

Laporan Tugas Kecil 3
IF 2211 Strategi Algoritma
Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding



Disusun oleh:

1. Raka Daffa Iftikhaar - 13523018
2. Ahsan Malik Al Farisi - 13523074

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA - KOMPUTASI
INSTITUT TEKNOLOGI BANDUNG
JL. GANESA 10, BANDUNG, 40132
2025

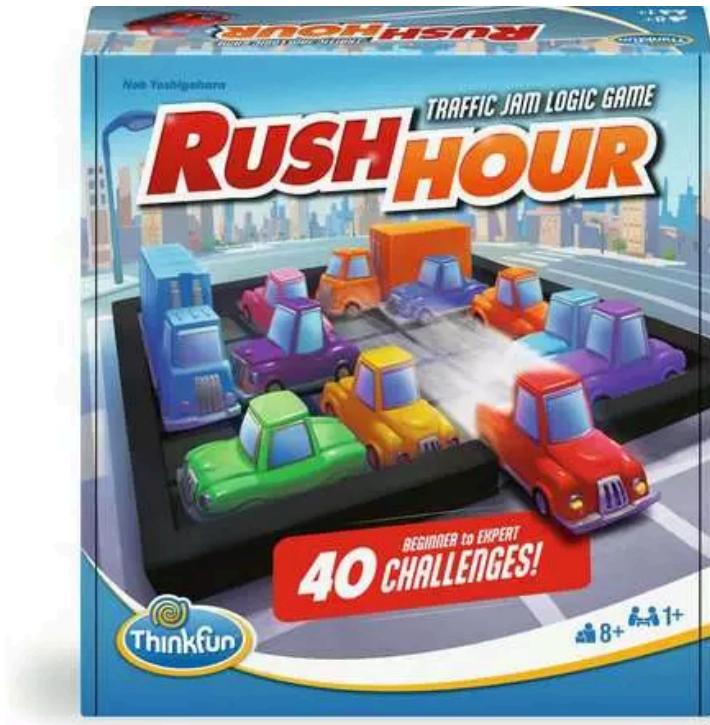
Daftar Isi

BAB I Deskripsi Tugas	1
BAB II Landasan Teori	5
2.1 Dasar Teori	5
2.1.1 Algoritma Uniform Cost Search	5
2.1.2 Algoritma Greedy Best First Search	6
2.1.3 Algoritma A*	7
2.1.4 Algoritma Iterative Deepening Depth First Search	8
BAB III Analisis	10
3.1 Analisis Algoritma	10
3.1.1 Definisi f(n) dan g(n)	10
3.1.2 Algoritma Uniform Cost Search	10
3.1.3 Algoritma Greedy Best First Search (GBFS)	11
3.1.4 Algoritma A* Search	11
3.1.5 Algoritma Iterative Deepening Depth First Search	12
3.1.6 Algoritma UCS dan BFS pada Rush Hour	12
3.1.7 Perbandingan A* dan UCS	13
BAB IV Implementasi dan Pengujian	14
4.1 Source Code	14
4.1.1 Components	14
Piece.java	14
Move.java	16
Board.java	17
State.java	23
IO.java	28
4.1.2 Solver	35
UniformCostSearch.java	35
Heuristic.java	36
GreedyBestFirstSearch.java	38
AStar.java	39
IterativeDeepeningSearch.java	41
4.1.3 GUI (Swing)	42
MainGUI.java	42
4.1.4 Main	51
Main.java	51
4.2 Hasil Pengujian	54
4.2.1 Test Case	54
4.2.1 Algoritma UCS	56
4.2.2 Algoritma GBFS	59
4.2.3 Algoritma A*	67
4.2.4 Algoritma IDDFS	75

4.3 Analisis Pengujian	78
4.3.1 Analisis Pengujian Uniform Cost Search (UCS)	78
4.3.2 Analisis Pengujian Greedy Best First Search (GBFS)	79
4.3.3 Analisis Pengujian A* Search	80
4.3.4 Analisis Pengujian Iterative Deepening Depth First Search (IDDFS)	80
4.3.5 Analisis Komparatif	81
BAB V Implementasi Bonus	82
5.1 Algoritma Alternatif	82
5.2 Heuristik Alternatif	82
5.3 Graphical User Interface	83
5.3.1 Tampilan Utama	83
5.3.2 Load Board	83
5.3.3 Pemilihan Algoritma dan Heuristik	83
5.3.4 Solve dan Visualisasi Solusi	83
5.3.5 Kontrol Animasi	83
5.3.6 Status dan Statistik	83
5.3.7 Save Solution	84
5.3.8 Visualisasi Papan	84
BAB VI Penutup	85
6.1 Kesimpulan	85
6.2 Saran	85
6.3 Refleksi	85
Daftar Pustaka	86
Lampiran	87

BAB I

Deskripsi Tugas



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*.

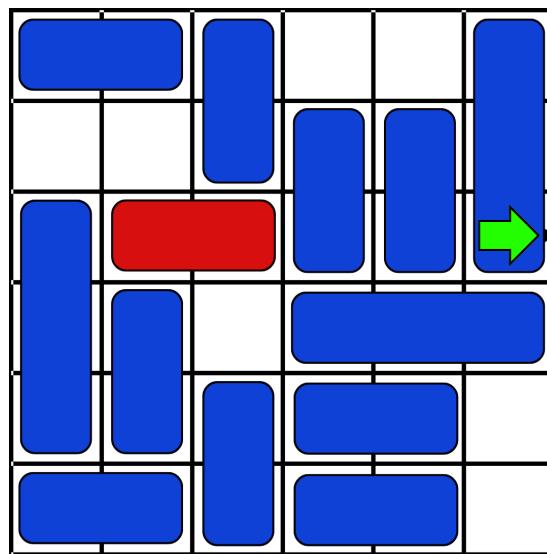
Hanya *primary piece* yang dapat digerakkan **keluar papan melewati pintu keluar**. *Piece* yang bukan *primary piece* tidak dapat digerakkan keluar papan. Papan memiliki

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.

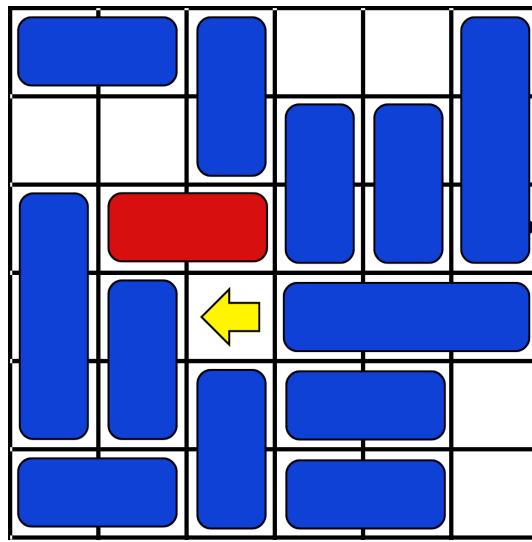
Ilustrasi kasus :

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.



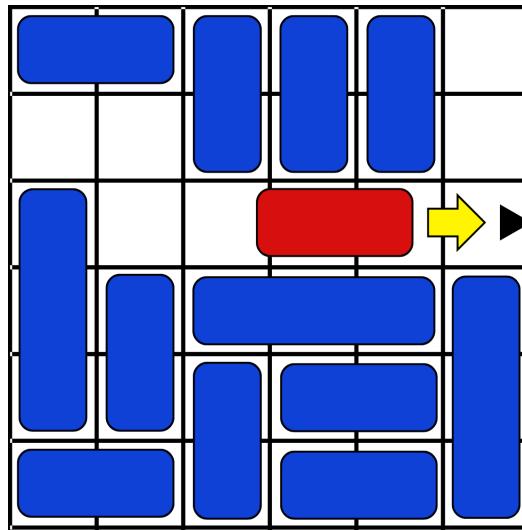
Gambar 2. Awal Permainan Game Rush Hour

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

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



Gambar 4. Pemain Menyelesaikan Permainan

Agar lebih jelas, silahkan amati video cara bermain berikut:



Anda juga dapat melihat gif berikut untuk melihat contoh permainan [Rush Hour Solution](#).

Spesifikasi Tugas Kecil 3:

- Buatlah program sederhana dalam bahasa C/C++/Java/Javascript yang mengimplementasikan **algoritma pathfinding Greedy Best First Search, UCS (Uniform Cost Search), dan A*** dalam menyelesaikan permainan Rush Hour.

- Tugas dapat dikerjakan **individu atau berkelompok** dengan anggota **maksimal 2 orang** (sangat disarankan). Boleh lintas kelas dan lintas kampus, tetapi **tidak boleh sama** dengan anggota kelompok pada **tugas kecil Strategi Algoritma sebelumnya**.
- Algoritma *pathfinding* minimal menggunakan **satu heuristic** (2 atau lebih jika mengerjakan *bonus*) yang ditentukan sendiri. Jika mengerjakan *bonus*, *heuristic* yang digunakan ditentukan berdasarkan input pengguna.
- Algoritma dijalankan secara terpisah. Algoritma yang digunakan ditentukan berdasarkan Input pengguna.

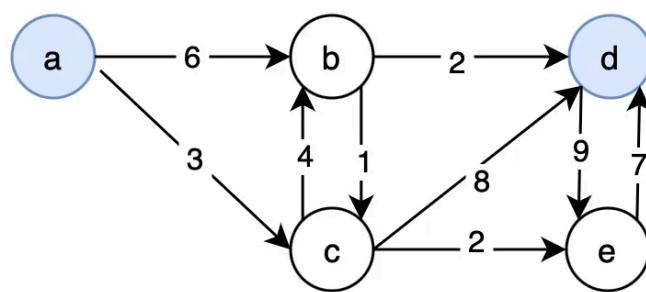
BAB II

Landasan Teori

2.1 Dasar Teori

2.1.1 Algoritma Uniform Cost Search

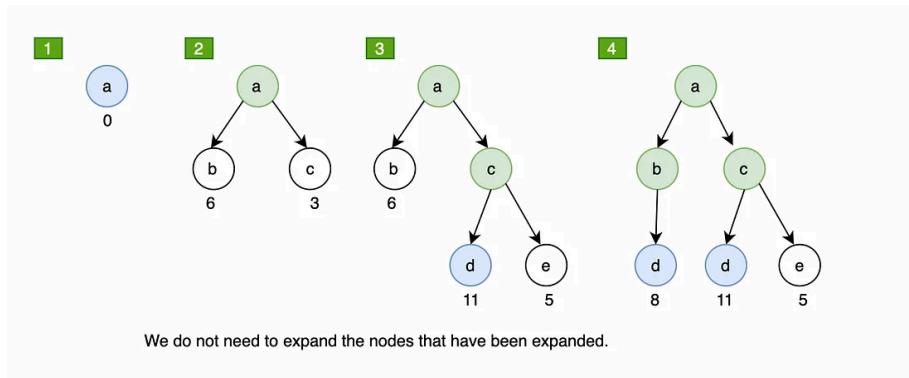
Uniform Cost Search (UCS) merupakan varian dari algoritma Dijkstra atau merupakan modifikasi dari Breadth First Search (BFS). Algoritma UCS berguna untuk mencari rute terpendek di antara dua node. Algoritma ini akan mengembangkan node dengan ongkos yang terkecil untuk sampai ke node tujuan, sehingga dibandingkan memakai queue First-In-First-Out UCS akan menggunakan priority queue dengan ongkos $g(n)$ untuk menyortir urutan node.



Gambar 5. Contoh graf untuk penelusuran UCS

Sumber:

(<https://medium.com/data-science/search-algorithm-dijkstras-algorithm-uniform-cost-search-with-python-ccbee250ba9>)



Gambar 6. Pogon Status Penelusuran UCS

Sumber:

(<https://medium.com/data-science/search-algorithm-dijkstras-algorithm-uniform-cost-search-with-python-ccbee250ba9>)

Dari ekspansi tiap node dapat dilihat bahwa ongkos path akan dikonsiderasi dan akan dikembangkan node dengan ongkos terkecil. Step dari 2 ke 3, node yang akan dikembangkan adalah c, karena memiliki ongkos path terkecil. Ketika mencapai node d mendapatkan total cost 11, namun ongkos untuk node b itu $6 < 11$, sehingga node dari b akan dikembangkan. Ketika node b mencapai d dan tidak ada node lagi yang akan ditelusuri maka, path terpendeknya adalah [a, b, d]. Kompleksitas waktu dari UCS adalah $O(m^{(1+floor(l/e))})$ dengan:

m : Jumlah maksimum tetangga yang dimiliki sebuah node

l : Panjang dari path terpendek ke tujuan node

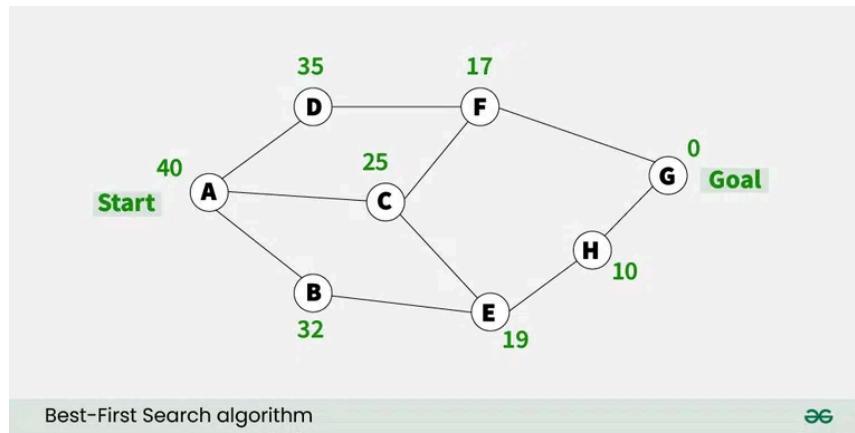
r : Ongkos terpendek dari sebuah node.

Kelebihan dari algoritma ini yaitu optimal dan lengkap jika semua biaya langkah non-negatif. Sedangkan kekurangannya adalah memerlukan banyak memori dan waktu pada graf yang besar.

2.1.2 Algoritma Greedy Best First Search

Algoritma ini bekerja dengan cara menggunakan fungsi heuristik untuk menentukan path mana yang paling bagus. Fungsi heuristik akan mempertimbangkan ongkos yang sekarang dan ongkos path sisanya. Jika cost yang sekarang lebih sedikit dari estimasi ongkos sisanya maka path yang sekarang akan dipilih. Proses ini akan diulang terus hingga tujuan tercapai. Cara kerja dari Greedy Best First Search, yaitu:

- Pencarian Greedy Best-First bekerja dengan mengevaluasi biaya setiap jalur yang mungkin dan kemudian memperluas jalur dengan biaya terendah. Proses ini diulang hingga mencapai tujuan.
- Algoritma ini menggunakan fungsi heuristik untuk menentukan jalur mana yang paling menjanjikan.
- Fungsi heuristik mempertimbangkan biaya jalur saat ini serta perkiraan biaya untuk mencapai tujuan.
- Jika biaya jalur saat ini lebih rendah dibandingkan perkiraan biaya jalur yang tersisa, maka jalur saat ini akan dipilih. Proses ini terus berlanjut hingga tujuan tercapai.



Gambar 7. Contoh graf untuk traversal Greedy Best First Search

Sumber: (<https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/>)

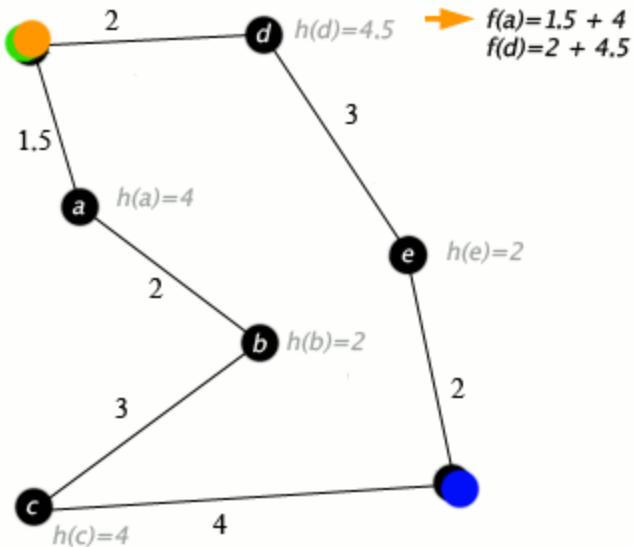
Proses dari penjelajahan ini adalah dengan pertama dimulai dari A, dan terdapat ke B (32), C (25), dan D (35). Dengan algoritma greedy maka akan akan dipilih cost yang terendah yaitu akan ke node C. Pada node C terdapat tetangga F (17) dan E (15), dengan demikian yang dipilih adalah F karena memiliki cost terkecil. Proses ini akan dilakukan terus hingga tujuan tercapai.

Kelebihan dari algoritma ini adalah cepat dalam menemukan solusi, terutama pada graf besar. Sedangkan kekurangannya adalah tidak optimal dan bisa masuk ke jalan buntu jika heuristik tidak akurat.

2.1.3 Algoritma A*

Algoritma A* (A Star) adalah algoritma pencarian yang digunakan untuk menemukan jalur terpendek antara titik awal dan akhir. Algoritma ini sering digunakan untuk penjelajahan peta guna menemukan jalur terpendek yang akan diambil. A* awalnya dirancang sebagai masalah penjelajahan graph (graph traversal), untuk membantu robot agar dapat menemukan arahnya sendiri. A* saat ini masih tetap menjadi algoritma yang sangat populer untuk graph traversal. Cara kerja algoritma A* yaitu:

- A* menggunakan Best First Search (BFS) dan menemukan jalur dengan biaya terkecil (least-cost path) dari node awal (initial node) yang diberikan ke node tujuan (goal node).
- Algoritma ini menggunakan fungsi heuristik jarak ditambah biaya (biasa dinotasikan dengan $f(x)$) untuk menentukan urutan di mana search-nya melalui node-node yang ada pada tree.
- Notasi yang dipakai oleh algoritma A* adalah $f(n) = g(n) + h(n)$ dimana $f(n)$ adalah biaya estimasi terendah $g(n)$ adalah biaya dari node awal ke node n $h(n) =$ perkiraan biaya dari node n ke node akhir



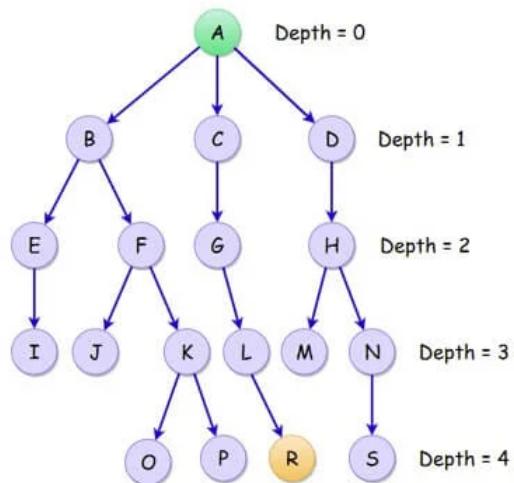
Gambar 8. Contoh graf untuk algoritma A*

Sumber: (<https://upload.wikimedia.org/>)

2.1.4 Algoritma Iterative Deepening Depth First Search

Algoritma Iterative Deepening Depth First Search (IDDFS) adalah algoritma pencarian dalam graf atau pohon yang menggabungkan kelebihan dari dua algoritma klasik: Depth First Search (DFS) dan Breadth First Search (BFS). Seperti DFS, IDDFS menggunakan sedikit memori karena hanya menyimpan jalur saat ini dalam memori (menggunakan rekursi atau stack), tetapi seperti BFS, IDDFS menjamin pencarian solusi dengan kedalaman minimum (optimal untuk pencarian tak berbobot). Algoritma ini bekerja dengan melakukan DFS secara bertahap hingga kedalaman tertentu (depth limit), kemudian mengulangi DFS dengan menambah batas kedalaman satu per satu hingga solusi ditemukan. Karena tiap level pencarian diulang dari awal, IDDFS mungkin tampak boros, namun dalam struktur seperti pohon pencarian, pengulangan ini tidak terlalu signifikan karena sebagian besar simpul berada di kedalaman terdalam. Cara kerja algoritma ini adalah:

- Tentukan batas kedalaman awal yang dimulai dari 0 dan tingkatkan secara bertahap.
- Lakukan pencarian DFS dengan Depth Limited Search (DLS), yaitu DFS yang berhenti jika mencapai kedalaman tertentu.
- Periksa apakah solusi ditemukan dalam batas kedalaman saat ini. Jika ya, hentikan dan kembalikan solusi. Jika tidak, tingkatkan batas kedalaman dan ulangi proses.
- Ulangi langkah 2 dan 3 sampai solusi ditemukan atau semua kemungkinan habis (dalam graf terbatas).



www.educba.com

Gambar 9. Contoh graf untuk algoritma Iterative Deepening Depth First Search
Sumber: (www.educba.com)

BAB III

Analisis

3.1 Analisis Algoritma

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

$g(n)$ adalah nilai yang merepresentasikan total biaya nyata yang diperlukan untuk mencapai sebuah simpul n dari simpul awal melalui jalur yang telah ditempuh sejauh ini. Nilai ini diperoleh dengan menjumlahkan seluruh biaya antar simpul pada lintasan yang sudah dilalui, tanpa melibatkan perkiraan atau asumsi. Dengan kata lain, $g(n)$ mencerminkan akumulasi dari semua langkah yang benar-benar sudah diambil dan dihitung berdasarkan informasi biaya yang tersedia secara pasti.

$h(n)$ adalah nilai yang menunjukkan estimasi atau perkiraan biaya terpendek dari simpul n menuju simpul tujuan (goal). Estimasi ini diperoleh melalui suatu fungsi heuristik yang dirancang untuk memberikan tebakan seakurat mungkin terhadap sisa perjalanan dari simpul saat ini ke tujuan. Nilai $h(n)$ tidak mencerminkan biaya sebenarnya, melainkan prediksi yang digunakan untuk membantu proses pengambilan keputusan dalam pencarian jalur. Agar dapat digunakan secara efektif, fungsi heuristik penyusun $h(n)$ idealnya bersifat admissible, yaitu tidak melebih-lebihkan estimasi biaya sebenarnya.

3.1.2 Algoritma *Uniform Cost Search*

Uniform Cost Search (UCS) adalah algoritma pencarian jalur yang selalu memilih node dengan biaya kumulatif terendah dari node awal. UCS menggunakan pendekatan best-first search berdasarkan cost-so-far, bukan kedalaman atau heuristic. UCS selalu menemukan solusi dengan biaya minimum jika semua biaya langkah non-negatif. Selain itu, UCS akan menemukan solusi jika solusi memang ada. UCS tidak menggunakan heuristic atau $h(n)$, berbeda dengan A* atau GBFS, UCS hanya mempertimbangkan cost-so-far. Bisa Terakhir, UCS dapat menangani kasus di mana setiap langkah memiliki biaya berbeda.

