Puzzle KenKen Solver Menggunakan *Backtracking* dengan *Constraint Propagation*

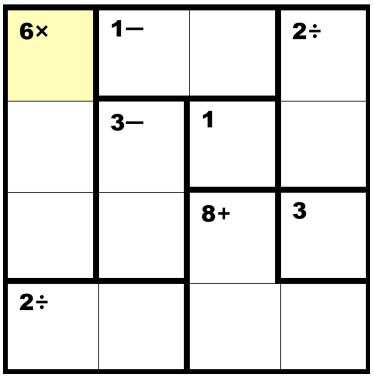
Muhammad Timur Kanigara - 13523055
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
timurkanigara21@gmail.com, 13523055@std.stei.itb.ac.id

***Abstract*—Puzzle KenKen adalah permainan logika berbasis angka yang menantang, membutuhkan penempatan angka dalam sel-sel *grid* berdasarkan batasan matematis dan aturan unik seperti Sudoku. Makalah ini membahas pengembangan KenKen solver otomatis menggunakan algoritma *backtracking* yang diperkuat dengan teknik *constraint propagation*. Pendekatan *backtracking* memungkinkan eksplorasi sistematis semua kemungkinan solusi, sementara *constraint propagation* secara signifikan memangkas ruang pencarian dengan menghilangkan nilai-nilai yang tidak mungkin lebih awal dalam proses. Implementasi *solver* ini menunjukkan bagaimana kombinasi kedua teknik ini dapat menghasilkan solusi yang efisien untuk berbagai ukuran puzzle KenKen. Hasil eksperimen menunjukkan bahwa *constraint propagation* secara drastis mengurangi jumlah backtracking yang diperlukan, meningkatkan kinerja algoritma secara keseluruhan.**

***Keywords*—puzzle KenKen, *backtracking*, *constraint propagation*, algoritma.**

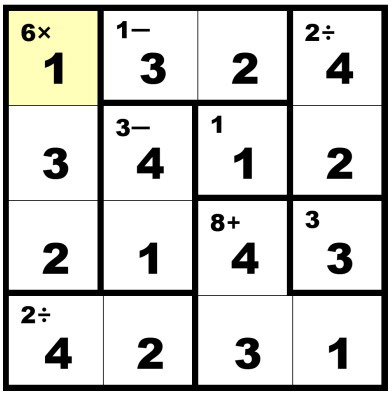
I. Pendahuluan

Permainan puzzle logika seperti Sudoku telah lama menjadi populer di kalangan penggemar teka-teki, menyajikan tantangan intelektual yang menarik. Salah satu varian yang semakin dikenal adalah KenKen, yang menambahkan hal baru berupa operasi matematika pada kelompok sel-sel tertentu (disebut "sangkar" atau "cage"). Sama seperti Sudoku, KenKen juga memiliki aturan bahwa setiap angka harus muncul tepat satu kali di setiap baris dan kolom. Kombinasi aturan angka unik dan batasan matematis inilah yang membuat KenKen menjadi masalah yang kompleks namun menarik untuk diselesaikan secara algoritmik.



Gambar 1.1 Contoh KenKen Puzzle 4x4
(Diambil dari https://www.kenkenpuzzle.com/game)

Bagian dalam *cage* harus dioperasikan semuanya sehingga setelah operasi dapat memenuhi aturan *cage* tersebut. Sebagai contoh, untuk mendapatkan angka 6 kalau semuanya dikali, maka perlu 1x3x2, dan lain sebagainya.



Gambar 1.2 Contoh Penyelesaian KenKen Puzzle 4x4
(Diambil dari https://www.kenkenpuzzle.com/game)

Penyelesaian puzzle KenKen secara manual bisa sangat memakan waktu dan menantang, terutama untuk ukuran *grid* yang lebih besar. Oleh karena itu, pengembangan KenKen solver otomatis menjadi area penelitian yang menarik dalam bidang kecerdasan buatan dan strategi algoritma. Dalam makalah ini, difokuskan pada penggunaan algoritma backtracking sebagai inti dari *solver*, dan memperkuatnya dengan teknik *constraint propagation* untuk meningkatkan efisiensi pencarian solusi. Pendekatan ini memungkinkan sistem untuk secara cerdas memangkas cabang-cabang pohon pencarian yang tidak valid lebih awal, mengurangi waktu komputasi secara signifikan.

