Optimasi Pencarian Solusi Permainan Rush Hour Menggunakan Algoritma A-Star Dengan Variasi Heuristik

Jovandra Otniel P. S. - 13523141

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung, Jalan Ganesha 10 Bandung

E-mail: jovandra.siregar@gmail.com, 13523141@std.stei.itb.ac.id

Abstract— ***Permainan Rush Hour adalah permainan papan (boardgame) baik melalui media fisik maupun media digital (online). Permainan ini sekilas tampak sederhana, biasanya terdiri papan 6×6 dan sejumlah kendaraan yang bisa digeser secara horizontal atau vertikal. Namun, di dalam kesederhanaan aturan tersebut, tersimpan tantangan besar: memindahkan kendaraan sehingga kendaraan primer (label 'P') bisa keluar melalui celah di tepi papan. Meski tak ada rotasi atau tumpang tindih, kombinasi posisi kendaraan bisa mencapai puluhan ribu variasi, sehingga solusi yang efisien memerlukan strategi pencarian yang tepat.***

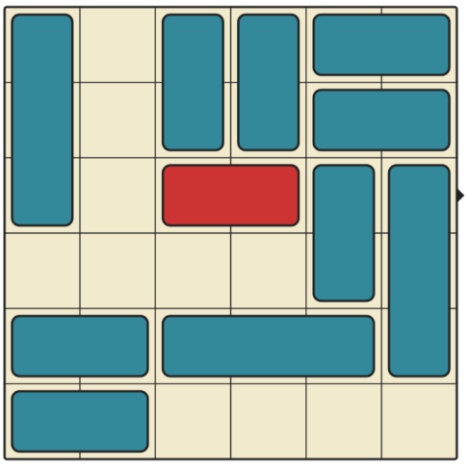
Keywords—a-star; rush hour; pathfinding; heuristic

I. PENDAHULUAN

Gambar 1.1 Permainan Rush Hour

Sumber:
https://www.lakeshorelearning.com/products/games/logic-games/rush-hour-logic-game/p/SBA5000/

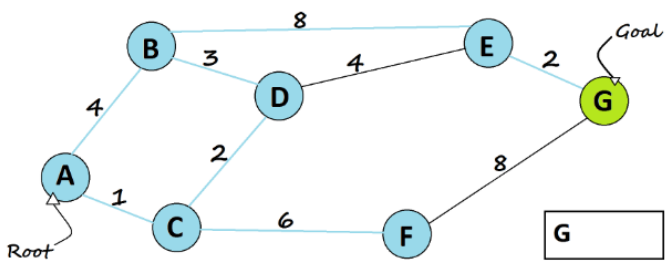
Permainan **Rush Hour** pada dasarnya merupakan sebuah teka-teki logika yang sederhana dalam konsep namun kompleks dalam implementasi. Permainan ini dimainkan pada papan berukuran enam kali enam sel, dengan sejumlah kendaraan yang hanya dapat bergerak secara horizontal atau vertikal, mengikuti arah orientasi masing-masing kendaraan tanpa kemampuan untuk berputar atau saling tumpang tindih.



Gambar 1.2 Ilustrasi Susunan Puzzle pada Rush Hour, persegi panjang merah mewakili kendaraan primer yang akan keluar

Sumber:
https://www.michaelfogleman.com/static/rush/#GoHIBBGoHICCGoAAJKooooJKDDEEEKFFoooo/23

Kehadiran rintangan statis yang dilabeli sebagai **X**, serta variasi posisi pintu keluar yang dilabeli **K** pada salah satu sisi papan, menambah dimensi kesulitan dan memperluas ruang status permainan secara signifikan. Kendaraan utama yang direpresentasikan sebagai rangkaian huruf **P** harus diarahkan hingga mencapai pintu keluar melalui serangkaian langkah strategis yang tidak hanya mengandalkan percobaan acak, tetapi juga pendekatan sistematis untuk meminimalkan jumlah langkah yang diperlukan.



Gambar 1.3 Ilustrasi Algoritma Path-Finding pada Graf

Sumber: https://medium.com/omarelgabrys-blog/path-finding-algorithms-f65a8902eb40

Dalam komputasi, setiap konfigurasi papan dapat direpresentasikan sebagai sebuah simpul pada graf tidak berarah, sedangkan setiap langkah pergeseran kendaraan satu sel diwakili sebagai sisi pada graf dengan bobot seragam, yakni satu satuan per sel. Jumlah kemungkinan konfigurasi yang dapat terbentuk, terutama pada tingkat kesulitan tinggi dengan banyak kendaraan dan rintangan, mencapai skala puluhan ribu hingga jutaan. Hal ini memunculkan tantangan komputasi yang serius, sebab jika eksplorasi ruang status dilakukan secara menyeluruh (*brute force*), maka pencariannya akan sangat tidak efisien baik dari segi kompleksitas waktu maupun ruang memori). Oleh karena itu, diperlukan metode pencarian jalur yang mampu menemukan solusi optimal dengan tetap menjaga efisiensi komputasi.

Makalah ini bertujuan untuk menganalisis pengaruh 3 fungsi heuristik terhadap algoritma A* yang dipakai dalam pencarian ruang solusi untuk puzzle Rush Hour. Makalah ini juga bertujuan untuk membandingkan performa algoritma A* dengan algoritma UCS (*Uniform Cost Search*) dan GBFS (*Greedy Best First Search*).

