

Laporan Tugas Kecil 3
IF2211 Strategi Algoritma 2025/2026
Penyelesaian Puzzle Rush Hour Menggunakan Algoritma
Pathfinding



Disusun oleh:

Varel Tiara (13523008)
Yonatan Edward Njoto (13523036)

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

2025

KATA PENGANTAR

Dengan penuh rasa syukur, kami menyusun dan mempersembahkan laporan ini sebagai bagian dari Tugas Kecil 3 mata kuliah IF2211 Strategi Algoritma. Topik yang diangkat dalam tugas besar ini adalah “*Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding*”, yang bertujuan untuk menerapkan berbagai strategi algoritma *pathfinding* dalam menyelesaikan masalah penyelesaian permainan puzzle Rush Hour.

Melalui tugas ini, kami berhasil mengembangkan sebuah aplikasi penyelesaian yang mampu menemukan solusi setiap puzzle dengan menampilkan konfigurasi setiap papan pada setiap pergeseran atau pergerakan dengan menggunakan berbagai algoritma *pathfinding* seperti Uniform Cost Search (UCS), Greedy Best First Search (GBFS), A*, Dijkstra, atau Fringe Search. Selain itu, di algoritma Greedy Best First Search dan A* terdapat beberapa heuristik yang dapat digunakan, seperti Manhattan Distance, Blocking Vehicles, dan Combine dari kedua heuristik tersebut.

Melalui penggeraan tugas kecil ini, kami belajar untuk menggabungkan aspek teoretis dan praktis secara seimbang, serta mengeksplorasi penerapan algoritma dalam konteks nyata dan menyenangkan. Oleh karena itu, kami mengucapkan terima kasih kepada dosen pengampu serta seluruh pihak yang telah membimbing dan memberikan dukungan selama penyusunan tugas ini. Semoga laporan ini dapat memberikan gambaran yang jelas mengenai hubungan antara strategi algoritma dan pengembangan aplikasi nyata, serta menjadi referensi yang bermanfaat bagi pembaca yang tertarik dalam bidang algoritma.

Bandung, 17 Mei 2025

Varel Tiara (13523008)

Yonatan Edward Njoto (13523036)

DAFTAR ISI

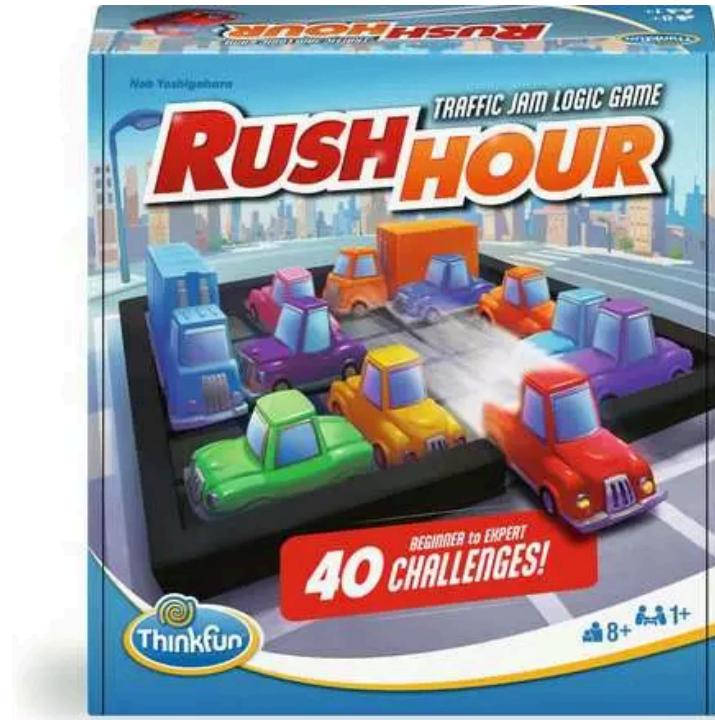
KATA PENGANTAR	2
DAFTAR ISI	3
BAB I	5
PENJELASAN ALGORITMA	5
2.1 Deskripsi Tugas	5
2.2 Algoritma Uniform-Cost Search (UCS)	8
2.3 Algoritma Greedy Best First Search (GBFS)	11
2.4 Algoritma A*	14
2.5 Algoritma Dijkstra	17
2.6 Algoritma Fringe Search	20
BAB II	24
ANALISIS PEMECAHAN MASALAH	24
3.1 Analisis Penyelesaian Puzzle Rush Hour	24
3.2 Fungsi Evaluasi $f(n)$ dan $g(n)$	25
3.3 Keadmissible-an Heuristik pada Algoritma A*	26
3.4 Perbandingan UCS dan BFS pada Penyelesaian Rush Hour	26
3.5 Efisiensi A* dibanding UCS pada Rush Hour	27
3.6 Apakah Greedy Best-First Search Menjamin Solusi Optimal?	28
BAB III	29
SOURCE PROGRAM	29
4.1 Type	29
4.2 Algoritma	31
4.3 Lainnya	37
BAB IV	54
PENGUJIAN	54
5.1 Hasil Pengujian	54
5.2 Analisis Hasil Pengujian	68
5.2.1 Algoritma Uniform Cost Search (UCS)	68
5.2.2 Algoritma Greedy Best-First Search (GBFS)	68
5.2.3 Algoritma A* Search	69
5.2.4 Algoritma Dijkstra	69
5.2.5 Algoritma Fringe Search	69
5.2.6 Analisis Hasil Pengujian	70
BAB V	72