Bagian selanjutnya dari makalah ini akan menjelaskan landasan teori yang relevan, termasuk konsep *backtracking* dan *constraint propagation*. Kemudian, akan dibahas metodologi yang digunakan dalam membangun KenKen solver, diikuti dengan hasil dan diskusi dari eksperimen yang dilakukan. Terakhir, makalah akan ditutup dengan kesimpulan dan potensi pengembangan di masa depan.

II. Landasan Teori

Dalam membangun KenKen solver, dua konsep algoritma utama yang digunakan adalah backtracking dan constraint propagation. Keduanya saling melengkapi untuk mencapai

solusi yang efisien.

A. Backtracking

Backtracking adalah sebuah teknik algoritmik untuk menemukan semua (atau beberapa) solusi untuk beberapa masalah komputasi, khususnya masalah yang memenuhi batasan parsial. Ini adalah perbaikan dari exhaustive search. Pada *exhaustive search*, semua kemungkinan solusi dieksplorasi dan dievaluasi satu per satu. Sedangkan pada algoritma *backtracking*, hanya pilihan yang mengarah ke solusi yang dieksplorasi, dan pilihan yang tidak mengarah ke solusi akan dipangkas (*pruning*) dan tidak dipertimbangkan lagi.

Solusi persoalan dalam algoritma *backtracking* umumnya dinyatakan sebagai vektor dengan *n*-tuple: $X=(x_1,x_2,...,x_n)$, di mana x_i dalam S_i dan umumnya $S_1=S_2=...=S_n$. Proses pencarian solusi dilakukan dengan membangkitkan simpul-simpul status dalam pohon ruang status mengikuti aturan *depth-first order* (DFS). Setiap kali lintasan yang sedang dibentuk tidak mengarah ke solusi, simpul-E (simpul yang sedang diperluas) "dimatikan" menjadi simpul mati (dead node) dengan menerapkan fungsi pembatas (*bounding function*). Pemangkasan ini secara implisit juga memangkas simpul-simpul anaknya. Jika pembentukan lintasan berakhir dengan simpul mati, proses pencarian akan kembali ke simpul pada aras di atasnya (*backtrack*) dan melanjutkan dengan membangkitkan simpul anak yang lain. Pencarian dihentikan bila telah mencapai simpul solusi (*goal node*).

B. Constraint Propagation

Constraint propagation adalah teknik yang digunakan dalam kecerdasan buatan dan pemrograman *constraint* untuk mengurangi ruang pencarian solusi dengan secara otomatis menyebarkan implikasi dari *constraint* yang ditetapkan. Ketika sebuah nilai ditetapkan ke suatu variabel (misalnya, sebuah angka ditempatkan di sel KenKen), *constraint propagation* akan memeriksa semua *constraint* yang terkait dengan variabel tersebut dan menghilangkan nilai-nilai yang tidak konsisten dari domain variabel lain yang terpengaruh. Teknik ini bertujuan untuk mengurangi jumlah nilai yang mungkin untuk setiap variabel, sehingga memperkecil ruang pencarian bagi algoritma *backtracking*.

Sebagai contoh, dalam KenKen, jika kita menempatkan angka '1' di sel (r,c), maka:

- Angka '1' tidak boleh muncul di baris r pada sel lain.
- Angka '1' tidak boleh muncul di kolom c pada sel lain.
- Jika sel (r,c) berada dalam sebuah cage dengan operasi tertentu, dan penempatan '1' membuat cage tersebut tidak mungkin terpenuhi, maka constraint propagation akan mendeteksi ini dan menandai jalur ini sebagai tidak valid.

