Strategi Penyelesaian Puzzle Color Connect Menggunakan Algoritma A* dan Backtracking

Theo Kurniady - 13523154
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: theokurniady12@gmail.com , 13523154@std.stei.itb.ac.id

***Abstrak*—Puzzle Color Connect merupakan permainan berbasis grid yang menantang pemain untuk menghubungkan pasangan titik berwarna tanpa saling bertabrakan dan mengisi seluruh area papan. Permasalahan ini dapat dimodelkan sebagai persoalan pencarian jalur, yang menuntut pencarian solusi optimal secara efisien. Dalam makalah ini, penulsi mengusulkan pendekatan penyelesaian dengan menggunakan algoritma A star yang dikombinasikan dengan strategi runut balik atau *backtracking* untuk menjelajahi kemungkinan jalur secara menyeluruh namun terarah. Algoritma A star menggunakan heuristik jarak Manhattan untuk mempercepat proses pencarian menuju titik tujuan, sedangkan *backtracking* digunakan untuk menangani kondisi kegagalan parsial dan memastikan solusi global tetap memungkinkan. Hasil implementasi menunjukkan bahwa pendekatan ini mampu menyelesaikan *puzzle* dengan efisien, terutama dalam puzzle berskala menengah hingga besar.**

***Kata Kunci*—Color Connect; Puzzle; Algoritma A*; Backtracking; Jarak Manhattan**

I. Pendahuluan

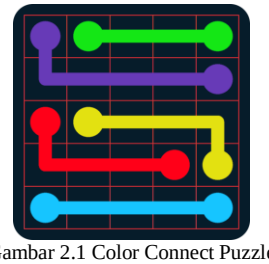
Permainan logika seperti Color Connect menjadi tantangan menarik dalam bidang pemrograman karena membutuhkan kemampuan eksplorasi solusi yang optimal di bawah batasan tertentu. Pada permainan ini, pemain diberikan sepasang titik-titik berwarna dalam sebuah grid dua dimensi. Pemain harus menghubungkan pasangan titik tersebut dengan jalur yang tidak bertabrakan, serta mengisi seluruh grid untuk menyelesaikan permainan.

Secara komputasional, penyelesaian Color Connect dapat dikategorikan sebagai persoalan pencarian jalur (*pathfinding*) dengan banyak kombinasi kemungkinan, sehingga tidak efisien diselesaikan dengan pencarian brute-force, yang sangat memakan waktu dan tidak efisien. Oleh karena itu, algoritma pencarian yang mampu menyeimbangkan eksplorasi dan efisiensi waktu yang diperlukan.

Algoritma A star atau A* dikenal sebagai salah satu algoritma pencarian jalur yang optimal dan efisien karena memanfaatkan fungsi heuristik untuk memprioritaskan jalur yang optimal. Akan tetapi, dalam permainan ini, hanya ada satu solusi untuk satu permasalahan dan jalur yang optimal belum tentu menjadi solusi yang valid. Oleh karena itu, A* dikombinasikan dengan *backtracking* untuk memungkinkan pemulihan dan eksplorasi ulang ketika solusi parsial menemui jalan buntu. Makalah ini akan membahas dan mengimplementasikan bagaimana algoritma A* diimplementasikan dalam menyelesaikan permainan Color Connect. Implementasi akan dibuat sebagai aplikasi menggunakan bahasa pemrograman Java.

II. Teori Dasar

A. Color Connect

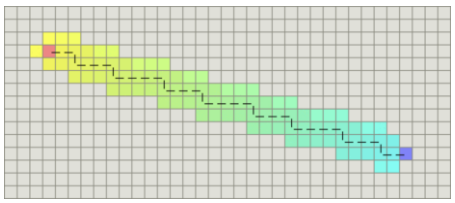


Gambar 2.1 Color Connect Puzzle
(Sumber: Connect Colors - Apps on Google Play)

Gambar 2.1 menunjukkan contoh permainan color connect pada sebuah grid 5x5. Color Connect atau dikenal juga sebagai Flow Free merupakan permainan puzzle logika di mana pengguna harus menghubungkan pasangan titik berwarna pada sebuah grid, tanpa saling menabrak atau melewati jalur warna lainnya. Aturannya meliputi:

1. Tiap pasangan warna harus terhubung dengan jalur kontinu.
2. Jalur tidak boleh saling bersilangan.
3. Umumnya seluruh sel pada grid harus terisi jalur.