UCS memiliki beberapa kelebihan seperti menjamin solusi dengan cost terendah., tidak memerlukan fungsi heuristic, dan tidak akan mengeksplorasi node yang sama berulang kali. Namun, UCS juga memiliki beberapa kekurangan seperti konsumsi memori besar karena harus menyimpan semua node yang telah diekspansi dan yang ada di queue. Lalu kurang efisien untuk ruang pencarian besar. Terakhir, tidak mempercepat pencarian karena tidak ada heuristic yang membantu mengarahkan pencarian ke solusi lebih cepat.

3.1.3 Algoritma Greedy Best First Search (GBFS)

Greedy Best-First Search (GBFS) adalah algoritma pencarian jalur yang selalu memilih node yang paling dekat dengan tujuan menurut fungsi heuristic. GBFS menggunakan pendekatan best-first search berdasarkan nilai heuristic (estimasi jarak ke goal), bukan cost-so-far atau kedalaman. GBFS tidak menjamin solusi dengan cost terendah, karena hanya mempertimbangkan heuristic. Pada beberapa kasus, bisa gagal menemukan solusi jika terjebak pada jalur buntu. Selain itu, kualitas heuristic sangat mempengaruhi performa dan hasil pencarian dari GBFS, jika heuristic mendekati jarak sebenarnya ke goal, pencarian bisa sangat efisien dan sebaliknya

GBFS memiliki beberapa kelebihan seperti cocok untuk ruang pencarian besar jika heuristic efektif dan implementasi yang sederhana karena hanya perlu fungsi heuristic, tanpa memperhitungkan cost-so-far. Kekurangan dari GBFS ini bisa terjebak pada local optimum, bisa gagal menemukan solusi jika ada siklus atau jalur buntu, sangat tergantung pada heuristic, dan tidak menjamin solusi optimal.

Seperti yang sudah disebutkan sebelumnya, GBFS (Greedy Best-First Search) tidak menjamin solusi optimal karena algoritma ini hanya mempertimbangkan nilai heuristic (perkiraan jarak ke tujuan) pada setiap langkahnya, tanpa memperhitungkan biaya yang sudah dikeluarkan (cost-so-far) dari titik awal ke node saat ini. Akibatnya, GBFS bisa saja memilih jalur yang tampak paling dekat ke goal menurut heuristic, tetapi sebenarnya memiliki total biaya yang lebih besar dibanding jalur lain yang heuristic-nya lebih tinggi namun total biayanya lebih kecil. Dengan kata lain, GBFS bisa melewati solusi yang lebih murah karena hanya “terpaku” pada estimasi jarak ke goal, bukan pada total biaya perjalanan.

3.1.4 Algoritma A* Search

A* (A Star Search) adalah algoritma pencarian jalur yang menggabungkan keunggulan Uniform Cost Search (UCS) dan Greedy Best-First Search (GBFS). A* memilih node berdasarkan penjumlahan cost-so-far (biaya dari awal ke node saat ini) dan heuristic (perkiraan biaya dari node saat ini ke tujuan). A* Search menjamin solusi optimal jika heuristic yang digunakan admissible (tidak melebih-lebihkan biaya ke goal). A* juga akan menemukan solusi jika solusi memang ada dan cost langkah non-negatif. Selain itu, A* juga lebih efisien dari UCS jika heuristic mendekati biaya sebenarnya ke goal dan bisa digunakan untuk berbagai masalah pencarian jalur, baik dengan cost seragam maupun berbeda.

A* Search memiliki beberapa kekurangan konsumsi memori besar, bergantung pada heuristic dan implementasi lebih rumit karena harus mengelola dua komponen ($g(n)$ dan $h(n)$). Heuristik dikatakan admissible jika untuk setiap simpul n , nilai heuristik $h(n)$

tidak pernah melebihi biaya sebenarnya dari simpul n ke tujuan. Secara matematis, ini ditulis sebagai $h(n) \leq h^*(n)$, di mana $h^*(n)$ adalah biaya nyata minimum dari n ke goal. Dengan kata lain, heuristik admissible selalu memberikan estimasi yang optimis, karena tidak pernah memperkirakan biaya lebih tinggi dari kenyataan.

Dalam penerapannya, A* bisa dikatakan admissible dan bisa juga dikatakan tidak admissible sesuai dengan penjelasan sebelumnya. Jika dalam algoritma A* fungsi heuristik yang digunakan adalah admissible, maka A* dijamin akan menemukan jalur yang optimal (paling murah). Hal ini karena A* akan selalu memilih jalur dengan total estimasi biaya $f(n) = g(n) + h(n)$ terendah, dan jika $h(n)$ tidak pernah melebihi biaya sebenarnya, maka algoritma tidak akan "tertipu" oleh estimasi yang terlalu rendah atau terlalu tinggi.

3.1.5 Algoritma Iterative Deepening Depth First Search

Iterative Deepening Depth First Search (IDDFS) adalah algoritma pencarian yang menggabungkan kelebihan Depth First Search (DFS) dan Breadth First Search (BFS). IDDFS melakukan pencarian DFS berulang kali dengan batas kedalaman (depth limit) yang semakin besar, dimulai dari 0 hingga ditemukan solusi atau batas maksimum tercapai. IDDFS akan menemukan solusi jika solusi ada dan ruang pencarian terbatas, akan menemukan solusi dengan langkah minimum, konsumsi memori rendah, dan node pada level bawah akan dieksplorasi berulang kali pada setiap iterasi.

IDDFS memiliki beberapa kelebihan seperti konsumsi memori yang sangat kecil, dapat menemukan solusi dengan langkah minimum pada cost seragam, tidak perlu menyimpan semua node, dan cocok untuk ruang pencarian besar. Namun IDDFS memiliki beberapa kekurangan seperti redundansi eksekusi dimana node pada level bawah dieksplorasi berulang kali, tidak efisien untuk cost berbeda, dan jika solusi sangat dalam, waktu pencarian bisa sangat lama.

3.1.6 Algoritma UCS dan BFS pada Rush Hour

Pada penyelesaian Rush Hour dengan asumsi setiap langkah (move) memiliki biaya yang sama seperti pada program kami, maka Uniform Cost Search (UCS) dan Breadth-First Search (BFS) secara prinsip akan menghasilkan path yang sama, yaitu solusi dengan jumlah langkah minimum. Hal ini dikarenakan BFS mengeksplorasi semua node pada level yang sama sebelum naik ke level berikutnya, sehingga solusi pertama yang ditemukan pasti memiliki jumlah langkah minimum. UCS dengan biaya seragam akan selalu memilih node dengan cost-so-far terkecil, yang identik dengan level pada BFS.

Urutan node yang dibangkitkan juga akan sangat mirip, karena kedua algoritma akan memproses semua node pada depth tertentu sebelum melanjutkan ke depth berikutnya. Namun, secara implementasi jika ada perbedaan dalam urutan penambahan node ke queue (misal, urutan possible moves pada setiap state), urutan eksplorasi node bisa sedikit berbeda, tetapi path solusi yang ditemukan tetap sama (yaitu path terpendek). Jika ada cost langkah yang berbeda-beda, UCS akan berbeda dengan BFS dan bisa menghasilkan solusi yang berbeda (lebih optimal).

Namun, jika menggunakan asumsi cost yang tidak selalu sama untuk setiap move, maka UCS (Uniform Cost Search) tidak selalu sama dengan BFS (Breadth-First Search) dalam hal urutan node yang dibangkitkan dan path yang dihasilkan, kecuali dalam kondisi tertentu. Dalam hal ini, UCS lebih "sensitif terhadap cost" dan mungkin akan mengeksplorasi jalur yang lebih panjang tetapi lebih murah, sedangkan BFS tetap fokus ke jalur terdangkal (langkah paling sedikit), bukan paling murah.

3.1.7 Perbandingan A* dan UCS

Secara teoritis, A* lebih efisien dibandingkan UCS pada penyelesaian Rush Hour, asalkan heuristik yang digunakan admissible dan cukup informatif. Hal itu dikarenakan UCS mengeksplorasi semua node berdasarkan cost-so-far tanpa mempertimbangkan seberapa dekat node tersebut ke goal. Akibatnya, UCS bisa mengeksplorasi banyak node yang sebenarnya “jauh” dari solusi. A* menggunakan cost-so-far dan heuristic. Dengan heuristic yang baik, A* akan lebih “terarah” menuju solusi, sehingga jumlah node yang dieksplorasi lebih sedikit dibanding UCS.

BAB IV

Implementasi dan Pengujian

4.1 Source Code

4.1.1 Components

```
Piece.java

package components;

public class Piece {
    protected int col;
    protected int row;
    protected int size;
    protected int totalPiece;
    protected char orientation;
    protected char letter;
    protected boolean isPrimary;

    private static char currentLetter;

    private static final char PRIMARY_PIECE = 'P';

    public Piece(char letter, int x, int y, int size, char orientation, boolean
isPrimary) {
        this.letter = letter;
        this.col = x;
        this.row = y;
        this.size = size;
        this.orientation = orientation;
        this.totalPiece = 0;
        this.isPrimary = isPrimary;
    }

    public static Piece pieceFromBoard(char letter, int x, int y, Board board)
throws Exception {
    if (letter == '.' || letter == 'K' || letter == '|' || letter == '-') {
        return null;
    }

    currentLetter = letter;

    char orientation = pieceOrientation(letter, x, y, board);
    int size = pieceSize(letter, x, y, orientation, board);
    if(size == 1){
        throw new Exception("Piece size cannot be 1");
    }

    return new Piece(letter, x, y, size, orientation, letter == PRIMARY_PIECE);
}

private static char pieceOrientation(char letter, int x, int y, Board board) {
    boolean hasHorizontalNeighbor = false;
    if (x + 1 < board.getCols() && board.getCell(x + 1, y) == letter) {
        hasHorizontalNeighbor = true;
    }

    if (hasHorizontalNeighbor) {
        return 'H';
    } else {
        return 'V';
    }
}
```

```

    }

}

private static int pieceSize(char letter, int x, int y, char orientation, Board
board) {
    int size = 1;

    if (orientation == 'H') {
        int j = x + 1;
        while (j < board.getCols() && board.getCell(j, y) == letter) {
            size++;
            j++;
        }
    } else {
        int i = y + 1;
        while (i < board.getRows() && board.getCell(x, i) == letter) {
            size++;
            i++;
        }
    }

    return size;
}

public Piece copy() {
    return new Piece(this.letter, this.col, this.row, this.size,
this.orientation, this.isPrimary);
}

public static char getCurrentLetter() {
    return currentLetter;
}

public int getCol() {
    return col;
}

public void setCol(int x) {
    this.col = x;
}

public int getRow() {
    return row;
}

public void setRow(int y) {
    this.row = y;
}

public int getSize() {
    return size;
}

public int getTotalPiece() {
    return totalPiece;
}

public char getOrientation() {
    return orientation;
}

public boolean isPrimary() {
}

```

```

        return isPrimary;
    }

    public boolean isHorizontal() {
        return orientation == 'H';
    }

    public boolean isVertical() {
        return orientation == 'V';
    }

    @Override
    public String toString() {
        return String.valueOf(this.letter);
    }

    public void setSize(int size) {
        this.size = size;
    }

    public void setTotalPiece(int totalPiece) {
        this.totalPiece = totalPiece;
    }

    public void setOrientation(char orientation) {
        this.orientation = orientation;
    }

    public char getLetter() {
        return this.letter;
    }

    public void setLetter(char letter) {
        this.letter = letter;
    }

    public void setPrimary(boolean isPrimary) {
        this.isPrimary = isPrimary;
    }

    public static void setCurrentLetter(char currentLetter) {
        Piece.currentLetter = currentLetter;
    }

    public static char getPrimaryPiece() {
        return PRIMARY_PIECE;
    }
}

```

Move.java

```

package components;

public class Move {
    private Piece piece;
    private int startX;
    private int startY;
    private String direction;
    private int steps;
}

```

```

public Move(Piece piece, int startX, int startY, String direction, int steps) {
    if (piece == null) {
        throw new IllegalArgumentException("Piece cannot be null");
    }
    if (direction == null) {
        throw new IllegalArgumentException("Direction cannot be null");
    }
    this.piece = piece;
    this.startX = startX;
    this.startY = startY;
    this.direction = direction;
    this.steps = steps;
}

public int[] getEndPosition() {
    int endX = startX;
    int endY = startY;

    switch (direction) {
        case "UP":
            endY -= steps;
            break;
        case "DOWN":
            endY += steps;
            break;
        case "LEFT":
            endX -= steps;
            break;
        case "RIGHT":
            endX += steps;
            break;
    }

    return new int[] { endX, endY };
}

public Piece getPiece() {
    return piece;
}

public int getStartX() {
    return startX;
}

public int getStartY() {
    return startY;
}

public String getDirection() {
    return direction;
}

public int getSteps() {
    return steps;
}
}

```

Board.java

package components;

```

import java.util.*;

public class Board {
    public static final char EMPTY_GRID = '.';

    private char[][] grid;
    private int cols;
    private int rows;
    private HashMap<String, Piece> pieces;

    public void printPieces() {
        System.out.println(pieces);
    }

    public Board(char[][] grid) throws Exception {
        if (grid == null || grid.length == 0 || grid[0].length == 0) {
            throw new IllegalArgumentException("Invalid grid dimensions");
        }

        this.grid = grid;
        this.rows = grid.length;
        this.cols = grid[0].length;
        this.pieces = new HashMap<>();
        try {
            initializePieces();
        } catch (Exception e) {
            throw e;
        }
    }

    public void printBoard() {
        System.out.println("A: " + cols + " B: " + rows + " N: " + pieces.size());
        for (char[] row : grid) {
            System.out.println(Arrays.toString(row));
        }
        System.out.println("\n" + pieces.keySet());
    }

    private void initializePieces() throws Exception {
        boolean[][] visited = new boolean[rows][cols];

        try {
            for (int i = 0; i < rows; i++) {
                for (int j = 0; j < cols; j++) {
                    char cell = grid[i][j];
                    if (!visited[i][j] && cell != '.' && cell != 'K' && cell != '-' && cell != '|') {
                        Piece piece = Piece.pieceFromBoard(cell, j, i, this);
                        if (piece != null) {
                            pieces.put(String.valueOf(cell), piece);
                            markVisited(piece, visited);
                        }
                    }
                }
            }
            if (pieces.size() - 1 != IO.getN()) {
                throw new Exception("Number of piece didnt match N");
            }
        } catch (Exception e) {
            throw e;
        }
    }
}

```

```

}

private void markVisited(Piece piece, boolean[][] visited) {
    if (piece.orientation == 'H') {
        for (int j = piece.col; j < piece.col + piece.size; j++) {
            if (isValidPosition(piece.row, j)) {
                visited[piece.row][j] = true;
            }
        }
    } else {
        for (int i = piece.row; i < piece.row + piece.size; i++) {
            if (isValidPosition(i, piece.col)) {
                visited[i][piece.col] = true;
            }
        }
    }
}

private boolean isValidPosition(int row, int col) {
    return row >= 0 && row < rows && col >= 0 && col < cols;
}

public Board copy() throws Exception {
    char[][] newGrid = new char[rows][cols];
    for (int i = 0; i < rows; i++) {
        System.arraycopy(grid[i], 0, newGrid[i], 0, cols);
    }

    Board newBoard = new Board(newGrid);

    newBoard.pieces.clear();
    for (Map.Entry<String, Piece> entry : this.pieces.entrySet()) {
        newBoard.pieces.put(entry.getKey(), entry.getValue().copy());
    }

    return newBoard;
}

public boolean isValidMove(Move move) {
    int[] end = move.getEndPosition();
    int endCol = end[0], endRow = end[1];

    if (endRow < 0 || endRow >= rows || endCol < 0 || endCol >= cols) {
        return false;
    }

    Piece piece = move.getPiece();
    int startCol = move.getStartTime();
    int startRow = move.getStartTime();
    String direction = move.getDirection();

    if ((piece.getOrientation() == 'H' && (direction.equals("UP") ||
    direction.equals("DOWN")))) ||
        (piece.getOrientation() == 'V' && (direction.equals("LEFT") ||
    direction.equals("RIGHT")))) {
        return false;
    }

    if (direction.equals("UP")) {
        for (int row = startRow - 1; row >= endRow; row--) {
            if (grid[row][startCol] != '.') {
                return false;
            }
        }
    }
}

```

```

        }
    }
} else if (direction.equals("DOWN")) {
    for (int row = startRow + piece.getSize(); row <= endRow + piece.getSize() - 1; row++) {
        if (row >= rows || grid[row][startCol] != '.') {
            return false;
        }
    }
} else if (direction.equals("LEFT")) {
    for (int col = startCol - 1; col >= endCol; col--) {
        if (grid[startRow][col] != '.') {
            return false;
        }
    }
} else if (direction.equals("RIGHT")) {
    for (int col = startCol + piece.getSize(); col <= endCol + piece.getSize() - 1; col++) {
        if (col >= cols || grid[startRow][col] != '.') {
            return false;
        }
    }
}

return true;
}

public void makeMove(Move move) {
    if (!isValidMove(move)) {
        throw new IllegalArgumentException("Invalid move: " + move);
    }

    Piece piece = move.getPiece();
    int startCol = move.getX();
    int startRow = move.getY();
    int[] end = move.getEndPosition();
    int endCol = end[0];
    int endRow = end[1];

    char pieceChar = piece.getLetter();

    if (piece.getOrientation() == 'H') {
        for (int i = 0; i < piece.getSize(); i++) {
            if (startRow >= 0 && startRow < rows && startCol + i >= 0 && startCol + i < cols) {
                grid[startRow][startCol + i] = '.';
            }
        }
    } else if (piece.getOrientation() == 'V') {
        for (int i = 0; i < piece.getSize(); i++) {
            if (startRow + i >= 0 && startRow + i < rows && startCol >= 0 && startCol < cols) {
                grid[startRow + i][startCol] = '.';
            }
        }
    } else {
        if (startRow >= 0 && startRow < rows && startCol >= 0 && startCol < cols) {
            grid[startRow][startCol] = '.';
        }
    }
}

if (piece.getOrientation() == 'H') {

```

```

        for (int i = 0; i < piece.getSize(); i++) {
            if (endRow >= 0 && endRow < rows && endCol + i >= 0 && endCol + i < cols)
{
                grid[endRow] [endCol + i] = pieceChar;
            }
        }
    } else if (piece.getOrientation() == 'V') {
        for (int i = 0; i < piece.getSize(); i++) {
            if (endRow + i >= 0 && endRow + i < rows && endCol >= 0 && endCol < cols)
{
                grid[endRow + i] [endCol] = pieceChar;
            }
        }
    } else {
        if (endRow >= 0 && endRow < rows && endCol >= 0 && endCol < cols) {
            grid[endRow] [endCol] = pieceChar;
        }
    }

    piece.setCol(endCol);
    piece.setRow(endRow);
}

public List<Move> getPossibleMoves() {
    List<Move> moves = new ArrayList<>();

    for (Map.Entry<String, Piece> entry : pieces.entrySet()) {
        Piece piece = entry.getValue();

        if (piece.getOrientation() == 'H') {
            for (int steps = 1; steps <= cols; steps++) {
                if (piece.getCol() - steps < 0)
                    break;

                Move leftMove = new Move(piece, piece.getCol(), piece.getRow(), "LEFT",
steps);
                if (isValidMove(leftMove)) {
                    moves.add(leftMove);
                } else {
                    break;
                }
            }

            for (int steps = 1; steps <= cols; steps++) {
                if (piece.getCol() + piece.getSize() + steps - 1 >= cols)
                    break;

                Move rightMove = new Move(piece, piece.getCol(), piece.getRow(),
"RIGHT", steps);
                if (isValidMove(rightMove)) {
                    moves.add(rightMove);
                } else {
                    break;
                }
            }
        } else if (piece.getOrientation() == 'V') {
            for (int steps = 1; steps <= rows; steps++) {
                if (piece.getRow() - steps < 0)
                    break;

                Move upMove = new Move(piece, piece.getCol(), piece.getRow(), "UP",
steps);
            }
        }
    }
}

```

```

        if (isValidMove (upMove) ) {
            moves.add (upMove);
        } else {
            break;
        }
    }

    for (int steps = 1; steps <= rows; steps++) {
        if (piece.getRow() + piece.getSize() + steps - 1 >= rows)
            break;

        Move downMove = new Move(piece, piece.getCol(), piece.getRow(), "DOWN",
steps);
        if (isValidMove (downMove) ) {
            moves.add (downMove);
        } else {
            break;
        }
    }
}

return moves;
}

public int getRows() {
    return this.rows;
}

public int getCols() {
    return this.cols;
}

public char getCell(int x, int y) {
    if (x < 0 || x >= cols || y < 0 || y >= rows) {
        return '.';
    }

    return this.grid[y] [x];
}

public HashMap<String, Piece> getPieces() {
    return this.pieces;
}

public Piece getPrimaryPiece() {
    return pieces.get("P");
}

@Override
public boolean equals(Object o) {
    if (this == o)
        return true;
    if (o == null || getClass() != o.getClass())
        return false;
    Board board = (Board) o;
    return Arrays.deepEquals(this.grid, board.grid);
}

@Override
public int hashCode() {
    return Arrays.deepHashCode(this.grid);
}

```

```

}

@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            sb.append(grid[i][j]);
        }
    }
    return sb.toString();
}

public char[][] getGrid() {
    return grid;
}

public void setGrid(char[][] grid) {
    this.grid = grid;
}

public void setCols(int cols) {
    this.cols = cols;
}

public void setRows(int rows) {
    this.rows = rows;
}

public void setPieces(HashMap<String, Piece> pieces) {
    this.pieces = pieces;
}

public static char getEmptyGrid() {
    return EMPTY_GRID;
}
}

```

State.java

```

package components;

import java.util.*;

public class State implements Comparable<State> {
    private Board board;
    private int costSoFar;
    private int heuristic;
    private State parent;
    private Move move;
    private int totalNodeVisited;
    private long executionTime;
    private String algorithm;
    private String heuristicType;

    public State(Board board, int costSoFar, int heuristic, State parent, Move
move, int totalNodeVisited, long executionTime, String algorithm, String
heuristicType) {