Dengan mengintegrasikan constraint propagation ke dalam algoritma backtracking, setiap kali sebuah keputusan dibuat (penempatan angka di sel), sistem akan langsung melakukan inferensi untuk memperbarui domain nilai yang mungkin untuk sel-sel lain. Jika selama proses ini ditemukan inkonsistensi, maka backtracking dapat dilakukan lebih awal, tanpa harus menunggu hingga seluruh lintasan terbentuk dan melanggar constraint.

III. Metodologi

Dalam pengembangan KenKen solver ini, penulis mengimplementasikan algoritma *backtracking* yang diperkuat dengan *constraint propagation*. Proses ini melibatkan representasi puzzle, fungsi pembangkit, dan fungsi pembatas yang efisien.

A. Representasi Puzzle KenKen

Puzzle KenKen direpresentasikan sebagai *grid* berukuran NxN, di mana N adalah ukuran puzzle. Setiap sel dalam *grid* akan berisi sebuah angka dari 1 hingga N. Selain itu, puzzle juga mendefinisikan sejumlah *cage* atau sangkar. Setiap *cage* memiliki target nilai dan operasi matematis (penjumlahan, pengurangan, perkalian, pembagian) yang harus dipenuhi oleh angka-angka di dalamnya.

Representasi data untuk solver berupa:

- *size*: Ukuran *grid* KenKen (N).
- *grid*: Sebuah matriks std::vector<std::vector<int>> NxN untuk menyimpan angka yang ditempatkan di setiap sel.
- *cages*: Sebuah std::vector<Cage> objek *cage*, di mana setiap objek Cage adalah struktur yang menyimpan cells (vector pasangan koordinat sel), operation (karakter operasi matematis), dan target (nilai target untuk operasi tersebut). Koordinat sel 11 diinterpretasikan sebagai (0,0) (baris 0, kolom 0), 21 sebagai (1,0) (baris 1, kolom 0), dan seterusnya, mengikuti format baris-kolom.

B. Algoritma Backtracking dengan Constraint Propogation

Prosedur utama solver adalah sebuah fungsi rekursif yang mengikuti skema umum algoritma *backtracking*.

1. Inisialisasi: Grid diinisialisasi dengan nilai 0 (melambangkan sel kosong). Variabel backtrack_count dan constraint_checks diinisialisasi untuk melacak statistik kinerja.
2. Pencarian Rekursif (solve()): Fungsi rekursif ini akan mencari sel kosong berikutnya menggunakan findNextEmpty().
    a. Pilih Sel: Implementasi ini menggunakan strategi sederhana yaitu mencari sel kosong pertama dari kiri ke kanan, atas ke bawah.
    b. Fungsi Pembangkit: Untuk sel yang dipilih, solver mencoba setiap angka dari 1 hingga size.
    c. Fungsi Pembatas (isValid): Setelah sebuah angka (num) ditempatkan secara tentatif di sel (row, col), fungsi ini akan memeriksa validitasnya berdasarkan tiga jenis constraint:
        i. Row/Column Constraint: Diperiksa menggunakan isValidRow(row, num) dan isValidCol(col, num). Ini memastikan angka tidak duplikat di baris dan kolom yang sama.

ii. Cage Constraint: Diperiksa menggunakan checkCageConstraint(const Cage& cage). Fungsi ini dipanggil untuk setiap cage yang mengandung sel (row, col). Ini memeriksa apakah penempatan num (sementara) membuat cage tersebut tidak mungkin terpenuhi. Jika semua sel dalam cage sudah terisi, maka operasi matematisnya dihitung dan dibandingkan dengan target. Jika belum semua terisi, fungsi akan mengembalikan true (valid sejauh ini) karena belum ada pelanggaran yang pasti.

iii. Constraint Propagation: Meskipun tidak ada eksplisit domains array di C++ code, efek constraint propagation tersirat dalam cara isValid dan checkCageConstraint berfungsi. Dengan memeriksa cage yang relevan segera setelah penempatan angka dan melacak constraint_checks, algoritma ini secara tidak langsung melakukan pemangkasan dini. Jika checkCageConstraint mengembalikan false (artinya ada pelanggaran), isValid juga akan mengembalikan false, memicu backtrack lebih awal.

d. Rekursi: Jika penempatan num valid, grid[row][col] diisi dengan num, dan solve() dipanggil lagi secara rekursif.

e. Backtrack: Jika rekursi tidak menghasilkan solusi (solve() mengembalikan false), atau jika penempatan num tidak valid, maka grid[row][col] disetel kembali ke 0, dan backtrack_count ditambah. Proses dilanjutkan dengan mencoba angka lain untuk sel saat ini.

