

IF2211 Strategi Algoritma

Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding

Laporan Tugas Kecil 3

Disusun untuk memenuhi tugas mata kuliah Strategi Algoritma pada Semester 4 (empat)

Tahun Akademik 2024/2025



Disusun Oleh:

Brian Ricardo Tamin (13523126)

Theo Kurniady (13523154)

**PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG**

BANDUNG 2025

Daftar Isi

Daftar Isi.....	2
BAB I	
DESKRIPSI MASALAH.....	3
BAB II	
Dasar Teori.....	7
2.1 Uniform Cost Search.....	7
2.2 Best First Search.....	8
2.3 A* Search.....	9
BAB III	
IMPLEMENTASI PROGRAM.....	10
3.1 Struktur Folder dan File.....	10
3.2 Model.....	11
3.3 Algoritma.....	18
3.4 Bonus.....	21
3.4.1 Graphical User Interface.....	21
3.4.2 Heuristics.....	28
BAB IV	
EKSPERIMEN.....	30
4.1 Uniform Cost Search.....	30
4.2 Best First Search.....	33
4.3 A* Search.....	39
BAB V	
ANALISIS ALGORITMA.....	48
5.1 Definisi f(n) dan g(n).....	48
5.2 Admissibility Heuristik A* pada Puzzle Rush Hour.....	48
5.3 Kesamaan UCS dan BFS pada Penyelesaian Puzzle Rush Hour.....	48
5.4 Keefisienan A* dibandingkan UCS dalam Rush Hour.....	48
5.5 Kekurang-optimalan GBFS dalam pencarian solusi pada Rush Hour.....	49
5.6 Cara Kerja Algoritma Pathfinding pada Rush Hour.....	50
5.7 Kompleksitas Algoritma.....	53
5.7.1 Uniform Cost Search (UCS).....	53
5.7.2 Greedy Best First Search (GBFS).....	54
5.7.3 A*	55
BAB VI	
PENUTUP.....	56
6.1 Kesimpulan.....	56
6.2 Saran.....	56
LAMPIRAN.....	57
DAFTAR PUSTAKA.....	58

BAB I

DESKRIPSI MASALAH



Gambar 1. Rush Hour Puzzle

(Sumber: <https://www.thinkfun.com/en-US/products/educational-games/rush-hour-76582>)

Rush Hour adalah sebuah permainan puzzle logika berbasis grid yang menantang pemain untuk menggeser kendaraan di dalam sebuah kotak (biasanya berukuran 6x6) agar mobil utama (biasanya berwarna merah) dapat keluar dari kemacetan melalui pintu keluar di sisi papan. Setiap kendaraan hanya bisa bergerak lurus ke depan atau ke belakang sesuai dengan orientasinya (horizontal atau vertikal), dan tidak dapat berputar. Tujuan utama dari permainan ini adalah memindahkan mobil merah ke pintu keluar dengan jumlah langkah seminimal mungkin.

Komponen penting dari permainan Rush Hour terdiri dari:

1. **Papan** – Papan merupakan tempat permainan dimainkan.

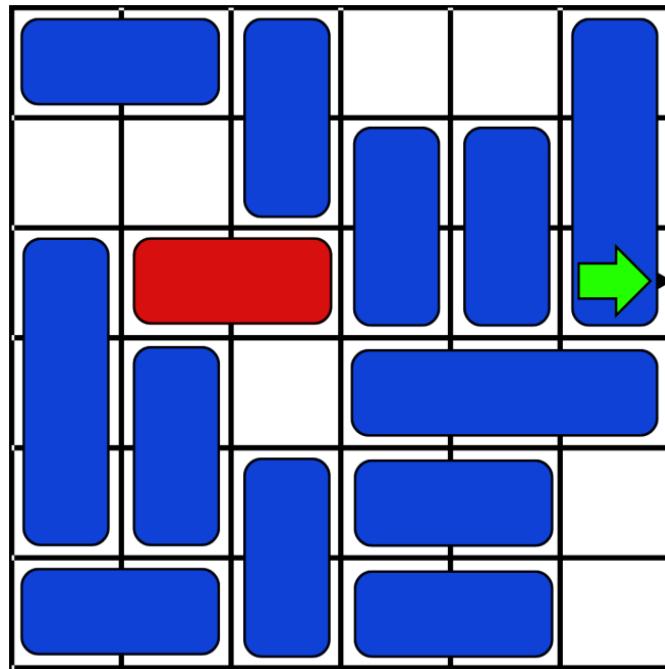
Papan terdiri atas cell, yaitu sebuah singular point dari papan. Sebuah piece akan menempati cell-cell pada papan. Ketika permainan dimulai, semua piece telah diletakkan di dalam papan dengan konfigurasi tertentu berupa lokasi piece dan orientasi, antara horizontal atau vertikal.

Hanya primary piece yang dapat digerakkan keluar papan melewati pintu keluar. Piece yang bukan primary piece tidak dapat digerakkan keluar papan. Papan memiliki satu pintu keluar yang pasti berada di dinding papan dan sejajar dengan orientasi primary piece.

2. **Piece** – Piece adalah sebuah kendaraan di dalam papan. Setiap piece memiliki posisi, ukuran, dan orientasi. Orientasi sebuah piece hanya dapat berupa horizontal atau vertikal—tidak mungkin diagonal. Piece dapat memiliki beragam ukuran, yaitu jumlah cell yang ditempati oleh piece. Secara standar, variasi ukuran sebuah piece adalah 2-piece (menempati 2 cell) atau 3-piece (menempati 3 cell). Suatu piece tidak dapat digerakkan melewati/menembus piece yang lain.
3. **Primary Piece** – Primary piece adalah kendaraan utama yang harus dikeluarkan dari papan (biasanya berwarna merah). Hanya boleh terdapat satu primary piece.
4. **Pintu Keluar** – Pintu keluar adalah tempat primary piece dapat digerakkan keluar untuk menyelesaikan permainan
5. **Gerakan** — Gerakan yang dimaksudkan adalah pergeseran piece di dalam permainan. Piece hanya dapat bergerak/bergeser lurus sesuai orientasinya (atas-bawah jika vertikal dan kiri-kanan jika horizontal). Suatu piece tidak dapat digerakkan melewati/menembus piece yang lain.

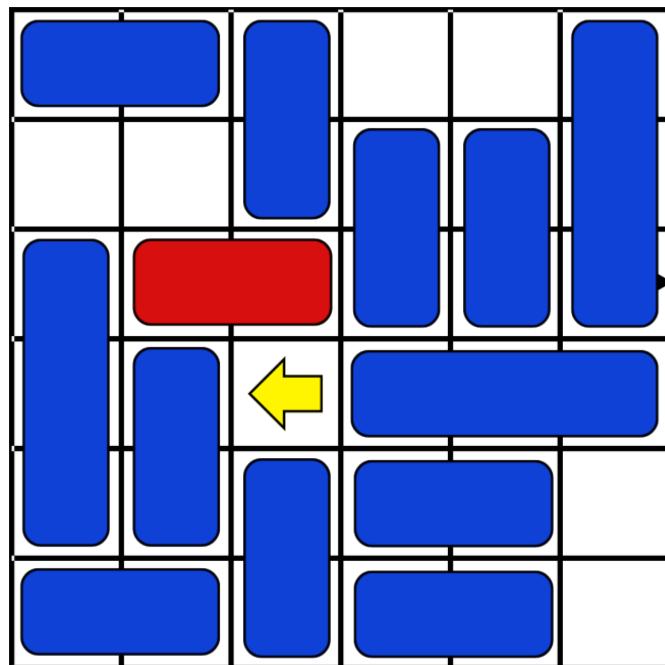
Ilustrasi kasus :

Diberikan sebuah papan berukuran 6 x 6 dengan 12 piece kendaraan dengan 1 piece merupakan primary piece. Piece ditempatkan pada papan dengan posisi dan orientasi sebagai berikut.

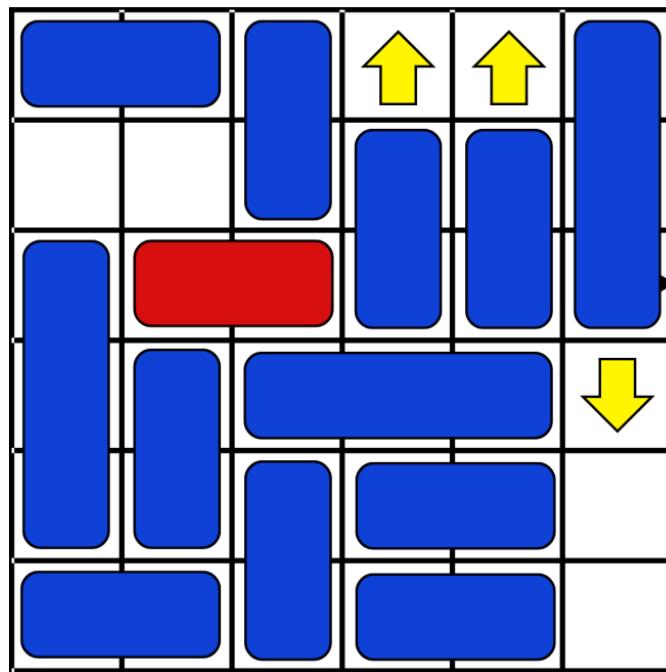


Gambar 2. Awal Permainan Game Rush Hour

Pemain dapat menggeser-geser piece (termasuk primary piece) untuk membentuk jalan lurus antara primary piece dan pintu keluar.

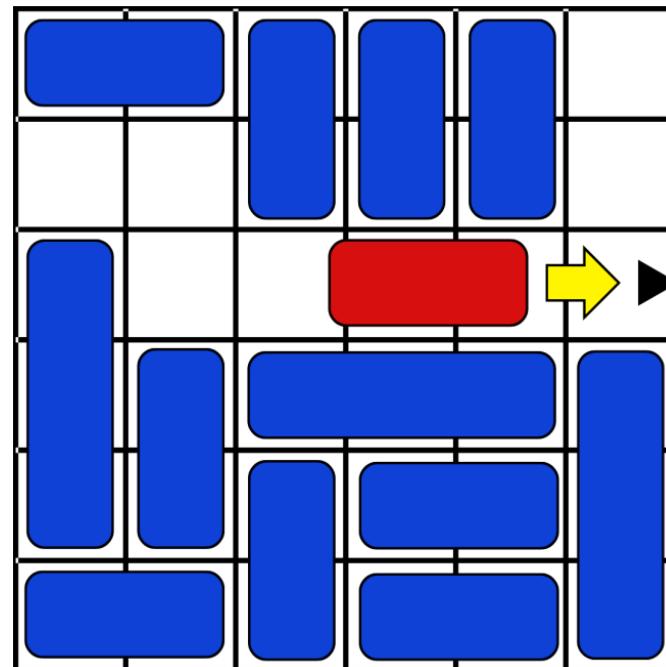


Gambar 3. Gerakan Pertama Game Rush Hour



Gambar 4. Gerakan Kedua Game Rush Hour

Puzzle berikut dinyatakan telah selesai apabila primary piece dapat digeser keluar papan melalui pintu keluar.



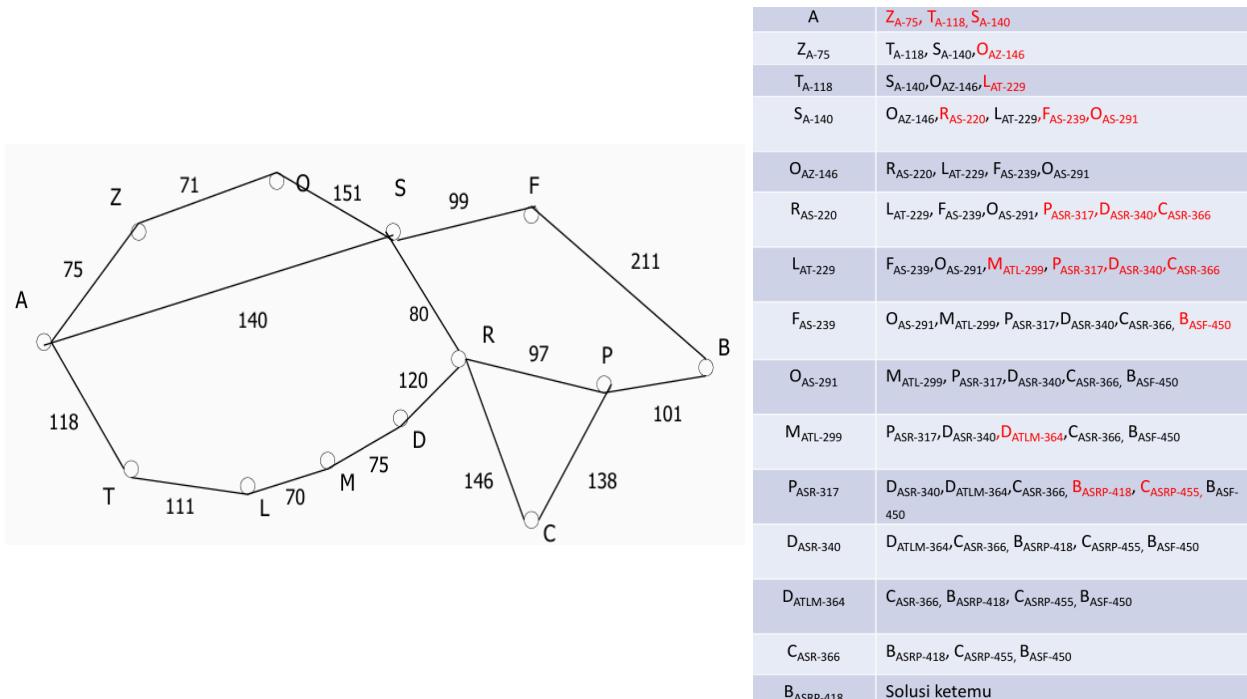
Gambar 5. Pemain Menyelesaikan Permainan

BAB II

Dasar Teori

2.1 Uniform Cost Search

Uniform Cost Search (UCS) adalah algoritma pencarian yang berfokus pada pencarian jalur dengan biaya total terkecil dari titik awal ke simpul tujuan. UCS menggunakan antrean prioritas berdasarkan nilai $g(n)$, yaitu biaya kumulatif dari simpul awal ke simpul saat ini. Algoritma ini memproses simpul dengan biaya terendah terlebih dahulu dan akan selalu menemukan solusi yang optimal selama semua bobot antar simpul (edge cost) bernilai non-negatif. Karena tidak menggunakan estimasi ke tujuan, UCS cenderung menyusuri jalur yang paling murah secara menyeluruh, meskipun belum tentu paling cepat menuju tujuan.

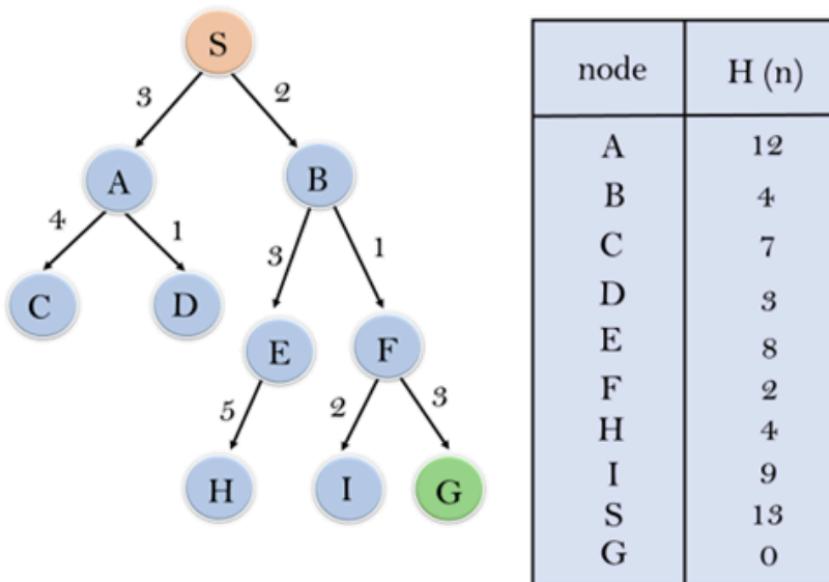


Gambar 2.1 Informed Graph and UCS solution

(Sumber : [Rinaldi Munir - Route Planning part 1](#))

2.2 Best First Search

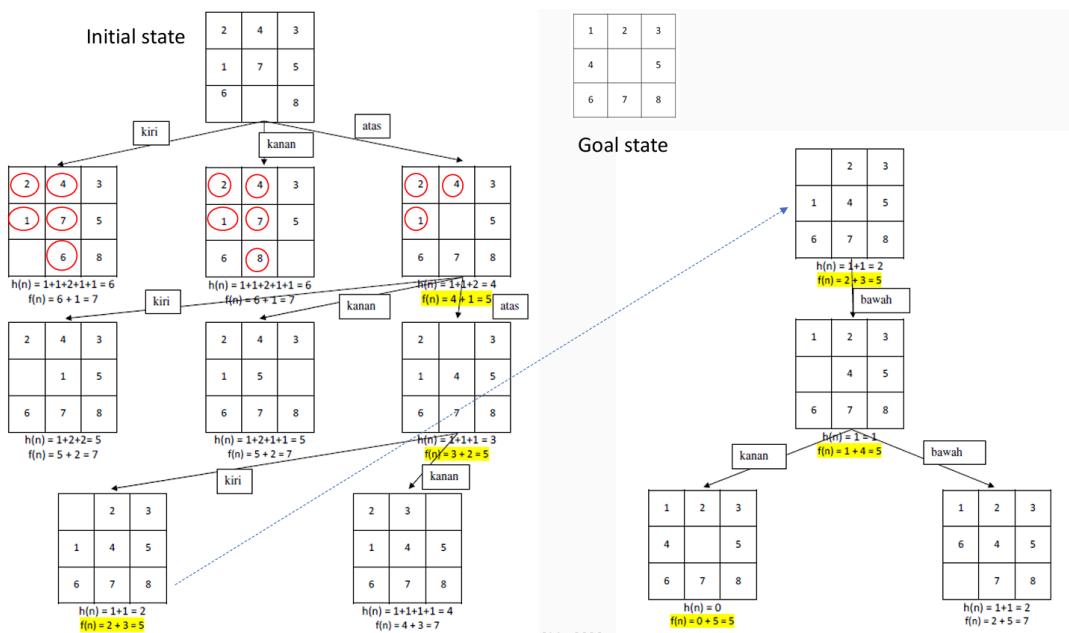
Best First Search adalah algoritma pencarian yang mengandalkan nilai heuristik $h(n)$ untuk memperkirakan jarak atau biaya dari simpul saat ini ke simpul tujuan. Algoritma ini akan memilih simpul yang tampaknya paling dekat ke tujuan menurut estimasi heuristik tersebut. Karena hanya mempertimbangkan $h(n)$ dan mengabaikan $g(n)$, Best First Search dapat bergerak cepat menuju tujuan, namun tidak menjamin bahwa jalur yang ditemukan adalah yang paling optimal. Algoritma ini cocok untuk masalah yang membutuhkan pencarian cepat dengan toleransi terhadap solusi tidak sempurna.