PENJELASAN BONUS	72
BAB VI	74
KESIMPULAN	74
BAB VIII	75
LAMPIRAN	75
6.1 Daftar Pustaka	75
6.2 Tautan Repository	75
6.3 Tautan Deployment	75
6.4 Tabel Checklist	76

BAB I

PENJELASAN ALGORITMA

2.1 Deskripsi Tugas



Gambar 1. Rush Hour Puzzle

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

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

Komponen penting dari permainan Rush Hour terdiri dari:

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

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

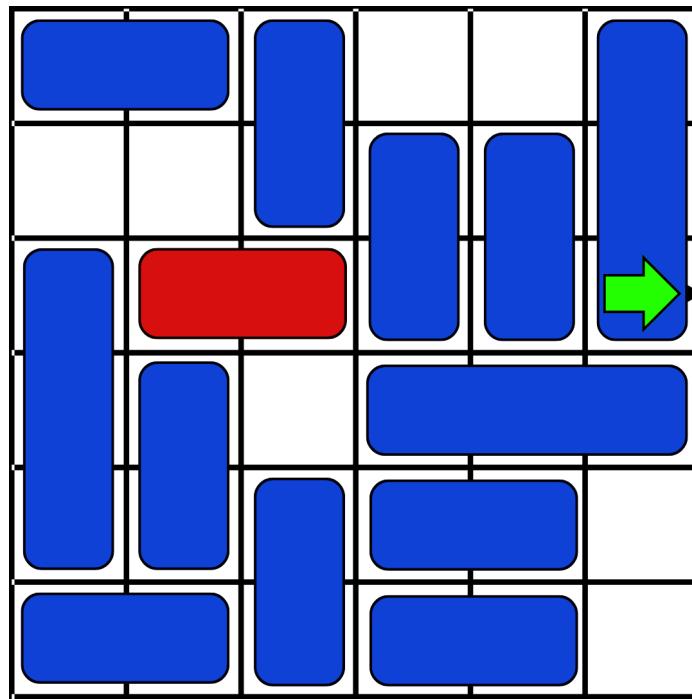
horizontal atau *vertikal*.

