Penerapan Algoritma A* dalam Penyelesaian Sliding Puzzle

Samy Muhammad Haikal - 13522151
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
13522151@std.stei.itb.ac.id

***Abstract*—Sliding puzzle merupakan permainan teka-teki yang melibatkan pemindahan ubin dalam grid persegi untuk mencapai susunan ubin yang diinginkan. Permainan ini terkenal karena tantangan logikanya yang memerlukan strategi penyelesaian yang efektif. Algoritma A* adalah salah satu algoritma pencarian rute pada graf. Algoritma ini menelusuri suatu graf dengan memperkirakan biaya heuristic pada setiap simpul. Makalah ini mencoba untuk membuat suatu solver untuk game ini menggunakan algoritma A*.**

Keywords—sliding puzzle, algoritma a*, route planning

I. PENDAHULUAN

Sliding puzzle adalah permainan teka-teki yang terdiri dari sebuah grid persegi dengan ubin-ubin bernomor yang bisa digeser, dengan tujuan menyusun kembali ubin-ubin tersebut ke dalam urutan tertentu. Permainan ini, yang pertama kali diperkenalkan pada abad ke-19, telah menjadi alat yang populer tidak hanya untuk hiburan tetapi juga untuk penelitian dalam bidang kecerdasan buatan dan ilmu komputer. Sliding puzzle sering kali digunakan untuk menguji dan mengembangkan algoritma pencarian jalur karena tantangan yang ditawarkan dalam menemukan solusi optimal dari posisi acak menuju konfigurasi yang diinginkan.

1	2	4
7	8	5
3	6	

Gambar 1 *Contoh dari Sliding Puzzle*
Sumber: dokumentasi pribadi

Sliding puzzle pertama kali diperkenalkan pada abad ke-19 dan telah mengalami berbagai variasi sejak saat itu. Salah satu varian yang paling terkenal adalah "15 Puzzle" yang ditemukan oleh Noyes Chapman pada tahun 1880. Permainan ini terdiri dari grid 4x4 dengan 15 ubin bernomor dan satu ruang kosong. Popularitas "15 Puzzle" meningkat pesat dan menjadi fenomena global, menarik perhatian para pemecah teka-teki dan matematikawan karena tantangan logikanya yang menarik.

Permainan ini tidak hanya menghibur tetapi juga mendidik, memberikan tantangan mental yang dapat mengasah keterampilan kognitif pemain. Sebagai alat pengajaran, sliding puzzle menyediakan contoh praktis dari konsep-konsep algoritmik yang digunakan dalam ilmu komputer dan kecerdasan buatan.

Berbagai metode dan strategi telah dikembangkan untuk menyelesaikan Sliding Puzzle. Salah satu pendekatan yang menarik adalah memanfaatkan Algoritma A*. Teori graf merupakan bagian penting dalam matematika diskrit yang mempelajari interaksi antara objek yang disebut simpul, yang terhubung melalui sisi atau tepi.

Dalam konteks Sliding Puzzle, penerapan Algoritma A* melibatkan pemetaan state yang dapat dihasilkan pada setiap Langkah menjadi simpul dalam graf . Pendekatan ini memungkinkan penggunaan algoritma A* untuk menganalisis dan menyelesaikan teka-teki Sliding Puzzle secara lebih sistematis dan efisien.

II. DASAR TEORI

A. Sliding Puzzle

Sliding puzzle adalah permainan teka-teki yang terdiri dari sebuah grid persegi dengan ubin-ubin bernomor yang bisa digeser. Grid ini memiliki satu ruang kosong yang memungkinkan ubin-ubin tersebut untuk digeser ke posisi yang diinginkan. Tujuan permainan ini adalah untuk menyusun kembali ubin-ubin tersebut ke dalam urutan tertentu, biasanya urutan numerik, dengan menggunakan ruang kosong yang tersedia untuk menggeser ubin.