```

```

heuristicType) {
    this.board = board;
    this.costSoFar = costSoFar;
    this.heuristic = heuristic;
    this.parent = parent;
    this.move = move;
    this.totalNodeVisited = totalNodeVisited;
    this.executionTime = executionTime;
    this.algorithm = algorithm;
    this.heuristicType = heuristicType;
}

public State(Board board, int costSoFar, int heuristic, State parent, Move
move, int totalNodeVisited) {
    this(board, costSoFar, heuristic, parent, move, totalNodeVisited, 0, "", "");
}

public String getHeuristicType() {
    return heuristicType;
}

public void setHeuristicType(String heuristicType) {
    this.heuristicType = heuristicType;
}

public int getTotalNodeVisited() {
    return totalNodeVisited;
}

public void setTotalNodeVisited(int totalNodeVisited) {
    this.totalNodeVisited = totalNodeVisited;
}

public int getF() {
    return costSoFar + heuristic;
}

public int compareTo(State other) {
    return Integer.compare(this.getF(), other.getF());
}

@Override
public boolean equals(Object o) {
    if (!(o instanceof State))
        return false;
    State other = (State) o;
    if (this.board == null || other.board == null)
        return false;
    return board.equals(other.board);
}

@Override
public int hashCode() {
    return board.hashCode();
}

public List<Move> getPathFromRoot() {
    List<Move> path = new ArrayList<>();
    State current = this;
    while (current.parent != null) {
        path.add(current.move);
    }
}

```

```

        current = current.parent;
    }
    Collections.reverse(path);
    return path;
}

public String[] getSolutionPath() {    List<String> steps = new ArrayList<>();
    State current = this;
    String heuristicType = (current.heuristicType == "") ? "-" :
current.heuristicType;
    String separator = "=" .repeat(50);
    String solutionTitle = String.format(
        "%s\n" +
        "          RUSH HOUR PUZZLE SOLUTION\n" +
        "%s\n" +
        "Algorithm      : %s\n" +
        "Heuristic       : %s\n" +
        "Nodes Visited   : %d\n" +
        "Time Taken      : %d ms\n" +
        "%s\n",
        separator,
        separator,
        current.algorithm,
        heuristicType,
        current.totalNodeVisited,
        current.executionTime,
        separator
    );
    steps.add(solutionTitle);

    List<State> path = new ArrayList<>();
    while (current != null) {
        path.add(current);
        current = current.parent;
    }
    Collections.reverse(path);

    for (State state : path) {
        int index = state.getCostSoFar() + 1;
        StringBuilder sb = new StringBuilder();
        sb.append(String.format("%d. ", index));

        Move move = state.getMove();
        if (move != null) {

            sb.append(String.format("Move piece %c %s by %d steps",
                move.getPiece().getLetter(),
                move.getDirection(),
                move.getSteps()));
        } else {
            sb.append("Initial State");
        }

        sb.append("\nHeuristic: ") .append(state.getHeuristic());
        sb.append("\nBoard:\n");

        for (int i = 0; i < state.getBoard().getRows(); i++) {
            for (int j = 0; j < state.getBoard().getCols(); j++) {
                sb.append(state.getBoard().getCell(j, i));
            }
            sb.append("\n");
        }
    }
}

```

```

        steps.add(sb.toString());
    }

    return steps.toArray(new String[0]);
}

public boolean isWin() {
    if (board == null) {
        return false;
    }

    Piece primaryPiece = board.getPieces().get("P");
    if (primaryPiece == null) {
        return false;
    }

    int kRow = IO.getKRow();
    int kCol = IO.getKCol();

    boolean kOnTopBorder = (kRow == 0);
    boolean kOnBottomBorder = (kRow == board.getRows() - 1);
    boolean kOnLeftBorder = (kCol == 0);
    boolean kOnRightBorder = (kCol == board.getCols() - 1);

    boolean kAboveBoard = (kRow == -1);
    boolean kBelowBoard = (kRow == board.getRows());
    boolean kLeftOfBoard = (kCol == -1);
    boolean kRightOfBoard = (kCol == board.getCols());

    if (primaryPiece.getOrientation() == 'H') {
        int pieceStart = primaryPiece.getCol();
        int pieceEnd = pieceStart + primaryPiece.getSize() - 1;
        int pieceRow = primaryPiece.getRow();

        if (kCol == -1) {
            return (pieceStart == 0) && (pieceRow == kRow);
        }

        if (kAboveBoard || kOnTopBorder) {
            return (pieceRow == 0) && (kCol >= pieceStart && kCol <= pieceEnd);
        } else if (kBelowBoard || kOnBottomBorder) {
            return (pieceRow == board.getRows() - 1) && (kCol >= pieceStart && kCol <= pieceEnd);
        } else if (kLeftOfBoard || kOnLeftBorder) {
            return (pieceStart == 0) && (pieceRow == kRow);
        } else if (kRightOfBoard || kOnRightBorder) {
            return (pieceEnd == board.getCols() - 1) && (pieceRow == kRow);
        }
    } else if (primaryPiece.getOrientation() == 'V') {
        int pieceStart = primaryPiece.getRow();
        int pieceEnd = pieceStart + primaryPiece.getSize() - 1;
        int pieceCol = primaryPiece.getCol();

        if (kAboveBoard || kOnTopBorder) {
            return (pieceStart == 0) && (pieceCol == kCol);
        } else if (kBelowBoard || kOnBottomBorder) {
            return (pieceEnd == board.getRows() - 1) && (pieceCol == kCol);
        } else if (kLeftOfBoard || kOnLeftBorder) {
            return (pieceCol == 0) && (kRow >= pieceStart && kRow <= pieceEnd);
        } else if (kRightOfBoard || kOnRightBorder) {
            return (pieceCol == board.getCols() - 1) && (kRow >= pieceStart && kRow <= pieceEnd);
        }
    }
}

```

```

<= pieceEnd);
    }
}

return false;
}

public void printState() {
    for (int i = 0; i < board.getRows(); i++) {
        if (i > 0) {
            System.out.println("");
        }
        for (int j = 0; j < board.getCols(); j++) {
            System.out.print(board.getCell(j, i));
        }
    }
}

public Board getBoard() {
    return board;
}

public int getCostSoFar() {
    return costSoFar;
}

public int getHeuristic() {
    return heuristic;
}

public State getParent() {
    return parent;
}

public Move getMove() {
    return move;
}

public void setBoard(Board board) {
    this.board = board;
}

public void setCostSoFar(int costSoFar) {
    this.costSoFar = costSoFar;
}

public void setHeuristic(int heuristic) {
    this.heuristic = heuristic;
}

public void setParent(State parent) {
    this.parent = parent;
}

public void setMove(Move move) {
    this.move = move;
}

public long getExecutionTime() {
    return executionTime;
}

```

```

public void setExecutionTime(long executionTime) {
    this.executionTime = executionTime;
}

public String getAlgorithm() {
    return algorithm;
}

public void setAlgorithm(String algorithm) {
    this.algorithm = algorithm;
}

```

IO.java

```

package components;

import java.io.*;
import java.util.*;

public class IO {
    private static int kRow;
    private static int kCol;
    private static int gridRow;
    private static int gridCol;
    private static int a;
    private static int b;
    private static int n;

    public static String[] readFile(String filepath) throws Exception {
        File file = new File(filepath);
        Scanner scanner = new Scanner(file);
        String size = "";
        String N = "";
        List<String> boardRows = new ArrayList<>();

        try {
            if (scanner.hasNextLine()) {
                size = scanner.nextLine();
                if (scanner.hasNextLine()) {
                    N = scanner.nextLine();
                    if (Integer.parseInt(N) >= 25) {
                        scanner.close();
                        throw new Exception("Number of piece cannot exceed 25");
                    }
                } else {
                    scanner.close();
                    throw new Exception("No N value found.");
                }
            } else {
                scanner.close();
                throw new Exception("Line is empty.");
            }

            while (scanner.hasNextLine()) {
                String row = scanner.nextLine();
                if (!row.isEmpty()) {
                    boardRows.add(row);
                }
            }
        } catch (Exception e) {

```

```

        throw e;
    }

    scanner.close();
    String[] result = new String[boardRows.size() + 2];
    result[0] = size;
    result[1] = N;

    for (int i = 0; i < boardRows.size(); i++) {
        result[i + 2] = boardRows.get(i);
    }

    return result;
}

private static int countOccurrences(String str, char ch) {
    int count = 0;
    for (int i = 0; i < str.length(); i++) {
        if (str.charAt(i) == ch) {
            count++;
        }
    }
    return count;
}

public static Board parseInput(String[] inputString) throws Exception {
    char[][] grid = null;
    char[][] innerGrid = null;
    try {
        String[] sizeStr = inputString[0].split(" ");
        String[] boardRows = new String[inputString.length - 2];
        for (int i = 2; i < inputString.length; i++) {
            boardRows[i - 2] = inputString[i];
        }
        a = Integer.parseInt(sizeStr[0]);
        b = Integer.parseInt(sizeStr[1]);
        gridRow = a - 1;
        gridCol = b - 1;
        if (a < 0 || b < 0)
            throw new IllegalArgumentException("Grid dimensions must be
non-negative.");
        n = Integer.parseInt(inputString[1]);
        System.out.println("A: " + a + " B: " + b + " N: " + n);

        int borderRow = a + 2;
        int borderCol = b + 2;
        grid = new char[borderCol][borderRow];
        for (String string : boardRows) {
            System.out.println(string);
        }
        boolean kFound = false;
        int kRowPosition = -1;
        int kColPosition = -1;

        int totalKCount = 0;
        for (String row : boardRows) {
            totalKCount += countOccurrences(row, 'K');
        }

        if (totalKCount > 1) {
            throw new IllegalArgumentException("Multiple 'K' characters
")
        }
    }
}

```

```

        found in the grid.");
    }

    if (boardRows[0].contains("K")) {
        if (kFound)
            throw new IllegalArgumentException("Multiple 'K' characters
found in the grid.");

        kFound = true;
        int leadingSpaces = boardRows[0].indexOf('K');
        System.out.println("Board Rows: " + Arrays.toString(boardRows));
        if (countOccurrences(boardRows[0], 'K') > 1) {
            throw new IllegalArgumentException(
                "The exit 'K' should be a single character, not a
multi-character piece.");
        }

        kRowPosition = -1;
        kColPosition = leadingSpaces;

        kRow = kRowPosition;
        kCol = kColPosition;

        List<String> rowsList = new
ArrayList<>(Arrays.asList(boardRows));
        rowsList.remove(0);
        boardRows = rowsList.toArray(new String[0]);
    }

    if (boardRows[boardRows.length - 1].contains("K")) {
        if (kFound)
            throw new IllegalArgumentException("Multiple 'K' characters
found in the grid.");

        kFound = true;
        int leadingSpaces = boardRows[boardRows.length -
1].indexOf('K');
        System.out.println("Board Rows: " + Arrays.toString(boardRows));
        if (countOccurrences(boardRows[boardRows.length - 1], 'K') > 1)
{
            throw new IllegalArgumentException(
                "The exit 'K' should be a single character, not a
multi-character piece.");
        }

        kRowPosition = a;
        kColPosition = leadingSpaces;

        kRow = a;
        kCol = kColPosition;

        List<String> rowsList = new
ArrayList<>(Arrays.asList(boardRows));
        rowsList.remove(boardRows.length - 1);
        boardRows = rowsList.toArray(new String[0]);
    }

    if (boardRows.length != a) {
        throw new IllegalArgumentException("Number of rows (" +
boardRows.length
                + ") does not match the specified dimension (" + a +
") .");
    }
}

```

```

    }

    for (int i = 0; i < borderRow; i++) {
        for (int j = 0; j < borderCol; j++) {
            if (i == 0 || i == borderRow - 1) {
                grid[i][j] = '-';
            } else if (j == 0 || j == borderCol - 1) {
                grid[i][j] = '|';
            } else {
                grid[i][j] = '.';
            }
        }
    }

    for (int i = 0; i < boardRows.length; i++) {
        int kIndex = boardRows[i].indexOf('K');
        if (kIndex != -1) {
            if (!kFound)
                throw new IllegalArgumentException("Multiple 'K' characters found in the grid.");
            if (kIndex + 1 < boardRows[i].length() &&
                boardRows[i].charAt(kIndex + 1) == 'K') {
                throw new IllegalArgumentException(
                    "The exit 'K' should be a single character, not a multi-character piece.");
            }

            if (i > 0 && kIndex < boardRows[i - 1].length() &&
                boardRows[i - 1].charAt(kIndex) == 'K') {
                throw new IllegalArgumentException(
                    "The exit 'K' should be a single character, not a multi-character piece.");
            }

            if (i < boardRows.length - 1 && kIndex < boardRows[i + 1].length()
                && boardRows[i + 1].charAt(kIndex) == 'K') {
                throw new IllegalArgumentException(
                    "The exit 'K' should be a single character, not a multi-character piece.");
            }

            kRowPosition = i;
            kColPosition = kIndex;
            kRow = kRowPosition;
            kCol = kColPosition;
            boardRows[i] = boardRows[i].substring(0, kIndex) +
                boardRows[i].substring(kIndex + 1);
            kFound = true;
            break;
        }
    }

    if (!kFound)
        throw new IllegalArgumentException("No exit 'K' found in the grid.");
}

for (int i = 0; i < a; i++) {
    for (int j = 0; j < b; j++) {
        if (i < boardRows.length && j < boardRows[i].length())
            grid[i + 1][j + 1] = boardRows[i].charAt(j);
        else
            grid[i + 1][j + 1] = '.';
    }
}

```

```

        }
    }

    if (kRowPosition == -1) {
        if (kColPosition >= 0 && kColPosition < b) {
            grid[0][kColPosition + 1] = 'K';
            System.out.println("Placed K on top border at column " +
(kColPosition + 1));
        } else {
            throw new IllegalArgumentException(
                "K position on top border is outside valid range: " +
+ kColPosition);
        }
    } else if (kRowPosition == a) {
        if (kColPosition >= 0 && kColPosition < b) {
            grid[borderRow - 1][kColPosition + 1] = 'K';
            System.out.println("Placed K on bottom border at column " +
(kColPosition + 1));
        } else {
            throw new IllegalArgumentException(
                "K position on bottom border is outside valid range: " +
+ kColPosition);
        }
    } else if (kColPosition == 0) {
        grid[kRowPosition + 1][0] = 'K';
        System.out.println("Placed K on left border at row " +
(kRowPosition + 1));
    } else if (kColPosition == b - 1
        || (kRowPosition < boardRows.length && kColPosition >=
boardRows[kRowPosition].length())) {
        grid[kRowPosition + 1][borderCol - 1] = 'K';
        System.out.println("Placed K on right border at row " +
(kRowPosition + 1));
    } else {
        throw new IllegalArgumentException(
            "Exit 'K' must be at the edge of the grid: row=" +
kRowPosition + ", col=" + kColPosition);
    }

    innerGrid = new char[a][b];
    for (int i = 0; i < a; i++) {
        for (int j = 0; j < b; j++) {
            innerGrid[i][j] = grid[i + 1][j + 1];
        }
    }

} catch (Exception e) {
    throw new IOException("Failed to parse input: " + e.getMessage());
}
System.out.printf("KCOL: %d, KROW: %d\n", IO.getKCol(), IO.getKRow());

return new Board(innerGrid);
}

public static void printGrid(char[][] grid) {
    for (char[] row : grid) {
        System.out.println(Arrays.toString(row));
    }
}

public static boolean isFileExists(String path) throws IOException {

```

```

        File file = new File(path);
        return file.exists();
    }

    public static void writeOutputToFile(String[] output, String outputPath)
throws IOException {
    FileWriter fileWriter = new FileWriter(outputPath);
    PrintWriter printWriter = new PrintWriter(fileWriter);

    for (int i = 0; i < output.length; i++) {
        printWriter.print(output[i]);
        printWriter.println();
    }

    printWriter.close();
}

public static void saveOutputToFile(String[] output, List<Piece> pieces)
throws IOException {
    Scanner scanner = new Scanner(System.in);
    String directoryPath = "test\\output\\";
    String outputPath = "";
    outputPath = outputPath.concat(directoryPath);

    while (true) {
        System.out.println("\nSave Hasil ke file?(Y/n)");
        char save = scanner.nextLine().charAt(0);
        if (save == 'Y' || save == 'y') {
            System.out.println("Masukkan nama file e.g (output): ");
            String filename = scanner.nextLine();
            outputPath = outputPath.concat(filename);

            boolean isFileExists = isFileExists(outputPath);
            if (isFileExists) {
                System.out.println("Terdapat file dengan nama yang sama
apakah anda ingin overwrite? (y/N)");
                char overwrite = scanner.next().charAt(0);
                if (overwrite == 'Y' || overwrite == 'y') {
                    writeOutputToFile(output, outputPath);
                    System.out.println("File berhasil di simpan pada " +
outputPath);
                    break;
                } else if (overwrite == 'n' || overwrite == 'N') {
                    System.out.println("File tidak disimpan\n");
                    break;
                } else {
                    System.out.println("Masukkan 'Y' untuk ya atau 'n' untuk
tidak\n");
                    break;
                }
            } else {
                writeOutputToFile(output, outputPath);
                System.out.println("File berhasil di simpan pada " +
outputPath);
            }
        }

        break;
    }

    else if (save == 'n' || save == 'N') {
        System.out.println("File tidak disimpan.\n");
        break;
    }
}

```

```

        } else {
            System.out.println("Masukkan 'Y' untuk ya atau 'n' untuk
tidak\n");
            break;
        }
    }
    scanner.close();
}

public static int getKRow() {
    return kRow;
}

public static int getKCol() {
    return kCol;
}

public static int getGridRow() {
    return gridRow;
}

public static int getGridCol() {
    return gridCol;
}

public static int getkRow() {
    return kRow;
}

public static void setkRow(int kRow) {
    IO.kRow = kRow;
}

public static int getkCol() {
    return kCol;
}

public static void setkCol(int kCol) {
    IO.kCol = kCol;
}

public static void setGridRow(int gridRow) {
    IO.gridRow = gridRow;
}

public static void setGridCol(int gridCol) {
    IO.gridCol = gridCol;
}

public static int getA() {
    return a;
}

public static void setA(int a) {
    IO.a = a;
}

public static int getB() {
    return b;
}

public static void setB(int b) {
}

```

```

        IO.b = b;
    }

    public static int getN() {
        return n;
    }

    public static void setN(int n) {
        IO.n = n;
    }
}

```

4.1.2 Solver

UniformCostSearch.java

```

package solver;

import components.*;
import java.util.*;

public class UniformCostSearch {
    private PriorityQueue<State> queue;
    private HashSet<String> visited;
    private int nodesExpanded;

    public UniformCostSearch() {
        this.queue = new
PriorityQueue<>(Comparator.comparingInt(State::getCostSoFar));
        this.visited = new HashSet<>();
        this.nodesExpanded = 0;
    }

    public State solve(Board initialBoard) throws Exception {
        queue.clear();
        visited.clear();
        nodesExpanded = 0;

        State initialState = new State(initialBoard, 0, 0, null, null, 0);
        queue.add(initialState);

        while (!queue.isEmpty()) {
            State current = queue.poll();
            Board currentBoard = current.getBoard();

            if (current != null && current.isWin()) {
                return current;
            }

            String boardHash = currentBoard.toString();
            if (visited.contains(boardHash)) {
                continue;
            }
            visited.add(boardHash);
            nodesExpanded++;

            for (Move move : current.getBoard().getPossibleMoves()) {
                Board newBoard = current.getBoard().copy();
                Piece movedPiece =
newBoard.getPieces().get(String.valueOf(move.getPiece().getLetter()));

```

```

        Move newMove = new Move(
            movedPiece,
            move.getStartTime(),
            move.getStartTime(),
            move.getDirection(),
            move.getSteps());
        newBoard.makeMove(newMove);

        String newBoardHash = newBoard.toString();
        if (visited.contains(newBoardHash)) {
            continue;
        }
        State newState = new State(
            newBoard,
            current.getCostSoFar() + 1,
            0,
            current,
            newMove,
            nodesExpanded);
        queue.add(newState);
    }
}

return null;
}

public int getNodesExpanded() {
    return nodesExpanded;
}
}

```

Heuristic.java

```

package solver;

import components.*;

public class Heuristic {
    public static int pieceToDest(Board board) {
        int exitRow = IO.getKRow();
        int exitCol = IO.getKCol();
        int boardRows = board.getRows();
        int boardCols = board.getCols();

        Piece primary = board.getPieces().get("P");
        if (primary == null)
            return 0;

        int distance = 0;

        boolean exitOnTop = (exitRow == -1);
        boolean exitOnBottom = (exitRow == boardRows);
        boolean exitOnLeft = (exitCol == -1);
        boolean exitOnRight = (exitCol == boardCols);

        if (primary.getOrientation() == 'H') {
            int pieceRow = primary.getRow();
            int pieceStartCol = primary.getCol();
            int pieceEndCol = pieceStartCol + primary.getSize() - 1;

```

```

        if (exitOnRight) {
            distance = Math.max(0, boardCols - 1 - pieceEndCol);
        } else if (exitOnLeft) {
            distance = pieceStartCol;
        } else if ((exitOnTop || exitOnBottom) &&
                   (exitCol >= 0 && exitCol < boardCols) &&
                   (exitCol >= pieceStartCol && exitCol <= pieceEndCol)) {
            distance = 0;
        } else {
            int targetCol = Math.min(Math.max(exitCol, 0), boardCols - 1);
            distance = Math.abs(targetCol - pieceEndCol)
                + Math.abs(Math.min(Math.max(exitRow, 0), boardRows - 1) - pieceRow);
        }
    } else {
        int pieceCol = primary.getCol();
        int pieceStartRow = primary.getRow();
        int pieceEndRow = pieceStartRow + primary.getSize() - 1;

        if (exitOnBottom) {
            distance = Math.max(0, boardRows - 1 - pieceEndRow);
        } else if (exitOnTop) {
            distance = pieceStartRow;
        } else if ((exitOnLeft || exitOnRight) &&
                   (exitRow >= 0 && exitRow < boardRows) &&
                   (exitRow >= pieceStartRow && exitRow <= pieceEndRow)) {
            distance = 0;
        } else {
            int targetRow = Math.min(Math.max(exitRow, 0), boardRows - 1);
            distance = Math.abs(targetRow - pieceEndRow)
                + Math.abs(Math.min(Math.max(exitCol, 0), boardCols - 1) - pieceCol);
        }
    }

    return distance;
}

public static int countBlockingPieces(Board board) {
    int exitRow = IO.getKRow();
    int exitCol = IO.getKCol();
    int boardRows = board.getRows();
    int boardCols = board.getColumns();
    int count = 0;

    Piece primary = board.getPieces().get("P");
    if (primary == null)
        return 0;

    boolean exitOnTop = (exitRow == -1);
    boolean exitOnBottom = (exitRow == boardRows);
    boolean exitOnLeft = (exitCol == -1);
    boolean exitOnRight = (exitCol == boardCols);

    if (primary.getOrientation() == 'H') {
        int row = primary.getRow();
        int pieceStartCol = primary.getCol();
        int pieceEndCol = pieceStartCol + primary.getSize() - 1;

        if (exitOnRight) {
            for (int col = pieceEndCol + 1; col < boardCols; col++) {
                if (isBlockingPiece(board, row, col)) {
                    count++;
                }
            }
        }
    }
}