B. *Algoritma A**



Gambar 2.1 Algoritma A*
(Sumber: https://paul.pub/a-star-algorithm/)

A* adalah algoritma pencarian jalur yang digunakan untuk menemukan rute terpendek dari satu titik ke titik tujuan, dengan efisiensi tinggi. Gambar 2.1 menunjukkan contoh sebuah pencarian jalur dari satu titik ke titik lain dengan jalur

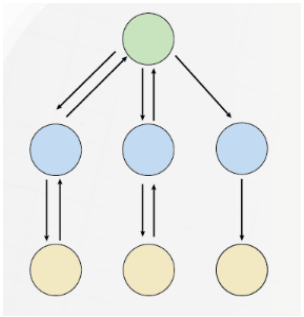
yang efisien. Algoritma ini merupakan gabungan atau perluasan dari algoritma Uniform Cost Search (UCS) dan Greedy Best First Search (GBFS). Algoritma A* memiliki fungsi evaluasi sebagai berikut

$$f(n) = g(n) + h(n)$$

,yang mana g(n) adalah biaya dari titik awal ke node saat ini dan h(n) adalah estimasi biaya dari node saat ini ke tujuan berdasarkan heuristik yang digunakan. f(n) sendiri adalah estimasi jumlah biaya jalur dari node saat ini menuju tujuan.

Dalam makalah ini, heuristik yang akan digunakan adalah jarak Manhattan. Dalam algoritma ini, heuristik h(n) harus *admissible*, yang artinya tidak melebihi jarak sebenarnya, agar algoritma tetap optimal. Algoritma ini dapat digunakan pada puzzle Color Connect karena dapat memprioritaskan jalur terpendek yang berpotensi mempercepat proses pencarian solusi.

C. Algoritma Runut Balik / *Backtracking*



Gambar 2.3 *Backtracking*
(Sumber: https://www.geeksforgeeks.org/dsa/backtracking-algorithms/)

Gambar 2.3 menunjukkan contoh sebuah pencarian dengan *backtracking*, yang mana pencarian kembali ke node sebelumnya ketika mencapai node yang tidak berujung pada solusi. *Backtracking* adalah metode pencarian solusi dengan pendekatan rekursif dan *trial-and-error*. Teknik ini mencoba membangun solusi secara bertahap dan mundur (backtrack) jika solusi sementara tidak valid. *Backtracking* memiliki langkah umum sebagai berikut.

1. Coba langkah awal, yaitu membangun jalur untuk warna pertama
2. Periksa bila jalur melanggar constraint.
3. Bila valid, lanjut ke langkah berikutnya.
4. Bila tidak valid, kembali ke langkah sebelumnya dan coba pilihan alternatif.

Backtracking memiliki kelebihan dan kekurangan. Algoritma ini mampu menjelajahi seluruh ruang solusi dan cocok untuk permasalahan yang memiliki solusi akhir yang tidak dapat dicapai hanya dengan memilih langkah terbaik pada setiap langkah, tanpa mempertimbangkan konsekuensi jangka panjang. Namun, algoritma ini akan memakan waktu jika tidak dibantu heurisitk atau secara bruteforce.

D. *Heuristik Jarak Manhattan*

Jarak Manhattan adalah sebuah metode pengukuran jarak antara dua titik dalam grid dua dimensi, berdasarkan total jumlah langkah horizontal dan vertikal yang diperlukan untuk mencapai titik akhir (x2, y2) dari titik awal (x1, y1).

$$\text{Jarak Manhattan} = |x1 - x2| + |y1 - y2|$$

III. Dekomposisi Masalah

Pada bagian ini, penulis menjelaskan pemecahan masalah sebelum menyelesaikan permasalahan color connect. Untuk mempermudah proses dan algoritma dari penyelesaian *puzzle* ini, komponen dari permainan akan dipisah menjadi beberapa struktur data. Struktur data yang pertama adalah grid, yang merupakan papan permainan color connect yang diisi dengan titik-titik pasangan warna yang akan disambung dalam area grid.

```java
public class Grid {
  private int rows, cols;
  private char[][] grid;

  public Grid(int rows, int cols, char[][] grid) {
      this.rows = rows;
      this.cols = cols;
      this.grid = grid;
  }
```

Gambar 3.1 Grid.java
(Sumber: https://github.com/TKurr/Makalah_Stima)