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

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

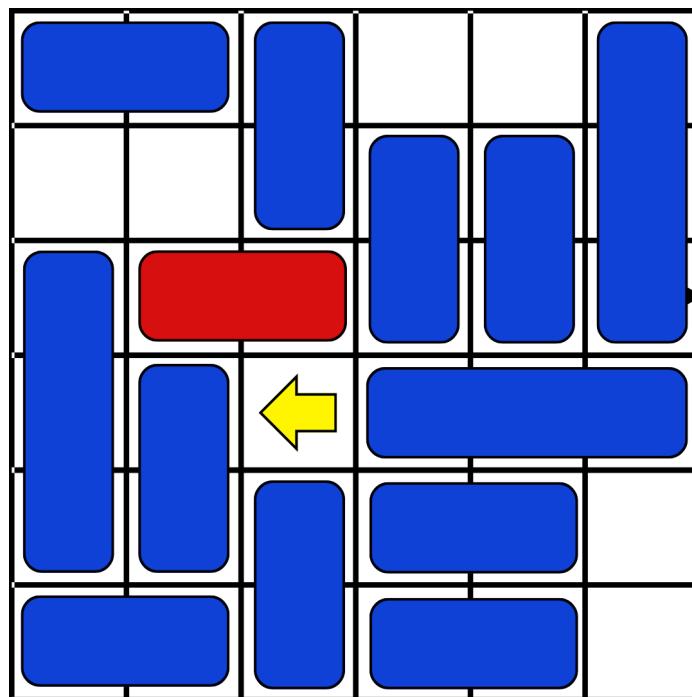
Ilustrasi kasus :

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



Gambar 2. Awal Permainan Game Rush Hour

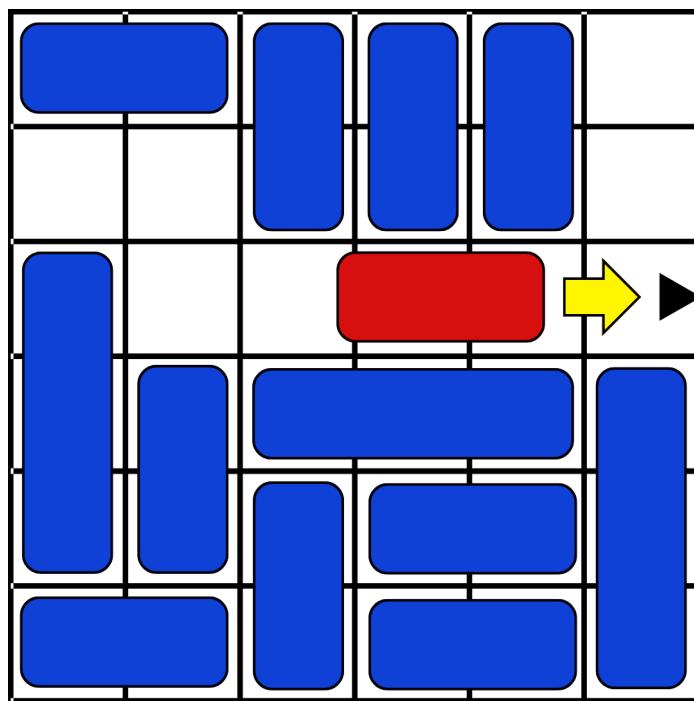
Pemain dapat menggeser-geser *piece* (termasuk *primary piece*) untuk membentuk jalan lurus antara *primary piece* dan *pintu keluar*.



Gambar 3. Gerakan Pertama Game Rush Hour

Gambar 4. Gerakan Kedua Game Rush Hour

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



Gambar 5. Pemain Menyelesaikan Permainan

2.2 Algoritma Uniform-Cost Search (UCS)

Algoritma Uniform-Cost Search (UCS) merupakan salah satu metode pencarian jalur dalam graf berbobot yang termasuk dalam kategori uninformed search, atau pencarian tanpa informasi tambahan (blind search). Berbeda dengan algoritma seperti Breadth-First Search (BFS) yang hanya mempertimbangkan jumlah langkah, UCS bekerja dengan memperhitungkan biaya kumulatif dari node awal menuju node tujuan. Oleh karena itu, UCS sangat cocok digunakan untuk menemukan jalur dengan total biaya paling rendah, terutama dalam situasi di mana setiap lintasan memiliki bobot yang merepresentasikan jarak, waktu, energi, atau bentuk biaya lainnya.

