Penerapan Algoritma Brute Force dan Backtracking untuk Menyelesaikan Puzzle Heyawake

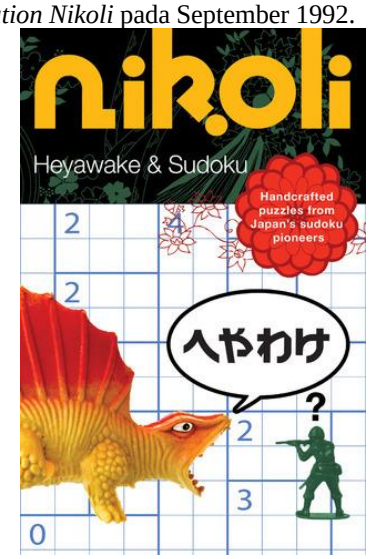
Zachary Samuel Tobing - 13522016
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13522016@std.stei.itb.ac.id

***Abstract*—Makalah ini menjelaskan mengenai pengembangan dan evaluasi dari dua jenis algoritma, *Brute Force* dan *Backtracking* dengan dua konsep dan pendekatan yang berbeda untuk menyelesaikan puzzle Heyawake. Heyawake adalah jenis puzzle logika yang berasal dari Jepang. Puzzle ini biasanya dimainkan di atas grid persegi, yang dibagi menjadi beberapa wilayah oleh garis tebal. Tujuannya adalah untuk menentukan sel mana yang akan dicat hitam. Algoritma *Brute Force* mencari solusi dengan mengenumerasi semua kemungkinan solusi sementara algoritma *Backtracking* mencari solusi dengan batasan dan *pruning* sehingga tidak setiap solusi ditelusuri.**

Keywords—brute force; backtracking; Heyawake; puzzle;

I. Introduction

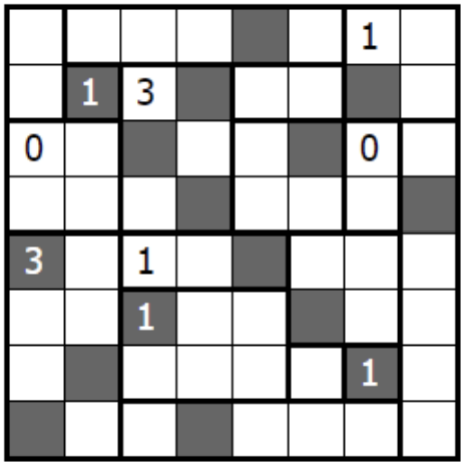
Heyawake (Jepang: へやわけ yang berarti “ruangan yang terpisah”) adalah puzzle logika biner yang diterbitkan oleh Nikoli, sebuah penerbit Jepang yang terkenal karena mempublikasikan berbagai jenis puzzle logika seperti *Sudoku*, *Crossword*, *Connect the dots*, dan *Maze* yang sangat populer di seluruh dunia. Nikoli telah menerbitkan banyak puzzle melalui majalahnya, *Puzzle Communication Nikoli* dalam banyak edisi. Hingga tahun 2013, Nikoli telah menerbitkan lima buku yang sepenuhnya terdiri dari puzzle Heyawake. Puzzle ini pertama kali muncul dalam edisi #39 dari majalah *Puzzle Communication Nikoli* pada September 1992.



Gambar 1 Buku Nikoli Heyawake

Heyawake dimainkan di atas grid persegi panjang yang ukurannya bervariasi. Grid ini dibagi menjadi beberapa "ruangan" persegi panjang dengan ukuran berbeda-beda oleh garis tebal yang mengikuti tepi sel-sel. Beberapa ruangan mungkin memiliki satu angka yang biasanya dicetak di sel kiri atas; meskipun dalam desain awalnya setiap ruangan diberi angka, hal ini tidak lagi selalu diperlukan untuk menyelesaikan puzzle.

Beberapa sel dalam puzzle harus diwarnai hitam; tujuan dari puzzle ini adalah untuk menentukan apakah setiap sel harus diwarnai hitam atau dibiarkan putih. Dalam praktiknya, seringkali lebih mudah untuk menandai sel "kosong" yang sudah diketahui, misalnya dengan meletakkan titik di tengah sel.



Gambar 2 Gambar Puzzle Heyawake

Aturan yang menentukan mana sel yang diwarnai adalah sebagai berikut:

- Sel yang diwarnai tidak boleh terhubung secara ortogonal (tidak boleh berbagi sisi, meskipun dapat bersentuhan secara diagonal).
- Semua sel putih harus saling terhubung (membentuk satu polimino tunggal).

- Sebuah angka menunjukkan dengan tepat berapa banyak sel yang diwarnai dalam ruangan tersebut.
- Ruangan yang tidak memiliki angka dapat berisi berapa pun jumlah sel yang diwarnai, atau tidak ada sama sekali.
- Garis lurus (ortogonal) dari sel-sel putih yang terhubung tidak boleh mengandung sel dari lebih dari dua ruangan—dengan kata lain, garis sel putih yang menghubungkan tiga ruangan atau lebih tidak diperbolehkan.

Aturan yang unik dan membedakannya dengna puzzle lain yaitu aturan terakhir, memastikan bahwa setiap sel yang diwarnai hitam memastikan sel putih sekelilingnya tidak membentuk garis ortogonal dengan panjang lebih dari dua ruangan.