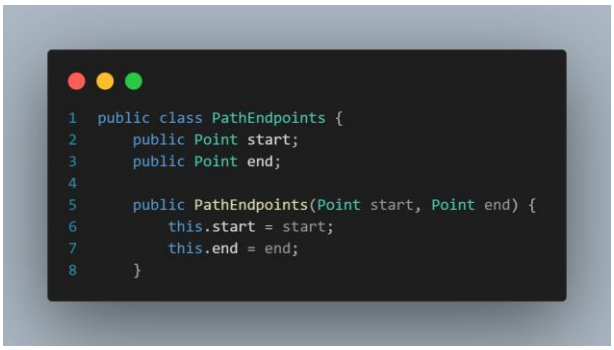
Gambar 3.1 menunjukkan kelas dan struktur data Grid. Grid berisi jumlah baris, kolom, dan elemen pada setiap sel grid yang disimpan dalam sebuah matriks dua dimensi. Jumlah baris dan kolom membantu menentukan ukuran dari grid pada puzzle. Kelas ini juga memiliki getter dan setter untuk membantu mengelola kelas ini ketika digunakan pada pencarian solusi.

```java
public class Point {
  private final int row;
  private final int col;

  public Point(int row, int col) {
      this.row = row;
      this.col = col;
  }

  @Override
  public boolean equals(Object o) {
      if (this == o) return true;
      if (!(o instanceof Point)) return false;
      Point other = (Point) o;
      return this.row == other.row && this.col == other.col;
  }

  public static int manhattan(Point a, Point b) {
    return Math.abs(a.getRow() - b.getRow()) + Math.abs(a.getCol() - b.getCol());
  }

  public static int manhattan(int r1, int c1, int r2, int c2) {
    return Math.abs(r1 - r2) + Math.abs(c1 - c2);
  }
```

Gambar 3.2 Point.java
(Sumber: https://github.com/TKurr/Makalah_Stima)

Gambar 3.2 menunjukkan kelas dan struktur data Point. Point berisi informasi baris dan kolom dari letak poin tersebut.

Kelas ini merepresentasikan titik warna yang ada pada grid. Point juga memiliki metode tambahan untuk perbandingan antar point dan metode untuk mencari jarak manhattan yang akan digunakan sebagai heuristik dalam algoritma. Selain itu, kelas ini juga dilengkapi getter dan setter untuk membantu mengelola kelas dari solver.



```java
public class PathEndpoints {
    public Point start;
    public Point end;

    public PathEndpoints(Point start, Point end) {
        this.start = start;
        this.end = end;
    }
```

Gambar 3.3 PathEndpoints.java
(Sumber: https://github.com/TKurr/Makalah_Stima)

Gambar 3.3 menunjukkan kelas dan struktur data PathEndpoints. PathEndpoints berisi informasi titik awal dan titik akhir dari sebuah pasangan warna, yang masing-masing titik dalam bentuk kelas point. Selain itu, kelas ini juga dilengkapi getter dan setter untuk membantu mengelola kelas dari solver.

```java
public class Solver {

    private final Grid grid;
    private final Map<Character, List<Point>> endpoints;

    private static final int[][] DIRS = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};

    public Solver(Grid grid) {
        this.grid = grid;
        this.endpoints = findEndpoints(grid);
    }

    private Map<Character, List<Point>> findEndpoints(Grid grid) {
        Map<Character, List<Point>> map = new HashMap<>();
        char[][] data = grid.getGrid();

        for (int row = 0; row < grid.getRows(); row++) {
            for (int col = 0; col < grid.getCols(); col++) {
                char ch = data[row][col];
                if (Character.isLetter(ch)) {
                    map.computeIfAbsent(ch, k -> new ArrayList<>()).add(new Point(row, col));
                }
            }
        }

        return map;
    }

    private boolean inBounds(int r, int c) {
        return r >= 0 && c >= 0 && r < grid.getRows() && c < grid.getCols();
    }

    private void sleep(long ms) {
        try { Thread.sleep(ms); } catch (InterruptedException ignored) {}
    }

    private char[][] copyGrid() {
        return copyGrid(grid.getGrid());
    }

    private char[][] copyGrid(char[][] original) {
        char[][] copy = new char[original.length][];
        for (int i = 0; i < original.length; i++) {
            copy[i] = Arrays.copyOf(original[i], original[i].length);
        }
        return copy;
    }
```

Gambar 3.4 Solver.java
(Sumber: https://github.com/TKurr/Makalah_Stima)

Gambar 3.4 menunjukkan kelas dan struktur data Solver. Solver adalah kelas utama yang digunakan untuk melakukan pencarian solusi puzzle. Solver berisi informasi grid dan sebuah map bernama endpoints yang berisi character sebagai key dan list of point sebagai value. Character yang dimaksud pada solver adalah pasangan warna yang unik. Untuk mempermudah pembacaan puzzle dari file .txt, penulis memisalkan warna sebagai karakter (seperti A, B, C, dst.). Value pada map berupa list of points karena setiap warna yang unik memiliki dua titik, yaitu titik awal dan titik akhir. Kelas solver juga memiliki informasi arah yang dapat ditelusuri ketika pencarian dalam sebuah matriks integer bernama DIRS. Matriks ini merupakan list dari list of integer, yang berarti list yang mengandung informasi pasangan perpindahan pada sumbu x dan y. Dalam setiap pasangan ini, indeks pertama menunjukkan sumbu x, sedangkan indeks kedua menunjukkan sumbu y.

