LAPORAN TUGAS KECIL 3
MATA KULIAH IF2211 STRATEGI ALGORITMA
PENYELESAIAN PUZZLE RUSH HOUR MENGGUNAKAN ALGORITMA PATHFINDING

Dosen Pengampu:
Dr. Nur Ulfa Maulidevi, S.T, M.Sc.

Disusun Oleh:

Nadhif Radityo Nugroho 13523045

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

MEI 2025

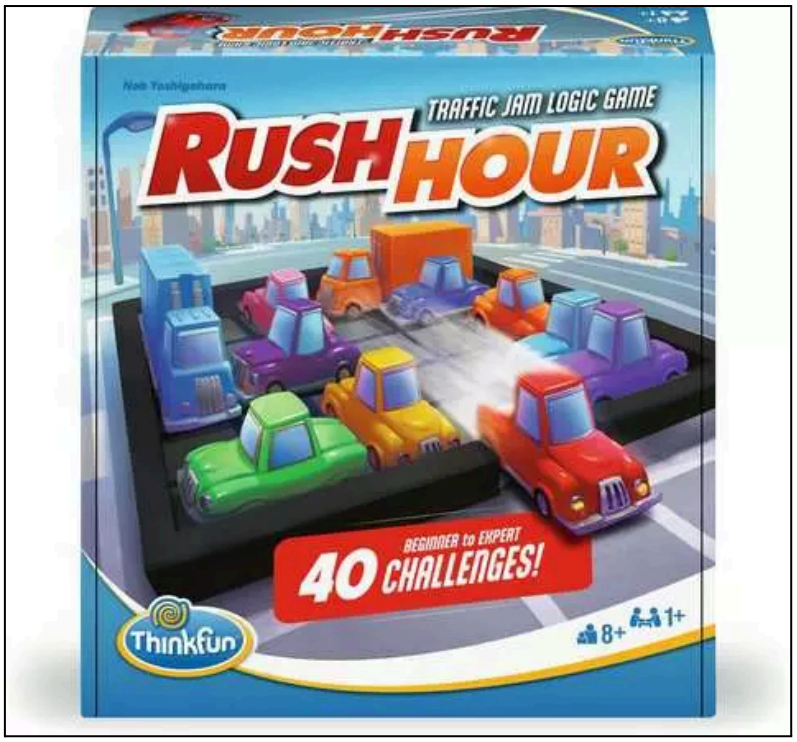
DAFTAR ISI

DAFTAR ISI **2**
BAB I LATAR BELAKANG **4**
1.1. Permainan Rush Hour 4
1.2. Uniform Cost Search (UCS) 5
1.3. Greedy Best First Search (GBFS) 6
1.4. A-Star (A*) 7
1.5. Iterative Deepening A-Star (IDA*) 8
BAB II DESKRIPSI DAN IMPLEMENTASI ALGORITMA **9**
2.1. Permainan Rush Hour 9
2.2. Algoritma Uniform Cost Search 10
2.3. Algoritma Greedy Best First Search 11
2.4. Algoritma A-Star 11
2.5. Algoritma Iterative Deepening A-Star 12
2.6. Branching Factor 13
2.7. Heuristik Car Distance 14
2.8. Heuristik Car Blocked 15
2.9. Heuristik Car Blocked Recursive 16
BAB III SEKILAS KODE PROGRAM **17**
3.1. Struktur Program 17
3.2. Implementasi Program 17
3.3. Optimasi Program - Iterating Cars for State Expansion 22
3.4. Optimasi Program - Fast Queue Insertion 23
3.5. Optimasi Program - Stack Solver (IDA*) Fasttrack Approximation 25
3.6. Optimasi Program - Data Structure, Caching, and Fulfilled Assumptions 26
3.7. Bonus - Algoritma Alternatif 29
3.8. Bonus - Heuristic Alternatif 30
3.9. Bonus - GUI 31
3.10. Bonus - Deployment 31
3.11. Bonus - Video Animation Output 32
BAB IV PERCOBAAN PROGRAM **33**
4.1. Jam-1 33
4.2. Jam-2 34
4.3. Jam-3 35
4.4. Jam-4 36
4.5. Jam-5 37
4.6. Jam-6 38
4.7. Jam-7 39
4.8. Jam-8 40
4.9. Jam-9 41

4.10. Jam-10 42
4.11. Jam-11 43
4.12. Jam-12 44
4.13. Jam-13 45
4.14. Jam-14 46
4.15. Jam-15 47
4.16. Jam-16 48
4.17. Jam-17 49
4.18. Jam-18 50
4.19. Jam-19 51
4.20. Jam-20 52
4.21. Jam-21 53
4.22. Jam-22 54
4.23. Jam-23 55
4.24. Jam-24 56
4.25. Jam-25 57
4.26. Jam-26 58
4.27. Jam-27 59
4.28. Jam-28 60
4.29. Jam-29 61
4.30. Jam-30 62
4.31. Jam-31 63
4.32. Jam-32 64
4.33. Jam-33 65
4.34. Jam-34 66
4.35. Jam-35 67
4.36. Jam-36 68
4.37. Jam-37 69
4.38. Jam-38 70
4.39. Jam-39 71
4.40. Jam-40 72
4.41. Invalid Input 73
BAB V ANALISIS 79
5.1. Analisis 79
5.2. Kesimpulan 80
5.3. Saran 80
LAMPIRAN 81
Tautan 81
Tabel Laporan 81

BAB I LATAR BELAKANG

1.1. Permainan Rush Hour



Gambar 1. Rush Hour *Puzzle*

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

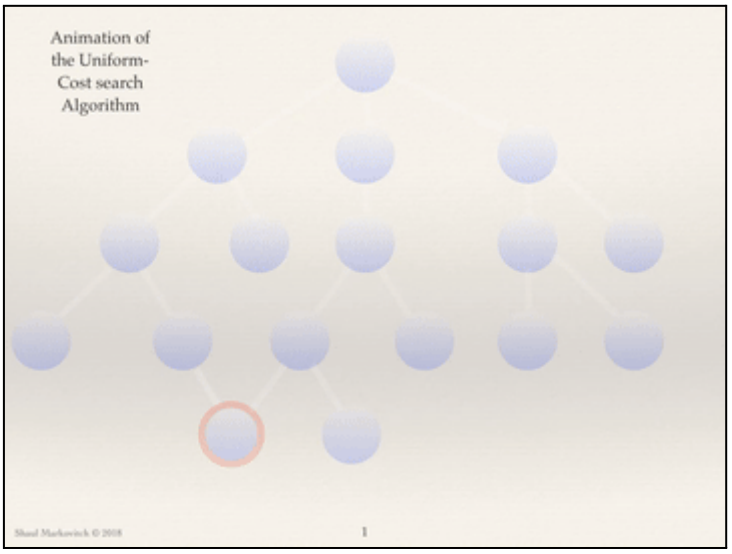
Rush Hour adalah sebuah permainan *puzzle* logika berbasis grid yang menantang pemain untuk menggeser kendaraan di dalam sebuah kotak (biasanya berukuran 6x6) agar mobil utama (biasanya berwarna merah) dapat keluar dari kemacetan melalui pintu keluar di sisi papan. Setiap kendaraan hanya bisa bergerak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

1.2. *Uniform Cost Search* (UCS)



Gambar 2. Animasi Pencarian Algoritma UCS

(Sumber:https://www.youtube.com/watch?v=XyoucHYKYSE)

Uniform Cost Search (UCS) adalah algoritma pencarian yang termasuk dalam kategori *uninformed search*, di mana pencarian dilakukan tanpa pengetahuan tambahan tentang tujuan akhir selain struktur graf. UCS bekerja dengan prinsip eksplorasi jalur berdasarkan biaya terkecil dari titik awal, menggunakan antrian prioritas untuk memilih simpul dengan total biaya terendah. Dalam konteks permainan Rush Hour, UCS akan mengevaluasi semua kemungkinan gerakan kendaraan berdasarkan total langkah yang telah dilakukan, tanpa mempertimbangkan seberapa dekat posisi tersebut dengan solusi akhir. Hal ini membuat UCS

bersifat optimal—selalu menemukan solusi dengan biaya terkecil jika biaya tiap langkah dianggap seragam—namun bisa sangat lambat karena mengevaluasi banyak kemungkinan.

Kelebihan utama UCS dalam Rush Hour adalah kemampuannya menemukan solusi optimal, yaitu jalur dengan jumlah gerakan paling sedikit yang diperlukan untuk mengeluarkan mobil utama dari papan. Namun, karena UCS tidak mempertimbangkan arah atau jarak menuju tujuan secara eksplisit, algoritma ini cenderung mengeksplorasi banyak jalur yang sebenarnya tidak efisien. Hal ini dapat menyebabkan waktu pencarian yang lama, terutama pada konfigurasi papan yang kompleks dengan banyak kendaraan penghalang. Meskipun demikian, UCS tetap menjadi tolok ukur penting dalam membandingkan performa algoritma pencarian lain seperti *Greedy Best-First Search*, A*, dan IDA*, yang menggunakan pendekatan heuristik untuk mengarahkan pencarian ke solusi lebih cepat.

1.3. *Greedy Best First Search* (GBFS)

Gambar 3. Animasi Pencarian Algoritma GBFS

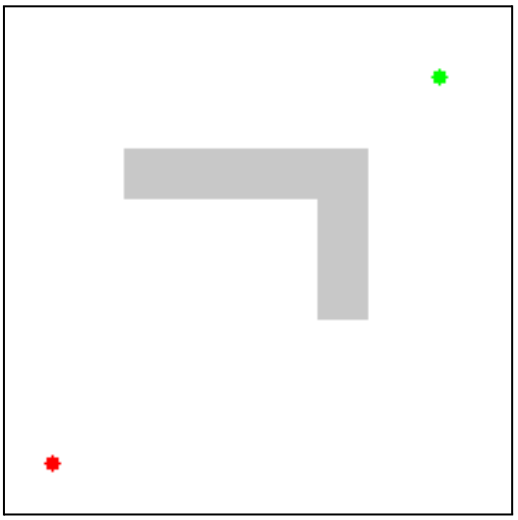
(Sumber: https://www.youtube.com/watch?v=A8pmud1Uh0Q)

Greedy Best-First Search (GBFS) adalah algoritma pencarian berbasis heuristik yang berfokus pada mempercepat proses pencarian solusi dengan memprioritaskan perluasan simpul yang tampak paling dekat dengan tujuan. Tidak seperti *Uniform Cost Search* yang mempertimbangkan biaya total dari titik awal, GBFS hanya mempertimbangkan nilai heuristik dari simpul saat ini untuk menentukan jalur mana yang akan dieksplorasi terlebih dahulu. Dalam permainan Rush Hour, GBFS akan memilih konfigurasi papan yang menurut fungsi heuristik tampak paling menjanjikan untuk mendekatkan mobil utama ke pintu keluar, tanpa memperhitungkan jumlah langkah yang sudah dilakukan.

Keunggulan utama GBFS adalah kecepatannya dalam menemukan solusi dibandingkan algoritma yang mempertimbangkan seluruh biaya jalur, karena hanya berfokus pada arah menuju tujuan. Namun, kelemahannya terletak pada sifatnya yang tidak optimal dan tidak

lengkap. Algoritma ini bisa saja terjebak pada jalur yang tampak menjanjikan namun sebenarnya membawa ke solusi tidak efisien. Dalam konteks Rush Hour, hal ini berarti GBFS dapat dengan cepat menemukan solusi, tetapi solusi tersebut belum tentu merupakan solusi dengan langkah paling sedikit. Oleh karena itu, meskipun cepat, GBFS lebih cocok digunakan ketika kecepatan lebih diutamakan daripada optimalitas.

1.4. *A-Star* (A*)



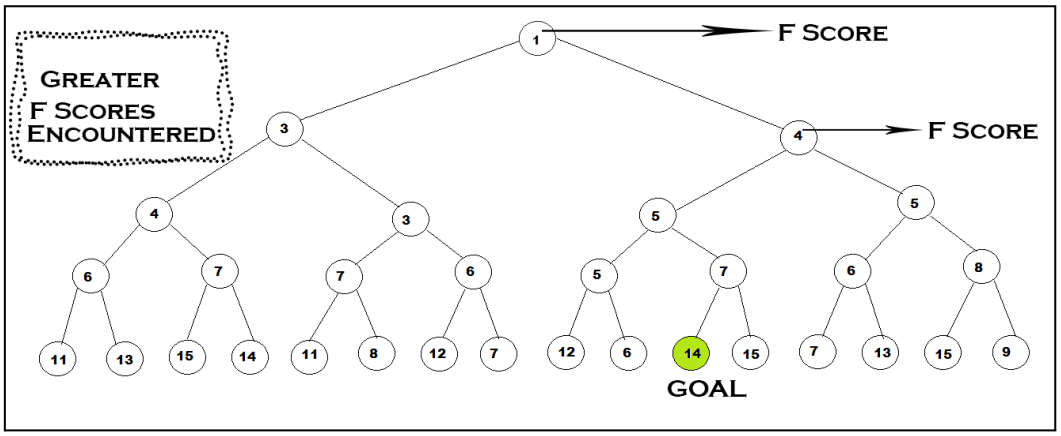
Gambar 4. Animasi Pencarian Algoritma A*

(Sumber: https://en.m.wikipedia.org/wiki/A*_search_algorithm)

A* (*A-Star*) adalah algoritma pencarian yang menggabungkan kelebihan dari *Uniform Cost Search* dan *Greedy Best-First Search* dengan menggunakan fungsi evaluasi gabungan, yaitu $f(n) = g(n) + h(n)$. Di mana $g(n)$ adalah biaya dari titik awal ke simpul saat ini, dan $h(n)$ adalah estimasi biaya dari simpul tersebut ke tujuan akhir berdasarkan fungsi heuristik. Dalam permainan Rush Hour, A* mempertimbangkan baik jumlah langkah yang sudah dilakukan maupun seberapa dekat posisi mobil utama dengan pintu keluar, sehingga pencarian solusi menjadi lebih terarah namun tetap optimal.

Kelebihan utama A* adalah kemampuannya menemukan solusi yang tidak hanya cepat, tetapi juga optimal selama heuristik yang digunakan bersifat *admissible* (tidak melebih-lebihkan jarak ke tujuan). Dalam kasus Rush Hour, ini berarti A* dapat menemukan jalur dengan jumlah langkah minimum sambil tetap menghindari eksplorasi konfigurasi papan yang tidak menjanjikan. Meskipun A* umumnya lebih efisien daripada UCS karena menggunakan heuristik sebagai panduan, performanya tetap sangat bergantung pada kualitas heuristik yang digunakan. Jika heuristik dirancang dengan baik, A* menjadi salah satu algoritma paling efektif untuk menyelesaikan *puzzle* seperti Rush Hour.

1.5. *Iterative Deepening A-Star* (IDA*)



Gambar 5. Animasi Pencarian Algoritma IDA*

(Sumber: https://algorithmsinsight.wordpress.com/graph-theory-2/ida-star-algorithm-in-general/)

Iterative Deepening A-Star (IDA*) adalah algoritma pencarian yang menggabungkan prinsip dari A* dan *Iterative Deepening Depth-First Search* (IDDFS). IDA* menggunakan pendekatan pencarian berbasis kedalaman dengan batasan biaya (*cost threshold*) yang diturunkan dari fungsi evaluasi A*, yaitu $f(n) = g(n) + h(n)$. Pada setiap iterasi, IDA* melakukan pencarian secara mendalam hingga batas $f(n)$ tertentu, dan jika tidak menemukan solusi, batas ini akan dinaikkan secara bertahap berdasarkan nilai terkecil yang melebihi ambang sebelumnya. Dalam konteks permainan Rush Hour, IDA* mencoba menjelajahi solusi dengan lebih sedikit konsumsi memori dibandingkan A*, karena tidak menyimpan seluruh node dalam antrian prioritas.

Kelebihan IDA* terletak pada efisiensi memorinya yang sangat rendah, menjadikannya cocok untuk masalah dengan ruang status besar seperti Rush Hour yang kompleks. Meskipun bisa lebih lambat dibandingkan A* karena mengulangi pencarian pada iterasi sebelumnya, IDA* tetap menjamin optimalitas solusi jika heuristik yang digunakan bersifat admissible. Dalam aplikasi pada Rush Hour, IDA* mampu menyeimbangkan antara kebutuhan memori dan kualitas solusi, menjadikannya alternatif menarik untuk menangani konfigurasi *puzzle* yang memerlukan banyak langkah atau memiliki banyak kendaraan penghalang.

