

Laporan Tugas Kecil 3
IF2211 Strategi Algoritma
Penyelesaian Puzzle Rush Hour Menggunakan Algoritma
Pathfinding



disusun oleh:
Bevinda Vivian
13523120

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

Kata Pengantar

Saya mengucapkan syukur kepada Tuhan Yang Maha Esa atas rahmat dan petunjuk-Nya telah mengarahkan saya untuk menyelesaikan laporan ini tepat waktu. Dokumen ini merupakan sebuah laporan yang disusun untuk memenuhi tugas kecil 3 dalam mata kuliah IF2211 Strategi Algoritma. Selain itu, laporan ini juga bertujuan untuk meningkatkan pemahaman saya mengenai penerapan Algoritma *Pathfinding* dengan menggunakan bahasa pemrograman Java.

Terima kasih saya ucapkan kepada Bapak Dr. Ir. Rinaldi, M.T. yang telah menjadi dosen pengampu mata kuliah di kelas K-02 IF2211 Strategi Algoritma. Selain itu saya juga ingin mengungkapkan terima kasih kepada kakak-kakak asisten dari Laboratorium Ilmu dan Rekayasa Komputasi yang senantiasa membantu saya dalam Tugas Kecil 3 Strategi Algoritma. Saya sadar bahwa laporan ini masih jauh dari kata sempurna. Oleh karena itu, jika terdapat kesalahan dalam penulisan atau ketidaksesuaian dalam materi yang saya sampaikan dalam laporan ini, saya mohon maaf. Tentunya saya juga sangat mengharapkan saran dan kritik yang membangun untuk meningkatkan kualitas tugas kecil ataupun laporan ini.

Bandung, 20 Mei 2025

A handwritten signature in black ink, appearing to read 'Bevinda Vivian', enclosed within a light gray rectangular border.

Bevinda Vivian

13523120

Daftar Isi

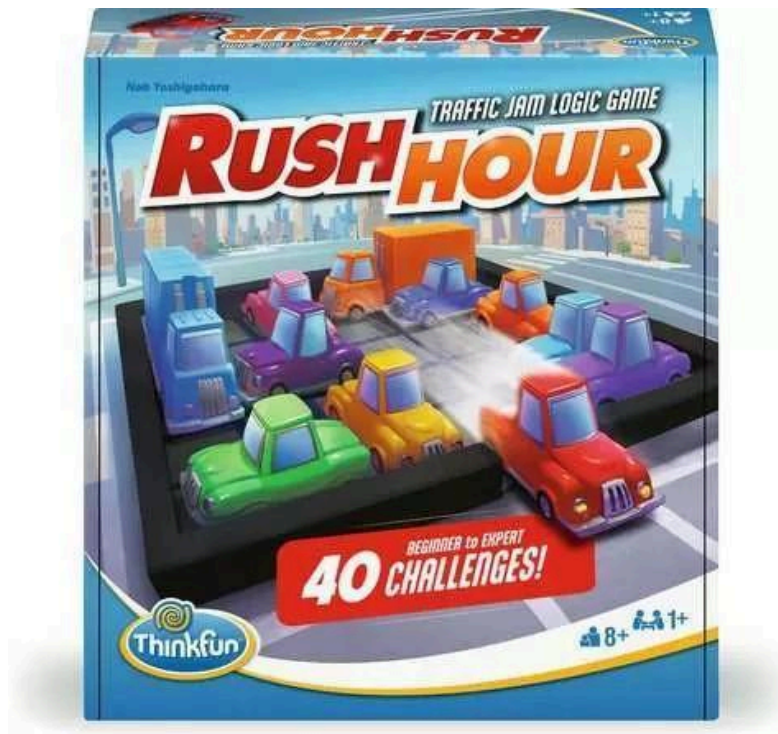
Kata Pengantar	2
Daftar Isi	3
Daftar Gambar	5
Bab 1 Deskripsi Masalah	6
Bab 2 Teori Singkat	13
2.1 Algoritma Pathfinding	13
Bab 3 Algoritma Program	15
3.1 Uniform Cost Search (UCS)	15
3.2 Greedy Best First Search	16
3.3 A*	17
3.4 Beam Search	18
3.5 Heuristik	20
3.6 Graphical User Interface (GUI)	21
3.7 Analisis Algoritma	22
Bab 4 Source Code	26
4.1 Piece.java	26
4.2 Board.java	28
4.3 State.java	37
4.4 AStar.java	47
4.5 BeamSearch.java	49
4.6 GBFS.java	52
4.7 UCS.java	54
4.8 Heuristic.java	56
4.9 SearchResult.java	60
4.10 Main.java	60
4.11 App.java	62
Bab 5 Eksperimen	78
5.1 Pengujian test1_result.txt	78
5.2 Pengujian test1.txt	78
5.3 Pengujian test2.txt	79
5.4 Pengujian test3.txt	81
5.5 Pengujian test4.txt	82
Bab 6 Lampiran	85

Daftar Gambar

Gambar 1. Rush Hour Puzzle	5
Gambar 2. Awal Permainan Game Rush Hour	7
Gambar 3. Gerakan Pertama Game Rush Hour	7
Gambar 4. Gerakan Kedua Game Rush Hour	8
Gambar 5. Pemain Menyelesaikan Permainan	8
Gambar 6. Validasi program saat memuat format yang salah	77
Gambar 7. Program berhasil mencari solusi	78
Gambar 8. Program berhasil menyimpan solusi dari kasus test1.txt	78
Gambar 9. Program berhasil memuat file test2.txt	79
Gambar 10. Program berhasil menemukan solusi dari kasus test2.txt	80
Gambar 11. Program berhasil menemukan solusi dari kasus file test3.txt (tidak ada solusi)	81
Gambar 12. Program berhasil menemukan solusi dari kasus file test4.txt	82

Bab 1

Deskripsi Masalah



Gambar 1. Rush Hour Puzzle

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

Rush Hour adalah sebuah permainan puzzle logika berbasis grid yang menantang pemain untuk menggeser kendaraan di dalam sebuah kotak (biasanya berukuran 6x6) agar mobil utama (biasanya berwarna merah) dapat keluar dari kemacetan melalui pintu keluar di sisi papan. Setiap kendaraan hanya bisa bergerak lurus ke depan atau ke belakang sesuai dengan

orientasinya (horizontal atau vertikal), dan tidak dapat berputar. Tujuan utama dari permainan ini adalah memindahkan mobil merah ke pintu keluar dengan jumlah langkah seminimal mungkin.

Komponen penting dari permainan Rush Hour terdiri dari:

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

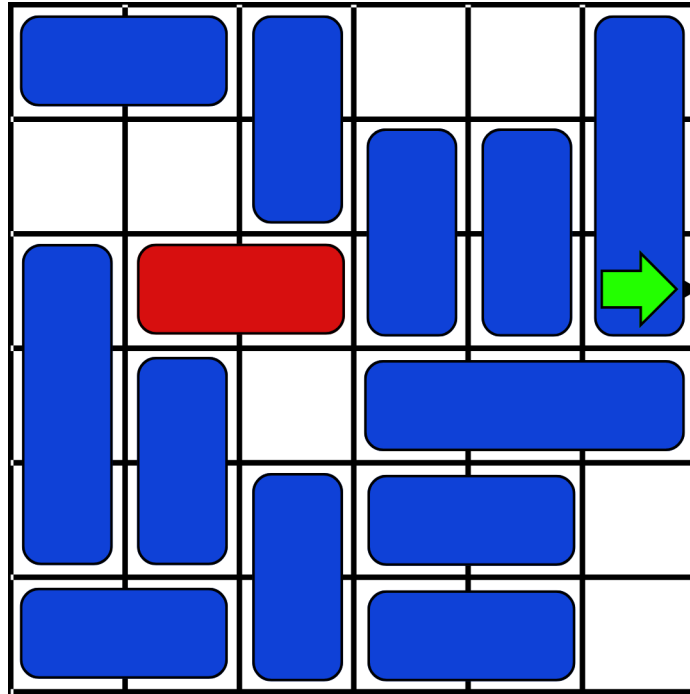
Papan terdiri atas *cell*, yaitu sebuah *singular point* dari papan. Sebuah *piece* akan menempati *cell-cell* pada papan. Ketika permainan dimulai, semua *piece* telah diletakkan di dalam papan dengan konfigurasi tertentu berupa lokasi *piece* dan *orientasi*, antara *horizontal* atau *vertikal*.

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

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

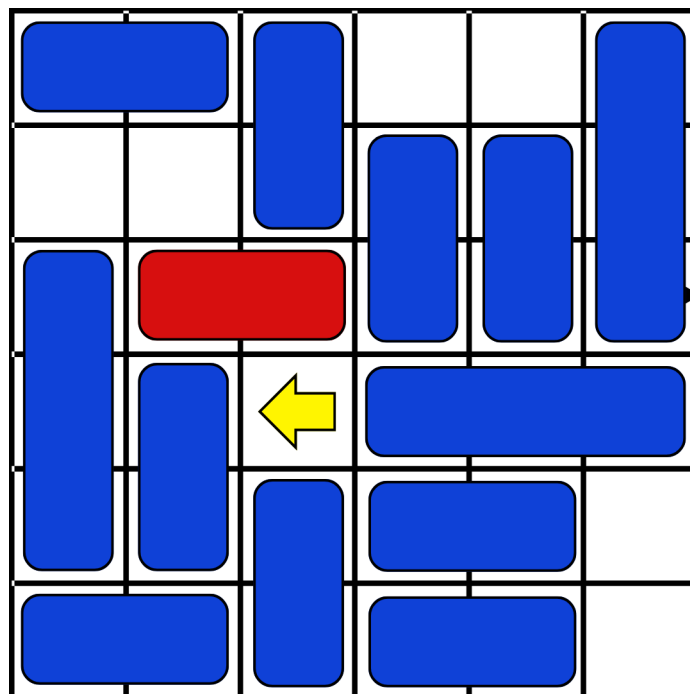
Ilustrasi kasus :

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

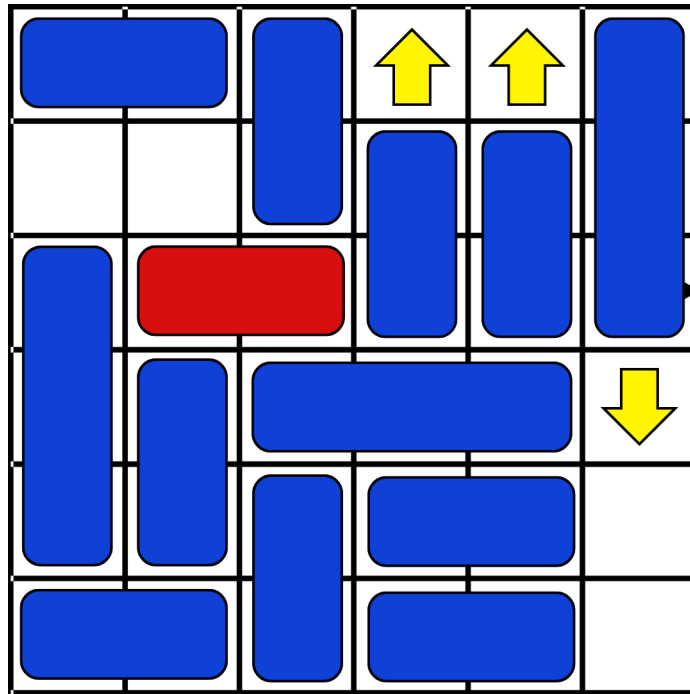


Gambar 2. Awal Permainan Game Rush Hour

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

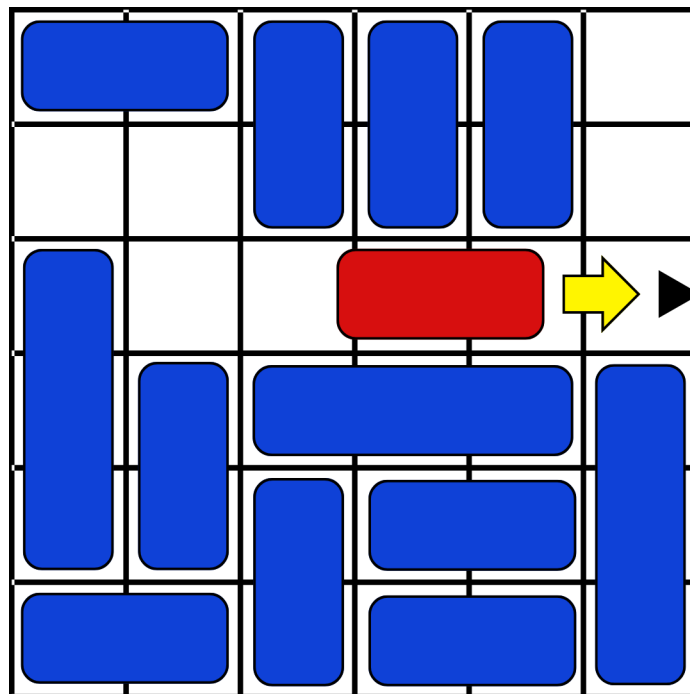


Gambar 3. Gerakan Pertama Game Rush Hour




Gambar 4. Gerakan Kedua Game Rush Hour

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



Gambar 5. Pemain Menyelesaikan Permainan

Agar lebih jelas, silahkan amati video cara bermain berikut:

 [The New Rush Hour by ThinkFun!](#)

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.

Alur Program:

1. **[INPUT] konfigurasi permainan/test case** dalam format ekstensi **.txt**. File *test case* tersebut berisi:
 1. **Dimensi Papan** terdiri atas dua buah variabel **A** dan **B** yang membentuk papan berdimensi **AxB**
 2. **Banyak *piece* BUKAN *primary piece*** direpresentasikan oleh variabel integer **N**.
 3. **Konfigurasi papan** yang mencakup penempatan *piece* dan *primary piece*, serta lokasi *pintu keluar*. *Primary Piece* dilambangkan dengan huruf **P** dan *pintu keluar* dilambangkan dengan huruf **K**. *Piece* dilambangkan dengan huruf dan karakter selain **P** dan **K**, dan huruf/karakter berbeda melambangkan *piece* yang berbeda. *Cell* kosong dilambangkan dengan karakter **'.'** (titik). (**Catatan:** ingat bahwa *pintu keluar* pasti berada di *dinding* papan dan sejajar dengan orientasi *primary piece*)

File .txt yang akan dibaca memiliki format sebagai berikut:

```
A B
N
konfigurasi_papan
```

Contoh Input

```
6 6
12
AAB..F
..BCDF
GPPCDFK
GH.III
GHJ...
LLJMM.
```

keterangan: “K” adalah pintu keluar, “P” adalah primary piece, Titik (“.”) adalah cell kosong.

Contoh konfigurasi papan lain yang mungkin berdasarkan letak *pintu keluar* (X adalah *piece/cell random*)

K	XXX	XXX
XXX	KXXX	XXX
XXX	XXX	XXX
XXX		K

2. [INPUT] algoritma *pathfinding* yang digunakan
3. [INPUT] *heuristic* yang digunakan (**bonus**)
4. [OUTPUT] Banyaknya **gerakan** yang diperiksa (alias banyak ‘node’ yang dikunjungi)
5. [OUTPUT] Waktu eksekusi program
6. [OUTPUT] konfigurasi *papan* pada setiap tahap pergerakan/pergeseran. Output ini tidak harus diimplementasi apabila mengerjakan *bonus output GUI*. **Gunakan print berwarna** untuk menunjukkan pergerakan *piece* dengan jelas. Cukup mewarnakan *primary piece*, *pintu keluar*, dan *piece yang digerakkan* saja (boleh dengan *highlight* atau *text color*). Pastikan ketiga komponen tersebut memiliki warna berbeda.