II. Landasan Teori

A. Representasi Ruang Status sebagai Graf Berarah

Dalam teori graf, suatu graf berarah berbobot didefinisikan sebagai triple (V,E,w), di mana V adalah himpunan simpul (*vertices*), $E \subseteq V \times V$ adalah himpunan sisi (*edges*), dan $w: E \rightarrow \mathrm{R}+$ adalah fungsi yang menetapkan bobot non-negatif kepada setiap sisi [1]. Untuk menerjemahkan *Rush Hour* ke dalam kerangka graf, setiap simpul $n \in V$ merepresentasikan satu konfigurasi papan, yaitu penempatan dan orientasi seluruh kendaraan serta rintangan 'X'. Setiap sisi $(n \rightarrow m) \in E$ terdefinisi jika dan hanya jika konfigurasi m dapat diperoleh dari n dengan menggeser tepat satu kendaraan sejauh Δ sel—baik horizontal maupun vertikal tanpa rotasi, *overlap*, atau melanggar batas papan. Bobot sisi $w(n,m)$ kemudian didefinisikan sebagai jumlah unit sel yang digeser, $|\Delta|$, sehingga total biaya kumulatif

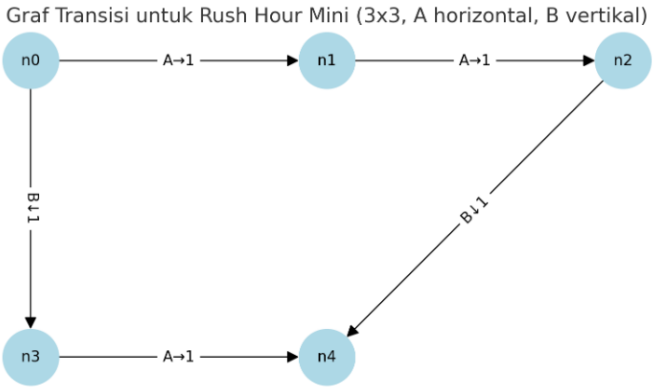
$$g(n) = \sum_{(u \rightarrow v) \in \pi} w(u,v)$$

menyatakan jumlah langkah unit sepanjang jalur π dari *state* (keadaan) awal ke *state* n [1].

Ruang status untuk puzzle *sliding-block* seperti *Rush Hour* terkenal memiliki kompleksitas eksponensial. Bila terdapat k kendaraan yang masing-masing dapat menempati hingga α posisi unik, maka potensi simpul $|V|$ dapat mendekati α^k. Pada papan 6×6 dengan 12–15 kendaraan, penghitungan praktis menunjukkan ratusan ribu atau bahkan jutaan konfigurasi. Rata-rata **branching factor** terdefinisi melalui persamaan berikut

$$b = \frac{1}{|V|} \sum_{n \in V} \deg^+(n)$$

dengan $deg^+(n)$ adalah jumlah tetangga m yang dapat dicapai dari n, biasanya terletak di kisaran 4 hingga 10 pada papan standar 6×6 [2].



Gambar 2.1. Subgraf transisi untuk dua kendaraan (A horizontal, B vertikal) pada papan 3×3

Sumber: Dokumentasi Pribadi

Pada graf tersebut, simpul n0, n1, n2, n3, n4 merepresentasikan status papan (*board*) berupa susunan mobil yang terparkir. Sisi (A → 1, B ↓ 1) menunjukkan setiap langkah yang dilakukan pada papan (perubahan status), Jika simpul daun menunjukkan mobil primer berhasil keluar melalui pintu, maka ruang solusinya berupa gabungan simpul dari akar hingga daun yang direpresentasikan sebagai ruang solusi.

Representasi graf ini bersifat sangat umum sehingga berbagai algoritma pencarian jalur optimal atau heuristik dapat diterapkan langsung. *Uniform-Cost Search* memperlakukan graf sebagai struktur bobot seragam dan mengekspansi state dengan $g(n)$ terendah terlebih dahulu, menjamin jalur minimum-biaya namun mengeksplorasi jauh lebih banyak simpul [3]. Sebaliknya, *Greedy Best-First Search* hanya mempertimbangkan estimasi heuristik $h(n)$ untuk memilih simpul berikutnya, sehingga sangat cepat namun tidak

menjamin optimalitas [4]. **A*** Search mengkombinasikan kekuatan keduanya melalui

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

di mana $h(n)$ adalah fungsi estimasi biaya tersisa yang *admissible*, yakni tidak pernah melebih biaya riil minimum ke goal, sehingga A* terjamin optimal dan komplet selama h memenuhi kondisi *admissible* dan konsisten [5].

Karena bobot sisi positif ($w(n,m)>0$), prioritas simpul dalam *open set* dapat diimplementasikan menggunakan *min-heap* atau *priority queue* dengan kompleksitas waktu $O(|E| + |V| log |V|)$, atau secara kasar $O(bd)$ untuk graf eksponensial, di mana d adalah kedalaman solusi. Tantangan utama dalam *Rush Hour* adalah menahan ledakan eksplorasi ini, memanfaatkan heuristik informatif dan teknik pruning seperti *closed set* untuk memblokir siklus serta menyimpan *best-cost* guna menghindari rekonstruksi jalur yang sama.

B. Algoritma Pencarian Jalur Optimal

Berbagai algoritma pencarian jalur dapat diterapkan untuk menemukan urutan langkah terpendek pada graf yang merepresentasikan ruang status Rush Hour. Pada subbab ini akan dibahas empat pendekatan utama beserta karakteristiknya, sebelum akhirnya memilih fokus pada satu metode.

Breadth-first Search menjelajahi graf secara bertingkat level by level mulai dari simpul awal. Pada graf berbiaya seragam (sisi selalu bernilai satu), BFS menjamin menemukan jalur dengan jumlah langkah minimum ke simpul tujuan. Namun kompleksitas memori BFS tumbuh eksponensial menurut kedalaman solusi dan faktor percabangan rata-rata, membuatnya cepat mencapai batas memori bahkan pada papan 6×6 dengan belasan kendaraan [6].