BAB II DESKRIPSI DAN IMPLEMENTASI ALGORITMA

2.1. Permainan Rush Hour

Dalam representasi komputer untuk masalah *puzzle* Rush Hour, class `State` berperan sebagai model yang menggambarkan setiap konfigurasi papan permainan pada suatu titik waktu tertentu. Setiap objek `State` menyimpan informasi lengkap mengenai posisi mobil-mobil yang ada, dinding, serta posisi pintu keluar pada papan dengan ukuran tertentu. Dengan mengorganisasi kondisi permainan secara hierarkis—dari keadaan awal (*root state*) hingga keadaan turunan yang merepresentasikan perubahan posisi mobil—*class* ini memungkinkan simulasi dan eksplorasi langkah-langkah solusi secara efisien. Selain itu, dengan menggunakan teknik pencatatan perbedaan posisi antar-*state* dan mekanisme *hashing* untuk mempercepat pengecekan kondisi yang telah pernah dijelajahi, representasi ini mendukung algoritma pencarian dalam menemukan urutan langkah yang membawa mobil utama ke pintu keluar, sehingga menyelesaikan *puzzle* Rush Hour.

```
class State:
    // Fields (private)
    parent           // Parent State (nullable)
    depth            // Depth in search tree
    width, height    // Board dimensions
    isDifference     // Flag: true if this state is a difference from parent
    allCars          // List of all cars
    carIds           // IDs of cars moved in this state
    carPositions     // Positions of cars (encoded)
    primaryCarPosition // Position of primary car (goal car)
    walls            // Positions of walls on the board
    exitPosition     // Position of exit on the board
    fields           // Array representing the board fields (car IDs or empty/wall)
    hashCode         // Cached hash code for this state
    cachedComputedCarPositions  // Cache computed car positions
    cachedStepDescription       // Cache of step description string
    cachedMoveDescription       // Cache of move description string

    // Static factory methods
    // Create root state with full board setup
    static new_root(width, height, allCars, carPositions, walls, exitPosition)

    // Create new state from parent by updating positions of given cars
    static new_isDifference(parent, carIds, carPositions)

    // Create new state by applying steps (car moves) from a string
    static new_fromSteps(parent, stepsString)

    // Return full array of car positions, combining parent's positions and differences
    getComputedCarPositions()

    // Return a string describing the moves from root to current state
    getStepDescription()

    // Return a human-readable description of the last move
    getMoveDescription()

    // Check if car can be placed in given position/direction without collisions
    canFillField(direction, position, size)

    // Return list of valid move steps (positive and negative) for a car
    getCarMoveOptions(direction, position, size)
```

```
    // Check if primary car occupies the exit position
    isSolved()

    // Return cached hash code of the state
    hashCode()

    // Return string representation of board for debugging
    toString()

    // Internal helper methods like #fillField(), #calculateHashCode(), etc.```
```

- Field internal menyimpan dimensi papan, posisi mobil, posisi dinding, posisi pintu keluar, serta representasi grid papan secara keseluruhan.
- Ada konsep *state root* sebagai kondisi awal lengkap, dan *state difference* yang hanya menyimpan perubahan posisi mobil relatif ke *parent state* untuk menghemat memori dan mempercepat perhitungan.
- Metode `canFillField` memeriksa apakah sebuah mobil bisa ditempatkan pada posisi tertentu tanpa bertabrakan.
- Metode `getCarMoveOptions` menghasilkan semua langkah valid yang dapat diambil sebuah mobil pada kondisi saat ini.
- Fungsi `isSolved` memeriksa apakah mobil utama sudah sampai di pintu keluar, menandai *puzzle* selesai.
- Kode juga menggunakan `hashCode` untuk mempercepat pencarian dan pengecekan status yang sudah pernah dijelajahi.
- Deskripsi langkah dan gerakan disediakan untuk membantu memvisualisasikan solusi langkah demi langkah.

2.2. Algoritma *Uniform Cost Search*

```
function UCS(start, goal):
    frontier ← priority queue ordered by path_cost
    frontier.push(start, path_cost=0)
    explored ← empty set

    while frontier is not empty:
        node ← frontier.pop()
        if node.state == goal:
            return solution(node)

        explored.add(node.state)
        for each child in expand(node):
            if child.state not in explored and child not in frontier:
                frontier.push(child, path_cost=child.g)
            else if child in frontier with higher path_cost:
                frontier.update(child, new lower path_cost)
    return failure
```

Algoritma *Uniform Cost Search* (UCS) bekerja dengan mengeksplorasi simpul berdasarkan biaya total terkecil dari titik awal ke simpul tersebut. Dalam pseudocode di atas,

UCS memulai dengan menempatkan simpul awal ke dalam *frontier* (antrian prioritas) dengan biaya 0. Pada setiap iterasi, simpul dengan biaya terkecil diambil dari antrian, dan jika simpul tersebut adalah tujuan, maka solusi ditemukan. Jika belum, simpul tersebut diekspansi dan semua anaknya diperiksa. Jika anak simpul belum pernah dieksplorasi dan tidak ada dalam *frontier*, maka ia ditambahkan. Namun, jika anak tersebut sudah ada dalam *frontier* tetapi memiliki jalur baru dengan biaya lebih rendah, maka nilai biayanya diperbarui. Proses ini berlanjut sampai solusi ditemukan atau semua kemungkinan telah dicoba. UCS menjamin solusi optimal selama biaya langkah antar simpul bersifat non-negatif.

2.3. Algoritma *Greedy Best First Search*

```
function GBFS(start, goal):
    frontier ← priority queue ordered by heuristic h(n)
    frontier.push(start)
    explored ← empty set

    while frontier is not empty:
        node ← frontier.pop()
        if node.state == goal:
            return solution(node)

        explored.add(node.state)
        for each child in expand(node):
            if child.state not in explored and child not in frontier:
                frontier.push(child)
    return failure
```

Greedy Best-First Search (GBFS) adalah algoritma pencarian yang berfokus pada mempercepat pencarian solusi dengan menggunakan fungsi heuristik $h(n)$ untuk memperkirakan seberapa dekat suatu simpul dengan tujuan. Dalam pseudocode di atas, simpul awal dimasukkan ke dalam frontier yang diatur sebagai antrian prioritas berdasarkan nilai heuristiknya. Pada setiap langkah, simpul dengan nilai $h(n)$ terendah diambil untuk dievaluasi. Jika simpul tersebut adalah tujuan, maka solusi dikembalikan. Jika tidak, simpul diekspansi dan semua anak yang belum dieksplorasi maupun belum ada di *frontier* ditambahkan ke dalam *frontier*. Karena hanya mempertimbangkan heuristik tanpa memperhitungkan jalur yang telah ditempuh, GBFS dapat bekerja lebih cepat, namun tidak menjamin solusi yang ditemukan adalah yang paling optimal.

2.4. Algoritma *A-Star*

```
function A*(start, goal):
    frontier ← priority queue ordered by f(n) = g(n) + h(n)
    start.g ← 0
    frontier.push(start, f(start))
    explored ← empty set

    while frontier is not empty:
        node ← frontier.pop()
```

```
        if node.state == goal:
            return solution(node)

        explored.add(node.state)
        for each child in expand(node):
            child.g ← node.g + cost(node, child)
            if child.state not in explored and child not in frontier:
                frontier.push(child, f(child) = child.g + h(child))
            else if child in frontier with higher f(n):
                frontier.update(child, f(child))
    return failure
```

A* adalah algoritma pencarian yang menggabungkan dua elemen penting: biaya dari awal ke simpul saat ini ($g(n)$) dan estimasi biaya dari simpul tersebut ke tujuan menggunakan heuristik ($h(n)$). Dalam pseudocode di atas, simpul awal dimasukkan ke dalam *frontier* dengan nilai $f(n)$ awal. Di setiap iterasi, simpul dengan nilai $f(n)$ terkecil diambil dari antrian prioritas. Jika simpul tersebut merupakan tujuan, maka solusi dikembalikan. Jika belum, simpul tersebut diekspansi, dan anak-anaknya dihitung nilai $g(n)$-nya. Jika anak belum pernah dieksplorasi atau ada jalur baru dengan nilai $f(n)$ yang lebih baik, simpul anak dimasukkan atau diperbarui dalam *frontier*. Karena A* mempertimbangkan jalur yang telah ditempuh dan estimasi menuju tujuan, ia mampu menemukan solusi optimal dengan lebih efisien dibandingkan UCS, selama heuristik yang digunakan bersifat admissible.

2.5. Algoritma *Iterative Deepening A-Star*

```
function IDA*(start, goal):
    threshold ← f(start) = g(start) + h(start)

    while True:
        temp ← search(start, 0, threshold)
        if temp == FOUND:
            return solution
        if temp == ∞:
            return failure
        threshold ← temp

function search(node, g, threshold):
    f ← g + h(node)
    if f > threshold:
        return f
    if node.state == goal:
        return FOUND
    min ← ∞
    for each child in expand(node):
        temp ← search(child, g + cost(node, child), threshold)
        if temp == FOUND:
            return FOUND
        if temp < min:
            min ← temp
    return min
```

IDA* (*Iterative Deepening A-Star*) merupakan algoritma pencarian yang memadukan prinsip dari A* dan teknik *iterative deepening*. Daripada menyimpan semua simpul dalam memori seperti A*, IDA* melakukan pencarian dengan batas threshold berdasarkan fungsi evaluasi $f(n) = g(n) + h(n)$, yang merupakan gabungan dari biaya aktual dan estimasi heuristik menuju tujuan. Pencarian dilakukan secara rekursif dan mendalam, dan jika dalam satu iterasi tidak ditemukan solusi, batas threshold akan diperbarui ke nilai $f(n)$ terkecil yang melebihi ambang sebelumnya. Proses ini diulang hingga solusi ditemukan atau semua kemungkinan telah dieksplorasi. Kelebihan utama IDA* adalah efisiensi memorinya, karena hanya menyimpan jejak jalur saat ini, menjadikannya cocok untuk masalah besar seperti Rush Hour meskipun berisiko melakukan eksplorasi berulang.

2.6. *Branching Factor*

```
function estimate_branching_factor(N, d):
    if d == 0:
        return 0

    low ← 1.0
    high ← N
    epsilon ← 0.0001

    while (high - low) > epsilon:
        mid ← (low + high) / 2
        total ← geometric_sum(mid, d)

        if total < N:
            low ← mid
        else:
            high ← mid

    return (low + high) / 2

function geometric_sum(b, d):
    return (b^(d + 1) - 1) / (b - 1)
```

Branching factor adalah jumlah rata-rata cabang atau kemungkinan tindakan yang dapat diambil dari setiap simpul dalam pohon pencarian. Dalam konteks algoritma pencarian seperti UCS, GBFS, A*, dan IDA*, *branching factor* berperan penting dalam menentukan kompleksitas waktu dan ruang pencarian. Semakin besar *branching factor*, semakin banyak simpul yang perlu dieksplorasi oleh algoritma, yang berarti waktu pencarian akan meningkat secara eksponensial terhadap kedalaman solusi. Dalam permainan Rush Hour, *branching factor* ditentukan oleh jumlah kendaraan yang dapat bergerak dan jumlah arah gerak yang tersedia (maju atau mundur), tergantung pada posisi kendaraan lain di papan.

Memahami *branching factor* membantu dalam memperkirakan seberapa besar ruang pencarian dan seberapa efisien suatu algoritma dapat menjelajahi solusi. Misalnya, jika suatu konfigurasi Rush Hour memiliki banyak kendaraan dengan ruang gerak bebas, maka

branching factor-nya akan tinggi dan pencarian akan menjadi lebih sulit serta lambat. Oleh karena itu, algoritma seperti A* dan GBFS yang menggunakan heuristik lebih efektif dalam membatasi eksplorasi cabang yang tidak relevan. Dengan menganalisis *branching factor*, maka dapat memilih atau merancang algoritma yang paling sesuai untuk menyelesaikan suatu masalah dengan mempertimbangkan keseimbangan antara kecepatan, memori, dan optimalitas solusi.

2.7. Heuristik *Car Distance*

```
function heuristicCarDistance(oldBound, state):
    if state.isSolved():
        return oldBound

    width = state.getWidth()
    primaryCar = state.getAllCars()[0]
    primaryCarPosition = state.getPrimaryCarPosition()
    primaryCarPositionX = primaryCarPosition mod width
    primaryCarPositionY = primaryCarPosition div width
    exitPosition = state.getExitPosition()
    exitPositionX = exitPosition mod width
    exitPositionY = exitPosition div width

    if primaryCar.direction == HORIZONTAL:
        if primaryCarPositionX > exitPositionX:
            return oldBound + (primaryCarPositionX - exitPositionX)
        else:
            return oldBound + (exitPositionX - primaryCarPositionX -
primaryCar.size)

    if primaryCar.direction == VERTICAL:
        if primaryCarPositionY > exitPositionY:
            return oldBound + (primaryCarPositionY - exitPositionY)
        else:
            return oldBound + (exitPositionY - primaryCarPositionY -
primaryCar.size)

    return Infinity
```

Heuristik `heuristicCarDistance` menghitung estimasi jarak minimal yang harus ditempuh mobil utama untuk mencapai posisi pintu keluar. Dalam konteks pencarian jalur seperti A*, nilai fungsi $f(n)$ didefinisikan sebagai jumlah dari $g(n)$ dan $h(n)$, di mana $g(n)$ adalah biaya aktual yang sudah dikeluarkan untuk mencapai *state* n (dalam kode ini direpresentasikan oleh `oldBound`), sedangkan $h(n)$ adalah estimasi biaya sisa dari n menuju solusi. Heuristik ini mengambil jarak horizontal atau vertikal antara posisi depan mobil utama dan pintu keluar, dikurangi ukuran mobil, sehingga memberikan perkiraan yang konsisten dan admissible untuk mempercepat pencarian solusi *puzzle* Rush Hour.

2.8. Heuristik *Car Blocked*

```
function heuristicCarBlocked(oldBound, state):
    if state.isSolved():
        return oldBound

    width = state.getWidth()
    height = state.getHeight()
    fields = state.getFields()
    primaryCar = state.getAllCars()[0]
    primaryCarPosition = state.getPrimaryCarPosition()
    primaryCarPositionX = primaryCarPosition mod width
    primaryCarPositionY = primaryCarPosition div width
    exitPosition = state.getExitPosition()
    exitPositionX = exitPosition mod width
    exitPositionY = exitPosition div width

    carBlocked = 0

    if primaryCar.direction == HORIZONTAL:
        if primaryCarPositionX > exitPositionX:
            for i from primaryCarPositionX - 1 down to 0:
                index = primaryCarPositionY * width + i
                if fields[index] != FIELD_EMPTY:
                    carBlocked += 1
        else:
            for i from primaryCarPositionX + primaryCar.size up to
width - 1:
                index = primaryCarPositionY * width + i
                if fields[index] != FIELD_EMPTY:
                    carBlocked += 1

    if primaryCar.direction == VERTICAL:
        if primaryCarPositionY > exitPositionY:
            for i from primaryCarPositionY - 1 down to 0:
                index = i * width + primaryCarPositionX
                if fields[index] != FIELD_EMPTY:
                    carBlocked += 1
        else:
            for i from primaryCarPositionY + primaryCar.size up to
height - 1:
                index = i * width + primaryCarPositionX
                if fields[index] != FIELD_EMPTY:
                    carBlocked += 1

    return oldBound + carBlocked
```

Heuristik `heuristicCarBlocked` memperkirakan kesulitan mencapai solusi dengan menghitung jumlah mobil atau objek lain yang menghalangi jalur mobil utama menuju pintu keluar. Dalam konteks pencarian seperti algoritma A*, fungsi $f(n)$ adalah total estimasi biaya untuk mencapai solusi dari node n, yang merupakan penjumlahan dari $g(n)$ dan $h(n)$. $g(n)$ merepresentasikan biaya aktual yang sudah dikeluarkan untuk mencapai *state* saat ini (di sini `oldBound`), sedangkan $h(n)$ adalah estimasi tambahan biaya yang diperlukan agar mobil

utama tidak terhalang kendaraan lain. Dengan memasukkan jumlah mobil yang menghalangi jalur sebagai bagian dari heuristik, algoritma dapat lebih efektif memprioritaskan jalur yang lebih mungkin menuju solusi optimal.

2.9. Heuristik *Car Blocked Recursive*

Heuristik `heuristicCarBlockedRecursive` adalah pendekatan estimasi yang bertujuan untuk menentukan seberapa sulit mencapai solusi dalam permainan Rush Hour, dengan menghitung jumlah mobil yang menghalangi jalur mobil utama menuju pintu keluar. Dalam konteks algoritma pencarian seperti A*, fungsi $f(n) = g(n) + h(n)$, di mana $g(n)$ adalah biaya aktual yang telah dikeluarkan sejauh ini (`oldBound`), dan $h(n)$ adalah estimasi biaya ke depan, yang dalam hal ini didasarkan pada jumlah mobil penghalang. Semakin banyak mobil yang menghalangi, semakin besar estimasi biaya tersebut.

Versi dasar heuristik ini hanya menghitung mobil yang secara langsung menghalangi mobil utama. Namun, sebuah pendekatan lanjutan mengembangkan ide ini dengan menghitung juga mobil-mobil yang menghalangi mobil-mobil penghalang, dan seterusnya, secara rekursif. Pendekatan ini memberikan estimasi yang lebih akurat terhadap kompleksitas *state* saat ini, namun dengan trade-off berupa waktu komputasi yang lebih tinggi. Tantangan dalam pendekatan ini termasuk penghitungan ruang yang diperlukan untuk memindahkan mobil, pengaruh tembok sebagai penghalang, dan mencegah perhitungan ganda terhadap mobil yang sama.

Implementasi lengkap pendekatan heuristik lanjutan ini dapat ditemukan dalam repositori proyek Rush Hour di GitHub: https://github.com/saschazar21/rushhour. Pendekatan tersebut ditulis dalam Java dan tersedia di direktori `Heuristics/AdvancedHeuristic.java`.