Format sekuens adalah sebagai berikut:

```
Papan Awal
[konfigurasi_papan_awal]

Gerakan 1: [piece]-[arah gerak]
[konfigurasi_papan_gerakan_1]
```

Gerakan 2: [piece]-[arah gerak]
[konfigurasi_papan_gerakan_2]

Gerakan [N]: [piece]-[arah gerak]
[konfigurasi_papan_gerakan_N]
dst

Contoh Output

```
Papan Awal
AAB..F
..BCDF
GPPCDFK
GH.III
GHJ...
LLJMM.

Gerakan 1: I-kiri
AAB..F
..BCDF
GPPCDFK
GH.III.
GHJ...
LLJMM.

Gerakan 2: F-bawah
AAB..F
..BCDF
GPPCDFK
GHIIIF
GHJ...
LLJMM.
```

Keterangan: **hanya sebagai contoh**. Pastikan output jelas dan mudah dimengerti. Warna dan highlight hanya untuk menunjukkan perubahan.

7. **[OUTPUT] animasi** gerakan-gerakan untuk mencapai solusi (**bonus GUI**).

Bab 2

Teori Singkat

2.1 Algoritma *Pathfinding*

Uniform Cost Search (UCS) adalah algoritma pencarian yang mengutamakan jalur dengan *cost* paling rendah dari simpul awal ke simpul tertentu. UCS merupakan pengembangan dari Breadth-First Search (BFS), namun tidak mengutamakan jumlah langkah terkecil, melainkan total path cost dari awal. Dengan kata lain, UCS selalu mengembangkan node yang memiliki nilai $g(n)$ (cost dari awal ke node n) paling kecil. UCS cocok untuk masalah di mana setiap aksi memiliki *cost* berbeda, dan menjamin solusi optimal asalkan semua *cost* lintasan non-negatif. Dalam konteks tugas kecil ini, UCS akan mengeksplorasi semua kemungkinan jalur berdasarkan jumlah langkah minimum tanpa mempertimbangkan estimasi jarak ke goal (jalan keluar), sehingga bisa menjadi tidak efisien ketika ruang pencarian sangat besar.

Greedy Best First Search (GBFS) adalah algoritma pencarian heuristik yang memilih node berdasarkan nilai estimasi jarak ke tujuan, yaitu $h(n)$. Algoritma ini berfokus pada seberapa dekat sebuah node terlihat terhadap goal, tanpa memperhatikan *cost* tempuh sejauh ini ($g(n)$). Karena hanya mengandalkan estimasi heuristik, Greedy Best First Search dapat sangat cepat dalam menemukan solusi, namun tidak menjamin solusi tersebut adalah yang paling optimal. Dalam konteks tugas kecil ini, algoritma ini bisa saja mengambil jalur yang terlihat menjanjikan namun ternyata tidak efisien, atau bahkan gagal menemukan solusi jika terjebak di local minima. Keunggulannya adalah kecepatan eksplorasi awal, tetapi dengan risiko solusi tidak optimal dan ketidaklengkapan.

A* Search adalah algoritma pencarian heuristik yang menggabungkan keunggulan UCS dan Greedy Best First Search. Ia menggunakan fungsi evaluasi $f(n) = g(n) + h(n)$, yaitu jumlah dari *cost* tempuh dari awal ke node n dan estimasi *cost* dari n ke tujuan. Dengan pendekatan ini, A* menyeimbangkan antara eksplorasi jalur murah dan jalur yang menjanjikan menuju goal. Jika heuristik $h(n)$ yang digunakan adalah admissible (tidak pernah melebihi *cost* sebenarnya ke goal) dan consistent (memenuhi syarat segitiga), maka A* menjamin solusi yang ditemukan adalah solusi optimal. Dalam konteks tugas kecil ini, A* cenderung lebih efisien dibanding UCS karena lebih selektif dalam mengembangkan node, dan lebih andal dibanding GBFS karena tetap mempertimbangkan *cost* yang telah ditempuh.

Beam Search adalah versi terbatas dari algoritma Best First Search yang hanya mempertahankan sejumlah simpul terbaik (disebut beam width) pada setiap langkah perluasan. Artinya, dari semua node yang mungkin dikembangkan pada suatu level, hanya beberapa node dengan nilai heuristik terbaik yang akan dilanjutkan eksplorasinya. Beam Search mengorbankan kelengkapan dan optimalitas demi efisiensi memori dan waktu, sehingga cocok untuk masalah dengan ruang pencarian sangat besar di mana solusi mendekati optimal sudah cukup. Dalam konteks tugas kecil ini, Beam Search bisa sangat cepat dalam menemukan solusi, namun ada risiko tinggi untuk melewati jalur terbaik jika beam width-nya terlalu kecil. Algoritma ini sering digunakan dalam aplikasi praktis seperti pemrosesan bahasa alami atau pengenalan suara, di mana efisiensi sangat penting.

Pranala repository: https://github.com/bevindav/Tucil3_13523120/

Bab 3

Algoritma Program

3.1 *Uniform Cost Search* (UCS)

Pseudocode:

```
function UCS(initialState):
    create empty PriorityQueue frontier ordered by path cost
    create empty Set explored
    add initialState to frontier

    while frontier is not empty:
        current = remove lowest-cost state from frontier

        if current is a goal state:
            return solution

        add current to explored

        for each next state from current:
            if next is not in explored and not in frontier:
                add next to frontier
            else if next is in frontier with higher cost:
                replace that frontier state with next

    return failure
```

UCS merupakan algoritma pencarian berbasis *cost* yang secara sistematis menjelajahi ruang state dengan memprioritaskan node berdasarkan *cost* kumulatif perjalanan dari state awal. UCS berjalan dengan langkah-langkah berikut:

- a. UCS dimulai dengan state awal yang dimasukkan ke dalam priority queue (frontier) dengan *cost* awal 0. Priority queue mengurutkan state berdasarkan *cost* kumulatif terendah.
- b. Selama frontier tidak kosong, algoritma melakukan langkah-langkah berikut
 - Mengambil state dengan *cost* terendah dari frontier (menggunakan *poll()* pada PriorityQueue).
 - Memeriksa apakah state tersebut adalah goal state. Jika ya, solusi ditemukan dan algoritma berhenti.
 - Menambahkan state saat ini ke *explored* untuk menandai bahwa state ini telah diperiksa.

- Membangkitkan semua state berikutnya yang dapat dicapai dari state saat ini (menggunakan getNextStates()).
- c. Untuk setiap state berikutnya yang dibangkitkan
 - Jika state belum pernah dieksplorasi dan belum ada di frontier, tambahkan ke frontier.
 - Jika state sudah ada di frontier tetapi dengan *cost* yang lebih tinggi, update state tersebut dengan *cost* yang lebih rendah.
- d. UCS berhenti ketika goal state ditemukan atau ketika frontier kosong (tidak ada solusi).

UCS menjamin solusi dengan *cost* terendah karena state dengan *cost* kumulatif terendah selalu diproses terlebih dahulu. Dalam implementasi, `compareTo` pada class `State` memastikan bahwa `PriorityQueue` mengurutkan state berdasarkan *cost* (`getCost()`).

3.2 Greedy Best First Search

Pseudocode:

```
function GBFS(initialState):
    create empty PriorityQueue frontier ordered by heuristic value
    create empty Set visited
    add initialState to frontier

    while frontier is not empty:
        current = remove state with lowest heuristic from frontier

        if current is already visited:
            continue

        mark current as visited

        if current is goal state:
            return solution

        for each next state from current:
            if next is not in visited:
                set parent of next to current
                add next to frontier

    return failure
```

Greedy Best First Search merupakan algoritma *informed search* yang memprioritaskan node berdasarkan estimasi *cost* untuk mencapai goal dari node tersebut, tanpa mempertimbangkan *cost* yang telah dikeluarkan. GBFS berjalan dengan langkah-langkah berikut:

- a. GBFS dimulai dengan state awal yang dimasukkan ke dalam priority queue. Priority queue mengurutkan state berdasarkan nilai heuristik terkecil (estimasi jarak ke goal).
- b. Selama queue tidak kosong:
 - Mengambil state dengan nilai heuristik terendah dari queue.
 - Memeriksa apakah state sudah pernah dikunjungi. Jika ya, lanjutkan ke state berikutnya.
 - Menandai state saat ini sebagai dikunjungi dan menambahkan ke himpunan visited.
 - Memeriksa apakah state ini adalah goal state. Jika ya, solusi ditemukan dan algoritma berhenti.
- c. Untuk setiap state berikutnya yang dapat dicapai, jika state tersebut belum dikunjungi, atur state saat ini sebagai parent dari state berikutnya. Tambahkan state berikutnya ke queue.
- d. GBFS berhenti ketika goal state ditemukan atau ketika queue kosong.

GBFS menggunakan komparator khusus (gbfsComparator) yang membandingkan state berdasarkan nilai heuristik saja (getHeuristic()). Ini berbeda dari UCS yang menggunakan *cost* kumulatif.

3.3 A*

Pseudocode:

```
function AStar(initialState):
    create empty PriorityQueue openSet ordered by  $f(n) = g(n) + h(n)$ 
    create empty Map bestFScore

    add initialState to openSet
    store f-score of initialState in bestFScore

    while openSet is not empty:
        current = remove state with lowest f-score from openSet

        if current is goal state:
```



```

    return solution

    for each next state from current:
        g = cost of next
        f = g + heuristic of next

        if next is not in bestFScore or f < bestFScore[next]:
            update bestFScore[next] to f
            add next to openSet

    return failure

```

A* Search menggabungkan pendekatan UCS dan GBFS dengan memprioritaskan node berdasarkan jumlah *cost* sebenarnya untuk mencapai node $g(n)$ dan estimasi *cost* untuk mencapai goal dari node tersebut $h(n)$. A* berjalan dengan langkah-langkah berikut:

- a. Algoritma dimulai dengan memasukkan state awal ke dalam openSet (priority queue) dan menyimpan nilai $f(n) = g(n) + h(n)$ dalam map bestFScore.
- b. Selama openSet tidak kosong:
 - Mengambil state dengan nilai $f(n)$ terendah dari openSet.
 - Memeriksa apakah state ini adalah goal state. Jika ya, solusi ditemukan.
- c. Untuk setiap state berikutnya yang dapat dicapai hitung $g(n)$ (*cost* sebenarnya) dan $f(n) = g(n) + h(n)$ untuk state tersebut. Jika state belum ada di bestFScore atau memiliki nilai $f(n)$ yang lebih baik dari yang tercatat, update nilai $f(n)$ dan tambahkan state ke openSet.
- d. A* berhenti ketika goal state ditemukan atau tidak ada solusi yang mungkin.

A* menggunakan komparator (aStarComparator) yang membandingkan state berdasarkan $f(n) = g(n) + h(n)$, yang menggabungkan *cost* kumulatif dan nilai heuristik. Implementasi ini tidak menggunakan himpunan closed seperti implementasi A* tradisional, tetapi menggunakan map bestFScore untuk melacak state dengan nilai $f(n)$ terbaik.

3.4 Beam Search

Pseudocode:

```

unction BeamSearch(initialState, beamWidth):
    create empty Set visited
    create List currentBeam containing initialState

```

```

while currentBeam is not empty:
    create empty List nextBeam

    for each current state in currentBeam:
        if current is goal state:
            return solution

        mark current as visited

        for each successor of current:
            if successor is not visited:
                add successor to nextBeam

    if nextBeam is empty:
        break

    sort nextBeam by evaluation function ( $f = g + h$ )

    if size of nextBeam > beamWidth:
        truncate nextBeam to keep only beamWidth best states

    currentBeam = nextBeam

return failure

```

Beam Search adalah algoritma pencarian yang membatasi jumlah state yang dipertimbangkan pada setiap level pencarian, sehingga lebih efisien dalam penggunaan memori. Algoritma ini berjalan dengan langkah-langkah berikut:

- a. BeamSearch dimulai dengan state awal dalam currentBeam (list of states) dan himpunan visited kosong.
- b. Algoritma berjalan dalam iterasi, dengan batas maksimum iterasi (untuk mencegah pencarian yang terlalu lama). Untuk setiap state dalam currentBeam:
 - Memeriksa apakah state adalah goal state. Jika ya, solusi ditemukan.
 - Menandai state sebagai dikunjungi.
 - Membangkitkan semua successor dari state tersebut.
- c. Setelah semua state di currentBeam diproses, jika tidak ada successor valid, pencarian berhenti. Successor diurutkan berdasarkan fungsi evaluasi ($f(n) = g(n) + h(n)$).
- d. Jika jumlah successor melebihi beam width, hanya state terbaik sebanyak beam width yang dipertahankan.
- e. BeamSearch berhenti ketika solusi ditemukan, tidak ada successor valid, atau mencapai jumlah iterasi maksimum.

Beam Search menggunakan parameter beam width untuk membatasi jumlah state yang dipertimbangkan, mengurangi penggunaan memori dengan trade-off potensial kehilangan solusi optimal. Algoritma juga menggunakan fungsi `getBoardKey()` untuk mengkonversi board ke string unik, memudahkan pengecekan state yang sudah dikunjungi.

3.5 Heuristik

a. Manhattan Distance

Manhattan Distance menghitung jarak langsung dari kendaraan utama (primary piece) ke pintu keluar. Implementasinya mempertimbangkan orientasi kendaraan (horizontal atau vertikal) dan posisi pintu keluar. Langkah perhitungannya:

- Identifikasi kendaraan utama ('P') dan board.
- Jika kendaraan utama berorientasi horizontal, hitung jarak dari ujung kanan kendaraan ke pintu keluar di kanan, atau hitung jarak dari ujung kiri kendaraan ke pintu keluar di kiri.
- Jika kendaraan utama berorientasi vertikal, hitung jarak dari ujung bawah kendaraan ke pintu keluar di bawah, atau hitung jarak dari ujung atas kendaraan ke pintu keluar di atas.

b. Blocking Vehicles

Heuristik Blocking Vehicles menghitung jumlah kendaraan yang menghalangi jalur langsung kendaraan utama ke pintu keluar. Langkah perhitungannya:

- Identifikasi kendaraan utama dan orientasinya.
- Jika kendaraan berorientasi horizontal, untuk pintu keluar di kanan hitung kendaraan dari ujung kanan kendaraan utama ke pintu keluar. Untuk pintu keluar di kiri, hitung kendaraan dari ujung kiri kendaraan utama ke pintu keluar.
- Jika kendaraan berorientasi vertikal, untuk pintu keluar di bawah, hitung kendaraan dari ujung bawah kendaraan utama ke pintu keluar. Untuk pintu keluar di atas, hitung kendaraan dari ujung atas kendaraan utama ke pintu keluar.

c. Advanced Blocking

Advanced Blocking memperluas heuristik Blocking Vehicles dengan mempertimbangkan kesulitan pemindahan kendaraan penghalang. Langkah perhitungannya:

- Hitung jumlah kendaraan penghalang dasar dengan Blocking Vehicles.
- Tambahkan penalti *movability* untuk kendaraan yang sulit dipindahkan.
- Identifikasi kendaraan penghalang yang berorientasi vertikal.
- Periksa apakah kendaraan tersebut dihalangi di atas atau bawah. Jika dihalangi di kedua sisi, tambahkan penalti 3 (sangat sulit dipindahkan). Jika dihalangi di satu sisi, tambahkan penalti 1 (cukup sulit dipindahkan).

d. Combined Heuristic

Combined Heuristic menggabungkan ketiga heuristik sebelumnya dengan pembobotan yang berbeda yaitu Manhattan Distance (bobot 1), Blocking Vehicles (bobot 2), Advanced Blocking (bobot dimodifikasi dengan fungsi akar kuadrat dan faktor 1.5). Pada combined ini dihitung dengan rumus $\text{ManhattanDistance} + \text{BlockingVehicles} * 2 + (\text{sqrt}(\text{AdvancedBlocking}) * 1.5)$.