Gambar 2.2 Example of Best First Search
(Sumber : [Best First Search algorithm](#))

2.3 A* Search

A Search* menggabungkan kekuatan UCS dan BestFS dengan menggunakan fungsi evaluasi $f(n) = g(n) + h(n)$, yaitu penjumlahan antara biaya aktual dari awal ke simpul saat ini ($g(n)$) dan estimasi biaya ke tujuan ($h(n)$). Pendekatan ini membuat A^* sangat efisien dalam menemukan jalur optimal sekaligus cepat, selama heuristik yang digunakan bersifat admissible (tidak melebihi biaya sebenarnya) dan consistent (memenuhi sifat segitiga). A^* Search merupakan salah satu algoritma pencarian jalur yang paling populer dan banyak digunakan dalam berbagai aplikasi, seperti game, robotika, dan sistem navigasi.



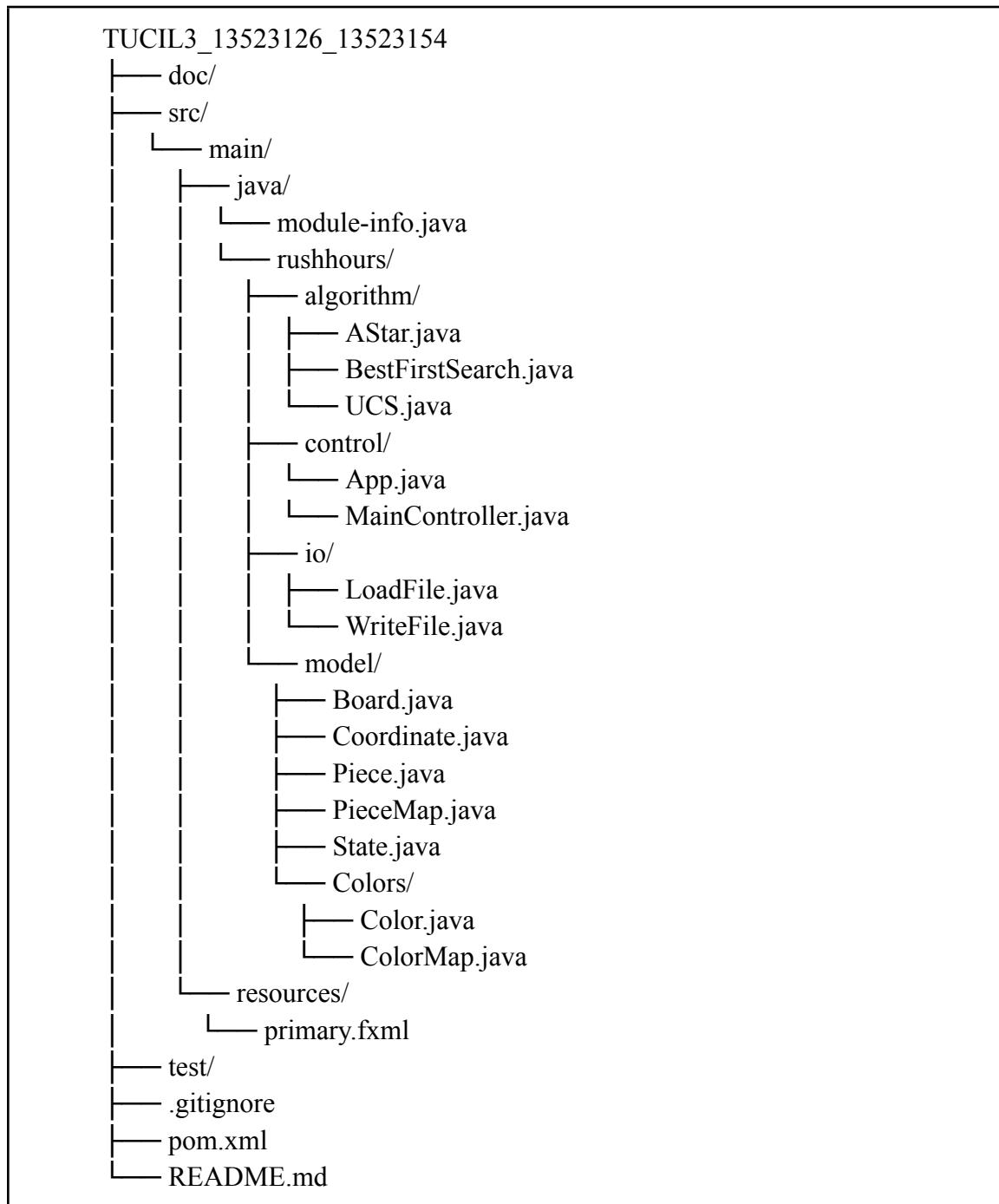
Gambar 2.3 Contoh Penyelesaian Puzzle 8 dengan A^*

(Sumber : [Rinaldi Munir - Route Planning part 2](#))

BAB III

IMPLEMENTASI PROGRAM

3.1 Struktur Folder dan File



3.2 Model

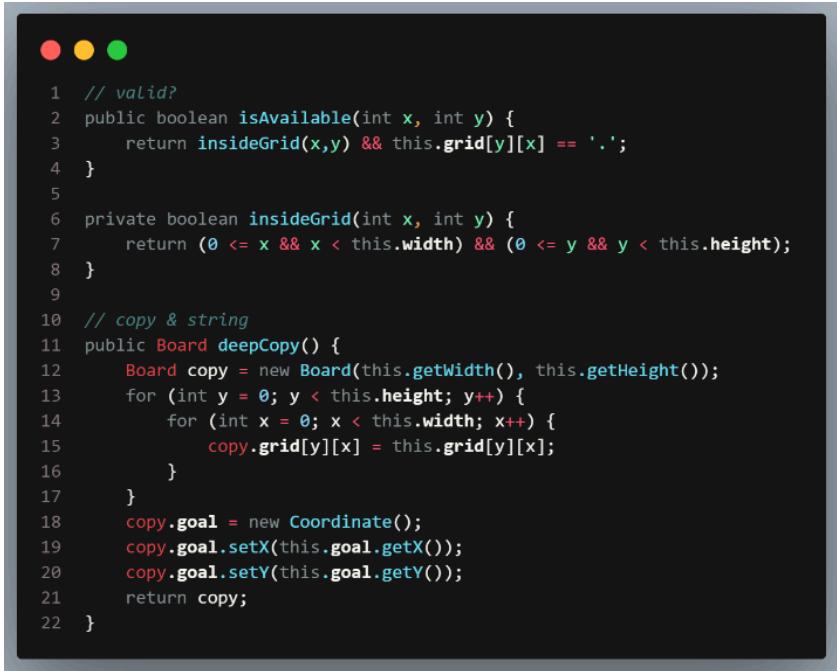
Untuk mempermudah proses dan algoritma rush hour solver, komponen dari permainan akan dipisah menjadi beberapa struktur data. Struktur data yang pertama adalah board, yang memuat papan permainan rush hour yang diisi dengan mobil-mobil / piece dan memiliki blok tujuan.



```
1 public class Board {  
2     private Coordinate goal;  
3     private int height;  
4     private int width;  
5     private char[][] grid;
```

Gambar 3.2.1. Struktur data board

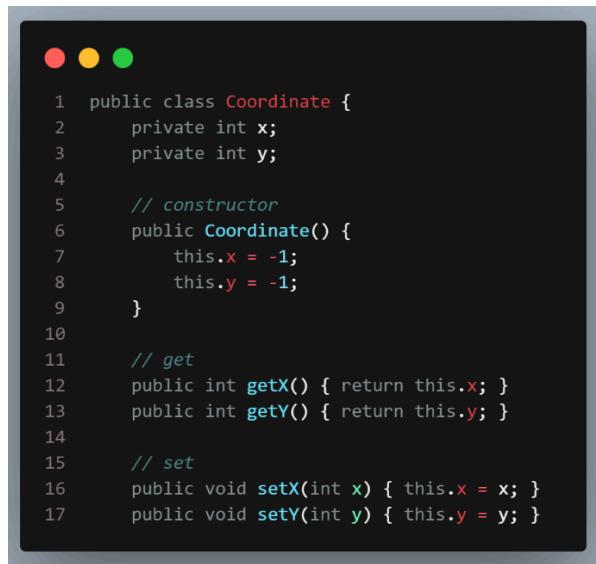
Board berisi koordinat goal, tinggi papan, lebar papan, dan matriks grid berupa isi papan. Board memiliki konstruktor, getter, dan setter seperti untuk setiap komponen di dalamnya. Selain itu, board memiliki fungsi isAvailable untuk memeriksa bila grid atau blok di dalam board masih belum terisi penuh. Board juga menyediakan fungsi deepCopy untuk mengambil semua nilai data dalam board lain dengan menggunakan memory lain.



```
1 // valid?  
2 public boolean isAvailable(int x, int y) {  
3     return insideGrid(x,y) && this.grid[y][x] == '.';  
4 }  
5  
6 private boolean insideGrid(int x, int y) {  
7     return (0 <= x && x < this.width) && (0 <= y && y < this.height);  
8 }  
9  
10 // copy & string  
11 public Board deepCopy() {  
12     Board copy = new Board(this.getWidth(), this.getHeight());  
13     for (int y = 0; y < this.height; y++) {  
14         for (int x = 0; x < this.width; x++) {  
15             copy.grid[y][x] = this.grid[y][x];  
16         }  
17     }  
18     copy.goal = new Coordinate();  
19     copy.goal.setX(this.goal.getX());  
20     copy.goal.setY(this.goal.getY());  
21     return copy;  
22 }
```

Gambar 3.2.3. isAvailable & deepCopy

Setiap sel dalam grid diisi dengan piece atau ‘.’ yang menandakan sel kosong. Setiap sel memiliki koordinat (x, y). Tidak hanya piece yang di dalam grid, tetapi juga goal.



```
1 public class Coordinate {
2     private int x;
3     private int y;
4
5     // constructor
6     public Coordinate() {
7         this.x = -1;
8         this.y = -1;
9     }
10
11    // get
12    public int getX() { return this.x; }
13    public int getY() { return this.y; }
14
15    // set
16    public void setX(int x) { this.x = x; }
17    public void setY(int y) { this.y = y; }
```

Gambar 3.2.4 Struktur data coordinate

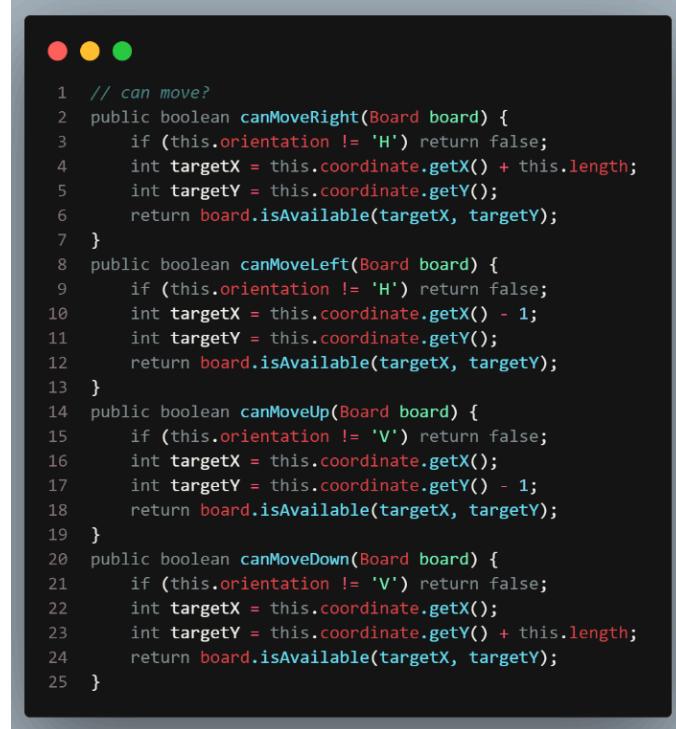
Piece adalah mobil dalam permainan rush hour. Setiap piece pasti memiliki informasi identitas, panjang, letak koordinat pada board yang dihitung dari paling kiri dan atas (tergantung orientasi), dan orientasinya.



```
1 public class Piece {
2     private char id;
3     private int length;
4     private Coordinate coordinate;
5     private char orientation;
```

Gambar 3.2.5 Struktur data piece

Setiap piece harus dapat digerakkan agar permainan dapat dijalankan. Validitas gerakan dapat dicek dan gerakan piece dilakukan dengan potongan kode berikut.



```
1 // can move?
2 public boolean canMoveRight(Board board) {
3     if (this.orientation != 'H') return false;
4     int targetX = this.coordinate.getX() + this.length;
5     int targetY = this.coordinate.getY();
6     return board.isAvailable(targetX, targetY);
7 }
8 public boolean canMoveLeft(Board board) {
9     if (this.orientation != 'H') return false;
10    int targetX = this.coordinate.getX() - 1;
11    int targetY = this.coordinate.getY();
12    return board.isAvailable(targetX, targetY);
13 }
14 public boolean canMoveUp(Board board) {
15     if (this.orientation != 'V') return false;
16     int targetX = this.coordinate.getX();
17     int targetY = this.coordinate.getY() - 1;
18     return board.isAvailable(targetX, targetY);
19 }
20 public boolean canMoveDown(Board board) {
21     if (this.orientation != 'V') return false;
22     int targetX = this.coordinate.getX();
23     int targetY = this.coordinate.getY() + this.length;
24     return board.isAvailable(targetX, targetY);
25 }
```

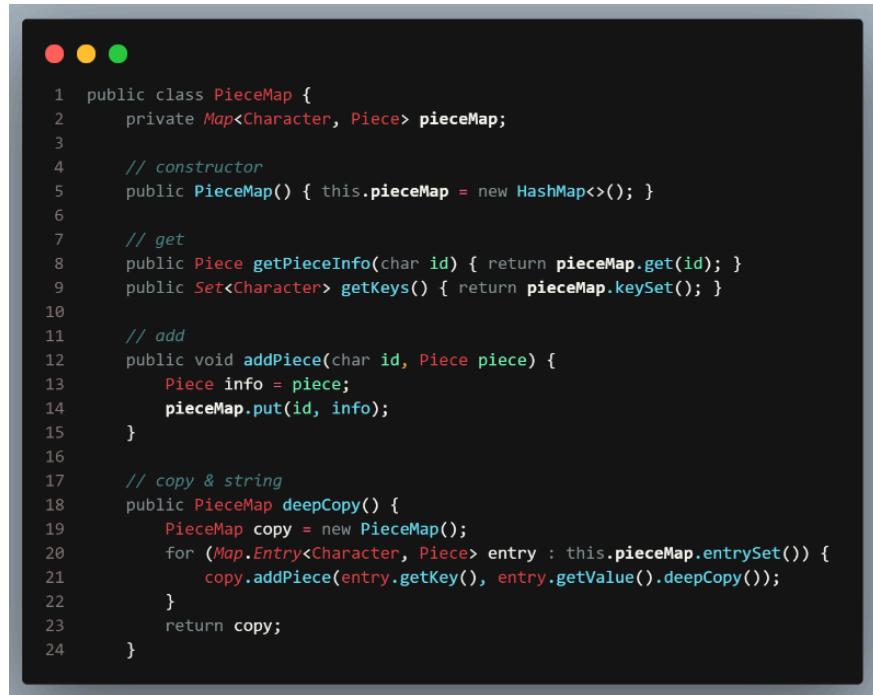
Gambar 3.2.6 canMove



```
1 // piece move
2 public void moveRight(Board board) {
3     int y = this.coordinate.getY();
4     int oldHeadX = this.coordinate.getX();
5     int oldTailX = this.coordinate.getX() + this.length - 1;
6     board.setCell(oldHeadX, y, '.');
7     board.setCell(oldTailX + 1, y, this.id);
8     this.setCoordinate(oldHeadX + 1, y);
9 }
10 public void moveLeft(Board board) {
11     int y = this.coordinate.getY();
12     int oldHeadX = this.coordinate.getX();
13     int oldTailX = this.coordinate.getX() + this.length - 1;
14     board.setCell(oldTailX, y, '.');
15     board.setCell(oldHeadX - 1, y, this.id);
16     this.setCoordinate(oldHeadX - 1, y);
17 }
18
19 public void moveUp(Board board) {
20     int x = this.coordinate.getX();
21     int oldHeadY = this.coordinate.getY();
22     int oldTailY = this.coordinate.getY() + this.length - 1;
23     board.setCell(x, oldTailY, '.');
24     board.setCell(x, oldHeadY - 1, this.id);
25     this.setCoordinate(x, oldHeadY - 1);
26 }
27
28 public void moveDown(Board board) {
29     int x = this.coordinate.getX();
30     int oldHeadY = this.coordinate.getY();
31     int oldTailY = this.coordinate.getY() + this.length - 1;
32     board.setCell(x, oldHeadY, '.');
33     board.setCell(x, oldTailY + 1, this.id);
34     this.setCoordinate(x, oldHeadY + 1);
35 }
```

Gambar 3.2.7 Move

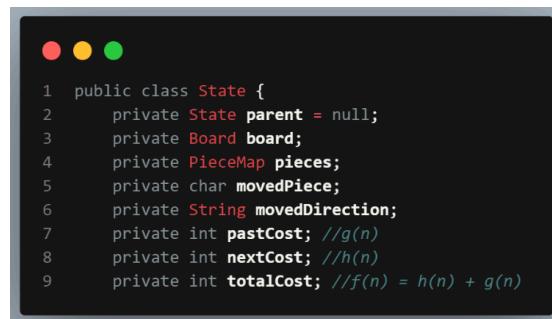
Class PieceMap akan melakukan mapping dari identifier tiap piece dengan tipe character terhadap informasi piece untuk setiap komponen.



```
1 public class PieceMap {
2     private Map<Character, Piece> pieceMap;
3
4     // constructor
5     public PieceMap() { this.pieceMap = new HashMap<>(); }
6
7     // get
8     public Piece getPieceInfo(char id) { return pieceMap.get(id); }
9     public Set<Character> getKeys() { return pieceMap.keySet(); }
10
11    // add
12    public void addPiece(char id, Piece piece) {
13        Piece info = piece;
14        pieceMap.put(id, info);
15    }
16
17    // copy & string
18    public PieceMap deepCopy() {
19        PieceMap copy = new PieceMap();
20        for (Map.Entry<Character, Piece> entry : this.pieceMap.entrySet()) {
21            copy.addPiece(entry.getKey(), entry.getValue().deepCopy());
22        }
23        return copy;
24    }
}
```