BAB III SEKILAS KODE PROGRAM

3.1. Struktur Program

Berikut merupakan struktur file dari program:

```
├── .gitignore
├── .vscode/
│   └── launch.json
├── index.mjs                 # Entrypoint backend server
├── index.route-esm.mjs       # Router caching file ESM frontend
├── logic.mjs                 # Implementasi utama algoritma puzzle
├── package-lock.json
├── package.json
├── postcss.config.mjs
├── protocols.mjs             # Definisi protokol komunikasi worker
├── docs/                     # Direktori dokumentasi
│   ├── Spesifikasi Tugas Kecil 3 Stima 2024_2025.pdf
│   └── Tucil3_13523045.pdf   # Dokumen laporan
├── public/                   # Direktori asset statis
│   ├── BoardView.mjs         # Komponen merender papan
│   ├── CarBlock.mjs          # Komponen visualisasi mobil
│   ├── assets/
│   │   ├── car-1.png         # Gambar mobil untuk visualisasi
│   │   ├── car-2.png
│   │   ├── car-3.png
│   │   ├── car-4.png
│   │   ├── car-5.png
│   │   ├── car-6.png
│   │   ├── exit.png          # Gambar pintu keluar
│   │   └── wall.png          # Gambar dinding
│   ├── index.mjs             # Entry point frontend
│   ├── main-template.css     # Template stylesheet dasar
│   ├── main.css              # Stylesheet aplikasi frontend
│   ├── protocols.mjs         # Definisi protokol komunikasi
│   ├── shared.mjs            # Utilitas frontend
│   └── worker.mjs            # RPC frontend via websocket
├── test/                     # Direktori pengujian
│   ├── perftest.mjs          # Pengujian performa algoritma
│   ├── record-format.txt     # Format data recording puzzle
│   ├── test-worker.mjs       # Entry point test worker
│   ├── test.mjs              # Pengujian umum aplikasi
│   └── test_output.log       # Hasil keluaran pengujian otomatis
├── views/                    # Template EJS HTML
│   ├── basic-head.ejs        # Template <head> HTML umum
│   └── index.ejs             # Template utama halaman
└── worker.mjs                # Entry point worker
```

3.2. Implementasi Program

logic.mjs cd35cffe86fe00fe2c1220c1814e0a6ca41bed66

```
export class QueueSolver {
      ... snip ...
      tick() {
            if(this.#solution != null || this.#stateQueue.length == 0)
                  return false;
            /** @type {[number, State]} */
            const [oldBound, state] = this.#stateQueue.shift();
            if(state.isSolved()) {
                  this.#solution = state;
                  return false;
            }
            /** @type {[number, State][]} */
            const toAddStateQueue = [];
            const width = state.getWidth();
            const allCars = state.getAllCars();
            const fields = state.getFields();
            // Optimisation: Do not compute car positions. Iterating the fields and
storing the visited car is faster.
            // Optimisation: Check the visited cars by bit fields because `Set` is
slower. This puts restriction to the max number of cars to 52.
            let visitedCars = 0;
            for(let i = 0; i < fields.length; i++) {
                  const fieldValue = fields[i];
                  if(fieldValue == FIELD_EMPTY || fieldValue == FIELD_WALL)
continue;
                  const carId = fieldValue - 1;
                  if(visitedCars & (1 << carId)) continue;
                  visitedCars |= (1 << carId);
                  const car = allCars[carId];
                  const x = i % width;
                  const y = Math.floor(i / width);
                  const moves = state.getCarMoveOptions(car.direction, i, car.size);
                  if(moves.length == 0) continue;
                  this.#searchCount += moves.length;
                  for(const move of moves) {
                        const newPosition = car.direction == HORIZONTAL ? y * width
+ (x + move) : (y + move) * width + x;
                        const expandedState = State.new_isDifference(state,
[car.id], ((((i + 1) << 16) | (newPosition & 0xFFFF)) >>> 0]);
                        const hashCode = expandedState.hashCode();
                        if(this.#closedHashes.has(hashCode)) continue;
                        this.#closedHashes.add(hashCode);
                        toAddStateQueue.push([this.#heuristicCalculator(oldBound,
expandedState), expandedState]);
                  }
            }
            // Optimisation: Sort the newly added states, and merge sort to the
queue.
```

```
            // The premise is, the newly added states will always be smaller than
the existing states.
            // Because inserting to queue first and sort the queue, will
unnecessarily compare sorted entries.
            toAddStateQueue.sort((a, b) => a[0] - b[0]);
            QueueSolver.#mergeSortedArraysInPlace(this.#stateQueue,
toAddStateQueue, (a, b) => a[0] - b[0]);
            return this.#stateQueue.length != 0;
      }
      ... snip ...
}
export class StackSolver {
      ... snip ...
      tick() {
            if(this.#solution != null || this.#bound == Infinity)
                  return false;
            const initialStateHashCode = this.#initialState.hashCode();
            const initialStateBound = this.#heuristicCalculator(0,
this.#initialState);
            this.#closedHashes.add(initialStateHashCode);
            const newBound = this.#search(this.#initialState, initialStateBound);
            this.#closedHashes.delete(initialStateHashCode);
            if(newBound instanceof State) {
                  this.#solution = newBound;
                  return false;
            }
            this.#bound = newBound;
            if(newBound == Infinity)
                  return false;
            return true;
      }
      ... snip ...
}
... snip ...

/** @type {HeuristicCalculator} */
export const heuristicDepth = (_, state) => state.getDepth();
/** @type {HeuristicCalculator} */
export const heuristicConstant = () => 0;
/** @type {HeuristicCalculator} */
export const heuristicCarDistance = (_, state) => {
      if(state.isSolved()) return 0;
      const width = state.getWidth();
      const primaryCar = state.getAllCars()[0];
      const primaryCarPosition = state.getPrimaryCarPosition();
      const primaryCarPositionX = primaryCarPosition % width;
      const primaryCarPositionY = Math.floor(primaryCarPosition / width);
      const exitPosition = state.getExitPosition();
      const exitPositionX = exitPosition % width;
```

```
      const exitPositionY = Math.floor(exitPosition / width);
      if(primaryCar.direction == HORIZONTAL) {
            if(primaryCarPositionX > exitPositionX)
                  return primaryCarPositionX - exitPositionX;
            else
                  return exitPositionX - primaryCarPositionX - primaryCar.size;
      }
      if(primaryCar.direction == VERTICAL) {
            if(primaryCarPositionY > exitPositionY)
                  return primaryCarPositionY - exitPositionY;
            else
                  return exitPositionY - primaryCarPositionY - primaryCar.size;
      }
      return Infinity;
};
/** @type {HeuristicCalculator} */
export const heuristicCarBlocked = (_, state) => {
      if(state.isSolved()) return 0;
      const width = state.getWidth();
      const height = state.getHeight();
      const fields = state.getFields();
      const primaryCar = state.getAllCars()[0];
      const primaryCarPosition = state.getPrimaryCarPosition();
      const primaryCarPositionX = primaryCarPosition % width;
      const primaryCarPositionY = Math.floor(primaryCarPosition / width);
      const exitPosition = state.getExitPosition();
      const exitPositionX = exitPosition % width;
      const exitPositionY = Math.floor(exitPosition / width);
      let carBlocked = 0;
      if(primaryCar.direction == HORIZONTAL) {
            if(primaryCarPositionX > exitPositionX) {
                  for(let i = primaryCarPositionX - 1; i >= 0; i--) {
                        const index = primaryCarPositionY * width + i;
                        if(fields[index] == FIELD_EMPTY) continue;
                        carBlocked++;
                  }
            } else {
                  for(let i = primaryCarPositionX + primaryCar.size; i < width; i++)
{
                        const index = primaryCarPositionY * width + i;
                        if(fields[index] == FIELD_EMPTY) continue;
                        carBlocked++;
                  }
            }
      }
      if(primaryCar.direction == VERTICAL) {
            if(primaryCarPositionY > exitPositionY) {
                  for(let i = primaryCarPositionY - 1; i >= 0; i--) {
                        const index = i * width + primaryCarPositionX;
```

```
                        if(fields[index] == FIELD_EMPTY) continue;
                        carBlocked++;
                    }
                } else {
                    for(let i = primaryCarPositionY + primaryCar.size; i < height;
i++) {
                        const index = i * width + primaryCarPositionX;
                        if(fields[index] == FIELD_EMPTY) continue;
                        carBlocked++;
                    }
                }
            }
            return carBlocked;
};
... snip ...

/**
 * @typedef {(oldBound: number, state: State) ⇒ number} HeuristicCalculator
 */

/**
 * @typedef {Object} Car
 * @property {number} id
 * @property {number} direction
 * @property {number} size
 */

export const FIELD_EMPTY = 0;
export const FIELD_WALL = 255;
export const FIELD_PRIMARY_CAR = 1;
export const HORIZONTAL = 0;
export const VERTICAL = 1;

// `State` is effectively constant. You should cache the computation over `State`
whenever possible.
export class State {
    ... snip ...

    /** @type {State?} */
    #parent;
    /** @type {number} */
    #depth;
    /** @type {number} */
    #width; // Optimisation: Positions are encoded as (y * width + x). Applies
for carPositions, walls, and exitPosition.
    /** @type {number} */
    #height;
    /** @type {boolean} */
    #isDifference; // Optimisation: Store the difference instead of copying all
```

```
objects.
      /** @type {Car[]} */
      #allCars; // Optimisation: Avoid traversing up to the root node.
      /** @type {number[]} */
      #carIds;
      /** @type {number[]} */
      #carPositions;
      /** @type {number} */
      #primaryCarPosition; // Optimisation: Heuristic needs this. Instead of
searching the fields, just cache when constructing state.
      /** @type {number[]} */
      #walls;
      /** @type {number} */
      #exitPosition;
      /** @type {Uint8Array} */
      #fields;
      /** @type {number} */
      #hashCode;
      ... snip ...
}

/**
 * @param {string} inputText
 * @returns {{
 *   width: number,
 *   height: number,
 *   cars: Car[],
 *   carPositions: number[],
 *   walls: number[],
 *   exitPosition: number
 * }}
 */
export function parseBoardInput(inputText) {
      ... snip ...
}
```

3.3. Optimasi Program - *Iterating Cars for State Expansion*

Optimasi ini ditujukan untuk meningkatkan performa pencarian *state* baru dalam algoritma pemecahan *puzzle* dengan menghindari perhitungan posisi mobil secara eksplisit. Daripada menghitung posisi tiap mobil dari objek mobil itu sendiri, algoritma ini langsung memindai seluruh grid (`fields`) dan mencatat mobil yang telah diproses menggunakan *bit field*, yang jauh lebih cepat dibandingkan struktur data `Set`. Setiap mobil hanya diproses satu kali, dengan identifikasi menggunakan *bit shifting* (maksimum 52 mobil karena keterbatasan representasi integer di IEEE754). Pendekatan ini mengurangi overhead alokasi objek dan operasi pencarian, sekaligus mempercepat deteksi mobil dan perluasan *state* baru. Strategi ini

secara signifikan mengurangi waktu komputasi selama eksplorasi *state* pada *puzzle* yang kompleks.

logic.mjs cd35cffe86fe00fe2c1220c1814e0a6ca41bed66

```
/** @type {[number, State][]} */
const toAddStateQueue = [];
const width = state.getWidth();
const allCars = state.getAllCars();
const fields = state.getFields();
// Optimisation: Do not compute car positions. Iterating the
fields and storing the visited car is faster.
// Optimisation: Check the visited cars by bit fields because
`Set` is slower. This puts restriction to the max number of cars
to 52.
let visitedCars = 0;
for(let i = 0; i < fields.length; i++) {
      const fieldValue = fields[i];
      if(fieldValue == FIELD_EMPTY || fieldValue == FIELD_WALL)
continue;
      const carId = fieldValue - 1;
      if(visitedCars & (1 << carId)) continue;
      visitedCars |= (1 << carId);
      const car = allCars[carId];
      const x = i % width;
      const y = Math.floor(i / width);
      const moves = state.getCarMoveOptions(car.direction, i,
car.size);
      if(moves.length == 0) continue;
      this.#searchCount += moves.length;
      for(const move of moves) {
            const newPosition = car.direction == HORIZONTAL ? y
* width + (x + move) : (y + move) * width + x;
            const expandedState = State.new_isDifference(state,
[car.id], [(((i + 1) << 16) | (newPosition & 0xFFFF)) >>> 0]);
            const hashCode = expandedState.hashCode();
            if(this.#closedHashes.has(hashCode)) continue;
            this.#closedHashes.add(hashCode);

toAddStateQueue.push([this.#heuristicCalculator(oldBound,
expandedState), expandedState]);
      }
}
```

3.4. Optimasi Program - *Fast Queue Insertion*

Optimasi ini bertujuan mempercepat penyisipan *state* baru ke dalam antrian prioritas dengan memanfaatkan asumsi bahwa semua *state* baru (`toAddStateQueue`) jumlahnya mungkin lebih kecil dibandingkan *state* yang sudah ada. Daripada langsung memasukkan dan

menyortir seluruh antrian, *state* baru terlebih dahulu diurutkan, lalu digabungkan secara efisien ke antrean utama menggunakan *in-place merge sort*. Proses penggabungan ini dioptimalkan lebih lanjut dengan pencarian posisi *insert* menggunakan binary search, sehingga meminimalkan jumlah perbandingan yang diperlukan. Strategi ini mengurangi kompleksitas dari operasi *insert* dan menjaga antrian tetap terurut tanpa penalti performa dari penyortiran penuh setiap iterasi.

logic.mjs cd35cffe86fe00fe2c1220c1814e0a6ca41bed66

```
// Optimisation: Sort the newly added states, and merge sort to
the queue.
// The premise is, the newly added states will always be smaller
than the existing states.
// Because inserting to queue first and sort the queue, will
unnecessarily compare sorted entries.
toAddStateQueue.sort((a, b) ⇒ a[0] - b[0]);
QueueSolver.#mergeSortedArraysInPlace(this.#stateQueue,
toAddStateQueue, (a, b) ⇒ a[0] - b[0]);
return this.#stateQueue.length ≠ 0;
```

```
/**
 * @template T
 * @param {T[]} aArray
 * @param {T[]} bArray
 * @param {(a: T, b: T) ⇒ number} compareFn
 */
static #mergeSortedArraysInPlace(aArray, bArray, compareFn) {
	let aIndex = 0;
	for(let bIndex = 0; bIndex < bArray.length; bIndex++) {
		const item = bArray[bIndex];
		// Optimisation: Since aArray and bArray both
sorted, we can optimize
		// the insertion further by searching the aIndex
using binary search.
		let high = aArray.length;
		while(aIndex < high) {
			const mid = (aIndex + high) >> 1;
			if(compareFn(aArray[mid], item) ⩽ 0)
				aIndex = mid + 1;
			else
				high = mid;
		}
		aArray.splice(aIndex, 0, item);
		aIndex++;
	}
}
```

3.5. Optimasi Program - *Stack Solver (IDA*) Fasttrack Approximation*

Optimasi pada `StackSolverApprox` bertujuan mempercepat pencarian solusi tanpa mengorbankan optimalitas hasil secara signifikan. Berbeda dari pendekatan `StackSolver` yang menghindari siklus dengan memeriksa jejak (*path*) dalam tumpukan secara lokal, versi ini menggunakan pendekatan global melalui `closedHashes`, yaitu *dictionary hash* yang menyimpan bound terkecil untuk setiap *state* yang telah diekspansi. Suatu *state* hanya akan diproses kembali jika nilai bound barunya lebih kecil dari yang tercatat sebelumnya, memungkinkan eksplorasi ulang hanya bila menjanjikan hasil lebih baik. Teknik ini berpotensi mengurangi jumlah ekspansi tidak perlu dan menyimpan informasi antar iterasi untuk mempercepat tick berikutnya. Meskipun belum diverifikasi secara keseluruhan, pendekatan ini menunjukkan performa menjanjikan dalam beberapa kasus uji, dengan kemungkinan untuk dioptimalkan lebih lanjut melalui analisis karakteristik *state* yang sering menghasilkan perluasan tidak optimal.

logic.mjs cd35cffe86fe00fe2c1220c1814e0a6ca41bed66

```
// This solver is an optimization over StackSolver. It may be faster for some test cases,
// but it should outputs optimal solution like StackSolver (though unverified for all cases).
// The main difference is how both solvers skip the closedHashes. In the "proper" implementation,
// it behaves like a hash path, where it skips the node if it's in the current stack trace.
// In contrast, this approx solver tries to use "global" closedHashes, where it still compares
// if the state bound is less than the last bound within the same hash. If it is, it'll probably
// resulting in fewer steps. Though, optimization can be further analyzed, since it has new
// information every tick to speed up the next tick. For example, skipping the state that
// always result in worst expansion.
export class StackSolverApprox {
	... snip ...
	tick() {
		if(this.#finished)
			return false;
		const initialStateBound = this.#heuristicCalculator(0, this.#initialState);
		this.#closedHashes.clear();
		this.#closedHashes.set(this.#initialState.hashCode(), initialStateBound);
		const newBound = this.#search(this.#initialState, initialStateBound);
		if(newBound instanceof State) {
			this.#finished = true;
			this.#solution = newBound;
			return false;
		}
		if(newBound == Infinity) {
			this.#finished = true;
			return false;
		}
		this.#bound = newBound;
		return true;
	}
	/**
	 * @param {State} state
	 * @param {number} bound
	 * @returns {number | State}
	 */
	#search(state, bound) {
```

```
                if(bound > this.#bound)
                        return bound;
                if(state.isSolved())
                        return state;
                /** @type {State?} */
                let solvedState = null;
                let minBound = Infinity;
                const width = state.getWidth();
                const allCars = state.getAllCars();
                const fields = state.getFields();
                let visitedCars = 0;
                for(let i = 0; i < fields.length; i++) {
                        const fieldValue = fields[i];
                        if(fieldValue == FIELD_EMPTY || fieldValue == FIELD_WALL) continue;
                        const carId = fieldValue - 1;
                        if(visitedCars & (1 << carId)) continue;
                        visitedCars |= (1 << carId);
                        const car = allCars[carId];
                        const x = i % width;
                        const y = Math.floor(i / width);
                        const moves = state.getCarMoveOptions(car.direction, i, car.size);
                        if(moves.length == 0) continue;
                        this.#searchCount += moves.length;
                        for(const move of moves) {
                                const newPosition = car.direction == HORIZONTAL ? y * width + (x + move)
: (y + move) * width + x;
                                const expandedState = State.new_isDifference(state, [car.id], [(((i + 1)
<< 16) | (newPosition & 0xFFFF)) >>> 0]);
                                const expandedStateBound = this.#heuristicCalculator(bound,
expandedState);
                                const hashCode = expandedState.hashCode();
                                const closedHashBound = this.#closedHashes.get(hashCode);
                                if(closedHashBound != null && closedHashBound <= expandedStateBound)
continue;
                                this.#closedHashes.set(hashCode, expandedStateBound);
                                const result = this.#search(expandedState, expandedStateBound);
                                if(result instanceof State) {
                                        if(expandedStateBound < minBound) {
                                                solvedState = result;
                                                minBound = expandedStateBound;
                                        }
                                        continue;
                                }
                                if(result < minBound) {
                                        solvedState = null;
                                        minBound = result;
                                }
                        }
                }
                if(solvedState != null)
                        return solvedState;
                return minBound;
        }
}
```

3.6. Optimasi Program - *Data Structure, Caching, and Fulfilled Assumptions*

Optimasi ini berfokus pada efisiensi struktur data, *caching*, dan asumsi validitas untuk mempercepat proses pembentukan dan evaluasi *state* dalam algoritma pemecahan *puzzle*.