3.6 Graphical User Interface (GUI)

Pada tugas kecil 3 ini, saya menerapkan Graphical User Interface (GUI) yang diimplementasikan menggunakan Java Swing. GUI dirancang agar mudah digunakan dan informatif, serta memenuhi kebutuhan input dan output secara grafis sesuai spesifikasi tugas kecil 3.

Pada bagian atas aplikasi, terdapat panel kontrol yang terdiri dari beberapa komponen utama. Pengguna dapat memuat papan permainan (board) dari file menggunakan tombol "Load Traffic". Setelah papan dimuat, pengguna dapat memilih algoritma pencarian yang diinginkan (UCS, GBFS, A*, atau Beam Search) melalui dropdown menu. Jika algoritma yang dipilih mendukung heuristic, dropdown menu heuristic akan otomatis aktif, sehingga pengguna dapat memilih fungsi heuristic yang ingin digunakan (misalnya Manhattan Distance, Blocking Vehicles, Advanced Blocking, atau Combined). Setelah konfigurasi selesai, pengguna dapat

menekan tombol "Solve" untuk memulai proses pencarian solusi. Terdapat juga tombol "Save Solution" untuk menyimpan solusi yang ditemukan ke dalam file.

Bagian utama aplikasi menampilkan papan permainan secara grafis melalui komponen GamePanel. Setiap kendaraan pada papan digambarkan dengan warna yang berbeda, dan posisi exit (keluar) juga ditandai secara jelas. Setelah solusi ditemukan, pengguna dapat memutar animasi langkah-langkah solusi dari awal hingga akhir menggunakan tombol "Play/Pause" dan "Reset". Kecepatan animasi dapat diatur dengan slider, sehingga pengguna dapat menyesuaikan kecepatan visualisasi sesuai keinginan. Selama animasi berlangsung, langkah yang sedang dijalankan akan ditampilkan secara real-time, dan progres solusi divisualisasikan dengan progress bar serta label step count.

Selain itu, GUI juga menyediakan area log di bagian bawah aplikasi untuk menampilkan pesan-pesan penting, seperti status pemuatan board, hasil pencarian solusi, waktu eksekusi, jumlah node yang dikunjungi, dan langkah-langkah solusi. Status proses juga ditampilkan secara jelas melalui label status di panel kontrol.

3.7 Analisis Algoritma

$g(n)$ adalah *cost* kumulatif untuk mencapai node n dari node awal. Ini merepresentasikan *cost* sebenarnya yang telah dikeluarkan dalam perjalanan pencarian. Dalam implementasi kode, $g(n)$ direpresentasikan oleh method `getCost()` pada class `State`. $f(n)$ adalah fungsi evaluasi yang mengestimasi total *cost* jalur melalui node n dari start ke goal. Dalam A^* , $f(n) = g(n) + h(n)$, di mana $h(n)$ adalah estimasi *cost* dari node n ke goal. Dalam implementasi kode, $h(n)$ direpresentasikan oleh method `getHeuristic()` dan $f(n)$ dihitung dalam `aStarComparator`.

Sebuah heuristik dikatakan admissible jika untuk setiap node n , $h(n) \leq h^*(n)$, di mana $h^*(n)$ adalah *cost* sebenarnya minimum untuk mencapai goal dari node n . Dengan kata lain, heuristik tidak pernah overestimate *cost* sebenarnya. Khususnya pada heuristik yang kucoba terapkan yaitu:

- a. Manhattan Distance.

Sangat admissible karena merepresentasikan jarak fisik minimum yang

harus ditempuh kendaraan utama untuk mencapai pintu keluar, tanpa mempertimbangkan penghalang.

b. Blocking Vehicles

Admissible karena setiap kendaraan penghalang memerlukan minimal satu langkah untuk dipindahkan. Bahkan dalam kasus terbaik, *cost* untuk menghilangkan penghalang tidak kurang dari jumlah kendaraan penghalang tersebut.

c. Advanced Blocking

Potensial admissible jika penalti yang ditambahkan untuk kesulitan pemindahan kendaraan tidak melebihi *cost* sebenarnya. Penalti 1 untuk kendaraan dengan satu penghalang dan 3 untuk kendaraan dengan penghalang di kedua sisi tampaknya masuk akal, namun keadmisibilitasnya bergantung pada karakteristik persis dari permainan Rush Hour.

d. Combined

Admissibility bergantung pada pembobotan. Penggunaan akar kuadrat untuk Advanced Blocking mengurangi potensi overestimation, namun penggandaan nilai Blocking Vehicles bisa menyebabkan non-admissibility jika beberapa kendaraan penghalang dapat dipindahkan bersamaan dalam satu langkah.

Dalam Rush Hour yang bersifat uniform-cost (setiap gerakan memiliki cost sama), UCS dan BFS memiliki karakteristik:

- a. Secara teori, jika semua langkah memiliki cost yang sama (misalnya 1), UCS akan memperluas node dalam urutan yang sama dengan BFS. Namun, implementasi aktual dapat berbeda karena:
 - UCS menggunakan PriorityQueue yang mengurutkan berdasarkan cost, sedangkan BFS menggunakan FIFO queue.
 - Ketika terdapat multiple nodes dengan cost yang sama, urutan ekspansi dapat bergantung pada implementasi PriorityQueue.
- b. Kedua algoritma akan menemukan solusi dengan jumlah langkah minimum karena keduanya menemukan jalur dengan cost terendah dalam grafik dengan cost seragam.
- c. Keduanya memiliki kompleksitas waktu dan ruang eksponensial dalam kasus terburuk, tetapi BFS lebih efisien dalam implementasi karena tidak memerlukan priority queue.

Pada dasarnya, untuk permainan Rush Hour dengan *cost* langkah seragam, UCS dan BFS akan menghasilkan solusi optimum yang sama, tetapi bisa jadi dengan urutan pembangkitan node yang sedikit berbeda.

Berdasarkan materi yang didapatkan dari salindia kuliah, A* lebih efisien daripada UCS jika $h(n)$ adalah heuristik yang admissible dan consistent (monotone). Dalam Rush Hour, hal ini terbukti dengan:

- a. A* menggunakan informasi heuristik yang mengarahkan pencarian ke arah tujuan, sedangkan UCS mencari secara "blind" berdasarkan cost kumulatif saja.
- b. A* akan mengekskansi subset dari node yang diekspansi UCS untuk menemukan solusi optimal.
- c. Rush Hour yang memiliki branching factor tinggi (banyak gerakan mungkin pada setiap state), A* dengan heuristik yang informatif seperti kombinasi Manhattan Distance dan Blocking Vehicles dapat secara signifikan mengurangi ruang pencarian.
- d. Kedua algoritma menjamin solusi optimal, tetapi A* cenderung menemukannya lebih cepat dengan mengevaluasi node yang lebih sedikit.
- e. A* memerlukan perhitungan heuristik tambahan untuk setiap node, yang bisa menjadi overhead komputasi, tetapi efisiensi pencarian biasanya mengkompensasi overhead ini.

Dalam Rush Hour, Greedy Best First Search tidak menjamin solusi optimal. Karena GBFS hanya mempertimbangkan nilai heuristik $h(n)$ tanpa mempertimbangkan cost kumulatif $g(n)$, sehingga dapat memilih jalur yang tampak menjanjikan di awal tetapi sebenarnya lebih panjang secara keseluruhan. GBFS mungkin memilih untuk memindahkan kendaraan utama lebih dekat ke pintu keluar (menurunkan nilai heuristik), tetapi membuat konfigurasi board di mana banyak kendaraan lain menjadi sulit untuk dipindahkan kemudian.

Lalu juga, GBFS dapat terjebak dalam optimum lokal, situasi di mana semua gerakan langsung meningkatkan nilai heuristik tetapi jalur keseluruhan tidak optimal. GBFS biasanya lebih cepat dan menggunakan memori lebih sedikit daripada A* dan UCS, tetapi dengan trade-off kehilangan jaminan optimalitas. GBFS memiliki kompleksitas ruang dan waktu yang lebih baik dalam kasus

rata-rata karena eksplorasi yang lebih terarah, tetapi masih eksponensial dalam kasus terburuk.

Bab 4

Source Code

4.1 Piece.java

```
public class Piece {
    private char pieceChar;
    private int size;

    public enum Orientation {
        HORIZONTAL,
        VERTICAL
    }

    private Orientation orientation;
    private int x; // x and y dari atas kiri
    private int y;
    private boolean isPrimary;

    public Piece(char pieceChar, int size, Orientation orientation, int x, int y) {
        this.pieceChar = pieceChar;
        this.size = size;
        this.orientation = orientation;
        this.x = x;
        this.y = y;
        this.isPrimary = false;
    }

    public char getPieceChar() {
        return pieceChar;
    }

    public int getSize() {
        return size;
    }

    public Orientation getOrientation() {
        return orientation;
    }
}
```

```

public int getX() {
    return x;
}

public int getY() {
    return y;
}

public boolean isPrimary() {
    return isPrimary;
}

public void setPrimary(boolean primary) {
    isPrimary = primary;
}

public void setPieceChar(char pieceChar) {
    this.pieceChar = pieceChar;
}

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

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

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

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

public void moveX(int deltaX) {
    if (orientation == Orientation.HORIZONTAL) {
        this.x += deltaX;
    } else {
        throw new UnsupportedOperationException("Cannot move X for vertical piece");
    }
}

```

```

public void moveY(int deltaY) {
    if (orientation == Orientation.VERTICAL) {
        this.y += deltaY;
    } else {
        throw new UnsupportedOperationException("Cannot move Y for horizontal piece");
    }
}

public void printPiece() {
    if (orientation == Orientation.HORIZONTAL) {
        for (int i = 0; i < size; i++) {
            System.out.print(pieceChar);
        }
        System.out.println();
    } else {
        for (int i = 0; i < size; i++) {
            System.out.println(pieceChar);
        }
    }
}

public Piece copy() {
    Piece newPiece = new Piece(this.pieceChar, this.size, this.orientation, this.x, this.y);
    newPiece.setPrimary(this.isPrimary);
    return newPiece;
}
}

```

4.2 Board.java

```

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

public class Board {
    private char[][] board;
    private int rows;
    private int cols;
    private int nonPieceCount;
    private char mainChar;
    private int exitX;
    private int exitY;
}

```

```

private List<Piece> nonPieceList = new ArrayList<>();
private Piece primaryPiece;

public Board(){
    this.rows = 0;
    this.cols = 0;
    this.mainChar = 'P';
    this.board = new char[rows][cols];
    this.nonPieceCount = 0;
}

public Board(int rows, int cols) {
    this.rows = rows;
    this.cols = cols;
    this.mainChar = 'P';
    this.board = new char[rows][cols];
    this.nonPieceCount = 0;
    for (int i = 0; i < rows; i++) {
        Arrays.fill(board[i], ' ');
    }
}

public Board(Board board) {
    this.rows = board.rows;
    this.cols = board.cols;
    this.mainChar = board.mainChar;
    this.nonPieceCount = board.nonPieceCount;
    this.board = new char[rows][cols];
    for (int i = 0; i < rows; i++) {
        System.arraycopy(board.board[i], 0, this.board[i], 0, cols);
    }
    this.exitX = board.exitX;
    this.exitY = board.exitY;
    this.nonPieceList = new ArrayList<>(board.nonPieceList);
    this.primaryPiece = board.primaryPiece;
}

public void setChar(int row, int col, char c) {
    if (isValid(row, col)) {
        board[row][col] = c;
    }
}

public char getChar(int row, int col) {

```

```

        return isValid(row, col) ? board[row][col] : ' ';
    }

    public void setMainChar(char c) {
        this.mainChar = c;
    }

    public char getMainChar() {
        return mainChar;
    }

    public void setNonPieceCount(int nonPieceCount) {
        this.nonPieceCount = nonPieceCount;
    }

    public int getNonPieceCount() {
        return nonPieceCount;
    }

    public Piece getPrimaryPiece() {
        return primaryPiece;
    }

    public int getRows() {
        return rows;
    }

    public int getCols() {
        return cols;
    }

    public void setBoard(char[][] board) {
        this.board = board;
    }

    public char[][] getBoard() {
        return board;
    }

    public void printBoard() {
        if (exitY == -1 && exitX >= 0 && exitX < cols) {
            for (int j = 0; j < cols; j++) {
                if (j == exitX) {
                    System.out.print("\u001B[32mK\u001B[0m");
                }
            }
        }
    }

```

```

        } else {
            System.out.print(" ");
        }
    }
    System.out.println();
}

for (int i = 0; i < rows; i++) {
    if (exitX == -1 && exitY == i) {
        System.out.print("\u001B[32mK\u001B[0m");
    } else if (exitX == -1) {
        System.out.print(" ");
    }

    for (int j = 0; j < cols; j++) {
        char cell = board[i][j];
        if (cell == 'P') {
            System.out.print("\u001B[31m" + cell + "\u001B[0m");
        } else if (cell == '.') {
            System.out.print(cell);
        } else {
            System.out.print("\u001B[34m" + cell + "\u001B[0m");
        }
    }

    if (exitX == cols && exitY == i) {
        System.out.print("\u001B[32mK\u001B[0m");
    }
    System.out.println();
}

if (exitY == rows && exitX >= 0 && exitX < cols) {
    for (int j = 0; j < cols; j++) {
        if (j == exitX) {
            System.out.print("\u001B[32mK\u001B[0m");
        } else {
            System.out.print(" ");
        }
    }
    System.out.println();
}
}

public void clearBoard() {

```

```

        for (int i = 0; i < rows; i++) {
            Arrays.fill(board[i], ' ');
        }
    }

    public void readFromFile(String fileName) {
        try (BufferedReader br = new BufferedReader(new FileReader(fileName))) {
            String line;
            line = br.readLine(); // dimensi
            if (line != null) {
                String[] dimensions = line.split(" ");
                try {
                    rows = Integer.parseInt(dimensions[0]);
                    cols = Integer.parseInt(dimensions[1]);
                } catch (NumberFormatException e) {
                    throw new IllegalArgumentException("Invalid dimension format. Make sure its
in valid format :)");
                }

                if (rows <= 0 || cols <= 0) {
                    throw new IllegalArgumentException("Invalid dimension values. Rows and
columns must be positive integers :)");
                }

                List<String> inputLines = new ArrayList<>();
                int maxLength = 0;
                int kRow = -1, kCol = -1;

                // nonPieceCount
                line = br.readLine();
                if (line != null) {
                    nonPieceCount = Integer.parseInt(line);
                }

                int rowCount = 0;
                while ((line = br.readLine()) != null && rowCount < rows + 1) { // +1 untuk check
K
                    inputLines.add(line);
                    maxLength = Math.max(maxLength, line.length());

                    for (int i = 0; i < line.length(); i++) {
                        if (line.charAt(i) == 'K') {
                            kRow = rowCount;
                            kCol = i;
                        }
                    }
                }
            }
        }
    }

```

```

        }
    }
    rowCount++;
}

board = new char[rows][cols];
for (int i = 0; i < rows; i++) {
    Arrays.fill(board[i], '.');
}

for (int i = 0; i < Math.min(inputLines.size(), rows); i++) {
    String currentLine = inputLines.get(i);
    for (int j = 0; j < Math.min(currentLine.length(), cols); j++) {
        board[i][j] = currentLine.charAt(j);
    }
}

if (kRow != -1 && kCol != -1) {
    exitY = kRow;
    exitX = kCol;
}
}

boolean[][] isChecked = new boolean[rows][cols];
nonPieceList.clear();

for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        if (!isChecked[i][j]) {
            isChecked[i][j] = true;
            char c = board[i][j];

            if (c == '.' || c == 'K') continue;

            if (j + 1 < cols && board[i][j + 1] == c) {
                int length = 1;
                while (j + length < cols && board[i][j + length] == c) {
                    isChecked[i][j + length] = true;
                    length++;
                }
                Piece piece = new Piece(c, length, Piece.Orientation.HORIZONTAL, j,
i);

                if (c == mainChar) {
                    piece.setPrimary(true);