Sliding puzzle biasanya hadir dalam bentuk grid persegi, eperti 3x3, 4x4, atau ukuran yang lebih besar. Grid terdiri dari ubin-ubin yang bernomor dan satu ruang kosong yang digunakan untuk memindahkan ubin-ubin tersebut. Setiap ubin memiliki nomor unik yang dicetak di atasnya. Ubin dapat digeser ke ruang kosong yang bersebelahan (baik secara horizontal maupun vertikal), memungkinkan pemain untuk memindahkan ubin ke posisi yang diinginkan.

Tujuan dari permainan adalah menyusun kembali ubin-ubin ke dalam urutan yang benar (biasanya dalam urutan numerik dari kiri ke kanan dan dari atas ke bawah) dengan memanfaatkan ruang kosong.

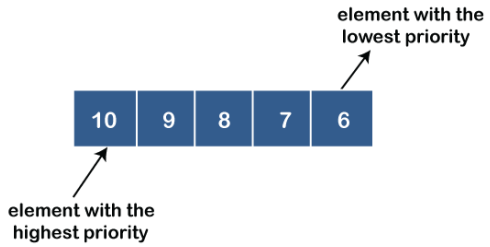
Permainan dimulai dengan ubin-ubin dalam susunan acak. Ruang kosong dapat berada di posisi mana saja di grid. Pemain dapat menggeser ubin yang bersebelahan dengan ruang kosong ke ruang kosong tersebut. Hanya satu ubin yang dapat digeser setiap kali langkah dilakukan.

Pemain harus merencanakan langkah-langkah mereka dengan hati-hati untuk menyusun ubin dalam urutan yang benar. Permainan ini menuntut kemampuan pemecahan masalah dan perencanaan ke depan. Permainan dianggap selesai ketika semua ubin berada dalam urutan yang benar, dengan ruang kosong di sudut kanan bawah (biasanya).

B. Priority Queue

Priority queue adalah struktur data abstrak yang mirip dengan queue (antrian), tetapi dengan prioritas elemen. Dalam priority queue, setiap elemen memiliki prioritas yang menentukan urutan penghapusannya. Elemen dengan prioritas tertinggi dikeluarkan terlebih dahulu. Ada beberapa istilah penting yang perlu diketahui saat bekerja dengan struktur data priority queue. Yang pertama adalah head dan tail, head adalah elemen terdepan dari sebuah queue dan tail adalah sisa dari queue tanpa head. Lalu ada juga enqueue dan dequeue, enqueue adalah proses memasukkan elemen baru ke sebuah priority queue, elemen ini nantinya akan diurutkan berdasarkan nilai prioritasnya dalam queue. Dequeue adalah proses untuk mengambil head dari sebuah queue.

Priority queue adalah struktur data yang penting dalam ilmu komputer, memungkinkan pengelolaan elemen dengan prioritas yang efisien. Implementasi yang paling umum menggunakan heap, yang menawarkan operasi insert dan delete yang efisien. Priority queue digunakan secara luas dalam algoritma graf, sistem operasi, simulasi, dan berbagai aplikasi lainnya di mana prioritas elemen perlu dikelola secara efisien.



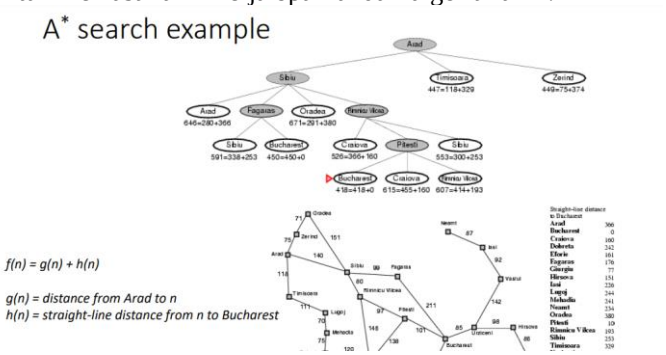
Gambar 2.1 *Contoh Priority Queue*
Sumber: [3]

*C. Algoritma A**

Algoritma A* adalah salah satu algoritma pencarian jalur yang popular dan efektif digunakan dalam kecerdasan buatan dan ilmu komputer. Algoritma ini digunakan untuk menemukan jalur terpendek antara dua titik, menggabungkan konsep dari algoritma Dijkstra dan Breadth-First Search (BFS). A* menggunakan fungsi heuristic untuk memperkirakan biaya tersisa yang diperlukan untuk mencapai tujuan, sehingga bisa mempercepat proses pencarian jalur optimal.