Solver memiliki metode tambahan diluar algoritma pencarian, seperti findEndpoints. Metode findEndpoints digunakan untuk mendapatkan setiap pasangan titik untuk setiap karakter/warna dari grid dengan melakukan interasi pada grid. Hasil kemudian disimpan dalam map endpoints. Selain itu, solver juga memiliki metode inBounds untuk mengecek apakah suatu posisi (x, y) berada di dalam batas grid. Untuk melakukan deep copy, solver memiliki metode copyGrid untuk menyalin isi grid sekarang atau menyalin isi suatu grid lain.

IV. Algoritma Penyelesaian Masalah

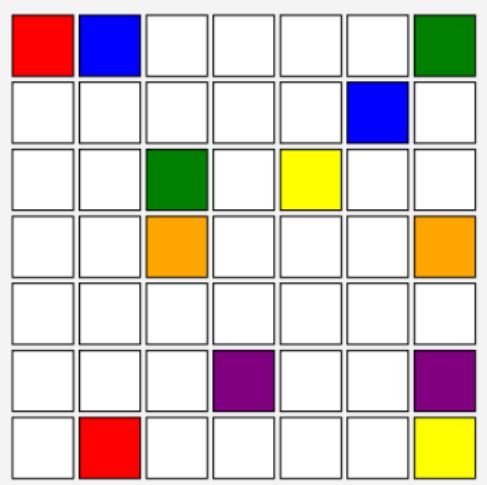
Pada bagian ini, penulis menjelaskan pendekatan algoritmik yang digunakan untuk menyelesaikan permasalahan Color Connect. Secara garis besar, penyelesaian menggunakan dua algoritma, yaitu A* dan *backtracking*. algoritma A* (A-Star) digunakan sebagai pendekatan utama dalam pencarian jalur antar pasangan titik. Algoritma A* dipilih karena mampu memberikan solusi yang optimal dan efisien, dengan mempertimbangkan estimasi biaya dari titik awal ke titik tujuan melalui fungsi heuristik. Heuristik yang digunakan dalam implementasi ini adalah jarak Manhattan, yaitu jumlah langkah horizontal dan vertikal minimum yang diperlukan untuk mencapai titik tujuan dari titik saat ini.

Proses penyelesaian dilakukan dengan pendekatan rekursif dan backtracking, di mana sistem mencoba menyambungkan satu per satu pasangan titik warna secara berurutan, dan jika suatu jalur tidak memungkinkan karena menghambat pasangan berikutnya, maka sistem akan kembali (backtrack) dan mencoba jalur alternatif. Dengan menggabungkan strategi pencarian jalur A* untuk setiap pasangan titik dan strategi backtracking untuk mempertahankan konsistensi seluruh grid, algoritma ini mampu menjelajahi ruang solusi secara sistematis namun tetap mengarahkan pencarian pada jalur yang menjanjikan.

```java
public boolean solveWith(String method,
                         Consumer<char[][]> onStep) {
    List<Map.Entry<Character, List<Point>>> sortedPairs =
                    new ArrayList<>(endpoints.entrySet());

    sortedPairs.sort(Comparator.comparingInt(entry -> {
        List<Point> pts = entry.getValue();
        return Point.manhattan(pts.get(0), pts.get(1));
    }));
    List<Character> keys = new ArrayList<>();
    for (Map.Entry<Character, List<Point>> entry : sortedPairs) {
        keys.add(entry.getKey());
    }

    return backtrack(0, keys, onStep, method);
}
```

Gambar 3.5 solveWith *method*
(Sumber: https://github.com/TKurr/Makalah_Stima)

Pada awal pencarian, program akan memanggil solver dan mencari endpoints dengan metode findEndpoints terlebih dahulu. Endpoints berupa map yang berisi urutan pasangan warna yang bergantung pada urutan kemunculan pertama di grid. Gambar 3.5 menunjukkan metode solveWith yang dipanggil di awal pencarian. Metode ini menyalin endpoints, yaitu pasangan titik-titik dengan huruf yang sama, ke dalam sortedPairs, yang mengurutkan berdasarkan jarak manhattan titik awal ke tujuan. Setelah proses penyortiran, hanya key (karakter warna) yang diambil ke dalam list keys, yang kemudian digunakan sebagai urutan pengerjaan dalam algoritma backtracking.



Gambar 3.6 contoh grid awal puzzle
(Sumber: https://github.com/TKurr/Makalah_Stima)

Gambar 3.6 adalah contoh dari tampilan grid puzzle sebelum pencarian dimulai. Urutan dimulai berdasarkan jarak manhattan dari pasangan warna. Hal ini berarti urutan pada gambar diatas adalah ungu, jingga, biru, hijau, kuning, dan terakhir merah.