```



```

        primaryPiece = piece;
    } else {
        nonPieceList.add(piece);
    }
}
else if (i + 1 < rows && board[i + 1][j] == c) {
    int length = 1;
    while (i + length < rows && board[i + length][j] == c) {
        isChecked[i + length][j] = true;
        length++;
    }
    Piece piece = new Piece(c, length, Piece.Orientation.VERTICAL, j, i);
    if (c == mainChar) {
        piece.setPrimary(true);
        primaryPiece = piece;
    } else {
        nonPieceList.add(piece);
    }
}
else if (c != '.') {
    Piece piece = new Piece(c, 1, Piece.Orientation.HORIZONTAL, j, i);
    if (c == mainChar) {
        piece.setPrimary(true);
        primaryPiece = piece;
    } else {
        nonPieceList.add(piece);
    }
}
}
}
}

} catch (IOException e) {
    System.err.println("Error reading the file, make sure the file is in valid format
:");
}

}

public boolean isFull() {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (board[i][j] == ' ') return false;
        }
    }
}
}

```

```

        return true;
    }

    public Board copy() {
        Board newBoard = new Board();
        newBoard.rows = this.rows;
        newBoard.cols = this.cols;
        newBoard.mainChar = this.mainChar;
        newBoard.nonPieceCount = this.nonPieceCount;
        newBoard.exitX = this.exitX;
        newBoard.exitY = this.exitY;

        newBoard.board = new char[rows][cols];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                newBoard.board[i][j] = this.board[i][j];
            }
        }

        return newBoard;
    }

    public boolean isEmpty(int row, int col) {
        return isValid(row, col) && board[row][col] == '.';
    }

    public void saveToFile(String fileName) {
        try (BufferedWriter bw = new BufferedWriter(new FileWriter(fileName))) {
            for (int i = 0; i < rows; i++) {
                for (int j = 0; j < cols; j++) {
                    bw.write(board[i][j]);
                }
                bw.newLine();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

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

    public int getExitX() {

```

```

        return exitX;
    }

    public int getExitY() {
        return exitY;
    }

    public void setExitX(int exitX) {
        this.exitX = exitX;
    }

    public void setExitY(int exitY) {
        this.exitY = exitY;
    }

    public void addPiece(Piece piece) {
        nonPieceList.add(piece);
    }

    public List<Piece> getNonPieceList() {
        return nonPieceList;
    }

    public void setNonPieceList(List<Piece> nonPieceList) {
        this.nonPieceList = nonPieceList;
    }

    public void clearNonPieceList() {
        nonPieceList.clear();
    }

    public void removePiece(int index) {
        if (index >= 0 && index < nonPieceList.size()) {
            nonPieceList.remove(index);
        }
    }

    public void removePiece(char pieceChar) {
        nonPieceList.removeIf(piece -> piece.getPieceChar() == pieceChar);
    }

    public void printAllPieces() {
        for (Piece piece : nonPieceList) {
            piece.printPiece();
        }
    }

```

```

    }
}

public Map<Character, Piece> getPieceMap() {
    Map<Character, Piece> pieces = new HashMap<>();
    for (Piece piece : nonPieceList) {
        pieces.put(piece.getPieceChar(), piece);
    }
    pieces.put('P', primaryPiece);
    return pieces;
}

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

```

4.3 State.java

```

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

public class State implements Comparable<State> {
    private Map<Character, Piece> pieces; //map dari vehicle ID dan Piece objek
    private Board board;
    private int cost;
    private State parent; // track solusi
    private String move;
    private State prevState;
    private int heuristic;

    public State(Map<Character, Piece> pieces, Board board, int cost, State parent, String move)
    {
        this.pieces = pieces;
        this.board = board;
        this.cost = cost;
    }
}

```

```

        this.parent = parent;
        this.move = move;
        this.heuristic = computeHeuristic();
    }

    public int getCost() {
        return cost;
    }

    public Board getBoard() {
        return board;
    }

    public int getHeuristic() {
        return heuristic;
    }

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

    public String getMove() {
        return move;
    }

    public State getParent() {
        return parent;
    }

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

    private int computeHeuristic() {
        Piece xCar = pieces.get('P');
        if (xCar == null) return 0;

        int count = 0;
        if (xCar.getOrientation() == Piece.Orientation.HORIZONTAL) {
            int row = xCar.getY();
            if (board.getExitX() == -1) {
                int leftCol = xCar.getX();
                for (int c = leftCol - 1; c >= 0; c--) {
                    if (board.getBoard()[row][c] != '.') count++;
                }
            }
        }
    }

```

```

    }
    } else {
        int rightEnd = xCar.getX() + xCar.getSize() - 1;
        for (int c = rightEnd + 1; c < board.getBoard()[0].length; c++) {
            if (board.getBoard()[row][c] != '.') count++;
        }
    }
} else {
    int col = xCar.getX();
    if (board.getExitY() == -1) {
        int topRow = xCar.getY();
        for (int r = topRow - 1; r >= 0; r--) {
            if (board.getBoard()[r][col] != '.') count++;
        }
    } else {
        int bottomEnd = xCar.getY() + xCar.getSize() - 1;
        for (int r = bottomEnd + 1; r < board.getBoard().length; r++) {
            if (board.getBoard()[r][col] != '.') count++;
        }
    }
}
}
return count;
}

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

public boolean isGoal() {
    Piece redCar = pieces.get('P');
    if (redCar == null) return false;

    if (redCar.getOrientation() == Piece.Orientation.HORIZONTAL) {
        int leftCol = redCar.getX();
        int rightEndCol = redCar.getX() + redCar.getSize() - 1;
        int row = redCar.getY();

        if (board.getExitX() == -1) {
            return board.getExitY() == row && leftCol == board.getExitX();
        } else {
            return board.getExitY() == row && rightEndCol == board.getExitX();
        }
    } else {

```

```

        int topRow = redCar.getY();
        int bottomEndRow = redCar.getY() + redCar.getSize() - 1;
        int col = redCar.getX();

        if (board.getExitY() == -1) {
            return board.getExitX() == col && topRow == board.getExitY();
        } else {
            return board.getExitX() == col && bottomEndRow == board.getExitY();
        }
    }
}

public List<State> getNextStates() {
    List<State> nextStates = new ArrayList<>();
    for (Piece p : pieces.values()) {
        for (int direction : new int[]{1, -1}) {
            int steps = 1;
            while (true) {
                if (!canMove(p, direction, steps)) break;
                State newState = moveVehicle(p, direction * steps);
                if (newState != null) nextStates.add(newState);
                steps++;
            }
        }
    }
    return nextStates;
}

private boolean canMove(Piece p, int direction, int steps) {
    if (p.getOrientation() == Piece.Orientation.HORIZONTAL) {
        int row = p.getY();
        if (direction == -1) {
            for (int i = 1; i <= steps; i++) {
                int col = p.getX() - i;
                if (col < 0) {
                    if (col == -1) {
                        if (i == steps) {
                            return board.getExitY() == row && board.getExitX() == -1;
                        } else return false;
                    }
                }
                return false;
            }
        }
        if (board.getBoard()[row][col] != '.') return false;
    }
}

```

```

        } else {
            int rightEnd = p.getX() + p.getSize() - 1;
            for (int i = 1; i <= steps; i++) {
                int col = rightEnd + i;
                if (col >= board.getBoard()[0].length) {
                    if (col == board.getBoard()[0].length && i == steps) {
                        return board.getExitY() == row && board.getExitX() == col;
                    } else return false;
                }
                if (col < board.getBoard()[0].length && board.getBoard()[row][col] != '.')
return false;
            }
        }
    } else {
        int col = p.getX();
        if (direction == -1) {
            for (int i = 1; i <= steps; i++) {
                int row = p.getY() - i;
                if (row < 0) {
                    if (row == -1 && i == steps) {
                        return board.getExitX() == col && board.getExitY() == -1;
                    } else return false;
                }
                if (board.getBoard()[row][col] != '.') return false;
            }
        } else {
            int bottomEnd = p.getY() + p.getSize() - 1;
            for (int i = 1; i <= steps; i++) {
                int row = bottomEnd + i;
                if (row >= board.getBoard().length) {
                    if (row == board.getBoard().length && i == steps) {
                        return board.getExitX() == col && board.getExitY() == row;
                    } else return false;
                }
                if (row < board.getBoard().length && board.getBoard()[row][col] != '.')
return false;
            }
        }
    }
    return true;
}

private State moveVehicle(Piece p, int move) {
    Map<Character, Piece> newPieces = new HashMap<>();

```



```

for (Map.Entry<Character, Piece> entry : pieces.entrySet()) {
    newPieces.put(entry.getKey(), entry.getValue().copy());
}

Piece movedPiece = newPieces.get(p.getPieceChar());

if (movedPiece.getOrientation() == Piece.Orientation.HORIZONTAL) {
    movedPiece.setX(movedPiece.getX() + move);
} else {
    movedPiece.setY(movedPiece.getY() + move);
}

char[][] newBoard = new char[board.getBoard().length][board.getBoard()[0].length];
for (int i = 0; i < newBoard.length; i++) {
    Arrays.fill(newBoard[i], '.');
}

for (Piece piece : newPieces.values()) {
    int r = piece.getY();
    int c = piece.getX();
    for (int i = 0; i < piece.getSize(); i++) {
        if (piece.getOrientation() == Piece.Orientation.HORIZONTAL) {
            int cc = c + i;
            if (r >= 0 && r < newBoard.length && cc >= 0 && cc < newBoard[0].length) {
                newBoard[r][cc] = piece.getPieceChar();
            }
        } else {
            int rr = r + i;
            if (rr >= 0 && rr < newBoard.length && c >= 0 && c < newBoard[0].length) {
                newBoard[rr][c] = piece.getPieceChar();
            }
        }
    }
}

int newCost = this.cost + 1;
Board newBoardCopy = board.copy();
newBoardCopy.setBoard(newBoard);

String direction = (movedPiece.getOrientation() == Piece.Orientation.HORIZONTAL)
    ? (move > 0 ? "right" : "left")
    : (move > 0 ? "down" : "up");

```

```

        return new State(newPieces, newBoardCopy, newCost, this, "Move " + p.getPieceChar() + "
to " + direction);
    }

    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof State)) return false;
        State other = (State) o;

        if (pieces.size() != other.pieces.size()) return false;

        for (char id : pieces.keySet()) {
            Piece v1 = pieces.get(id);
            Piece v2 = other.pieces.get(id);
            if (v2 == null) return false;
            if (v1.getX() != v2.getX() || v1.getY() != v2.getY()) return false;
        }
        return true;
    }

    @Override
    public int hashCode() {
        int result = 17;
        for (Piece v : pieces.values()) {
            result = 31 * result + v.getPieceChar();
            result = 31 * result + v.getY();
            result = 31 * result + v.getX();
        }
        return result;
    }

    @Override
    public String toString() {
        final String RESET = "\033[0m";
        final String RED = "\033[31m";
        final String GREEN = "\033[32m";
        final String BLUE = "\033[34m";

        StringBuilder sb = new StringBuilder();
        sb.append("\n===== \n");
        sb.append("  Rush Hour Solver [Jakarta Edition] \n");
        sb.append("===== \n");

        int exitX = this.getBoard().getExitX();

```

```

int exitY = this.getBoard().getExitY();

char[][] board = this.getBoard().getBoard();
int rows = board.length;
int cols = board[0].length;

// make sure warna sesuai dengan posisi exit dan primary piece
for (int y = 0; y < rows; y++) {
    for (int x = 0; x < cols; x++) {
        if (x == exitX && y == exitY) {
            sb.append(GREEN + " K " + RESET);
        } else {
            char piece = board[y][x];
            if (piece == 'P') {
                sb.append(RED + " P " + RESET);
            } else if (piece == '.') {
                sb.append(BLUE + " . " + RESET);
            } else {
                sb.append(" " + piece + " ");
            }
        }
    }
    sb.append("\n");
}

sb.append("\n=====");
return sb.toString();
}

public void saveSolutionToFile(int nodeCount, Long executionTime) {
    BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
    System.out.println("Enter the filename to save the solution (ex: test\\test.txt:");
    String filename;
    try {
        filename = reader.readLine();
    } catch (IOException e) {
        System.err.println("Error reading input: " + e.getMessage());
        return;
    }

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

```

```

        current = current.getParent();
    }
    Collections.reverse(path);

    //save tanpa warna
    try (PrintWriter writer = new PrintWriter(filename)) {
        for (State state : path) {
            writer.println("Move: " + state.getMove());
            writer.println(state.toStringWithoutColor());
        }
        writer.println("Visited nodes: " + nodeCount);
        writer.println("Execution time: " + executionTime + " ms");
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public Piece getPieces(char id) {
    return pieces.get(id);
}

public String toStringWithoutColor() {
    int exitX = this.getBoard().getExitX();
    int exitY = this.getBoard().getExitY();

    char[][] board = this.getBoard().getBoard();
    int rows = board.length;
    int cols = board[0].length;

    int minRow = Math.min(0, exitY);
    int maxRow = Math.max(rows - 1, exitY);
    int minCol = Math.min(0, exitX);
    int maxCol = Math.max(cols - 1, exitX);

    StringBuilder sb = new StringBuilder();

    for (int y = minRow; y <= maxRow; y++) {
        for (int x = minCol; x <= maxCol; x++) {
            if (x == exitX && y == exitY) {
                sb.append("K");
            } else {
                if (y >= 0 && y < rows && x >= 0 && x < cols) {
                    sb.append(board[y][x]);
                } else {

```

```

        sb.append(" ");
    }
}
}
sb.append("\n");
}

return sb.toString();
}

public void saveNoSolutionToFile(int nodeCount, Long executionTime) {
    BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
    System.out.println("Enter the filename to save the solution (ex: test\\test.txt):");
    String filename;
    try {
        filename = reader.readLine();
    } catch (IOException e) {
        System.err.println("Error reading input: " + e.getMessage());
        return;
    }
    try (PrintWriter writer = new PrintWriter(filename)) {
        writer.println("No solution found.");
        writer.println("Visited nodes: " + nodeCount);
        writer.println("Execution time: " + executionTime + " ms");
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public void printSolution() {
    if (parent != null) {
        parent.printSolution();
    }
    System.out.println(move);
    board.printBoard();
    System.out.println();
}

public void setPrevState(State prev) {
    this.prevState = prev;
}

public State getPrevState() {
    return prevState;
}

```

```

    }

    public List<State> getPath() {
        List<State> path = new ArrayList<>();
        State current = this;
        while (current != null) {
            path.add(0, current);
            current = current.getPrevState();
        }
        return path;
    }
}

```

4.4 AStar.java

```

import java.util.*;

public class AStar {
    private static final Comparator<State> aStarComparator = (s1, s2) -> {
        int f1 = s1.getCost() + s1.getHeuristic();
        int f2 = s2.getCost() + s2.getHeuristic();
        return Integer.compare(f1, f2);
    };

    public static void solve(State startState) {
        Long startTime = System.currentTimeMillis();
        int visitCount = 0;
        PriorityQueue<State> openSet = new PriorityQueue<>(aStarComparator);
        Map<State, Integer> bestFScore = new HashMap<>();

        openSet.add(startState);
        bestFScore.put(startState, startState.getCost() + startState.getHeuristic());

        while (!openSet.isEmpty()) {
            State current = openSet.poll();
            visitCount++;

            if (current.isGoal()) {
                Long endTime = System.currentTimeMillis();
                System.out.println("Execution time: " + (endTime - startTime) + " ms");
                System.out.println("Visited nodes: " + visitCount);
                current.printSolution();
            }
        }
    }
}