Gambar 3.2.8. PieceMap

State adalah kelas yang merepresentasikan simpul (node) dalam graph untuk algoritma pathfinding pada permainan Rush Hour. Kelas ini menyimpan berbagai atribut penting, yaitu: parent yang mereferensikan state sebelumnya untuk membentuk jalur solusi; board yang merepresentasikan konfigurasi papan pada state tersebut; pieceMap, yaitu struktur peta yang menyimpan posisi setiap piece secara efisien; movedPiece, yaitu piece yang digerakkan untuk mencapai state ini; movedDirection, yaitu arah pergerakan piece tersebut; serta cost, yaitu nilai yang digunakan untuk menentukan urutan eksekusi node sesuai dengan algoritma pencarian yang digunakan (seperti UCS, Greedy, atau A*).



```
1 public class State {
2     private State parent = null;
3     private Board board;
4     private PieceMap pieces;
5     private char movedPiece;
6     private String movedDirection;
7     private int pastCost; //g(n)
8     private int nextCost; //h(n)
9     private int totalCost; //f(n) = h(n) + g(n)
}
```

Gambar 3.2.9 Struktur data state

```

1  public Set<State> generateNextStates(HashSet<String> stateSets, String heuristicType) {
2      Set<State> childList = new HashSet<>();
3      for (char key : this.pieces.getKeys()) {
4
5          // right
6          for (int steps = 1; ; steps++) {
7              State child = this.createChild();
8              boolean valid = true;
9              for (int i = 0; i < steps; i++) {
10                  if (child.pieces.getPieceInfo(key).canMoveRight(child.board)) {
11                      child.pieces.getPieceInfo(key).moveRight(child.board);
12                  } else {
13                      valid = false;
14                      break;
15                  }
16              }
17              if (!valid) break;
18              String boardState = child.getBoardState();
19              if (!stateSets.contains(boardState)) {
20                  int value = child.getHeuristicValue(heuristicType, child.board);
21                  child.setPastCost(this.pastCost + 1);
22                  child.setNextCost(value);
23                  child.setTotalCost((child.pastCost + child.nextCost));
24                  child.movedPiece = key;
25                  child.movedDirection = "Right";
26                  childList.add(child);
27              }
28          }
29
30          // Left
31          for (int steps = 1; ; steps++) {
32              State child = this.createChild();
33              boolean valid = true;
34              for (int i = 0; i < steps; i++) {
35                  if (child.pieces.getPieceInfo(key).canMoveLeft(child.board)) {
36                      child.pieces.getPieceInfo(key).moveLeft(child.board);
37                  } else {
38                      valid = false;
39                      break;
40                  }
41              }
42              if (!valid) break;
43              String boardState = child.getBoardState();
44              if (!stateSets.contains(boardState)) {
45                  int value = child.getHeuristicValue(heuristicType, child.board);
46                  child.setPastCost(this.pastCost + 1);
47                  child.setNextCost(value);
48                  child.setTotalCost((child.pastCost + child.nextCost));
49                  child.movedPiece = key;
50                  child.movedDirection = "Left";
51                  childList.add(child);
52              }
53          }
54      }
55  }

```

Gambar 3.2.10 generateNextStates (part 1)



```
1     // up
2     for (int steps = 1; ; steps++) {
3         State child = this.createChild();
4         boolean valid = true;
5         for (int i = 0; i < steps; i++) {
6             if (child.pieces.getPieceInfo(key).canMoveUp(child.board)) {
7                 child.pieces.getPieceInfo(key).moveUp(child.board);
8             } else {
9                 valid = false;
10                break;
11            }
12        }
13        if (!valid) break;
14        String boardState = child.getBoardState();
15        if (!stateSets.contains(boardState)) {
16            int value = child.getHeuristicValue(heuristicType, child.board);
17            child.setPastCost(this.pastCost + 1);
18            child.setNextCost(value);
19            child.setTotalCost(child.pastCost + child.nextCost);
20            child.movedPiece = key;
21            child.movedDirection = "Up";
22            childList.add(child);
23        }
24    }
25
26    // down
27    for (int steps = 1; ; steps++) {
28        State child = this.createChild();
29        boolean valid = true;
30        for (int i = 0; i < steps; i++) {
31            if (child.pieces.getPieceInfo(key).canMoveDown(child.board)) {
32                child.pieces.getPieceInfo(key).moveDown(child.board);
33            } else {
34                valid = false;
35                break;
36            }
37        }
38        if (!valid) break;
39        String boardState = child.getBoardState();
40        if (!stateSets.contains(boardState)) {
41            int value = child.getHeuristicValue(heuristicType, child.board);
42            child.setPastCost(this.pastCost + 1);
43            child.setNextCost(value);
44            child.setTotalCost(child.pastCost + child.nextCost);
45            child.movedPiece = key;
46            child.movedDirection = "Down";
47            childList.add(child);
48        }
49    }
50 }
51 return childList;
52 }
```

Gambar 3.2.11 generateNextStates (part 2)

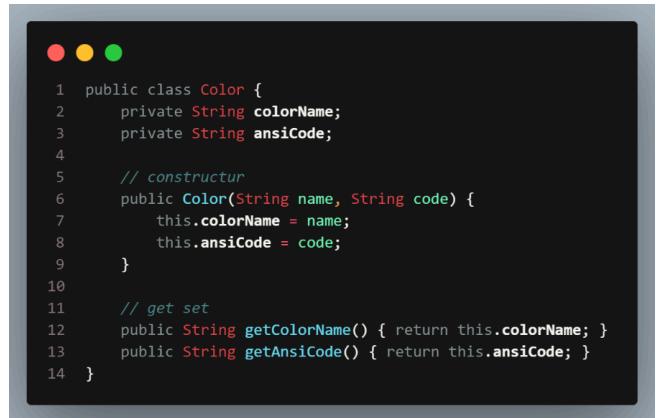
Gambar 3.2.10 mengacu pada fungsi generateNextStates, yang digunakan untuk menghasilkan sejumlah child yang mungkin untuk dijalani. Semua piece dalam papan akan diiterasi lalu dicek arah yang mereka mampu lalui. Bila mereka bisa geser menuju arah tertentu (dicek dengan isAvailable), maka program akan menambah child sebanyak berapa kali ia bisa digeser. Child dibuat dengan melakukan deepCopy dari state dengan potongan kode dibawah.



```
1 public State createChild() {
2     Board boardCopy = this.board.deepCopy();
3     PieceMap piecesCopy = this.pieces.deepCopy();
4     State child = new State(boardCopy, piecesCopy);
5     child.parent = this;
6     return child;
7 }
8
```

Gambar 3.2.12 createChild

Untuk memperindah tampilan dalam CLI maupun GUI, terdapat struktur data Color dan ColorMap.



```
1 public class Color {
2     private String colorName;
3     private String ansiCode;
4
5     // constructor
6     public Color(String name, String code) {
7         this.colorName = name;
8         this	ansiCode = code;
9     }
10
11    // get set
12    public String getColorName() { return this.colorName; }
13    public String getAnsiCode() { return this.ansiCode; }
14 }
```

Gambar 3.2.13 Struktur data color

Color mengandung nama warna dan ansiCode yang berupa kode yang mengacu pada warna tersebut. Sebanyak 24 warna yang berbeda dikumpulkan dalam ColorMap untuk memberi warna kepada seluruh abjad piece dengan pengecualian ‘P’ untuk primary piece dan ‘K’ untuk exit.

```

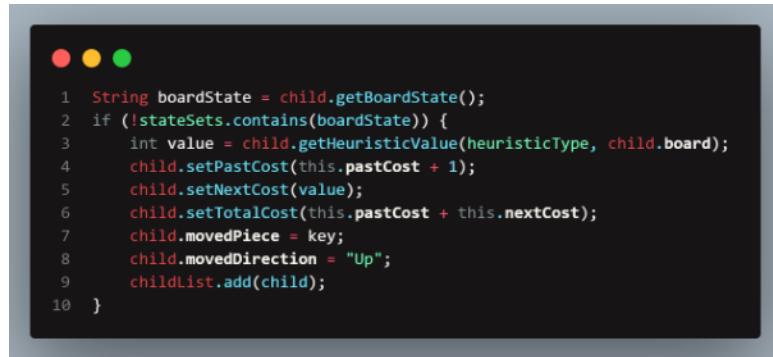
1  public class ColorMap {
2      HashMap <Character, Color> colorMap;
3      ArrayList <Color> colorSet;
4
5      // constructor
6      public ColorMap() {
7          this.colorMap = new HashMap<>();
8          this.colorSet = new ArrayList<>();
9
10         this.colorSet.add(new Color("GREEN", "\u001B[42;30m"));
11        this.colorSet.add(new Color("YELLOW", "\u001B[43;30m"));
12        this.colorSet.add(new Color("BLUE", "\u001B[44;37m"));
13        this.colorSet.add(new Color("MAGENTA", "\u001B[45;30m"));
14        this.colorSet.add(new Color("CYAN", "\u001B[46;30m"));
15
16        this.colorSet.add(new Color("BRIGHT_GREEN", "\u001B[102;30m"));
17        this.colorSet.add(new Color("BRIGHT_YELLOW", "\u001B[103;30m"));
18        this.colorSet.add(new Color("BRIGHT_BLUE", "\u001B[104;30m"));
19        this.colorSet.add(new Color("BRIGHT_MAGENTA", "\u001B[105;30m"));
20        this.colorSet.add(new Color("BRIGHT_CYAN", "\u001B[106;30m"));
21
22        this.colorSet.add(new Color("BOLD_GREEN", "\u001B[1;42;30m"));
23        this.colorSet.add(new Color("BOLD_YELLOW", "\u001B[1;43;30m"));
24        this.colorSet.add(new Color("BOLD_BLUE", "\u001B[1;44;37m"));
25        this.colorSet.add(new Color("BOLD_MAGENTA", "\u001B[1;45;30m"));
26        this.colorSet.add(new Color("BOLD_CYAN", "\u001B[1;46;30m"));
27
28        this.colorSet.add(new Color("BRIGHT_BOLD_GREEN", "\u001B[1;102;30m"));
29        this.colorSet.add(new Color("BRIGHT_BOLD_YELLOW", "\u001B[1;103;30m"));
30        this.colorSet.add(new Color("BRIGHT_BOLD_BLUE", "\u001B[1;104;37m"));
31        this.colorSet.add(new Color("BRIGHT_BOLD_MAGENTA", "\u001B[1;105;30m"));
32        this.colorSet.add(new Color("BRIGHT_BOLD_CYAN", "\u001B[1;106;30m"));
33
34        this.colorSet.add(new Color("GRAY", "\u001B[100;30m"));
35        this.colorSet.add(new Color("BRIGHT_GRAY", "\u001B[1;100;30m"));
36        this.colorSet.add(new Color("BOLD_GRAY", "\u001B[1;100;37m"));
37        this.colorSet.add(new Color("BRIGHT_BOLD_GRAY", "\u001B[1;100;97m"));
38    }
39
40
41    // map
42    public void mapColorToPieces(PieceMap pieceMap) {
43        int colorCount = this.colorSet.size();
44        int i = 0;
45        this.colorMap.clear();
46        for (char key : pieceMap.getKeys()) {
47            if(key == 'P') continue;
48            this.colorMap.put(key, this.colorSet.get(i % colorCount));
49            i++;
50        }
51    }
52
53    public HashMap<Character, Color> getColorMap() { return this.colorMap; }
54    public Color getColor(char key) { return this.colorMap.get(key); }
55 }

```

Gambar 3.2.14 Stuktur data colorMap

3.3 Algoritma

Poin utama dari program ini adalah algoritmanya. Algoritma yang tersedia dalam penyelesaian permainan ini ada 3, yaitu Uniform Cost Search, Greedy Best First Search, dan A* Search. Algoritma dibawah ini mengimplementasikan fungsi generateNextStates() untuk setiap fungsinya, yang pada dasarnya fungsi tersebut sudah menginisialisasikan g(n), h(n), dan f(n) untuk setiap algoritma setiap pembuatan child, terlepas apakah digunakan atau tidak.



```
● ● ●

1 String boardState = child.getBoardState();
2 if (!stateSets.contains(boardState)) {
3     int value = child.getHeuristicValue(heuristicType, child.board);
4     child.setPastCost(this.pastCost + 1);
5     child.setNextCost(value);
6     child.setTotalCost(this.pastCost + this.nextCost);
7     child.movedPiece = key;
8     child.movedDirection = "Up";
9     childList.add(child);
10 }
```

Gambar 3.3.1 Inisialisasi g(n), h(n), f(n)

Seperti yang telah dijelaskan pada Bab analisis algoritma, maka solve() pada tiap fungsi di kelas algoritma sudah tidak perlu kuatir mengenai inisialisasi cost child, selain inisialisasi cost dirinya sendiri.

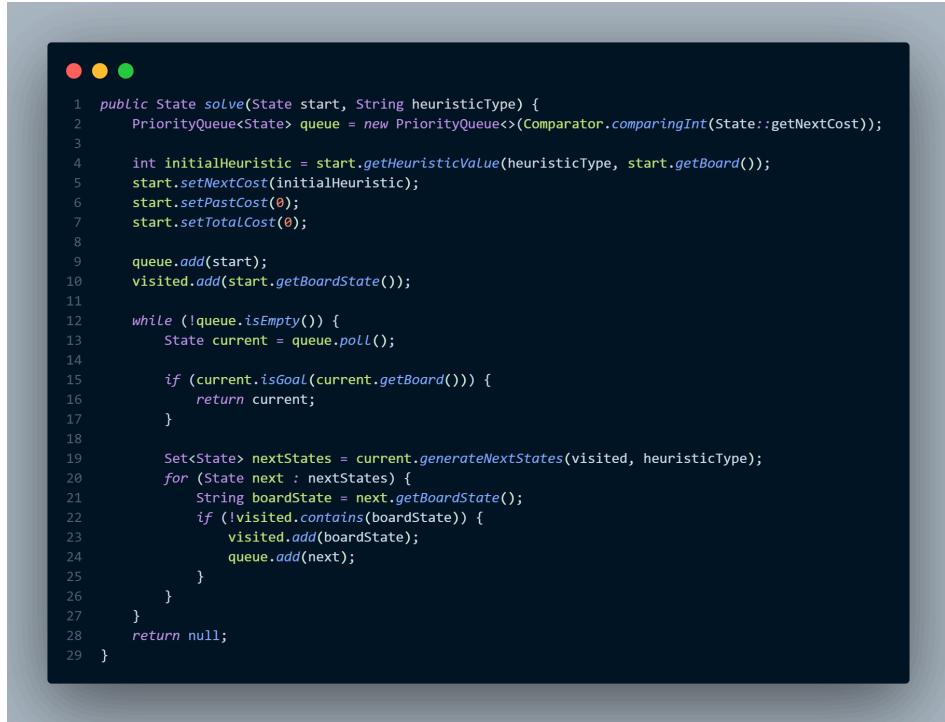


```
● ● ●

1 public State solve(State initialState) {
2     PriorityQueue<State> queue = new PriorityQueue<>(Comparator.comparingInt(State::getPastCost));
3
4     initialState.setNextCost(0);
5     initialState.setPastCost(0);
6     initialState.setTotalCost(0);
7
8     queue.add(initialState);
9     while (!queue.isEmpty()) {
10         State current = queue.poll();
11
12         if (current.isGoal(current.getBoard())) {
13             return current;
14         }
15
16         String stateKey = current.getBoardState();
17         if (visited.contains(stateKey)) continue;
18         visited.add(stateKey);
19
20         for (State next : current.generateNextStates(new HashSet<>(visited), "UCS(No Heuristic)")) {
21             if (!visited.contains(next.getBoardState())) {
22                 next.setNextCost(current.primaryDistanceToGoal(next.getBoard()));
23                 next.setTotalCost(next.getPastCost());
24                 queue.add(next);
25             }
26         }
27     }
28     return null;
29 }
```

Gambar 3.3.2 UCS Solver

UCS tidak menggunakan heuristik , maka itu input parameter untuk fungsinya tidak memerlukan heuristicType bertipe String.



```
1 public State solve(State start, String heuristicType) {
2     PriorityQueue<State> queue = new PriorityQueue<>(Comparator.comparingInt(State::getNextCost));
3
4     int initialHeuristic = start.getHeuristicValue(heuristicType, start.getBoard());
5     start.setNextCost(initialHeuristic);
6     start.setPastCost(0);
7     start.setTotalCost(0);
8
9     queue.add(start);
10    visited.add(start.getBoardState());
11
12    while (!queue.isEmpty()) {
13        State current = queue.poll();
14
15        if (current.isGoal(current.getBoard())) {
16            return current;
17        }
18
19        Set<State> nextStates = current.generateNextStates(visited, heuristicType);
20        for (State next : nextStates) {
21            String boardState = next.getBoardState();
22            if (!visited.contains(boardState)) {
23                visited.add(boardState);
24                queue.add(next);
25            }
26        }
27    }
28    return null;
29 }
```

Gambar 3.3.3 Best First Search Solver



```
1 public State solve(State initialState, String heuristicType) {
2     PriorityQueue<State> queue = new PriorityQueue<>(Comparator.comparingInt(State::getTotalCost));
3
4     initialState.setPastCost(0);
5     initialState.setNextCost(initialState.getHeuristicValue(heuristicType, initialState.getBoard()));
6     initialState.setTotalCost(initialState.getPastCost() + initialState.getNextCost());
7
8     queue.add(initialState);
9     while (!queue.isEmpty()) {
10        State current = queue.poll();
11
12        if (current.isGoal(current.getBoard())) {
13            visited.add(current.getBoardState());
14            return current;
15        }
16
17        String stateKey = current.getBoardState();
18        if (visited.contains(stateKey)) continue;
19        visited.add(stateKey);
20
21        for (State next : current.generateNextStates(new HashSet<>(visited), heuristicType)) {
22            if (!visited.contains(next.getBoardState())) {
23                queue.add(next);
24            }
25        }
26    }
27    return null;
28 }
```

Gambar 3.3.4 A* Solver

Best First Search dan A* disisi lain memerlukan heuristicType bertipe String pada parameter inputnya pada fungsi Solve() untuk menentukan heuristik mana yang akan digunakan berdasarkan input dari user.

```

1 public final int getVisitedNode() {
2     return this.visited.size();
3 }
4

```

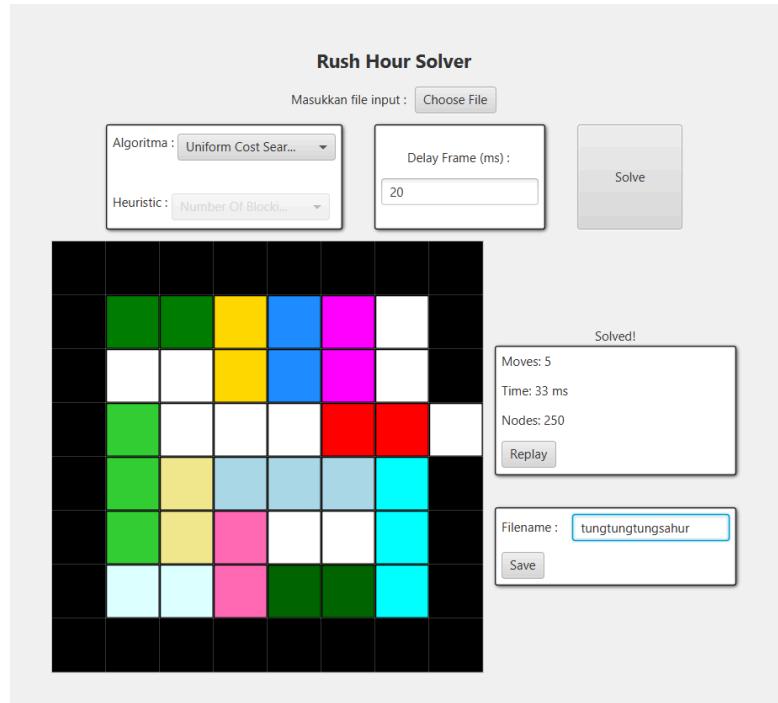
Gambar 3.3.5 getVisitedNode

Fungsi `getVisitedNode()` merupakan fungsi yang digunakan untuk menentukan node jumlah yang telah dilalui oleh algoritma tertentu. Pada implementasi yang kami lakukan, kami menyimpan state board dalam bentuk string pada hashset untuk memastikan bahwa setiap state pada board adalah beda dan tidak dilakukan pengecekan pada state board yang sama (node yang sama), oleh karena itu dengan mengembalikan size dari hashset, sudah dipastikan bahwa jumlah node yang dilalui adalah ukuran jumlah hashset itu sendiri.