Uniform-cost Search menggeneralisasi BFS ke kondisi bobot sisi yang tidak seragam dengan selalu mengekspansi simpul berbiaya kumulatif terendah terlebih dahulu. Algoritma ini menjamin keoptimalan solusi terpendek pada graf berarah maupun tidak berarah, terlepas dari nilai bobot tiap sisi. Dalam konteks Rush Hour, di mana bobot sisi tetap satu per sel, Uniform-cost Search memiliki perilaku identik dengan BFS, tetapi dengan *overhead* manajemen prioritas tambahan yang tidak diperlukan jika bobot seragam [3].

Greedy Best-first Search mempercepat eksplorasi dengan hanya mempertimbangkan fungsi heuristik $h(n)$, yaitu estimasi biaya dari simpul n menuju tujuan. Dengan memprioritaskan simpul paling "dekat" menurut heuristik, algoritma ini dapat sangat mengurangi jumlah simpul yang diperiksa. Kekurangannya terletak pada ketiadaan jaminan keoptimalan di mana heuristik yang terlalu agresif dapat membawa pencarian ke jalur suboptimal [4].

A* menggabungkan kelebihan Uniform-cost Search dan Best-first Search melalui fungsi evaluasi

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

di mana $g(n)$ adalah biaya sebenarnya dari simpul awal ke n dan $h(n)$ adalah estimasi biaya dari n ke simpul tujuan. Dengan menggunakan heuristik *admissible* (tidak melebihkan biaya riil) seperti jarak Manhattan pada grid, A* menjamin menemukan jalur terpendek sekaligus memfokuskan ekspansi pada simpul yang paling menjanjikan. Keunggulan ini membuat A* secara konsisten lebih efisien daripada Uniform-cost Search dan lebih andal ketimbang Best-first Search saja dalam konteks pencarian jalur optimal pada ruang status besar [5].

Karena kombinasi jaminan keoptimalan dan efisiensi eksplorasi, pada penelitian ini fokus diberikan pada implementasi dan optimasi algoritma A* untuk problematika permainan Rush Hour.

C. Heuristik dan Sifat Admissible

Heuristik adalah fungsi evaluasi $h(n)$ yang memberikan estimasi biaya terendah dari simpul n menuju simpul tujuan tanpa melakukan eksplorasi penuh. Dalam kerangka algoritma A*, heuristik berperan sebagai "indera keenam" yang membimbing pencarian menuju jalur yang paling menjanjikan, sehingga dapat mengurangi jumlah simpul yang harus diperiksa. Dua sifat utama yang harus dimiliki heuristik untuk menjamin keoptimalan A* adalah admisibilitas (*admissibility*) dan, idealnya, konsistensi (*consistency*).

Sebuah heuristik dikatakan *admissible* apabila untuk setiap simpul n, nilai $h(n)$ tidak pernah melebihi biaya sebenarnya $h^*(n)$ dari n ke tujuan:

$$\forall n: 0 \leq h(n) \leq h^*(n).$$

Karena $h(n)$ berperan sebagai *lower bound* (batas bawah) pada biaya nyata, A* dengan heuristik *admissible* akan menjamin penemuan solusi terpendek pertama kali saat mencapai simpul tujuan, tanpa pernah terperangkap pada jalur suboptimal [1], [3]. Sebagai contoh paling sederhana, heuristik nol $h(n)=0$ untuk semua n selalu *admissible*, tetapi A* akan berperilaku seperti *uniform-cost search* dan kehilangan manfaat yang diperoleh dari fungsi heuristik [7].

Konsistensi atau kemonotonan mensyaratkan bahwa estimasi heuristik di simpul induk tidak boleh terlalu jauh lebih besar daripada biaya langkah satu unit ditambah heuristik di simpul anak:

$$\forall (n, n'): h(n) \leq c(n, n') + h(n'),$$

Di mana $c(n,n')$ adalah biaya sisi antara n dan n'. Konsistensi memastikan bahwa nilai fungsi $f(n)=g(n)+h(n)$ tidak menurun sepanjang lintasan, sehingga simpul yang sudah di-*expand* tidak perlu diperluas ulang. Semua heuristik konsisten bersifat *admissible*, meskipun tidak selalu berlaku kebalikannya.

Heuristik yang dapat dipakai dalam pencarian Solusi permainan Rush Hour ini adalah Jarak Manhattan. Pada papan grid, jarak

Manhattan menghitung perbedaan koordinat horizontal ditambah vertikal antara kendaraan primer dan pintu keluar:

$$h_{Manhattan}(n) = | x_P - x_K | + | y_P - y_K |$$

Dengan p mewakili kendaraan primer dan k mewakili pintu keluar [8].

Heuristik ini memiliki kompleksitas yang sangat murah (O(1) per simpul), konsisten, dan *admissible* karena setiap langkah hanya memindahkan kendaraan satu sel sejajar sumbu [3].

Khusus permainan Rush Hour dan kasus lain yang serupa, heuristik lain yang dapat dipakai ialah jumlah kendaraan penghalang. Heuristik ini menghitung berapa banyak kendaraan lain yang berada langsung di depan jalur kendaraan primer menuju celah keluar. Setiap kendaraan penghalang setidaknya memerlukan satu langkah untuk digeser, sehingga

$$h_{block}(n) = jumlah\_blokir(n)$$

bersifat *admissible* walau kurang akurat ketimbang Manhattan. Untuk meningkatkan akurasi pada aplikasi ini, kedua heuristik h_{block} dan $h_{Manhattan}$ ditambah menjadi h:

$$h(n) = h_{block}(\mathrm{n}) + h_{Manhattan}(\mathrm{n}).$$

III. IMPLEMENTASI

Bagian ini mendokumentasikan realisasi perangkat lunak **RushHour Solver** yang ditulis seluruhnya dalam Java 21 tanpa dependensi eksternal. Seluruh berkas sumber ditempatkan di satu folder—memudahkan proses kompilasi dengan satu perintah javac *.java. Implementasi dibagi menjadi lapisan data-model (*Board*, *Piece*, *State*), lapisan algoritme pencarian (*Searchers*), lapisan heuristik (*Heuristics*), antarmuka baris-perintah (*Main* + Printer), dan antarmuka grafis (*MainGUI* + BoardPanel). Gambar 3-1 merangkum diagram komponen dan aliran kontrol program.