Representasi posisi (termasuk mobil, dinding, dan pintu keluar) dikompresi dalam bentuk `y * width + x`, mengurangi kebutuhan pemrosesan *array* dua dimensi. Untuk mempercepat pembuatan *state* turunan, pendekatan *difference-based* diterapkan, hanya menyimpan perbedaan terhadap *parent* daripada menyalin keseluruhan data. Informasi penting seperti posisi mobil utama langsung disimpan saat *state* dibuat, menghindari pencarian di *field*. Beberapa data turunan seperti posisi seluruh mobil, deskripsi langkah, dan deskripsi pergerakan dikalkulasi satu kali dan disimpan menggunakan *lazy caching*. Proses pengisian *field* tidak melakukan validasi ulang dengan asumsi bahwa parent valid dan langkah pergerakan telah diverifikasi sebelumnya. Selain itu, `hashCode` *state* dihitung secara efisien dengan teknik XOR dan perkalian sederhana, meskipun tidak sepenuhnya akurat, karena mengorbankan akurasi demi kecepatan, selama tidak ada mobil berukuran satu. Keseluruhan optimasi ini secara signifikan mengurangi overhead per iterasi, mempercepat traversal dan penyimpanan *state* dalam pencarian solusi.

logic.mjs cd35cffe86fe00fe2c1220c1814e0a6ca41bed66

```
/** @type {State?} */
#parent;
/** @type {number} */
#depth;
/** @type {number} */
#width; // Optimisation: Positions are encoded as (y * width +
x). Applies for carPositions, walls, and exitPosition.
/** @type {number} */
#height;
/** @type {boolean} */
#isDifference; // Optimisation: Store the difference instead of
copying all objects.
/** @type {Car[]} */
#allCars; // Optimisation: Avoid traversing up to the root node.
/** @type {number[]} */
#carIds;
/** @type {number[]} */
#carPositions;
/** @type {number} */
#primaryCarPosition; // Optimisation: Heuristic needs this.
Instead of searching the fields, just cache when constructing
state.
/** @type {number[]} */
#walls;
/** @type {number} */
#exitPosition;
/** @type {Uint8Array} */
#fields;
/** @type {number} */
```

```
#hashCode;
```

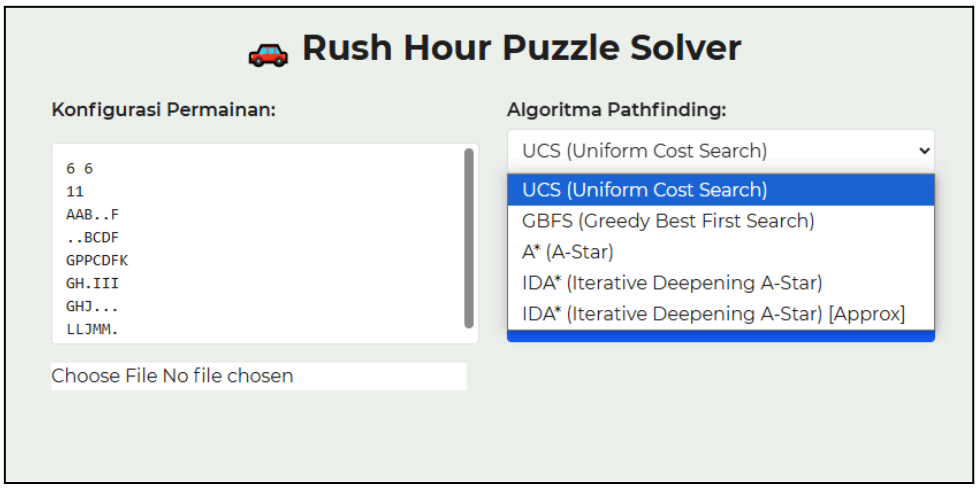
```
for(let i = 0; i < this.#carIds.length; i++) {
	const car = this.#allCars[this.#carIds[i]];
	const carPosition = this.#carPositions[i];
	const newPosition = carPosition & 0×FFFF;
	// Optimisation: Assume valid. Premise: Parent is valid and getCarMovePositions returns valid moves.
	// if(!this.canFillField(car.direction, newPosition, car.size)) {
	//	const x = newPosition % this.#width;
	//	const y = Math.floor(newPosition / this.#width);
	//	throw new Error(`Cannot place Car#${car.id} at position (${x}, ${y}) with direction ${car.direction} and size ${car.size}.\n${this.toString()}`);
	// }
	this.#fillField(car.direction, newPosition, car.size, car.id + 1);
	if(car.id == 0)
		this.#primaryCarPosition = newPosition;
}
```

```
#calculateHashCode() {
	let hash = 0×811c9dc5;
	// Optimisation: This hashCode is not 100% correct. But that's what we sacrifice to half the computation.
	// Optimisation: Compute the hashCode over checkered board pattern to reduce clashes.
	// This hash is safe as long as there's no one-sized car.
	// for(let y = 0; y < this.#height; y++) {
	//	for(let x = y % 2 == 0 ? 0 : 1; x < this.#width; x++) {
	//		const i = y * this.#width + x;
	//		hash ^= this.#fields[i];
	//		hash = (hash * 0×010193) >>> 0;
	//	}
	// }
	for(let i = 0; i < this.#fields.length; i++) {
		hash ^= this.#fields[i];
		hash = (hash * 0×1193) >>> 0;
	}
	this.#hashCode = hash;
}
```

```
/** @type {number[]} */
#__cachedComputedCarPositions = null; // Optimisation: Cache.
getComputedCarPositions() {
	if(this.#__cachedComputedCarPositions != null)
		return this.#__cachedComputedCarPositions;
	... snip ...
```

```
        return this.#__cachedComputedCarPositions = computedCars;
}
/** @type {string?} */
#__cachedStepDescription = null; // Optimisation: Cache.
getStepDescription() {
        if(this.#__cachedStepDescription != null)
                return this.#__cachedStepDescription;
        ... snip ...
        return this.#__cachedStepDescription = descriptions.join("
");
}
/** @type {string?} */
#__cachedMoveDescription = null; // Optimisation: Cache.
getMoveDescription() {
        if(this.#__cachedMoveDescription != null)
                return this.#__cachedMoveDescription;
        ... snip ...
        return this.#__cachedMoveDescription = descriptions.length
== 0 ? "No change in car positions" : descriptions.join("\n");
}
```

3.7. Bonus - Algoritma Alternatif

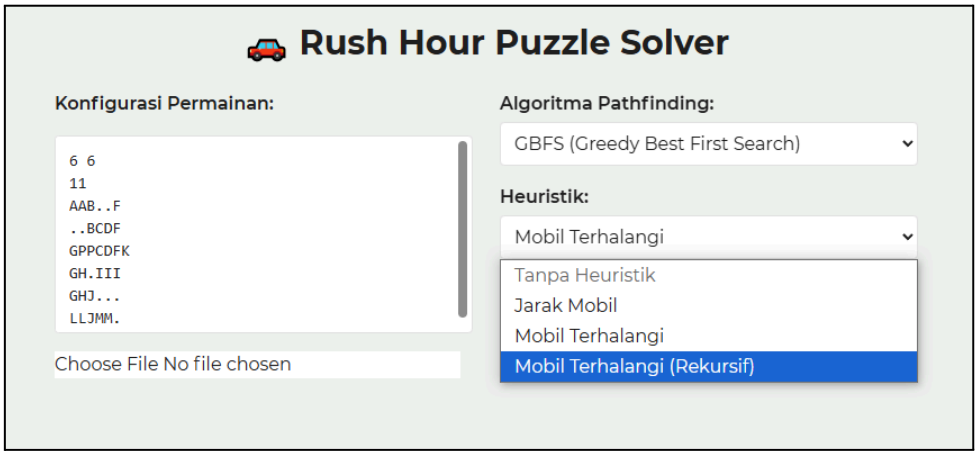


Gambar X. Macam-Macam Algoritma yang Ditawarkan

Sebagai bagian dari bonus, program ini mengimplementasikan dua varian algoritma pencarian: IDA* (*Iterative Deepening A-Star*) dan IDA* Approx. Keduanya dirancang untuk menemukan solusi optimal dengan memori yang jauh lebih rendah dibanding A*, dengan melakukan pencarian berbasis batas heuristik yang meningkat secara bertahap. IDA* menjaga akurasi dengan hanya melewati *node* yang berada dalam *stack trace* untuk menghindari siklus. Sementara itu, IDA* Approx memperkenalkan pendekatan heuristik yang lebih agresif dengan menggunakan `closedHashes` global, memungkinkan pengabaian *state* yang dianggap tidak efisien berdasarkan bound sebelumnya. Pendekatan ini tidak menjamin

optimalitas sempurna, namun seringkali memberikan hasil lebih cepat pada kasus tertentu dengan jumlah langkah eksplorasi yang lebih sedikit. Kedua varian ini menunjukkan eksplorasi arah optimasi yang lebih eksperimental namun berpotensi menjanjikan dalam konteks efisiensi.

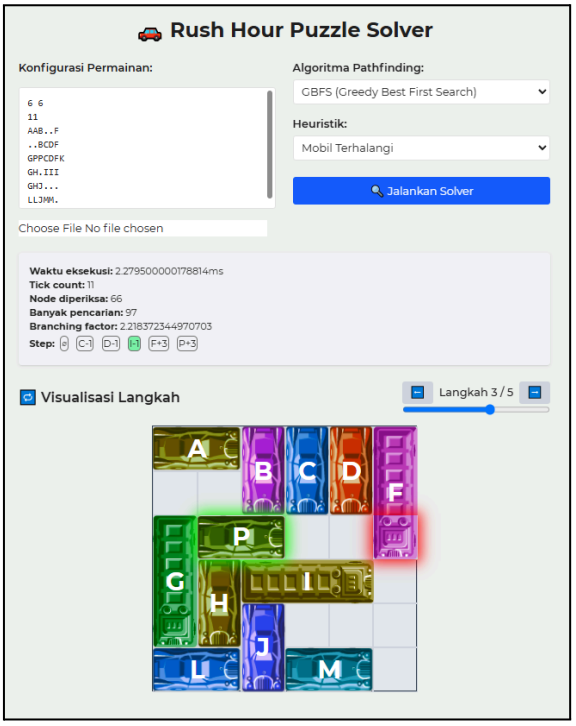
3.8. Bonus - Heuristic Alternatif



Gambar X. Macam-Macam Heuristik yang Ditawarkan

Sebagai fitur bonus, program ini menyediakan tiga alternatif fungsi heuristik untuk meningkatkan efisiensi pencarian solusi: *car distance*, *car blocked*, dan *car blocked recursive*. Heuristik *car distance* menghitung jarak dari mobil utama ke pintu keluar, memberikan estimasi dasar namun cepat. Heuristik *car blocked* menambahkan pertimbangan jumlah mobil yang menghalangi jalur mobil utama, sehingga lebih informatif dalam memperkirakan tingkat kesulitan. Sementara itu, *car blocked recursive* memperluas konsep ini dengan menghitung hambatan secara rekursif—yakni mempertimbangkan mobil yang menghalangi mobil penghalang—sehingga memberikan estimasi lebih akurat terhadap kompleksitas konfigurasi papan. Ketiga pendekatan ini menunjukkan fleksibilitas sistem dalam mengeksplorasi strategi evaluasi yang berbeda, dari yang ringan hingga kompleks, untuk menyesuaikan performa dan akurasi pencarian.

3.9. Bonus - GUI



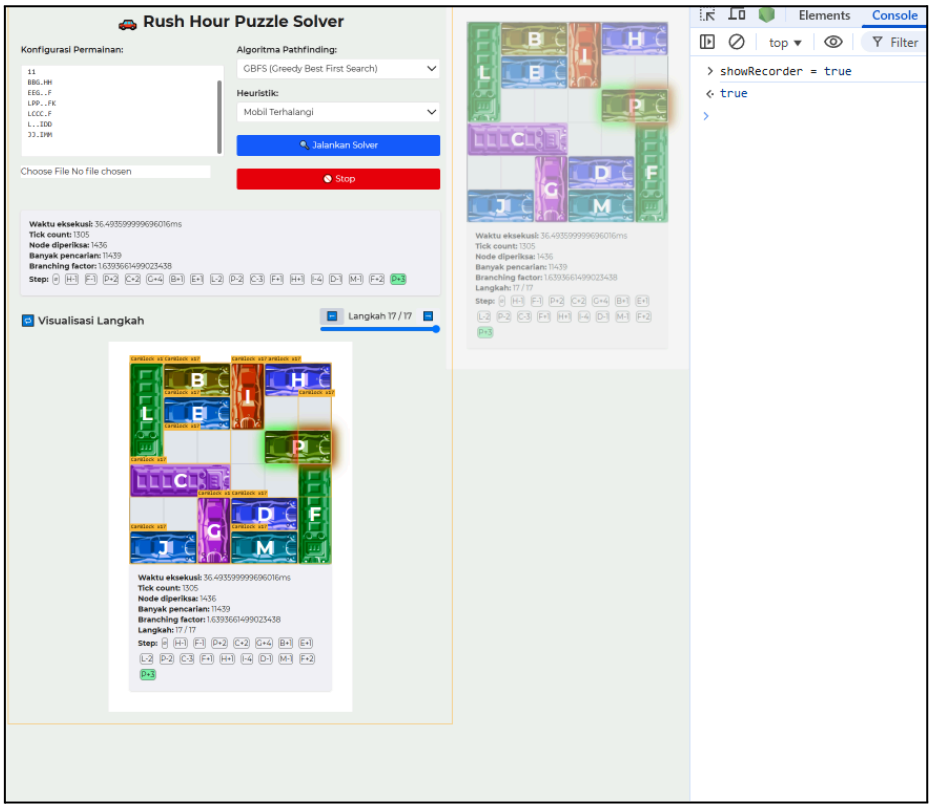
Gambar X. Tampilan GUI Program

Sebagai fitur bonus, program ini dilengkapi dengan antarmuka grafis (GUI) berbasis web yang memudahkan pengguna dalam berinteraksi dengan sistem. Pada halaman utama, pengguna dapat memasukkan *puzzle* dalam bentuk *string* melalui *form input*, atau memilih *file puzzle* langsung melalui tombol unggah. Selain itu, tersedia *selector* untuk memilih algoritma pencarian dan fungsi heuristik yang ingin digunakan, memberikan fleksibilitas dalam eksperimen dan evaluasi. Setelah tombol "Solve" ditekan dan solusi ditemukan, GUI akan menampilkan visualisasi papan permainan beserta animasi langkah-langkah penyelesaiannya, memungkinkan pengguna untuk mengikuti jalur solusi dengan jelas dan intuitif.

3.10. Bonus - *Deployment*

Sebagai bagian dari bonus, website ini telah di-*deploy* secara publik untuk dapat diakses oleh pengguna melalui domain pribadi di alamat https://rushhour.bwks.link. Proses *deployment* dilakukan pada sebuah VM server *on-premise* yang dikelola secara mandiri, menjalankan sistem operasi Windows Server 2022 Datacenter dengan spesifikasi 4 core CPU, 4 GB RAM, dan kapasitas penyimpanan sebesar 92 GB. Akses ke server diamankan dan di-rutekan menggunakan layanan Cloudflare Tunnel, dengan DNS yang juga dikelola melalui Cloudflare untuk memastikan kestabilan dan kemudahan pengelolaan domain. Domain yang digunakan merupakan milik pribadi, dan layanan ini direncanakan memiliki uptime selama kurang lebih tiga bulan sebagai periode demonstrasi dan pengujian publik.

3.11. Bonus - *Video Animation Output*



Gambar X. Merekam Animasi Solusi *Puzzle*

Sebagai fitur bonus tambahan, aplikasi ini menyediakan kemampuan untuk merekam animasi solusi dalam bentuk video. Pada tampilan antarmuka, terdapat tombol "Record" yang dapat muncul dengan mengaktifkan pengaturan `showRecorder = true` melalui *console inspector* browser. Setelah diaktifkan, tombol ini memungkinkan pengguna merekam animasi pergerakan solusi *puzzle* secara otomatis. Selama proses rekaman, animasi dijalankan dari awal hingga akhir tanpa perlu interaksi manual, dan hasil video disimpan ke file system. Fitur ini sangat berguna untuk menghasilkan dokumentasi visual solusi dari setiap *puzzle* secara massal dan terstruktur.

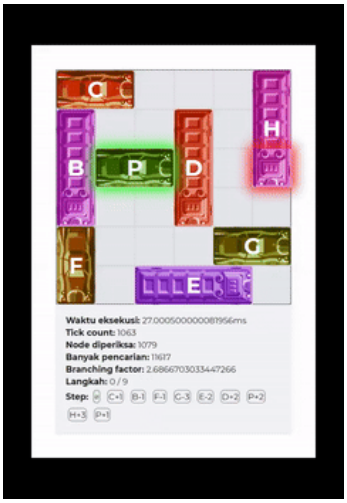
BAB IV PERCOBAAN PROGRAM

Semua gambar di bawah adalah animasi. Buka link docs untuk melihatnya secara langsung.