3.4 Bonus

3.4.1 Graphical User Interface

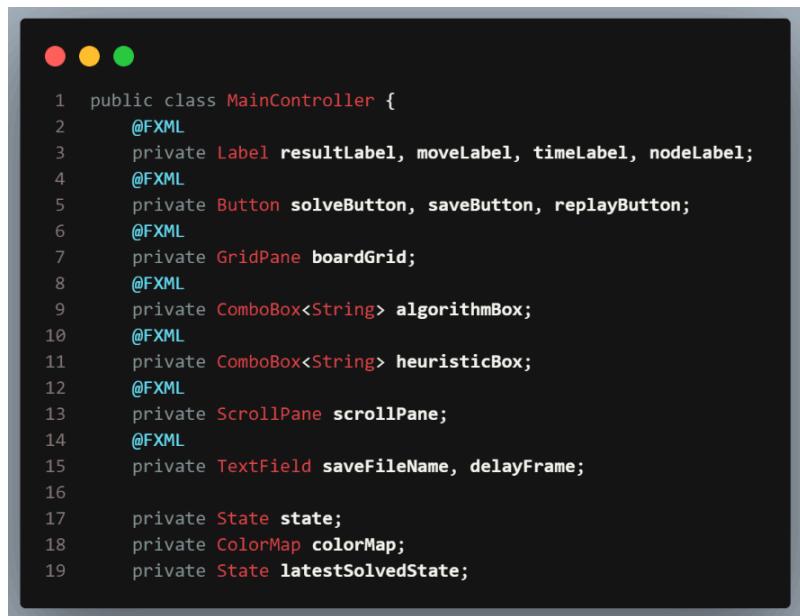
Untuk menampilkan program dengan GUI, kami menggunakan JavaFX. Tiga komponen utama GUI adalah file FXML, MainController dan App. FXML adalah markup language seperti HTML yang dirancang untuk JavaFX. Komponen ini adalah bagian yang ditampilkan pada GUI. Berikut ini adalah tampilan dari program kami menggunakan FXML.



Gambar 3.4.1.1 Tampilan Setelah Melakukan Solve

Sebelum dapat melakukan solve, pengguna harus melakukan 3 sampai 4 hal, yaitu memilih file.txt yang berisi input dimensi dan papan dengan piece, jenis algoritma, delay frame dalam penampilan animasi, dan heuristik. Heuristik berlaku untuk Best First Search dan A*. Ketika memilih UCS, program akan men-disable kotak heuristik karena tidak digunakan. Setelah melakukan solve, pengguna bisa menekan tombol replay untuk mengulangi animasi. Fitur ini diadakan untuk menghindari solve ulang yang lama demi melihat animasi. Terakhir, pengguna mampu menyimpan langkah pada file .txt sesuai dengan nama file yang ditulis.

Untuk menghubungkan tampilan dengan logic, dibutuhkan perantara yang menghubungkan keduanya. Perantara ini adalah MainController.



```
1 public class MainController {
2     @FXML
3     private Label resultLabel, moveLabel, timeLabel, nodeLabel;
4     @FXML
5     private Button solveButton, saveButton, replayButton;
6     @FXML
7     private GridPane boardGrid;
8     @FXML
9     private ComboBox<String> algorithmBox;
10    @FXML
11    private ComboBox<String> heuristicBox;
12    @FXML
13    private ScrollPane scrollPane;
14    @FXML
15    private TextField saveFileName, delayFrame;
16
17    private State state;
18    private ColorMap colorMap;
19    private State latestSolvedState;
```

Gambar 3.4.1.2 Private attribute MainController

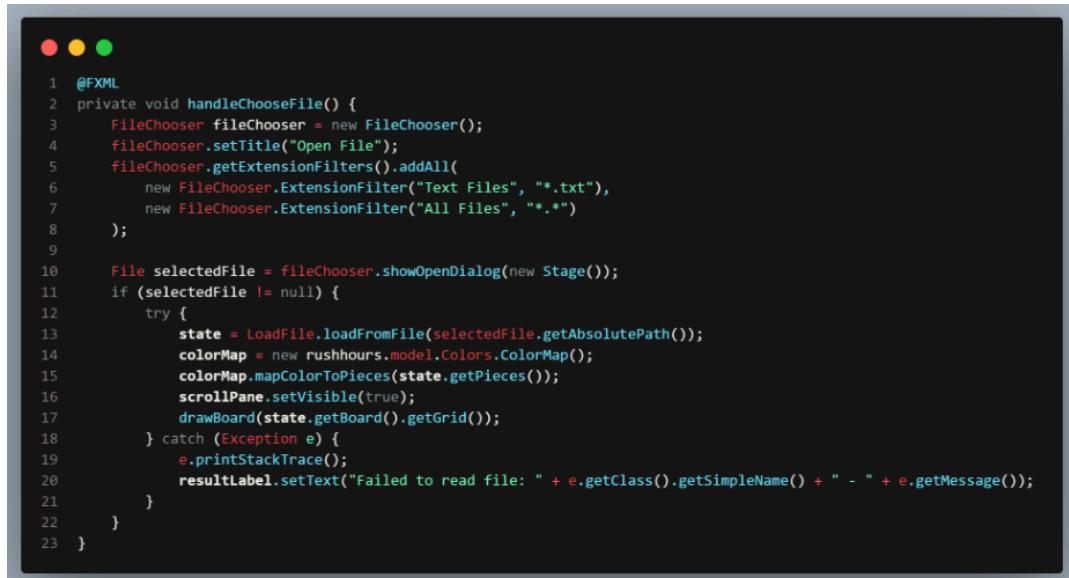
Gambar diatas menunjukkan atribut privat yang digunakan pada kelas ini. Atribut dengan `@FXML` berarti atribut pada FXML. MainController membantu inisialisasi tampilan seperti berikut.



```
1 @FXML
2 public void initialize() {
3     algorithmBox.setItems(FXCollections.observableArrayList(
4         "Uniform Cost Search",
5         "Best First Search",
6         "A* Search"
7     ));
8
9     heuristicBox.setItems(FXCollections.observableArrayList(
10        "Distance To Goal",
11        "Number Of Blocking Piece",
12        "Distance To Goal & Number Of Blocking Piece"
13    ));
14
15     scrollPane.setVisible(false);
16     saveButton.setVisible(false);
17     saveButton.setDisable(true);
18     replayButton.setVisible(false);
19     replayButton.setDisable(true);
20     heuristicBox.setDisable(true);
21
22     checkAlgo();
23 }
24
25 @FXML
26 private void checkAlgo() {
27     algorithmBox.valueProperty().addListener((obs, oldVal, newVal) -> {
28         if ("Uniform Cost Search".equals(newVal)) {
29             heuristicBox.setDisable(true);
30         } else {
31             heuristicBox.setDisable(false);
32         }
33     });
34 }
```

Gambar 3.4.1.3 Initialize

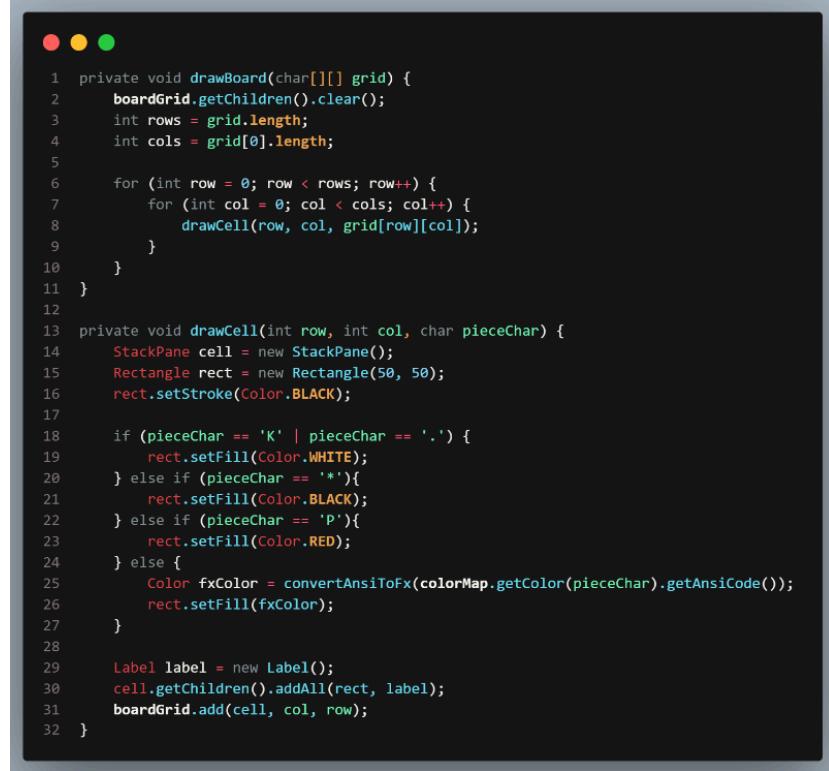
Inisialisasi yang dilakukan adalah mengisi isi dari kotak pilihan algoritma dan heuristik. Metode diatas juga memastikan hanya UCS yang tidak dapat memilih heuristik. Untuk membaca file melalui GUI, dibuat fungsi handleChooseFile yang berguna untuk memilih file melalui tombol.



```
1 @FXML
2 private void handleChooseFile() {
3     FileChooser fileChooser = new FileChooser();
4     fileChooser.setTitle("Open File");
5     fileChooser.getExtensionFilters().addAll(
6         new FileChooser.ExtensionFilter("Text Files", "*.txt"),
7         new FileChooser.ExtensionFilter("All Files", "*.*")
8     );
9
10    File selectedFile = fileChooser.showOpenDialog(new Stage());
11    if (selectedFile != null) {
12        try {
13            state = LoadFile.loadFromFile(selectedFile.getAbsolutePath());
14            colorMap = new rushhours.model.Colors.ColorMap();
15            colorMap.mapColorToPieces(state.getPieces());
16            scrollPane.setVisible(true);
17            drawBoard(state.getBoard().getGrid());
18        } catch (Exception e) {
19            e.printStackTrace();
20            resultLabel.setText("Failed to read file: " + e.getClass().getSimpleName() + " - " + e.getMessage());
21        }
22    }
23 }
```

Gambar 3.4.1.4 handleChooseFile

Grid dan sel didalamnya pada GUI ditampilkan dengan metode berikut. Pewarnaan setiap sel memanfaatkan struktur data color dan colormap yang berisi 24 warna berbeda untuk menunjukkan warna piece selain primary piece, exit / sel kosong, dan border.



```
● ● ●
1 private void drawBoard(char[][] grid) {
2     boardGrid.getChildren().clear();
3     int rows = grid.length;
4     int cols = grid[0].length;
5
6     for (int row = 0; row < rows; row++) {
7         for (int col = 0; col < cols; col++) {
8             drawCell(row, col, grid[row][col]);
9         }
10    }
11 }
12
13 private void drawCell(int row, int col, char pieceChar) {
14     StackPane cell = new StackPane();
15     Rectangle rect = new Rectangle(50, 50);
16     rect.setStroke(Color.BLACK);
17
18     if (pieceChar == 'K' | pieceChar == '.') {
19         rect.setFill(Color.WHITE);
20     } else if (pieceChar == '*'){
21         rect.setFill(Color.BLACK);
22     } else if (pieceChar == 'P'){
23         rect.setFill(Color.RED);
24     } else {
25         Color fxColor = convertAnsiToFx(colorMap.getColor(pieceChar).getAnsiCode());
26         rect.setFill(fxColor);
27     }
28
29     Label label = new Label();
30     cell.getChildren().addAll(rect, label);
31     boardGrid.add(cell, col, row);
32 }
```

Gambar 3.4.1.5 drawBoard & drawCell

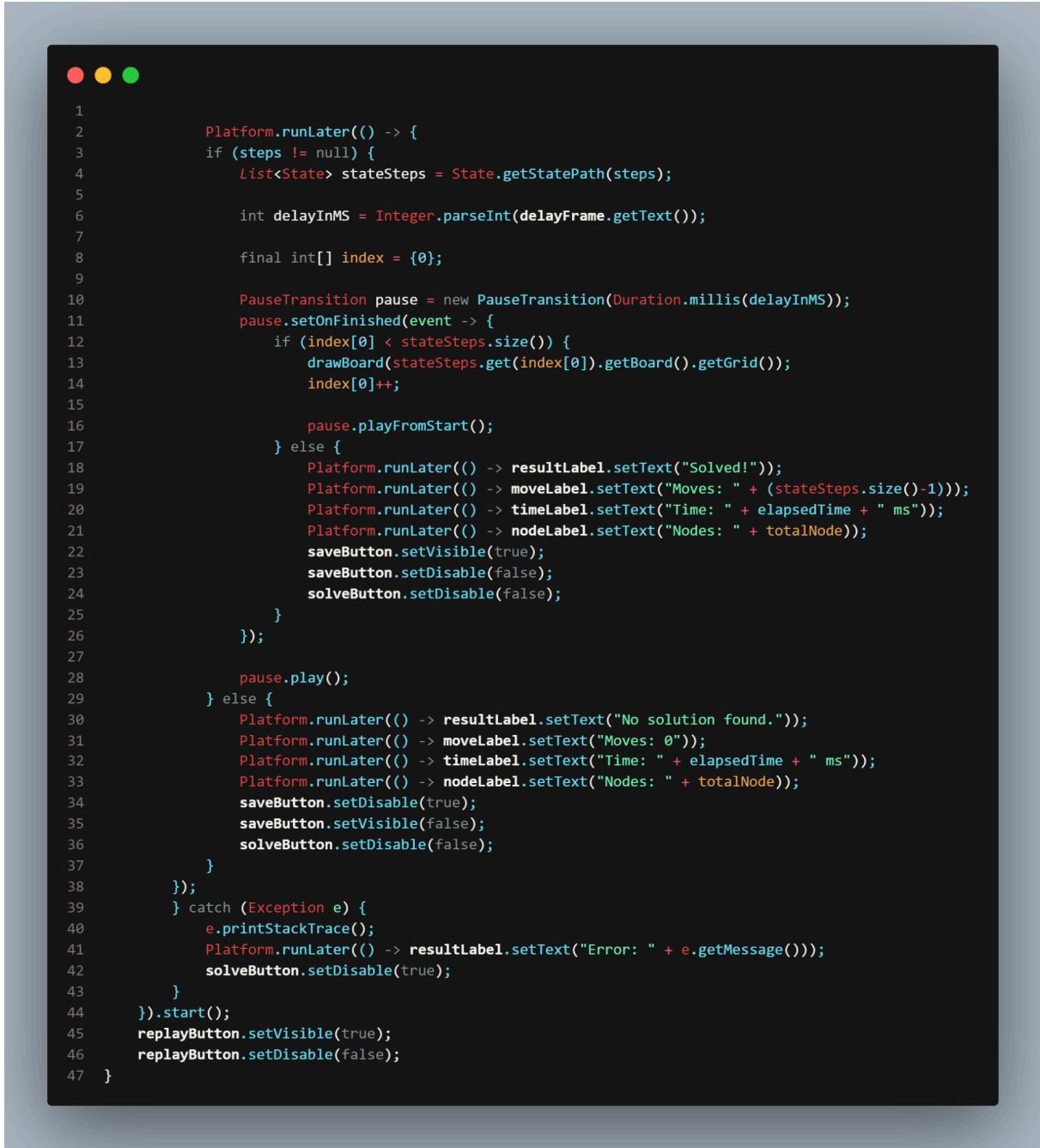
Tombol solve pada tampilan akan memanggil method solvePuzzle. Metode ini akan memanggil solver sesuai algoritma dan heuristik yang dipilih. Waktu pencarian, jumlah node yang dikunjungi, dan gerakan piece akan dicatat.