Gambar 3.1 Arsitektur logis RushHour Solver.

A. Lingkungan & Ketergantungan

Program dikembangkan serta diuji di Jawa Development Kit 23 pada Windows 11 x6. 4Karena kode hanya memakai pustaka standar (java.util.*, java.nio.file.*, dan javax.swing.*), ia dapat dijalankan di JVM mana pun versi 11 ke atas. Tidak diperlukan instalasi Maven/Gradle ataupun GUI toolkit pihak ketiga.

B. Struktur Berkas

```
├── Main.java
├── Parser.java
├── Printer.java
├── Board.java
│   ├─ record Pos // (r,c) untuk rintangan 'X'
│   ├─ class Piece // id, row, col, length,
│   │  orient
│   ├─ class State // Board state, parent,
│   │  Move, g, h, isGoal
│   ├─ enum Orientation     // HORIZONTAL,
│   │  VERTICAL
│   └─ methods:
│      • parse(…)          // baca file,
│        deteksi pintu 4 sisi,    rintang
│      • expand(State)     // gen semua child
│        State
│          - slideHorizontal(…)
│          - slideVertical(…)
│      • createChild(…)    // buat State baru
│        & grid baru
├── Searchers.java
│   ├─ interface PathFinder
│   │  {search(Board,Heuristic) }
│   ├─ class SearchResult    // goal State +
│   │  visitedCount
│   ├─ UniformCostSearch     // implements
│   │  PathFinder
│   ├─ GreedyBestFirstSearch // implements
│   │  PathFinder
│   └─ AStarSearch           // implements
│      PathFinder
└── Heuristics.java
    ├─ interface Heuristic { estimate(Board) }
    └─ class Heuristics
       • H1, H2, H3   // tiga fungsi h(n)
       • byId(int)          // selector
```

C. Pemodelan Papan

Kelas Board bersifat *immutable*: setelah dibuat, tidak diubah; setiap gerakan menghasilkan objek Board baru. Empat *field* primer:

```
public class Board {
    // atribut public final untuk memudahkan akses
    public final int rows, cols;
    public final char[][] grid;             // '.' untuk kosong, 'X' untuk halangan
    public final Map<Character, Piece> pieces; // id -> Piece (tidak termasuk exit)
    public final Piece primary;             // primary piece 'P'
    public final int exitRow, exitCol;      // koordinat 'K'

    public record Pos(int r,int c){}
    public final Set<Pos> obstacles;
```

Gambar 3.1 Kode Sumber Kelas Board (Papan)
Sumber: Dokumentasi Pribadi

Posisi kendaraan dikelola oleh kelas Piece yang menyimpan id, row, col, length,
serta Orientation (HORIZONTAL/VERTICAL). Fungsi bantu tailRow() dan tailCol() memudahkan deteksi ujung kendaraan.

Board.parse(List<String>) memproses file .txt dengan langkah-langkah berikut:

1. *Header* : membaca dimensi *R*×*C*; baris kedua (jumlah piece) dilewati jika ada.
2. Deteksi pintu 'K' : jika K berdiri sendiri di baris sebelum grid ⇒ exitRow=-1; jika di baris setelah grid

⇒ exitRow=rows; jika menempel di kiri atau kanan baris grid ⇒ exitCol=-1 atau cols.

3. Pengisian grid : setiap karakter yang bukan ”.” ,X dicatat pada locs; X dimasukkan ke set *obstacles*.
4. Konversi ke objek Piece : kumpulan koordinat per id dikelompokkan untuk menentukan orientasi dan panjang kendaraan.

Parsing memvalidasi bahwa:

- Terdapat tepat satu *primary* ‘P’.
- Pintu keluar sejajar orientasi ‘P’.
- Tidak ada karakter ilegal dalam grid.

D. Pembuatan Simpul Tetangga (Neighbour Expansion)

Metode expand(State) melintasi seluruh kendaraan dan memanggil slideHorizontal atau slideVertical. Tiap fungsi melaksanakan:

- Iterasi kiri/kanan (atau atas/bawah) sampai sel berikutnya bukan ‘.’ atau keluar batas.
- Untuk setiap jarak δ yang valid, dibuat State baru dengan:
    - grid baru (*re-render* keseluruhan; biaya $O(RC)$ namun tetap konstan pada papan 6×6),
    - biaya kumulatif g’=g+|δ|,
    - objek Move(id,δ) untuk rekonstruksi jalur (menyatakan pergerakan potongan puzzle).
- Jika kendaraan adalah ‘P’ dan jalur ke exitRow/exitCol kosong, dibuat *goal-state* dengan isGoal=true.

```java
public List<State> expand(State parent) {
    List<State> next = new ArrayList<>();
    for (Piece p : pieces.values()) {
        if (p.orient == Orientation.HORIZONTAL) {
            slideHorizontal(parent, next, p);
        } else {
            slideVertical(parent, next, p);
        }
    }
    return next;
}
```