```

```

        current.saveSolutionToFile(visitCount, endTime - startTime);
        return;
    }

    for (State next : current.getNextStates()) {
        int g = next.getCost();
        int f = g + next.getHeuristic();

        if (!bestFScore.containsKey(next) || f < bestFScore.get(next)) {
            bestFScore.put(next, f);
            openSet.add(next);
        }
    }
}

System.out.println("No solution found.");
Long endTime = System.currentTimeMillis();
startState.saveNoSolutionToFile(visitCount, endTime - startTime);
}

public static SearchResult GUIsolve(State initialState, int heuristicType) {
    Long startTime = System.currentTimeMillis();
    int visitCount = 0;
    // Comparator tetap sama, pakai cost + heuristic
    PriorityQueue<State> openSet = new PriorityQueue<>(aStarComparator);
    Map<State, Integer> bestFScore = new HashMap<>();

    // Set heuristic sesuai pilihan user
    initialState.setHeuristic(Heuristic.calculate(initialState, heuristicType));
    openSet.add(initialState);
    bestFScore.put(initialState, initialState.getCost() + initialState.getHeuristic());

    while (!openSet.isEmpty()) {
        State current = openSet.poll();
        visitCount++;

        if (current.isGoal()) {
            Long endTime = System.currentTimeMillis();
            return new SearchResult(current.getPath(), endTime - startTime, visitCount);
        }

        for (State next : current.getNextStates()) {
            // Set heuristic untuk next state sesuai pilihan user
            next.setHeuristic(Heuristic.calculate(next, heuristicType));

```

```

        int g = next.getCost();
        int f = g + next.getHeuristic();

        // Set parent/prevState agar path bisa ditelusuri di GUI
        next.setPrevState(current);

        if (!bestFScore.containsKey(next) || f < bestFScore.get(next)) {
            bestFScore.put(next, f);
            openSet.add(next);
        }
    }
}

long endTime = System.currentTimeMillis();
List<State> noSolution = new ArrayList<>();
noSolution.add(initialState);
return new SearchResult(noSolution, endTime - startTime, visitCount);
}
}

```

4.5 BeamSearch.java

```

import java.util.*;

public class BeamSearch {
    private static int beamWidth = 100;

    public static void solve(State initialState) {
        long startTime = System.currentTimeMillis();
        Set<String> visited = new HashSet<>();
        int nodeCount = 0;
        List<State> currentBeam = new ArrayList<>();
        currentBeam.add(initialState);

        int iterations = 0;
        int maxIterations = 10000;

        while (!currentBeam.isEmpty() && iterations < maxIterations) {
            iterations++;
            List<State> nextBeam = new ArrayList<>();
            for (State current : currentBeam) {
                if (current.isGoal()) {
                    long endTime = System.currentTimeMillis();

```



```

        System.out.println("Execution time: " + (endTime - startTime) + " ms");
        System.err.println("Visited nodes: " + nodeCount);
        current.printSolution();
        current.saveSolutionToFile(nodeCount, endTime - startTime);
        return;
    }
    String boardKey = getBoardKey(current.getBoard().getBoard());
    visited.add(boardKey);
    List<State> successors = current.getNextStates();
    for (State successor : successors) {
        String successorKey = getBoardKey(successor.getBoard().getBoard());
        if (!visited.contains(successorKey)) {
            nextBeam.add(successor);
            nodeCount++;
        }
    }
}
if (nextBeam.isEmpty()) {
    break;
}

Collections.sort(nextBeam, (s1, s2) -> {
    int f1 = s1.getCost() + s1.getHeuristic();
    int f2 = s2.getCost() + s2.getHeuristic();
    return Integer.compare(f1, f2);
});

currentBeam = nextBeam.size() <= beamWidth ?
    nextBeam :
    nextBeam.subList(0, beamWidth);
}

System.out.println("No solution found after " + iterations + " iterations.");
Long endTime = System.currentTimeMillis();
initialState.saveNoSolutionToFile(nodeCount, endTime - startTime);
}

private static String getBoardKey(char[][] board) {
    StringBuilder key = new StringBuilder();
    for (char[] row : board) {
        key.append(String.valueOf(row));
    }
    return key.toString();
}
}

```

```

public static SearchResult GUIsolve(State initialState, int beamWidth, int heuristicType) {
    long startTime = System.currentTimeMillis();
    Set<String> visited = new HashSet<>();
    int nodeCount = 0;
    List<State> currentBeam = new ArrayList<>();
    // Set heuristic untuk initialState sesuai pilihan user
    initialState.setHeuristic(Heuristic.calculate(initialState, heuristicType));
    currentBeam.add(initialState);

    int iterations = 0;
    int maxIterations = 10000;

    while (!currentBeam.isEmpty() && iterations < maxIterations) {
        iterations++;
        List<State> nextBeam = new ArrayList<>();
        for (State current : currentBeam) {
            if (current.isGoal()) {
                long endTime = System.currentTimeMillis();
                return new SearchResult(current.getPath(), endTime - startTime, nodeCount);
            }
            String boardKey = getBoardKey(current.getBoard().getBoard());
            visited.add(boardKey);
            List<State> successors = current.getNextStates();
            for (State successor : successors) {
                String successorKey = getBoardKey(successor.getBoard().getBoard());
                if (!visited.contains(successorKey)) {
                    successor.setPrevState(current); // Penting: set parent/prevState!
                    // Set heuristic untuk successor sesuai pilihan user
                    successor.setHeuristic(Heuristic.calculate(successor, heuristicType));
                    nextBeam.add(successor);
                    nodeCount++;
                }
            }
        }
        if (nextBeam.isEmpty()) {
            break;
        }

        Collections.sort(nextBeam, (s1, s2) -> {
            int f1 = s1.getCost() + s1.getHeuristic();
            int f2 = s2.getCost() + s2.getHeuristic();
            return Integer.compare(f1, f2);
        });
    }
}

```

```

        currentBeam = nextBeam.size() <= beamWidth ?
            nextBeam :
            nextBeam.subList(0, beamWidth);
    }

    Long endTime = System.currentTimeMillis();
    List<State> noSolution = new ArrayList<>();
    noSolution.add(initialState);
    return new SearchResult(noSolution, endTime - startTime, nodeCount);
}
}

```

4.6 GBFS.java

```

import java.util.*;

public class GBFS {
    // comparator GBFS:  $f(n) = h(n)$ 
    private static final Comparator<State> gbfsComparator =
    Comparator.comparingInt(State::getHeuristic);

    public static void solve(State initialState) {
        Long startTime = System.currentTimeMillis();
        int visitCount = 0;
        Set<State> visited = new HashSet<>();
        PriorityQueue<State> queue = new PriorityQueue<>(gbfsComparator);

        queue.add(initialState);

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

            if (visited.contains(current)) continue;
            visited.add(current);
            visitCount++;

            if (current.isGoal()) {
                Long endTime = System.currentTimeMillis();
                System.out.println("Execution time: " + (endTime - startTime) + " ms");
                System.out.println("Visited nodes: " + visitCount);
                current.printSolution();
            }
        }
    }
}

```

```

        current.saveSolutionToFile(visitCount, endTime - startTime);
        return;
    }

    for (State next : current.getNextStates()) {
        if (!visited.contains(next)) {
            next.setPrevState(current); // Set parent/prevState
            queue.add(next);
        }
    }
}

System.out.println("No solution found.");
Long endTime = System.currentTimeMillis();
initialState.saveNoSolutionToFile(visitCount, endTime - startTime);
}

public static SearchResult GUIsolve(State initialState, int heuristicType) {
    Long startTime = System.currentTimeMillis();
    int visitCount = 0;
    Set<State> visited = new HashSet<>();
    Comparator<State> guiComparator = Comparator.comparingInt(s -> Heuristic.calculate(s,
heuristicType));
    PriorityQueue<State> queue = new PriorityQueue<>(guiComparator);

    // Set heuristic untuk initialState
    initialState.setHeuristic(Heuristic.calculate(initialState, heuristicType));
    queue.add(initialState);

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

        if (visited.contains(current)) continue;
        visited.add(current);
        visitCount++;

        if (current.isGoal()) {
            Long endTime = System.currentTimeMillis();
            return new SearchResult(current.getPath(), endTime - startTime, visitCount);
        }

        for (State next : current.getNextStates()) {
            if (!visited.contains(next)) {
                next.setPrevState(current);
            }
        }
    }
}

```

```

        next.setHeuristic(Heuristic.calculate(next, heuristicType)); // Set heuristic
        // Set heuristic for next state
        queue.add(next);
    }
}
}
Long endTime = System.currentTimeMillis();
List<State> noSolution = new ArrayList<>();
noSolution.add(initialState);
return new SearchResult(noSolution, endTime - startTime, visitCount);
}
}

```

4.7 UCS.java

```

import java.util.*;

public class UCS {

    public static void uniformCostSearch(State initialState) {
        Long startTime = System.currentTimeMillis();
        int visitCount = 0;
        PriorityQueue<State> frontier = new PriorityQueue<>();
        Set<State> explored = new HashSet<>();

        frontier.add(initialState);

        while (!frontier.isEmpty()) {
            State current = frontier.poll();
            visitCount++;

            if (current.isGoal()) {
                Long endTime = System.currentTimeMillis();
                System.out.println("Execution time: " + (endTime - startTime) + " ms");
                System.out.println("Visited nodes: " + visitCount);
                current.printSolution();
                current.saveSolutionToFile(visitCount, endTime - startTime);
                return;
            }

            explored.add(current);
        }
    }
}

```

```

        for (State next : current.getNextStates()) {
            if (!explored.contains(next) && !frontier.contains(next)) {
                frontier.add(next);
            } else if (frontier.contains(next)) {
                for (State stateInFrontier : frontier) {
                    if (stateInFrontier.equals(next) && stateInFrontier.getCost() >
next.getCost()) {
                        frontier.remove(stateInFrontier);
                        frontier.add(next);
                        break;
                    }
                }
            }
        }
    }
}

System.out.println("No solution found.");
Long endTime = System.currentTimeMillis();
initialState.saveNoSolutionToFile(visitCount, endTime - startTime);
}

public static SearchResult GUIUniformCostSearch(State initialState) {
    Long startTime = System.currentTimeMillis();
    int visitCount = 0;
    PriorityQueue<State> frontier = new PriorityQueue<>();
    Set<State> explored = new HashSet<>();

    frontier.add(initialState);

    while (!frontier.isEmpty()) {
        State current = frontier.poll();
        visitCount++;

        if (current.isGoal()) {
            Long endTime = System.currentTimeMillis();
            return new SearchResult(current.getPath(), endTime - startTime, visitCount);
        }

        explored.add(current);

        for (State next : current.getNextStates()) {
            next.setPrevState(current); // Penting: set parent/prevState!
            if (!explored.contains(next) && !frontier.contains(next)) {
                frontier.add(next);
            } else if (frontier.contains(next)) {

```

```

        for (State stateInFrontier : frontier) {
            if (stateInFrontier.equals(next) && stateInFrontier.getCost() >
next.getCost()) {
                frontier.remove(stateInFrontier);
                frontier.add(next);
                break;
            }
        }
    }
}

long endTime = System.currentTimeMillis();
java.util.List<State> noSolution = new java.util.ArrayList<>();
noSolution.add(initialState);
return new SearchResult(noSolution, endTime - startTime, visitCount);
}
}

```

4.8 Heuristic.java

```

public class Heuristic {
    public static final int MANHATTAN_DISTANCE = 0;
    public static final int BLOCKING_VEHICLES = 1;
    public static final int ADVANCED_BLOCKING = 2;
    public static final int COMBINED = 3;

    public static int calculate(State state, int heuristicType) {
        switch (heuristicType) {
            case MANHATTAN_DISTANCE:
                return calculateManhattanDistance(state);
            case BLOCKING_VEHICLES:
                return calculateBlockingVehicles(state);
            case ADVANCED_BLOCKING:
                return calculateAdvancedBlocking(state);
            case COMBINED:
                return calculateCombined(state);
            default:
                return calculateManhattanDistance(state);
        }
    }

    private static int calculateManhattanDistance(State state) {

```

```

    Piece primaryPiece = state.getPieces('P');
    Board board = state.getBoard();
    int exitX = board.getExitX();
    int exitY = board.getExitY();

    if (primaryPiece.getOrientation() == Piece.Orientation.HORIZONTAL) {
        int rightEnd = primaryPiece.getX() + primaryPiece.getSize() - 1;
        int distance = 0;

        if (exitX > rightEnd) {
            distance = exitX - rightEnd;
        } else if (exitX == -1) { // Exit di kiri
            distance = primaryPiece.getX();
        }

        return distance;
    } else if (primaryPiece.getOrientation() == Piece.Orientation.VERTICAL) {
        int bottomEnd = primaryPiece.getY() + primaryPiece.getSize() - 1;
        int distance = 0;

        if (exitY > bottomEnd) {
            distance = exitY - bottomEnd;
        } else if (exitY == -1) { // Exit di atas
            distance = primaryPiece.getY();
        }

        return distance;
    }
    return 0;
}

private static int calculateBlockingVehicles(State state) {
    Piece primaryPiece = state.getPieces('P');
    Board board = state.getBoard();
    int count = 0;
    char[][] boardArray = board.getBoard();

    if (primaryPiece.getOrientation() == Piece.Orientation.HORIZONTAL) {
        int row = primaryPiece.getY();
        int rightEnd = primaryPiece.getX() + primaryPiece.getSize() - 1;
        int exitX = board.getExitX();

        // Check if exit is on the right
        if (exitX > rightEnd) {

```



```

        // Count blocking vehicles between car and right exit
        for (int col = rightEnd + 1; col < exitX; col++) {
            if (boardArray[row][col] != '.') {
                count++;
            }
        }
    }
    // Check if exit is on the left
    else if (exitX == -1) {
        // Count blocking vehicles between car and left exit
        for (int col = primaryPiece.getX() - 1; col >= 0; col--) {
            if (boardArray[row][col] != '.') {
                count++;
            }
        }
    }
} else if (primaryPiece.getOrientation() == Piece.Orientation.VERTICAL) {
    int col = primaryPiece.getX();
    int bottomEnd = primaryPiece.getY() + primaryPiece.getSize() - 1;
    int exitY = board.getExitY();

    // Check if exit is below
    if (exitY > bottomEnd) {
        // Count blocking vehicles between car and bottom exit
        for (int row = bottomEnd + 1; row < exitY; row++) {
            if (boardArray[row][col] != '.') {
                count++;
            }
        }
    }
    // Check if exit is above
    else if (exitY == -1) {
        // Count blocking vehicles between car and top exit
        for (int row = primaryPiece.getY() - 1; row >= 0; row--) {
            if (boardArray[row][col] != '.') {
                count++;
            }
        }
    }
}

return count;
}

```

```

private static int calculateAdvancedBlocking(State state) {
    // Implementasi Lebih kompleks yang mempertimbangkan kesulitan memindahkan kendaraan
    int basicBlocking = calculateBlockingVehicles(state);
    int movabilityPenalty = 0;
    Board board = state.getBoard();
    char[][] boardArray = board.getBoard();
    Piece primaryPiece = state.getPieces('P');

    // Periksa "movability" kendaraan yang menghalangi
    if (primaryPiece.getOrientation() == Piece.Orientation.HORIZONTAL) {
        int row = primaryPiece.getY();
        int rightEnd = primaryPiece.getX() + primaryPiece.getSize() - 1;
        int exitX = board.getExitX();

        // Jika exit di kanan
        if (exitX > rightEnd) {
            for (int col = rightEnd + 1; col < exitX; col++) {
                if (boardArray[row][col] != '.') {
                    char blockingPieceId = boardArray[row][col];
                    Piece blockingPiece = state.getPieces(blockingPieceId);

                    // Jika kendaraan penghalang vertikal, periksa apakah sulit dipindahkan
                    if (blockingPiece != null && blockingPiece.getOrientation() ==
Piece.Orientation.VERTICAL) {
                        // Periksa apakah ada penghalang di atas atau bawah
                        int blockingRow = blockingPiece.getY();
                        int blockingHeight = blockingPiece.getSize();

                        // Periksa penghalang di atas
                        boolean blockedAbove = blockingRow > 0 && boardArray[blockingRow -
1][col] != '.';

                        // Periksa penghalang di bawah
                        boolean blockedBelow = blockingRow + blockingHeight <
boardArray.length &&
boardArray[blockingRow + blockingHeight][col] !=
'.';

                        if (blockedAbove && blockedBelow) {
                            movabilityPenalty += 3; // Sangat sulit dipindahkan
                        } else if (blockedAbove || blockedBelow) {
                            movabilityPenalty += 1; // Cukup sulit dipindahkan
                        }
                    }
                }
            }
        }
    }
}