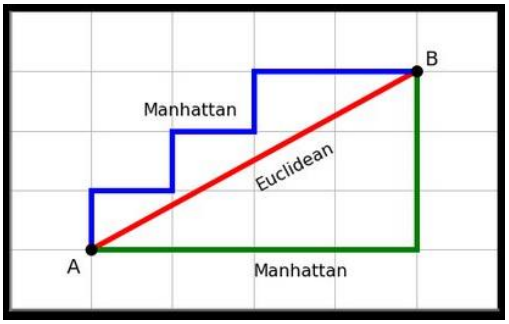
Dalam algoritma A*, setiap node memiliki tiga komponen biaya: g(n), h(n), dan f(n). Komponen g(n) merepresentasikan biaya dari node awal ke node saat ini, sedangkan h(n) adalah fungsi heuristik yang menentukan estimasi biaya dari node saat ini ke node tujuan. Total biaya f(n) adalah penjumlahan dari g(n) dan h(n). Node dengan nilai f(n) terendah akan dieksplorasi terlebih dahulu, memungkinkan algoritma untuk secara efisien mencari jalur terpendek dengan menggabungkan biaya yang sudah diketahui dan perkiraan biaya yang tersisa.

Kelebihan Algoritma A* adalah kemampuannya untuk menemukan solusi optimal dengan lebih efisien dibandingkan jika kita mencari dengan blind search seperti BFS atau DFS. Ini disebabkan karena kita menggunakan heuristik untuk membantu pencarian, jadi simpul yang diekspan terlebih dahulu adalah simpul yang paling dekat dengan tujuan. Namun, algoritma A* memiliki beberapa kekurangan. Salah satu tantangan utama adalah konsumsi memori yang tinggi, terutama dalam graf besar atau masalah dengan banyak kemungkinan jalur. Selain itu, efisiensi A* sangat bergantung pada kualitas heuristik yang digunakan. Heuristik yang buruk dapat membuat A* kurang efisien atau bahkan tidak efektif. Oleh karena itu, pemilihan heuristik yang tepat sangat penting untuk memastikan kinerja optimal dari algoritma ini.



Gambar 2.2 *Contoh Pencarian dengan algoritma A**
Sumber: [2]

D. Manhattan Distance



Gambar 2.3 *Ilustrasi Manhattan distance*
Sumber: https://www.omnicalculator.com/math/manhattan-distance

Manhattan distance adalah suatu metode untuk mengukur jarak antara dua titik dalam sebuah grid atau ruang yang terstruktur seperti kota atau peta, di mana pergerakan hanya diizinkan dalam arah horizontal dan vertikal. Konsep ini dinamai berdasarkan tata letak jalan-jalan di Manhattan, New York, yang berbentuk grid dengan jalan-jalan yang berjalan

secara ortogonal (tegak lurus). Oleh karena itu, Manhattan distance juga sering disebut sebagai "taxicab distance" atau "L1 distance." Secara umum, untuk menghitung Manhattan distance antara dua titik (x1, y1) dan (x2, y2) kita menggunakan rumus:

Manhattan Distance = |x1-x2| + |y1-y2|

Rumus ini menjumlahkan perbedaan absolut antara koordinat x dan y dari kedua titik tersebut. Hasilnya adalah jarak total yang harus ditempuh jika seseorang hanya dapat bergerak secara horizontal dan vertikal.

Manhattan distance sangat berguna dalam berbagai aplikasi, terutama yang melibatkan navigasi di dalam ruang yang terstruktur seperti grid. Contoh penerapan Manhattan distance termasuk dalam algoritma pencarian jalur seperti A* (A-star) untuk menemukan jalur terpendek di peta, dalam permainan papan seperti sliding puzzle, dan dalam analisis kluster di data mining di mana objek-objek sering diwakili dalam bentuk grid atau matriks.