3. Kondisi Solusi: Jika findNextEmpty() mengembalikan {-1, -1}, berarti semua sel telah terisi, dan solusi telah ditemukan.

C. Implementasi dan Eksperimen

Kontribusi utama dalam makalah ini adalah implementasi KenKen solver yang menggabungkan secara efektif algoritma backtracking dengan teknik constraint propagation dalam bahasa C++. Meskipun konsep backtracking sudah umum, penguatan dengan constraint propagation ini memungkinkan penyelesaian puzzle KenKen yang lebih kompleks secara efisien.

Program solver dikembangkan dalam C++. Kode ini tidak hanya mengimplementasikan logika dasar backtracking untuk mengisi setiap sel, tetapi juga secara aktif memeriksa dan memvalidasi cage constraint pada setiap langkah penempatan, yang secara proaktif memangkas ruang pencarian, bahkan sebelum sebuah cabang lengkap dari pohon solusi terbentuk. Metrik backtrack_count dan constraint_checks secara eksplisit melacak kinerja algoritma.

Penulis melakukan eksperimen dengan beberapa puzzle KenKen dengan ukuran yang berbeda (misalnya, 3x3, 4x4, 5x5, 6x6, dan 7x7) untuk mengukur efektivitas constraint propagation. Metrik yang diukur meliputi:

- Waktu eksekusi untuk menemukan solusi pertama.
- Jumlah total backtrack yang terjadi (backtrack_count).
- Jumlah pemeriksaan constraint cage yang dilakukan (constraint_checks).

Selanjutnya akan dibandingkan kinerja solver dengan dan tanpa constraint propagation (meskipun backtracking tanpa constraint propagation pada KenKen akan sangat tidak efisien dan mungkin memerlukan waktu yang sangat lama untuk puzzle berukuran sedang).