The screenshot shows a Java code editor with a dark theme. The window title bar has three colored circles (red, yellow, green) at the top left. The main area contains the following Java code:

```
1  @FXML
2  private void solvePuzzle() {
3      solveButton.setDisable(true);
4      resultLabel.setText("Mikir dulu....");
5
6      new Thread(() -> {
7          try {
8              long startTime = System.currentTimeMillis();
9
10         State startState = state;
11         State steps;
12
13         String algo = algorithmBox.getValue();
14         String heur = heuristicBox.getValue();
15
16         if (algo != "Uniform Cost Search"){
17             if (algo == null || heur == null){
18                 Platform.runLater(() -> resultLabel.setText("Pilih algoritma dan/atau heuristik terlebih dahulu"));
19                 solveButton.setDisable(false);
20                 return;
21             }
22         } else if (delayFrame.getText() == null || delayFrame.getText().trim().isEmpty()) {
23             Platform.runLater(() -> resultLabel.setText("Isi dulu delay framesnya.."));
24             solveButton.setDisable(false);
25             return;
26         }
27
28         final int totalNode;
29         if (algo == "Best First Search"){
30             BestFirstSearch solver = new BestFirstSearch();
31             steps = solver.solve(startState, heur);
32             totalNode = solver.getVisitedNode();
33         } else if (algo == "A* Search") {
34             AStar solver = new AStar();
35             steps = solver.solve(startState, heur);
36             totalNode = solver.getVisitedNode();
37         } else if (algo == "Uniform Cost Search") {
38             UCS solver = new UCS();
39             steps = solver.solve(startState);
40             totalNode = solver.getVisitedNode();
41
42         } else {
43             return;
44         }
45
46         long elapsedTime = System.currentTimeMillis() - startTime;
47         latestSolvedState = steps;
48
49         Platform.runLater(() -> resultLabel.setText("Solving.."));
50     }
}
```

Gambar 3.4.1.6 solvePuzzle() (part 1)

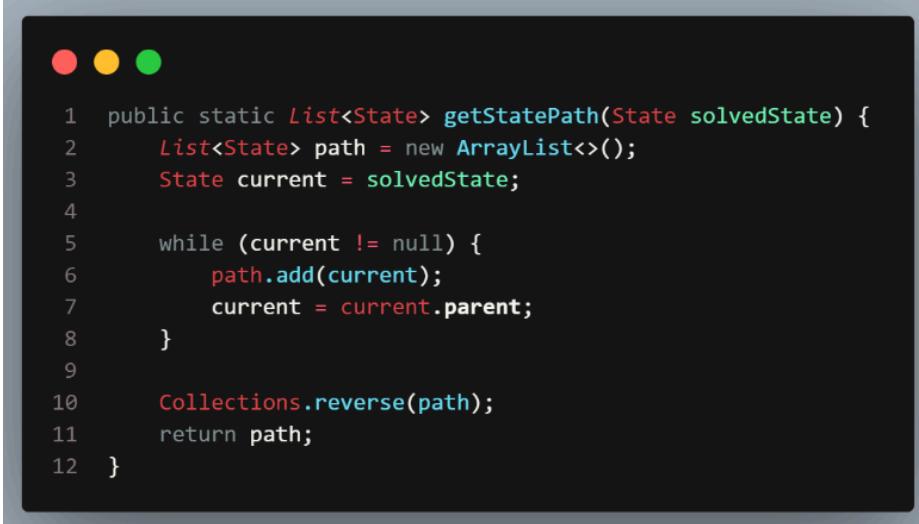


The screenshot shows a Java application window with a dark theme. At the top, there are three colored window control buttons (red, yellow, green). Below them is a code editor containing Java code. The code is part of a method named solvePuzzle(). It handles solving a puzzle by running later tasks to update labels and draw a board. It also handles exceptions and starts a timer if no solution is found.

```
1     Platform.runLater(() -> {
2         if (steps != null) {
3             List<State> stateSteps = State.getStatePath(steps);
4
5             int delayInMS = Integer.parseInt(delayFrame.getText());
6
7             final int[] index = {0};
8
9             PauseTransition pause = new PauseTransition(Duration.millis(delayInMS));
10            pause.setOnFinished(event -> {
11                if (index[0] < stateSteps.size()) {
12                    drawBoard(stateSteps.get(index[0]).getBoard().getGrid());
13                    index[0]++;
14
15                    pause.playFromStart();
16                } else {
17                    Platform.runLater(() -> resultLabel.setText("Solved!"));
18                    Platform.runLater(() -> moveLabel.setText("Moves: " + (stateSteps.size()-1)));
19                    Platform.runLater(() -> timeLabel.setText("Time: " + elapsedTime + " ms"));
20                    Platform.runLater(() -> nodeLabel.setText("Nodes: " + totalNode));
21
22                    saveButton.setVisible(true);
23                    saveButton.setDisable(false);
24                    solveButton.setDisable(false);
25                }
26            });
27
28            pause.play();
29        } else {
30            Platform.runLater(() -> resultLabel.setText("No solution found."));
31            Platform.runLater(() -> moveLabel.setText("Moves: 0"));
32            Platform.runLater(() -> timeLabel.setText("Time: " + elapsedTime + " ms"));
33            Platform.runLater(() -> nodeLabel.setText("Nodes: " + totalNode));
34
35            saveButton.setDisable(true);
36            saveButton.setVisible(false);
37            solveButton.setDisable(false);
38        }
39    } catch (Exception e) {
40        e.printStackTrace();
41        Platform.runLater(() -> resultLabel.setText("Error: " + e.getMessage()));
42        solveButton.setDisable(true);
43    }
44 }).start();
45 replayButton.setVisible(true);
46 replayButton.setDisable(false);
47 }
```

Gambar 3.4.1.7 solvePuzzle() (part 2)

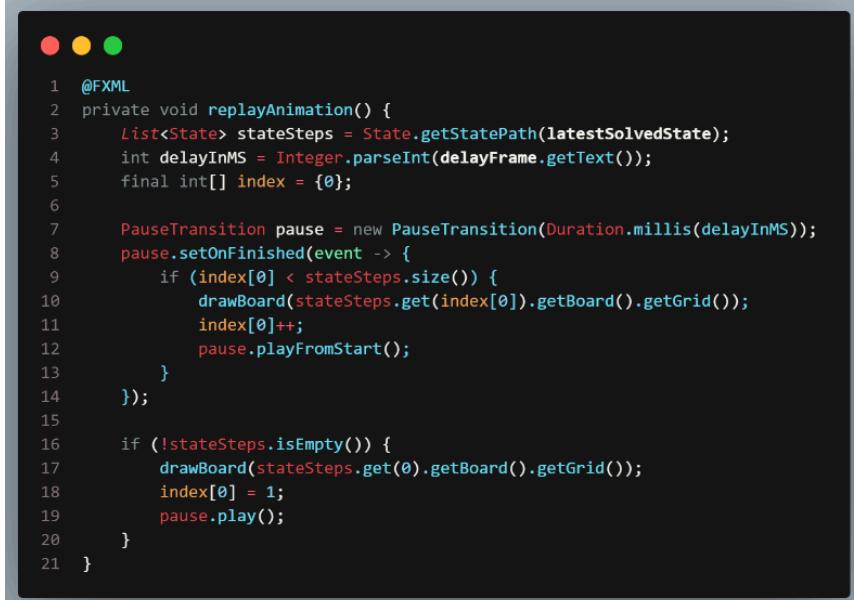
Setelah permasalahan di-solve, hasil yang berupa state akan diambil langkah-langkahnya melalui metode getStatePath.



```
1 public static List<State> getStatePath(State solvedState) {  
2     List<State> path = new ArrayList<>();  
3     State current = solvedState;  
4  
5     while (current != null) {  
6         path.add(current);  
7         current = current.parent;  
8     }  
9  
10    Collections.reverse(path);  
11    return path;  
12 }
```

Gambar 3.4.1.8 getSPATH

Langkah-langkah dalam rupa list of state sehingga dapat ditampilkan seperti animasi dengan iterasi drawBoard dengan delayFrame sesuai input. Replay button memanfaatkan atribut latestSolvedState, yang menyimpan hasil solve terakhir yang berhasil.



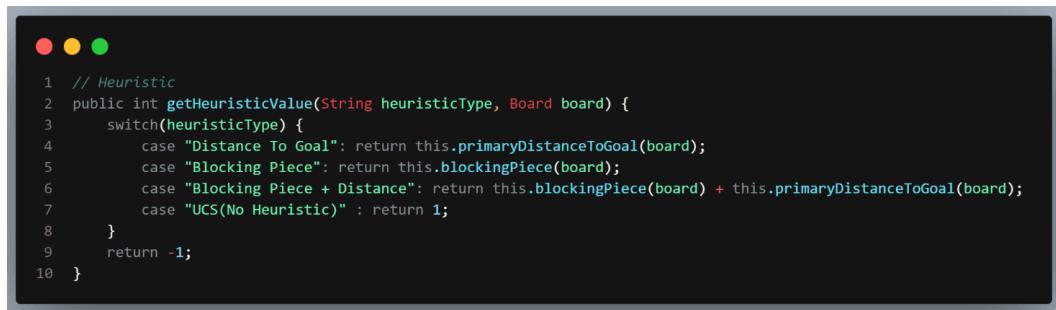
```
1 @FXML  
2 private void replayAnimation() {  
3     List<State> stateSteps = State.getStatePath(latestSolvedState);  
4     int delayInMS = Integer.parseInt(delayFrame.getText());  
5     final int[] index = {0};  
6  
7     PauseTransition pause = new PauseTransition(Duration.millis(delayInMS));  
8     pause.setOnFinished(event -> {  
9         if (index[0] < stateSteps.size()) {  
10             drawBoard(stateSteps.get(index[0]).getBoard().getGrid());  
11             index[0]++;
12             pause.playFromStart();
13         }
14     });
15
16     if (!stateSteps.isEmpty()) {
17         drawBoard(stateSteps.get(0).getBoard().getGrid());
18         index[0] = 1;
19         pause.play();
20     }
21 }
```

Gambar 3.4.1.9 replayAnimation

3.4.2 Heuristics

Heuristik adalah fungsi estimasi yang digunakan untuk menilai seberapa dekat suatu node terhadap tujuan. Heuristik biasanya dilambangkan sebagai $h(n)$. Tujuan dari heuristik adalah untuk membantu algoritma pencarian untuk lebih cepat menemukan solusi dengan memandu pencarian ke arah yang lebih menjanjikan.

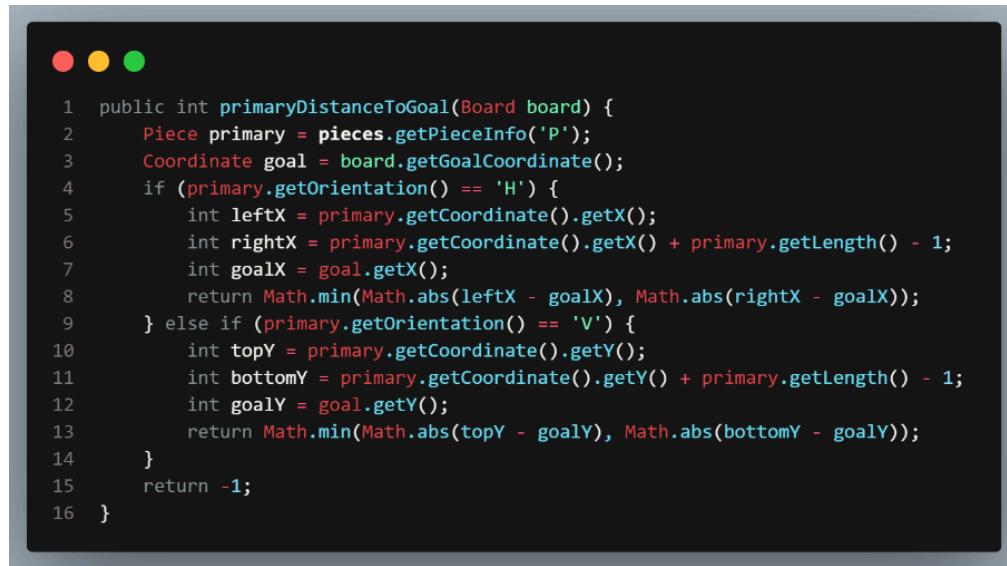
Pada program ini, terdapat 3 jenis heuristik, yaitu jarak ke tujuan, piece yang menghalangi, dan gabungan keduanya.



```
● ● ●
1 // Heuristic
2 public int getHeuristicValue(String heuristicType, Board board) {
3     switch(heuristicType) {
4         case "Distance To Goal": return this.primaryDistanceToGoal(board);
5         case "Blocking Piece": return this.blockingPiece(board);
6         case "Blocking Piece + Distance": return this.blockingPiece(board) + this.primaryDistanceToGoal(board);
7         case "UCS(No Heuristic)": return 1;
8     }
9     return -1;
10 }
```

Gambar 3.4.2.1 getHeuristicValue

Heuristik “Distance To Goal” adalah pendekatan dengan memeriksa jarak primary piece terhadap tujuan. Pendekatan ini menjadi fokus bagi solver algoritma.



```
● ● ●
1 public int primaryDistanceToGoal(Board board) {
2     Piece primary = pieces.getPieceInfo('P');
3     Coordinate goal = board.getGoalCoordinate();
4     if (primary.getOrientation() == 'H') {
5         int leftX = primary.getCoordinate().getX();
6         int rightX = primary.getCoordinate().getX() + primary.getLength() - 1;
7         int goalX = goal.getX();
8         return Math.min(Math.abs(leftX - goalX), Math.abs(rightX - goalX));
9     } else if (primary.getOrientation() == 'V') {
10        int topY = primary.getCoordinate().getY();
11        int bottomY = primary.getCoordinate().getY() + primary.getLength() - 1;
12        int goalY = goal.getY();
13        return Math.min(Math.abs(topY - goalY), Math.abs(bottomY - goalY));
14    }
15    return -1;
16 }
```

Gambar 3.4.2.2 primaryDistanceToGoal

Heuristik “Blocking Piece” adalah pendekatan dengan memeriksa jumlah piece yang menghalangi jalur primary piece ke tujuan. Pendekatan ini menjadi fokus bagi solver algoritma.



```
1 public int blockingPiece(Board board) {
2     Piece primary = pieces.getPieceInfo('P');
3     Coordinate goal = board.getGoalCoordinate();
4     int count = 0;
5
6     if (primary.getOrientation() == 'H') {
7         int row = primary.getCoordinate().getY();
8         int leftX = primary.getCoordinate().getX();
9         int rightX = leftX + primary.getLength() - 1;
10
11        if (goal.getX() == board.getWidth() - 1) {
12            for (int x = rightX + 1; x < board.getWidth() - 1; x++) {
13                char c = board.getCell(x, row);
14                if (c != '.' && c != '*') count++;
15            }
16        }
17        else if (goal.getX() == 0) {
18            for (int x = leftX - 1; x > 0; x--) {
19                char c = board.getCell(x, row);
20                if (c != '.' && c != '*') count++;
21            }
22        }
23    } else if (primary.getOrientation() == 'V') {
24        int col = primary.getCoordinate().getX();
25        int topY = primary.getCoordinate().getY();
26        int bottomY = topY + primary.getLength() - 1;
27
28        if (goal.getY() == board.getHeight() - 1) {
29            for (int y = bottomY + 1; y < board.getHeight() - 1; y++) {
30                char c = board.getCell(col, y);
31                if (c != '.' && c != '*') count++;
32            }
33        }
34        else if (goal.getY() == 0) {
35            for (int y = topY - 1; y > 0; y--) {
36                char c = board.getCell(col, y);
37                if (c != '.' && c != '*') count++;
38            }
39        }
40    }
41    return count;
42 }
```

Gambar 3.4.2.3 blockingPiece

Heuristik terakhir adalah gabungan dari kedua heuristik sebelumnya. Heuristik ini menjumlahkan nilai dari kedua heuristik.



```
1 public int getCompositeValue(Board board){
2     return this.blockingPiece(board) + this.primaryDistanceToGoal(board);
3 }
```

Gambar 3.4.2.3 getCompositeValue

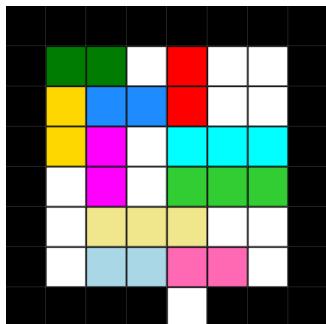
BAB IV

EKSPERIMENT

4.1 Uniform Cost Search

Test Case	Input	Output
1		<p>Rush Hour Solver</p> <p>Masukkan file input : <input type="button" value="Choose File"/></p> <p>Algoritma : <input type="button" value="Uniform Cost Sear..."/></p> <p>Heuristic : <input type="button" value="Distance To Goal"/></p> <p>Delay Frame (ms) : <input type="text" value="20"/></p> <p><input type="button" value="Solve"/></p> <p>File berhasil disimpan</p> <p>Moves: 26 Time: 353 ms Nodes: 1277 <input type="button" value="Replay"/></p> <p>Filename : <input type="text" value="UCS1"/> <input type="button" value="Save"/></p>

2



Rush Hour Solver

Masukkan file input :

Algoritma : Heuristic :

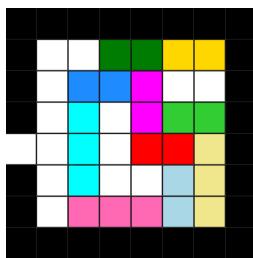
Delay Frame (ms) : Solve

Solved!

Moves: 20
Time: 3025 ms
Nodes: 11072

Filename :

3



Rush Hour Solver

Masukkan file input :

Algoritma : Heuristic :

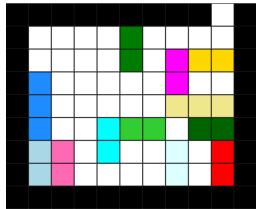
Delay Frame (ms) : Solve

Solved!