Manhattan distance memiliki karakteristik unik dibandingkan dengan Euclidean distance (jarak Euclidean), yang mengukur jarak terpendek langsung antara dua titik menggunakan garis lurus. Dalam konteks tertentu, Manhattan distance lebih relevan dan berguna karena memperhitungkan keterbatasan pergerakan. Misalnya, dalam navigasi perkotaan, kendaraan biasanya harus mengikuti jalan-jalan yang membatasi gerakan mereka ke arah horizontal dan vertikal, sehingga Manhattan distance lebih menggambarkan jarak yang sebenarnya ditempuh dibandingkan dengan Euclidean distance.

Selain itu, Manhattan distance sering digunakan dalam kecerdasan buatan dan pembelajaran mesin sebagai metrik jarak dalam algoritma clustering, seperti K-means, di mana data sering direpresentasikan dalam bentuk grid atau matriks. Dalam kasus ini, Manhattan distance membantu dalam menentukan kesamaan antara data dan mengelompokkan data yang serupa berdasarkan jarak grid.

Secara keseluruhan, Manhattan distance adalah alat yang sangat berguna dan serbaguna dalam berbagai bidang, mulai dari perencanaan rute, permainan, hingga analisis data, memberikan cara yang intuitif dan efektif untuk mengukur jarak dalam lingkungan yang terstruktur seperti grid.

III. ANALISIS

A. *Mapping Persoalan ke A**

Pencarian menggunakan algoritma A* dapat direpresentasikan menjadi sebuah graf pohon. Hal pertama yang perlu dilakukan adalah menentukan tiap simpul merepresentasikan apa. Pada kasus ini tiap simpul pada graf merepresentasikan state papan permainan saat ini. Semua simpul tetangga dari sebuah simpul merepresentasikan Langkah-langkah yang bisa diambil dari state saat ini.

Fungsi heuristik yang diambil adalah Manhattan distance pada setiap ubin di papan saat ini. Sebagai contoh papan di bawah ini jika dicari h(n)-nya akan bernilai 3.

1	4	2
6	5	7
3	8	

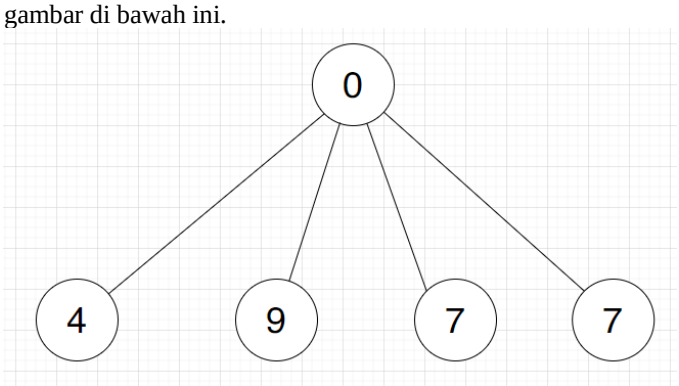
Gambar 3.1 *Contoh kasus papan permainan*

Sumber: dokumentasi pribadi

Untuk setiap ubin yang tidak berada di posisi yang benar, kita hitung Manhattan distance-nya:

- Ubin 1 berada di posisi (0, 0) dan tujuan di (0, 0) => |0-0| + |0-0| = 0
- Ubin 2 berada di posisi (0, 2) dan tujuan di (0, 1) => |0-1| + |0-1| = 1
- Ubin 3 berada di posisi (2, 0) dan tujuan di (0, 2) => |2-0| + |0-2| = 4
- Ubin 4 berada di posisi (0, 1) dan tujuan di (1, 0) => |0-1| + |1-0| = 2
- Ubin 5 berada di posisi (1, 1) dan tujuan di (1, 1)=> |1-1| + |1-1| = 0
- Ubin 6 berada di posisi (1, 0) dan tujuan di (1, 2)=> |1-1| + |0-2| = 2
- Ubin 7 berada di posisi (1, 2) dan tujuan di (2, 0)=> |1-2| + |2-0| = 3
- Ubin 8 berada di posisi (2, 1) dan tujuan di (2, 1)=> |2-1| + |2-1| = 0

Total Manhattan distance: 0+1+4+2+0+2+3+0 = 12.
Jika dipetkan menjadi sebuah graf kira-kira akan jadi seperti gambar di bawah ini.