Proses pencarian akhirnya dimulai secara rekursi melalui memanggil metode backtrack, memulai dari indeks 0, yaitu yang paling awal.

```java
private boolean backtrack(int idx,
                          List<Character> keys,
                          Consumer<char[][]> onStep,
                          String method) {
    if (idx == keys.size()) return true;

    char ch = keys.get(idx);
    List<Point> points = endpoints.get(ch);
    Point start = points.get(0);
    Point end = points.get(1);

    Set<String> triedPaths = new HashSet<>();
    char[][] originalGrid = copyGrid();

    while (true) {
        List<Point> path = switch (method.toLowerCase()) {
            case "dfs" -> findDFSPath(start, end, triedPaths);
            case "ucs" -> findUCSPath(start, end, triedPaths);
            case "gbfs" -> findGreedyPath(start, end, triedPaths);
            case "astar" -> findAStarPath(start, end, triedPaths);
            default -> null;
        };

        if (path == null) break;

        triedPaths.add(pathToString(path));
        for (Point p : path) {
            grid.getGrid()[p.getRow()][p.getCol()] = ch;
            onStep.accept(copyGrid());
            sleep(10);
        }

        if (backtrack(idx + 1, keys, onStep, method)) return true;

        grid.setGrid(copyGrid(originalGrid));
        onStep.accept(copyGrid());
        sleep(10);
    }

    return false;
}
```

Gambar 3.7 backtrack *method*
(Sumber: https://github.com/TKurr/Makalah_Stima)

Gambar 3.7 menunjukkan metode *backtrack*, yang merupakan inti dari strategi penyelesaian puzzle color connect menggunakan pendekatan rekursif. Fungsi ini bertanggung jawab untuk mencoba menyambungkan semua pasangan titik warna satu per satu dan menghindari kegagalan. Bila ada suatu jalur yang gagal, program akan mundur (*backtrack)* dan mencoba jalur alternatif lain.

Fungsi ini menerima 4 parameter, yaitu idx sebagai indeks dalam pasangan warna, keys sebagai daftar pasangan warna dalam bentuk karakter, onStep sebagai fungsi *callback* yang digunakan untuk menampilkan setiap langkah solusi, dan method yang menandakan algoritma penelusuran jalur yang digunakan.

Karena backtrack adalah proses rekursif, fungsi ini juga memiliki basis. Basis dari fungsi ini adalah ketika idx sudah mencapai jumlah pasangan warna yang ada berdasarkan ukuran list keys. Hal tersebut artinya program telah menyambungkan semua pasangan warna dan mencoba untuk

memanggil rekursi lagi, tetapi dianggap selesai karena idx telah mencapai basis.

Pada awal proses rekursi, sebuah karakter warna yang akan diproses diambil dari list keys berdasarkan idx. Setelah itu, program akan mendapatkan dua titik endpoint (awal dan akhir) untuk warna tersebut (ch) dari map endpoints yang telah dibentuk sebelumnya saat diinisialisasi kelas solver. Kedua titik dipisah ke dalam variabel start dan end, yang masing-masing berupa point. Untuk menyimpan representaasi string dari jalur-jalur yang telah dicoba untuk pasangan warna saat ini, program akan membuat set dari string bernama triedPaths. Tujuan dari penggunaan set ini adalah agar jalur yang sama tidak akan dicoba berulang-ulang. Sebelum mulai mencoba menyambungkan pasangan titik warna, sistem menyimpan salinan dari kondisi grid saat ini. Hal ini penting dalam mekanisme backtrack agar jika pencarian jalur gagal, grid dapat dipulihkan ke kondisi sebelumnya.

Pencarian jalur akan dilakukan secara berulang untuk pasangan titik ini hingga tidak ada lagi jalur yang memungkinkan. Karena program menyediakan beberapa algoritma, terdapat potongan program untuk memilih algoritma pencarian jalur sesuai input dari parameter method. Jalur akan dicari berdasarkan algoritma yang dipilih dan dimasukkan ke list dari point bernama path. Bila tidak ada jalur yang valid ditemukan, path akan bernilai null.

```
private String pathToString(List<Point> path) {
    StringBuilder sb = new StringBuilder();
    for (Point p : path) {
        sb.append(p.getRow())
        .append(',')
        .append(p.getCol())
        .append(';');
    }
    return sb.toString();
}
```

Gambar 3.8 pathToString *method*
(Sumber: https://github.com/TKurr/Makalah_Stima)

Setelah path diisi, akan dilakukan pengecekan bila ada jalur dari awal ke akhir yang dapat dilalui atau tidak. Bila tidak ada jalur lagi yang bisa dicoba untuk pasangan titik warna ini, maka program akan keluar dari loop ini dan menanggap pencarian untuk pasangan gagal. Akan tetapi, bila jalur ditemukan, path akan diubah menjadi bentuk string unik melalui metode pathToString pada gambar 3.8 dan dimasukkan ke dalam triedPaths, yang berupa set dari string, agar tidak digunakan lagi pada iterasi berikutnya. Hal ini bertujuan untuk menghindari pencarian redundan.