4.1. Jam-1

Nama	:	Jam-1	`6 6`
			`7`
			`CC...H`
			`B..D.H`
			`BPPD.HK`
			`B..D..`
			`F...GG`
			`F.EEE.`

Banyak Step Optimal	:	8	
Step Optimal	:	`ϕ C+1 B-1 F-1 G-3 H+3 E-2 D+2 P+3`	

UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

4.2. Jam-2

Nama	:	Jam-2	`6 6`
			`10`
			`B..FFF`
			`B..G.L`
			`PP.GHLK`
			`CCC.HL`
			`..E.II`
			`DDEJJ.`

Banyak Step Optimal	:	8	
Step Optimal	:	`ø P+1 B+1 F-3 G-1 H-2 I-1 L+2 P+3`	

UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

4.3. Jam-3

Nama	:	Jam-3	6 6
			5
			`......`
			`......`
			`.PPC..K`
			`.BBC.F`
			`.D.C.F`
			`.DEE.F`

Banyak Step Optimal	:	14	
Step Optimal	:	∅ C-2 F-3 B+3 D-1 E-1 C+3 P+1 D-3 P-1 C-3 B-3 C+3 F+3 P+3	

UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

4.4. Jam-4

Nama	:	Jam-4	6 6 6 D .. F .. D .. F .. DPPF .. K .. BCCC .. B .. E .. GGGE
Banyak Step Optimal	:	9	
Step Optimal	:	ø D+3 P-1 B-3 P+1 D-3 C-3 G-2 F+3 P+3	

UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	A* Car Blocked	A* Car Blocked Recursive
		

4.5. Jam-5

Nama	:	Jam-5	`6 6`
			`10`
			`BB.F.G`
			`L..FCG`
			`LPPFCJK`
			`LHHHCJ`
			`E...II`
			`E...DD`

Banyak Step Optimal	:	9	
Step Optimal	:	`ø B+1 L-1 H-1 I-3` `J+1 D-3 F+3 C+2 P+3`	

UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	A* Car Blocked	A* Car Blocked Recursive
		

4.6. Jam-6

Nama	:	Jam-6	`6 6`
			`10`
			`BB.G..`
			`HH.GFL`
			`.PPCFLK`
			`EEICFL`
			`D.IC..`
			`D..JJJ`

Banyak Step Optimal	:	9	
Step Optimal	:	`ϕ P-1 I-3 E+1 D-1` `J-3 F+2 L+2 C+1 P+4`	

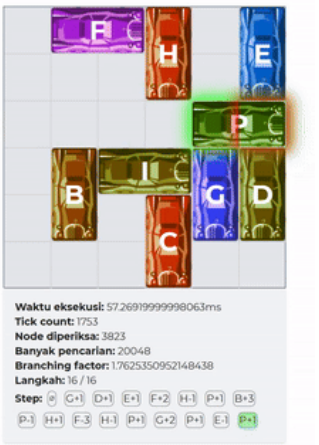
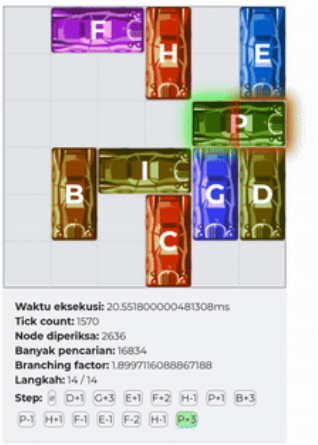
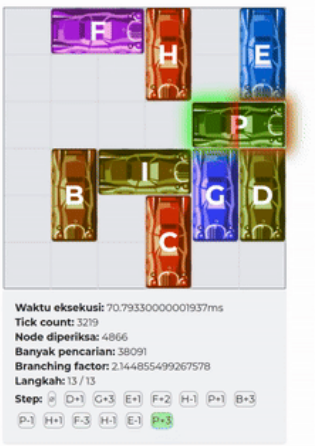
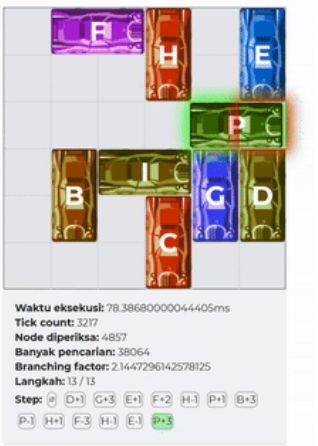
UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

4.7. Jam-7

Nama	:	Jam-7	`6 6`
			`8`
			`.BFFGE`
			`.B.HGE`
			`.PPH.DK`
			`..II.D`
			`...C..`
			`...C..`

Banyak Step Optimal	:	13	

Step Optimal	:	`ø G+3 D+1 E+1 F+2`
		`H-1 P+1 B+3 P-1 H+1`
		`F-3 H-1 E-1 P+3`	

UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

4.8. Jam-8

Nama	:	Jam-8	6 6 13 ...BBF ..GGHF PPEIHFK DDEIJJ MMNLLL OONCCC
Banyak Step Optimal	:	12	
Step Optimal	:	ø B-3 G-2 H-1 E-2 I-2 P+3 J-1 N-2 L-1 C-1 F+3 P+1	

UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

4.9. Jam-9

Nama	:	Jam-9	`6 6`
			`11`
			`.BGGHH`
			`.B.EII`
			`PP.EFDK`
			`LCCCFD`
			`L.J.FM`
			`L.J..M`

Banyak Step Optimal	:	12	

Step Optimal	:	`ϕ P+1 F+1 L-3 C-1`
		`E+2 I-1 P+1 B+1 G-1`
		`H-1 D-2 P+2`	

UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

4.10. Jam-10

Nama	:	Jam-10	`6 6`
			`11`
			`BBG.HH`
			`EEG..F`
			`LPP..FK`
			`LCCC.F`
			`L..IDD`
			`JJ.IMM`

Banyak Step Optimal	:	17	

Step Optimal	:	`ø H-1 F-1 P+2 C+2`
		`G+4 B+1 E+1 L-2 P-2`
		`C-3 F+1 H+1 I-4 D-1`
		`M-1 F+2 P+3`	

UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

4.11. Jam-11

Nama	:	Jam-11	6 6 7 EBBH .. E .. H .. EPPH .. K .. FCCC .. F .. G .. DDDG
Banyak Step Optimal	:	25	
Step Optimal	:	∅ E+3 P-1 F-2 C-1 G-3 C+1 D+1 F+3 P+1 E-3 C-3 H+1 B+3 H-1 C+3 E+3 P-1 F-4 P+1 E-3 C-3 G+2 D-3 H+3 P+3	

UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

4.12. Jam-12

Nama	:	Jam-12	`6 6`
			`7`
			`BFF..E`
			`B.H..E`
			`PPH..EK`
			`..HCCC`
			`....G.`
			`DDD.G.`

Banyak Step Optimal	:	17	

Step Optimal	:	`ϕ F+2 H-1 C-2 G-3`
		`C+2 D+3 H+3 F-1 G-1`
		`P+3 B+1 F-2 H-3 C-1`
		`D-1 E+3 P+1`	

UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

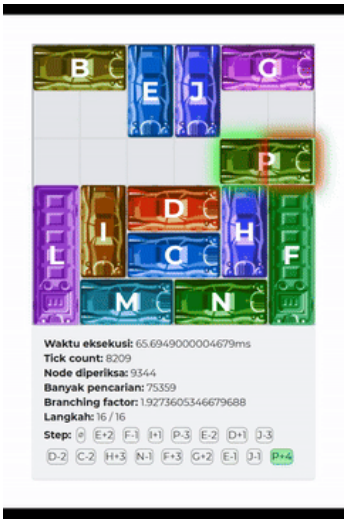
4.13. Jam-13

Nama	:	Jam-13	`6 6`
			`12`
			`BBGGH.`
			`..E.HF`
			`.IEPPFK`
			`LI.DDF`
			`L..JCC`
			`LMMJNN`

Banyak Step Optimal	:	16	

Step Optimal	:	`ø E+2 F-1 I+1 P-3`
		`E-2 D+1 J-3 D-2 C-2`
		`H+3 N-1 F+3 G+2 E-1`
		`J-1 P+4`	

UCS



GBFS Car Distance

GBFS Car Blocked

A* Car Distance

A* Car Blocked

A* Car Blocked Recursive

4.14. Jam-14

Nama	:	Jam-14	`6 6`
			`11`
			`BBG...`
			`..G.HH`
			`EIPPDJK`
			`EIMMDJ`
			`..L.CC`
			`FFL...`

Banyak Step Optimal	:	17	

Step Optimal	:	`ø E+1 I+1 P-2 G+1`
		`B+3 G-1 P+2 E-3 I-3`
		`P-2 M-2 L-2 C-4 L+1`
		`D+1 J+1 P+4`	

UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

4.15. Jam-15

Nama	:	Jam-15	`6 6`
			`13`
			`.BBGG.`
			`HHEEFL`
			`CMPPFLK`
			`CMIDFL`
			`CMIDJJ`
			`.NNOO.`

Banyak Step Optimal	:	23	

Step Optimal	:	`ø B-1 G+1 C+1 O+1`
		`N+1 M+1 P-2 I-1 D-1`
		`J-2 F+1 L+1 E+2 I-2`
		`D-2 P+2 C-1 M-1 N-2`
		`O-2 F+1 L+1 P+2`	

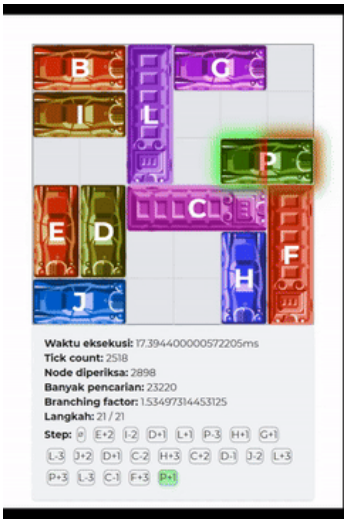
UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

4.16. Jam-16

Nama	:	Jam-16	`6 6`
			`10`
			`BBGGHF`
			`E.IIHF`
			`EDLPPFK`
			`.DLCCC`
			`..L...`
			`JJ....`

Banyak Step Optimal	:	21	

Step Optimal	:	`ø E+2 I-2 D+1 L+1`
		`P-3 H+1 G+1 L-3 J+2`
		`D+1 C-2 H+3 C+2 D-1`
		`J-2 L+3 P+3 L-3 C-1`
		`F+3 P+1`	

UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

4.17. Jam-17

Nama	:	Jam-17	6 6 11 BFFF.. B.GGHH PPE...K IIEL.. CCCLDJ MMMLDJ
Banyak Step Optimal	:	24	
Step Optimal	:	ø F+2 G-1 L-2 D-2 J-2 C+3 M+3 E+2 P+1 I+1 B+4 G-1 P-1 I-1 E-4 P+1 I+1 B-2 C-3 D+1 J+1 M-3 L+2 P+3	

UCS	GBFS Car Distance	GBFS Car Blocked
 **Waktu eksekusi:** 9.9262000005692 24ms **Tick count:** 2107 **Node diperiksa:** 2152 **Banyak pencarian:** 19664 **Branching factor:** 1.4366607666015625 **Langkah:** 24 / 24 **Step:** ø F+2 G-1 L-2 D-2 J-2 C+3 M+3 E+2 P+1 I+1 B+4 G-1 P-1 I-1 E-4 P+1 I+1 B-2 C-3 D+1 J+1 M-3 L+2 P+3	 **Waktu eksekusi:** 9.635999999940395ms **Tick count:** 993 **Node diperiksa:** 1246 **Banyak pencarian:** 9078 **Branching factor:** 1.33112335205078 12 **Langkah:** 27 / 27 **Step:** ø G-1 L-2 D-2 J-2 C+3 E+1 P+1 F+2 M+3 E+1 I+1 B+4 G-1 I-1 P-1 E-4 P+1 I+1 B-2 C-3 M-3 L+2 P+1 D+1 P+1 J+1 P+1	 **Waktu eksekusi:** 6.762299999594688ms **Tick count:** 864 **Node diperiksa:** 1098 **Banyak pencarian:** 7951 **Branching factor:** 1.32377624511718 75 **Langkah:** 27 / 27 **Step:** ø G-1 L-2 D-2 J-2 C+3 E+1 F+2 P+1 M+3 E+1 I+1 B+4 G-1 P-1 I-1 E-4 C-1 J+1 C-1 D+1 P+1 I+1 B-2 C-1 M-3 L+2 P+3

A* Car Distance	A* Car Blocked	A* Car Blocked Recursive
 **Waktu eksekusi:** 11.6979000000283 12ms **Tick count:** 2069 **Node diperiksa:** 2155 **Banyak pencarian:** 19406 **Branching factor:** 1.4130325317382812 **Langkah:** 25 / 25 **Step:** ø G-1 L-2 D-2 J-2 C+3 M+3 E+2 P+1 F+2 I+1 B+4 G-1 I-1 P-1 E-4 P+1 I+1 B-2 C-3 D+1 M-3 L+2 P+2 J+1 P+1	 **Waktu eksekusi:** 13.386600000783801ms **Tick count:** 1783 **Node diperiksa:** 1971 **Banyak pencarian:** 16898 **Branching factor:** 1.4043884277343 75 **Langkah:** 25 / 25 **Step:** ø F+2 G-1 L-2 D-2 J-2 C+3 M+3 E+2 P+1 I+1 B+4 G-1 P-1 I-1 E-4 C-2 D+1 J+1 P+1 I+1 B-2 C-1 M-3 L+2 P+3	 **Waktu eksekusi:** 50.556200000457466ms **Tick count:** 1973 **Node diperiksa:** 2041 **Banyak pencarian:** 18656 **Branching factor:** 1.4105682373046875 **Langkah:** 25 / 25 **Step:** ø F+2 G-1 L-2 D-2 J-2 C+3 M+3 E+2 P+1 I+1 B+4 G-1 P-1 I-1 E-4 C-2 D+1 J+1 P+1 I+1 B-2 C-1 M-3 L+2 P+3

4.18. Jam-18

Nama	:	Jam-18	`6 6`
			`8`
			`BBGF..`
			`HHGF..`
			`IPPF..K`
			`ICCC..`
			`IEE...`
			`DDD...`

Banyak Step Optimal	:	25	

Step Optimal	:	`ø C+2 E+3 D+3 I+1`
		`P-1 G+4 B+1 H+1 P+1`
		`I-3 C-3 F+2 B+3 H+3`
		`F-2 C+3 I+3 P-1 G-4`
		`P+1 I-3 C-3 D-3 F+3`
		`P+3`	

UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

4.19. Jam-19

Nama	:	Jam-19	6 6 7 ..BFF. ..B.G. .DPPG.K .DHHC. .EEEC. ......
Banyak Step Optimal	:	22	
Step Optimal	:	ø D-2 P-2 B+1 F-1 G-1 H-2 C-1 E+2 B+3 P+2 H+2 D+4 F-2 P-2 H-2 B-4 P+2 H+2 D-2 E-2 C+1 P+2	

UCS	GBFS Car Distance	GBFS Car Blocked
 **Waktu eksekusi:** 16.4556999999816716ms **Tick count:** 484 **Node diperiksa:** 527 **Banyak pencarian:** 3572 **Branching factor:** 1.3662185668945312 **Langkah:** 22 / 22 **Step:** ø D-2 P-2 B+1 F-1 G-1 H-2 C-1 E+2 B+3 P+2 H+2 D+4 F-2 P-2 H-2 B-4 P+2 H+2 D-2 E-2 C+1 P+2	 **Waktu eksekusi:** 1.7201000000500679ms **Tick count:** 431 **Node diperiksa:** 477 **Banyak pencarian:** 3250 **Branching factor:** 1.3389587402343375 **Langkah:** 23 / 23 **Step:** ø D-2 H-2 P-2 B+1 F-1 G-1 C-1 E+2 B+2 P+2 B+1 H+2 D+4 F-2 H-2 P-2 B-4 P+2 H+2 D-2 E-2 C+1 P+2	 **Waktu eksekusi:** 14.769399999640882ms **Tick count:** 434 **Node diperiksa:** 475 **Banyak pencarian:** 3258 **Branching factor:** 1.3597106933593375 **Langkah:** 22 / 22 **Step:** ø D-2 P-2 B+1 F-1 G-1 H-2 C-1 E+2 B+3 P+2 H+2 D+4 F-2 P-2 H-2 B-4 P+2 H+2 D-2 E-2 C+1 P+2
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
 **Waktu eksekusi:** 1.6309000002220273ms **Tick count:** 480 **Node diperiksa:** 527 **Banyak pencarian:** 3554 **Branching factor:** 1.3658599853515625 **Langkah:** 22 / 22 **Step:** ø D-2 H-2 P-2 B+1 F-1 G-1 C-1 E+2 B+3 P+2 H+2 D+4 F-2 H-2 P-2 B-4 P+2 H+2 D-2 E-2 C+1 P+2	 **Waktu eksekusi:** 7.223900000564754ms **Tick count:** 483 **Node diperiksa:** 527 **Banyak pencarian:** 3568 **Branching factor:** 1.3661422729492188 **Langkah:** 22 / 22 **Step:** ø D-2 P-2 H-2 B+1 F-1 G-1 C-1 E+2 B+3 P+2 H+2 D+4 F-2 P-2 H-2 B-4 P+2 H+2 D-2 E-2 C+1 P+2	 **Waktu eksekusi:** 13.499100000597537ms **Tick count:** 482 **Node diperiksa:** 526 **Banyak pencarian:** 3562 **Branching factor:** 1.3660202026367188 **Langkah:** 22 / 22 **Step:** ø D-2 P-2 H-2 B+1 F-1 G-1 C-1 E+2 B+3 P+2 H+2 D+4 F-2 P-2 H-2 B-4 P+2 H+2 D-2 E-2 C+1 P+2