Langkah solusi pertama untuk memulai penyelesaian puzzle biasanya dimulai dengan ruangan dengan angka satu, nol, atau angka yang sama dengan jumlah sel dalam ruangannya.

II. LANDASAN TEORI

A. *Brute Force*

Algoritma *brute force* adalah pendekatan pemecahan masalah yang mencoba semua kemungkinan solusi untuk menemukan yang benar. Dalam konteks komputasi, *brute force* mengacu pada teknik mencari solusi dengan cara menguji satu per satu semua kemungkinan kombinasi. Algoritma ini tidak memerlukan kecerdasan atau heuristik tertentu; ia hanya mencoba setiap kemungkinan dan memeriksa apakah itu solusi yang benar.

Algoritma *brute force* memiliki kelebihan khusus yaitu pendekatannya yang sederhana dan mudah diimplementasikan sehingga tidak perlu strategi atau suatu bentuk heuristik kompleks yang ada dalam algoritma lain.

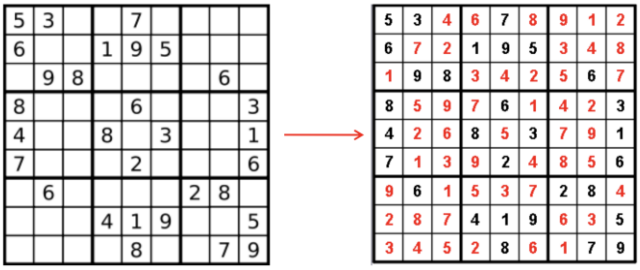
Kelebihan lain dari algoritma ini yaitu kepastiannya untuk mendapatkan solusi. Ketika algoritma lain mungkin tidak menemukan solusinya atau tidak menemukan solusi yang paling optimal sesuai persoalan, algoritma *brute force* pasti akan menemukan sebuah solusi terhadap setiap persoalan karena mencoba semua kemungkinan solusi dari permasalahan yang ada.

Akan tetapi, algoritma *brute force* memiliki beberapa kekurangan. Pertama, algoritmanya yang tidak efisien karena mencoba semua solusi yang ada, tidak seperti algoritma lain yang dapat menyempitkan ruang solusi (kemungkinan solusi) sehingga dapat lebih cepat dari algoritma *brute force*.

Kekurangan yang kedua yaitu skalabilitasnya. Karena algoritma *brute force* mencoba semua solusi dari ruang solusi permasalahannya, jika ruang solusi besar (jumlah kemungkinan solusi banyak), algoritma *brute force* akan memiliki kompleksitas waktu dan ruang yang sangat tinggi.

Salah satu contoh penggunaan algoritma *brute force* ada pada penyelesaian puzzle Sudoku. Algoritma *brute force* akan mengisi setiap sel dengan setiap kemungkinan angka dengan memeriksa aturan yang dimiliki Sudoku, mengubah angka pada sel dengan suatu urutan tertentu jika tidak memenuhi salah satu aturan Sudoku. Metode ini dilakukan untuk setiap sel dalam sebuah urutan.

Jika pada saat pengisian salah satu sel tidak dapat dimasukkan angka apapun dari 1-9, algoritma akan kembali pada sel yang sebelumnya diisi agar diubah. Ini dilakukan sampai solusi ditemukan, yaitu setiap sel dapat diisi dengan sebuah angka tanpa melanggar aturan Sudoku.



Gambar 3 Contoh Brute Force : Sudoku

Ada juga sebuah variasi dari algoritma *brute force* yaitu *exhaustive search*. Seperti algoritma *brute* force, *exhaustive search* adalah metode yang juga mencoba semua kemungkinan solusi, tetapi sering kali menggunakan teknik yang lebih terstruktur dibandingkan *brute force* untuk menghindari pengulangan dan persoalan kombinatorik.

Langkah yang digunakan pada *exhaustive search* adalah:

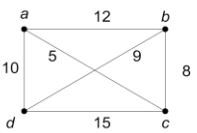
- Enumerasi (mendaftarkan) setiap kemungkinan solusi dengan cara yang sistematis.
- Evaluasi setiap kemungkinan solusi satu per satu, simpan solusi terbaik yang ditemukan sampai sejauh ini.
- Bila pencarian berakhir, dipilih solusi terbaik.

Algoritma *exhaustive search* dapat dipercepat dengan teknik heuristik, sebuah teknik untuk mengurangi jumlah kemungkinan solusi yang ada sehingga tidak semua harus dicoba dan ditelusuri, biasanya menggunakan teknik yang tidak formal, mengabaikan pembuktian tekniknya secara matematis.

Selain hal itu, algoritma ini memiliki sifat, kelebihan dan kekurangan yang sama dengan algoritma *brute force*.

Contoh dari algoritma *exhaustive search* adalah persoalan TSP (*Travelling Salesman Problem*) yang dilakukan dengan langkah:

1. Enumerasi semua sirkuit Hamilton dari graf lengkap.
2. Hitung bobot setiap sirkuit yang ditemukan.
3. Solusinya berupa sirkuit dengan bobot terkecil.