Gambar 3.1 *Representasi Papan pada graf*
Sumber: dokumentasi pribadi

Setiap simpul merepresentasikan papan permainan pada state berbeda, dan angka di dalam simpul adalah nilai dari f(n), yaitu nilai g(n) + h(n).

B. *Implementasi*

Berikut adalah implementasi berbagai elemen dalam Bahasa java.

A. Implementasi Papan Permainan

```
static class BoardNode {
    int[][] board;
    String boardString;
    int cost;
```

```
    public BoardNode(int[][] board, int cost) {
        this.board = board;
        this.boardString = new SlidingPuzzle(board).boardToString(board);
        this.cost = cost;
    }
}
```

B. Implementasi Manhattan Distance

```
private int getManhattanDistance(int[][] board, int[][] goal) {
int manhattanDistance = 0;
int n = board.length;

for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        int value = board[i][j];
        if (value != 0) {
            // Find the goal position of the current tile value
            for (int x = 0; x < n; x++) {
                for (int y = 0; y < n; y++) {
                    if (goal[x][y] == value) {
                        manhattanDistance += Math.abs(i - x) + Math.abs(j - y);
                        break;
                    }
                }
            }
        }
    }
}
return manhattanDistance;
}
```

C. Implementasi Pencarian A*

```
public class SlidingPuzzle {
  private int[][] goalBoard;
  private static final int[][] DIRECTIONS = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

  public SlidingPuzzle(int[][] goalBoard) {
     this.goalBoard = goalBoard;
  }

  public List<int[][]> findShortestPath(int[][] startBoard) {
     PriorityQueue<BoardNode> frontier = new PriorityQueue<>(Comparator.comparingInt(node -> node.cost));
     Map<String, String> cameFrom = new HashMap<>();
     Map<String, Integer> costSoFar = new HashMap<>();

     String start = boardToString(startBoard);
     String goal = boardToString(goalBoard);

     frontier.add(new BoardNode(startBoard, 0));
     cameFrom.put(start, null);
     costSoFar.put(start, 0);

     int nodeCount = 0;
     while (!frontier.isEmpty()) {
        BoardNode current = frontier.poll();

        if (current.boardString.equals(goal)) {
           System.out.println(" ");
           System.out.println("Node visited\t: " + nodeCount);
           return reconstructPath(cameFrom, current.boardString);
        }

        for (int[][] nextBoard : getNeighbors(current.board)) {
           String next = boardToString(nextBoard);
           int newCost = costSoFar.get(current.boardString) + 1 + getManhattanDistance(nextBoard, goalBoard);
           if (!costSoFar.containsKey(next)) {
              costSoFar.put(next, newCost);
              frontier.add(new BoardNode(nextBoard, newCost));
              cameFrom.put(next, current.boardString);
           }
           nodeCount++;
        }
     }
     return null; // No path found
  }

  private List<int[][]> reconstructPath(Map<String, String> cameFrom, String current) {
     List<int[][]> path = new ArrayList<>();
     while (current != null) {
        path.add(0, stringToBoard(current));
        current = cameFrom.get(current);
     }
     return path;
  }

  private List<int[][]> getNeighbors(int[][] board) {
     List<int[][]> neighbors = new ArrayList<>();
     int n = board.length;
     int emptyRow = -1, emptyCol = -1;

     // Find empty cell
     for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
           if (board[i][j] == 0) {
              emptyRow = i;
              emptyCol = j;
           }
```

```
        }
    }

    // Move empty cell in all 4 possible directions
    for (int[] dir : DIRECTIONS) {
        int newRow = emptyRow + dir[0];
        int newCol = emptyCol + dir[1];

        if (newRow >= 0 && newRow < n && newCol >= 0 && newCol < n) {
            int[][] newBoard = copyBoard(board);
            newBoard[emptyRow][emptyCol] = newBoard[newRow][newCol];
            newBoard[newRow][newCol] = 0;
            neighbors.add(newBoard);
        }
    }
    return neighbors;
  }



  private int[][] copyBoard(int[][] board) {
    int n = board.length;
    int[][] newBoard = new int[n][n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            newBoard[i][j] = board[i][j];
        }
    }
    return newBoard;
  }

  private String boardToString(int[][] board) {
    StringBuilder sb = new StringBuilder();
    int n = board.length;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            sb.append(board[i][j]).append(',');
        }
    }
    return sb.toString();
  }
  private int[][] stringToBoard(String boardString) {
    String[] values = boardString.split(",");
    int n = (int) Math.sqrt(values.length);
    int[][] board = new int[n][n];
    int index = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            board[i][j] = Integer.parseInt(values[index++]);
        }
    }
    return board;
  }
}
```