Gambar 3.2 Kode Sumber Fungsi expand()
Sumber: Dokumentasi Pribadi

```java
private void slideHorizontal(State parent, List<State> out, Piece p) {
    // left
    int c = p.col - 1;
    while (c >= 0 && grid[p.row][c] == '.') { out.add(createChild(parent, p, c - p.col)); c--; }
    // right
    c = p.tailCol() + 1;
    while (c < cols && grid[p.row][c] == '.') { out.add(createChild(parent, p, c - p.tailCol())); c++; }
    // special: horizontal exit for primary piece
    if (p.id == 'P' && p.row == exitRow) {
        // exit on right side
        if (exitCol == cols) {
            boolean clear = true;
            for (int cc = p.tailCol() + 1; cc < cols; cc++) {
                if (grid[p.row][cc] != '.') { clear = false; break; }
            }
            if (clear) {
                State goal = createChild(parent, p, cols - p.tailCol());
                goal.isGoal = true; out.add(goal);
            }
        }
        // exit on left side
        else if (exitCol == -1) {
            boolean clear = true;
            for (int cc = p.col - 1; cc >= 0; cc--) {
                if (grid[p.row][cc] != '.') { clear = false; break; }
            }
            if (clear) {
                int delta = -(p.col + 1);
                State goal = createChild(parent, p, delta);
                goal.isGoal = true; out.add(goal);
            }
        }
    }
}
```

Gambar 3.3 Kode Sumber Fungsi slideHorizontal()
Sumber: Dokumentasi Pribadi

```java
private void slideVertical(State parent, List<State> out, Piece p) {
    int r = p.row - 1;
    while (r >= 0 && grid[r][p.col] == '.') {
        out.add(createChild(parent, p, r - p.row));
        r--;
    }
    // geser ke bawah
    r = p.tailRow() + 1;
    while (r < rows && grid[r][p.col] == '.') {
        out.add(createChild(parent, p, r - p.tailRow()));
        r++;
    }
    // special: exit vertikal (atas/bawah)
    if (p.id == 'P' && p.col == exitCol) {
        // exit di bawah (exitRow == rows)
        if (exitRow == rows) {
            boolean clear = true;
            for (int rr = p.tailRow() + 1; rr < rows; rr++) {
                if (grid[rr][p.col] != '.') { clear = false; break; }
            }
            if (clear) {
                // geser P sampai keluar bawah
                State goal = createChild(parent, p, rows - p.tailRow());
                goal.isGoal = true;
                out.add(goal);
            }
        }
        // exit di atas (exitRow == -1)
        else if (exitRow == -1) {
            boolean clear = true;
            for (int rr = p.row - 1; rr >= 0; rr--) {
                if (grid[rr][p.col] != '.') { clear = false; break; }
            }
            if (clear) {
                // geser P sampai keluar atas
                int delta = -(p.row + 1);
                State goal = createChild(parent, p, delta);
                goal.isGoal = true;
                out.add(goal);
            }
        }
    }
}
```

Gambar 3.4 Kode Sumber Fungsi slideHorizontal()
Sumber: Dokumentasi Pribadi

E. *Algoritma Pencarian*

1) *Uniform-Cost Search*

Menggunakan PriorityQueue<State> ber-komparator g. Map bestCost menyimpan biaya terendah ke setiap konfigurasi agar duplikasi state ditolak. Kompleksitas: $O(|E|+|V|\log|V|)$.

2) *Greedy Best-First Search*

Priority queue memakai komparator h. Closed set (HashSet<Board>) mencegah revisit. Implementasi sederhana karena g diabaikan.

3) *A* Search*

Priority queue default (compareTo) berdasarkan $f=g+h$. Struktur bestCost identik dengan UCS, tetapi penambahan heuristik memperkecil frontier. Heuristik diberikan melalui antarmuka fungsional Heuristic.

```
public class AStarSearch implements PathFinder {
    @Override
    public SearchResult search(Board start, Heuristic h) {
        PriorityQueue<State> open = new PriorityQueue<>();
        Map<Board, Integer> best = new HashMap<>();
        State root = new State(start, null, new Move('-', 0), 0);
        root.h = h.estimate(start); open.add(root); best.put(start, 0);
        long visited = 0;
        while (!open.isEmpty()) {
            State cur = open.poll(); visited++;
            if (cur.isGoal) return new SearchResult(cur, visited);
            for (State nxt : cur.board.expand(cur)) {
                nxt.h = h.estimate(nxt.board);
                if (best.getOrDefault(nxt.board, Integer.MAX_VALUE) > nxt.g) {
                    best.put(nxt.board, nxt.g);
                    open.add(nxt);
                }
            }
        }
        return new SearchResult(null, visited);
    }
}
```

Gambar 3.5 Kode Sumber Kelas AstarSearch
Sumber: Dokumentasi Pribadi

F. *Heuristik*

Tiga heuristik diterapkan sebagai berikut:

ID	Rumus	Sifat	Kompleksitas evaluasi
H1	dist + 2×blockers	*admissible*	$O(dist)$
H2	dist + 1×blockers	*admissible*	$O(dist)$
H3	dist + 2×blockers + secondaryPenalty	*inadmissible*	$O(dist)$ + pengecekan sisi