```

```

        }
    } else if (exitOnLeft) {
        for (int col = 0; col < pieceStartCol; col++) {
            if (isBlockingPiece(board, row, col)) {
                count++;
            }
        }
    } else {
        int col = primary.getCol();
        int pieceStartRow = primary.getRow();
        int pieceEndRow = pieceStartRow + primary.getSize() - 1;

        if (exitOnBottom) {
            for (int row = pieceEndRow + 1; row < boardRows; row++) {
                if (isBlockingPiece(board, row, col)) {
                    count++;
                }
            }
        } else if (exitOnTop) {
            for (int row = 0; row < pieceStartRow; row++) {
                if (isBlockingPiece(board, row, col)) {
                    count++;
                }
            }
        }
    }

    return count;
}

private static boolean isBlockingPiece(Board board, int row, int col) {
    char cell = board.getCell(col, row);
    return cell != '.' && cell != 'K' && cell != '-' && cell != '|';
}

public static int combineTwo(Board board) {
    return pieceToDest(board) + countBlockingPieces(board);
}

public static int getHeuristic(Board board, String heuristicType) {
    if ("countBlockingPieces".equalsIgnoreCase(heuristicType)) {
        return countBlockingPieces(board);
    } else if ("pieceToDest".equalsIgnoreCase(heuristicType)) {
        return pieceToDest(board);
    } else {
        return combineTwo(board);
    }
}
}

```

GreedyBestFirstSearch.java

```

package solver;

import components.*;
import java.util.*;

public class GreedyBestFirstSearch {
    public State solve(Board initialBoard, String type) throws Exception {

```

```

PriorityQueue<State> queue = new
PriorityQueue<>(Comparator.comparingInt(State::getHeuristic));
HashSet<String> visited = new HashSet<>();
int nodesExpanded = 0;

int initialHeuristic = Heuristic.getHeuristic(initialBoard, type);
State initialState = new State(initialBoard, 0, initialHeuristic, null, null,
0);
queue.add(initialState);

while (!queue.isEmpty()) {
    State current = queue.poll();
    Board currentBoard = current.getBoard();

    if (current != null && current.isWin()) {
        return current;
    }

    String boardHash = currentBoard.toString();
    if (visited.contains(boardHash)) {
        continue;
    }
    visited.add(boardHash);
    nodesExpanded++;

    System.out.println("Jumlah node:" + nodesExpanded);

    for (Move move : currentBoard.getPossibleMoves()) {
        Board newBoard = currentBoard.copy();
        Piece movedPiece =
newBoard.getPieces().get(String.valueOf(move.getPiece().getLetter()));
        Move newMove = new Move(
            movedPiece,
            move.getStartX(),
            move.getStartY(),
            move.getDirection(),
            move.getSteps());
        newBoard.makeMove(newMove);

        String newBoardHash = newBoard.toString();
        if (visited.contains(newBoardHash)) {
            continue;
        }

        int heuristic = Heuristic.getHeuristic(newBoard, type);
        State newState = new State(
            newBoard,
            current.getCostSoFar() + 1,
            heuristic,
            current,
            newMove,
            nodesExpanded);
        queue.add(newState);
    }
}
return null;
}
}

```

AStar.java

```

package solver;

import java.util.*;
import components.*;

public class AStar {

    public static State solve(Board initialBoard, String heuristicType) throws
Exception {
        PriorityQueue<State> openList = new PriorityQueue<>();
        Set<String> closedList = new HashSet<>();

        int initialHeuristic = Heuristic.getHeuristic(initialBoard,
heuristicType);
        State initialState = new State(initialBoard, 0, initialHeuristic, null,
null, 0);
        openList.add(initialState);
        int statesExplored = 0;

        while (!openList.isEmpty()) {
            State current = openList.poll();
            statesExplored++;

            if (current.isWin()) {
                System.out.println("Solution found after exploring " +
statesExplored + " states!");
                current.getBoard().printBoard();
                return current;
            }

            String boardHash = current.getBoard().toString();
            if (closedList.contains(boardHash)) {
                continue;
            }
            closedList.add(boardHash);

            List<Move> possibleMoves = current.getBoard().getPossibleMoves();
            for (Move move : possibleMoves) {
                Board nextBoard = current.getBoard().copy();
                Piece movedPiece =
nextBoard.getPieces().get(String.valueOf(move.getPiece().getLetter()));

                Move newMove = new Move(
                    movedPiece,
                    move.getStartX(),
                    move.getStartY(),
                    move.getDirection(),
                    move.getSteps());
                try {
                    nextBoard.makeMove(newMove);

                    int heuristic = Heuristic.getHeuristic(nextBoard,
heuristicType);
                    State nextState = new State(nextBoard, current.getCostSoFar() +
1, heuristic, current, move,
                        statesExplored);

                    String nextBoardHash = nextBoard.toString();
                    if (!closedList.contains(nextBoardHash)) {
                        openList.add(nextState);
                    }
                } catch (IllegalArgumentException e) {

```

```

        System.err.println("Warning: " + e.getMessage());
    }
}

System.out.println("No solution found after exploring " + statesExplored
+ " states.");
return null;
}
}

```

IterativeDeepeningSearch.java

```

package solver;

import components.*;
import java.util.*;

public class IterativeDeepeningSearch {
    private int nodesExpanded;
    private int currentTotalVisited;

    public IterativeDeepeningSearch() {
        this.nodesExpanded = 0;
    }

    public State solve(Board initialBoard) throws Exception {
        int maxDepth = 0;
        currentTotalVisited = 0;

        while (true) {
            nodesExpanded = 0;
            State result = depthLimitedSearch(new State(initialBoard, 0, 0, null, null,
currentTotalVisited), maxDepth);

            if (result != null) {
                return result;
            }
            maxDepth++;
        }
    }

    private State depthLimitedSearch(State current, int depthLimit) throws
Exception {
        Stack<State> stack = new Stack<>();
        Set<String> visited = new HashSet<>();
        stack.push(current);

        while (!stack.isEmpty()) {
            State state = stack.pop();
            nodesExpanded++;

            if (state.isWin()) {
                return state;
            }

            if (state.getCostSoFar() < depthLimit) {
                for (Move move : state.getBoard().getPossibleMoves()) {
                    Board newBoard = state.getBoard().copy();
                    Piece movedPiece =

```

```

newBoard.getPieces().get(String.valueOf(move.getPiece().getLetter()));

        Move newMove = new Move(
            movedPiece,
            move.getStartX(),
            move.getStartY(),
            move.getDirection(),
            move.getSteps());
        newBoard.makeMove(newMove);
        String newBoardHash = newBoard.toString();
        if (!visited.contains(newBoardHash)) {
            visited.add(newBoardHash);
            State newState = new State(newBoard, state.getCostSoFar() + 1, 0,
state, move,
                currentTotalVisited + nodesExpanded);
            stack.push(newState);
        }
    }
}

return null;
}

public int getNodesExpanded() {
    return nodesExpanded;
}
}

```

4.1.3 GUI (Swing)

MainGUI.java

```

import components.*;
import solver.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class MainGUI extends JFrame {
    private Board board;
    private JPanel boardPanel;
    private JLabel statusLabel;
    private JLabel timeLabel;
    private JLabel nodesVisitedLabel;
    private JComboBox<String> algoBox;
    private JComboBox<String> heuristicBox;
    private JButton solveButton, loadButton, saveButton;
    private File currentFile;
    private Timer animationTimer;
    private List<State> solutionStates;
    private int currentStateIndex;
    private JButton playButton;
    private JButton nextButton;

```

```

private JButton prevButton;
private JList<String> movesList;
private DefaultListModel<String> movesListModel;
private static final int ANIMATION_DELAY = 500;

public static void main(String[] args) {
    SwingUtilities.invokeLater(MainGUI::new);
}

public MainGUI() {
    setTitle("Rush Hour Solver");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLayout(new BorderLayout());

    JPanel topPanel = new JPanel();
    loadButton = new JButton("Load Board");
    loadButton.addActionListener(e -> loadBoard());
    topPanel.add(loadButton);

    saveButton = new JButton("Save Solution");

    saveButton.addActionListener(e -> saveSolutionToFile());
    saveButton.setEnabled(false);
    topPanel.add(saveButton);

    algoBox = new JComboBox<>(new String[] {
        "Uniform Cost Search", "Greedy Best First Search", "A*", "Iterative
Deepening DFS"
    });
    topPanel.add(algoBox);

    heuristicBox = new JComboBox<>(new String[] {
        "Jarak Piece ke K", "Jumlah Piece Penghalang", "Gabungan Dua Heuristic"
    });
    heuristicBox.setEnabled(false);
    topPanel.add(heuristicBox);

    algoBox.addActionListener(e -> {
        String selectedAlgo = (String) algoBox.getSelectedItem();
        if (selectedAlgo.equals("Greedy Best First Search") ||
selectedAlgo.equals("A*")) {
            heuristicBox.setEnabled(true);
        } else {
            heuristicBox.setEnabled(false);
        }
    });
    solveButton = new JButton("Solve");
    solveButton.addActionListener(e -> solveBoard());
    solveButton.setEnabled(false);
    topPanel.add(solveButton);

    add(topPanel, BorderLayout.NORTH);
    boardPanel = new JPanel();
    add(boardPanel, BorderLayout.CENTER);
    JPanel centerPanel = new JPanel(new BorderLayout());

    JPanel boardWrapper = new JPanel(new GridBagLayout());
    boardWrapper.add(boardPanel);
    centerPanel.add(boardWrapper, BorderLayout.CENTER);

    movesListModel = new DefaultListModel<>();

```

```

movesList = new JList<>(movesListModel);
movesList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
movesList.setCellRenderer(new DefaultListCellRenderer() {
    @Override
    public Component getListCellRendererComponent(JList<?> list, Object value,
        int index, boolean isSelected, boolean cellHasFocus) {
        JLabel label = (JLabel) super.getListCellRendererComponent(
            list, value, index, isSelected, cellHasFocus);
        label.setBorder(BorderFactory.createEmptyBorder(5, 10, 5, 10));
        return label;
    }
});
movesList.addListSelectionListener(e -> {
    if (!e.getValueIsAdjusting() && movesList.getSelectedIndex() != -1) {
        currentStateIndex = movesList.getSelectedIndex();
        updateBoardDisplay();
        movesList.ensureIndexIsVisible(currentStateIndex);
    }
});

JSScrollPane scrollPane = new JSScrollPane(movesList);
scrollPane.setPreferredSize(new Dimension(250, 0));
centerPanel.add(scrollPane, BorderLayout.EAST);

add(centerPanel, BorderLayout.CENTER);
JPanel bottomPanel = new JPanel(new BorderLayout());
JPanel statusPanel = new JPanel(new BorderLayout());

JPanel labelsPanel = new JPanel(new FlowLayout(FlowLayout.LEFT, 10, 0));

statusLabel = new JLabel("Please load a board file.");
timeLabel = new JLabel("Time: -");
nodesVisitedLabel = new JLabel("Nodes visited: -");

statusLabel.setBorder(BorderFactory.createEmptyBorder(0, 10, 0, 10));
timeLabel.setBorder(BorderFactory.createEmptyBorder(0, 10, 0, 10));
nodesVisitedLabel.setBorder(BorderFactory.createEmptyBorder(0, 10, 0, 10));

labelsPanel.add(statusLabel);
labelsPanel.add(timeLabel);
labelsPanel.add(nodesVisitedLabel);

statusPanel.add(labelsPanel, BorderLayout.CENTER);

bottomPanel.add(statusPanel, BorderLayout.NORTH);
createAnimationControls(bottomPanel);

add(bottomPanel, BorderLayout.SOUTH);

setSize(700, 700);
setVisible(true);

animationTimer = new Timer(ANIMATION_DELAY, e -> showNextState());
animationTimer.setRepeats(true);
}

private void saveSolutionToFile() {
    JFileChooser fileChooser = new JFileChooser("test/output");
    fileChooser.setDialogTitle("Save Solution File");
    int userSelection = fileChooser.showSaveDialog(this);
}

```

```

if (userSelection == JFileChooser.APPROVE_OPTION) {
    File fileToSave = fileChooser.getSelectedFile();

    try {
        State finalState = solutionStates.get(solutionStates.size() - 1);
        if (finalState != null) {
            String[] solutionSteps = finalState.getSolutionPath();
            Files.write(fileToSave.toPath(), Arrays.asList(solutionSteps));
            JOptionPane.showMessageDialog(this, "Solution saved to " +
fileToSave.getAbsolutePath());
        } else {
            JOptionPane.showMessageDialog(this, "No solution available. Please
solve the board first.");
        }
    } catch (IOException ex) {
        JOptionPane.showMessageDialog(this, "Error saving file: " +
ex.getMessage());
        ex.printStackTrace();
    }
}

private void loadBoard() {
    JFileChooser chooser = new JFileChooser("test/input");
    int res = chooser.showOpenDialog(this);
    if (res == JFileChooser.APPROVE_OPTION) {
        currentFile = chooser.getSelectedFile();
        try {
            String[] input = components.IO.readFile(currentFile.getPath());
            board = components.IO.parseInput(input);
            drawBoard();
            statusLabel.setText("Board loaded: " + currentFile.getName());
            solveButton.setEnabled(true);
        } catch (Exception ex) {
            showAlertDialog(ex.getMessage());
        }
    }
}

private void showErrorDialog(String message) {
    JOptionPane.showMessageDialog(
        this,
        message,
        "Error Loading Board",
        JOptionPane.ERROR_MESSAGE);
}

private void drawBoard() {
    boardPanel.removeAll();
    if (board == null) {
        boardPanel.revalidate();
        boardPanel.repaint();
        return;
    }

    int rows = board.getRows();
    int cols = board.getCols();
    boardPanel.setLayout(new GridLayout(rows + 2, cols + 2));

    int size = Math.min(getHeight() - 150, getWidth() - 300);
    size = Math.min(size, Math.min(600, Math.max(300, size)));
    boardPanel.setPreferredSize(new Dimension(size, size));
}

```

```

char[][] grid = board.getGrid();
int kRow = components.IO.getKRow();
int kCol = components.IO.getKCol();

for (int i = -1; i <= rows; i++) {
    for (int j = -1; j <= cols; j++) {
        JPanel cell = new JPanel();
        cell.setBorder(BorderFactory.createLineBorder(Color.BLACK));
        cell.setOpaque(true);

        if (i == -1 || i == rows || j == -1 || j == cols) {
            cell.setBackground(Color.LIGHT_GRAY);

            if ((i == kRow && j == kCol) ||
                (kRow == -1 && i == -1 && j == kCol) ||
                (kRow == rows && i == rows && j == kCol) ||
                (kCol == -1 && i == kRow && j == -1) ||
                (kCol == cols && i == kRow && j == cols)) {
                cell.setBackground(Color.GREEN);
                JLabel label = new JLabel("K", SwingConstants.CENTER);
                label.setFont(new Font("Arial", Font.BOLD, 20));
                cell.add(label);
            }
        } else {
            char piece = grid[i][j];
            if (piece == '.') {
                cell.setBackground(Color.WHITE);
            } else if (piece == 'P') {
                cell.setBackground(Color.RED);
            } else {
                cell.setBackground(new Color(
                    (piece * 83) % 255,
                    (piece * 157) % 255,
                    (piece * 223) % 255));
            }

            if (piece != '.') {
                JLabel label = new JLabel(String.valueOf(piece),
                SwingConstants.CENTER);
                label.setFont(new Font("Arial", Font.BOLD, 20));
                cell.add(label);
            }
        }

        boardPanel.add(cell);
    }
}

boardPanel.revalidate();
boardPanel.repaint();
}

private void solveBoard() {
    if (board == null) {
        System.out.println("Board is null!");
        statusLabel.setText("No board loaded.");
        return;
    }

    if (animationTimer.isRunning()) {
        animationTimer.stop();
    }
}

```

```

    }

    playButton.setText("▶");
    prevButton.setEnabled(false);
    playButton.setEnabled(false);
    nextButton.setEnabled(false);

    solutionStates = null;
    currentStateIndex = 0;
    movesListModel.clear();

    try {
        String[] input = components.IO.readFile(currentFile.getPath());
        board = components.IO.parseInput(input);
        drawBoard();
    } catch (Exception ex) {
        statusLabel.setText("Failed to reset board: " + ex.getMessage());
        return;
    }
    System.out.println("Starting solve process...");
    statusLabel.setText("Solving...");
    timeLabel.setText("Time: -");
    nodesVisitedLabel.setText("Nodes visited: -");
    solveButton.setEnabled(false);

    SwingWorker<components.State, Void> worker = new
    SwingWorker<components.State, Void>() {
        long time = 0;

        @Override
        protected components.State doInBackground() {           components.State
            solution = null;
            String algo = (String) algoBox.getSelectedItem();
            System.out.println("Using algorithm: " + algo);
            long start = System.currentTimeMillis();
            String heuristic = "";
            String heuristicType = "";

            if (algo.equals("Greedy Best First Search") || algo.equals("A*")) {
                String selectedHeuristic = heuristicBox.getSelectedItem().toString();
                if (selectedHeuristic.equals("Jarak Piece ke K")) {
                    heuristic = "pieceToDest";
                    heuristicType = "Piece to Destination";
                } else if (selectedHeuristic.equals("Jumlah Piece Penghalang")) {
                    heuristic = "countBlockingPieces";
                    heuristicType = "Count Blocking Pieces";
                } else {
                    heuristic = "combineTwo";
                    heuristicType = "Combination of Count Blocking Pieces and Piece
to Destination";
                }
            }

            try {
                switch (algo) {
                    case "Uniform Cost Search":
                        System.out.println("Starting UCS...");
                        solution = new UniformCostSearch().solve(board.copy());
                        break;
                    case "Greedy Best First Search":
                        System.out.println("Starting GBFS...");
                        solution = new GreedyBestFirstSearch().solve(board.copy(),
heuristic);
                }
            }
        }
    }
}

```

```

        break;
    case "A*":
        System.out.println("Starting A*...");
        solution = AStar.solve(board.copy(), heuristic);
        break;
    case "Iterative Deepening DFS":
        System.out.println("Starting IDDFS...");
        solution = new IterativeDeepeningSearch().solve(board.copy());
        break;
    }
} catch (Exception e) {
    System.out.println("Error during solving: " + e.getMessage());
    e.printStackTrace();
}
time = System.currentTimeMillis() - start;
solution.setExecutionTime(time);
solution.setAlgorithm(algo);
solution.setHeuristicType(heuristicType);
System.out.println("Solve completed in " + time + "ms");
return solution;
}

@Override
protected void done() {
    try {
        components.State solution = get();
        if (solution != null) {
            System.out.println("Solution found!");
            solutionStates = new ArrayList<>();
            components.State current = solution;
            while (current != null) {
                solutionStates.add(0, current);
                current = current.getParent();
            }
            currentStateIndex = 0;

            statusLabel.setText("Solution found! " + (solutionStates.size() - 1)
+ " moves");
            timeLabel.setText(String.format("Time: %d ms", time));
            nodesVisitedLabel.setText("Nodes visited: " +
solution.getTotalNodeVisited());

            movesListModel.clear();
            movesListModel.addElement("Step 0: Initial State");
            List<Move> moves = solution.getPathFromRoot();
            for (int i = 0; i < moves.size(); i++) {
                Move move = moves.get(i);
                String direction = move.getDirection();
                char piece = move.getPiece().getLetter();
                int steps = move.getSteps();
                movesListModel.addElement(String.format("Step %d: Move %c %s by
%d",
                    i + 1, piece, direction, steps));
            }
            movesList.setSelectedIndex(0);

            prevButton.setEnabled(true);
            playButton.setEnabled(true);
            nextButton.setEnabled(true);

            updateBoardDisplay();
            saveButton.setEnabled(true);
        }
    }
}

```

```

        animationTimer.start();
        playButton.setText("▶");
    } else {
        System.out.println("No solution found!");
        statusLabel.setText("No solution found.");
    }
} catch (Exception e) {
    System.out.println("Error in done(): " + e.getMessage());
    e.printStackTrace();
    statusLabel.setText("Error during solving: " + e.getMessage());
}
solveButton.setEnabled(true);
}
};

worker.execute();
}

private void createAnimationControls(JPanel bottomPanel) {
    JPanel controlPanel = new JPanel();
    controlPanel.setBorder(BorderFactory.createEmptyBorder(5, 0, 5, 0));

    prevButton = new JButton("◀");
    prevButton.setEnabled(false);
    prevButton.addActionListener(e -> showPreviousState());

    playButton = new JButton("▶");
    playButton.setEnabled(false);
    playButton.addActionListener(e -> toggleAnimation());

    nextButton = new JButton("→");
    nextButton.setEnabled(false);
    nextButton.addActionListener(e -> showNextState());

    controlPanel.add(prevButton);
    controlPanel.add(playButton);
    controlPanel.add(nextButton);

    bottomPanel.add(controlPanel, BorderLayout.CENTER);
}

private void toggleAnimation() {
    if (animationTimer.isRunning()) {
        animationTimer.stop();
        playButton.setText("▶");
    } else {
        animationTimer.start();
        playButton.setText("⏸");
    }
}

private void showNextState() {
    if (solutionStates != null && currentStateIndex < solutionStates.size() - 1)
    {
        currentStateIndex++;
        updateBoardDisplay();

        char[][] grid = board.getGrid();
        if (currentStateIndex == solutionStates.size() - 1) {
            int kRow = components.IO.getKRow();
            int kCol = components.IO.getKCol();
            int rows = board.getRows();
        }
    }
}

```

```

int cols = board.getCols();

int pRow = -1, pCol = -1, pSize = 0;
boolean isHorizontal = false;

for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        if (grid[i][j] == 'P') {
            if (pRow == -1) {
                pRow = i;
                pCol = j;
                if (j + 1 < cols && grid[i][j + 1] == 'P') {
                    isHorizontal = true;
                }
            }
            pSize++;
        }
    }
}

boolean atExit = false;
if (kRow == -1 && kCol >= 0 && kCol < cols && grid[0][kCol] == 'P' &&
!isHorizontal)
    atExit = true;
else if (kRow == rows && kCol >= 0 && kCol < cols && grid[rows - 1][kCol]
== 'P' && !isHorizontal)
    atExit = true;
else if (kCol == -1 && kRow >= 0 && kRow < rows && grid[kRow][0] == 'P'
&& isHorizontal)
    atExit = true;
else if (kCol == cols && kRow >= 0 && kRow < rows && grid[kRow][cols - 1]
== 'P' && isHorizontal)
    atExit = true;

if (atExit) {
    animationTimer.stop();

    Timer exitTimer = new Timer(ANIMATION_DELAY, e -> {
        for (int i = 0; i < grid.length; i++) {
            for (int j = 0; j < grid[0].length; j++) {
                if (grid[i][j] == 'P') {
                    grid[i][j] = '.';
                }
            }
        }
        drawBoard();
        statusLabel.setText("Puzzle solved! The red piece has escaped!");
        playButton.setText("▶");
        ((Timer) e.getSource()).stop();
    });
    exitTimer.setRepeats(false);
    exitTimer.start();
}
}

} else {
    animationTimer.stop();
    playButton.setText("▶");
}
}

private void showPreviousState() {
    if (solutionStates != null && currentStateIndex > 0) {