Moves: 24
Time: 309 ms
Nodes: 2919

Filename :

4



Rush Hour Solver

Masukkan file input :

Algoritma :

Heuristic :

Delay Frame (ms) :

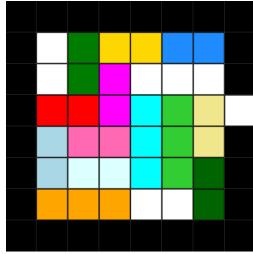
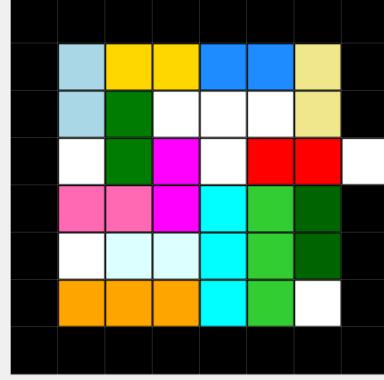
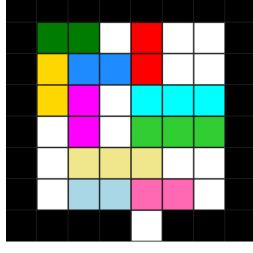
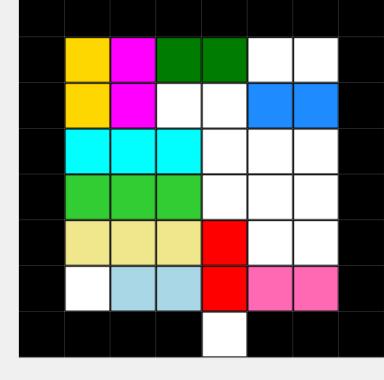
Solved!

Moves: 5
Time: 5421 ms
Nodes: 14920

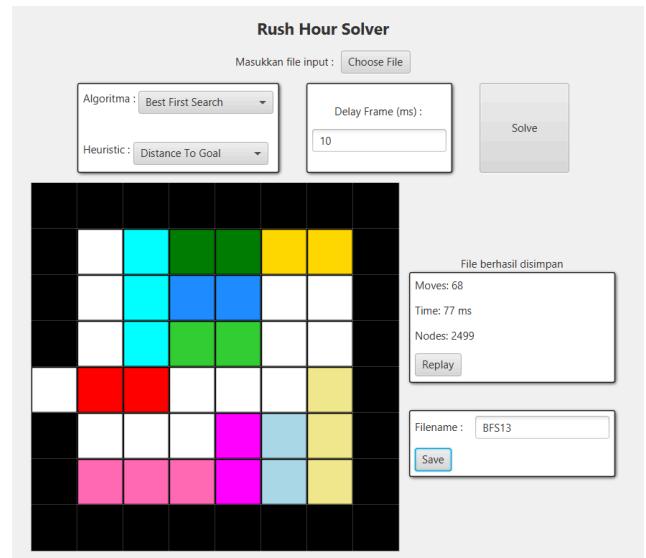
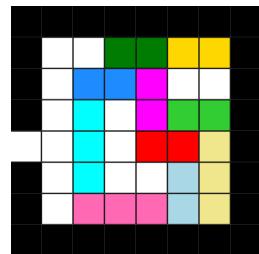
Filename :

The same 10x10 grid as the initial state, but solved. The blue car is now at the bottom-left, and all other cars have moved to form a path to the right. The grid includes black obstacle squares and empty white squares.

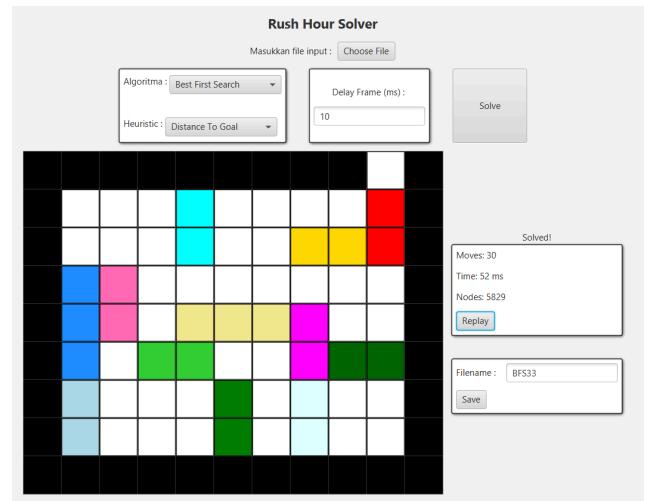
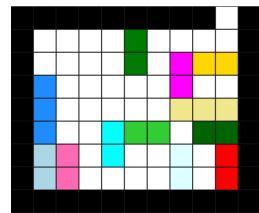
4.2 Best First Search

Test Case	Heuristik	Input	Output
1	Distance to Goal		<p>Rush Hour Solver</p> <p>Masukkan file input : <input type="button" value="Choose File"/></p> <p>Algoritma : <input type="button" value="Best First Search"/> Heuristic : <input type="button" value="Distance To Goal"/></p> <p>Delay Frame (ms) : <input type="text" value="10"/> Solve</p>  <p>Solved! Moves: 84 Time: 16 ms Nodes: 821 <input type="button" value="Replay"/></p> <p>Filename : <input type="text"/> <input type="button" value="Save"/></p>
2			<p>Rush Hour Solver</p> <p>Masukkan file input : <input type="button" value="Choose File"/></p> <p>Algoritma : <input type="button" value="Best First Search"/> Heuristic : <input type="button" value="Distance To Goal"/></p> <p>Delay Frame (ms) : <input type="text" value="10"/> Solve</p>  <p>Solved! Moves: 711 Time: 107 ms Nodes: 8853 <input type="button" value="Replay"/></p> <p>Filename : <input type="text" value="BFS12"/> <input type="button" value="Save"/></p>

3

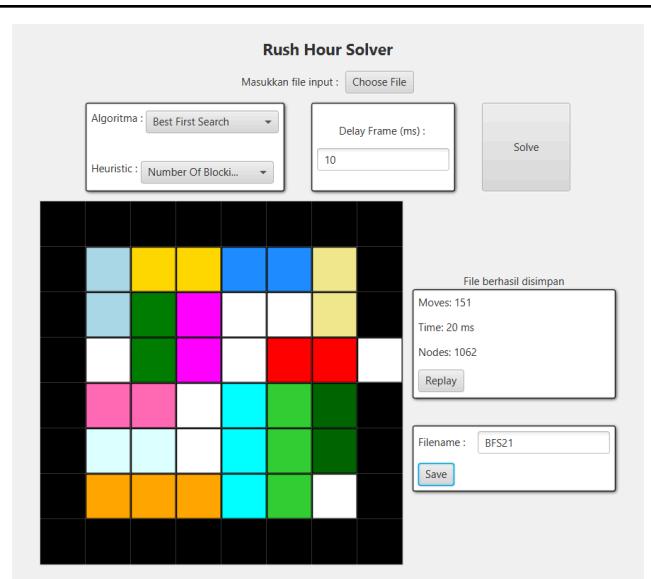
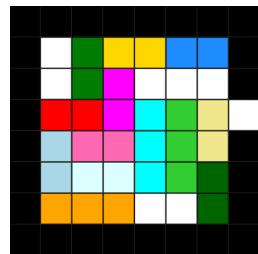


4

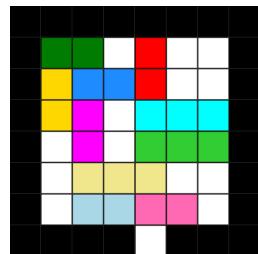


5

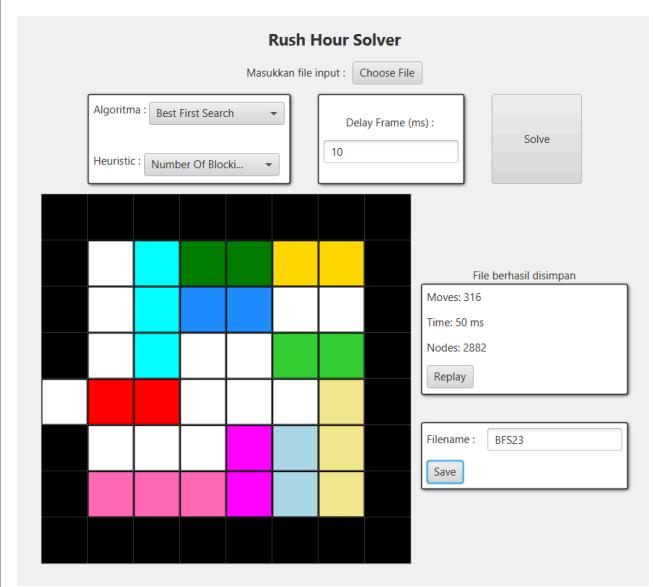
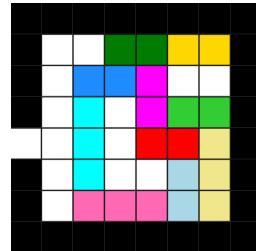
Number of
Blocking Piece



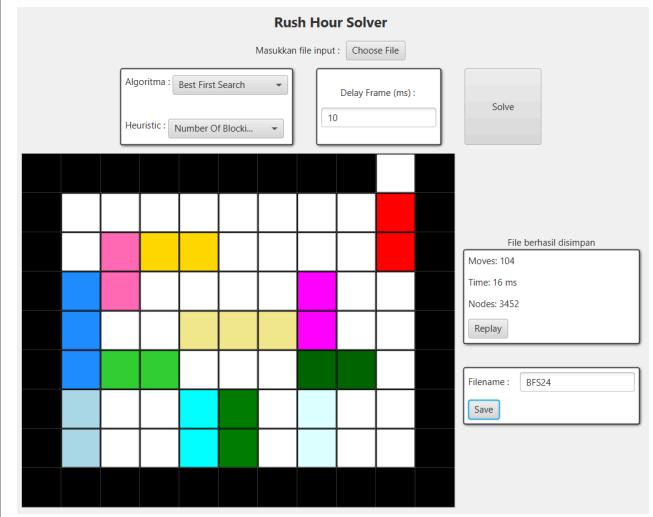
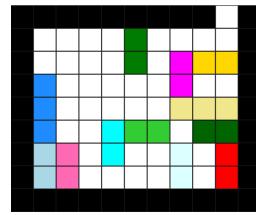
6



7

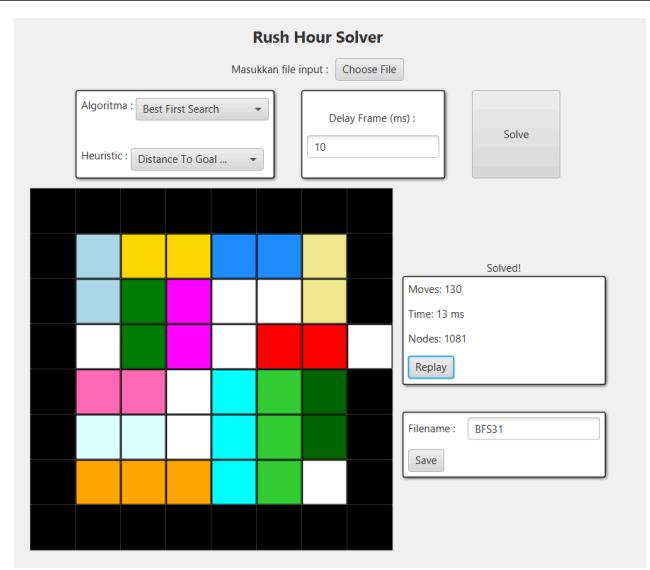
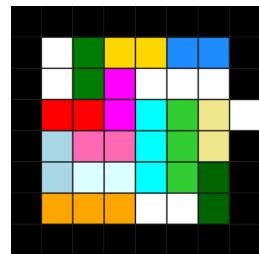


8

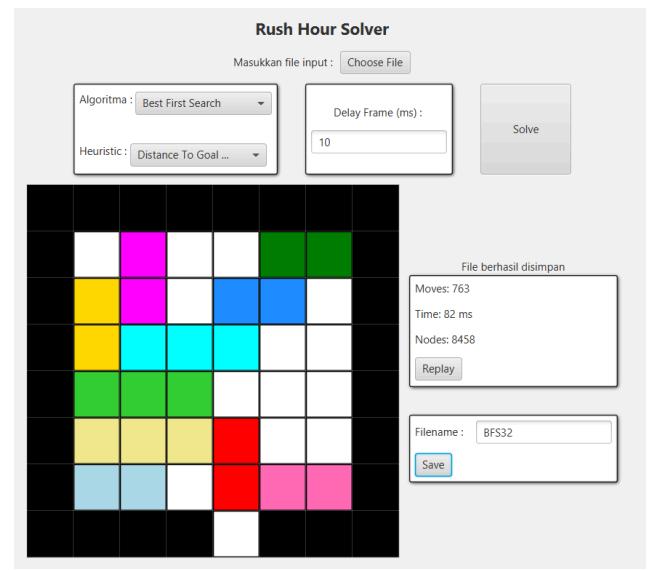
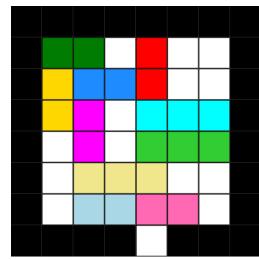


9

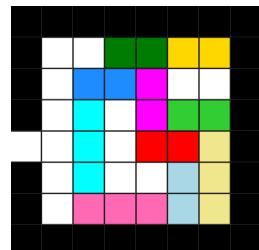
Distance to Goal
+ Number of
Blocking Piece
Combination



10



11



Rush Hour Solver

Masukkan file input :

Algoritma : Best First Search

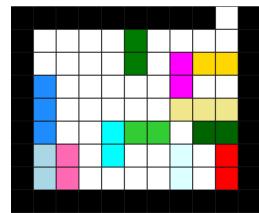
Delay Frame (ms) : 10

File berhasil disimpan

Moves: 291
Time: 32 ms
Nodes: 2652

Filename : BFS33

12



Rush Hour Solver

Masukkan file input :

Algoritma : Best First Search

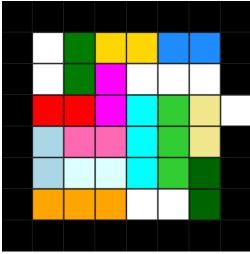
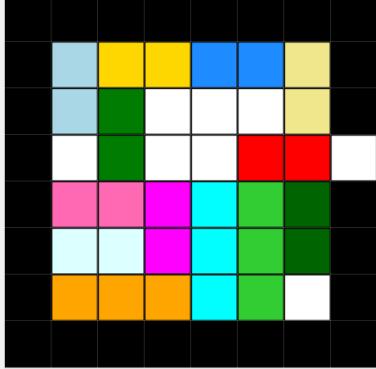
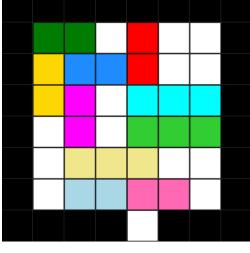
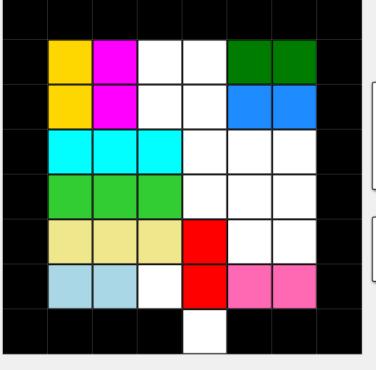
Delay Frame (ms) : 10

Solved!

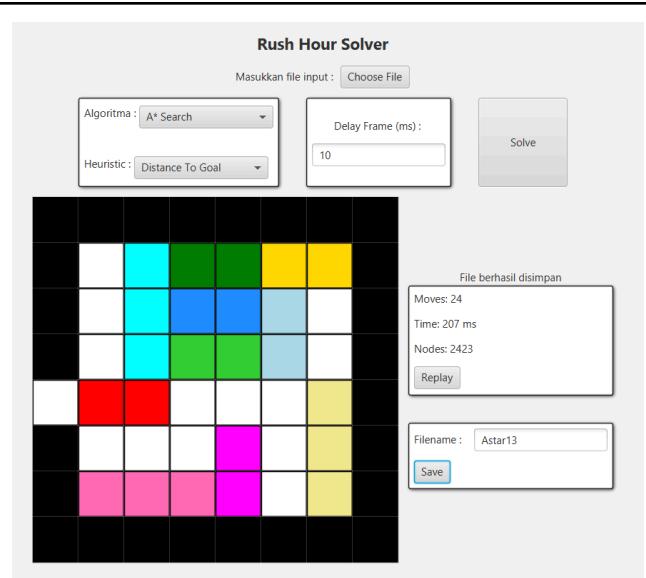
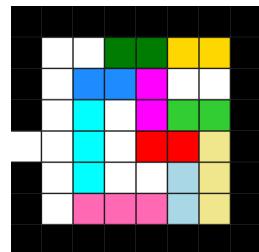
Moves: 66
Time: 10 ms
Nodes: 2229

Filename : BFS34

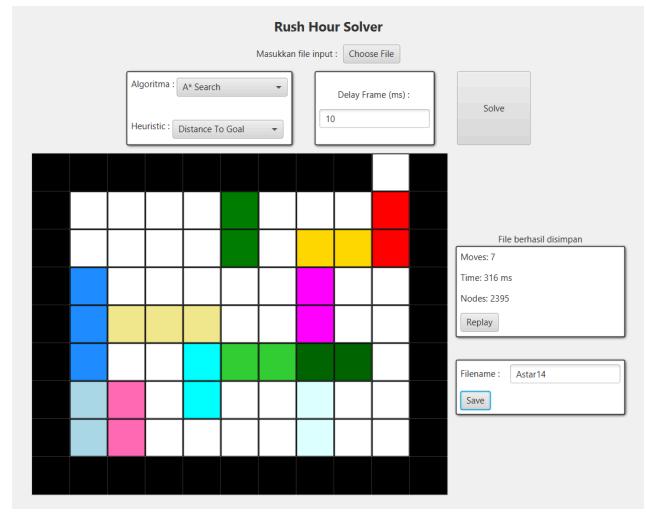
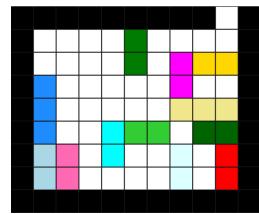
4.3 A* Search

Test Case	Heuristik	Input	Output
1	Distance to Goal		<p>Rush Hour Solver</p> <p>Masukkan file input : <input type="button" value="Choose File"/></p> <p>Algoritma : A* Search <input type="button" value="▼"/></p> <p>Heuristic : Distance To Goal <input type="button" value="▼"/></p> <p>Delay Frame (ms) : <input type="text" value="10"/> <input type="button" value="Solve"/></p>  <p>Solved!</p> <p>Moves: 26 Time: 119 ms Nodes: 1137 <input type="button" value="Replay"/></p> <p>Filename : <input type="text" value="Astar11"/> <input type="button" value="Save"/></p>
2			<p>Rush Hour Solver</p> <p>Masukkan file input : <input type="button" value="Choose File"/></p> <p>Algoritma : A* Search <input type="button" value="▼"/></p> <p>Heuristic : Distance To Goal <input type="button" value="▼"/></p> <p>Delay Frame (ms) : <input type="text" value="10"/> <input type="button" value="Solve"/></p>  <p>File berhasil disimpan</p> <p>Moves: 21 Time: 2274 ms Nodes: 10012 <input type="button" value="Replay"/></p> <p>Filename : <input type="text" value="Astar12"/> <input type="button" value="Save"/></p>