Algoritma UCS menggunakan struktur data priority queue, di mana node dengan biaya total terkecil akan selalu menjadi prioritas utama untuk diproses terlebih dahulu. Setiap kali sebuah node diekstraksi dari antrian, algoritma akan memeriksa apakah node tersebut adalah tujuan akhir. Jika ya, maka jalur dengan biaya minimum telah ditemukan dan pencarian dapat dihentikan. Jika belum, semua tetangga dari node tersebut akan dimasukkan ke dalam antrian dengan prioritas berdasarkan biaya akumulatif dari node awal. Jika sebuah node sudah ada dalam antrian namun ditemukan jalur yang lebih murah menuju node tersebut, maka biaya pada antrian akan diperbarui. Pendekatan ini membuat UCS sangat mirip dengan algoritma Dijkstra, namun dalam UCS, simpul-simpul dimasukkan ke dalam antrian secara bertahap, bukan sekaligus di awal.

Fungsi yang digunakan dalam UCS untuk menentukan urutan ekspansi node adalah $f(n) = g(n)$, di mana $g(n)$ adalah biaya total dari node awal menuju node n. UCS menjamin solusi optimal selama semua bobot lintasan bernilai non-negatif. Dari segi kompleksitas waktu, algoritma ini dapat dinyatakan dalam bentuk $O(b^{(1 + C/\varepsilon)})$, di mana b adalah rata-rata banyaknya cabang (branching factor), C adalah total biaya optimal ke tujuan, dan ε adalah biaya terkecil dari setiap langkah. Kompleksitas ini menunjukkan bahwa meskipun UCS menjamin jalur termurah, ia bisa menjadi mahal secara komputasi, terutama jika terdapat banyak jalur dengan biaya yang hampir serupa.

Notasi Pseudocode:

```
function UCS(initialBoard, initialPieces):
    # 1. Inisialisasi
    MAX_COST ← boardWidth * boardHeight * 50
    startTime ← current time
    nodesVisited ← 0

    visitedStates ← empty set      # untuk mencegah memproses state yang sama dua kali
    openList ← empty min-heap ordered by cost (g(n))

    # Buat state awal
    initialState:
        board ← initialBoard
```

```

pieces ← initialPieces
stateString ← stringify(initialBoard) # representasi unik papan
cost ← 0                                # g(initial) = 0
moves ← []                                 # tidak ada langkah awal

openList.push(initialState)                # enqueue state awal

bestSolution:
cost ← Infinity
moves ← []

# 2. Proses frontier hingga kosong atau solusi ditemukan
while openList is not empty:
    nodesVisited ← nodesVisited + 1

    current ← openList.pop()              # ambil state dengan g(n) terendah

    # Abaikan jika sudah pernah dikunjungi
    if current.stateString in visitedStates:
        continue
    visitedStates.add(current.stateString)

    # Jika ini state goal, kita punya solusi optimal pertama
    if isSolved(current.pieces):
        bestSolution ← { cost: current.cost, moves: current.moves }
        elapsed ← current.time - startTime
        return SolutionResult(
            solved = true,
            moves = current.moves,
            nodesVisited = nodesVisited,
            executionTime = elapsed
        )

    # Jika cost sudah terlalu besar, skip untuk menghindari loop tak berujung
    if current.cost > MAX_COST:
        continue

    # 3. Expand semua langkah valid dari current
    for each move in getValidMoves(current.board, current.pieces):
        (nextBoard, nextPieces) ← applyMove(current.board, current.pieces, move)
        nextKey ← stringify(nextBoard)

        # Abaikan jika nextKey sudah diproses
        if nextKey in visitedStates:

```

```

        continue

    nextCost ← current.cost + 1      # setiap move menambah cost 1
    nextMoves ← current.moves + [move]

    nextState:
        board ← nextBoard
        pieces ← nextPieces
        stateString ← nextKey
        cost ← nextCost
        moves ← nextMoves

    openList.push(nextState)          # enqueue untuk nanti dieksplorasi

    # 4. Jika tidak ada solusi dan openList habis
    elapsed ← current time – startTime
    return SolutionResult(
        solved = (bestSolution.cost ≠ Infinity),
        moves = bestSolution.moves,
        nodesVisited = nodesVisited,
        executionTime = elapsed
    )