```

```

        currentStateIndex--;
        updateBoardDisplay();
    }
}

private void updateBoardDisplay() {
    if (solutionStates != null && currentStateIndex >= 0 && currentStateIndex <
solutionStates.size()) {
        State currentState = solutionStates.get(currentStateIndex);
        board = currentState.getBoard();
        drawBoard();
        statusLabel.setText("Move " + currentStateIndex + " of " +
(solutionStates.size() - 1));
    }
}
}

```

4.1.4 Main

Main.java

```

import components.*;
import solver.*;
import java.util.*;

public class Main {
    public static void main(String[] args) {
        try {
            System.out.println("Enter input file path (from test/input folder): ");
            Scanner scanner = new Scanner(System.in);
            String inputFile = scanner.nextLine();
            String filepath = "test/input/" + inputFile;

            String[] inputString = IO.readFile(filepath);
            Board initialBoard = IO.parseInput(inputString);

            System.out.println("\nInitial Board State:");
            initialBoard.printBoard();
            System.out.printf("Primary Col: %d, KCol: %d\n",
initialBoard.getPrimaryPiece().getCol(), IO.getKCol());

            if (initialBoard.getPrimaryPiece().getOrientation() == 'H') {
                if (initialBoard.getPrimaryPiece().getRow() != IO.getKRow()) {
                    System.out.println("Unsolvable: Exit location and primary piece is not
alligned");
                    scanner.close();
                    return;
                }
            } else {
                if (initialBoard.getPrimaryPiece().getCol() != IO.getKCol()) {
                    System.out.println("Unsolvable: Exit location and primary piece is not
alligned");
                    scanner.close();
                    return;
                }
            }

            System.out.println("\nChoose search algorithm:");
            System.out.println("1. Uniform Cost Search");
            System.out.println("2. Greedy Best First Search");
        }
    }
}

```

```

System.out.println("3. A*");
System.out.println("4. Iterative Deepening Depth First Search");
System.out.print("Enter choice (1-3): ");

int choice = scanner.nextInt();
State solution = null;
String algorithm = "";
String heuristicType = "";
String heuristic = "";
long startTime = System.currentTimeMillis();

switch (choice) {
    case 1:
        UniformCostSearch ucs = new UniformCostSearch();
        algorithm = "Uniform Cost Search";
        solution = ucs.solve(initialBoard);
        break;
    case 2:
        algorithm = "Greedy Best First Search";
        System.out.println("Choose heuristic:");
        System.out.println("1. Jarak Piece ke K");
        System.out.println("2. Jumlah Piece Penghalang");
        System.out.println("3. Gabungan Dua Heuristic");
        System.out.print("Enter heuristic (1-3): ");
        int hChoice = scanner.nextInt();
        scanner.nextLine();

        if (hChoice < 1 || hChoice > 3) {
            System.out.println("Invalid heuristic choice, using default.");
            heuristicType = "pieceToDest";
        }

        if (hChoice == 1) {
            heuristicType = "pieceToDest";
        } else if (hChoice == 2) {
            heuristicType = "countBlockingPieces";
        } else {
            heuristicType = "combineTwo";
        }

        GreedyBestFirstSearch gbfs = new GreedyBestFirstSearch();
        solution = gbfs.solve(initialBoard, heuristicType);
        break;
    case 3:
        algorithm = "A*";
        System.out.println("Choose heuristic:");
        System.out.println("1. Jarak Piece ke K");
        System.out.println("2. Jumlah Piece Penghalang");
        System.out.println("3. Gabungan Dua Heuristic");
        System.out.print("Enter heuristic (1-3): ");
        hChoice = scanner.nextInt();
        scanner.nextLine();

        if (hChoice < 1 || hChoice > 3) {
            System.out.println("Invalid heuristic choice, using default.");
            heuristicType = "pieceToDest";
        }

        if (hChoice == 1) {
            heuristicType = "pieceToDest";
        } else if (hChoice == 2) {
            heuristicType = "countBlockingPieces";
        }
}

```

```

        } else {
            heuristicType = "combineTwo";
        }
        solution = AStar.solve(initialBoard, heuristicType);
        break;
    case 4:
        algorithm = "Iterative Deepening Depth First Search";
        IterativeDeepeningSearch idfs = new IterativeDeepeningSearch();
        solution = idfs.solve(initialBoard);
        break;

    default:
        System.out.println("Invalid choice");
        return;
    }

long endTime = System.currentTimeMillis();

if (solution != null) {
    List<Move> path = solution.getPathFromRoot();
    System.out.println("\nSolution found!");
    System.out.println("Number of moves: " + path.size());
    System.out.println("Number of moves: " + path.size());
    System.out.println("Time taken: " + (endTime - startTime) + "ms");

    System.out.println("\nMoves:");
    for (int i = 0; i < path.size(); i++) {
        Move move = path.get(i);
        System.out.printf("%d. Move piece %c %s by %d steps\n",
            i + 1,
            move.getPiece().getLetter(),
            move.getDirection(),
            move.getSteps());
    }
}

System.out.println("\nFinal Board State:");
solution.printState();
solution.setExecutionTime(endTime - startTime);
solution.setAlgorithm(algorithm);
solution.setHeuristicType(heuristicType);

String[] output = solution.getSolutionPath();

IO.saveOutputToFile(output, new
ArrayList<>(initialBoard.getPieces().values()));
} else {
    System.out.println("\nNo solution found!");
}

scanner.close();

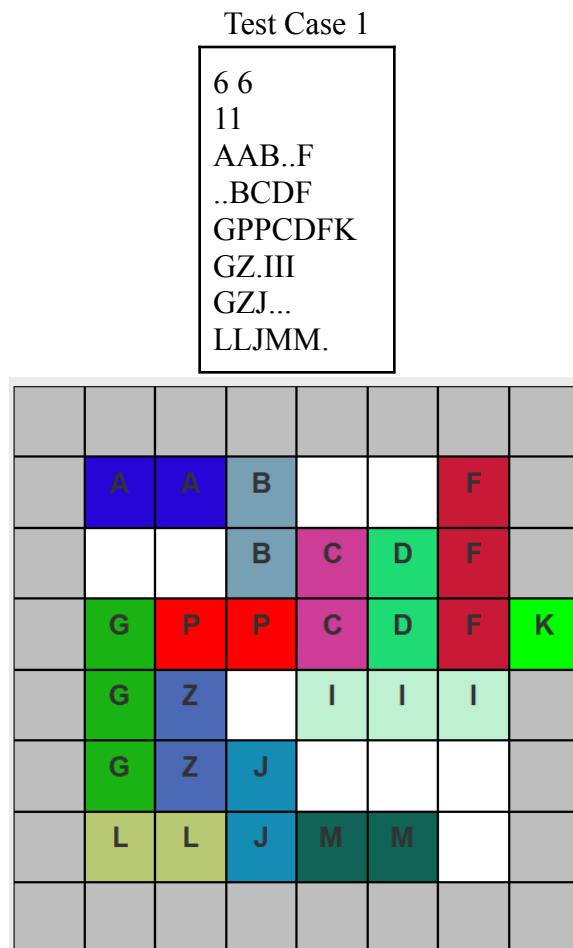
} catch (Exception e) {
    System.err.println("Error: " + e.getMessage());
    e.printStackTrace();
}
}
}

```

4.2 Hasil Pengujian

4.2.1 Test Case

Pada pengujian kali ini, kami menggunakan empat test case untuk masing-masing algoritma. Khusus untuk algoritma GBFS dan A*, kami menerapkan masing-masing test case untuk setiap metode heuristik yang kami buat.



Gambar 10. Test Case 1

Test Case 2

6 6									
11									
AAB..F									
..BCDF									
KGPPCDF									
GZ.III									
GZJ...									
LLJMM.									

	A	A	B				F	
			B	C	D		F	
K	G	P	P	C	D		F	
G	Z			I	I	I		
G	Z	J						
L	L	J	M	M				

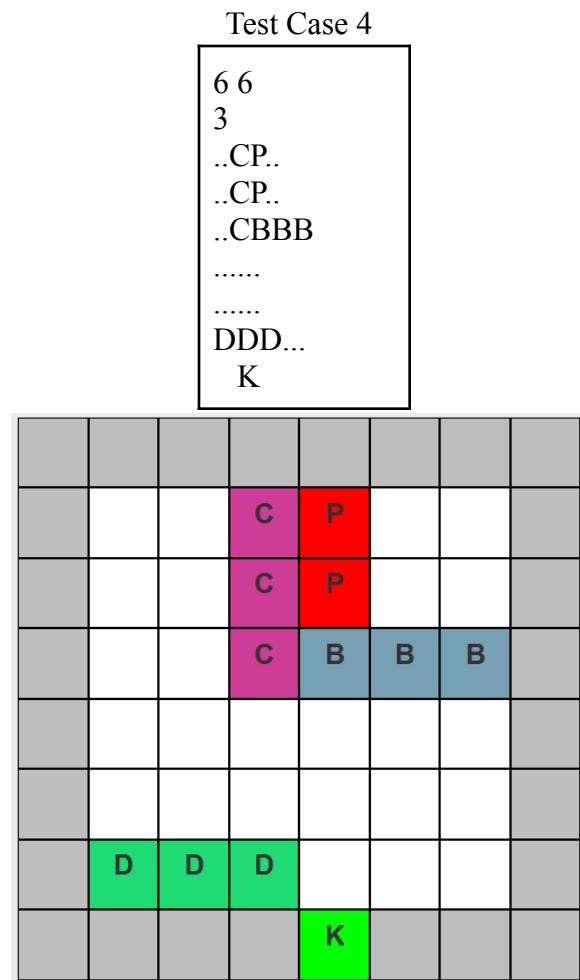
Gambar 11. Test Case 2

Test Case 3

6 6
3
K
...DDD
.....
.....
BBBC..
..PC..
..PC..