No.	Rute perjalanan (*tour*)	Bobot
1.	$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	12+8+15+10 = 45
2.	$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	12+9+15+5 = 41
3.	$\mathbf{a \rightarrow c \rightarrow b \rightarrow d \rightarrow a}$	5+8+9+10 = **32** → optimal
4.	$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	5+15+9+12 = 41
5.	$\mathbf{a \rightarrow d \rightarrow b \rightarrow c \rightarrow a}$	10+9+8+5 = **32** → optimal
6	$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	10+15+8+12 = 45

Gambar 4 Contoh Exhaustive Search : TSP

B. Backtracking

Algoritma *backtracking* adalah pendekatan sistematis untuk mencari solusi masalah dengan membangun solusi secara inkremental, satu langkah pada satu waktu, dan membatalkan langkah-langkah tersebut jika tidak mengarah ke solusi yang valid. Ini adalah teknik yang sering digunakan untuk masalah-masalah yang melibatkan pencarian solusi di ruang solusi yang besar.

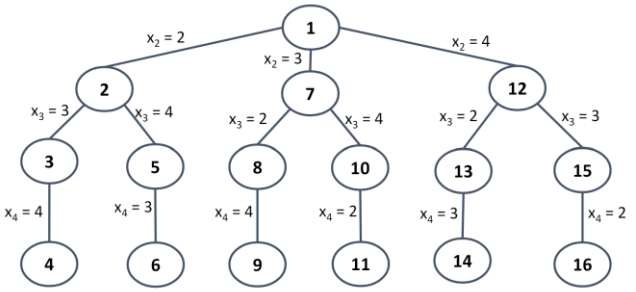
Algoritma ini adalah perbaikan dari *exhaustive search*, menelusuri pilihan yang mengarah ke solusi, memangkas (*pruning*) simpul yang tidak mengarah ke solusi.

Algoritma *backtracking* memiliki solusi dengan bentuk vektor dengan n-tuple, fungsi pembangkit (T()) untuk sebagai nilai setiap simpul, dan fungsi pembatas (B(x)) untuk menentukan arah ke solusi, memangkas jika bernilai *false*.

Semua kemungkinan solusi dari sebuah persoalan disebut dengan ruang solusi (*solution space*), diatur dan dibentuk dalam sebuah struktur pohon berakar. Setiap simpul melambangkan status persoalan, sisi diberi label x_i. Lintasan dari akar daun menunjukkan kemungkinan solusi, dapat terdiri dari beberapa jalur yang menjadi solusi disebut ruang solusi. Pohon ini diacu sebagai sebagai pohon ruang status (*state space tree*).

Solusi ditelusuri dengan membangkitkan simpul status untuk membentuk suatu lintasan dari akar ke daun. Pembangkitan solusi dilakukan dengan metode DFS (*Depth First Search*). Simpul yang ada juga dapat dikategorikan, simpul hidup (*live node*)untuk simpul yang sudah dibangkitkan, simpul E (*expand node*) untuk simpul yang sedang diperluas, simpul mati (*dead node*) untuk lintasan yang dibentuk yang tidak mengarah ke solusi, ditentukan dengan fungsi pembatas.

Simpul mati yang secara implisit melakukan *pruning* (memangkas) simpul anak-anaknya. Jika lintasan berakhir dengan simpul mati, proses dilanjutkan dengan *backtrack* ke simpul atasnya dan melanjutkan untuk simpul lainnya. Ini yang menjadi ciri utama sesuai namanya (*backtracking*) dan pencariannya berhenti ketika mencapai simpul tujuan (*goal node*) ketika solusi sudah ditemukan.



Gambar 5 Contoh Backtracking: Sirkuit Hamilton

Contoh dari penggunaan algoritma *backtracking* adalah menentukan semua sirkuit Hamilton dengan fungsi pembangkit simpul bertetangga yang belum dikunjungi dan fungsi pembatas berupa jalur yang memungkinkan untuk membangun sirkuit Hamilton ketika tidak ada simpul bertetangga yang belum dikunjungi tapi jumlah simpul dalam jalur belum sampai jumlah simpul pada graf.

Gambar 3 Contoh Backtracking: Sirkuit Hamilton

III. Implementasi

Before you begin to format your paper, first write and save the content as a separate text file. Keep your text and graphic files separate until after the text has been formatted and styled. Do not use hard tabs, and limit use of hard returns to only one return at the end of a paragraph. Do not add any kind of pagination anywhere in the paper. Do not number text heads-the template will do that for you.

Finally, complete content and organizational editing before formatting. Please take note of the following items when proofreading spelling and grammar:

A. Kelas

Untuk mewujudkan algoritma penyelesaian puzzle Heyawake, makalah ini menggunakan Bahasa Java dan digunakan beberapa kelas yaitu:

1. Cell

   Cell melambangkan setiap sel yang ada pada papan permianan. Cell akan mengandung beberapa data sesuai kebutuhan yaitu identitas grup (*region*) untuk "id" ruangannya, *value* untuk beberapa sel yang mendefinisikan jumlah sel hitam pada suatu ruangan, dan statusnya berupa *boolean* (*true* untuk sel hitam dan *false* untuk sel putih).

```
public class Cell {
    public int group;
    public int value = -1;
    public boolean status = false;
```