```

2.3 Algoritma Greedy Best First Search (GBFS)

Greedy Best-First Search merupakan salah satu algoritma pencarian yang menggunakan pendekatan heuristik untuk menemukan jalur menuju tujuan. Algoritma ini berfokus pada perluasan node yang tampaknya paling dekat dengan tujuan berdasarkan suatu fungsi evaluasi, yaitu $f(n) = h(n)$ di mana $h(n)$ adalah fungsi heuristik yang memberikan estimasi jarak atau biaya dari node saat ini menuju node tujuan. Karena hanya menggunakan estimasi dari posisi saat ini ke tujuan, Greedy Best-First Search tidak memperhitungkan biaya dari titik awal hingga titik tersebut.

Cara kerja Greedy Best-First Search cukup sederhana namun ketika diimplementasikan harus strategis. Setiap node yang dapat dijangkau dari posisi saat ini akan dievaluasi menggunakan nilai heuristiknya, dan algoritma akan memilih node dengan nilai $h(n)$ terkecil,

yang artinya node tersebut diperkirakan paling dekat dengan tujuan. Proses ini akan diulang terus-menerus, memperluas node yang memiliki estimasi terbaik menuju tujuan, sampai akhirnya node tujuan ditemukan atau tidak ada lagi node yang bisa dieksplorasi. Kemudian, karena hanya fokus pada $h(n)$, algoritma ini bergerak secara greedy yang langsung bergerak ke node yang kelihatannya paling menjanjikan tanpa mempertimbangkan jalur sebelumnya (optimum local).

Greedy Best-First Search cukup cepat dan efisien dalam menemukan suatu solusi, apalagi jika fungsi heuristik yang digunakan cukup akurat. Ia juga memiliki kebutuhan memori yang relatif rendah karena hanya menyimpan node-node yang sedang dipertimbangkan untuk diekspansi. Namun demikian, algoritma ini memiliki beberapa kelemahan, seperti algoritma ini tidak menjamin akan menemukan solusi jika memang ada, terutama jika pencarian terjebak pada kondisi lokal yang tampak optimal (local minima) atau datar (plateau). Selain itu, karena hanya mengandalkan estimasi heuristik, algoritma ini rentan menghasilkan solusi yang tidak optimal. Ia bisa melewati jalur terbaik hanya karena salah mengestimasi jarak atau biaya. Kekurangan lainnya adalah sifatnya yang irrevocable atau tidak dapat dibatalkan sehingga jika sekali memilih jalur tertentu, algoritma tidak mengevaluasi kembali apakah jalur tersebut benar-benar yang terbaik. Hal ini menyebabkan potensi tersesat dalam pencarian, terutama pada lingkungan dengan banyak percabangan atau nilai heuristik yang kurang representatif.

Notasi Pseudocode:

```
function GreedyBestFirst(initialBoard, initialPieces, heuristicFunc):
    # 1. Inisialisasi
    MAX_COST ← boardWidth * boardHeight * 50
    startTime ← current time
    nodesVisited ← 0

    visitedStates ← empty set      # untuk mencegah eksplorasi ulang
    openList ← empty min-heap ordered by heuristic h(n)

    # Buat state awal
    initialState:
```

```

board ← initialBoard
pieces ← initialPieces
stateString ← stringify(initialBoard) # representasi unik papan
heuristic ← heuristicFunc(board, pieces) # h(initial)
cost ← 0 # g(initial) = 0
moves ← [] # belum ada langkah

openList.push(initialState) # enqueue state awal

# 2. Loop utama pencarian
while openList is not empty:
    nodesVisited ← nodesVisited + 1

    current ← openList.pop() # ambil node dengan h terkecil

    # skip jika sudah pernah dikunjungi
    if current.stateString in visitedStates:
        continue
    visitedStates.add(current.stateString)

    # jika ini goal, langsung return (tidak menjamin optimal g)
    if isSolved(current.pieces):
        elapsed ← current.time – startTime
        return SolutionResult(
            solved = true,
            moves = current.moves,
            nodesVisited = nodesVisited,
            executionTime = elapsed
        )

    # skip bila cost g sudah melebihi batas
    if current.cost > MAX_COST:
        continue

    # 3. Expand semua langkah valid
    for each move in getValidMoves(current.board, current.pieces):
        nextBoard, nextPieces ← applyMove(current.board, current.pieces, move)
        nextKey ← stringify(nextBoard)

        # skip jika sudah dikunjungi
        if nextKey in visitedStates:
            continue

        newCost ← current.cost + 1 # g naik 1 per langkah