4.20. Jam-20

Nama	:	Jam-20	6 6 9 B..FFF BGGH.. PPEH.JK ..E..J ..IDDJ ..ICCC
Banyak Step Optimal	:	10	
Step Optimal	:	∅ H+1 G+2 E-2 I-2 D-3 I+1 H+1 C-1 J+1 P+4	

UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

4.21. Jam-21

Nama	:	Jam-21	6 6 6 BBFE.. G.FE.. GPPE..K GCCC.. ....DDD
Banyak Step Optimal	:	21	
Step Optimal	:	ø G+2 P-1 C+2 F+4 B+1 P+1 G-3 C-3 E+1 B+3 E-1 C+3 G+3 P-1 F-4 P+1 G-3 C-3 D-3 E+3 P+3	

UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

4.22. Jam-22

Nama	:	Jam-22	`6 6`
			`11`
			`..BFFF`
			`G.BLHH`
			`GPPL..K`
			`.E.LII`
			`DEJJ.M`
			`DCCC.M`

Banyak Step Optimal	:	26	

Step Optimal	:	`ø G-1 P-1 D-1 J+1`
		`B+3 P+1 G+1 F-3 L-1`
		`I-1 M-2 J+1 C+2 E+1`
		`B+1 I-2 L+2 H-3 L-2`
		`M-2 I+3 E-1 B-1 C-3`
		`L+3 P+3`	

UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

4.23. Jam-23

Nama	:	Jam-23	6 6 9 ..FFFJ ..BGGJ ..BPPJK ..HEII ..HEDD ..CCC.
Banyak Step Optimal	:	29	
Step Optimal	:	∅ C-2 E+1 I-1 J+1 F+1 B-1 P-3 B+1 F-1 J-1 I+1 E-1 C+3 H+1 B+1 G-3 B-1 E-2 I-4 J+1 F+1 B-1 H-2 D-4 E+2 H+1 C-1 J+2 P+4	

UCS	GBFS Car Distance	GBFS Car Blocked
		

A* Car Distance	A* Car Blocked	A* Car Blocked Recursive
		

4.24. Jam-24

Nama	:	Jam-24	6 6 9 ..BGG. .HB... EHPPI.K EDD.I. FFF.J. CC..J.
Banyak Step Optimal	:	25	
Step Optimal	:	ø H-1 E-2 P-2 B+1 G-1 I-2 D-1 J-2 F+3 C+3 B+3 P+2 D+2 E+4 H+4 G-2 P-2 D-2 B-4 P+2 D+2 H-2 F-2 J+1 P+2	

UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

4.25. Jam-25

Nama	:	Jam-25	`6 6`
			`12`
			`BBG.HH`
			`EEG..F`
			`LPP.IFK`
			`LCCCIF`
			`LD.JMM`
			`.D.JNN`

Banyak Step Optimal	:	27	

Step Optimal	:	`ø H-1 F-1 L+1 P-1`
		`I-1 C+2 G+4 B+1 E+1`
		`P+1 L-3 C-3 J-3 C+3`
		`L+3 B-1 E-1 P-1 G-4`
		`C-2 M-2 I+2 N-1 F+3`
		`H+1 J-1 P+4`	

UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

4.26. Jam-26

Nama	:	Jam-26	`6 6`
			`11`
			`.B.FFF`
			`GB.HM.`
			`GPPHMEK`
			`ILLLME`
			`I.D..J`
			`..DCCJ`

Banyak Step Optimal	:	28	

Step Optimal	:	`ϕ F-1 E-2 I+1 L-1`
		`J-1 C+1 H+3 P+1 B+1`
		`F-2 M-1 L+2 B+3 P-1`
		`L-2 H-4 P+1 C-1 J+1`
		`L+3 I-1 B-3 D-1 C-3`
		`D+1 L-2 M+3 P+2`	

UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

4.27. Jam-27

Nama	:	Jam-27	6 6 9 BGGF.. BHHF.. PPEF.JK ..EIIJ ..D..J ..DCCC
Banyak Step Optimal	:	28	
Step Optimal	:	ø J-2 I+1 F+2 G+2 H+2 E-2 P+1 B+4 P-1 E+2 G-3 H-3 E-1 F-2 I-4 F+2 E+1 H+3 E-2 P+1 I+1 B-3 I-1 D-1 C-3 J+3 F+1 P+3	

UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

4.28. Jam-28

Nama	:	Jam-28	`6 6`
			`11`
			`FFFB..`
			`..LBGG`
			`PPL...K`
			`HELIIM`
			`HECCCM`
			`DDJJ.M`

Banyak Step Optimal	:	30	

Step Optimal	:	`ø B+1 F+3 L-1 M-1`
		`J+2 D+2 H+1 E+1 I-3`
		`B+1 G-1 M-1 C+1 L+2`
		`G-3 L-2 B-1 I+3 H-1`
		`E-1 D-2 L+3 F-3 B-1`
		`P+3 L-2 C-1 J-1 M+2`
		`P+1`	

UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

4.29. Jam-29

Nama	:	Jam-29	`6 6`
			`11`
			`FFF.L.`
			`..B.L.`
			`PPB.LGK`
			`HEEIIG`
			`HDDJ.M`
			`CCCJ.M`

Banyak Step Optimal	:	31	

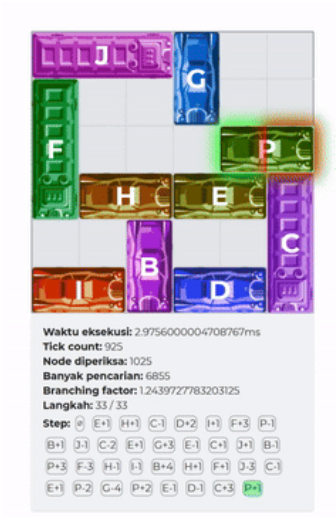
Step Optimal	:	`ϕ F+1 G-1 I+1 J-3`
		`I-1 D+2 M-1 C+3 H+1`
		`E-1 B+3 P+1 E+1 H-4`
		`P-1 E-1 B-3 D-2 C-3`
		`M+1 I+1 J+3 I-2 L+3`
		`F+2 B-1 P+3 B+1 F-1`
		`G-1 P+1`	

UCS	GBFS Car Distance	GBFS Car Blocked
		

A* Car Distance	A* Car Blocked	A* Car Blocked Recursive
		

4.30. Jam-30

Nama	:	Jam-30	6 6 9 F.BJJJ F.BG.. FPPG..K HHEE.C .....C IIDD.C
Banyak Step Optimal	:	32	
Step Optimal	:	ø E+1 H+1 C-1 D+2 I+1 F+3 P-1 B+1 J-1 C-2 E+1 G+3 E-1 C+1 J+1 B-1 P+3 F-3 H-1 I-1 B+4 P-2 H+1 F+1 J-3 C-1 E+1 G-4 E-1 D-1 C+3 P+3	

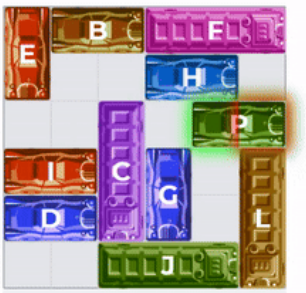
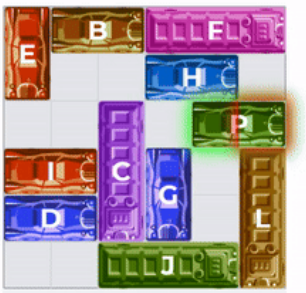
UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

4.31. Jam-31

Nama	:	Jam-31	`6 6`
			`10`
			`BB.FFF`
			`...GHH`
			`EPPG.LK`
			`E.CIIL`
			`DDC..L`
			`..CJJJ`

Banyak Step Optimal	:	37	

Step Optimal	:	`ø B+1 E-2 P-1 C-2`
		`D+3 C+2 P+1 E+4 B-1`
		`P-1 C-3 I-3 C+3 G+1`
		`H-1 L-1 D+1 G+1 P+3`
		`C-3 I+1 E-3 I-1 C+3`
		`B+1 E-1 P-3 C-2 G-1`
		`D-4 C+2 G+1 P+3 C-1`
		`J-1 L+2 P+1`	

UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

4.32. Jam-32

Nama	:	Jam-32	`6 6`
			`10`
			`BBFGHH`
			`..FG..`
			`PPF...K`
			`EIIDDL`
			`E..J.L`
			`CC.J.L`

Banyak Step Optimal	:	37	

Step Optimal	:	`ø G+1 H-1 L-3 D+1`
		`J-1 C+4 E+1 I-1 F+3`
		`J+1 D-1 L+1 H+1 G-1`
		`P+3 F-3 I+1 E-3 I-1`
		`F+3 B+1 E-1 P-3 G+1`
		`H-1 L-1 F-1 D+1 J-1`
		`C-4 F+1 J+1 D-1 L+3`
		`H+1 G-1 P+4`	

UCS	GBFS Car Distance	GBFS Car Blocked
		

A* Car Distance	A* Car Blocked	A* Car Blocked Recursive
		

4.33. Jam-33

Nama	:	Jam-33	`6 6`
			`11`
			`.BF.GG`
			`.BF...`
			`PPF...K`
			`MEEIIL`
			`MDDJHL`
			`CCCJHL`

Banyak Step Optimal	:	40	

Step Optimal	:	`ϕ G-1 L-3 I+1 J-3 C+1`
		`M+1 E-1 I-2 H-3 I+2`
		`D+3 J+2 C+2 F+3 G-1`
		`H-1 P+3 B+1 G-2 F-3`
		`E+1 M-3 E-1 F+3 G+2`
		`M-1 B-1 P-3 F-2 J-1`
		`D-4 F+2 J+1 P+3 F-1`
		`C-3 J+1 I-1 L+3 P+1`	

UCS	GBFS Car Distance	GBFS Car Blocked
		

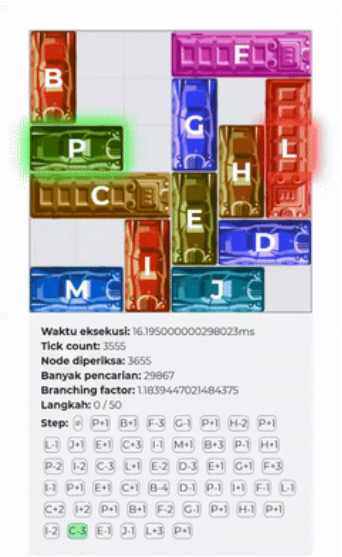
A* Car Distance	A* Car Blocked	A* Car Blocked Recursive
		

4.34. Jam-34

Nama	:	Jam-34	6 6 11 B..FFF B..G.L PP.GHLK CCCEHL ..IEDD MMIJJ.
Banyak Step Optimal	:	43	
Step Optimal	:	ϕ F-1 L-1 P+1 H-1 J+1 E+1 C+3 B+3 F-2 G-1 P-1 I-3 M+1 B+1 C-3 L+1 E-2 D-3 E+2 G+1 F+3 I-1 P+1 C+1 B-4 P-1 I+1 F-1 L-1 C+2 D-1 M-1 I+3 P+1 B+1 F-2 G-1 H-1 C-3 E-1 J-1 L+3 P+3	

UCS

GBFS Car Distance



GBFS Car Blocked

A* Car Distance

A* Car Blocked

A* Car Blocked Recursive

4.35. Jam-35

Nama	:	Jam-35	`6 6`
			`10`
			`..FBBL`
			`..FG.L`
			`PPFG.LK`
			`HCCC..`
			`HEEID.`
			`JJ.ID.`

Banyak Step Optimal	:	43	

Step Optimal	:	`ϕ L+1 B+1 G-1 D-3 J+1`
		`H+1 C-1 I-2 E+3 I+2 G+1`
		`B-1 L-1 C+3 H-1 J-1 F+3`
		`P+1 H-3 P-1 F-3 C-3 L+1`
		`B+1 G-1 I-2 E-4 D+3 I+2`
		`G+1 B-1 L-1 C+3 F+3 B-2`
		`G-1 P+3 H+1 B-1 F-3 C-1`
		`L+3 P+1`	

UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

4.36. Jam-36

Nama	:	Jam-36	`6 6`
			`11`
			`FLLLBB`
			`FGHH.C`
			`FGPP.CK`
			`MMME.C`
			`..IEDD`
			`JJI...`

Banyak Step Optimal	:	44	

Step Optimal	:	`ϕ H+1 P+1 E+1 M+1 F+1`
		`L-1 B-1 C-1 M+2 I-3 M-1`
		`C+1 B+1 L+1 F-1 M-2 E-1`
		`J+4 E+1 M+1 F+1 L-1 B-1`
		`C-1 M+2 G+3 I+3 H-2 P-2`
		`M-1 C+1 B+1 L+1 F-1 M-2`
		`E-3 M+1 F+1 L-1 E-1 D-1`
		`J-1 C+2 P+3`	

UCS	GBFS Car Distance	GBFS Car Blocked
		

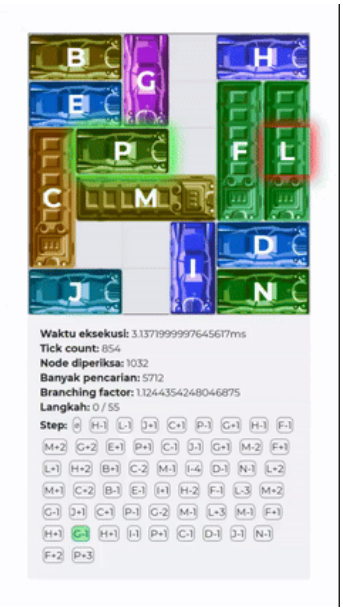
A* Car Distance	A* Car Blocked	A* Car Blocked Recursive
		

4.37. Jam-37

Nama	:	Jam-37	`6 6`
			`12`
			`BBG.HH`
			`EEG.FL`
			`CPP.FLK`
			`CMMMFL`
			`C..IDD`
			`JJ.INN`

Banyak Step Optimal	:	47	

Step Optimal	:	`ϕ J+1 C+1 P-1 G+1 H-2`
		`F-1 L-1 M+2 G+2 E+1 P+1`
		`C-1 J-1 G+1 M-2 F+1 H+1`
		`B+1 C-2 M-1 I-3 M+1 C+1`
		`B-1 H-1 F-1 M+2 G-1 J+1`
		`C+2 E-1 P-1 G-2 M-2 F+1`
		`H+1 G-1 P+1 C-1 D-2 J-1`
		`N-2 L+3 H+1 I-1 F+2 P+3`	

UCS	GBFS Car Distance	GBFS Car Blocked
		
A* Car Distance	**A* Car Blocked**	**A* Car Blocked Recursive**
		

4.38. Jam-38

Nama	:	Jam-38	`6 6`
			`10`
			`B..FFF`
			`BGGH..`
			`PPEH.LK`
			`..EIIL`
			`..DJJL`
			`..DCCC`

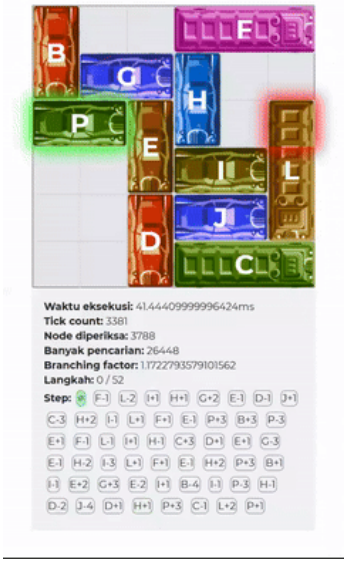
Banyak Step Optimal	:	48	
Step Optimal	:	ø F-1 L-2 I+1 H+1 G+2 E-1 D-1 J+1 C-3 H+2 I-1 L+1 F+1 E-1 P+1 B+3 P-1 E+1 F-1 L-1 I+1 H-1 C+3 B+1 D+1 E+1 G-3 E-1 H-2 I-4 L+1 F+1 E+1 H+1 G+3 E-2 P+1 I+1 B-4 P-1 I-1 D-2 J-4 D+1 H+1 C-1 L+2 P+4	