Gambar 6 Kelas Cell

2. Rules

   Kelas *Rules* digunakan untuk mengecek kesesuaian keadaan papan permainan dengan peraturan-peraturan yang ada pada puzzle Heyawake. Kelas ini hanya terdiri dari *method* untuk pengecekan aturan-aturannya.

```
// iterate, checking number of black cells
public static boolean isAmountValid(Cell[][] matrix) {
    Map<Integer, Integer> tempMap = MatrixFunctions.getRegionMap(matrix);
    for (Cell[] cellArray : matrix) {
        for (Cell cell : cellArray) {
            // count if coloured
            if (cell.getStatus()) {
                int tempGroup = cell.getGroup();
                tempMap.put(tempGroup, tempMap.get(tempGroup) - 1);
            }
        }
    }
    return tempMap.values().stream().allMatch(value -> value == 0);
}
```

Gambar 7 Metode isAmountValid

Metode ini memeriksa jumlah sel hitam pada setiap *region* dibandingkan dengan nilai yang tertera pada salah satu selnya, diimplementasikan dengan sebuah *map* antara id *region* dengan jumlahnya dengan metode getRegionMap pada kelas MatrixFunctions. Metode ini akan menelusuri seluruh matriks sekaligus karena memeriksa semua *region*.

```java
// check specific black cell cell for adjacent black cells
public static boolean existsAdjacentBlackCells(Cell[][] matrix, int row, int column) {
    // assume row and column is valid
    if (row != 0) {
        if (matrix[row - 1][column].getStatus()) {
            return true;
        }
    }
    if (column != 0) {
        if (matrix[row][column - 1].getStatus()) {
            return true;
        }
    }
    if (row != matrix.length - 1) {
        if (matrix[row + 1][column].getStatus()) {
            return true;
        }
    }
    if (column != matrix[row].length - 1) {
        if (matrix[row][column + 1].getStatus()) {
            return true;
        }
    }
    return false;
}
```

Gambar 8 Metode existsAdjacentBlackCells

Metode ini ada untuk memeriksa aturan sel hitam yang bersebelahan tidak boleh terjadi. Ini dilakukan dengan memeriksa setiap sel pada matriks sesuai indeks pada parameternya.

```java
private static boolean isWhiteAcrossThreeRegionsHelper(Cell[][] matrix, int row, int column, int rowDir,
        int colDir) {
    int itrRow = row;
    int itrColumn = column;
    int regionCounter = 1;
    List<Integer> visitedGroups = new ArrayList<>();
    visitedGroups.add(matrix[row][column].getGroup());

    while (true) {
        itrRow += rowDir;
        itrColumn += colDir;

        if (itrRow < 0 || itrRow >= matrix.length || itrColumn < 0 || itrColumn >= matrix[0].length) {
            break;
        }

        Cell tempCell = matrix[itrRow][itrColumn];
        if (tempCell.getStatus()) {
            break; // stop if black cell is encountered
        }

        if (!visitedGroups.contains(tempCell.getGroup())) {
            visitedGroups.add(tempCell.getGroup());
            regionCounter++;
        }

        if (regionCounter == 3) {
            return true;
        }
    }

    return false;
}
```

Gambar 9 Metode isWhiteAcrossThreeRegionHelper

Metode ini adalah metode pembantu untuk memeriksa aturan puzzle Heyawake lainnya yaitu tidak boleh adanya deretan sel putih yang melintasi tiga ruangan yang berbeda. Metode ini akan menghitung sampai tiga atau sampai bertemu dengan sel hitam secara rekursif berdasarkan arah yang tercatat pada parameternya (rowDir dan colDir).

```java
// check specific cell (changed cell) for white cells crossing 3 regions
public static boolean isWhiteAcrossThreeRegions(Cell[][] matrix, int row, int column) {
    // current row and column is white
    if (!matrix[row][column].getStatus()) {
        // check row-wise and column-wise
        return isWhiteAcrossThreeRegionsHelper(matrix, row, column, -1, colDir:0) || // row descending
                isWhiteAcrossThreeRegionsHelper(matrix, row, column, rowDir:1, colDir:0) || // row ascending
                isWhiteAcrossThreeRegionsHelper(matrix, row, column, rowDir:0, -1) || // column descending
                isWhiteAcrossThreeRegionsHelper(matrix, row, column, rowDir:0, colDir:1); // column ascending
    }
    return false;
}
```

Gambar 10 Metode isWhiteAcrossThreeRegions

Metode ini yang memanggil metode *helper* sebelumnya, memberikan parameter arahnya (kiri, kanan, atas, bawah) dari suatu sel matriks pada indeks baris dan kolom pad a parameternya. Metode ini akan mengembalikan *true* jika salah satu hasil dari keempat arah fungsi *helper* menghasilkan *true*.

```java
public static boolean isValid(Cell[][] matrix) {
    if (!isAmountValid(matrix)) {
        return false;
    }
    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < matrix[i].length; j++) {
            if (!isAmountValid(matrix)) {
                System.out.println(x:"Black cell amount not valid!");
            }
            if (isWhiteAcrossThreeRegions(matrix, i, j)) {
                System.out.println(x:"Exists white cells across three regions!");
            }
            if (existsAdjacentBlackCells(matrix, i, j)) {
                System.out.println(x:"Exists adjacent black cells!");
            }
            // black and adjacent
            if (matrix[i][j].getStatus() && existsAdjacentBlackCells(matrix, i, j)) {
                return false;
            }
            // white and whiteAcrossThreeRegions
            if (!matrix[i][j].getStatus() && isWhiteAcrossThreeRegions(matrix, i, j)) {
                return false;
            }
        }
    }
    return isConnected(matrix);
}
```