```

```

    }
    }
}

return basicBlocking * 2 + movabilityPenalty;
}

private static int calculateCombined(State state) {
    // Kombinasi dari beberapa heuristic dengan bobot
    return calculateManhattanDistance(state) +
        calculateBlockingVehicles(state) * 2 +
        (int)(Math.sqrt(calculateAdvancedBlocking(state)) * 1.5);
}
}

```

4.9 SearchResult.java

```

public class SearchResult {
    public final java.util.List<State> path;
    public final long executionTime;
    public final int visitedNodes;

    public SearchResult(java.util.List<State> path, long executionTime, int visitedNodes) {
        this.path = path;
        this.executionTime = executionTime;
        this.visitedNodes = visitedNodes;
    }
}

```

4.10 Main.java

```

import java.io.*;
import java.nio.file.*;
import java.util.*;

public class Main {
    public static void main(String[] args) {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

        System.out.println("=====");
    }
}

```

```

System.out.println(" Rush Hour Solver [Jakarta Edition]");
System.out.println("=====");

String fileName = "";
Path filePath = null;

while (filePath == null || !Files.exists(filePath)) {
    System.out.println("\nEnter the filename (ex: test1.txt) located in the 'test'
folder:");

    try {
        fileName = reader.readLine().trim();
    } catch (IOException e) {
        System.err.println("Failed to read input. Make sure the file is in valid format
:");
    }

    filePath = Paths.get("test", fileName);
    if (!Files.exists(filePath)) {
        System.err.println("File not found in the 'test' folder: " + fileName);
    }
}

Board board = new Board();
board.readFromFile(filePath.toString());
Map<Character, Piece> pieces = board.getPieceMap();
pieces.put('P', board.getPrimaryPiece());
System.out.println("\nInitial Board Configuration:");
board.printBoard();

State initialState = new State(pieces, board, 0, null, "Initial State");

int choice = -1;
while (choice < 1 || choice > 4) {
    System.out.println("\nSelect the search algorithm:");
    System.out.println("1. UCS (Uniform Cost Search)");
    System.out.println("2. GBFS (Greedy Best First Search)");
    System.out.println("3. A* (A Star Search)");
    System.out.println("4. Beam Search");

    try {
        String input = reader.readLine().trim();
        choice = Integer.parseInt(input);
    }
}

```

```

        if (choice < 1 || choice > 4) {
            System.err.println("Invalid input. Please select a valid option (1-4).");
        }
    } catch (IOException | NumberFormatException e) {
        System.err.println("Invalid input. Please enter a number between 1 and 4.");
    }
}

switch (choice) {
    case 1:
        System.out.println("\nUsing UCS Algorithm");
        UCS.uniformCostSearch(initialState);
        break;
    case 2:
        System.out.println("\nUsing GBFS Algorithm");
        GBFS.solve(initialState);
        break;
    case 3:
        System.out.println("\nUsing A* Algorithm");
        AStar.solve(initialState);
        break;
    case 4:
        System.out.println("\nUsing Beam Search Algorithm");
        BeamSearch.solve(initialState);
        break;
    default:
        System.err.println("Invalid choice.");
        return;
}
}
}

```

4.11 App.java

```

import javax.swing.*.*;
import java.awt.*.*;
import java.io.*.*;
import java.util.*.*;
import java.util.List;
import javax.swing.Timer;

public class App extends JFrame {

```

```

private static final int WIDTH = 800;
private static final int HEIGHT = 700;

private Board board;
private State initialState;
private String selectedAlgorithm = "UCS";
private List<State> solutionPath;
private int currentStep = 0;
private boolean isAnimating = false;
private int selectedHeuristic = Heuristic.MANHATTAN_DISTANCE;

private GamePanel boardPanel;
private JButton loadButton;
private JComboBox<String> algorithmComboBox;
private JComboBox<String> heuristicComboBox;
private JButton solveButton;
private JButton saveButton;
private JButton playPauseButton;
private JButton resetButton;
private JTextArea logArea;
private JSlider speedSlider;
private Timer animationTimer;
private JLabel statusLabel;
private JLabel stepCountLabel;
private JLabel execTimeLabel;
private JLabel visitedNodesLabel;
private JProgressBar solutionProgress;

private final Color ROAD_COLOR = new Color(50, 50, 50);
private final Color EXIT_COLOR = new Color(0, 200, 0);
private final Color PRIMARY_CAR_COLOR = new Color(220, 50, 50);
private final Color BOARD_BACKGROUND = new Color(150, 150, 150);
private final Map<Character, Color> colorMap = new HashMap<>();

public interface SolverCallback {
    void onStep(State currentState);
    void onSolutionFound(State solution, Long executionTime, int visitedNodes);
    void onNoSolution(Long executionTime, int visitedNodes);
}

public App() {
    setTitle("Rush Hour Solver [Jakarta Edition]");
    setSize(WIDTH, HEIGHT);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setLayout(new BorderLayout(5, 5));

```

```

        setLocationRelativeTo(null);
        try {
            UIManager.setLookAndFeel("javax.swing.plaf.nimbus.NimbusLookAndFeel");
        } catch (Exception e) {
            log("Could not set Nimbus look and feel");
        }
        initUI();
    }

    private void initUI() {
        boardPanel = new GamePanel();
        boardPanel.setPreferredSize(new Dimension(WIDTH, HEIGHT - 200));
        boardPanel.setBackground(BOARD_BACKGROUND);

        JPanel boardContainer = new JPanel(new BorderLayout());
        boardContainer.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
        boardContainer.add(boardPanel, BorderLayout.CENTER);

        loadButton = createButton("Load Traffic", "Load a traffic puzzle from file");
        loadButton.addActionListener(e -> loadTraffic());

        String[] algorithms = {"UCS", "GBFS", "A*", "Beam Search"};
        algorithmComboBox = new JComboBox<>(algorithms);
        algorithmComboBox.setToolTipText("Select search algorithm");
        algorithmComboBox.addActionListener(e -> {
            String algo = (String) algorithmComboBox.getSelectedItem();
            heuristicComboBox.setEnabled(!algo.equals("UCS"));
            selectedAlgorithm = algo;
        });

        String[] heuristics = {"Manhattan Distance", "Blocking Vehicles", "Advanced Blocking",
"Combined"};
        heuristicComboBox = new JComboBox<>(heuristics);
        heuristicComboBox.setToolTipText("Select heuristic function");
        heuristicComboBox.setEnabled(false);
        heuristicComboBox.addActionListener(e -> {
            selectedHeuristic = heuristicComboBox.getSelectedIndex();
        });
        solveButton = createButton("Solve", "Solve the puzzle with selected algorithm");
        solveButton.addActionListener(e -> solvePuzzle());

        saveButton = createButton("Save Solution", "Save solution to file");
        saveButton.addActionListener(e -> saveSolution());
    }

```

```

playPauseButton = createButton("Play", "Play/Pause animation");
playPauseButton.setEnabled(false);
playPauseButton.addActionListener(e -> toggleAnimation());

execTimeLabel = new JLabel("Execution time: - ms");
execTimeLabel.setFont(new Font("Arial", Font.PLAIN, 13));
visitedNodesLabel = new JLabel("Visited nodes: -");
visitedNodesLabel.setFont(new Font("Arial", Font.PLAIN, 13));

resetButton = createButton("Reset", "Reset to initial state");
resetButton.setEnabled(false);
resetButton.addActionListener(e -> resetAnimation());

speedSlider = new JSlider(1, 10, 5);
speedSlider.setMajorTickSpacing(1);
speedSlider.setPaintTicks(true);
speedSlider.setPaintLabels(true);
speedSlider.setToolTipText("Animation speed");
speedSlider.setPreferredSize(new Dimension(150, 40));
speedSlider.addChangeListener(e -> updateAnimationSpeed());

stepCountLabel = new JLabel("Step: 0/0");
stepCountLabel.setFont(new Font("Arial", Font.BOLD, 14));

solutionProgress = new JProgressBar(0, 100);
solutionProgress.setStringPainted(true);
solutionProgress.setString("No solution");
solutionProgress.setPreferredSize(new Dimension(150, 20));

statusLabel = new JLabel("Status: Ready");
statusLabel.setForeground(new Color(0, 100, 200));
statusLabel.setFont(new Font("Arial", Font.BOLD, 14));

JPanel controlPanel = new JPanel();
controlPanel.setLayout(new BoxLayout(controlPanel, BoxLayout.Y_AXIS));

JPanel topControls = new JPanel(new FlowLayout(FlowLayout.LEFT, 5, 0));
topControls.add(loadButton);
topControls.add(new JLabel("Algorithm:"));
topControls.add(algorithmComboBox);
topControls.add(new JLabel("Heuristic:"));
topControls.add(heuristicComboBox);
topControls.add(solveButton);

```



```

topControls.add(saveButton);

JPanel animationControls = new JPanel(new FlowLayout(FlowLayout.LEFT, 5, 0));
animationControls.add(playPauseButton);
animationControls.add(resetButton);
animationControls.add(new JLabel("Speed:"));
animationControls.add(speedSlider);
animationControls.add(stepCountLabel);

JPanel statusPanel = new JPanel(new FlowLayout(FlowLayout.LEFT, 5, 0));
statusPanel.add(statusLabel);
statusPanel.add(Box.createHorizontalStrut(20));
statusPanel.add(solutionProgress);

statusPanel.add(Box.createHorizontalStrut(20));
statusPanel.add(execTimeLabel);
statusPanel.add(Box.createHorizontalStrut(10));
statusPanel.add(visitedNodesLabel);

controlPanel.add(Box.createVerticalStrut(5));
controlPanel.add(topControls);
controlPanel.add(Box.createVerticalStrut(5));
controlPanel.add(animationControls);
controlPanel.add(Box.createVerticalStrut(5));
controlPanel.add(statusPanel);
controlPanel.add(Box.createVerticalStrut(5));

controlPanel.setBorder(BorderFactory.createCompoundBorder(
    BorderFactory.createEmptyBorder(5, 10, 5, 10),
    BorderFactory.createCompoundBorder(
        BorderFactory.createLineBorder(Color.GRAY),
        BorderFactory.createEmptyBorder(5, 5, 5, 5)
    )
));

logArea = new JTextArea(6, 50);
logArea.setEditable(false);
logArea.setFont(new Font("Consolas", Font.PLAIN, 12));
logArea.setBackground(new Color(240, 240, 240));
logArea.setBorder(BorderFactory.createEmptyBorder(5, 5, 5, 5));

JScrollPane logScroll = new JScrollPane(logArea);
logScroll.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);

```

```

        logScroll.setBorder(BorderFactory.createTitledBorder(BorderFactory.createEtchedBorder(),
"Log Output"));

        add(boardContainer, BorderLayout.CENTER);
        add(controlPanel, BorderLayout.NORTH);
        add(logScroll, BorderLayout.SOUTH);

        animationTimer = new Timer(200, e -> stepAnimation());
    }

    private JButton createButton(String text, String tooltip) {
        JButton button = new JButton(text);
        button.setToolTipText(tooltip);
        button.setFocusPainted(false);
        return button;
    }

    private void toggleAnimation() {
        if (solutionPath == null || solutionPath.isEmpty()) {
            return;
        }

        if (isAnimating) {
            animationTimer.stop();
            playPauseButton.setText("Play");
            isAnimating = false;
        } else {
            if (currentStep >= solutionPath.size()) {
                currentStep = 0;
            }
            animationTimer.start();
            playPauseButton.setText("Pause");
            isAnimating = true;
        }
    }

    private void resetAnimation() {
        if (solutionPath == null || solutionPath.isEmpty()) {
            return;
        }

        animationTimer.stop();
        isAnimating = false;
        currentStep = 0;
    }

```

```

        if (solutionPath.size() > 0) {
            board = solutionPath.get(0).getBoard().copy();
            updateStepDisplay();
            boardPanel.repaint();
        }

        playPauseButton.setText("Play");
    }

    private void updateAnimationSpeed() {
        if (animationTimer != null) {
            int delay = 1000 / speedSlider.getValue();
            animationTimer.setDelay(delay);
        }
    }

    private void stepAnimation() {
        if (solutionPath == null || currentStep >= solutionPath.size()) {
            animationTimer.stop();
            playPauseButton.setText("Play");
            isAnimating = false;
            return;
        }

        State currentState = solutionPath.get(currentStep);
        board = currentState.getBoard().copy();
        log("Move: " + currentState.getMove());
        updateStepDisplay();
        boardPanel.repaint();
        currentStep++;
        updateAnimationProgress();
    }

    private void updateAnimationProgress() {
        if (solutionPath != null && solutionPath.size() > 1) {
            int percentage = (int) Math.round((currentStep * 100.0) / Math.max(1,
solutionPath.size() - 1));
            percentage = Math.max(0, Math.min(100, percentage));
            solutionProgress.setValue(percentage);
            solutionProgress.setString(percentage + "% Complete");
        }
    }
}

```

```

private void updateStepDisplay() {
    if (solutionPath != null) {
        stepCountLabel.setText("Step: " + currentStep + "/" + (solutionPath.size() - 1));
        if (solutionPath.size() > 1) {
            int percentage = (currentStep * 100) / (solutionPath.size() - 1);
            solutionProgress.setValue(percentage);
            solutionProgress.setString(percentage + "% Complete");
        }
    }
}

private void loadTraffic() {
    JFileChooser fileChooser = new JFileChooser("test");
    fileChooser.setDialogTitle("Load Traffic Puzzle");
    int result = fileChooser.showOpenDialog(this);
    if (result == JFileChooser.APPROVE_OPTION) {
        try {
            File file = fileChooser.getSelectedFile();
            board = new Board();
            board.readFromFile(file.getPath());
            Map<Character, Piece> pieces = board.getPieceMap();
            pieces.put('P', board.getPrimaryPiece());
            initialState = new State(pieces, board, 0, null, "Initial State");
            log("Loaded board from: " + file.getName());
            status("Board loaded successfully!");
            solutionPath = null;
            currentStep = 0;
            solutionProgress.setValue(0);
            solutionProgress.setString("No solution");
            stepCountLabel.setText("Step: 0/0");
            playPauseButton.setEnabled(false);
            resetButton.setEnabled(false);
            refreshColorMap();
            boardPanel.repaint();
        } catch (Exception e) {
            showError("Error loading board: " + e.getMessage());
        }
    }
}

private void refreshColorMap() {
    colorMap.clear();
    colorMap.put('P', PRIMARY_CAR_COLOR);
    colorMap.put('K', EXIT_COLOR);
}