				K				
					D	D	D	
	B	B	B	C				
				P	C			
				P	C			

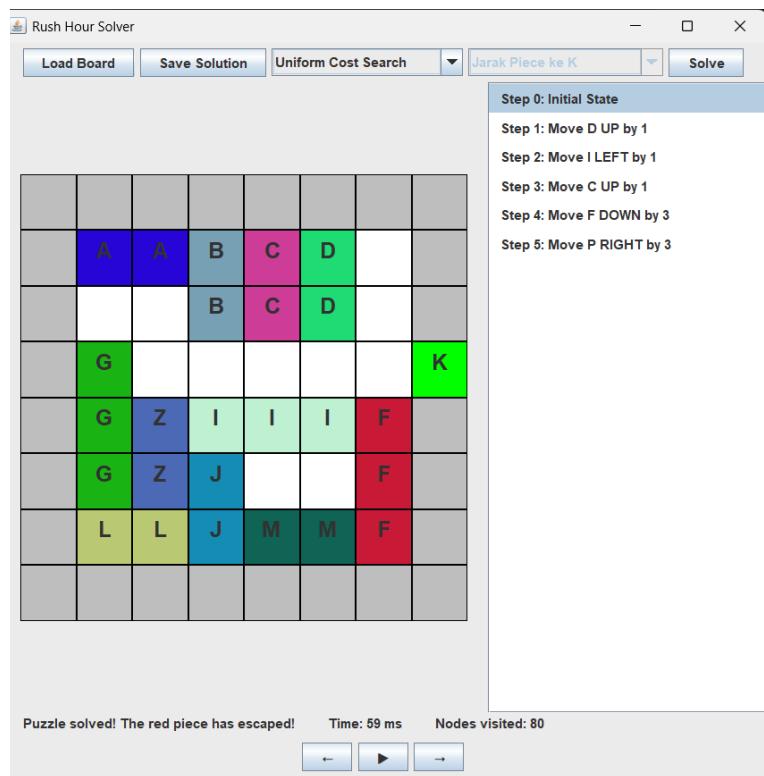
Gambar 12. Test Case 3



Gambar 13. Test Case 4

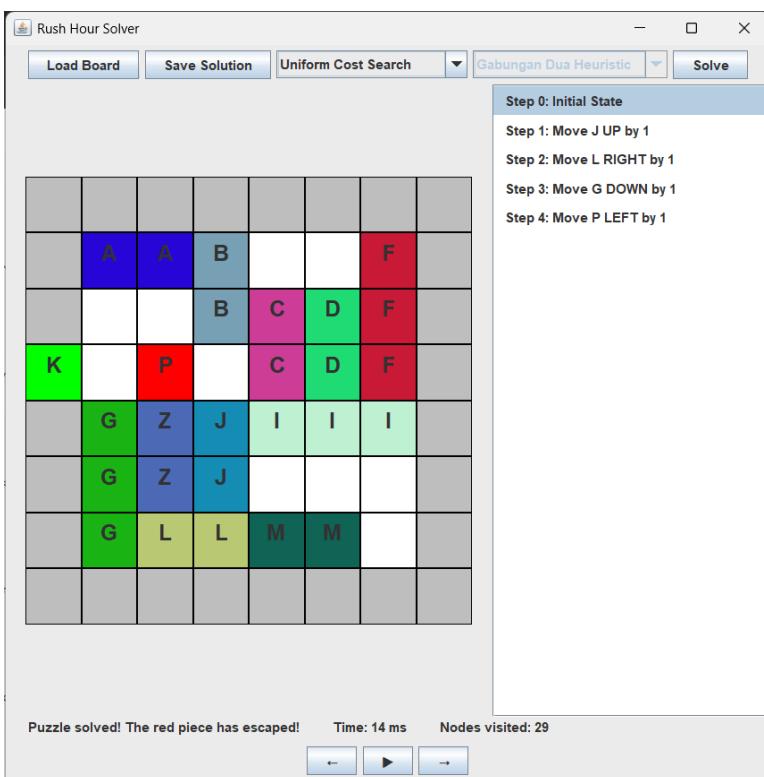
4.2.1 Algoritma UCS

- Test Case 1



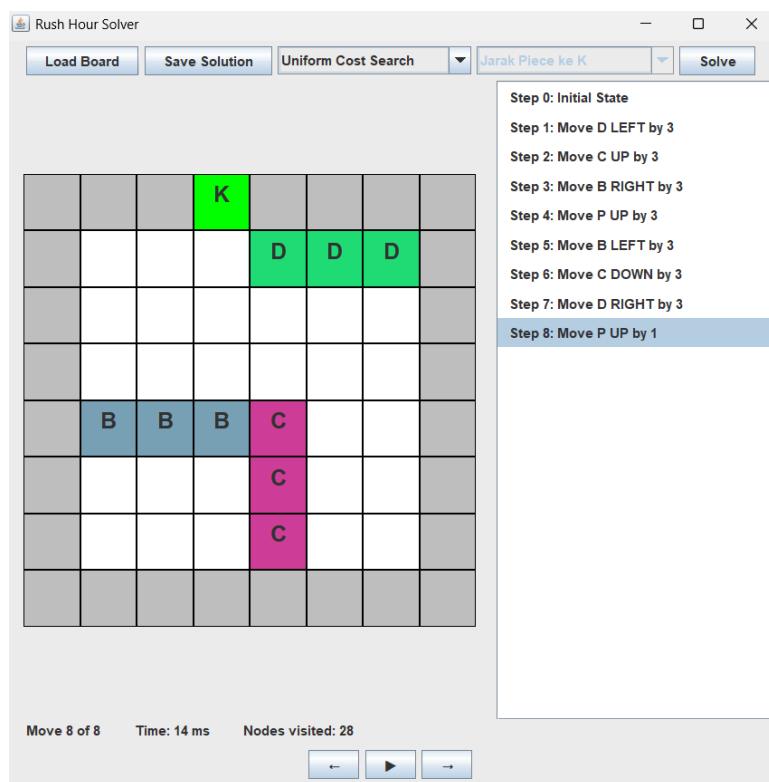
Gambar 14. Test Case UCS 1

- Test Case 2



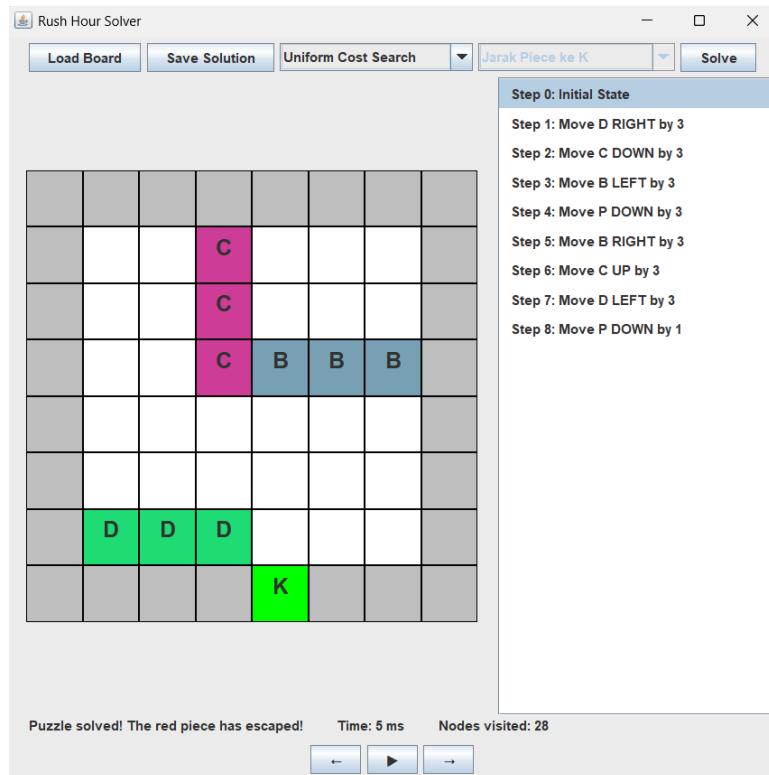
Gambar 15. Test Case UCS 2

- Test Case 3



Gambar 16. Test Case UCS 3

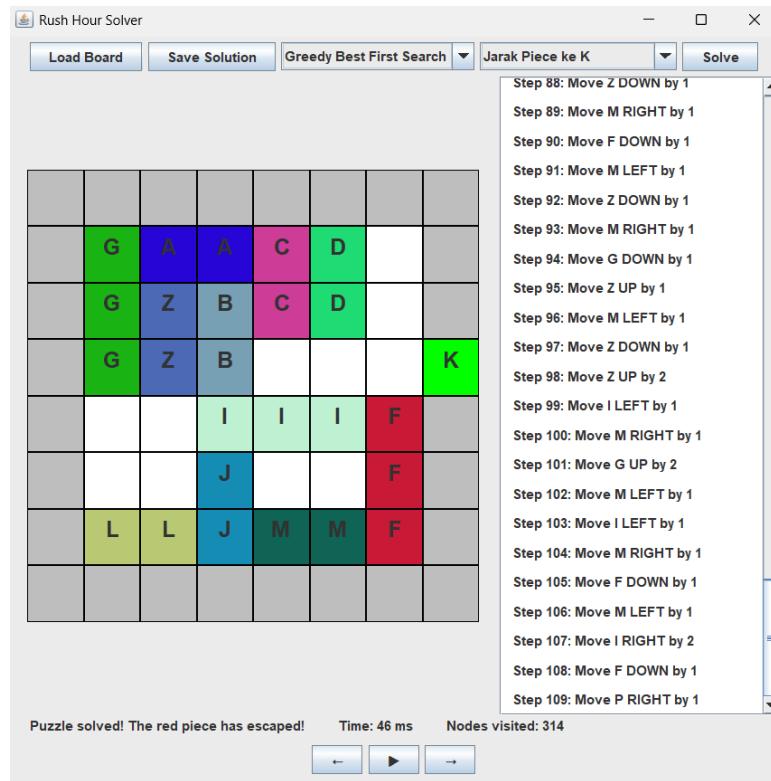
- Test Case 4



Gambar 17. Test Case UCS 4

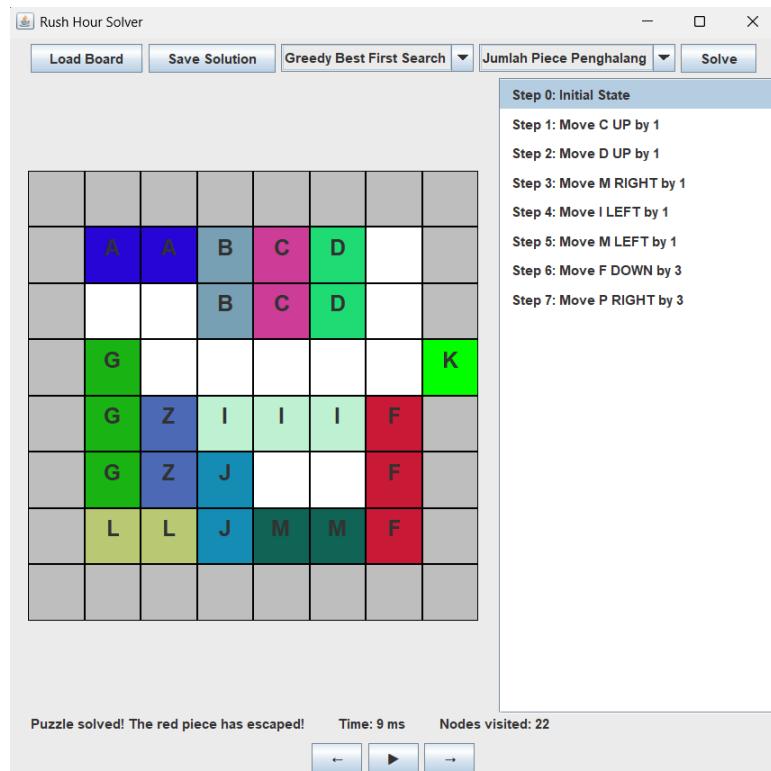
4.2.2 Algoritma GBFS

- Test Case 1
 - Heuristik Jarak Piece ke K



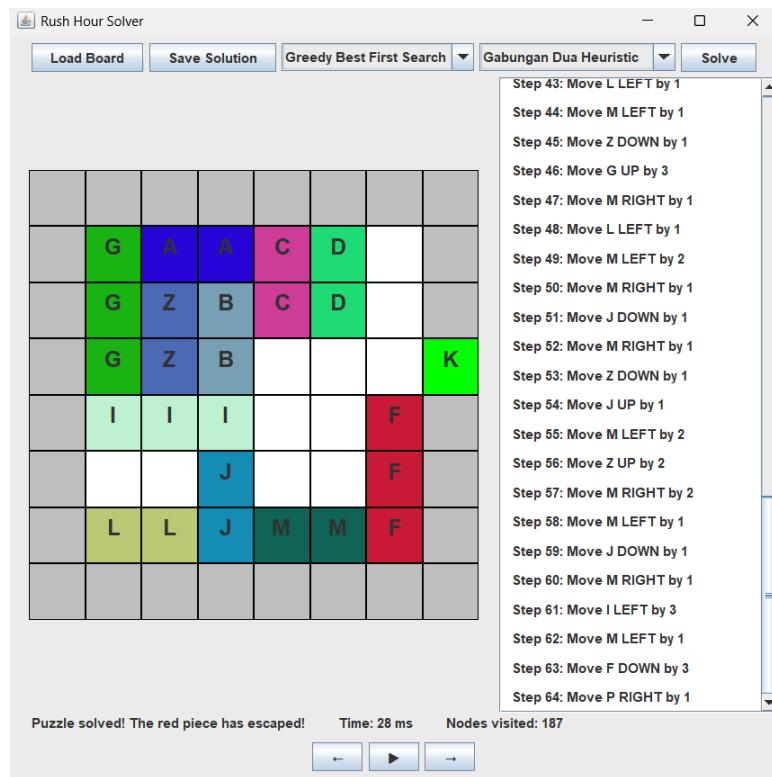
Gambar 18. Test Case GBFS 1 Heuristik 1

- Heuristik Jumlah Piece Penghalang



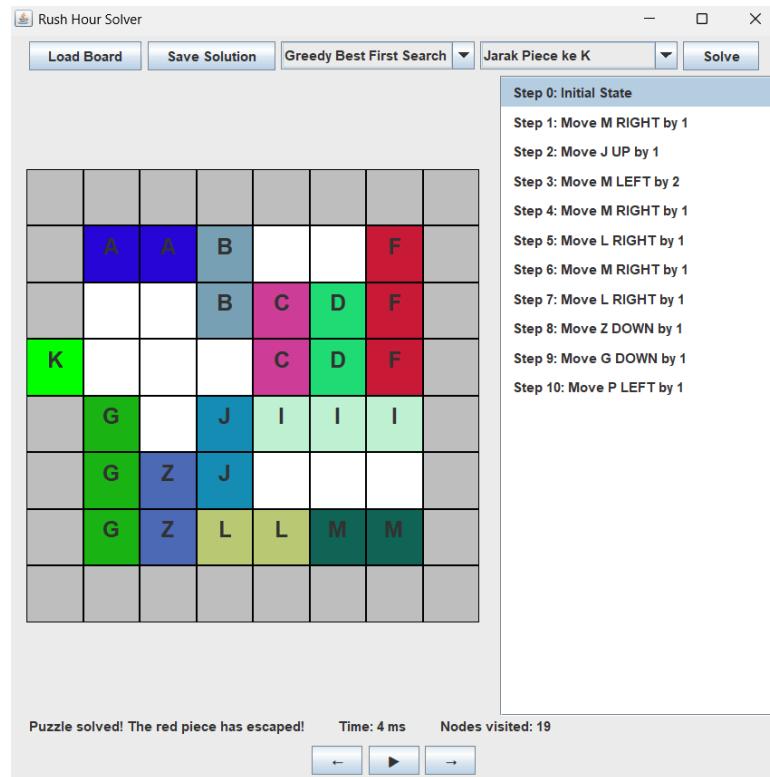
Gambar 19. Test Case GBFS 1 Heuristik 2

- Heuristik Gabungan Dua Heuristik



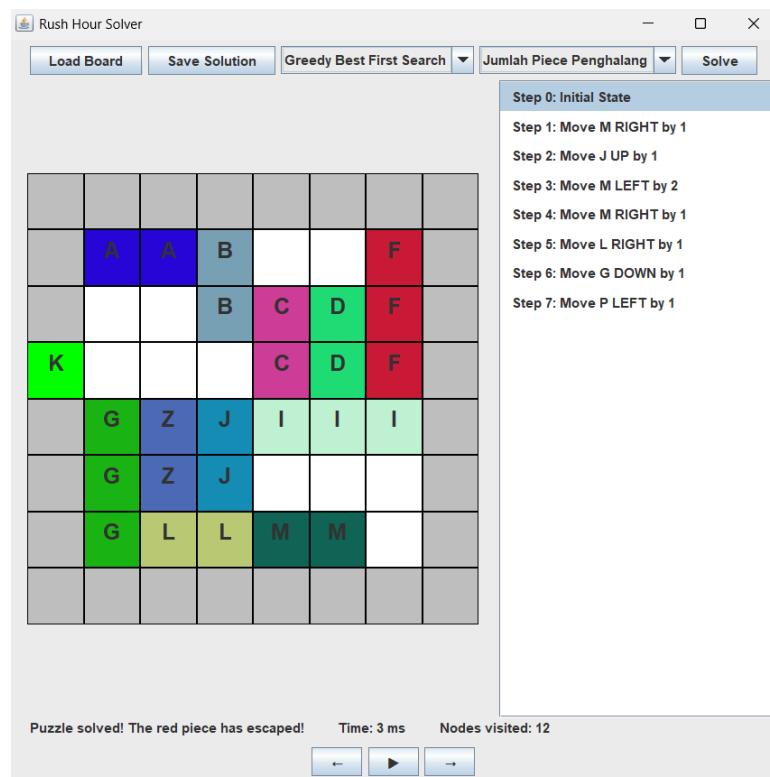
Gambar 20. Test Case GBFS 1 Heuristik 3

- Test Case 2
 - Heuristik Jarak Piece ke K



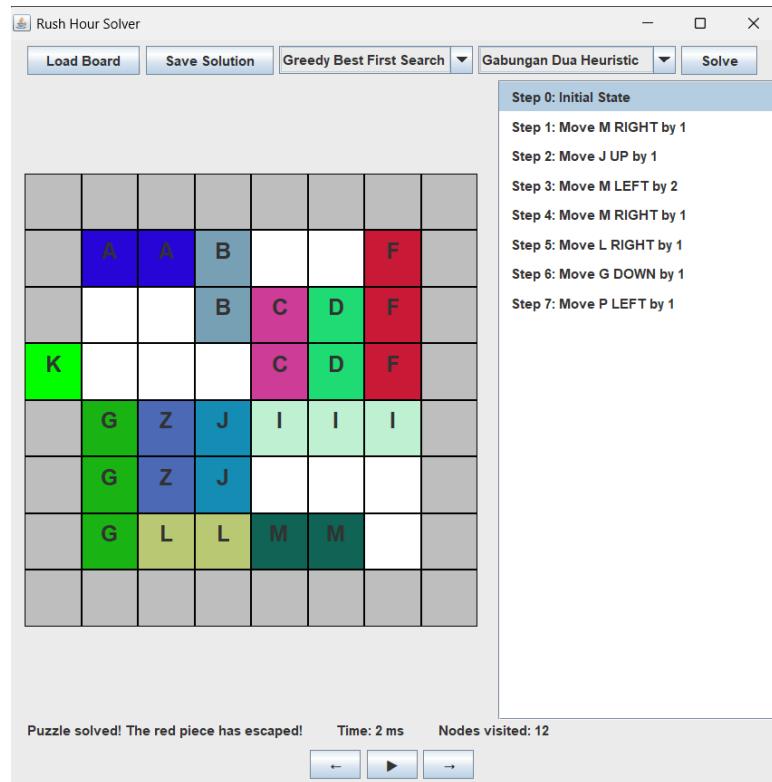
Gambar 21. Test Case GBFS 2 Heuristik 1

- Heuristik Jumlah Piece Penghalang



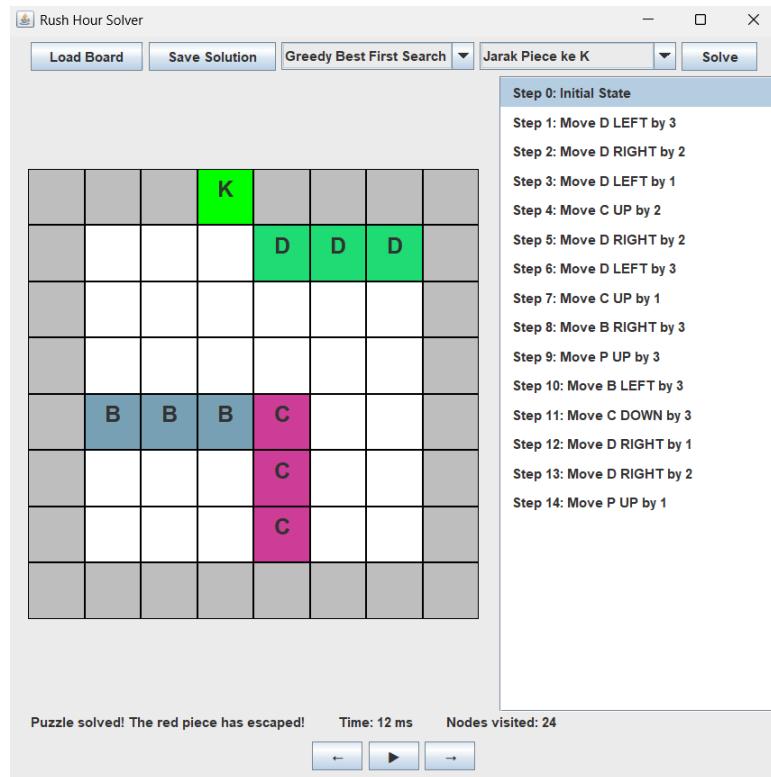
Gambar 22. Test Case GBFS 2 Heuristik 2

- Heuristik Gabungan Dua Heuristik



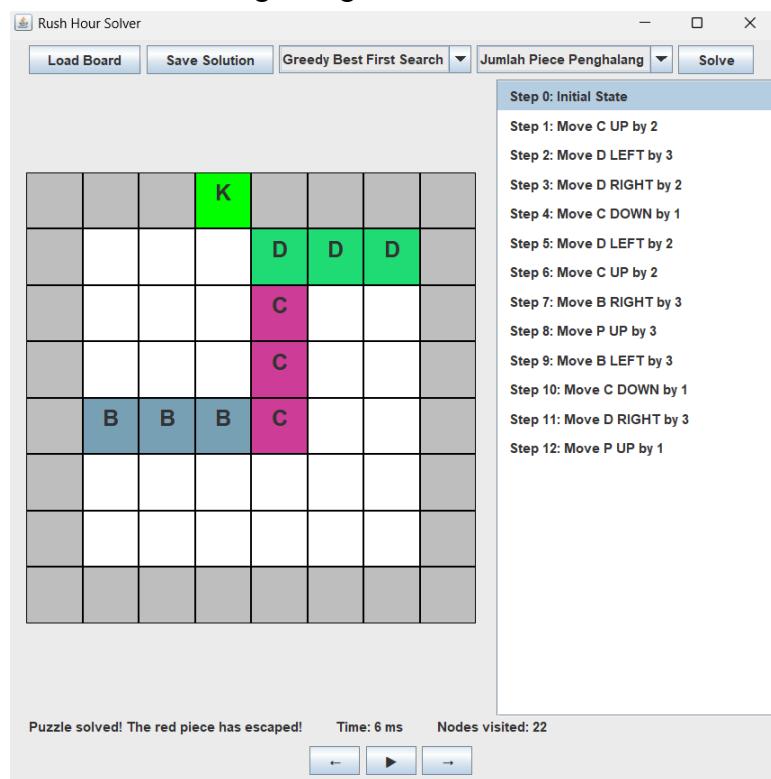
Gambar 23. Test Case GBFS 2 Heuristik 3

- Test Case 3
 - Heuristik Jarak Piece ke K



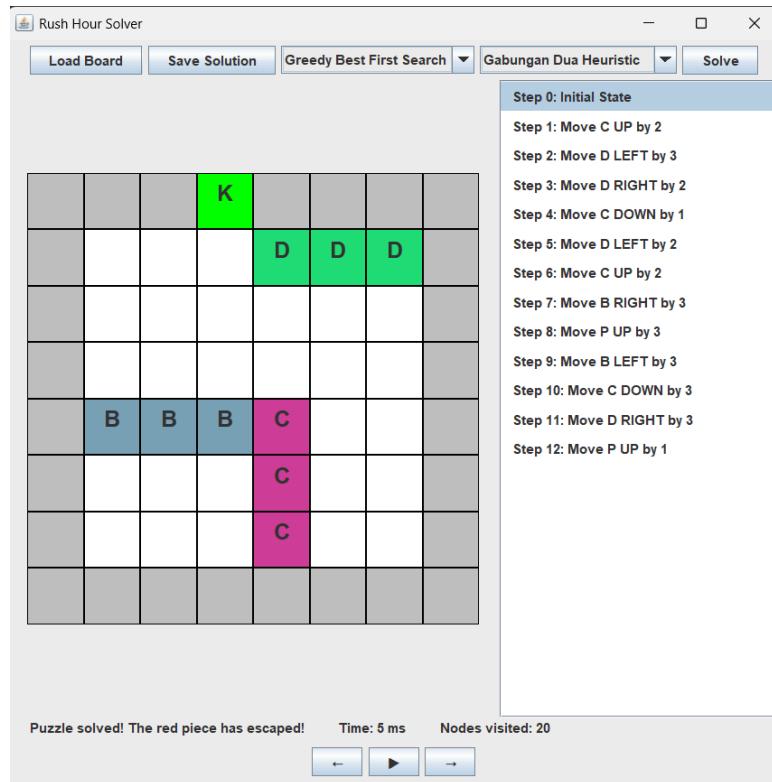
Gambar 24. Test Case GBFS 3 Heuristik 1

- Heuristik Jumlah Piece Penghalang



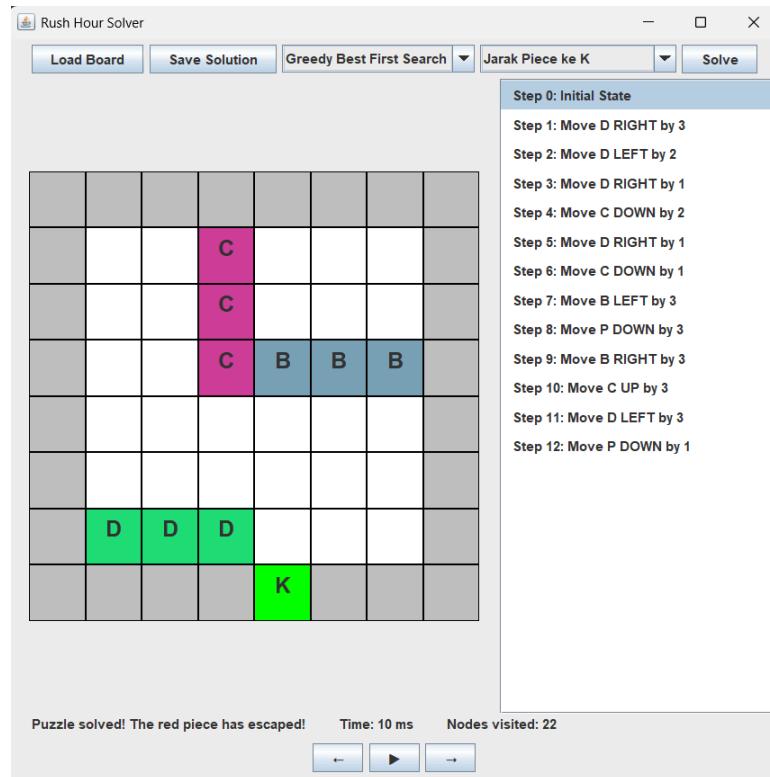
Gambar 25. Test Case GBFS 3 Heuristik 2

- Heuristik Gabungan Dua Heuristik



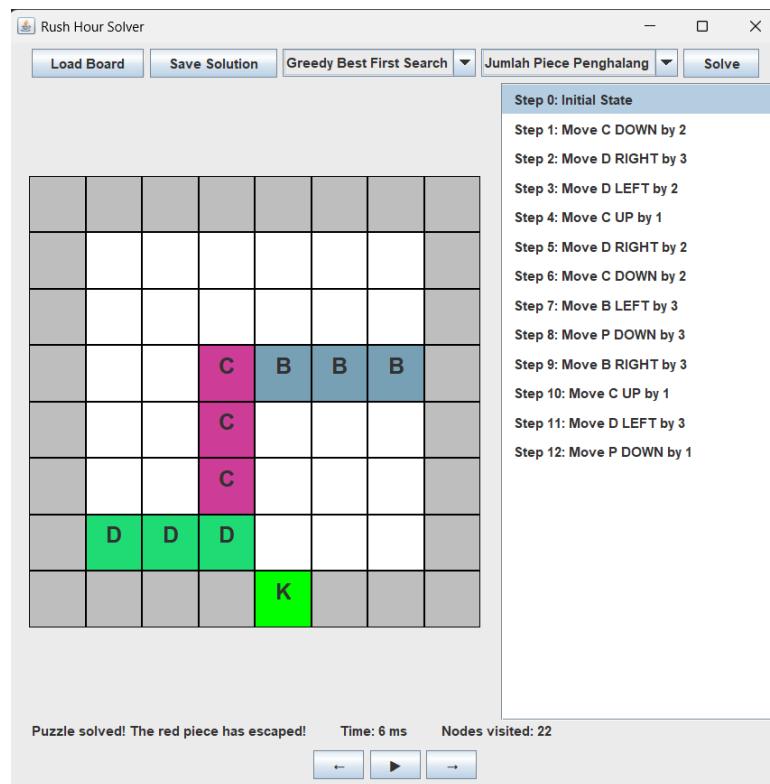
Gambar 26. Test Case GBFS 3 Heuristik 3

- Test Case 4
 - Heuristik Jarak Piece ke K



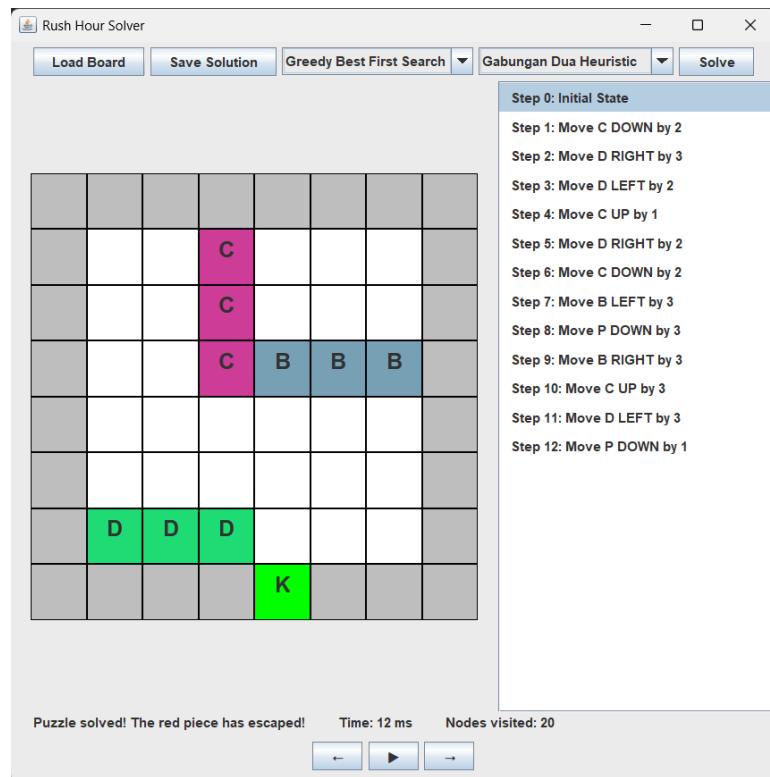
Gambar 26. Test Case GBFS 4 Heuristik 1

- Heuristik Jumlah Piece Penghalang



Gambar 27. Test Case GBFS 4 Heuristik 2

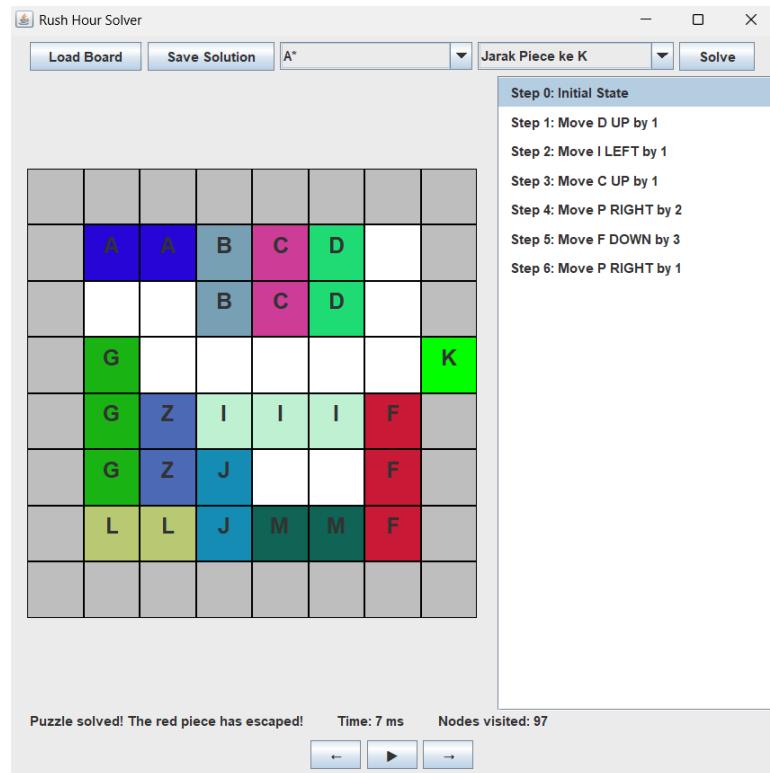
- Heuristik Gabungan Dua Heuristik



Gambar 28. Test Case GBFS 4 Heuristik 3

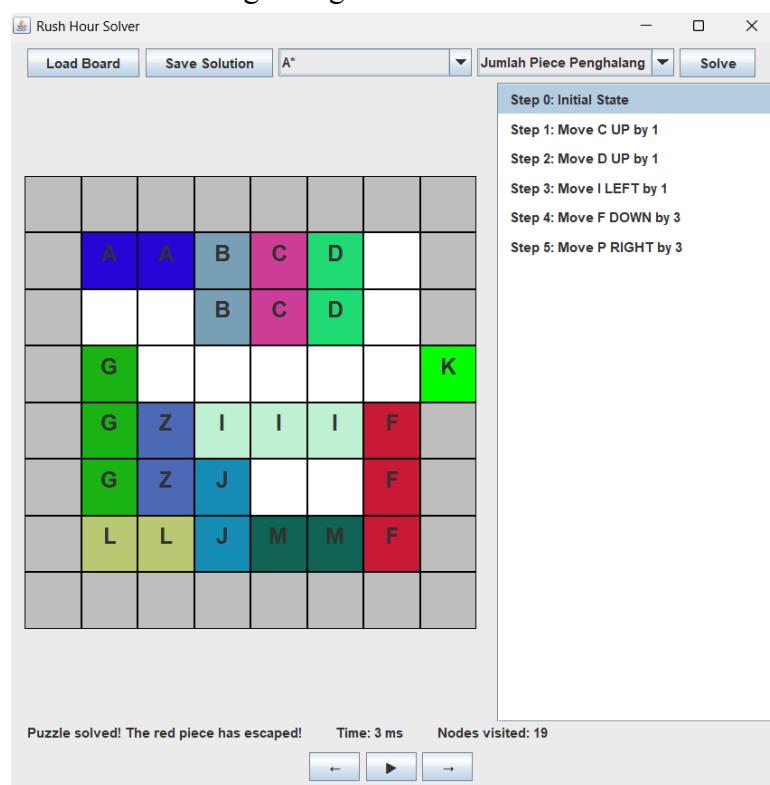
4.2.3 Algoritma A*

- Test Case 1
 - Heuristik Jarak Piece ke K



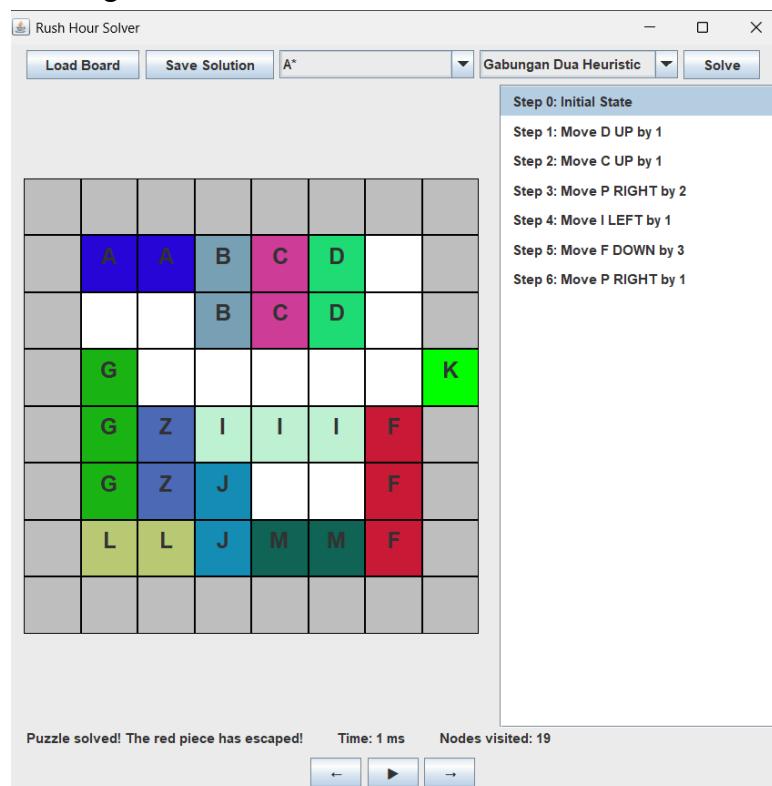
Gambar 29. Test Case A* 1 Heuristik 1

- Heuristik Jumlah Piece Penghalang



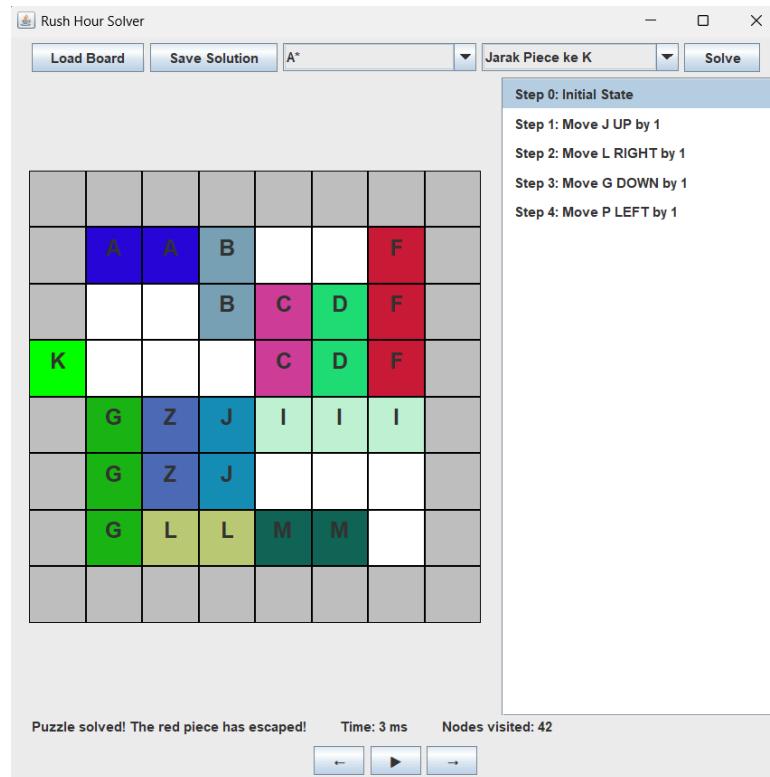
Gambar 30. Test Case A* 1 Heuristik 2

- Heuristik Gabungan Dua Heuristik



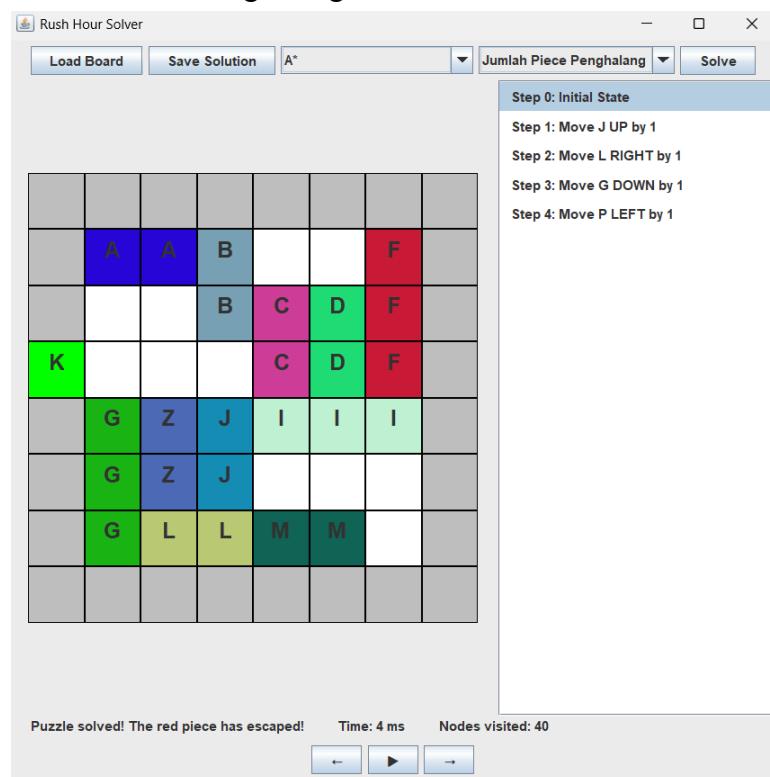
Gambar 31. Test Case A* 1 Heuristik 3

- Test Case 2
 - Heuristik Jarak Piece ke K



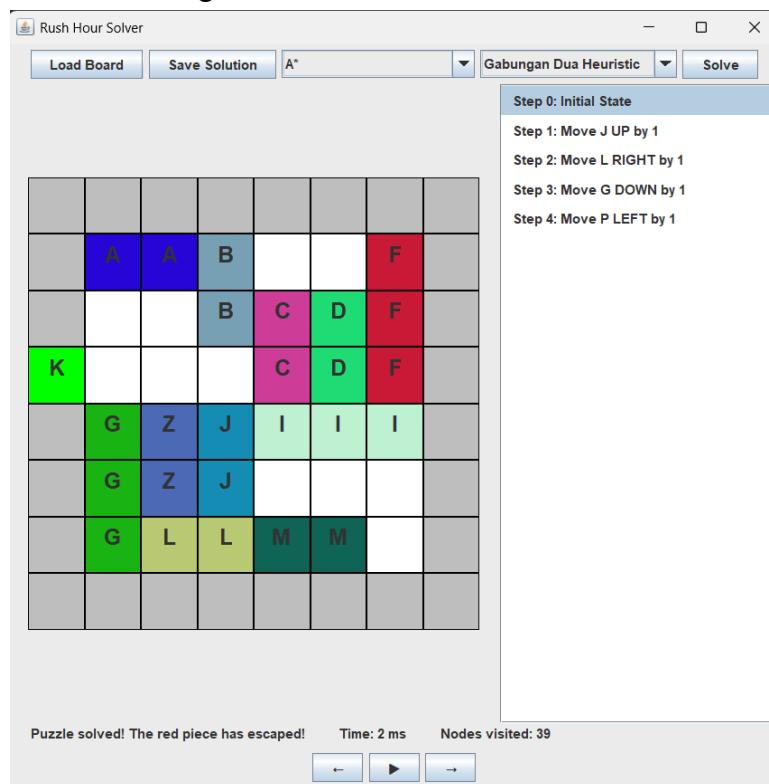
Gambar 32. Test Case A* 2 Heuristik 1

- Heuristik Jumlah Piece Penghalang



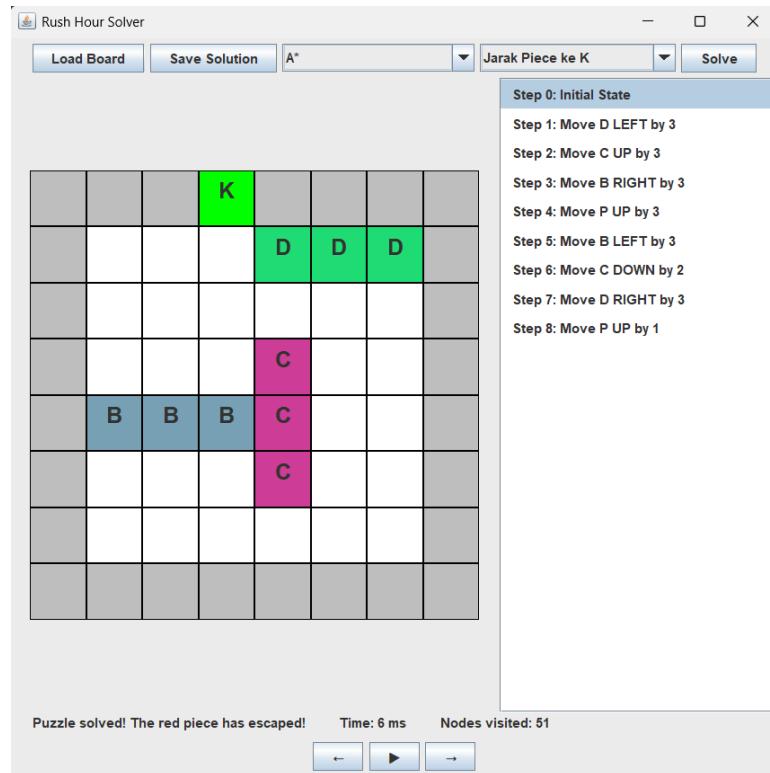
Gambar 33. Test Case A* 2 Heuristik 2

- Heuristik Gabungan Dua Heuristik



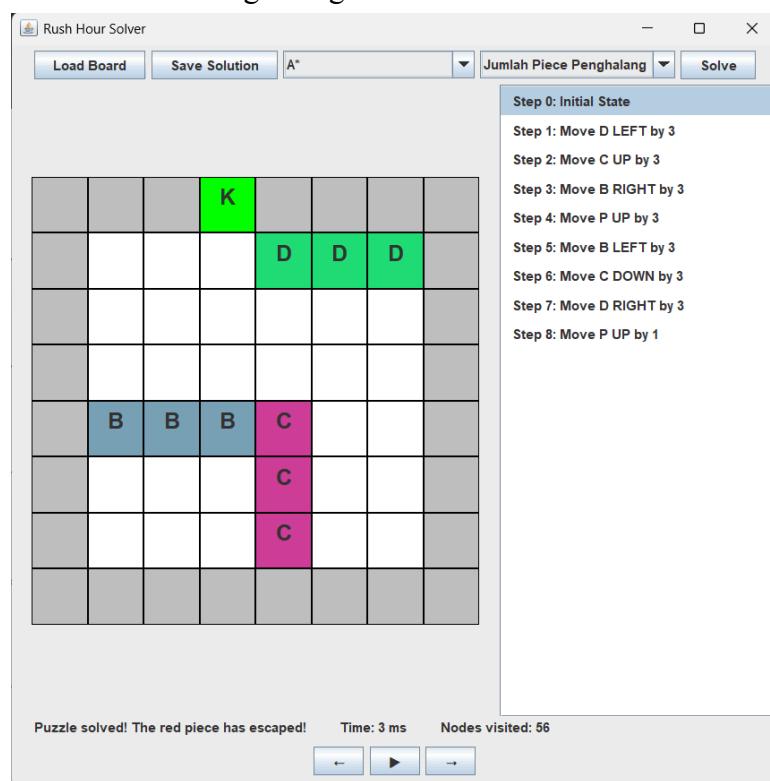
Gambar 34. Test Case A* 2 Heuristik 3

- Test Case 3
 - Heuristik Jarak Piece ke K



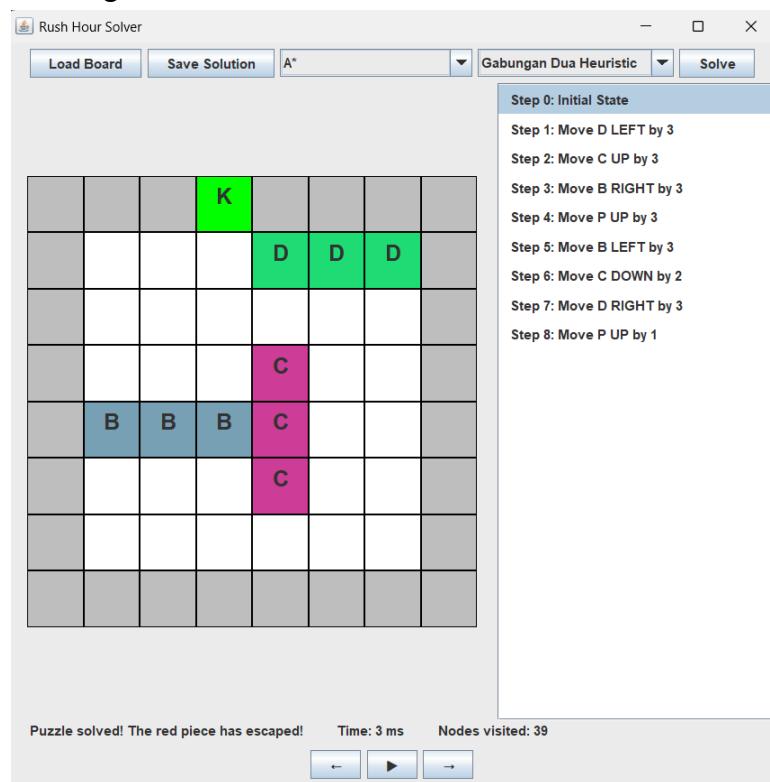
Gambar 35. Test Case A* 3 Heuristik 1

- Heuristik Jumlah Piece Penghalang



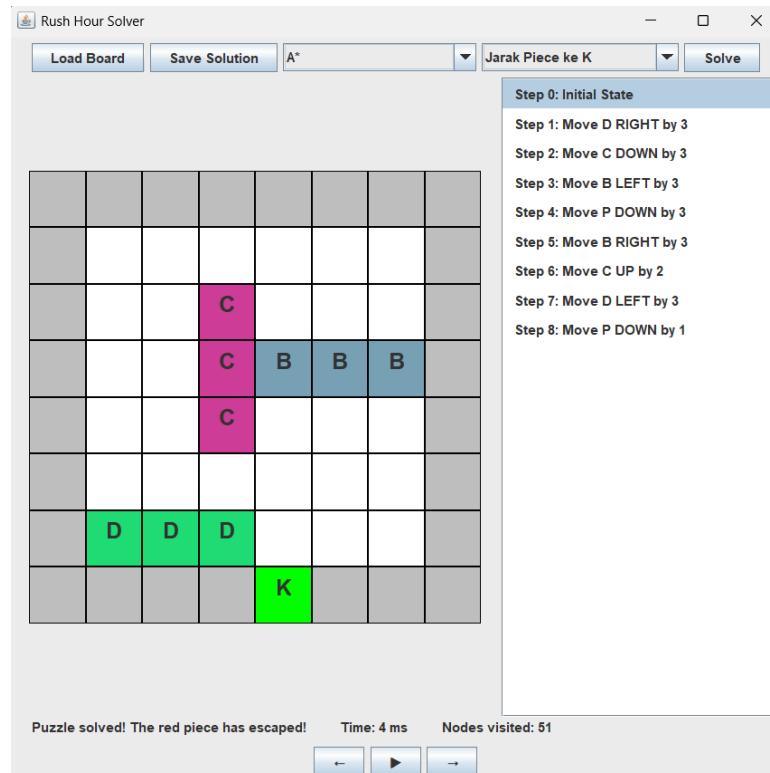
Gambar 36. Test Case A* 3 Heuristik 2

- Heuristik Gabungan Dua Heuristik



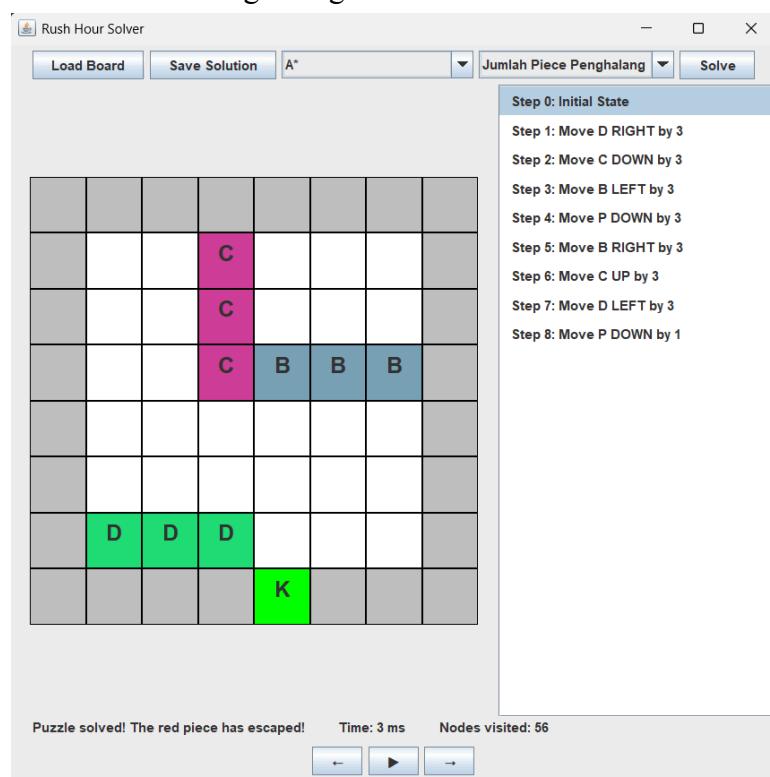
Gambar 37. Test Case A* 3 Heuristik 3

- Test Case 4
 - Heuristik Jarak Piece ke K



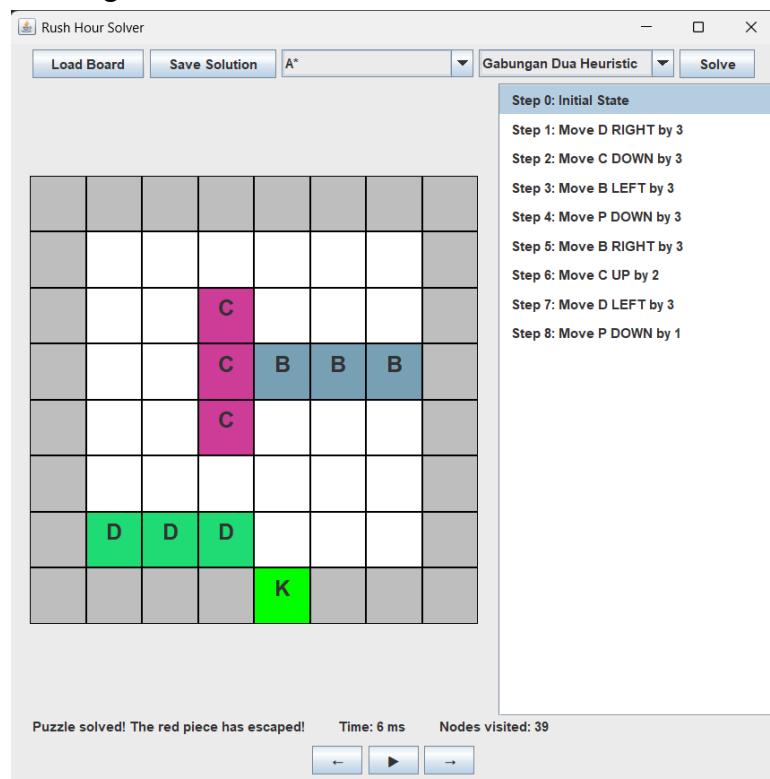
Gambar 38. Test Case A* 4 Heuristik 1

- Heuristik Jumlah Piece Penghalang



Gambar 39. Test Case A* 4 Heuristik 2

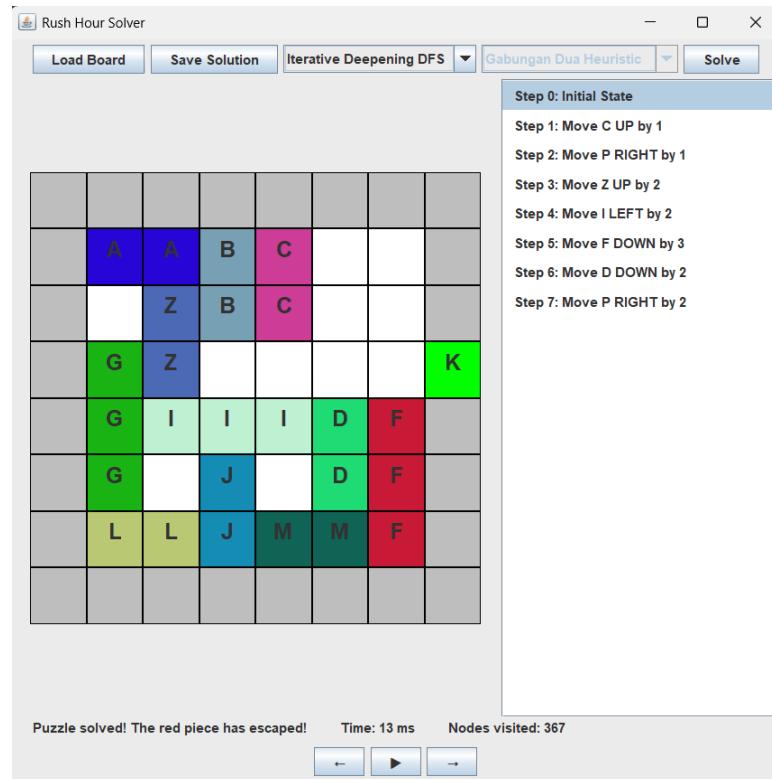
- Heuristik Gabungan Dua Heuristik



Gambar 40. Test Case A* 4 Heuristik 3

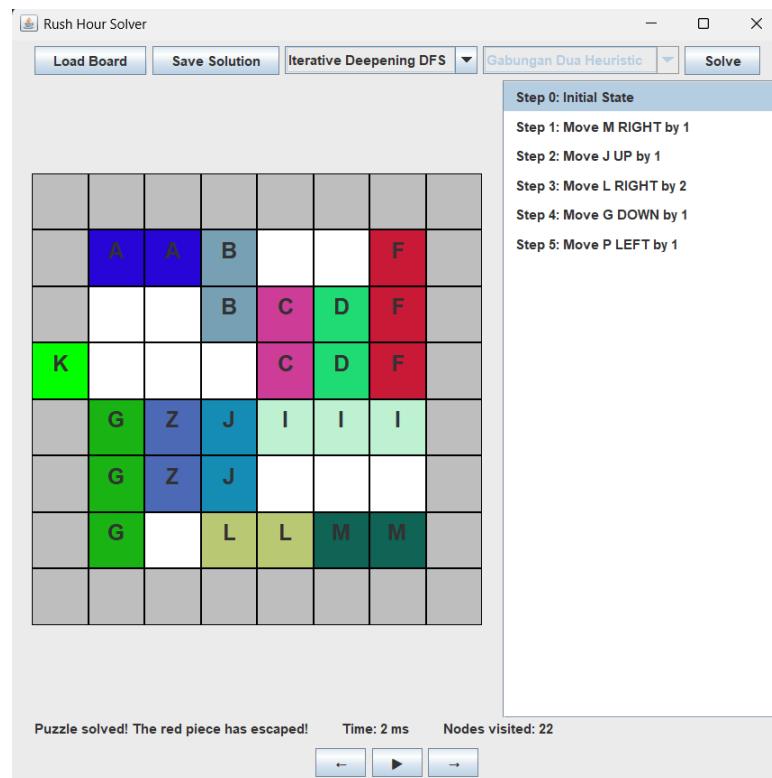
4.2.4 Algoritma IDDFS

- Test Case 1



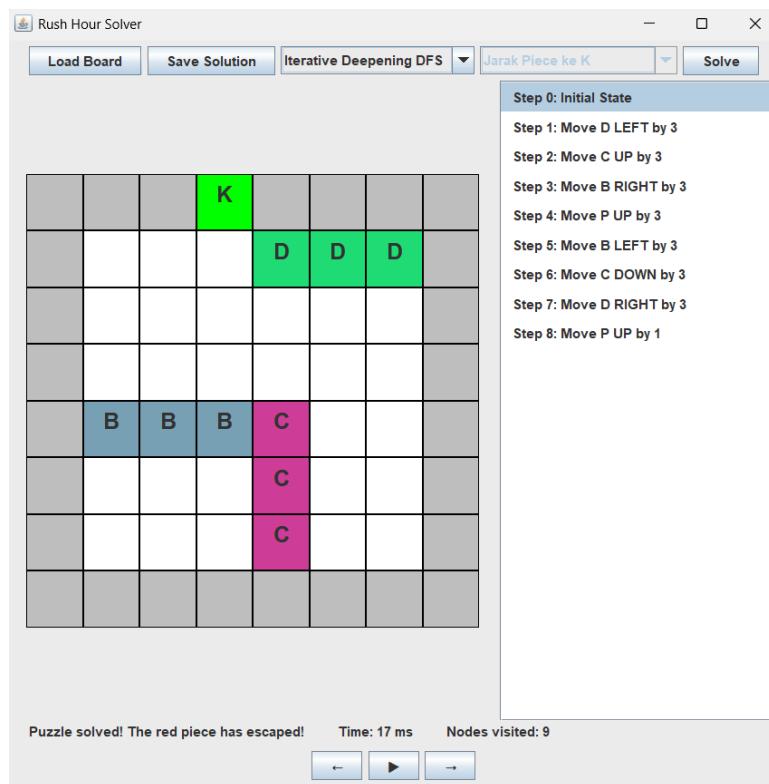
Gambar 41. Test Case IDDFS 1

- Test Case 2



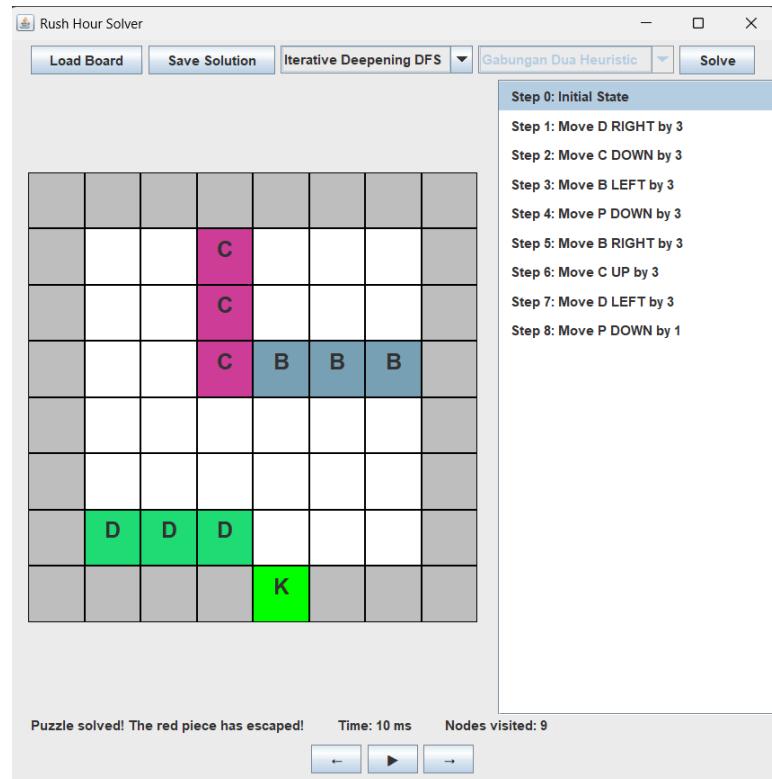
Gambar 42. Test Case IDDFS 2

- Test Case 3



Gambar 43. Test Case IDDFS 3

- Test Case 4



Gambar 44. Test Case IDDFS 4

4.3 Analisis Pengujian

Berdasarkan hasil pengujian yang telah kami lakukan terhadap empat test case, dari seluruh algoritma dan heuristik yang dipakai menunjukkan bahwa sudah bisa mendapatkan hasil yang sesuai dengan aturan permainan. Setiap algoritma dan heuristik juga memiliki hasilnya masing-masing. Oleh karena itu, berikut kami lampirkan analisis kami terhadap hasil pengujian.

4.3.1 Analisis Pengujian Uniform Cost Search (UCS)

Algoritma UCS merupakan algoritma pencarian yang menghasilkan solusi optimal, karena memilih jalur dengan biaya kumulatif terkecil menggunakan priority queue yang diurutkan berdasarkan total ongkos perjalanan.

Dari hasil pengujian terhadap empat test case, UCS mampu menemukan jalur terpendek secara konsisten. Namun, kekurangannya adalah jumlah node yang dikunjungi relatif banyak, karena algoritma ini akan mengeksplorasi semua kemungkinan dengan biaya rendah terlebih dahulu, meskipun arahnya tidak langsung menuju tujuan.

Kompleksitas Waktu:

- $O(b^d * \log(b^d))$ di mana:

- b adalah faktor percabangan (branching factor) atau rata-rata jumlah langkah yang mungkin per state
 - d adalah kedalaman solusi
 - $\log(b^d)$ berasal dari operasi pada priority queue
- Dalam konteks Rush Hour:
 - Faktor percabangan (b) tergantung pada jumlah kendaraan dan ukuran papan
 - Setiap kendaraan bisa bergerak beberapa langkah dalam satu gerakan

Kompleksitas Ruang:

- $O(b^d)$ untuk menyimpan semua node yang telah dikunjungi dalam HashSet
- Membutuhkan memori tambahan untuk priority queue yang menyimpan state yang akan dieksplorasi

4.3.2 Analisis Pengujian Greedy Best First Search (GBFS)

Algoritma GBFS menunjukkan performa yang cepat dalam menemukan solusi karena hanya mempertimbangkan nilai heuristik (estimasi jarak ke tujuan) tanpa memperhatikan biaya dari titik awal.

Walaupun jumlah node yang dikunjungi relatif lebih sedikit dibandingkan UCS dan A*, solusi yang dihasilkan tidak selalu optimal. Hal ini karena GBFS bisa terjebak pada optimum lokal yang tampak menjanjikan di awal, tetapi ternyata tidak mengarah pada solusi yang valid, atau bahkan ke jalan buntu.