Gambar 11 Metode isValid

Metode ini ada untuk menggabungkan semua metode untuk aturan sebagai satu metode utuh. Pada metode ini akan dipanggil ketiga metode sebelumnya sesuai cirinya masing-masing. Untuk metode existsAdjacentBlackCells dan metode isWhiteAcrossThreeRegions dilakukan pada setiap sel dengan iterasi, sementara metode isAmountValid dipanggil sekali pada akhir sehingga jika kedua metode untuk setiap iterasi benar, maka hasil validitasnya akan bergantung sepenuhnya pada metode isAmountValid.

```java
// check if all white cells are connected
public static boolean isConnected(Cell[][] matrix) {
    boolean[][] visited = new boolean[matrix.length][matrix[0].length];
    int whiteCellCount = 0;

    // Find the first white cell to start the flood fill
    int startRow = -1;
    int startCol = -1;
    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < matrix[i].length; j++) {
            if (!matrix[i][j].getStatus()) {
                startRow = i;
                startCol = j;
                whiteCellCount++;
            }
        }
    }

    // if no white cells are found, the puzzle is trivially connected
    if (startRow == -1) {
        return true;
    }

    // flood fill to count connected white cells
    int connectedWhiteCells = floodFill(matrix, visited, startRow, startCol);

    // check if all white cells are connected
    return connectedWhiteCells == whiteCellCount;
}

private static int floodFill(Cell[][] matrix, boolean[][] visited, int row, int col) {
    if (row < 0 || col < 0 || row >= matrix.length || col >= matrix[0].length || visited[row][col]
            || matrix[row][col].getStatus()) {
        return 0;
    }

    visited[row][col] = true;
    int count = 1; // Current cell

    // visit all directions (left,right,up,down)
    count += floodFill(matrix, visited, row - 1, col);
    count += floodFill(matrix, visited, row + 1, col);
    count += floodFill(matrix, visited, row, col - 1);
    count += floodFill(matrix, visited, row, col + 1);

    return count;
}
```

Gambar 12 Metode isConnected

Aturan terakhir pada puzzle Heyawake yang harus diperiksa yaitu keterhubungan antar sel putih, memastikan semua sel putih yang ada pada papan permainan terhubung satu sama selain secara ortogonal. Untuk mewujudkan itu, digunakan metode isConnected dan floodFill.

Metode floodFill adalah metode pembantu dari metode isConnected yang berfungsi untuk menjelajah semua sel dalam setiap arah secara rekursif. sementara metode isConnected berfungsi untuk mencari sel putih pertama untuk dijadikan patokan dan memastikan jumlah akhir dari sel putih yang terhubung sama dengan jumlah sel putih yang ada pada seluruh papan.

```java
public abstract class Algorithm {
    // matrix data for direct access and modification
    public Cell[][] matrix;

    // matrix already contains groups and values
    public Algorithm(Cell[][] matrix) {
        this.matrix = matrix;
    }

    public abstract boolean solve(int row, int column);
}
```

Gambar 13 Kelas Algorithm

3. Algorithm

Setiap algoritma yang akan digunakan (*Brute Force* dan *Backtracking*) akan dibuat menjadi suatu kelas, dengan *parent class* Algorithm, memiliki atribut matriks untuk memperbolehkan akses pada matriks secara langsung.

Kelas Algorithm juga memiliki satu buah metode abstrak solve yang akan diimplementasikan oleh setiap algoritma yang nanti mewarisi kelas Algorithm, mengembalikan sebuah *boolean*, dapat memperoleh hasil atau tidak.

```java
import java.util.Scanner;
import java.util.HashMap;
import java.util.Map;

public class Game {
    private Cell[][] matrix;
    private Scanner scanner;

    public Game() {
        scanner = new Scanner(System.in);
    }

    public void startGame() {
        System.out.println(x:"Heyawake Puzzle:");
        System.out.println();
    }
```

Gambar 14 Kelas Game

```java
public void inputMatrix() {
    // input matrix dimensions
    System.out.print(s:"Number of Rows: ");
    int rows = scanner.nextInt();
    System.out.print(s:"Number of Columns: ");
    int columns = scanner.nextInt();
    scanner.nextLine();

    System.out.println();

    // initialize matrix
    this.matrix = new Cell[rows][columns];

    // initialize map for region <-> number of black cells mapping
    Map<Integer, Integer> tempInputMap = new HashMap<>();

    // input matrix values
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < columns; j++) {
            System.out.println("Matrix [" + (i + 1) + "][" + (j + 1) + "]:");
            System.out.print(s:"    Region: ");
            // assume region always positive integer
            int tempRegion = scanner.nextInt();
            scanner.nextLine();
            // assume value always integer
            do {
                System.out.print(s:"    Value (-1 for valueless): ");
                int tempValue = scanner.nextInt();
                scanner.nextLine();
                if (tempInputMap.containsKey(tempRegion) && tempValue != -1) {
                    System.out.println(x:"Value for region already exists!");
                } else {
                    tempInputMap.put(tempRegion, tempValue);
                    matrix[i][j] = new Cell(tempRegion, tempValue);
                    break;
                }
            } while (true);
        }
    }
    System.out.println();
}
```

Gambar 15 Metode inputMatrix

Kelas Game ini ada untuk mengatur keberjalanan permainannya, dari permulaan, konfigurasi, penyelesaian, hingga akhir dari permainannya.