Implementasi di atas akan menghasilkan solusi dari sliding puzzle yang diberikan. Misalnya kita memberikan board awal seperti berikut.

```
{2, 4, 3},
{1, 7, 5},
{6, 0, 8}
```

Maka kode akan mencari rute terpendek untuk mencapai solusi.

```
Node visited    : 3550
Path found:
Steps: 20
2 4 3
1 7 5
6 0 8

1 2 3
4 5 6
7 0 8

1 2 3
4 5 6
7 8 0
```

Gambar 3.2 *Hasil Testing program*
Sumber: dokumentasi pribadi

IV. KESIMPULAN

Dari percobaan yang telah dilakukan dapat dilihat bahwa persoalan sliding puzzle dapat dimodelkan menjadi sebuah graf. Lalu, untuk mendapatkan solusinya kita bisa menjalankan algoritma A*. Algoritma A* telah terbukti menjadi metode yang sangat efektif dan efisien untuk menyelesaikan sliding puzzle. Dengan menggunakan heuristik Manhattan distance, algoritma A* dapat menentukan jalur terpendek dari konfigurasi awal ke konfigurasi tujuan. Manhattan distance memberikan estimasi yang admissible dan konsisten, sehingga menjamin bahwa algoritma A* akan menemukan solusi optimal jika ada.

Keberhasilan algoritma A* dalam menyelesaikan sliding puzzle menunjukkan potensi besar dari pendekatan ini untuk diterapkan pada masalah pencarian jalur lainnya yang serupa. Keandalannya dalam menemukan solusi optimal dengan efisiensi tinggi membuatnya menjadi alat yang sangat berharga dalam bidang kecerdasan buatan, perencanaan rute, dan berbagai aplikasi lainnya yang memerlukan pencarian jalur dalam ruang yang terstruktur.

UCAPAN TERIMA KASIH

Dengan penuh rasa syukur, penulis ingin mengucapkan puji dan syukur kepada Tuhan Yang Maha Esa atas segala rahmat, karunia, serta taufik dan hidayah-Nya, yang telah memandu dan memberikan keberkahan sehingga penulis berhasil menyelesaikan makalah berjudul " Penerapan Algoritma A* dalam Penyelesaian Sliding Puzzle". Makalah ini disusun

sebagai bagian dari tugas mata kuliah Strategi Algoritma IF2221.

Penulis juga mengucapkan terima kasih yang setinggi-tingginya kepada Bapak Ir. Rila Mandala, M.Eng., Ph.D. dan Bapak Monterico Adrian, S.T., M.T., M.T. dan Bapak Monterico Adrian, S.T., M.T., sebagai dosen pengajar dalam mata kuliah Matematika Diskrit IF2120 Kelas K3. Terima kasih atas dedikasi, bimbingan, dan pengajaran yang telah diberikan selama satu semester ini, memberikan kontribusi besar dalam pembentukan pemahaman kami sebagai mahasiswa.

REFERENCES

[1] *R. Munir, "Penentuan Rute (Route/Path Planning) (Bag.1)." Diakses dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/RoutePlanning-Bagian1-2021.pdf, pada 12 Juni 2024.*

[2] R. Munir, "Penentuan Rute (Route/Path Planning) (Bag.2)." Diakses dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/RoutePlanning-Bagian2-2021.pdf, pada 12 Juni 2024.

[3] *Priority queue (data structures) - javatpoint*. Priority queue (data structures). (n.d.). https://www.javatpoint.com/ds-priority-queue, pada 12 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

Samy Muhammad Haikal 13522151