Tabel 3.1 Perbandingan Fungsi Heuristik

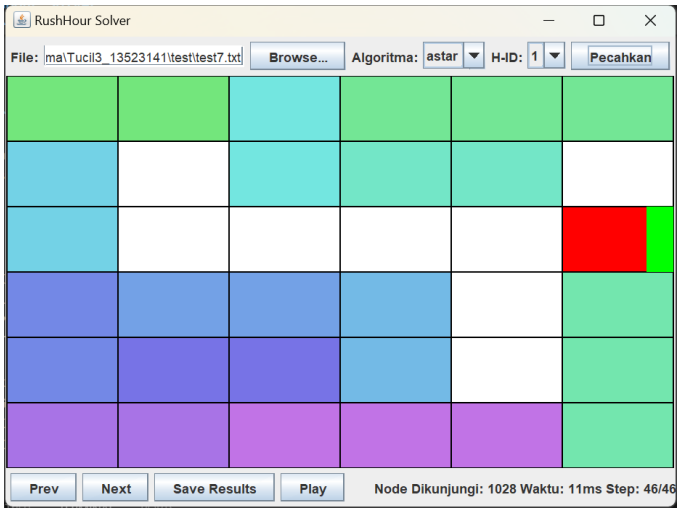
Dalam implementasi algoritma A* untuk permainan Rush Hour ini, digunakan tiga variasi heuristik yang masing-masing memiliki pendekatan perhitungan dan sifat evaluasi yang berbeda-beda. Ketiganya berbasis kombinasi jarak (jumlah sel kosong antara kendaraan primer dan pintu keluar) serta jumlah kendaraan penghalang langsung di jalur kendaraan primer. *Dist* berarti jarak terdekat dari kendaraan primer ke pintu keluar, *blockers* berarti jumlah penghalang antara kendaraan primer dan pintu keluar. Sementara itu, *secondaryPenalty* merupakan jumlah penghalang sekunder yang menghalangi penghalang primer (penghalang yang menghalangi penghalang).

G. *Antarmuka GUI*

MainGUI memperluas JFrame dan menjalankan pencarian di *Event Dispatch Thread* melalui SwingUtilities.invokeLater. Elemen kunci:

- *File chooser* untuk memuat berkas.
- *Combo box* algoritma serta heuristik.
- *BoardPanel* yang menerjemahkan grid ke warna: P merah, kendaraan lain palet HSB, X abu-gelap, K hijau.
- *Navigasi tombol Prev/Next* memungkinkan pengguna menelusuri jalur solusi langkah demi langkah; status bar menunjukkan "Node Dikunjungi, Waktu, Langkah i/N".
- *Save Results* menyimpan solusi berupa susunan puzzle dari awal (simpul akar) hingga akhir (simpul daun) ke .txt.

Board digambar ter-centre dengan ukuran sel adaptif berdasarkan dimensi panel. Pintu di luar grid digambar sebagai persegi panjang hijau tebal setinggi lebar seperempat sel.



Gambar 3.6 Tampilan GUI Aplikasi Rush Hour Solver

Sumber: Dokumentasi Pribadi

IV. Eksperimen

Pada makalah ini, dilakukan percobaan penyelesaian puzzle Rush Hour menggunakan aplikasi yang telah diimplementasikan sebelumnya. Berbagai variasi, mulai dari papan, potongan puzzle yang berupa mobil, pintu keluar, hingga halangan (*obstacles*). Pada papan, terdapat variasi berupa ukuran papan (biasanya 6x6). Potongan puzzle bervariasi dari jumlah, ukuran, dan orientasi puzzle. Pintu keluar bervariasi dari lokasi pintu keluar yang bisa berada di sisi kiri, kanan, bawah, atau atas puzzle (*mutually exclusive*). Terakhir, halangan mempunyai variasi yang sama dengan potongan puzzle namun halangan tidak bisa digerakkan.

Masukan Susunan Puzzle Rush Hour pada Percobaan Pertama, Kedua, Ketiga:

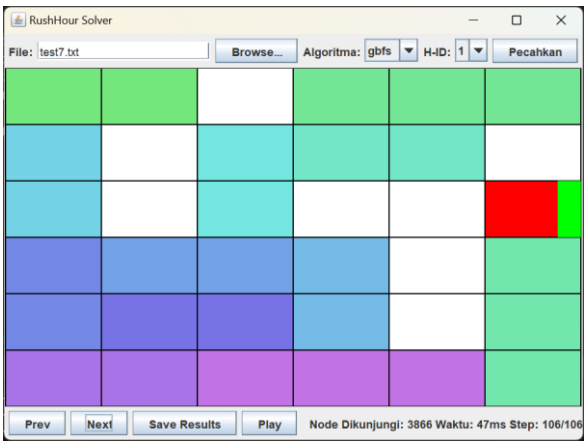
```
6 6
13
AABBBC
DDE..C
F.EPPCK
F..GHH
I..GJJ
ILLMMM
```

Keluaran Percobaan Pertama (UCS):

Gambar 4.1 Tampilan Keluaran GUI Aplikasi

Sumber: Dokumentasi Pribadi

Keluaran Percobaan Kedua (GBFS):



Gambar 4.2 Tampilan Keluaran GUI Aplikasi

Sumber: Dokumentasi Pribadi

Keluaran Percobaan Ketiga (A*):

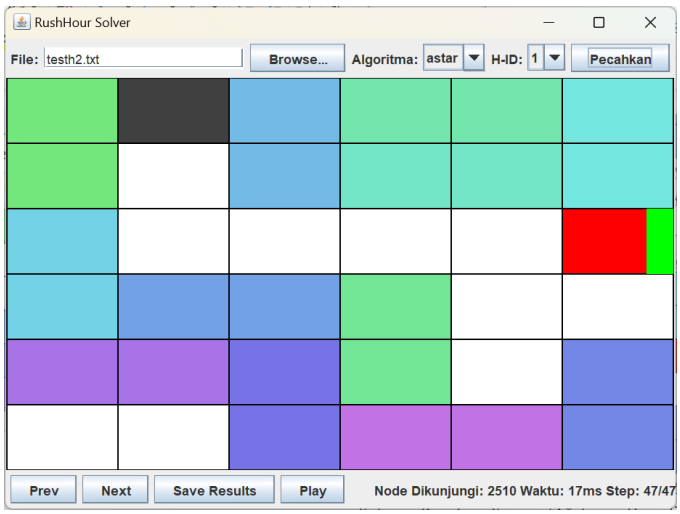
Gambar 4.3 Tampilan Keluaran GUI Aplikasi