Awal dan akhir permainan hanya mengeluarkan pesan biasa. Metode inputMatrix digunakan untuk meminta nilai dari setiap sel pada matriks, meminta data id grup dan *value* (jika ada, -1 jika tidak ada angkanya), dan data status memiliki nilai *false* untuk setiap matriks sebagai konfigurasi awal, menandakan sel putih yang belum diubah sama sekali.

Setelah metode inputMatrix, ada metode solve pada kelas Game, berbeda dengan pada kelas Algorithm, metode solve ini adalah untuk rangkaian pelaksanaan pencarian solusi untuk puzzle Heyawake, diawali dengan masukan jenis algoritma (dibuat asumsi selalu valid), pencatatan waktu dan memori awal, dilanjutkan dengan pemanggilan algoritma sesuai pilihan, penampilan solusi, pencatatan akhir memori dan waktu akhir, dan menampilkan waktu dan memori yang digunakan sepanjang algoritma penyelesaian puzzle dijalankan.

Scanner untuk menerima input dari pengguna juga dibuat sebagai sebuah atribut pada kelas Game agar memudahkan pengelolaan *scanner*, memastikan semua input diterima setelah permainan dimulai dan *scanner* ditutup pada akhir permainan, setelah mendapatkan solusi.

Waktu dan tempat yang digunakan pada setiap prosesnya dilihat untuk membandingkan kedua algoritma yang

digunakan dan menentukan algoritma yang paling tepat dan efektif untuk penyelesaian puzzle Heyawake.

```java
// check if all white cells are connected
public static boolean isConnected(Cell[][] matrix) {
    boolean[][] visited = new boolean[matrix.length][matrix[0].length];
    int whiteCellCount = 0;

    // Find the first white cell to start the flood fill
    int startRow = -1;
    int startCol = -1;
    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < matrix[i].length; j++) {
            if (!matrix[i][j].getStatus()) {
                startRow = i;
                startCol = j;
                whiteCellCount++;
            }
        }
    }

    // if no white cells are found, the puzzle is trivially connected
    if (startRow == -1) {
        return true;
    }

    // flood fill to count connected white cells
    int connectedWhiteCells = floodFill(matrix, visited, startRow, startCol);

    // check if all white cells are connected
    return connectedWhiteCells == whiteCellCount;
}

private static int floodFill(Cell[][] matrix, boolean[][] visited, int row, int col) {
    if (row < 0 || col < 0 || row >= matrix.length || col >= matrix[0].length || visited[row][col]
            || matrix[row][col].getStatus()) {
        return 0;
    }

    visited[row][col] = true;
    int count = 1; // Current cell

    // visit all directions (left,right,up,down)
    count += floodFill(matrix, visited, row - 1, col);
    count += floodFill(matrix, visited, row + 1, col);
    count += floodFill(matrix, visited, row, col - 1);
    count += floodFill(matrix, visited, row, col + 1);

    return count;
}
```

Gambar 16 Metode isConnected

Aturan terakhir pada puzzle Heyawake yang harus diperiksa yaitu keterhubungan antar sel putih, memastikan semua sel putih yang ada pada papan permainan terhubung satu sama selain secara ortogonal. Untuk mewujudkan itu, digunakan metode isConnected dan floodFill.

Metode floodFill adalah metode pembantu dari metode isConnected yang berfungsi untuk menjelajah semua sel dalam setiap arah secara rekursif. sementara metode isConnected berfungsi untuk mencari sel putih pertama untuk dijadikan patokan dan memastikan jumlah akhir dari sel putih yang terhubung sama dengan jumlah sel putih yang ada pada seluruh papan.

```java
public void solve() {
    System.out.println(x:"Algorithm options:");
    System.out.println(x:"1. Brute Force");
    System.out.println(x:"2. Backtracking");
    System.out.println();
    System.out.print(s:"Input algorithm option: ");
    int algorithm = scanner.nextInt();
    scanner.nextLine();

    System.out.println();

    // runtime space
    Runtime runtime = Runtime.getRuntime();
    long initialMemory = runtime.totalMemory() - runtime.freeMemory();
    // start execution time
    long startTime = System.currentTimeMillis();

    MatrixFunctions.printMatrix(matrix);

    if (algorithm == 1) {
        new BruteForce(matrix).solve(row:0, column:0);
    } else if (algorithm == 2) {
        new Backtracking(matrix).solve(row:0, column:0);
    }

    MatrixFunctions.printMatrix(matrix);

    System.out.println();

    // solution validation
    if (!Rules.isValid(matrix)) {
        System.out.println(x:"No solution exists!");
    } else {
        System.out.println(x:"Solution:");
        MatrixFunctions.printMatrix(matrix);
    }

    System.out.println();

    // end runtime space
    long finalMemory = runtime.totalMemory() - runtime.freeMemory();
    // end execution time
    long endTime = System.currentTimeMillis();

    // evaluating time and space
    long memoryUsed = finalMemory - initialMemory;
    System.out.println("Memory used: " + memoryUsed + " bytes");
    long elapsedTime = (endTime - startTime) / 1000000;
    System.out.println("Execution time: " + elapsedTime + " milliseconds");

    System.out.println();
}

public void endGame() {
    System.out.println(x:"Game ends!");
    scanner.close();
}
```

Gambar 17 Metode solve dan endGame

B. Implementasi Brute Force

Seperti dijelaskan di atas, setiap algoritma yang akan digunakan (*Brute Force* dan *Backtracking*) dijadikan sebuah kelas turunan dari kelas *Algorithm* yang akan mengimplementasikan metode *solve*.

