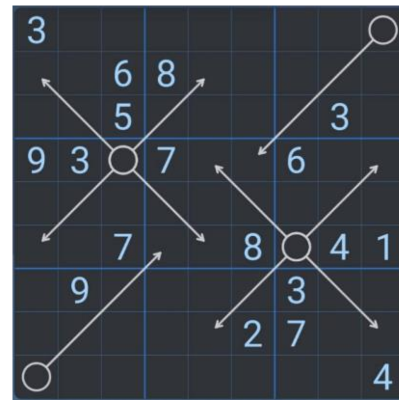
Gambar 8. Persoalan *Medium*
(Sumber: Aplikasi "Sudoku & Variants by Logic Wiz" di Android)



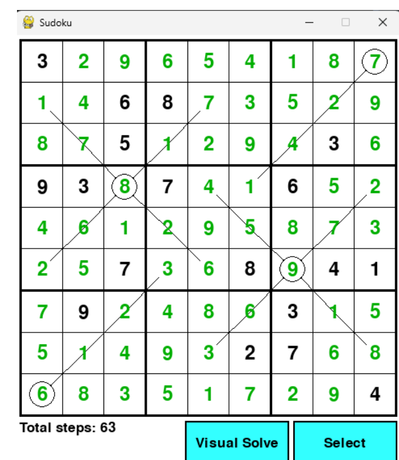
Gambar 9. Solusi dari Persoalan *Medium*

Pada solusi *medium* juga terlihat bahwa jumlah langkah sama dengan jumlah sel kosong yang artinya algoritma juga sudah efisien untuk tingkat kesulitan *medium.*

3. *Hard*



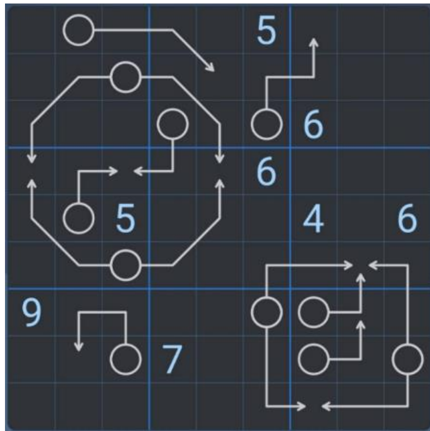
Gambar 10. Persoalan *Hard*
(Sumber: Aplikasi "Sudoku & Variants by Logic Wiz" di Android)



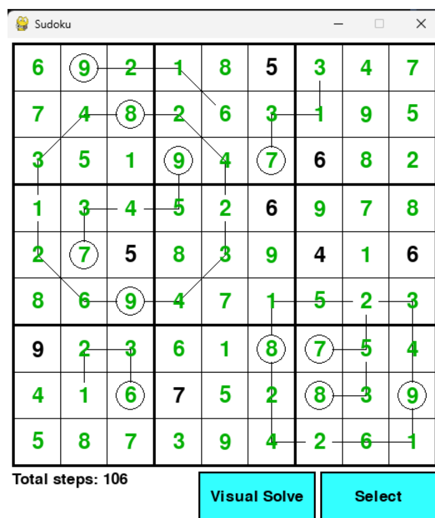
Gambar 11. Solusi dari Persoalan *Hard*

Pada solusi *hard* juga terlihat bahwa jumlah langkah sama dengan jumlah sel kosong yang artinya algoritma juga sudah efisien untuk tingkat kesulitan *hard.*

4. *Expert*



Gambar 12. Persoalan *Expert*
(Sumber: Aplikasi "Sudoku & Variants by Logic Wiz" di Android)



Gambar 13. Solusi dari Persoalan *Expert*

Pada solusi *expert*, baru terlihat perbedaan antara jumlah sel kosong dengan jumlah langkah. Jumlah sel kosong adalah 73 dengan jumlah langkah 106. Artinya, terjadi beberapa langkah *backtrack* pada proses pencarian solusi. Akan tetapi, ternyata hingga tingkat kesulitan tertinggi, algoritma yang dibuat sudah sangat efisien dengan jumlah langkah yang sangat sedikit.

IV. KESIMPULAN

Sudoku merupakan *puzzle* yang sangat populer meski dengan aturan yang cukup sederhana. Banyak metode yang dapat dilakukan untuk mencari solusi dari permainan Sudoku, salah satunya adalah dengan algoritma *backtracking.* Bahkan, dalam algoritma *backtracking* untuk mencari solusi Sudoku pun terdapat berbagai macam pendekatan untuk dapat mencapai algoritma yang efisien. Dengan adanya aturan tambahan pada Sudoku seperti yang ada pada Excalibur Sudoku, algoritma yang dibuat juga perlu dilakukan pendekatan yang berbeda supaya dapat menyesuaikan dengan aturan tambahan yang ada. Salah satu pendekatan yang dilakukan oleh penulis adalah dengan menyimpan nilai-nilai yang mungkin pada tiap sel pada papan permainan untuk mencari langkah yang paling efektif.

LINK VIDEO YOUTUBE

https://youtu.be/GCkZudEy6EI

LINK REPOSITORY GITHUB

https://github.com/arsaizdihar/excalibur-sudoku-visualization

UCAPAN TERIMA KASIH

Penulis ingin mengucapkan terima kasih kepada beberapa pihak. Pertama, penulis mengucapkan puji syukur kepada Allah SWT karena atas nikmat dan rahmat-Nya, penulis dapat menyelesaikan makalah berjudul "Visualisasi Algoritma Backtracking dalam Permainan Sudoku dengan Aturan Tambahan (Excalibur Sudoku)" dengan baik. Selain itu, tidak lupa juga ucapan terima kasih kepada dosen mata kuliah Strategi Algoritma saya yaitu Bapak Dr. Ir. Rinaldi Munir, M. T. yang telah membimbing penulis selama berkuliah di mata kuliah ini. Penulis juga mengucapkan terima kasih kepada seluruh sumber yang dijadikan referensi pada makalah ini.

REFERENSI

[1] Kistler21, "pygame-sudoku" https://github.com/Kistler21/pygame-sudoku (Diakses 21 Mei 2023)

[2] Munir, Rinaldi. 2021. "Algoritma runut-balik (backtracking) (Bagian 1)" https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-backtracking-2021-Bagian1.pdf (Diakses 21 Mei 2023)

[3] Munir, Rinaldi. 2023. "Pengantar Strategi Algoritma" https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2022-2023/Pengantar-Strategi-Algoritma-(2023).pdf (Diakses 21 Mei 2023)

[4] Delahaye, Jean-Paul. 2006, "The Science Behind Sudoku" https://www.researchgate.net/publication/7071130_The_Science_behind_Sudoku (Diakses 22 Mei 2023)

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, 22 Mei 2023

Arsa Izdihar Islam 13521101