Sumber: Dokumentasi Pribadi

Masukan susunan puzzle pada percobaan keempat, kelima dan keenam:

```
6 6
12
AX.BCC
ADDB.E
F.GPPEK
F.GHHI
..JLLI
..JMM.
```

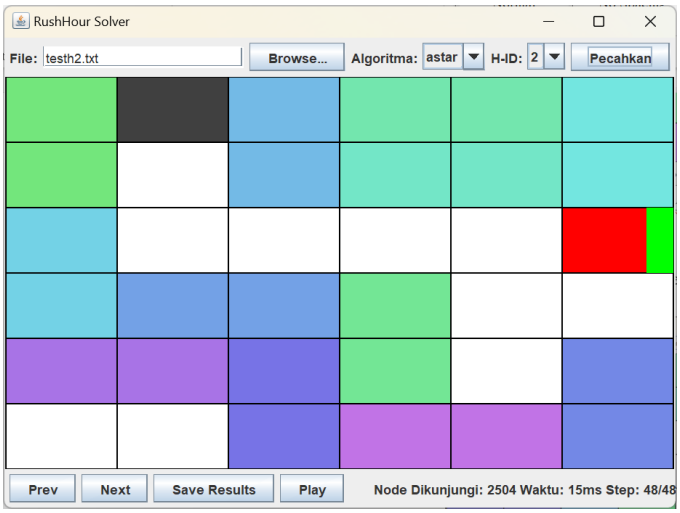
Keluaran Percobaan Keempat (A* dengan Heuristik H1: *blocker + 2 * distance*):



Gambar 4.4 Tampilan Keluaran GUI Aplikasi

Sumber: Dokumentasi Pribadi

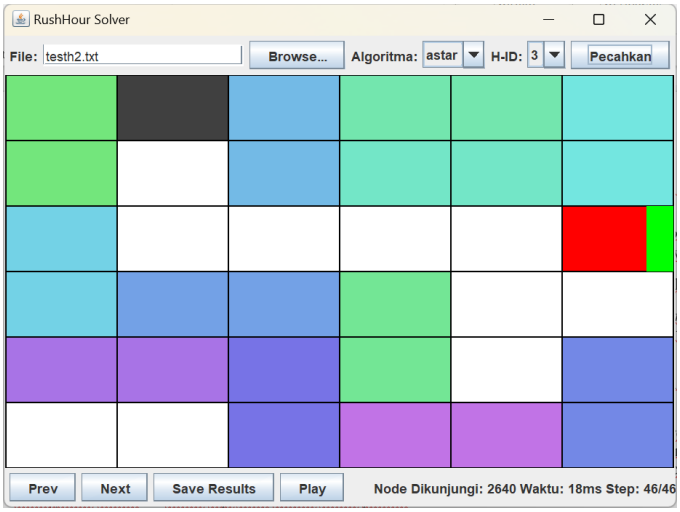
Keluaran Percobaan Kelima (A* dengan Heuristik H2: *blocker + distance*):



Gambar 4.5 Tampilan Keluaran GUI Aplikasi

Sumber: Dokumentasi Pribadi

Keluaran Percobaan Keenam (A* dengan Heuristik H3: *H1 + secondary_block*):



Gambar 4.6 Tampilan Keluaran GUI Aplikasi

Sumber: Dokumentasi Pribadi

Pada percobaan pertama, kedua, dan ketiga, terlihat perbedaan yang signifikan antara ketiga algoritma. *UCS* mengekspansi 1.170 simpul dalam waktu sekitar 12 ms dan menghasilkan solusi 49 langkah. *GBFS*, meskipun sekilas "hemat biaya ekspansi awal", justru mengunjungi jauh lebih banyak simpul (3.866 nodes) dan butuh waktu sekitar 47 ms, serta jalurnya malah jauh lebih panjang (106 langkah) karena ia murni mengejar heuristik tanpa mempertimbangkan biaya sejauh ini.

Sementara itu, *A** mencapai keseimbangan yang terbaik: hanya 1.028 simpul yang diperiksa (lebih sedikit daripada UCS), waktu 17 ms, dan rute terpendek (46 langkah). Ini menegaskan bahwa *A** bukan saja lebih efisien dalam jumlah node dibanding UCS berkat heuristik *admissible*-nya, tetapi juga lebih optimal dan lebih cepat daripada GBFS pada puzzle Rush Hour ini.

Pada percobaan keempat, kelima, dan keenam, eksperimen dilakukan pada konfigurasi permainan Rush Hour berukuran 6×6 dengan total 12 kendaraan, di mana kendaraan primer (berwarna merah) harus digerakkan menuju pintu keluar di sisi kanan papan. Tiga varian heuristik diterapkan menggunakan algoritma A*, yakni H1 (jarak + 2×penghalang), H2 (jarak + 1×penghalang), dan H3 (jarak + 2×penghalang + penalti sekunder). Setiap heuristik diuji untuk mengukur jumlah *node* yang dikunjungi, waktu eksekusi, serta panjang solusi (jumlah langkah yang diperlukan hingga kendaraan primer keluar).

Hasil menunjukkan bahwa H1 mampu menghasilkan solusi optimal dengan 47 langkah, mengunjungi 2.510 node dalam waktu sekitar 17 milidetik. Heuristik ini bersifat *admissible* sehingga A* dijamin menemukan jalur terpendek, meskipun jumlah node yang dikunjungi tetap signifikan karena estimasinya yang konservatif. Sementara itu, H2 yang juga *admissible*, memberikan hasil sedikit berbeda: panjang solusi 48 langkah dengan 2.504 node dikunjungi dalam 15 milidetik. H2 cenderung lebih longgar dalam estimasi karena hanya memberikan bobot satu pada setiap kendaraan penghalang, sehingga kadang memperluas jalur yang sedikit kurang efisien namun dengan biaya eksplorasi yang lebih rendah.