Pada algoritma *Brute Force*, metode *solve* akan dilakukan secara rekursif, menelusuri dan mengecek setiap sel individu dengan urutan dari kiri atas hingga kanan bawah, memiliki 2 parameter *integer* melambangkan indeks baris dan indeks kolom dari matriks yang sedang ditelusuri.

Penelusuran rekursifnya secara umum akan berjalan dari kiri ke kanan, atas ke bawah. Jika indeks kolom sama dengan banyak kolom papan permainan, indeks baris akan ditambahkan. Jika indeks baris sudah sama dengan jumlah baris yang ada pada papan permainan, papan akan diperiksa secara menyeluruh dengan menggunakan kelas *MatrixFunctions* kesesuaiannya dengan peraturan-peraturan puzzle Heyawake.

Setiap kemungkinan pada solusi puzzle Heyawake akan ditelusuri dengan pemanggilan rekursif pada baris dan kolom berikutnya. Jika hasil pada pemanggilan rekursi terbelakang *true* sesuai peraturan, pemanggil rekursi akan mengembalikan nilai *true* juga, terus berantai hingga awal, menyatakan solusi yang benar untuk puzzle.

Evaluasi dari setiap solusi ada pada akhir saja (kolom dan baris terakhir) sehingga setiap solusi akan tetap dienumerasi oleh program.

```java
// recursive
public boolean solve(int row, int column) {
    // all cells filled
    if (row == matrix.length) {
        return Rules.isValid(matrix);
    }

    // move to next row
    if (column == matrix[row].length) {
        return solve(row + 1, column:0);
    }

    // Try setting the cell to white
    matrix[row][column].setStatus(status:false);
    ;
    if (solve(row, column + 1)) {
        return true;
    }

    // Try setting the cell to black
    matrix[row][column].setStatus(status:true);
    ;
    if (solve(row, column + 1)) {
        return true;
    }

    return false;
}
```

Gambar 18 Metode solve Algoritma Brute Force

C. *Implementasi Backtracking*

Seperti dijelaskan di atas, setiap algoritma yang akan digunakan (*Brute Force* dan *Backtracking*) dijadikan sebuah kelas turunan dari kelas *Algorithm* yang akan mengimplementasikan metode *solve*.

Pada algoritma *Backtracking*, metode *solve* akan dilakukan secara rekursif, dan serupa dengan *brute force*, menelusuri dan mengecek setiap sel individu dengan urutan dari kiri atas hingga kanan bawah, memiliki 2 parameter *integer* melambangkan indeks baris dan indeks kolom dari matriks yang sedang ditelusuri.

Perbedaan kedua algoritma terdapat pada validasinya. Jika algoritma *brute force* memvalidasi setiap jalur di panggilan terakhir rekursi, algoritma *backtracking* memvalidasi pada setiap tahap rekursi, memanggil rekursi berikutnya jika masih valid dan membalikkan keadaan semula jika tidak valid sehingga akan memangkas seluruh anak-anak simpul (rekursi selanjutnya) sejak simpul menyatakan tidak valid.

```java
// recursive
public boolean solve(int row, int column) {
    // all cells filled
    if (row == matrix.length) {
        return true;
    }

    // move to next row
    if (column == matrix[row].length) {
        return solve(row + 1, column:0);
    }

    // try both black (1) and white (0)
    for (int val = 0; val <= 1; val++) {
        boolean originalStatus = matrix[row][column].getStatus();
        matrix[row][column].status = (val == 1);

        // validate and recurse, returning true if valid
        if (Rules.isValid(matrix) && solve(row, column + 1)) {
            return true;
        }

        // reset to original status (backtrack) if not valid
        matrix[row][column].status = originalStatus;
    }

    return false;
}
```

Gambar 19 Metode solve Algoritma Backtracking

IV. PERCOBAAN DAN PEMBAHASAN

Percobaan akan melihat perbandingan antara kedua algoritma dan pendekatannya yang berbeda dalam beberapa kasus.