Kompleksitas Waktu:

- $O(b^m * \log(b^m))$ di mana:
 - b adalah faktor percabangan
 - m adalah kedalaman maksimum dari ruang pencarian
 - $\log(b^m)$ untuk operasi priority queue
- Dalam konteks Rush Hour:
 - Lebih cepat dari UCS karena menggunakan heuristik untuk mengarahkan pencarian
 - Tidak menjamin solusi optimal

Kompleksitas Ruang:

- $O(b^m)$ untuk menyimpan node yang sudah dikunjungi
- Memori yang dibutuhkan lebih kecil dari A* karena hanya mempertimbangkan nilai heuristik

4.3.3 Analisis Pengujian A* Search

Algoritma A* menggabungkan kelebihan UCS dan GBFS dengan mempertimbangkan baik biaya kumulatif dari titik awal maupun estimasi heuristik ke tujuan.

Dari pengujian, A* menunjukkan hasil yang lebih efisien dibandingkan UCS dalam hal jumlah node yang dikunjungi, dan lebih optimal dibandingkan GBFS dalam hal solusi yang ditemukan. A* secara umum memberikan hasil seimbang antara optimalitas dan efisiensi pencarian.

Kompleksitas Waktu:

- $O(b^d * \log(b^d))$ di mana:
 - b adalah faktor percabangan
 - d adalah kedalaman solusi
 - Menggunakan kombinasi $g(n) + h(n)$
- $g(n)$ adalah biaya dari initial state ke state saat ini
- $h(n)$ adalah nilai heuristik ke goal state

Kompleksitas Ruang:

- $O(b^d)$ untuk menyimpan semua node dalam memori
- Membutuhkan memori lebih besar dari GBFS karena menyimpan informasi path cost

4.3.4 Analisis Pengujian Iterative Deepening Depth First Search (IDDFS)

Algoritma IDDFS menggabungkan pendekatan DFS dengan iterasi bertingkat terhadap kedalaman pencarian. IDDFS sangat baik dalam hal penggunaan memori karena DFS bersifat rekursif dan tidak menyimpan semua node di memori. Namun, karena pencarinya dilakukan berulang dari awal hingga kedalaman tertentu, maka IDDFS kurang efisien dalam waktu pencarian untuk solusi yang dalam.

Dari pengujian, IDDFS cenderung lebih lambat dan mengunjungi banyak node yang berulang, meskipun tetap dapat menemukan solusi jika diberikan batas kedalaman yang tepat.

Kompleksitas Waktu:

- $O(b^d)$ di mana:
 - b adalah faktor percabangan
 - d adalah kedalaman solusi
 - Meskipun mengunjungi level-level atas berulang kali, kompleksitas asimetrik tetapi $O(b^d)$ karena jumlah node di level terdalam mendominasi total node yang dikunjungi

Kompleksitas Ruang:

- $O(d)$ karena hanya perlu menyimpan path dari root ke node saat ini
- Membutuhkan memori paling sedikit di antara semua algoritma yang diimplementasi

4.3.5 Analisis Komparatif

Berdasarkan hasil pengujian terhadap beberapa test case, dapat dilihat bahwa setiap algoritma memiliki kelebihan dan kekurangan sebagai berikut:

- UCS unggul dalam hal optimalitas solusi, namun membutuhkan waktu lebih lama dan banyak node.
- GBFS sangat cepat dan mengunjungi sedikit node, tetapi seringkali memberikan solusi tidak optimal.
- A* seimbang antara efisiensi waktu dan kualitas solusi, dan menjadi algoritma yang paling serbaguna.
- IDDFS hemat memori dan mampu menemukan solusi, namun lambat karena banyak pengulangan pencarian pada kedalaman yang sama.
- Heuristik dengan menghitung jumlah mobil yang menghalangi lebih baik daripada heuristik jarak ataupun kombinasi.

BAB V

Implementasi Bonus

5.1 Algoritma Alternatif

Algoritma alternatif yang kami gunakan adalah algoritma Iterative Deepening Depth First Search. Algoritma Iterative Deepening Depth First Search (IDDFS) adalah algoritma pencarian dalam graf atau pohon yang menggabungkan kelebihan dari dua algoritma klasik: Depth First Search (DFS) dan Breadth First Search (BFS). Seperti DFS, IDDFS menggunakan sedikit memori karena hanya menyimpan jalur saat ini dalam memori (menggunakan rekursi atau stack), tetapi seperti BFS, IDDFS menjamin pencarian solusi dengan kedalaman minimum (optimal untuk pencarian tak berbobot). Algoritma ini bekerja dengan melakukan DFS secara bertahap hingga kedalaman tertentu (depth limit), kemudian mengulangi DFS dengan menambah batas kedalaman satu per satu hingga solusi ditemukan. Karena tiap level pencarian diulang dari awal, IDDFS mungkin tampak boros, namun dalam struktur seperti pohon pencarian, pengulangan ini tidak terlalu signifikan karena sebagian besar simpul berada di kedalaman terdalam. Cara kerja algoritma ini adalah:

- Tentukan batas kedalaman awal yang dimulai dari 0 dan tingkatkan secara bertahap.
- Lakukan pencarian DFS dengan Depth Limited Search (DLS), yaitu DFS yang berhenti jika mencapai kedalaman tertentu.
- Periksa apakah solusi ditemukan dalam batas kedalaman saat ini. Jika ya, hentikan dan kembalikan solusi. Jika tidak, tingkatkan batas kedalaman dan ulangi proses.
- Ulangi langkah 2 dan 3 sampai solusi ditemukan atau semua kemungkinan habis (dalam graf terbatas).

5.2 Heuristik Alternatif

Heuristik alternatif yang kami gunakan pada program kami adalah heuristic menggunakan jumlah mobil yang menutupi dan kombinasi antara kedua heuristik yaitu jarak mobil/piece yang menutupi ditambah jarak dari primary piece ke K. Heuristik pertama adalah heuristik jumlah mobil/piece yang menghalangi. Heuristik ini menghitung jumlah kendaraan (pieces) yang menghalangi jalur utama mobil utama menuju pintu keluar. Untuk mobil horizontal, ia menghitung berapa banyak kendaraan yang berada di antara mobil utama dan pintu keluar pada baris yang sama. Untuk mobil vertikal, ia menghitung kendaraan yang berada di antara mobil utama dan pintu keluar pada kolom yang sama. Semakin sedikit kendaraan penghalang, semakin dekat solusi.

Heuristik kedua adalah heuristik kombinasi. Heuristik ini merupakan gabungan dari dua nilai yaitu jarak mobil utama ke pintu keluar (pieceToDest) dan jumlah kendaraan

penghalang (countBlockingPieces). Nilai heuristik dihitung dari jarak ke pintu keluar ditambah jumlah penghalang. Kelebihan dari heuristik ini adalah lebih informatif karena memperhitungkan dua aspek sekaligus, yaitu seberapa jauh mobil utama dari pintu keluar dan seberapa banyak hambatan yang harus disingkirkan.