UCS

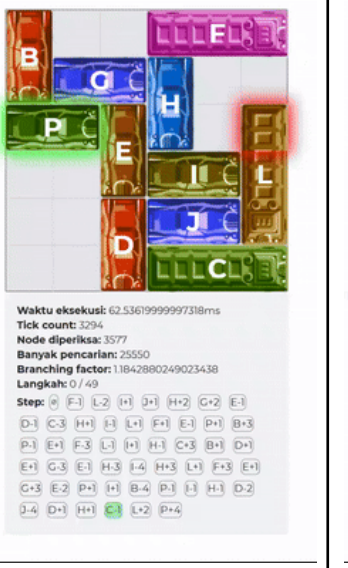
GBFS Car Distance

GBFS Car Blocked

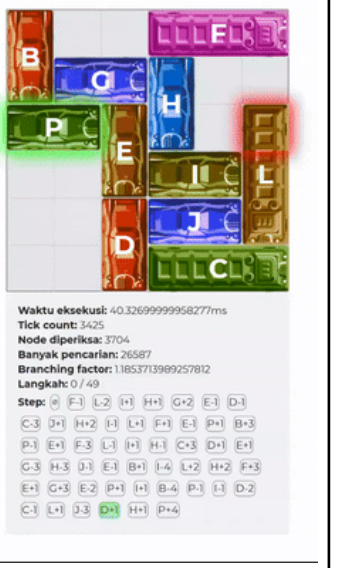
A* Car Distance



A* Car Blocked

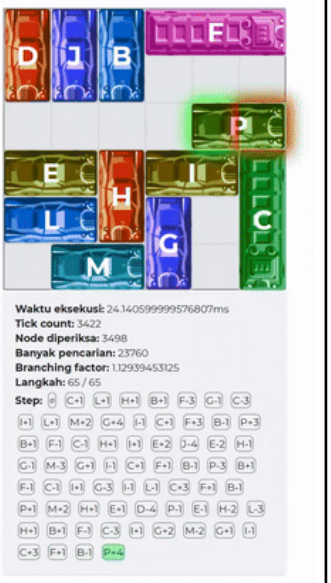


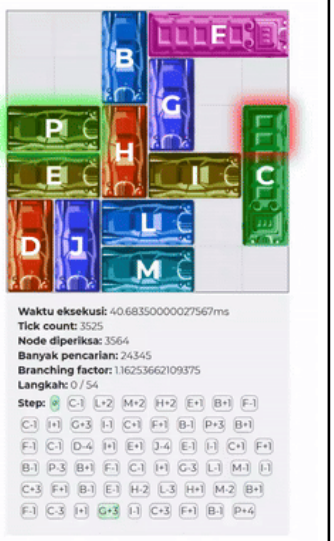
A* Car Blocked Recursive



4.39. Jam-39

Nama	:	Jam-39	6 6 11 ..BFFF ..BG.. PPHG.CK EEHIIC DJLL.C DJMM..
Banyak Step Optimal	:	50	
Step Optimal	:	∅ C-1 L+2 M+2 H+2 B+1 F-1 C-1 I+1 G+3 I-1 C+1 F+1 B-1 P+3 B+1 F-1 C-1 I+1 E+2 D-4 J-4 E-2 I-1 C+1 F+1 B-1 P-3 B+1 F-1 C-1 I+1 G-3 I-1 C+1 F+1 B-1 H-2 L-4 H+1 B+1 F-1 C-1 I+1 M-3 G+3 I-1 C+3 F+1 B-1 P+4	

UCS	GBFS Car Distance	GBFS Car Blocked
 **Waktu eksekusi:** 10.846199999563396ms **Tick count:** 3600 **Node diperiksa:** 3709 **Banyak pencarian:** 24859 **Branching factor:** 1.17906951904296 88 **Langkah:** 0 / 50 **Step:** ∅ C-1 L+2 M+2 H+2 B+1 F-1 C-1 I+1 G+3 I-1 C+1 F+1 B-1 P+3 B+1 F-1 C-1 I+1 E+2 D-4 J-4 E-2 I-1 C+1 F+1 B-1 P-3 B+1 F-1 C-1 I+1 G-3 I-1 C+1 F+1 B-1 H-2 L-4 H+1 B+1 F-1 C-1 I+1 M-3 G+3 I-1 C+3 F+1 B-1 P+4	 **Waktu eksekusi:** 15.050800000317395ms **Tick count:** 3984 **Node diperiksa:** 4165 **Banyak pencarian:** 27561 **Branching factor:** 1.13739013671875 **Langkah:** 0 / 63 **Step:** ∅ L+1 H+1 P+1 C-1 L+1 M+2 P-1 B+1 F-1 C-1 I+1 G+3 I-1 C+1 F+1 B-1 P+3 B+1 F-1 C-1 H+1 I+1 E+2 D-4 J-4 E-2 I-1 C+1 F+1 B-1 P-3 B+1 F-1 C-1 I+1 G-3 I-1 C+1 F+1 B-1 H-2 L-4 H+1 B+1 F-1 C-1 I+1 G+3 I-1 C+1 F+1 B-1 P+3 B+1 F-1 C-1 I+1 G-1 M-3 G+1 I-1 C+3 P+1	 **Waktu eksekusi:** 24.140599999576807ms **Tick count:** 3422 **Node diperiksa:** 3498 **Banyak pencarian:** 23760 **Branching factor:** 1.12939453125 **Langkah:** 65 / 65 **Step:** ∅ C-1 L+1 H+1 B+1 F-3 G-1 C-3 I+1 L+1 M+2 G+4 I-1 C+1 F+3 B-1 P+3 B+1 F-1 C-1 H+1 I+1 E+2 J-4 E-2 H-1 G-1 M-3 G+1 I-1 C+1 F+1 B-1 P-3 B+1 F-1 C-1 I+1 G-3 I-1 L-1 C+3 F+1 B-1 P+1 M+2 H+1 E+1 D-4 P-1 E-1 H-2 L-3 H+1 B+1 F-1 C-3 I+1 G+2 M-2 G+1 I-1 C+3 F+1 B-1 P+4

A* Car Distance	A* Car Blocked	A* Car Blocked Recursive
 **Waktu eksekusi:** 61.393699999989867ms **Tick count:** 3732 **Node diperiksa:** 3854 **Banyak pencarian:** 25787 **Branching factor:** 1.1800460815429688 **Langkah:** 0 / 50 **Step:** ∅ C-1 L+2 M+2 H+2 B+1 F-1 C-1 I+1 G+3 I-1 C+1 F+1 B-1 P+3 B+1 F-1 C-1 I+1 E+2 D-4 J-4 E-2 I-1 C+1 F+1 B-1 P-3 B+1 F-1 C-1 I+1 G-3 I-1 C+1 F+1 B-1 H-2 L-4 H+1 B+1 F-1 C-1 I+1 M-3 G+3 I-1 C+3 F+1 B-1 P+4	 **Waktu eksekusi:** 47.360600000247 36ms **Tick count:** 3542 **Node diperiksa:** 3605 **Banyak pencarian:** 24457 **Branching factor:** 1.1590652465820312 **Langkah:** 0 / 55 **Step:** ∅ C+1 L+1 H+1 B+1 F-3 G-1 C-3 I+1 L+1 M+2 G+4 I-1 C+1 F+3 B-1 P+3 B+1 F-1 C-1 H+1 I+1 E+2 D-4 J-4 E-2 I-1 C+1 F+1 B-1 P-3 H-1 B+1 F-1 C-1 I+1 G-3 I-1 L-1 M-3 C+3 F+1 B-1 H-1 L-3 H+1 B+1 F-1 C-3 I+1 G+3 I-1 C+3 F+1 B-1 P+4	 **Waktu eksekusi:** 40.683500000275 67ms **Tick count:** 3525 **Node diperiksa:** 3564 **Banyak pencarian:** 24345 **Branching factor:** 1.1625366210937 5 **Langkah:** 0 / 54 **Step:** ∅ C-1 L+2 M+2 H+2 E+1 B+1 F-1 C-1 I+1 G+3 I-1 C+1 F+1 B-1 P+3 B+1 F-1 C-1 D-4 I+1 E+1 J-4 E-1 I-1 C+1 F+1 B-1 P-3 B+1 F-1 C-1 I+1 G-3 L-1 M-1 I-1 C+3 F+1 B-1 E-1 H-2 L-3 H+1 M-2 B+1 F-1 C-3 I+1 G+3 I-1 C+3 F+1 B-1 P+4

4.40. Jam-40

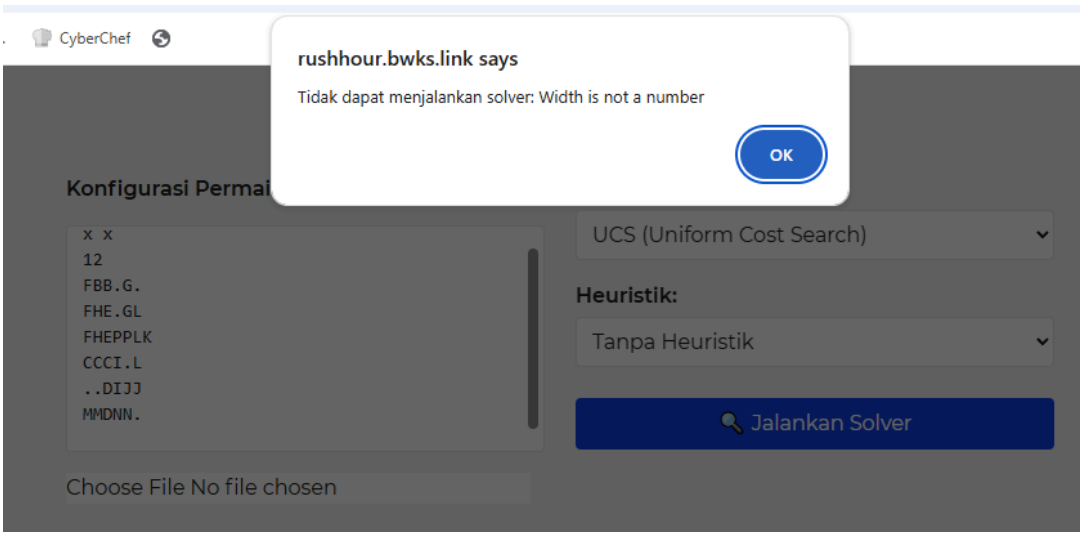
Nama	:	Jam-40	6 6 12 FBB.G. FHE.GL FHEPPLK CCCI.L ..DIJJ MMDNN.
Banyak Step Optimal	:	51	
Step Optimal	:	ø L-1 N+1 I+1 C+3 H+2 D-1 M+1 F+3 B-1 E-1 P-3 E+1 G+1 B+3 E-1 P+2 F-3 H-3 P-1 M-1 D+1 C-3 L+1 B+1 I-4 P+1 C+2 H+3 P-1 I+1 B-1 L-1 C+1 D-1 J-1 M+1 F+3 P-1 E+1 B-3 E-1 I-1 G-1 P+1 F-1 M-1 D+1 C-1 N-1 L+3 P+3	

UCS	GBFS Car Distance	GBFS Car Blocked
 **Waktu eksekusi:** 26.08600000012666ms **Tick count:** 3024 **Node diperiksa:** 3202 **Banyak pencarian:** 24451 **Branching factor:** 1.1743698120117188 **Langkah:** 51 / 51	 **Waktu eksekusi:** 13.316199999302626ms **Tick count:** 2296 **Node diperiksa:** 2772 **Banyak pencarian:** 18484 **Branching factor:** 1.155853271484375 **Langkah:** 0 / 54	 **Waktu eksekusi:** 14.595999999903142ms **Tick count:** 1577 **Node diperiksa:** 1981 **Banyak pencarian:** 11900 **Branching factor:** 1.1451568603515625 **Langkah:** 0 / 54

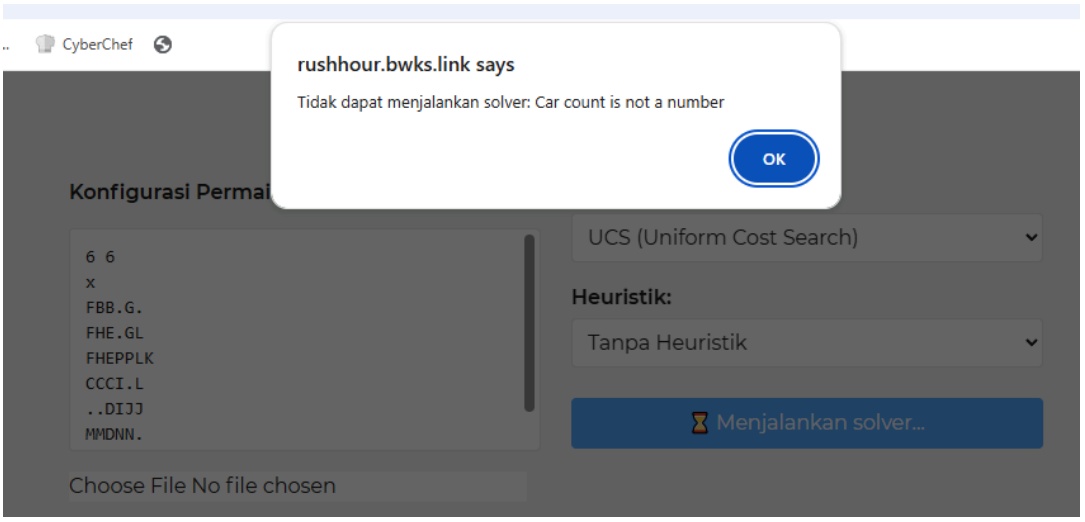
A* Car Distance	A* Car Blocked	A* Car Blocked Recursive
 **Waktu eksekusi:** 56.75549999999702ms **Tick count:** 2912 **Node diperiksa:** 3171 **Banyak pencarian:** 23622 **Branching factor:** 1.1734695434570312 **Langkah:** 0 / 51	 **Waktu eksekusi:** 46.148999999277294ms **Tick count:** 2704 **Node diperiksa:** 3032 **Banyak pencarian:** 21892 **Branching factor:** 1.1674880981445312 **Langkah:** 0 / 52	 **Waktu eksekusi:** 27.233400002928ms **Tick count:** 2240 **Node diperiksa:** 2635 **Banyak pencarian:** 17493 **Branching factor:** 1.1617965698242188 **Langkah:** 0 / 52

4.41. *Invalid Input*

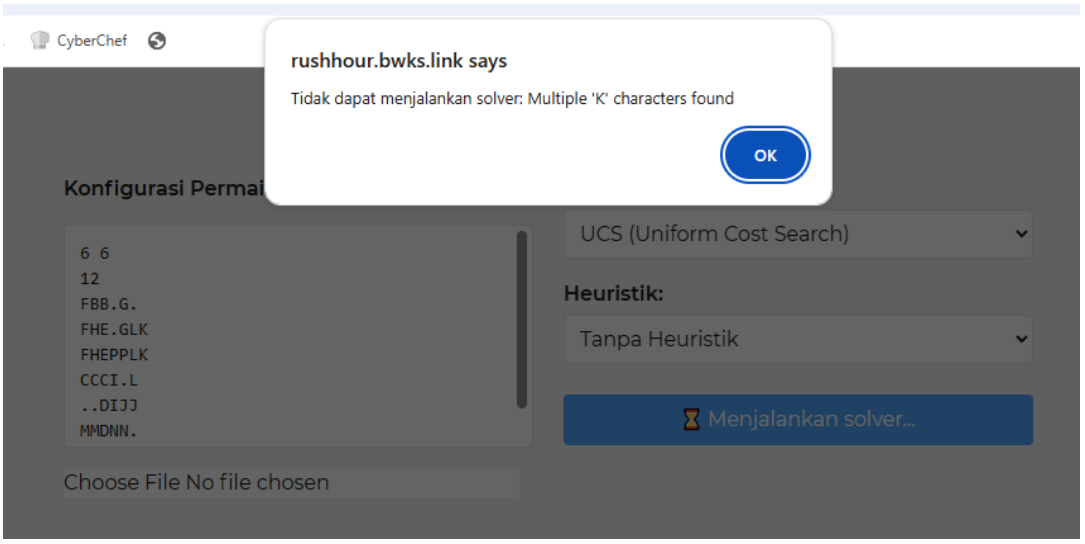
Alasan	:	Tinggi/Lebar bukan angka	`x x`
			`12`
			`FBB.G.`
			`FHE.GL`
			`FHEPPLK`
			`CCCI.L`
			`..DIJJ`
			`MMDNN.`



Alasan	:	Jumlah *piece* bukan angka	`6 6`
			`x`
			`FBB.G.`
			`FHE.GL`
			`FHEPPLK`
			`CCCI.L`
			`..DIJJ`
			`MMDNN.`

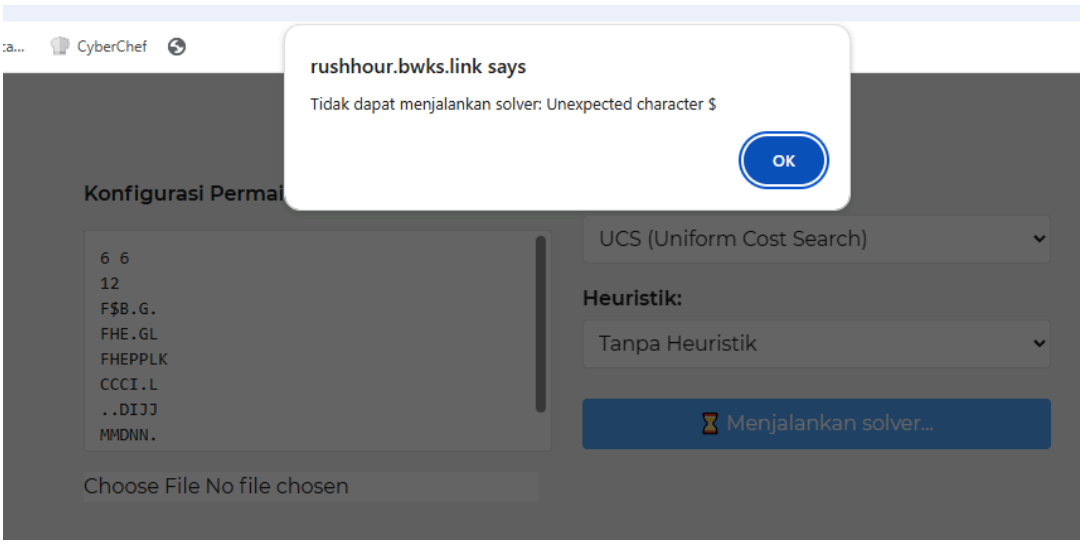


Alasan	:	Terdpat exit ‘K’ lebih dari 1	6 6 12 FBB.G. FHE.GLK FHEPPLK CCCI.L ..DIJJ MMDNN.