```

```

Random r = new Random(42);
if (board != null) {
    for (Piece piece : board.getNonPieceList()) {
        char c = piece.getPieceChar();
        if (!colorMap.containsKey(c)) {
            int hue = (int)(c * 20) % 360;
            float saturation = 0.7f + r.nextFloat() * 0.3f;
            float brightness = 0.7f + r.nextFloat() * 0.3f;
            colorMap.put(c, Color.getHSBColor(hue/360f, saturation, brightness));
        }
    }
}

private void solvePuzzle() {
    if (initialState == null) {
        showError("Please load a board first!");
        return;
    }

    solveButton.setEnabled(false);
    status("Solving with " + selectedAlgorithm + "... Please wait");
    solutionProgress.setIndeterminate(true);
    solutionProgress.setString("Solving...");
    SwingWorker<Void, State> worker = new SwingWorker<Void, State>() {
        SearchResult result = null;

        @Override
        protected Void doInBackground() {
            switch (selectedAlgorithm) {
                case "UCS":
                    result = UCS.GUIUniformCostSearch(initialState);
                    execTimeLabel.setText("Execution time: " + result.executionTime + " ms");
                    visitedNodesLabel.setText("Visited nodes: " + result.visitedNodes);
                    break;
                case "GBFS":
                    result = GBFS.GUIIsolve(initialState, selectedHeuristic);
                    execTimeLabel.setText("Execution time: " + result.executionTime + " ms");
                    visitedNodesLabel.setText("Visited nodes: " + result.visitedNodes);
                    break;
                case "A*":
                    result = AStar.GUIIsolve(initialState, selectedHeuristic);
                    execTimeLabel.setText("Execution time: " + result.executionTime + " ms");
                    visitedNodesLabel.setText("Visited nodes: " + result.visitedNodes);

```

```

        break;
    case "Beam Search":
        //2
        result = BeamSearch.GUISolve(initialState, 2, selectedHeuristic);
        execTimeLabel.setText("Execution time: " + result.executionTime + " ms");
        visitedNodesLabel.setText("Visited nodes: " + result.visitedNodes);
        break;
    }
    return null;
}

@Override
protected void done() {
    solutionProgress.setIndeterminate(false);
    if (result != null) {
        solutionPath = result.path;
        currentStep = 0;
        log("Execution time: " + result.executionTime + " ms");
        log("Visited nodes: " + result.visitedNodes);
        log("Steps in solution: " + (solutionPath.size() - 1));
        if (solutionPath != null && solutionPath.size() > 1) {
            solutionProgress.setValue(0);
            solutionProgress.setString("0% (Ready to play)");
            stepCountLabel.setText("Step: 0/" + (solutionPath.size() - 1));
            status("Solution found! Press Play to start animation.");
            playPauseButton.setEnabled(true);
            resetButton.setEnabled(true);
        } else {
            solutionProgress.setValue(0);
            solutionProgress.setString("No solution found");
            status("No solution found!");
        }
    } else {
        solutionProgress.setValue(0);
        solutionProgress.setString("No solution found");
        status("No solution found!");
    }
    solveButton.setEnabled(true);
}

};
worker.execute();
}

private void saveSolution() {

```

```

        if (solutionPath == null || solutionPath.isEmpty()) {
            showError("No solution to save.");
            return;
        }

        JFileChooser fileChooser = new JFileChooser("test");
        fileChooser.setDialogTitle("Save Solution");
        int result = fileChooser.showSaveDialog(this);
        if (result == JFileChooser.APPROVE_OPTION) {
            File file = fileChooser.getSelectedFile();
            String path = file.getPath().endsWith(".txt") ? file.getPath() : file.getPath() +
".txt";

            try (PrintWriter writer = new PrintWriter(path)) {
                writer.println("Rush Hour Solution - " + new java.util.Date());
                writer.println("Algorithm used: " + selectedAlgorithm);
                writer.println("Total steps: " + (solutionPath.size() - 1));
                writer.println("-----");
                writer.println();
                for (int i = 0; i < solutionPath.size(); i++) {
                    State s = solutionPath.get(i);
                    writer.println("Step " + i + ": " + s.getMove());
                    writer.println(s.toStringWithoutColor());
                    writer.println();
                }
                log("Solution saved to " + path);
                status("Solution saved successfully!");
            } catch (IOException e) {
                showError("Failed to save solution: " + e.getMessage());
            }
        }
    }

    private void log(String msg) {
        logArea.append(msg + "\n");
        logArea.setCaretPosition(logArea.getDocument().getLength());
    }

    private void showError(String msg) {
        JOptionPane.showMessageDialog(this, msg, "Error", JOptionPane.ERROR_MESSAGE);
        status("Error: " + msg);
    }

    private void status(String msg) {
        statusLabel.setText("Status: " + msg);
    }

```

```

    }

    public static void main(String[] args) {
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (Exception e) {
            System.err.println("Failed to set look and feel");
        }
        SwingUtilities.invokeLater(() -> {
            App app = new App();
            app.setVisible(true);
        });
    }

    private class GamePanel extends JPanel {
        private static final int CELL_PADDING = 4;
        private static final int CELL_CORNER_RADIUS = 15;
        private final int ROAD_MARK_COUNT = 5;
        private final int ROAD_MARK_WIDTH = 20;
        private final int ROAD_MARK_HEIGHT = 5;

        public GamePanel() {
            setBackground(ROAD_COLOR);
        }

        @Override
        protected void paintComponent(Graphics g) {
            super.paintComponent(g);
            if (board == null) {
                drawEmptyBoard(g);
                return;
            }
            Graphics2D g2 = (Graphics2D) g;
            g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
            int rows = board.getRows();
            int cols = board.getCols();
            int cellWidth = getWidth() / cols;
            int cellHeight = getHeight() / rows;
            int cellSize = Math.min(cellWidth, cellHeight);
            int xOffset = (getWidth() - (cellSize * cols)) / 2;
            int yOffset = (getHeight() - (cellSize * rows)) / 2;
            g2.setColor(ROAD_COLOR);
            g2.fillRect(xOffset, yOffset, cellSize * cols, cellSize * rows);

```



```

g2.setColor(Color.YELLOW);
for (int row = 1; row < rows; row++) {
    int y = yOffset + row * cellSize - ROAD_MARK_HEIGHT / 2;
    for (int i = 0; i < ROAD_MARK_COUNT; i++) {
        int x = xOffset + (i * 2 * cellSize / ROAD_MARK_COUNT);
        g2.fillRect(x, y, ROAD_MARK_WIDTH, ROAD_MARK_HEIGHT);
    }
}
for (int col = 1; col < cols; col++) {
    int x = xOffset + col * cellSize - ROAD_MARK_WIDTH / 2;
    for (int i = 0; i < ROAD_MARK_COUNT; i++) {
        int y = yOffset + (i * 2 * cellSize / ROAD_MARK_COUNT);
        g2.fillRect(x, y, ROAD_MARK_WIDTH, ROAD_MARK_HEIGHT);
    }
}
for (int row = 0; row < rows; row++) {
    for (int col = 0; col < cols; col++) {
        drawCell(g2, row, col, cellSize, xOffset, yOffset);
    }
}
int exitX = board.getExitX();
int exitY = board.getExitY();
if (exitX >= 0 && exitY >= 0) {
    drawExit(g2, exitY, exitX, cellSize, xOffset, yOffset);
} else if (exitX == -1) {
    drawExit(g2, exitY, -1, cellSize, xOffset, yOffset);
} else if (exitY == -1) {
    drawExit(g2, -1, exitX, cellSize, xOffset, yOffset);
} else if (exitX >= cols) {
    drawExit(g2, exitY, cols, cellSize, xOffset, yOffset);
} else if (exitY >= rows) {
    drawExit(g2, rows, exitX, cellSize, xOffset, yOffset);
}
drawGameInfo(g2);
}

private void drawEmptyBoard(Graphics g) {
    Graphics2D g2 = (Graphics2D) g;
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
    g2.setColor(BOARD_BACKGROUND);
    g2.fillRect(0, 0, getWidth(), getHeight());
    g2.setColor(Color.DARK_GRAY);
    g2.setFont(new Font("Arial", Font.BOLD, 24));

```

```

        String message = "Load a traffic puzzle to start";
        FontMetrics fm = g2.getFontMetrics();
        int textWidth = fm.stringWidth(message);
        int x = (getWidth() - textWidth) / 2;
        int y = getHeight() / 2;
        g2.drawString(message, x, y);
    }

    private void drawCell(Graphics2D g2, int row, int col, int cellSize, int xOffset, int
yOffset) {
        char c = board.getChar(row, col);
        if (c == '.') return;
        int x = col * cellSize + xOffset;
        int y = row * cellSize + yOffset;
        Color pieceColor = colorMap.getOrDefault(c, Color.LIGHT_GRAY);
        g2.setColor(pieceColor);
        g2.fillRoundRect(
            x + CELL_PADDING,
            y + CELL_PADDING,
            cellSize - 2 * CELL_PADDING,
            cellSize - 2 * CELL_PADDING,
            CELL_CORNER_RADIUS,
            CELL_CORNER_RADIUS
        );
        g2.setColor(new Color(0, 0, 0, 40));
        g2.fillRoundRect(
            x + CELL_PADDING,
            y + CELL_PADDING,
            cellSize - 2 * CELL_PADDING,
            cellSize / 4,
            CELL_CORNER_RADIUS,
            CELL_CORNER_RADIUS
        );
        if (c == 'P') {
            drawCarDetails(g2, x, y, cellSize);
        }
        g2.setColor(getContrastColor(pieceColor));
        g2.setFont(new Font("Arial", Font.BOLD, cellSize / 3));
        FontMetrics fm = g2.getFontMetrics();
        String text = String.valueOf(c);
        int textWidth = fm.stringWidth(text);
        int textHeight = fm.getHeight();
        g2.drawString(text,
            x + (cellSize - textWidth) / 2,

```

```

        y + (cellSize + textHeight / 3) / 2
    );
}

private void drawCarDetails(Graphics2D g2, int x, int y, int cellSize) {
    g2.setColor(new Color(200, 230, 255, 180));
    int windowSize = cellSize / 4;
    g2.fillRoundRect(
        x + cellSize / 3,
        y + cellSize / 4,
        windowSize,
        windowSize,
        5, 5
    );
    g2.setColor(Color.YELLOW);
    int lightSize = cellSize / 10;
    g2.fillOval(
        x + cellSize - cellSize / 4,
        y + cellSize / 3,
        lightSize,
        lightSize
    );
}

private void drawExit(Graphics2D g2, int row, int col, int cellSize, int xOffset, int
yOffset) {
    int x, y;
    int width = cellSize / 2;
    int height = cellSize / 2;
    if (col == -1) {
        x = xOffset - width / 2;
        y = yOffset + row * cellSize + cellSize / 4;
    } else if (col >= board.getCols()) {
        x = xOffset + board.getCols() * cellSize - width / 2;
        y = yOffset + row * cellSize + cellSize / 4;
    } else if (row == -1) {
        x = xOffset + col * cellSize + cellSize / 4;
        y = yOffset - height / 2;
    } else if (row >= board.getRows()) {
        x = xOffset + col * cellSize + cellSize / 4;
        y = yOffset + board.getRows() * cellSize - height / 2;
    } else {
        x = xOffset + col * cellSize + cellSize / 4;
        y = yOffset + row * cellSize + cellSize / 4;
    }
}

```

```

    }
    g2.setColor(EXIT_COLOR);
    g2.fillRoundRect(x, y, width, height, 10, 10);
    g2.setColor(Color.WHITE);
    g2.setStroke(new BasicStroke(2));
    g2.drawRoundRect(x, y, width, height, 10, 10);
    g2.setFont(new Font("Arial", Font.BOLD, cellSize / 6));
    g2.setColor(Color.WHITE);
    FontMetrics fm = g2.getFontMetrics();
    String exitText = "EXIT";
    int textWidth = fm.stringWidth(exitText);
    g2.drawString(exitText, x + (width - textWidth) / 2, y + height / 2 + fm.getHeight()
/ 4);
}

private void drawGameInfo(Graphics2D g2) {
    g2.setFont(new Font("Arial", Font.BOLD, 14));
    g2.setColor(Color.WHITE);
    g2.drawString("Rush Hour Solver", 10, 20);
    if (solutionPath != null && currentStep > 0 && currentStep < solutionPath.size()) {
        String move = solutionPath.get(currentStep).getMove();
        g2.setFont(new Font("Arial", Font.BOLD, 16));
        g2.setColor(Color.YELLOW);
        g2.drawString(move, 10, 50);
    }
}

private Color getContrastColor(Color bg) {
    double brightness = (bg.getRed() * 299 + bg.getGreen() * 587 + bg.getBlue() * 114) /
1000.0;
    return brightness > 128 ? Color.BLACK : Color.WHITE;
}
}

```

Bab 5

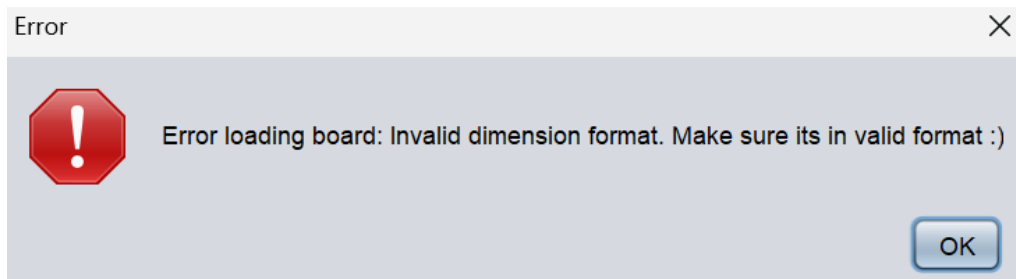
Eksperimen

Eksperimen dijalankan dengan menggunakan GUI yaitu pada file App.java dengan dicompile terlebih dahulu dengan urutan instruksi:

1. `javac -d bin src/*.java`
2. `java -cp bin App`

5.1 Pengujian test1_result.txt

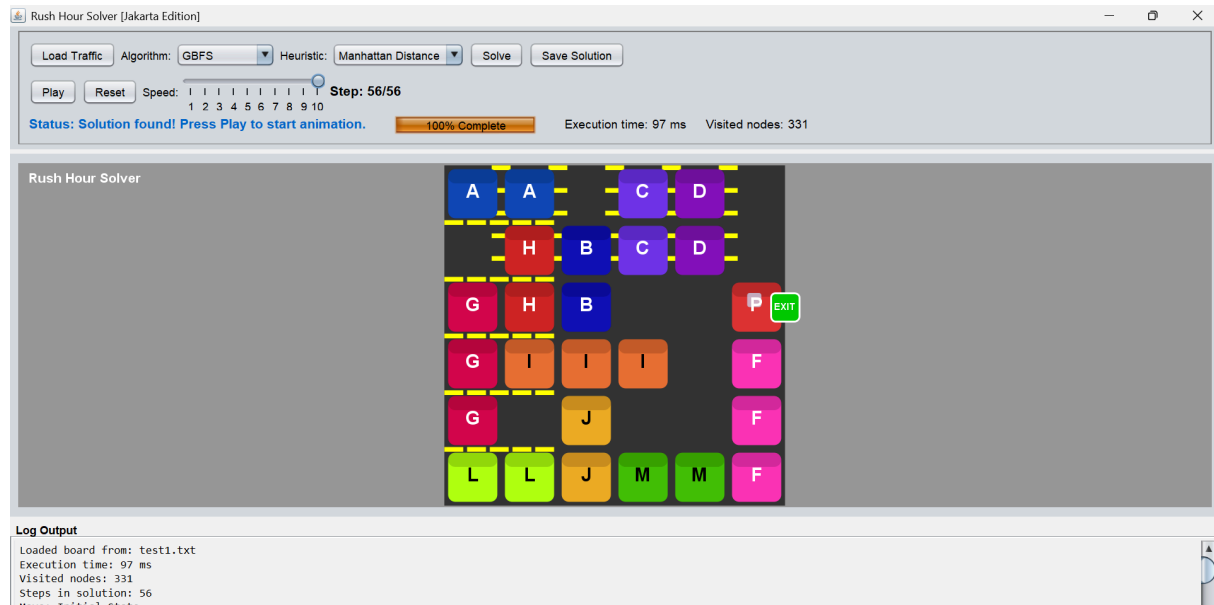
Input tidak valid karena format tidak sesuai



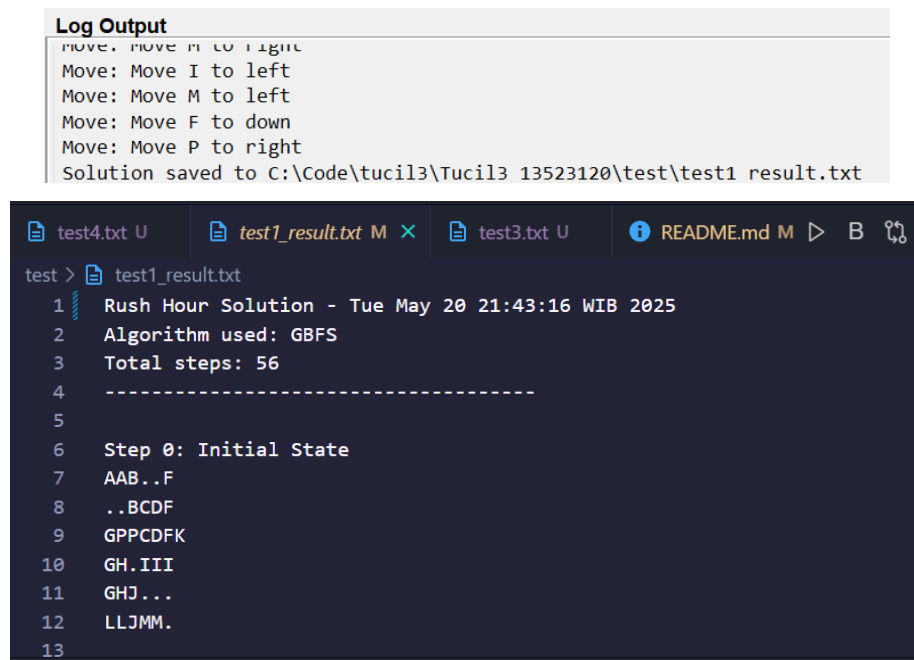
Gambar 6. Validasi program saat memuat format yang salah

5.2 Pengujian test1.txt

6 6
12
AAB..F
..BCDF
GPPCDFK
GH.III
GHJ...
LLJMM.