3

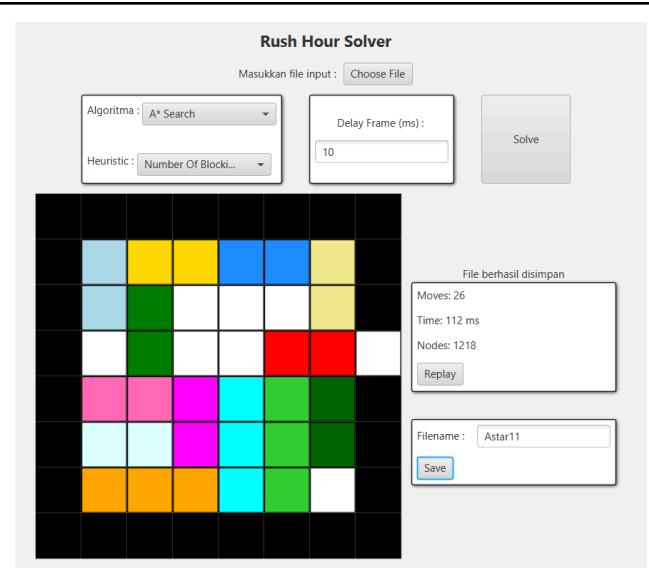
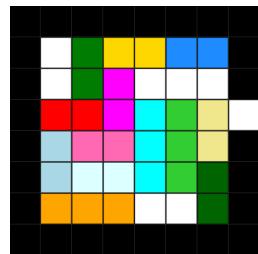


4

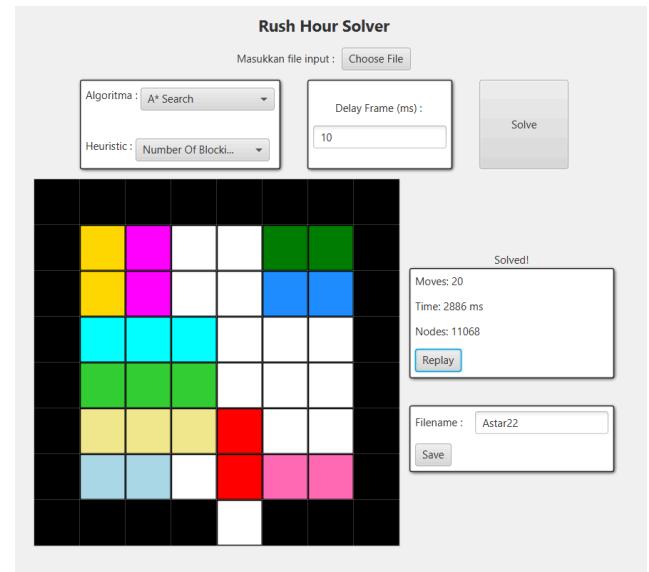
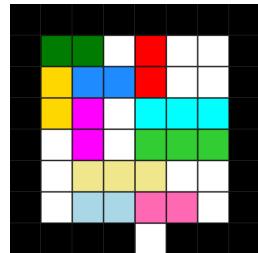


5

Number of
Blocking Piece

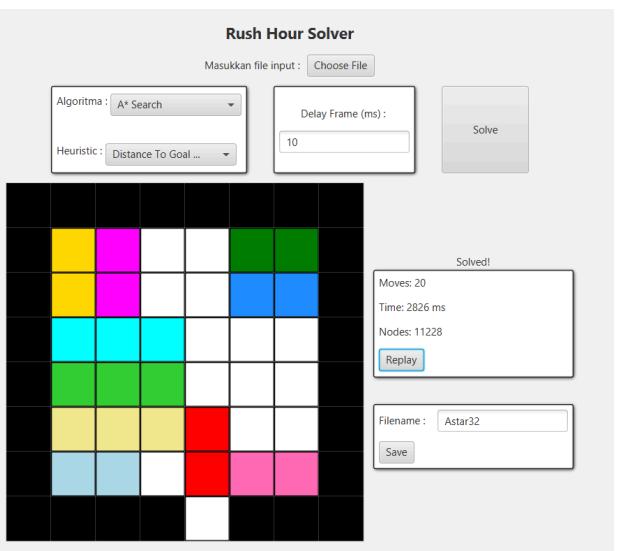
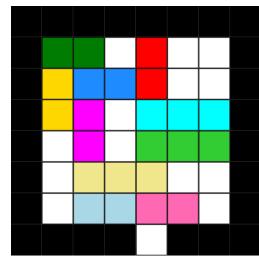


6

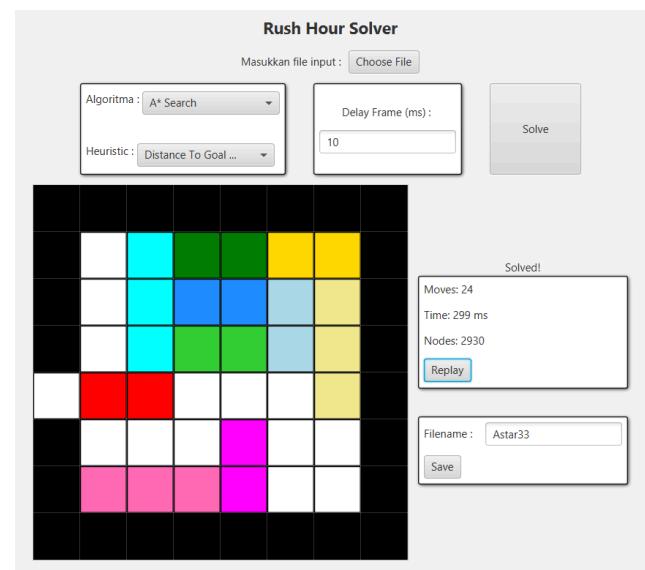
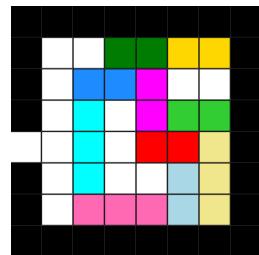


7		<p>Rush Hour Solver</p> <p>Masukkan file input : <input type="button" value="Choose File"/></p> <p>Algoritma : A* Search Heuristic : Number Of Block...</p> <p>Delay Frame (ms) : 10</p> <p>Solve</p> <p>File berhasil disimpan</p> <p>Moves: 24 Time: 267 ms Nodes: 2915</p> <p>Replay</p> <p>Filename : Astar23 Save</p>	
8		<p>Rush Hour Solver</p> <p>Masukkan file input : <input type="button" value="Choose File"/></p> <p>Algoritma : A* Search Heuristic : Number Of Block...</p> <p>Delay Frame (ms) : 10</p> <p>Solve</p> <p>File berhasil disimpan</p> <p>Moves: 5 Time: 4304 ms Nodes: 12621</p> <p>Replay</p> <p>Filename : Astar24 Save</p>	
9	<p>Distance to Goal + Number of Blocking Piece Combination</p>		<p>Rush Hour Solver</p> <p>Masukkan file input : <input type="button" value="Choose File"/></p> <p>Algoritma : A* Search Heuristic : Distance To Goal ...</p> <p>Delay Frame (ms) : 10</p> <p>Solve</p> <p>File berhasil disimpan</p> <p>Moves: 26 Time: 120 ms Nodes: 1271</p> <p>Replay</p> <p>Filename : Astar31 Save</p>

10



11



12		
----	--	--

Dibawah ini merupakan tabel hasil test case yang telah kami lakukan pada penyelesaian puzzle rush hours menggunakan 3 algoritma dari 3 heuristik yang berbeda yaitu UCS, A*, dan GBFS (pada UCS tidak memiliki Heuristik menjadi alasan ke-3 baris pada heuristik digabung menjadi 1).

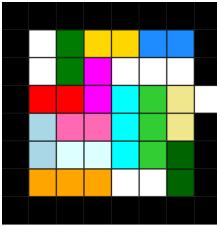
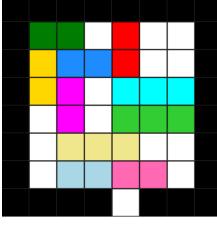
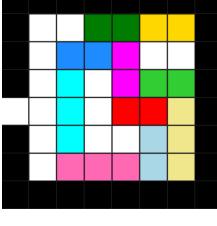
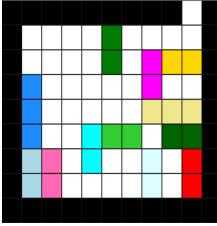
- **Evaluasi Hasil Test Case**

Berdasarkan tabel yang telah kami buat dibawah ini, UCS merupakan algoritma dengan pendekatan solusi moves terpendek dibandingkan dengan seluruh hasil percobaan GBFS dan A* dengan heuristik apapun. Tetapi memiliki satu kelemahan , yaitu untuk mencari solusi dengan moves terpendek UCS harus menjelajahi node lebih banyak dibandingkan A* dan GBFS dengan rata-rata 50 - 60 % lebih banyak node yang membuat UCS menjadi algoritma dengan waktu eksekusi algoritma terlama dibanding keduanya.

Disisi lain, GBFS memiliki kelebihan dibandingkan UCS dan A*, dimana waktu eksekusi untuk menemukan solusi adalah yang paling cepat dengan perbedaan waktu eksekusi diantara yang lain memiliki selisih yang sangat signifikan, tetapi memiliki solusi moves yang paling tidak efisien / paling banyak diantara yang lain. Hal ini yang membuat GBFS menjadi algoritma pendekatan yang bisa dibilang tidak optimal, karena hanya mengandalkan $h(n)$ next node sebagai parameter pembanding pengurutan. GBFS hanya melihat solusi suboptimal yang membuatnya tidak efisien (akan dijelaskan lebih detail pada bab 5).

Terakhir, pendekatan algoritma A* menunjukkan hasil yang paling seimbang: menghasilkan solusi dengan jumlah langkah paling sedikit setelah UCS, namun dengan jumlah node yang dijelajahi lebih sedikit dibanding UCS. Hal ini menjadikan A* sebagai pendekatan paling efisien dan unggul dibanding keduanya. Pada kasus ini, A* bekerja secara optimal dan efisien berkat heuristik yang admissible (penjelasan lebih lanjut terdapat pada Bab 5).

Tabel 1 Perbandingan hasil dari test case dari heuristik yang berbeda (UCS tidak memiliki heuristik)

Input		Algoritma		
Input	Heuristic	UCS	GBFS	A*
	<i>Distance to goal</i>	<i>moves: 26</i> <i>time: 353 ms</i> <i>nodes: 1277</i>	<i>moves: 84</i> <i>time: 16 ms</i> <i>nodes: 821</i>	<i>moves: 26</i> <i>time: 119 ms</i> <i>nodes: 1137</i>
	<i>Blocking Piece</i>		<i>moves: 151</i> <i>time: 20 ms</i> <i>nodes: 1062</i>	<i>moves: 26</i> <i>time: 112 ms</i> <i>nodes: 1218</i>
	<i>Composite</i>		<i>moves: 130</i> <i>time: 13 ms</i> <i>nodes: 1081</i>	<i>moves: 26</i> <i>time: 120 ms</i> <i>nodes: 1271</i>
	<i>Distance to goal</i>	<i>moves: 20</i> <i>time: 3025 ms</i> <i>nodes: 11072</i>	<i>moves: 711</i> <i>time: 107 ms</i> <i>nodes: 8853</i>	<i>moves: 21</i> <i>time: 2274 ms</i> <i>nodes: 10012</i>
	<i>Blocking Piece</i>		<i>moves: 360</i> <i>time: 43 ms</i> <i>nodes: 4764</i>	<i>moves: 20</i> <i>time: 2886 ms</i> <i>nodes: 11068</i>
	<i>Composite</i>		<i>moves: 763</i> <i>time: 82 ms</i> <i>nodes: 8458</i>	<i>moves: 20</i> <i>time: 2826 ms</i> <i>nodes: 11228</i>
	<i>Distance to goal</i>	<i>moves: 24</i> <i>time: 309 ms</i> <i>nodes: 2919</i>	<i>moves: 68</i> <i>time: 77 ms</i> <i>nodes: 2499</i>	<i>moves: 24</i> <i>time: 207 ms</i> <i>nodes: 2423</i>
	<i>Blocking Piece</i>		<i>moves: 316</i> <i>time: 50 ms</i> <i>nodes: 2882</i>	<i>moves: 24</i> <i>time: 267 ms</i> <i>nodes: 2915</i>
	<i>Composite</i>		<i>moves: 291</i> <i>time: 32 ms</i> <i>nodes: 2652</i>	<i>moves: 24</i> <i>time: 299 ms</i> <i>nodes: 2930</i>
	<i>Distance to goal</i>	<i>moves: 5</i> <i>time: 5421 ms</i> <i>nodes: 14920</i>	<i>moves: 30</i> <i>time: 52 ms</i> <i>nodes: 5829</i>	<i>moves: 7</i> <i>time: 316 ms</i> <i>nodes: 2395</i>
	<i>Blocking Piece</i>		<i>moves: 104</i> <i>time: 16 ms</i> <i>nodes: 3452</i>	<i>moves: 5</i> <i>time: 4304 ms</i> <i>nodes: 12621</i>
	<i>Composite</i>		<i>moves: 66</i> <i>time: 10 ms</i> <i>nodes: 2229</i>	<i>moves: 5</i> <i>time: 4168 ms</i> <i>nodes: 13008</i>

- **Evaluasi Kuantitatif Heuristik**

Untuk mengevaluasi performa tiga heuristik yang digunakan dalam GBFS dan A* (Distance to Goal, Blocking Piece, dan Composite), kami menganalisis data eksperimen dari Tabel 1 (Bab IV) berdasarkan tiga metrik: jumlah langkah solusi (moves), waktu eksekusi, dan jumlah node yang dikunjungi. Berikut adalah analisis kuantitatif untuk masing-masing heuristik: 2

- **Distance to Goal:** Heuristik ini mengukur jarak primary piece ke pintu keluar (dalam jumlah sel). Heuristik ini admissible karena tidak pernah melebih-lebihkan jumlah langkah minimum yang diperlukan, mengingat primary piece hanya dapat bergerak lurus sesuai orientasinya. Pada test case di Bab IV, heuristik ini menghasilkan performa yang cukup baik untuk A*, dengan jumlah langkah mendekati optimal (misalnya, 7 langkah pada test case 4) dan jumlah node yang relatif kecil (239510012 node). Namun, pada GBFS, heuristik ini sering menghasilkan solusi yang jauh dari optimal (misalnya, 711 langkah pada test case 2) karena hanya fokus pada jarak tanpa mempertimbangkan biaya jalur sebelumnya.

- **Blocking Piece:** Heuristik ini menghitung jumlah piece yang menghalangi jalur primary piece ke pintu keluar. Heuristik ini juga admissible karena jumlah piece yang menghalangi selalu lebih kecil atau sama dengan jumlah gerakan minimum yang diperlukan untuk membersihkan jalur. Dalam eksperimen, heuristik ini cenderung menghasilkan lebih banyak node yang dikunjungi pada A* (misalnya, 12621 node pada test case 4) dibandingkan Distance to Goal, tetapi masih lebih efisien daripada UCS. Pada GBFS, heuristik ini menghasilkan solusi yang sangat tidak optimal (misalnya, 360 langkah pada test case 2) karena fokus pada menghilangkan piece yang menghalangi tanpa mempertimbangkan efisiensi gerakan.

- **Composite (Distance to Goal + Blocking Piece):** Heuristik ini menjumlahkan nilai dari Distance to Goal dan Blocking Piece. Meskipun tetap admissible (karena penjumlahan dua heuristik admissible masih admissible selama tidak melebihi biaya sebenarnya), heuristik ini tidak selalu lebih baik daripada kedua heuristik lainnya. Pada A*, performanya mirip dengan Blocking Piece (misalnya, 13008 node pada test case 4), tetapi pada GBFS, heuristik ini menghasilkan solusi yang sangat tidak optimal (misalnya, 763 langkah pada test case 2). Hal ini menunjukkan bahwa kombinasi heuristik tidak selalu meningkatkan efisiensi, terutama pada GBFS yang hanya bergantung pada $h(n)$.

Tabel 2 Perbandingan Performa Heuristik pada A* (Rata-rata dari Test Case 14)

Heuristik	Rata-rata Moves	Rata-rata Waktu (ms)	Rata-rata Node
Distance to Goal	19.5	978.5	5991.75
Blocking Piece	18.75	1880.25	7955.5
Composite	18.75	1873.25	8109.25

Heuristik Distance to Goal menunjukkan performa terbaik pada A* dengan jumlah node yang dikunjungi paling sedikit (rata-rata 5991.75 node) dan waktu eksekusi tercepat (rata-rata 978.5 ms). Blocking Piece dan Composite cenderung menghasilkan lebih banyak node yang dikunjungi karena fokus pada piece yang menghalangi dapat mengarahkan pencarian ke jalur yang kurang efisien. Pada GBFS, ketiga heuristik menghasilkan solusi yang jauh dari optimal, dengan Composite sering kali menghasilkan jumlah langkah terbanyak (rata-rata 312.5 langkah dibandingkan 18.7519.5 untuk A*). Oleh karena itu, Distance to Goal adalah heuristik paling efektif untuk A* dalam konteks Rush Hour, sementara Blocking Piece dan Composite lebih cocok untuk kasus tertentu dengan banyak piece penghalang.

BAB V

ANALISIS ALGORITMA

5.1 Definisi $f(n)$ dan $g(n)$

Dalam konteks algoritma pencarian jalur seperti UCS, Greedy Best First Search, dan A*, fungsi $g(n)$ merepresentasikan biaya sebenarnya yang telah dikeluarkan untuk mencapai node n dari node awal. Misalnya, pada permainan Rush Hour, $g(n)$ adalah jumlah langkah atau gerakan yang sudah dilakukan dari posisi awal sampai konfigurasi papan saat ini. Sedangkan $f(n)$ adalah estimasi total biaya untuk mencapai tujuan melalui node n , yang didefinisikan sebagai penjumlahan antara $g(n)$ dan $h(n)$, di mana $h(n)$ adalah nilai heuristic yang mengestimasi biaya tersisa dari node n menuju solusi akhir.