Berbeda dari keduanya, H3 menerapkan penalti tambahan untuk kendaraan yang menghalangi pergerakan kendaraan penghalang langsung, serta secara agresif menghindari konfigurasi yang tidak dapat diselesaikan (misalnya ketika terdapat rintangan ‘X’ di jalur keluar). Meskipun bersifat *inadmissible*, H3 justru berhasil menemukan solusi lebih pendek yakni 46 langkah. Namun, untuk mencapai hasil ini, A* dengan H3 memperluas lebih banyak node (2.640 node) dan membutuhkan waktu eksekusi sedikit lebih lama (18 milidetik), akibat *overhead* evaluasi penalti tambahan di setiap simpul.

Dari hasil tersebut terlihat adanya *trade-off* yang jelas antara ketepatan estimasi dan efisiensi eksplorasi. Heuristik H1 memberikan keseimbangan terbaik untuk aplikasi yang mengutamakan optimalitas mutlak, sedangkan H2 cocok diterapkan ketika efisiensi eksplorasi dan waktu proses lebih diutamakan tanpa mengorbankan jaminan solusi optimal. Sementara itu, H3 bisa menjadi pilihan menarik pada kasus-kasus yang mentoleransi deviasi kecil dari optimalitas demi mempercepat penyelesaian atau memotong (*pruning*) jalur eksplorasi yang tidak menjanjikan.

V. Kesimpulan

Berdasarkan hasil percobaan sebelumnya, dapat disimpulkan bahwa heuristik H1 dan H2 menjamin estimasi biaya tidak berlebihan. Kedua heuristik menghasilkan jalur optimal sambil menahan jumlah simpul yang dikunjungi dan waktu pencarian pada level yang seimbang. Di antara keduanya, H2 terbukti paling cepat karena estimasinya lebih longgar, meski terkadang menambah satu langkah pada hasil akhir. Sementara itu, H3 dengan penalti sekunder mampu memangkas panjang solusi menjadi paling singkat, namun ada *trade*-off dengan eksplorasi simpul lebih banyak dan durasi komputasi yang meningkat.

Selain heuristik, algoritma yang dipakai juga menghasilkan jumlah langkah (*step*) dan waktu komputasi yang berbeda juga. Dari ketiga algoritma yang diuji (UCS, GBFS, A*), dapat disimpulkan bahwa A* dengan heuristik “jarak + 1×blocker” (H2) menawarkan kombinasi terbaik antara keoptimalan dan efisiensi. A* secara konsisten mengekspansi lebih sedikit simpul daripada UCS, sehingga mengurangi waktu eksekusi tanpa mengorbankan panjang solusi, sedangkan UCS meski selalu menemukan jalur terpendek, cenderung mengekspansi wilayah pencarian yang jauh lebih luas sehingga lebih lambat. GBFS, walaupun sangat cepat pada langkah-langkah awal karena hanya mengejar estimasi heuristik, tidak menjamin jalur optimal dan dalam beberapa kasus mengunjungi jauh lebih banyak simpul serta menghasilkan langkah solusi yang lebih panjang.

VI. Penutup

Puji Syukur kepada Tuhan Yang Maha Esa atas rahmat-Nya dalam pembuatan makalah IF2211 Strategi Algoritma dengan topik “Optimasi Pencarian Solusi Permainan Rush Hour Menggunakan Algoritma A-Star Dengan Variasi Heuristik”. Saya mengucapkan terima kasih kepada dosen IF2211 beserta para penulis yang tulisannya menjadi referensi dalam makalah ini.

Referensi

[1] K. Kask, “Informed Heuristic Search,” UC Irvine, 2024. [Online]. Available: https://www.ics.uci.edu/~kkask/Fall-2016%20CS271/slides/03-InformedHeuristicSearch.pdf. [Diakses 24 Juni 2025].

[2] E. B. B. Gary William Flake, “Rush Hour is PSPACE-complete, or “Why you should generously tip parking lot attendants”,” *Theoretical Computer Science,* vol. 270, no. 1-2, pp. 895-911, 2002.

[3] GeeksForGeeks, “Uniform Cost Search (UCS) in AI,” Geeks For Geeks, 23 Agustus 2024. [Online]. Available: https://www.geeksforgeeks.org/artificial-intelligence/uniform-cost-search-ucs-in-ai/. [Diakses 24 Juni 2025].

[4] R. Munir, “Penentuan Rute (Route/Path Planning) Bagian 1: BFS, DFS, UCS, Greedy Best First Search,” 2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-(2025)-Bagian1.pdf. [Diakses 24 Juni 2025].

[5] R. Munir, “Penentuan Rute (Route/Path Planning) Bagian 2: Algoritma A*,” 2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf. [Diakses 24 Juni 2025].

[6] D. Tamara, “BFS (Breadth First Search) : Pengertian, Kekurangan, Kelebihan, dan Contohnya,” Medium, 15 Oktober 2021. [Online].

Available: https://medium.com/@defytamara2610/bfs-breadth-first-search-pengertian-kekurangan-kelebihan-dan-contohnya-775a7d808fbc. [Diakses 24 Juni 2025].

[7] G. F. Geeks, "Admissibility of A* Algorithm," Geeks For Geeks, 6 April 2023. [Online]. Available: https://www.geeksforgeeks.org/dsa/a-is-admissible/. [Diakses 24 Juni 2025].

[8] V. Chugani, "What is Manhattan Distance?," datacamp, 17 Juli 2024. [Online]. Available: https://www.datacamp.com/tutorial/manhattan-distance. [Diakses 24 Juni 2025].

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 24 Juni 2025

Jovandra Otniel P. S. (13523141)