Setiap titik dalam jalur yang ditemukan diisi dengan huruf warna (ch) pada grid. Setelah setiap langkah perubahan, fungsi onStep akan dipanggil untuk memperbarui tampilan visual. Setelah pasangan warna saat ini berhasil disambungkan, fungsi memanggil dirinya (*backtrack)* secara rekursif untuk menyelesaikan pasangan warna berikutnya (idx + 1). Jika seluruh pasangan berikutnya berhasil tersambung, maka hasil boolean berupa true dikembalikan ke atas rekursi. Namun, jika langkah rekursif berikutnya gagal, maka sistem akan kembali ke kondisi sebelumnya dengan cara mengembalikan grid ke originalGrid. Tampilan juga akan diperbarui ulang untuk menampilkan hasil pengembalian kondisi grid. Loop dapat berakhir ketika semua kemungkinan jalur telah dicoba untuk pasangan warna titik ini dan semuanya gagal. Jika tidak ada jalur yang valid untuk pasangan warna saat ini, atau jalur tersebut menyebabkan pasangan warna berikutnya gagal, akan dikembalikan hasil boolean berupa false sebagai tanda bahwa solusi tidak ditemukan dari konfigurasi jalur ini.

Dalam proses *backtracking,* telah disinggung pencarian jalur menggunakan algoritma pathfinding sesuai input dari parameter method. Dalam hal ini, algoritma yang akan menjadi fokus utama adalah A* karena algoritma ini bekerja sangat efisien dan efektif.

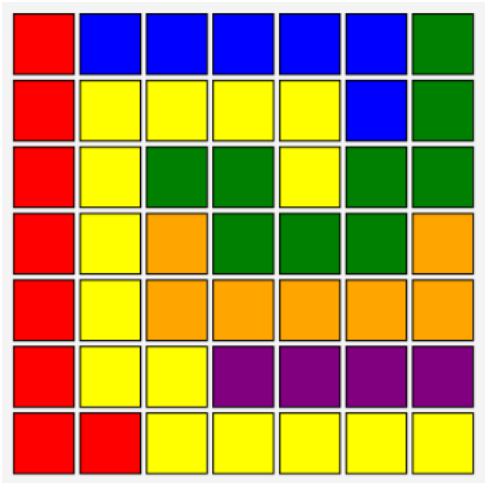
```
private List<Point> findAStarPath(Point start, Point end, Set<String> tried) {
    PriorityQueue<List<Point>> pq = new PriorityQueue<>(Comparator.comparingInt(
        path -> path.size() + Point.manhattan(path.get(path.size() - 1), end)
    ));
    Set<String> visited = new HashSet<>();

    pq.add(List.of(start));

    while (!pq.isEmpty()) {
        List<Point> path = pq.poll();
        Point curr = path.get(path.size() - 1);

        if (curr.equals(end)) {
            String sig = pathToString(path);
            if (!tried.contains(sig)) return path;
            else continue;
        }

        for (int[] d : DIRS) {
            int nr = curr.getRow() + d[0], nc = curr.getCol() + d[1];
            Point next = new Point(nr, nc);

            if (inBounds(nr, nc) &&
                (grid.getGrid()[nr][nc] == '.' || next.equals(end)) &&
                !path.contains(next)) {

                List<Point> newPath = new ArrayList<>(path);
                newPath.add(next);
                String sig = pathToString(newPath);

                if (!visited.contains(sig)) {
                    pq.add(newPath);
                    visited.add(sig);
                }
            }
        }
    }

    return null;
}
```

Gambar 3.9 findAStarPath method
(Sumber: https://github.com/TKurr/Makalah_Stima)

Gambar 3.9 menunjukkan fungsi dari algoritma A* untuk program ini. Fungsi ini memiliki parameter tried untuk menyimpan path yang telah dicoba agar tidak dicoba ulang saat *backtracking.* Algoritma dimulai dengan membuat priority queue yang menyimpan seluruh path dalam bentuk list dari point. Comparator digunakan untuk mengurutkan jalur

berdasarkan total biaya A*, yang merupakan jumlah dari g(n) atau panjang path dengan h(n) atau estimasi jarak manhattan ke tujuan. Program juga membuat seet dari string bernama visited, yang digunakan untuk mencatat jalur yang sudah pernah dimasukkan ke antrian agar tidak dimasukkan ulang. Sebelum memulai, priority queue akan diisi path awal, yang berupa titik awal.