1. Test Case 1 : 1 x 1 Puzzle

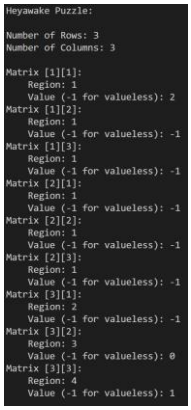
```
Heyawake Puzzle:

Number of Rows: 1
Number of Columns: 1

Matrix [1][1]:
    Region: 1
    Value (-1 for valueless): 1

Algorithm options:
1. Brute Force
2. Backtracking

Input algorithm option: 1

G:1 V:1 S:false

Solution:
G:1 V:1 S:true

Memory used: 461464 bytes
Execution time: 8673 microseconds

Game ends!
```

Gambar 20 Test Case 1 (Brute Force)

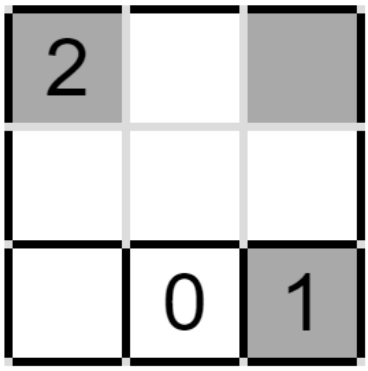
```
Heyawake Puzzle:

Number of Rows: 1
Number of Columns: 1

Matrix [1][1]:
    Region: 1
    Value (-1 for valueless): 1

Algorithm options:
1. Brute Force
2. Backtracking

Input algorithm option: 2

G:1 V:1 S:false

Solution:
G:1 V:1 S:true

Memory used: 461464 bytes
Execution time: 8654 microseconds

Game ends!
```

Gambar 21 Test Case 1 (Backtracking)

2. Test Case 2 : 3 x 3 Puzzle



Gambar 22 Test Case 2 (Input)



Gambar 23 Test Case 2 (Gambaran)

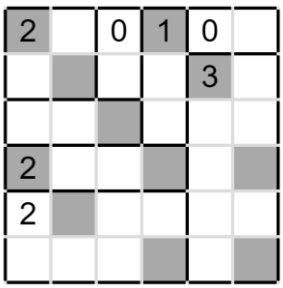
```
Solution:
G:1 V:2 S:false G:1 V:-1 S:false G:1 V:-1 S:true
G:1 V:-1 S:true G:1 V:-1 S:false G:1 V:-1 S:false
G:2 V:-1 S:false G:3 V:0 S:false G:4 V:1 S:true

Memory used: 0 bytes
Execution time: 19726 microseconds
```

Gambar 24 Test Case 2 (Brute Force)

```
No solution exists!

Memory used: 0 bytes
Execution time: 3765 microseconds

Game ends!
```

Gambar 25 Test Case 2 (Backtracking)

3. Test Case 3 : 6 x 6 puzzle



Gambar 26 Test Case 3 (Gambaran)

```
No solution exists!

Memory used: 0 bytes
Execution time: 5826 microseconds

Game ends!
```

Gambar 27 Test Case 3 (Backtracking)

V. KESIMPULAN

Dalam dua pendekatan berbeda yang digunakan pada makalah ini untuk menemukan penyelesaian dari puzzle Heyawake, dapat disimpulkan bahwa algoritma *backtracking* jauh lebih cepat dan efisien. Hal ini dikarenakan algoritma *brute force* yang menelusuri setiap kemungkinan solusi dari puzzle Heyawake sementara algoritma *backtracking* memungkinkan adanya *pruning* (pemangkasan) simpul-simpul, mengurangi jumlah solusi yang ditelusuri.

Ada juga *test case* yang menunjukkan bahwa solusi pada algoritma *brute force* ada tetapi solusi pada algoritma *backtracking* tidak ada. Ini bisa terjadi karena nilai *default* awal yang putih untuk setiap sel menganggu validasi sehingga seharusnya nilai awal berupa selain putih dan hitam (masukan untuk penelitian selanjutnya) sehingga pada semua cabang langsung ditentukan bahwa solusi salah.

VIDEO LINK AT YOUTUBE *(Heading 5)*

https://youtu.be/tTHRlyvguhM

ACKNOWLEDGMENT *(Heading 5)*

Penulis ingin mengucapakan terimakasih kepada:

- Tuhan yang Maha Esa karena penyertaan-Nya dapat memungkinan penulis untuk menyelesaikan makalah ini.
- Orang tua karena dengan dukungan mereka maka penulis dapat menempuh studi dan memungkinkan untuk membuat dan menyelesaikan mata kuliah dan makalah ini.
- Dr. Ir. Rinaldi Munir, M.T., Dr. Nur Ulfa Maulidevi, dan Dr. Ir. Rila Mandala, para dosen mata kuliah Strategi Algoritma tahun 2023/2024 untuk semua ilmu yang diperoleh sehingga penulis dapat menyelesaikan makalah ini.

REFERENCES

[1] Munir, Rinaldi. 2021. "Algoritma Brute Force (Bagian 1)".https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Brute-Force-(2022)-Bag1.pdf, diakses 12 Juni 2024

[2] Munir, Rinaldi. 2021. "Algoritma Brute Force (Bagian 2)".https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Brute-Force-(2022)-Bag2.pdf, diakses 12 Juni 2024

[3] 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 12 Juni 2024

[4] Munir, Rinaldi. 2021. “Algoritma Runut-balik (Backtracking) (Bagian 2)”. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-backtracking-2021-Bagian2.pdf, diakses 12 Juni 2024

[5] Nikoli Website about Heyawake. https://www.nikoli.co.jp/en/puzzles/heyawake, diakses 10 Juni 2024

[6] Heyawake Puzzles. https://www.puzzle-heyawake.com, diakses 8 Juni 2024

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, 12 Juni 2024

Zachary Samuel Tobing 13522016