Gambar 7. Program berhasil mencari solusi

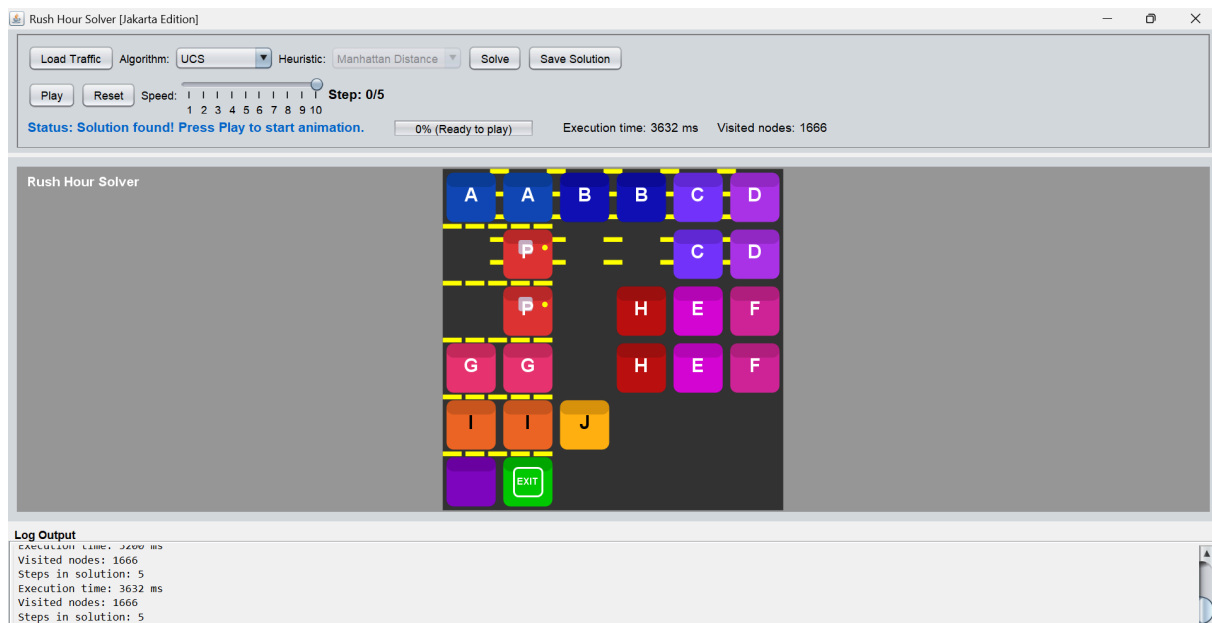


Gambar 8. Program berhasil menyimpan solusi dari kasus test1.txt

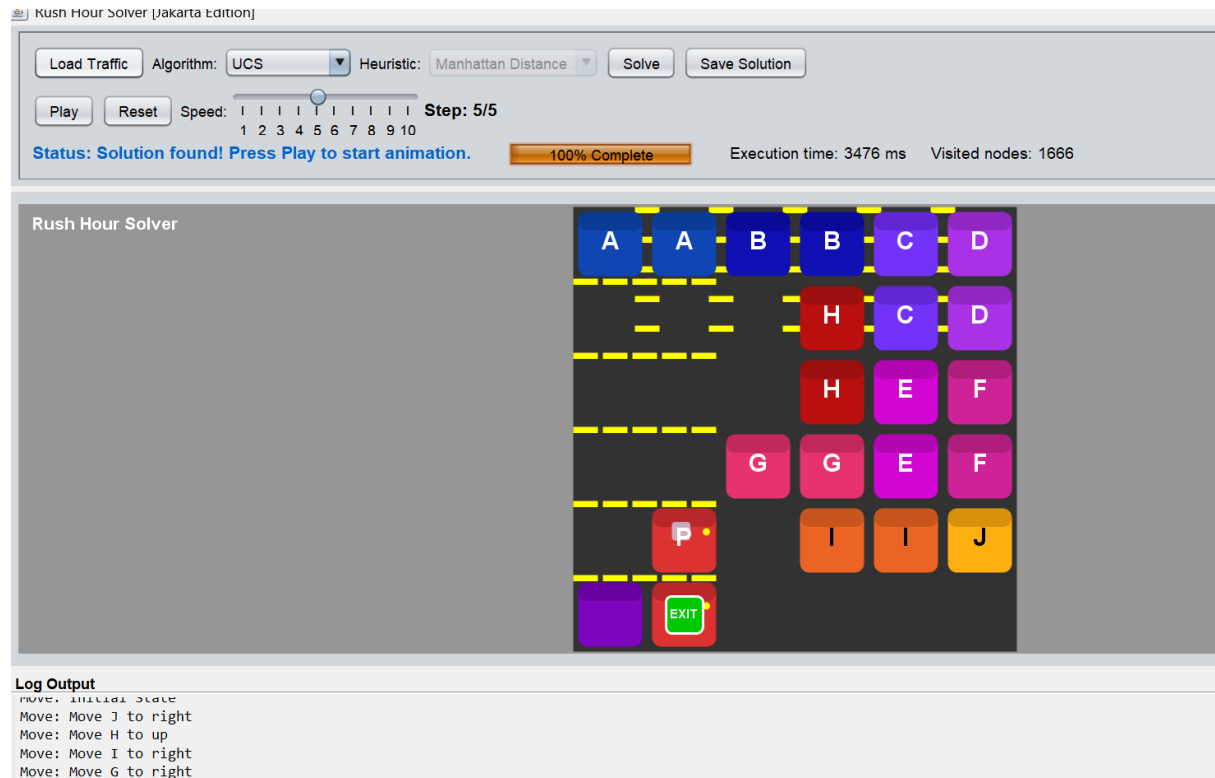
5.3 Pengujian test2.txt

test2.txt:

6 6
 10
 AABBCD
 .P..CD
 .P.HEF
 GG.HEF
 IIJ...
 K



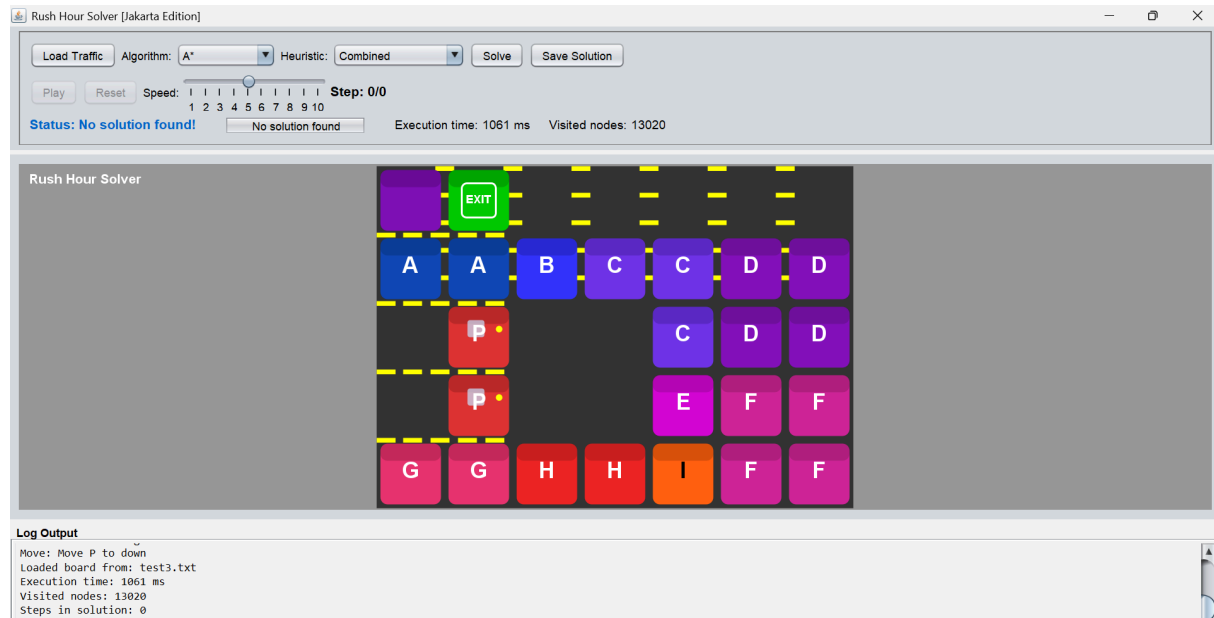
Gambar 9. Program berhasil memuat file test2.txt



Gambar 10. Program berhasil menemukan solusi dari kasus test2.txt

5.4 Pengujian test3.txt

5 7
 9
 K
 AABCCDD
 .P..CDD
 .P..EFF
 GGHHIFF



Gambar 11. Program berhasil menemukan solusi dari kasus file test3.txt (tidak ada solusi)

5.5 Pengujian test4.txt

8 8

14

.AA.BB..

.P..D.EC

.P..D.EC

FFGG..HH

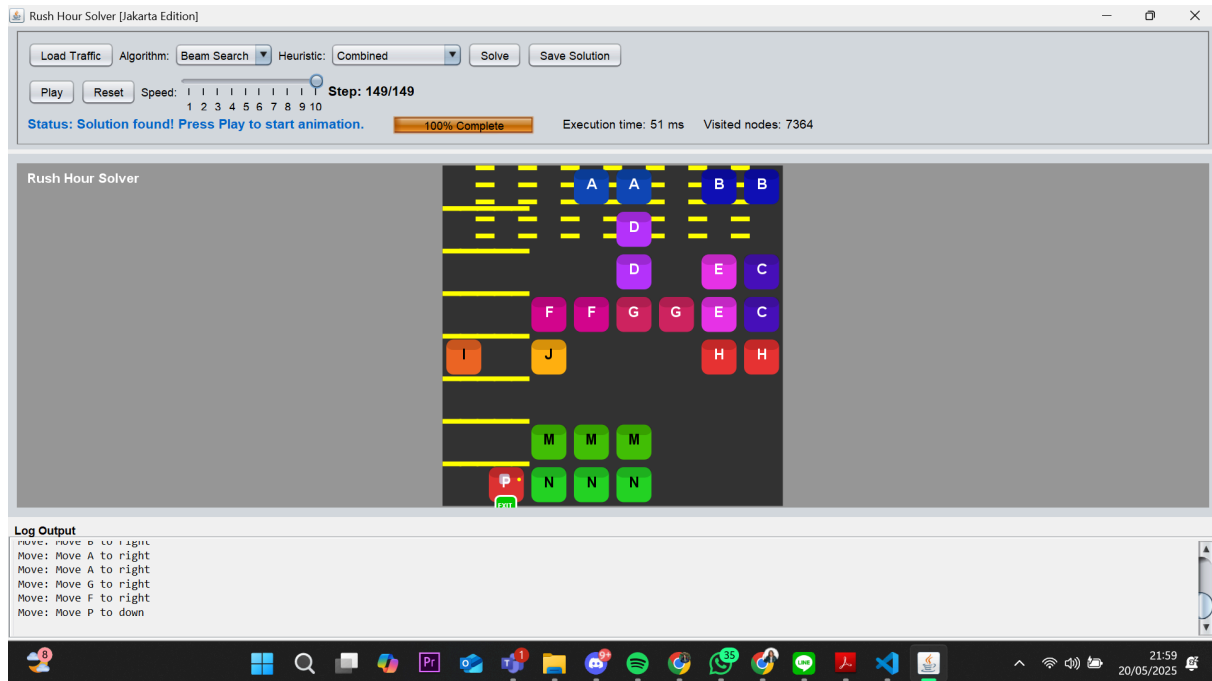
I.J...HH

..KKK...

..MMM...

..NNN...

K



Gambar 12. Program berhasil menemukan solusi dari kasus file test4.txt

Analisis pengujian:

Uniform Cost Search (UCS) menunjukkan hasil optimal dengan waktu eksekusi yang relatif lambat pada papan besar, karena mengeksplorasi semua jalur berdasarkan *cost* langkah tanpa mempertimbangkan estimasi jarak ke goal. Kompleksitas waktu UCS adalah $O(b^d)$, dengan b adalah branching factor dan d adalah kedalaman solusi. Jumlah node yang dikunjungi pada kasus sedang bisa mencapai ribuan.

Greedy Best First Search (GBFS) bekerja sangat cepat pada kasus tertentu karena hanya mempertimbangkan $h(n)$ (estimasi jarak ke goal). Namun, pada papan yang kompleks, GBFS dapat gagal memberikan solusi optimal atau bahkan terjebak di jalur yang salah. Kompleksitas waktu bergantung pada kualitas heuristik, namun dalam praktik sering lebih kecil dari UCS karena cakupan pencarian lebih sempit.

A* Search menggabungkan UCS dan GBFS dengan $f(n) = g(n) + h(n)$, dan secara konsisten memberikan solusi optimal selama heuristik admissible. Dibanding UCS, A* mengunjungi lebih sedikit simpul karena lebih selektif. Kompleksitas teoretisnya

juga $O(b^d)$, namun biasanya lebih cepat secara praktik karena pruning berdasarkan estimasi.

Beam Search sebagai algoritma bonus membatasi jumlah node yang dipertimbangkan per level (beam width). Hasilnya, Beam Search memberikan waktu eksekusi yang sangat cepat, bahkan pada kasus besar. Namun, karena membatasi eksplorasi, solusi yang ditemukan belum tentu optimal, dan kadang tidak menemukan solusi sama sekali. Beam Search tidak lengkap dan tidak optimal secara teori, namun efisien secara praktiknya.


Bab 6

Lampiran

Pranala *repository*:

https://github.com/bevindav/Tucil3_13523120/

Referensi:

 Beam Search Algorithm in Artificial Intelligence | All Imp Points | Heuristic Sear...

[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-\(2025\)-Bagian1.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-(2025)-Bagian1.pdf)

[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-\(2025\)-Bagian2.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf)

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	