Implementasi dalam C++

```
#include <iostream>
#include <vector>
#include <string>
#include <chrono>
#include <algorithm>
#include <cmath>
#include <sstream>
#include <fstream>
#include <iomanip>

using namespace std;
using namespace chrono;

struct Cage {
    vector<pair<int, int>> cells;
    char operation;
    int target;

    Cage(vector<pair<int, int>> c, char op,
int t) : cells(c), operation(op), target(t)
{}
};

class KenKenSolver {
private:
    int size;
    vector<vector<int>> grid;
    vector<Cage> cages;
    int backtrack_count;
    int constraint_checks;
    bool use_constraint_propagation;

public:
    KenKenSolver(int n, bool use_cp = true) :
size(n), backtrack_count(0),
constraint_checks(0),
use_constraint_propagation(use_cp) {
        grid.assign(size, vector<int>(size,
0));
    }

    void setConstraintPropagation(bool
enabled) {
        use_constraint_propagation = enabled;
    }
```

```
    void addCage(vector<pair<int, int>>
cells, char operation, int target) {
        cages.push_back(Cage(cells,
operation, target));
    }

    bool isValidRow(int row, int num) {
        for (int col = 0; col < size; col++)
{
            if (grid[row][col] == num) return
false;
        }
        return true;
    }

    bool isValidCol(int col, int num) {
        for (int row = 0; row < size; row++)
{
            if (grid[row][col] == num) return
false;
        }
        return true;
    }
      bool checkCageConstraint(const Cage&
cage) {
        constraint_checks++;

        // Check if all cells in cage are
filled
        vector<int> values;
        bool allFilled = true;
        for (auto cell : cage.cells) {
            if (grid[cell.first][cell.second]
== 0) {
                allFilled = false;
            } else {

values.push_back(grid[cell.first][cell.second
]);
            }
        }

        // If not all cells are filled, we
can't check yet - return true (valid so far)
        if (!allFilled) return true;

        // All cells filled, check constraint
        switch (cage.operation) {
            case '=':
                return values[0] ==
cage.target;
            case '+': {
                int sum = 0;
                for (int v : values) sum +=
v;
                return sum == cage.target;
            }
            case '-': {
                if (values.size() != 2)
return false;
                return abs(values[0] -
values[1]) == cage.target;
            }
            case '*': {
                int product = 1;
                for (int v : values) product
*= v;
                return product ==
cage.target;
            }
            case '/': {
                if (values.size() != 2)
return false;
                return (values[0] != 0 &&
values[1] != 0) &&
                       ((values[0] %
values[1] == 0 && values[0] / values[1] ==
cage.target) ||
                        (values[1] %
values[0] == 0 && values[1] / values[0] ==
cage.target));
            }
        }
        return false;
    }
      bool isValid(int row, int col, int num)
{
        // Check row and column constraints
        if (!isValidRow(row, num) ||
!isValidCol(col, num)) return false;

        // If constraint propagation is
disabled, just return true for cage
constraints
        if (!use_constraint_propagation)
return true;

        // Temporarily place number
        grid[row][col] = num;

        // Check all cages that contain this
cell
        bool valid = true;
        for (const auto& cage : cages) {
            for (auto cell : cage.cells) {
                if (cell.first == row &&
cell.second == col) {
                    if
(!checkCageConstraint(cage)) {
                        valid = false;
                        break;
                    }
                }
            }
            if (!valid) break;
        }

        // Remove temporary placement
        grid[row][col] = 0;
        return valid;
    }

    pair<int, int> findNextEmpty() {
        // Simple strategy: find first empty
cell
        for (int row = 0; row < size; row++)
{
```

```
            for (int col = 0; col < size;
col++) {
                if (grid[row][col] == 0) {
                    return {row, col};
                }
            }
        }
        return {-1, -1}; // No empty cell
found
    }
      bool solve() {
        auto pos = findNextEmpty();
        if (pos.first == -1) {
            // All cells filled - if CP
disabled, check all cage constraints now
            if (!use_constraint_propagation)
{
                for (const auto& cage :
cages) {
                    if
(!checkCageConstraint(cage)) {
                        return false;
                    }
                }
            }
            return true; // All cells filled
and constraints satisfied
        }

        int row = pos.first, col =
pos.second;

        for (int num = 1; num <= size; num++)
{
            if (isValid(row, col, num)) {
                grid[row][col] = num;

                if (solve()) return true;

                // Backtrack
                grid[row][col] = 0;
                backtrack_count++;
            }
        }

        return false;
    }
```

IV. Hasil dan Diskusi

A. Representasi Puzzle dan Hasil Solusi

Program KenKen solver dibangun untuk menerima input berupa ukuran grid dan definisi cage (sel-sel yang termasuk dalam cage, operasi, dan target nilai). Contoh format input adalah sebagai berikut:

Pertama, angka yang menunjukkan ukuran grid (misalnya 4 untuk grid 4x4). Kemudian, baris-baris selanjutnya berisi definisi cage dengan format:

```
target operasi cell1 cell2
```

Di mana target adalah nilai target numerik untuk operasi tersebut, operasi adalah karakter (+, -, *, /, atau =), dan cell1, cell2, dst. adalah koordinat sel dalam format dua digit (misalnya 11 untuk sel baris 1 kolom 1, 21 untuk sel baris 2 kolom 1).