```

```

newHeuristic ← heuristicFunc(nextBoard, nextPieces)

# hanya enqueue jika cost masih di bawah batas
if newCost ≤ MAX_COST:
    nextState:
        board ← nextBoard
        pieces ← nextPieces
        stateString ← nextKey
        heuristic ← newHeuristic
        cost ← newCost
        moves ← current.moves + [move]
        openList.push(nextState)

# 4. jika frontier kosong tanpa solusi
elapsed ← current time – startTime
return SolutionResult(
    solved = false,
    moves = [],
    nodesVisited = nodesVisited,
    executionTime = elapsed
)

```

2.4 Algoritma A*

Algoritma A* merupakan algoritma pencarian yang digunakan untuk menemukan jalur terpendek antara titik awal dan titik tujuan dalam sebuah graf berbobot. A* dirancang dengan pendekatan yang menggabungkan dua strategi pencarian yaitu Uniform Cost Search (UCS) dan Greedy Best-First Search. Fungsi evaluasi total dari algoritma ini adalah $f(n) = g(n) + h(n)$, di mana $g(n)$ adalah biaya sebenarnya dari titik awal ke simpul n, dan $h(n)$ adalah estimasi biaya dari simpul n ke tujuan akhir. Dengan memprioritaskan simpul berdasarkan nilai $f(n)$ yang paling kecil, A* mampu memperkirakan jalur terbaik secara efisien.

Kelebihan A* terletak pada penggunaan heuristic function $h(n)$, yang membantu algoritma fokus pada jalur-jalur yang menjanjikan tanpa harus mengeksplorasi semua kemungkinan secara menyeluruh. Jika $h(n)$ selalu lebih kecil atau sama dengan $h^*(n)$ (biaya

aktual dari n ke tujuan), maka heuristik tersebut disebut admissible, dan A* dijamin akan menemukan solusi optimal. Jika heuristik juga memenuhi sifat konsistensi (monotonicity), maka A* tidak hanya optimal tetapi juga lebih efisien dalam eksplorasi.

Secara teori, A* bersifat complete, yaitu menjamin solusi ditemukan jika ada, dan optimal, yaitu solusi yang ditemukan adalah yang paling efisien, namun memiliki kompleksitas waktu dan ruang yang tinggi, yakni eksponensial dalam kasus terburuk. Hal ini karena A* menyimpan semua simpul yang telah diperiksa dalam memori untuk menghindari duplikasi jalur.