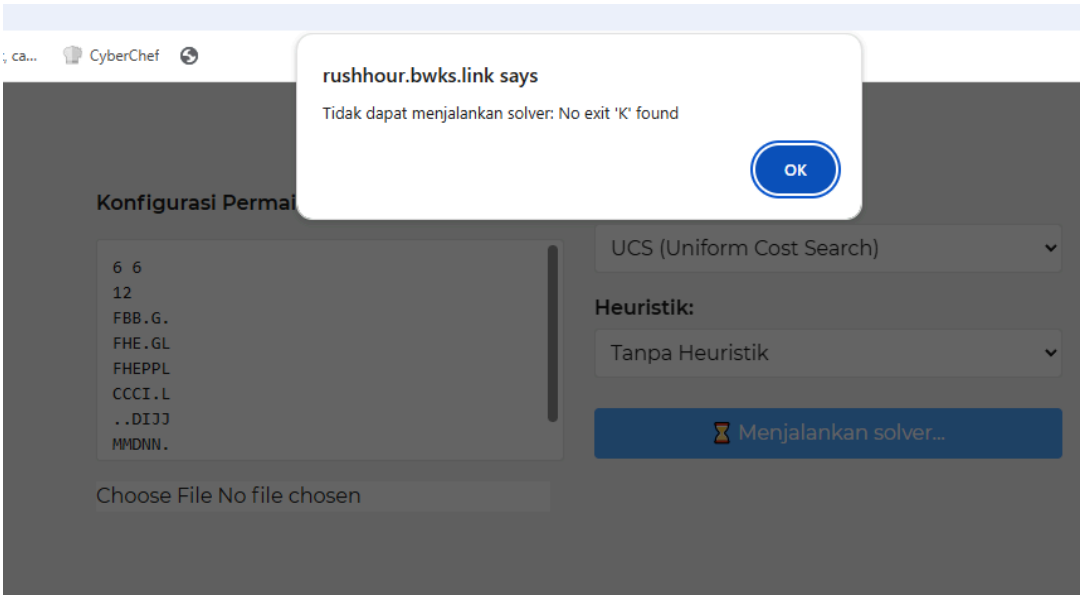
Alasan	:	Karakter tidak dikenal/invalid	6 6 12 F$B.G. FHE.GL FHEPPLK CCCI.L

```
..DIJJ
MMDNN.
```



Alasan : Tidak ada exit ‘K’

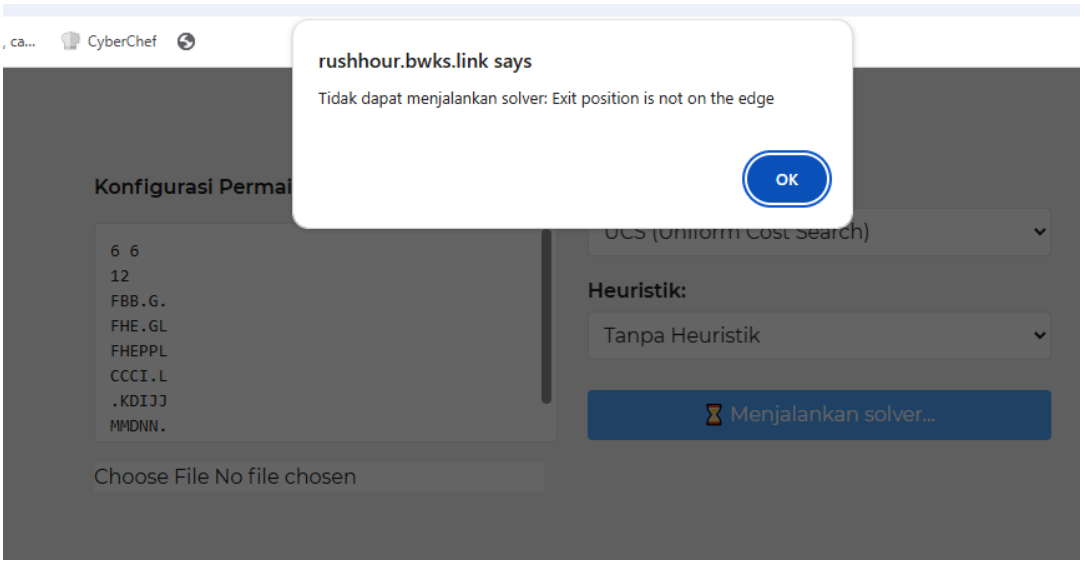
```
6 6
12
FBB.G.
FHE.GL
FHEPPL
CCCI.L
..DIJJ
MMDNN.
```



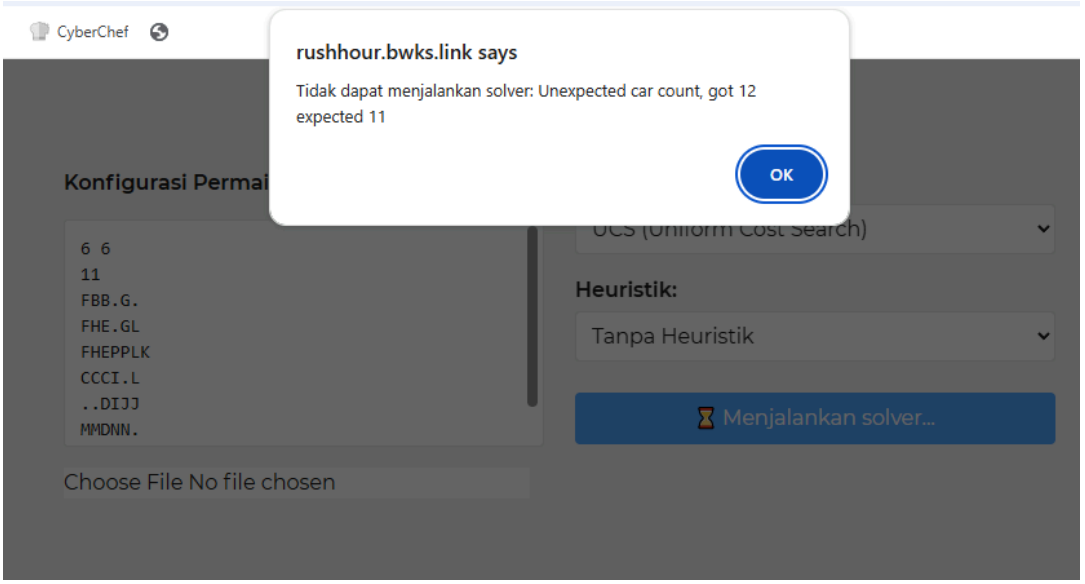
Alasan : Exit tidak berada di pinggir

```
6 6
```

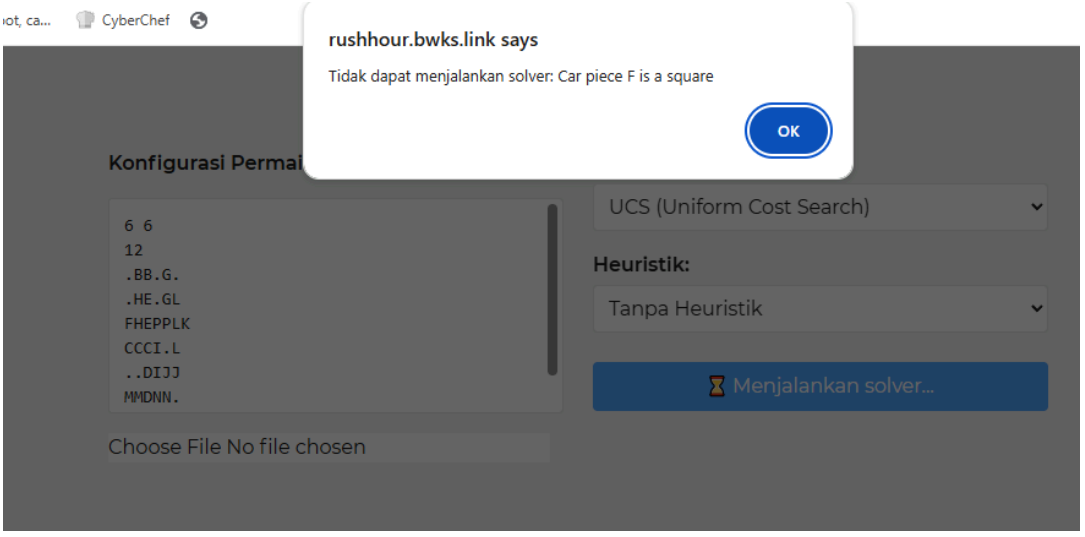
	papan	12 FBB.G. FHE.GL FHEPPL CCCI.L .KDIJJ MMDNN.



Alasan	:	Jumlah piece tidak sesuai	6 6 11 FBB.G. FHE.GL FHEPPLK CCCI.L ..DIJJ MMDNN.

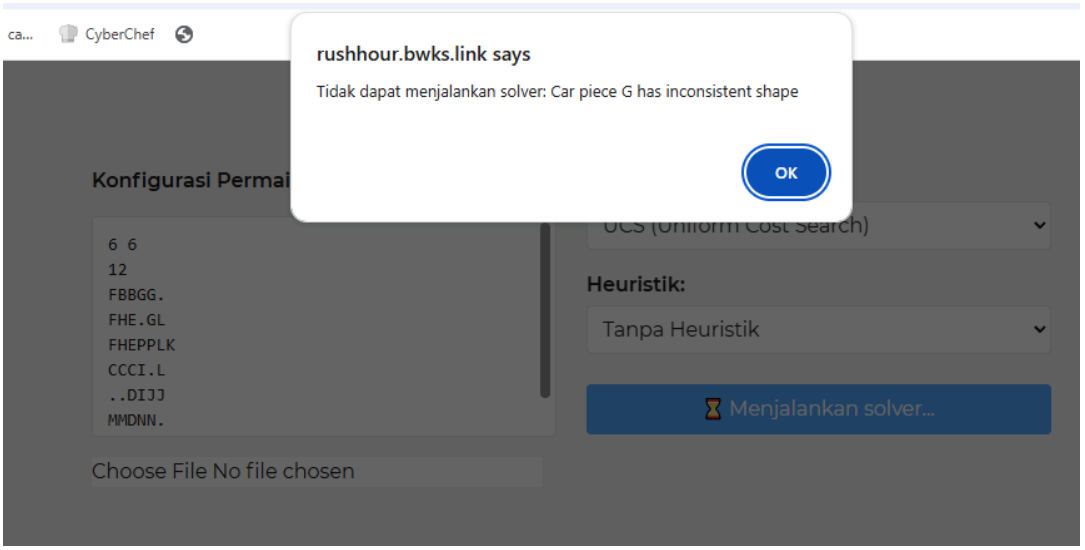


Alasan	:	Piece dengan width 1 dan height 1	6 6 12 .BB.G. .HE.GL FHEPPLK CCCI.L ..DIJJ MMDNN.



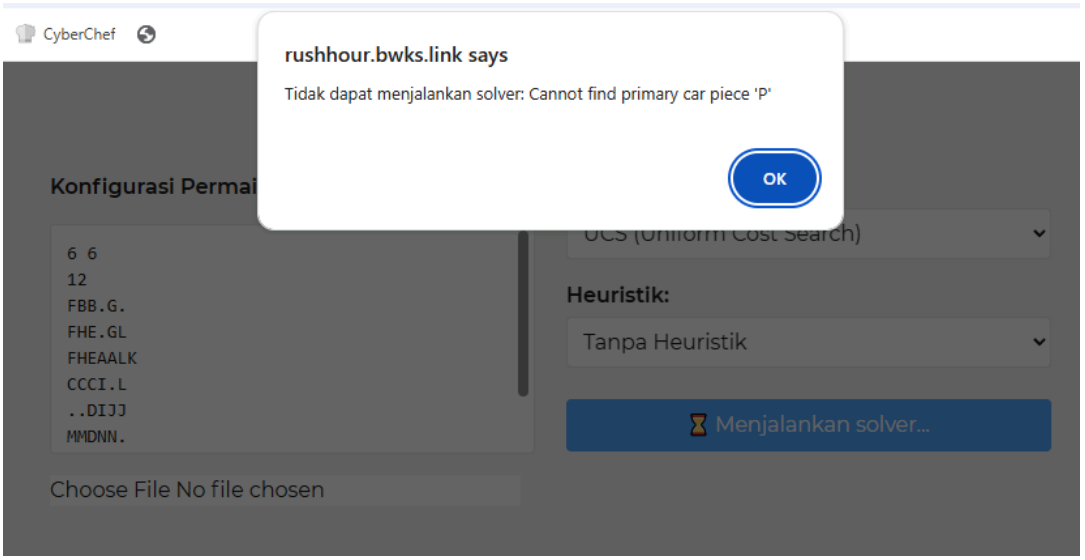
Alasan	:	Ukuran piece tidak konsisten	6 6 12 FBBGG. FHE.GL

```
FHEPPLK
CCCI.L
..DIJJ
MMDNN.
```



Alasan : Tidak ada mobil utama

```
6 6
12
FBB.G.
FHE.GL
FHEAALK
CCCI.L
..DIJJ
MMDNN.
```



BAB V ANALISIS

5.1. Analisis

Algoritma *Uniform Cost Search* (UCS) menjamin solusi yang optimal. Namun, kelemahannya adalah eksplorasi state yang sangat luas, sehingga secara teori membutuhkan waktu komputasi yang besar. Meskipun telah dilakukan optimasi melalui `QueueSolverUniform`—yang menghindari revisit state dengan `hashCode` yang sama selama heuristik bersifat konsisten—performa waktu yang dihasilkan dapat menyerupai A* secara teknis, meskipun pendekatannya tetap murni UCS. Dalam UCS, fungsi evaluasi adalah `g(n) = step_cost(n)` dengan `h(n) = 0`, yang berarti tidak ada heuristik yang digunakan.

Greedy Best First Search (GBFS) memiliki keunggulan dalam kecepatan eksekusi karena selalu memilih state dengan nilai heuristik terkecil tanpa memperhitungkan biaya langkah sebelumnya. Pendekatan ini sering kali menghasilkan waktu penyelesaian tercepat dalam *wall clock*, namun tidak menjamin solusi yang optimal. GBFS bersifat rakus terhadap keuntungan lokal dan dapat mengabaikan jalur yang lebih baik. Dalam formulasi GBFS, digunakan `g(n) = 0` dan `h(n) = heuristik(n)`.

A* merupakan paduan antara UCS dan GBFS. A* mempertimbangkan baik biaya langkah (`step_cost`) maupun nilai heuristik untuk mengarahkan pencarian ke solusi lebih efisien. A* akan mengeksplorasi semua kemungkinan solusi seperti UCS, tetapi lebih terfokus karena bantuan heuristik, yang membuatnya cenderung lebih efisien. Rumus evaluasi A* adalah `g(n) = step_cost(n)` dan `h(n) = heuristik(n)`. Secara teoretis, A* lebih efisien daripada UCS selama heuristik yang digunakan konsisten dan *admissible*.

Iterative Deepening A* (IDA*) adalah versi A* yang diimplementasikan menggunakan pendekatan *recursive stack*. Keuntungan teoretis IDA* adalah efisiensi memori karena tidak menyimpan seluruh antrian state seperti A*. Namun, dalam implementasi berbasis JavaScript dan V8 engine, penghematan memori ini tidak tampak signifikan dalam hasil pengujian. IDA* menggunakan fungsi evaluasi yang sama seperti A*, yaitu `g(n) = step_cost(n)` dan `h(n) = heuristik(n)`.

Dalam hal heuristik, tiga pendekatan digunakan. *Car Distance* merupakan heuristik paling dasar namun admissible, karena hanya menghitung jarak langsung antara mobil utama ke tujuan tanpa mempertimbangkan hambatan. *Car Blocking* memperbaikinya dengan menghitung jumlah mobil yang menghalangi jalan, tetap admissible dan lebih efektif dari sebelumnya. *Car Blocking Recursive* mengembangkan konsep *Car Blocking* dengan menganalisis hambatan secara rekursif, memberikan estimasi yang lebih akurat tetapi memerlukan waktu komputasi yang lebih tinggi untuk satu iterasi solver.

Perlu dicatat bahwa UCS dan BFS akan menghasilkan urutan ekspansi dan solusi yang identik dalam konteks Rush Hour karena semua gerakan memiliki biaya yang sama. A* secara teoritis lebih efisien dibandingkan UCS karena memanfaatkan arah dari heuristik. Sebaliknya, GBFS tidak menjamin solusi optimal karena hanya berfokus pada estimasi heuristik dan dapat mengabaikan jalur yang lebih baik secara global.

5.2. Kesimpulan

Program penyelesaian *puzzle* Rush Hour berhasil dikembangkan sesuai dengan spesifikasi yang ditetapkan, menggunakan bahasa pemrograman JavaScript. Seluruh fitur utama seperti implementasi algoritma pencarian (UCS, GBFS, A*, IDA*), berbagai heuristik (*car distance*, *car blocking*, dan *car blocking recursive*), serta optimasi struktural dan fungsional telah direalisasikan dengan baik. Selain itu, program juga dilengkapi GUI pengguna berbasis web yang interaktif, mendukung input dari teks maupun file, serta animasi visual solusi. Dengan penerapan berbagai teknik optimasi dan fitur tambahan seperti output video dan deployment online, program ini menunjukkan performa dan fungsionalitas yang solid dan dapat diandalkan.

5.3. Saran

Tugas Kecil ini telah berjalan dengan cukup baik, dan saya berterima kasih atas bantuan asisten yang telah memberikan kemudahan serta arahan dalam proses pengerjaannya. Sebagai saran, ke depannya semoga bisa lebih dalam dan dapat lebih memfokuskan perhatian pada aspek optimasi program, khususnya dalam menganalisis jalur (path) yang diambil oleh setiap algoritma dan bagaimana strategi teknis maupun algoritmik dapat digunakan untuk mempercepat proses pencarian solusi. Meskipun demikian, saya cukup bangga karena tugas kecil ini dapat diselesaikan dengan baik dan mencerminkan pemahaman yang mendalam terhadap materi yang diberikan.

LAMPIRAN

Tautan

Link Github : Tucil3_13523045 Link *Deployment* : https://rushhour.bwks.link/

Link Docs : Tucil3_13523045

Tabel Laporan

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