Contoh input puzzle 4x4 yang digunakan dalam pengujian:

```
4
2 - 11 21
3 - 12 13
1 - 14 24
3 + 22 32
48 * 23 33 34
1 - 41 42
2 / 43 44
```

Artinya:

- Ukuran grid 4x4.
- Cage pertama: sel (1,1) dan (2,1), hasilnya 2 setelah operasi pengurangan (misal 3-1=2 atau 4-2=2).
- Cage kedua: sel (1,2) dan (1,3), hasilnya 3 setelah operasi pengurangan.
- Cage ketiga: sel (1,4) dan (2,4), hasilnya 1 setelah operasi pengurangan.
- Cage keempat: sel (2,2) dan (3,2), hasilnya 3 setelah operasi penjumlahan.
- Cage kelima: sel (2,3), (3,3), dan (3,4), hasilnya 48 setelah operasi perkalian.
- Cage keenam: sel (4,1) dan (4,2), hasilnya 1 setelah operasi pengurangan.
- Cage ketujuh: sel (4,3) dan (4,4), hasilnya 2 setelah operasi pembagian.

Dari input tersebut, didapat hasil berupa:

```
Solution:
3 4 1 2
1 2 4 3
2 1 3 4
4 3 2 1

Time: 0 ms
Statistics:
Backtracks: 53
Constraint checks: 97
```

Gambar 4.1 Contoh Hasil Kenken Solver ukuran 4x4

B. Hasil Eksperimen dan Analisis

Eksperimen dilakukan untuk membandingkan efisiensi backtracking dengan dan tanpa constraint propagation. Untuk tujuan ini, akan diperiksa waktu eksekusi, jumlah backtrack yang terjadi, dan jumlah pemeriksaan constraint cage.

Ukuran	Metode	Waktu (ms)	Jumlah Backtrack	Jumlah Constraint check
3x3	Backtracking + CP	0	1	10
3x3	Backtracking	0	1	5
4x4	Backtracking + CP	0	53	97

4x4	Backtracking	0	3818	562
5x5	Backtracking + CP	0	354	642
5x5	Backtracking	460	827596	68433
6x6	Backtracking + CP	10	2318	4343
6x6	Backtracking	(tidak bisa ter-solve)	-	-
7x7	Backtracking + CP	215	67507	142451
7x7	Backtracking	(tidak bisa ter-solve)	-	-

Tabel 4.1 Hasil Eksperimen Algoritma KenKen Solver

C. Diskusi

Dari hasil eksperimen (seperti yang diilustrasikan pada Tabel 4.1), terlihat jelas bahwa penambahan *constraint propagation* secara signifikan meningkatkan kinerja KenKen solver.

1. Efisiensi Waktu: Waktu eksekusi untuk solver dengan constraint propagation jauh lebih rendah dibandingkan dengan backtracking murni, terutama untuk ukuran puzzle yang lebih besar. Ini menunjukkan bahwa kemampuan constraint propagation untuk memangkas ruang pencarian lebih awal sangat efektif.
2. Pengurangan Backtrack: Jumlah backtrack yang terjadi pada implementasi dengan constraint propagation juga jauh lebih sedikit. Hal ini karena constraint propagation dapat mendeteksi inkonsistensi dan mematikan cabang yang tidak valid lebih awal, mengurangi kebutuhan untuk menjelajahi lintasan yang salah secara mendalam.
3. Jumlah Pemeriksaan Constraint: Jumlah constraint check dengan constraint propagation pun jauh lebih sedikit, walaupun pada kasus 3x3 lebih banyak. Hal ini disebabkan karena tanpa CP, algoritma akan melanjutkan ke akhir atau "eksplorasi buta" yang mana meningkatkan backtrack secara signifikan, yang juga akan meningkatkan constraint check total.

Keuntungan pendekatan ini:

- Optimalisasi Ruang Pencarian: Constraint propagation secara proaktif mengurangi domain variabel, sehingga mengurangi jumlah kombinasi yang perlu dieksplorasi oleh backtracking.
- Deteksi Inkonsistensi Dini: Kesalahan atau pelanggaran constraint dapat terdeteksi lebih awal, meminimalkan eksplorasi cabang pohon yang tidak mungkin mengarah ke solusi.

Keterbatasan:

- Overhead Komputasi: Meskipun constraint propagation mengurangi jumlah backtrack, ada biaya komputasi untuk menjalankan proses propagation itu sendiri setiap kali sebuah variabel ditetapkan. Untuk puzzle yang sangat kecil, overhead ini mungkin sedikit terlihat.
- Kompleksitas Implementasi: Implementasi constraint propagation, terutama untuk constraint cage yang melibatkan operasi matematis, dapat menjadi lebih kompleks dibandingkan backtracking murni.