Notasi Pseudocode:

```
function AStar(initialBoard, initialPieces, heuristicFunc):
    # 1. Inisialisasi
    MAX_COST ← boardWidth * boardHeight * 50
    startTime ← current time
    nodesVisited ← 0

    visitedStates ← empty set      # untuk mencegah eksplorasi ulang state yang sama
    openList ← empty min-heap ordered by f(n) = g(n) + h(n)

    # Buat state awal
    initialState:
        board ← initialBoard
        pieces ← initialPieces
        stateString ← stringify(initialBoard) # representasi unik papan
        cost ← 0          # g(start) = 0
        heuristic ← heuristicFunc(board, pieces) # h(start)
        f ← cost + heuristic      # f(start) = g + h
        moves ← []           # jalur kosong

        openList.push(initialState)      # enqueue state awal

    # 2. Loop utama pencarian
    while openList is not empty:
        nodesVisited ← nodesVisited + 1

        current ← openList.pop()      # ambil node dengan f terkecil

        # Skip jika sudah pernah diproses
```

```

if current.stateString in visitedStates:
    continue
visitedStates.add(current.stateString)

# Jika ini state goal, return hasil langsung
if isSolved(current.pieces):
    elapsed ← current.time – startTime
    return SolutionResult(
        solved = true,
        moves = current.moves,
        nodesVisited = nodesVisited,
        executionTime = elapsed
    )

# Skip jika g(n) sudah melebihi batas
if current.cost > MAX_COST:
    continue

# 3. Expand tetangga
for each move in getValidMoves(current.board, current.pieces):
    nextBoard, nextPieces ← applyMove(current.board, current.pieces, move)
    nextKey ← stringify(nextBoard)

    # Skip jika sudah diproses
    if nextKey in visitedStates:
        continue

    newCost ← current.cost + 1      # g(next) = g(current) + 1
    if newCost > MAX_COST:
        continue          # batas keamanan

    newHeuristic ← heuristicFunc(nextBoard, nextPieces) # h(next)
    newF ← newCost + newHeuristic      # f(next) = g + h
    newMoves ← current.moves + [move]

    nextState:
        board ← nextBoard
        pieces ← nextPieces
        stateString ← nextKey
        cost ← newCost
        heuristic ← newHeuristic
        f ← newF
        moves ← newMoves

```

```

openList.push(nextState)           # enqueue untuk eksplorasi selanjutnya

# 4. Jika frontier kosong dan tidak ditemukan solusi
elapsed ← current time – startTime
return SolutionResult(
    solved = false,
    moves = [],
    nodesVisited = nodesVisited,
    executionTime = elapsed
)

```

2.5 Algoritma Dijkstra

Algoritma Dijkstra merupakan metode pencarian jalur terpendek dari satu titik awal ke titik lain dalam graf berbobot, di mana semua bobot lintasan bernilai non-negatif. Algoritma ini termasuk ke dalam kategori uninformed search dan sangat mirip dengan Uniform-Cost Search (UCS). Namun, Dijkstra bekerja dengan prinsip ekspansi global terhadap seluruh node dalam graf hingga semua jarak minimum dari titik awal ke semua titik lainnya diketahui. Oleh karena itu, algoritma ini sangat cocok digunakan ketika tujuan tidak ditentukan sejak awal, atau ketika ingin menghitung semua jalur terpendek dari satu titik sumber ke seluruh node lainnya.

Pada dasarnya, algoritma Dijkstra menggunakan struktur data priority queue (biasanya dengan heap) untuk menyimpan node-node yang akan dievaluasi, dengan prioritas ditentukan oleh jarak kumulatif terkecil dari sumber. Algoritma dimulai dengan memberikan nilai jarak 0 pada titik awal dan tak hingga (∞) pada semua simpul lainnya. Kemudian, secara iteratif, node dengan jarak terkecil akan diekstraksi, dan jarak menuju semua tetangganya akan dihitung ulang jika ditemukan jalur yang lebih pendek melalui node tersebut. Jika iya, jaraknya diperbarui dalam antrian.

Berbeda dengan algoritma Greedy atau A*, Dijkstra tidak menggunakan fungsi heuristik. Fungsi evaluasi yang digunakan hanyalah $f(n) = g(n)$, yaitu biaya sebenarnya dari titik awal

ke simpul n, sama seperti UCS. Namun, seluruh graf dapat tetap dievaluasi sepenuhnya, bahkan jika tujuan sudah ditemukan, kecuali dioptimalkan.

Salah satu kelebihan utama algoritma Dijkstra adalah jaminan solusi optimal untuk graf dengan bobot non-negatif. Dijkstra juga lengkap, artinya selalu menemukan solusi jika ada. Namun demikian, Dijkstra cenderung kurang efisien secara waktu dan memori jika diterapkan pada graf yang sangat besar dan hanya butuh jalur menuju satu simpul tujuan, karena ia tidak memanfaatkan informasi heuristik untuk mempersempit pencarian.

Dari segi kompleksitas waktu, implementasi standar Dijkstra dengan array memiliki kompleksitas $O(V^2)$, sedangkan jika menggunakan min-heap (seperti binary heap) dan adjacency list, kompleksitasnya dapat diturunkan menjadi $O((V + E) \log V)$, di mana V adalah jumlah simpul dan E adalah jumlah sisi.