5.2 Admissibility Heuristik A* pada Puzzle Rush Hour

Heuristic yang admissible adalah heuristic yang tidak pernah melebih-lebihkan biaya sebenarnya untuk mencapai tujuan dari suatu node ,memiliki cost prediction $h(n)$ yang optimis tetapi realistik. Dalam permainan Rush Hour, heuristic admissible dapat berupa estimasi jarak minimal yang diperlukan oleh primary piece untuk mencapai pintu keluar dan banyak piece yang menghalangi primary piece untuk mencapai pintu keluar . Heuristic ini selalu lebih kecil atau sama dengan biaya sebenarnya yang dibutuhkan, sehingga menjamin bahwa algoritma A* tidak melewatkkan solusi optimal.

5.3 Kesamaan UCS dan BFS pada Penyelesaian Puzzle Rush Hour

UCS dan BFS pada kasus ini (rush hour), adalah sama setidaknya untuk implementasi yang kami lakukan. Secara konsep BFS merupakan eksekusi node satu persatu setiap level setelah level tertentu sudah di cek keseluruhan nodenya, dalam kasus ini , UCS melakukan hal yang sama dengan BFS tetapi dengan perbedaan sorting berdasarkan past cost yang telah dilewati dari genesis node sampai node tertentu. Perhitungan past cost implementasi pada program kami , memperhitungkan jumlah move yang sudah dilakukan hingga node tertentu, sehingga secara tidak langsung, sorting yang dilakukan oleh UCS merupakan sama saja dengan BFS secara urutan node yang diperiksa sudah pasti terurut secara otomatis.

5.4 Keefisienan A* dibandingkan UCS dalam Rush Hour

A* biasanya lebih efisien dibandingkan UCS dalam menyelesaikan Rush Hour karena A* menggunakan fungsi evaluasi $f(n) = g(n) + h(n)$, di mana $g(n)$ adalah jumlah move yang dilakukan piece apapun hingga node n dan $h(n)$ adalah heuristic yang memperkirakan biaya dari node n ke exit area. Heuristic yang baik (admissible dan konsisten) membantu A* memprioritaskan eksplorasi node yang lebih dekat ke solusi, sehingga mengurangi jumlah node yang perlu diperiksa dibanding UCS yang hanya mengandalkan biaya jalur tanpa arahan ke tujuan. Dengan demikian, A* dapat menemukan solusi lebih cepat dan dengan eksplorasi yang lebih sedikit dibanding UCS pada puzzle Rush Hour. Hal ini diperkuat dengan beberapa

eksperimen yang kami lakukan , yang menunjukkan bahwa A* memiliki kecepatan algoritma dan total step yang lumayan lebih cepat dan minim dibandingkan dengan UCS.

5.5 Kekurang-optimalan GBFS dalam pencarian solusi pada Rush Hour

Greedy Best First Search (GBFS) tidak menjamin solusi optimal pada puzzle Rush Hour. GBFS menggunakan heuristik untuk memilih node berikutnya yang dianggap paling dekat dengan tujuan berdasarkan estimasi heuristik $h(n)$, tanpa mempertimbangkan biaya $g(n)$ yang sudah dikeluarkan untuk mencapai node tersebut. Karena fokusnya yang hanya kepada heuristik, GBFS bisa memiliki jalur yang terlihat cepat menuju tujuan, namun sebenarnya lebih mahal atau tidak optimal untuk kasus ini. GBFS menemukan solusi suboptimal tetapi tidak optimal secara keseluruhan. Berbeda dengan GBFS, algoritma pathfinding seperti A* dan UCS menjamin solusi optimal dalam Rush Hour selama heuristik pada A* adalah admissible dan UCS memperhitungkan biaya jalur yang ditempuh (past cost) secara nyata. A* menggabungkan heuristik dan biaya yang telah dilewati untuk efisiensi sekaligus memastikan optimalitas, sedangkan UCS menjamin solusi optimal berdasarkan biaya minimal tanpa menggunakan heuristik.

5.6 Cara Kerja Algoritma Pathfinding pada Rush Hour

1. UCS

```
function UCS(initial_state):
    visited ← empty set
    priority_queue ← empty priority queue ordered by past_cost

    initial_state.past_cost ← 0
    initial_state.total_cost ← 0
    initial_state.next_cost ← 0

    enqueue priority_queue with initial_state

    while priority_queue is not empty:
        current ← dequeue priority_queue

        if current is goal:
            return current

        state_key ← current.board_state_string
        if state_key in visited:
            continue
        add state_key to visited

        for each next_state in current.generate_next_states(visited, "UCS (No
Heuristic)"):
            if next_state.board_state_string not in visited:
                next_state.next_cost ← current.primary_distance_to_goal()
                next_state.total_cost ← next_state.past_cost
                enqueue priority_queue with next_state

    return null
```

- Buat visited sebagai HashSet kosong
- Buat priority_queue yang mengurutkan berdasarkan past_cost ($g(n)$).
- Set nilai awal past_cost = 0, total_cost = 0, next_cost = 0 pada initial state, pada kasus ini past_cost = 0 karena tidak ada node sebelum initial, next cost dan total cost dengan value = 0 , karena tidak terpakai untuk UCS.
- Iterate while priority_queue tidak kosong
- Ambil past_cost terkecil dari priority queue, iterate ulang
- Jika state sudah goal atau jarak primary ke goal adalah 1 (dihitung dari jarak terdekat), return node dimana solusi ditemukan.
- Tambahkan pada visited hashSet jika suatu state sudah diproses agar tidak terjadi pengecekan ulang pada state yang sama
- Generate_next_states yaitu men generate list of child node yang menghasilkan semua kemungkinan konfigurasi papan berikutnya , dilakukan dengan mengiterasikan semua piece pada papan, dan memasukan semua kemungkinan 1 piece dapat bergerak ke arah tertentu sebagai child. (pada proses ini sudah di set past cost sebagai (parent pastcost+1), dengan 1 adalah move yang dilakukan ;
- Masukan ke priority queue dan mengulangi proses selama goal tidak ditemukan atau semua node belum dikunjungi.

2. Greedy Best First Search

```
function BestFirstSearch(start_state, heuristic_type):
    visited ← empty set
    priority_queue ← empty priority queue ordered by next_cost (h(n))

    start_state.past_cost ← 0
    start_state.next_cost ← heuristic_value(start_state, heuristic_type)
    start_state.total_cost ← 0

    enqueue priority_queue with start_state
    add start_state.board_state_string to visited

    while priority_queue is not empty:
        current ← dequeue priority_queue

        if current is goal:
            return current

        next_states ← generate_next_states(current, visited, heuristic_type)
        for each next_state in next_states:
            if next_state.board_state_string not in visited:
                add next_state.board_state_string to visited
                enqueue priority_queue with next_state

    return null
```

- Buat visited sebagai HashSet kosong & priority_queue yang mengurutkan berdasarkan next_cost ($h(n)$).
- Set nilai awal untuk start state , pastcost = 0, next_cost sebagai $h(n)$ dimana $h(n)$ merupakan heuristik pilihan user, dan totalCost yang tidak digunakan.
- Start dimasukkan ke dalam priority_queue dan visited.
- Loop selama priority_queue tidak kosong.
- Ambil node dengan next_cost ($h(n)$) terkecil dari antrian.
- Return jika goal (jarak ke goal adalah 1 terhitung dari bagian piece terdekat).
- Generate next state dari current state, dimana pada fungsi ini sudah menginisialisasi past_cost sebagai past_cost+1 (tetapi tidak digunakan pada algoritma ini), dan menginisialisasikan next_cost sebagai value dari heuristik yang digunakan, begitu pula dengan totalCost (tidak digunakan pada algoritma ini).
- Cek apakah hasil state belum pernah dikunjungi
- Ulangi sampai menemukan goal atau queue kosong (tidak ada solusi).

3. A*

```
function AStar(initial_state, heuristic_type):
    visited ← empty set
    priority_queue ← empty priority queue ordered by total_cost (f(n) = g(n) +
    h(n))

    initial_state.past_cost ← 0
    initial_state.next_cost ← heuristic_value(initial_state, heuristic_type)
    initial_state.total_cost ← initial_state.past_cost + initial_state.next_cost

    enqueue priority_queue with initial_state

    while priority_queue is not empty:
        current ← dequeue priority_queue

        if current is goal:
            add current.board_state_string to visited
            return current

        state_key ← current.board_state_string
        if state_key in visited:
            continue
        add state_key to visited

        for each next_state in generate_next_states(current, visited,
heuristic_type):
            if next_state.board_state_string not in visited:
                enqueue priority_queue with next_state

    return null
```

- Buat visited sebagai HashSet kosong dan priority_queue yang mengurutkan berdasarkan total_cost dimana pada AStar menggunakan nilai total $f(n) = g(n) + h(n)$.
- Inisialisasi nilai awal initial_state past_cost dengan value 0 , next cost dengan nilai heuristic input, dan total_cost sebagai past_cost + next_cost.
- Tambahkan initial-state ke priority_queue
- Lakukan iterasi selama priority_queue tidak kosong dengan pengurutan berdasarkan total_cost
- Mengambil current dengan total_cost terkecil dari priority_queue
- Jika current adalah goal , return sebagai node solusi yang nantinya akan di backtrack menggunakan this.parent sampai null
- Pada setiap iterasi akan dibandingkan state board node tertentu dengan HashSet agar tidak mengulangi board state yang sama (bertipe String).
- Generate semua kemungkinan node dengan mengkombinasikan banyak piece yang dapat bergerak ke arah tertentu sebagai total jumlah next state graph, sambil menginisialisasikan past_cost sebagai = parent.past_cost + 1, constant 1 sebagai move yang sudah digerakkan (increment by 1), next_cost = heuristic dari next_state ke goal, dan total_cost = past+next.
- Return null jika tidak ada solusi

5.7 Kompleksitas Algoritma

Dalam konteks permainan Rush Hour, kompleksitas algoritma Uniform Cost Search (UCS), Greedy Best First Search (GBFS), dan A* dipengaruhi oleh dua faktor utama: *branching factor* (rata-rata jumlah gerakan yang mungkin dari suatu state) dan kedalaman solusi (jumlah langkah minimum untuk mencapai tujuan). Untuk papan Rush Hour berukuran 6x6 dengan 12 piece, branching factor rata-rata dapat diestimasi berdasarkan jumlah piece dan gerakan yang valid (maju atau mundur sesuai orientasi). Misalnya, setiap piece dapat memiliki hingga 2 gerakan (maju atau mundur), dengan kemungkinan beberapa langkah jika ada ruang kosong. Berdasarkan eksperimen, branching factor rata-rata untuk Rush Hour dengan 12 piece berkisar antara 10 hingga 20, tergantung pada kepadatan papan. Kedalaman solusi (d) bervariasi tergantung pada kompleksitas papan, tetapi berdasarkan test case pada Bab IV, solusi optimal biasanya memerlukan 5 hingga 26 langkah.

5.7.1 Uniform Cost Search (UCS)

Time Complexity: UCS menjelajahi semua node berdasarkan urutan past cost $g(n)$ terkecil, yang dalam Rush Hour dihitung sebagai jumlah gerakan yang telah dilakukan. Dalam kasus terburuk, UCS akan memeriksa semua kemungkinan konfigurasi papan hingga menemukan solusi terdangkal (dengan kedalaman d). Dengan branching factor b , jumlah node maksimum yang dieksplorasi adalah:

$$T(n) = O(b^d)$$

Untuk Rush Hour 6x6 dengan $b \approx 15$ (rata-rata) dan $d \approx 20$ (berdasarkan test case di Bab IV), kompleksitas waktu dapat mencapai 1520, meskipun dalam praktiknya hanya sebagian kecil node yang dieksplorasi karena UCS menghentikan pencarian saat solusi ditemukan. Namun, karena UCS tidak menggunakan heuristik, ia cenderung menjelajahi lebih banyak node dibandingkan A*.

Space Complexity: UCS menyimpan semua node yang telah dikunjungi dalam HashSet (untuk menghindari duplikasi) dan node dalam antrian prioritas. Dalam kasus terburuk, jumlah node yang disimpan juga sebanding dengan:

$$O(b^d)$$

Ini menjadikan UCS kurang efisien untuk papan dengan branching factor besar atau solusi yang dalam, seperti terlihat pada test case dengan 14920 node yang dikunjungi (Tabel 1, Bab IV).

5.7.2 Greedy Best First Search (GBFS)

Time Complexity: GBFS hanya mempertimbangkan nilai heuristik $h(n)$, yang mengestimasi jarak ke tujuan (misalnya, jarak primary piece ke pintu keluar atau jumlah piece yang menghalangi). Dalam kasus terburuk, GBFS dapat menjelajahi jalur yang salah hingga kedalaman maksimum m , yang dalam Rush Hour bisa jauh lebih besar dari kedalaman solusi optimal 1 ($m \gg d$). Misalnya, jika heuristik memprioritaskan gerakan yang tampak menjanjikan tetapi tidak optimal, GBFS dapat menjelajahi hingga seluruh ruang state, dengan kompleksitas:

$$T(n) = O(b^m)$$

Dalam Rush Hour, m bisa sangat besar (misalnya, ratusan langkah untuk konfigurasi kompleks), tetapi berdasarkan eksperimen (Tabel 1, Bab IV), GBFS cenderung cepat menemukan solusi (10107 ms) karena fokus pada heuristik, meskipun solusinya sering kali tidak optimal (misalnya, 763 langkah pada test case 2 dengan heuristik Composite).

Space Complexity: Sama seperti UCS, GBFS menyimpan node dalam antrian prioritas dan HashSet untuk state yang telah dikunjungi, sehingga:

$$O(b^m)$$

Namun, karena GBFS sering kali menjelajahi lebih sedikit node dibandingkan UCS (misalnya, 22298853 node pada test case di Bab IV), kebutuhan memori biasanya lebih kecil. Optimalitas: GBFS tidak menjamin solusi optimal karena hanya bergantung pada $h(n)$ dan mengabaikan $g(n)$. Hal ini terlihat pada test case di Bab IV, di mana GBFS menghasilkan solusi dengan jumlah langkah jauh lebih banyak (misalnya, 711 langkah vs. 20 langkah untuk A* pada test case 2).

5.7.3 A*

Time Complexity: A* menggunakan fungsi evaluasi $f(n) = g(n) + h(n)$, yang menggabungkan biaya aktual $g(n)$ dan estimasi heuristik $h(n)$. Dengan heuristik yang admissible dan consistent, A* hanya menjelajahi node yang kemungkinan besar berada di jalur optimal, sehingga kompleksitas waktu dalam kasus terburuk adalah:

$$T(n) = O(b^d)$$

Namun, dalam praktik, A* jauh lebih efisien daripada UCS karena heuristik mengurangi jumlah node yang dieksplorasi. Misalnya, pada test case di Bab IV, A* hanya menjelajahi 239513008 node dibandingkan 127714920 node untuk UCS, dengan waktu eksekusi yang lebih cepat (1194304 ms vs. 3095421 ms untuk UCS).

Space Complexity: Sama seperti UCS, A* menyimpan node dalam antrian prioritas dan HashSet untuk state yang dikunjungi, sehingga:

$$O(b^d)$$

Meskipun demikian, heuristik yang baik mengurangi jumlah node yang perlu disimpan, seperti terlihat pada test case dengan 2395 node untuk A* dibandingkan 14920 node untuk UCS (test case 4).

BAB VI

PENUTUP

6.1 Kesimpulan

A* pada penyelesaian permainan rush hours merupakan pendekatan yang paling optimal dari segi jumlah node yang ditelusuri dan moves hasil solusi yang paling minim setelah UCS. Meskipun UCS memiliki solusi dengan moves paling sedikit tetapi jumlah node yang ditelusuri adalah yang paling banyak, menyebabkan waktu eksekusi terlama di antara keduanya. GBFS disisi lain memiliki waktu pencarian solusi tercepat, namun solusi yang didapatkan bukan solusi dengan move yang optimal untuk menyelesaikan permainan.

6.2 Saran

Untuk meningkatkan implementasi program penyelesaian puzzle Rush Hour, tambahkan mekanisme caching state untuk mempercepat algoritma UCS yang menjelajahi banyak node. Lakukan eksperimen dengan variasi papan (misalnya, 4x4 atau 8x8) dan test case lebih kompleks untuk menguji skalabilitas, serta pertimbangkan algoritma alternatif seperti IDA* atau pendekatan machine learning untuk eksplorasi solusi yang lebih optimal.

LAMPIRAN

Github Repository:  [Sigma boyzz](#)

Tabel berikut

No	Poin	Ya	Tidak
1	Program berhasil dikompilasi tanpa kesalahan	<input checked="" type="checkbox"/>	<input type="checkbox"/>
2	Program berhasil dijalankan	<input checked="" type="checkbox"/>	<input type="checkbox"/>
3	Solusi yang diberikan program benar dan mematuhi aturan permainan	<input checked="" type="checkbox"/>	<input type="checkbox"/>
4	Program dapat membaca masukan berkas .txt dan menyimpan solusi berupa print board tahap per tahap dalam berkas .txt	<input checked="" type="checkbox"/>	<input type="checkbox"/>
5	[Bonus] Implementasi algoritma pathfinding alternatif	<input type="checkbox"/>	<input checked="" type="checkbox"/>
6	[Bonus] Implementasi 2 atau lebih heuristik alternatif	<input checked="" type="checkbox"/>	<input type="checkbox"/>
7	[Bonus] Program memiliki GUI	<input checked="" type="checkbox"/>	<input type="checkbox"/>
8	Program dan laporan dibuat (kelompok) sendiri	<input checked="" type="checkbox"/>	<input type="checkbox"/>

DAFTAR PUSTAKA

- [1] R. Munir, *Route Planning - Bagian 1*, Institut Teknologi Bandung, 2025. [Online]. Available: https://edunexcontentprodhot.blob.core.windows.net/edunex/2025/73323-Algorithm-Strategy-Parent-Class/347610-Route-Planning/file/1746612438845_21-Route-Planning-2025-Bagian1
- [2] R. Munir, *Route Planning - Bagian 2*, Institut Teknologi Bandung, 2025. [Online]. Available: https://edunexcontentprodhot.blob.core.windows.net/edunex/2025/73323-Algorithm-Strategy-Parent-Class/347610-Route-Planning/file/1746612468144_22-Route-Planning-2025-Bagian2
- [3] A. Garg, "Best First Search Algorithm", OpenGenus IQ, 2021. [Online]. Available: <https://iq.opengenus.org/best-first-search/>