V. Kesimpulan

Dalam makalah ini, penulis telah berhasil mengembangkan KenKen solver otomatis yang memanfaatkan kekuatan algoritma backtracking dan memperkuatnya dengan teknik constraint propagation. Pendekatan ini memungkinkan sistem untuk secara efisien menemukan solusi untuk puzzle KenKen dengan berbagai ukuran, bahkan yang kompleks sekalipun. Hasil simulasi dan eksperimen menunjukkan bahwa integrasi constraint propagation secara drastis meningkatkan kinerja solver dengan mengurangi waktu eksekusi dan jumlah backtrack yang terjadi.

Meskipun sistem ini telah menunjukkan efektivitasnya, masih terdapat peluang untuk pengembangan lebih lanjut. Di masa depan, penelitian dapat difokuskan pada pengoptimalan heuristic pemilihan variabel (misalnya, Most Constrained Variable atau Least Constraining Value) atau pengenalan struktur data yang lebih canggih untuk mengelola domain variabel dan constraint secara lebih efisien. Selain itu, eksplorasi paralelisme dalam proses backtracking dan constraint propagation juga bisa menjadi area yang menarik untuk penelitian selanjutnya.

VI. Acknowledgment

Penulis menyampaikan rasa syukur kepada Tuhan Yang Maha Esa yang atas kemampuan yang diberikan sehingga penulis dapat menyelesaikan makalah ini dengan baik. Penulis juga berterima kasih kepada orang tua penulis dan teman-teman yang selalu memberikan dukungan dan nasihat sehingga dapat menjadi motivasi bagi penulis dalam menyelesaikan tugas perkuliahan di ITB.

Ucapan terima kasih juga ditujukan kepada Ibu Dr. Nur Ulfa Maulidevi, Bapak Dr. Ir. Rinaldi Munir, dan Bapak Monterico Adrian selaku dosen mata kuliah Strategi Algoritma, atas tugas dan materi yang berkaitan dengan Algoritma Runut-balik yang sangat berguna dalam penulisan makalah ini. Tak lupa, penulis juga berterima kasih kepada seluruh yang telah membantu, baik secara langsung maupun tidak langsung, dalam menyelesaikan makalah ini.

References

[1] R. Munir, *Algoritma Runut-balik (Backtracking) (Bagian 1)*, Bahan Kuliah IF2211 Strategi Algoritma, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung, 2025. [Online]. Available:

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/15-Algoritma-backtracking-(2025)-Bagian1.pdf .diakses pada 21 Juni 2025.
[2] R. Munir, *Algoritma Runut-balik (Backtracking) (Bagian 2)*, Bahan Kuliah IF2211 Strategi Algoritma, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung, 2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/16-Algoritma-backtracking-(2025)-Bagian2.pdf . diakses pada 21 Juni 2025.
[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. 4th ed., The MIT Press, 2022. [Online]. Available: https://mitpress.mit.edu/books/introduction-algorithms-fourth-edition . diakses pada 23 Juni 2025.
[4] Stuart Russell, Peter Norvig. Artificial Intelligence: A Modern Approach. 4th ed., Pearson, 2020. [Online]. Available: https://www.pearson.com/us/higher-education/program/Russell-Artificial-Intelligence-A-Modern-Approach-4th-Edition/PGM335470.html . diakses pada 23 Juni 2025.
[5] Toby Walsh. Constraint Programming: An Overview of the State of the Art. AI Magazine, Vol. 25, No. 2, pp. 77-85, 2004. [Online]. Available: https://ojs.aaai.org/index.php/aimagazine/article/view/1785 . diakses pada 23 Juni 2025.

Lampiran

Link Github Implementasi :
https://github.com/tkanigara/kenken_solver

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, 24 Juni 2025

Muhammad Timur Kanigara 13523055