5.3 Graphical User Interface

Program kami menggunakan Java Swing untuk membangun antarmuka grafis yang interaktif dan mudah digunakan. Berikut fitur-fitur utama kami dalam GUI yang kami buat.

5.3.1 Tampilan Utama

Jendela utama menampilkan papan permainan Rush Hour, kontrol pemilihan algoritma, pemilihan heuristik, serta tombol-tombol aksi seperti Load, Solve, dan Save.

5.3.2 Load Board

Tombol "Load Board" digunakan untuk memuat file papan permainan dari komputer. Setelah file dimuat, papan akan divisualisasikan pada area utama.

5.3.3 Pemilihan Algoritma dan Heuristik

Pengguna dapat memilih algoritma pencarian (UCS, GBFS, A*, IDDFS) melalui dropdown. Jika memilih algoritma yang membutuhkan heuristik (GBFS atau A*), dropdown heuristik akan aktif dan dapat dipilih (misalnya jarak ke exit, jumlah penghalang, dan gabungan).

5.3.4 Solve dan Visualisasi Solusi

Tombol "Solve" akan menjalankan algoritma yang dipilih untuk mencari solusi. Setelah solusi ditemukan, langkah-langkah solusi akan ditampilkan dalam daftar di sisi kanan, dan papan akan divisualisasikan langkah demi langkah.

5.3.5 Kontrol Animasi

Terdapat tombol navigasi (\leftarrow , \rightarrow , II) untuk mengontrol animasi solusi:

- \leftarrow : Mundur ke langkah sebelumnya.
- II : Memulai atau menghentikan animasi otomatis.
- \rightarrow : Maju ke langkah berikutnya.

Pengguna juga dapat memilih langkah tertentu dari daftar untuk langsung melihat posisi papan pada langkah tersebut.

5.3.6 Status dan Statistik

Bagian bawah menampilkan status proses (misal: board loaded, solving, solution found), waktu eksekusi, dan jumlah node yang dikunjungi selama pencarian solusi.

5.3.7 Save Solution

Setelah solusi ditemukan, tombol "Save Solution" dapat digunakan untuk menyimpan langkah-langkah solusi ke file teks.

5.3.8 Visualisasi Papan

Papan divisualisasikan dengan warna berbeda untuk setiap kendaraan/piece, dan kendaraan utama ("P") diberi warna merah. Pintu keluar (exit) ditandai dengan warna hijau dan label "K".

BAB VI

Penutup

6.1 Kesimpulan

Melalui Tugas Kecil 3 mata kuliah IF2211 Strategi Algoritma ini, kelompok kami berhasil membangun sebuah aplikasi yang mampu memecahkan permasalahan puzzle Rush Hour secara otomatis menggunakan berbagai algoritma pencarian. Permasalahan Rush Hour dikonversi menjadi representasi papan dan state, di mana setiap langkah pergerakan kendaraan direpresentasikan sebagai transisi antar state.

Beberapa algoritma pencarian yang diimplementasikan dalam aplikasi ini adalah Uniform Cost Search (UCS), Greedy Best First Search (GBFS), A*, dan Iterative Deepening Depth First Search (IDDFS). Masing-masing algoritma memiliki keunggulan dan kelemahannya sendiri. UCS menjamin solusi optimal, GBFS dan A* memanfaatkan heuristic untuk mempercepat pencarian solusi, sedangkan IDDFS menggabungkan keunggulan DFS dan BFS dalam hal memori dan kelengkapan pencarian. Selain itu, aplikasi ini juga menyediakan antarmuka grafis (GUI) yang interaktif, sehingga pengguna dapat memuat papan, menjalankan solver, serta melihat visualisasi langkah-langkah solusi secara animasi.

Penggunaan struktur data seperti hash map untuk penyimpanan state, serta pemanfaatan fungsi hash dan equals pada board, sangat membantu dalam menghindari eksplorasi state yang sama secara berulang sehingga meningkatkan efisiensi algoritma. Dengan demikian, tugas ini tidak hanya menjadi latihan dalam mengimplementasikan berbagai algoritma pencarian yang telah dipelajari di kelas, tetapi juga mengintegrasikan konsep teori algoritma, pemrograman berorientasi objek, serta pengembangan aplikasi berbasis GUI untuk visualisasi solusi secara real-time.

6.2 Saran

Terdapat beberapa saran untuk pengembangan selanjutnya, diantaranya:

- Penambahan Algoritma Lain
- Optimasi Heuristik
- Peningkatan Antarmuka Pengguna
- Optimalisasi Program

6.3 Refleksi

Setelah mengerjakan tugas kecil ini, kami merasa lebih memahami materi tentang algoritma route finding. Kami merasa lebih mudah belajar karena kami harus memahami kode yang kami buat untuk menyelesaikan tugas kecil ini. Walaupun progress kami sedikit tertinggal dari teman-teman kami, kami merasa puas dengan apa yang sudah kami kerjakan.

Daftar Pustaka

- De-Yu, Chao. "Search Algorithm: Dijkstra's Algorithm and Uniform-Cost Search, with Python | by Chao De-Yu | TDS Archive." *Medium*, 6 December 2021, <https://medium.com/data-science/search-algorithm-dijkstras-algorithm-uniform-cost-search-with-python-ccbee250ba9>. Accessed 20 May 2025.
- Jain, Sandeep. "Greedy Best first search algorithm." *GeeksforGeeks*, 18 January 2024, <https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/>. Accessed 20 May 2025.
- Pedamkar, Priya. "Iterative Deepening Depth-First Search | Advantages and Disadvantages." *EDUCBA*, 28 July 2023, <https://www.educba.com/iterative-deepening-depth-first-search/>. Accessed 20 May 2025.
- Trivusi. "Algoritma A* (A Star): Pengertian, Cara Kerja, dan Kegunaannya." *Trivusi*, 20 January 2023, <https://www.trivusi.web.id/2023/01/algoritma-a-star.html>. Accessed 20 May 2025.
- Wikipedia. "A* search algorithm." *Wikipedia*, 2020, https://en.m.wikipedia.org/wiki/A*_search_algorithm. Accessed 20 May 2025.

Lampiran

- Link Repository:
https://github.com/rakdaf08/Tucil3_13523018_13523074
- Tabel Keselesaian Tugas

Poin	Ya	Tidak
1. Program berhasil dikompilasi tanpa kesalahan	V	
2. Program berhasil dijalankan	V	
3. Solusi yang diberikan program benar dan mematuhi aturan permainan	V	
4. Program dapat membaca masukan berkas .txt dan menyimpan solusi berupa print board tahap per tahap dalam berkas .txt	V	
5. [Bonus] Implementasi algoritma pathfinding alternatif	V	
6. [Bonus] Implementasi 2 atau lebih heuristik alternatif	V	
7. [Bonus] Program memiliki GUI	V	
8. Program dan laporan dibuat (kelompok) sendiri	V	