Selama ada jalur yang belum diproses algoritma akan terus mencari dengan mengambil path dengan prioritas tertinggi. Jalur dengan biaya total terendah menjadi prioritas. Lalu, program akan mendapatkan titik terakhir dari path saat ini atau titik yang sedang dikunjungi pada saat penelusuran. Jika sudah sampai ke tujuan, jalur diperiksa bila sudah pernah dicoba atau belum. Bila belum, fungsi akan mengembalikan path, sedangkan jika sudah, jalur diabaikan dan fungsi dilanjutkan. Bila lanjut, fungsi akan melakukan iterasi ke seluruh arah berdasarkan DIRS yang sudah terdefinisi di kelas solver. Koordinat tetangga akan dihitung berdasarkan arah. Koordinat juga akan diperiksa bila masih dalam batas grid, kosong, atau tujuan, dan belum dikunjungi dalam path. Jika memenuhi batasan, program akan membuat jalur baru dari path saat ini dengan menambahkan titik tetangga. Jalur akan dikonversi ke string. Jika belum pernah dikunjungi, jalur dimasukkan ke priority queue dan tandai sebagai visited. Pada akhir fungsi, bila jalur tidak ditemukan, fungsi akan mengembalikan nilai null yang menandakan jalur tidak ditemukan.

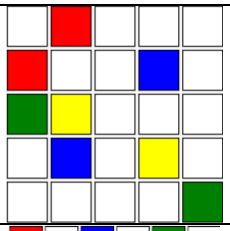
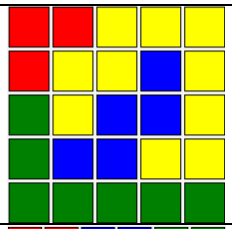
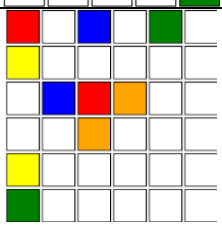
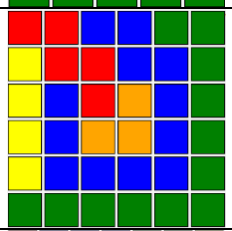
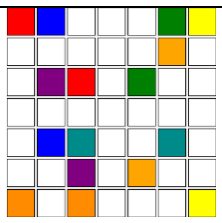
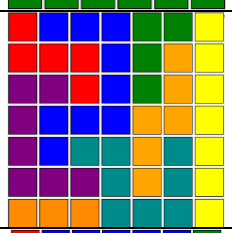
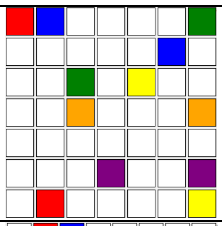
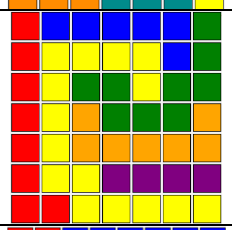
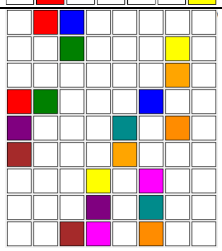
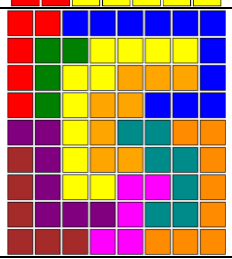


Gambar 3.10 contoh grid akhir puzzle
(Sumber: https://github.com/TKurr/Makalah_Stima)

Secara keseluruhan, fungsi A* dan backtrack bekerja secara berulang dengan pola yang konsisten. Untuk setiap pasangan titik warna, backtrack memanggil A* untuk mencari satu jalur terbaik yang belum pernah dicoba sebelumnya. Jika jalur tersebut berhasil, warna akan diisi di grid dan backtrack lanjut ke pasangan berikutnya. Namun jika pada pasangan selanjutnya tidak ditemukan jalur, backtrack akan membatalkan jalur sebelumnya dan memanggil A* lagi untuk mencoba jalur alternatif lain. Proses ini terus berulang dengan A* mencari jalur, backtrack menguji kelanjutan solusi hingga seluruh pasangan warna berhasil terhubung seperti pada gambar 3.10 atau semua kemungkinan habis dicoba.

V. EKSPERIMEN

Pada bagian ini, strategi penyelesaian color connect dengan A* dan *backtracking* akan diuji. Algoritma diuji pada kumpulan file puzzle dengan kompleksitas dan ukuran yang bervariasi. Pengujian dilakukan dengan mengecek bila puzzle dapat diselesaikan dengan algoritma yang telah dibahas, sekaligus mencatat waktu eksekusi yang dibutuhkan untuk menyelesaikan setiap puzzle. Eksperimen dijalankan dalam lingkungan komputasi yang sama untuk memastikan keadilan dalam perbandingan performa. Hasil eksperimen digunakan untuk membuktikan efektivitas algoritma dalam keberhasilan algoritma dan aspek waktu.

No.	Test Case Awal	Test Case Akhir	Waktu (ms)
1			2068
2			1253
3			8123
4			19254
5			52484

VI. Diskusi dan Pembahasan

Implementasi algoritma A* dalam penyelesaian puzzle Color Connect memberikan pendekatan yang efisien untuk menemukan jalur terpendek antara pasangan titik berwarna, dengan mempertimbangkan estimasi biaya sisa melalui heuristik jarak Manhattan. Dalam pengembangannya, algoritma A* tidak berdiri sendiri, melainkan dikombinasikan dengan strategi backtracking untuk memastikan bahwa setiap pasangan warna dapat terhubung tanpa saling bertabrakan. Strategi ini memungkinkan sistem untuk menghapus jalur yang telah dibuat apabila pasangan warna berikutnya tidak dapat disambungkan, dan kemudian mencoba alternatif jalur lain yang mungkin. Hal ini menjadikan proses pencarian solusi lebih fleksibel dan menyeluruh, meskipun berdampak pada waktu komputasi yang meningkat ketika ruang solusi sangat besar atau kompleksitas jalurnya tinggi. Kombinasi A* dan backtracking terbukti efektif dalam menemukan solusi yang valid secara keseluruhan pada puzzle berukuran sedang hingga besar. Meskipun demikian, efisiensi algoritma tetap sangat dipengaruhi oleh urutan pasangan warna dan bentuk grid yang diberikan. Oleh karena itu, dalam implementasi praktis, pemilihan urutan penyambungan titik serta pemanfaatan heuristik yang lebih adaptif dapat menjadi arah pengembangan lebih lanjut.

VII. Kesimpulan

Dalam makalah ini, telah dibahas strategi penyelesaian puzzle Color Connect dengan menggabungkan algoritma A* dan metode *backtracking* secara terpadu. Algoritma A* digunakan untuk mencari jalur optimal antara pasangan titik dengan mempertimbangkan heuristik jarak manhattan, sementara *backtracking* memastikan bahwa jika sebuah solusi tidak berhasil menghubungkan seluruh pasangan warna, maka sistem akan kembali (rollback) dan mencoba jalur alternatif. Pendekatan ini memungkinkan penyelesaian puzzle secara menyeluruh dan fleksibel, karena mampu menelusuri semua kemungkinan rute yang valid dengan pertimbangan urutan rute yang efisien.

Lampiran

1. Github Repository : TKurr/Makalah_Stima
   Test Case berada pada folder /src/main/resources
2. Link Video : https://youtu.be/zX6DLZeM9Fc

Ucapan Terima Kasih

Penulis ingin menyampaikan rasa terima kasih kepada dosen pengampu mata kuliah Strategi Algoritma Kelas 3, Bapak Monterico Adrian dari Institut Teknologi Bandung, atas penyampaian materi yang jelas dan terstruktur. Penjelasan yang mendalam dan metode pengajaran yang interaktif sangat membantu penulis dalam memahami konsep-konsep yang diajarkan dengan baik.

Selain itu, penulis juga ingin menyampaikan rasa terima kasih kepada Bapak Rinaldi Munir atas penugasan pembuatan makalah ini. Penugasan tersebut memberikan kesempatan berharga bagi penulis untuk mendalami aplikasi praktis dari materi yang telah dipelajari di kelas. Melalui tugas ini, penulis tidak hanya memahami teori secara konseptual, tetapi juga mampu menerapkannya dalam konteks yang relevan dengan permasalahan nyata.

References

[1] [1] R. Munir, "Route planning dengan algoritma pencarian jalur optimal (Bagian 2)," *Kuliah Strategi Algoritma, Sekolah Teknik Elektro dan Informatika ITB*, 2025. [Online]. Tersedia: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf

[2] [2] R. Munir, "Algoritma backtracking (Bagian 1)," *Kuliah Strategi Algoritma, Sekolah Teknik Elektro dan Informatika ITB*, 2025. [Online]. Tersedia: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/15-Algoritma-backtracking-(2025)-Bagian1.pdf

[3] [3] R. Munir, "Algoritma backtracking (Bagian 2)," *Kuliah Strategi Algoritma, Sekolah Teknik Elektro dan Informatika ITB*, 2025. [Online]. Tersedia: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/16-Algoritma-backtracking-(2025)-Bagian2.pdf

[4] [4] FortisLabs, *Connect Colors - Puzzle Game*, versi aplikasi Android, Google Play Store. [Online]. Tersedia: https://play.google.com/store/apps/details?id=com.fortislabs.connectcolors

[5] [5] Paul Pu Liang, "A* algorithm explained with practical example," *Paul.pub*, [Online]. Tersedia: https://paul.pub/a-star-algorithm/

[6] [6] GeeksforGeeks, "Backtracking - Data Structures and Algorithms," [Online]. Tersedia: https://www.geeksforgeeks.org/dsa/backtracking-algorithms/

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

Theo Kurniady, 13523154