Notasi Pseudocode:

```
function Dijkstra(initialBoard, initialPieces):
    # 1. Inisialisasi
    MAX_COST ← boardWidth * boardHeight * 50
    startTime ← current time
    nodesVisited ← 0

    distanceMap ← empty map<string, number> # menyimpan jarak terpendek ke setiap state
    openList ← empty min-heap ordered by cost

    # Buat state awal
    initialState:
        board ← initialBoard
        pieces ← initialPieces
        stateString ← stringify(initialBoard) # representasi unik papan
        cost ← 0                                # g(start) = 0
        moves ← []                                # belum ada langkah

    openList.push(initialState)
    distanceMap[stateString] ← 0

    # 2. Loop utama pencarian
    while openList is not empty:
        nodesVisited ← nodesVisited + 1
```

```

current ← openList.pop()           # ambil node dengan cost terendah
currKey ← current.stateString

# Jika cost lebih besar daripada yang sudah dicatat, skip
if current.cost > distanceMap[currKey]:
    continue

# Jika ini goal, langsung return solusi
if isSolved(current.pieces):
    elapsed ← current.time – startTime
    return SolutionResult(
        solved = true,
        moves = current.moves,
        nodesVisited = nodesVisited,
        executionTime = elapsed
    )

# Skip jika cost sudah melebihi batas aman
if current.cost > MAX_COST:
    continue

# 3. Expand semua langkah valid
for each move in getValidMoves(current.board, current.pieces):
    nextBoard, nextPieces ← applyMove(current.board, current.pieces, move)
    nextKey ← stringify(nextBoard)
    newCost ← current.cost + 1      # setiap langkah menambah cost 1

    # Jika jalur ini lebih pendek dari sebelumnya
    if newCost < (distanceMap[nextKey] or Infinity):
        distanceMap[nextKey] ← newCost

    nextState:
        board ← nextBoard
        pieces ← nextPieces
        stateString ← nextKey
        cost ← newCost
        moves ← current.moves + [move]

    openList.push(nextState)

# 4. Jika openList habis tanpa menemukan solusi
elapsed ← current.time – startTime
return SolutionResult()

```

```
solved = false,  
moves = [],  
nodesVisited = nodesVisited,  
executionTime = elapsed  
)
```

2.6 Algoritma Fringe Search

Fringe Search adalah algoritma pencarian graf berbobot terarah yang menggabungkan efisiensi memori dari Iterative Deepening A* (IDA*) dengan performa A*. Seperti A*, algoritma ini menggunakan fungsi evaluasi total $f(n) = g(n) + h(n)$, di mana $g(n)$ adalah biaya aktual dari titik awal ke simpul n , dan $h(n)$ adalah estimasi admissible dari biaya tersisa menuju tujuan. Namun, alih-alih mempertahankan satu frontier besar seperti A*, Fringe Search bekerja dalam iterasi berulang dengan batas nilai f yang meningkat, sehingga hanya simpul dengan $f(n)$ di bawah atau sama dengan batas saat ini yang dikembangkan secara penuh.

Pada setiap iterasi, Fringe Search memelihara dua daftar:

1. Current Fringe, yaitu simpul-simpul yang akan diekspansi dalam batas f_limit sekarang.
2. Next Fringe, yang mengumpulkan simpul dengan $f(n)$ melebihi batas untuk dieksplorasi pada iterasi berikutnya setelah batas diperbesar.

Algoritma dimulai dengan menghitung nilai f_limit dari simpul awal dan memasukkannya ke dalam Current Fringe. Kemudian secara berulang, ia mengekstrak satu per satu simpul dari Current Fringe. Jika simpul tersebut menyelesaikan masalah, algoritma berakhir. Jika tidak, semua tetangganya dihasilkan dengan tiap tetangga yang tetap dalam batas f_limit ditambahkan kembali ke Current Fringe, mekanisme ini seringkali di bagian depannya menyerupai eksplorasi depth-first, sedangkan yang melebihi batas ditambahkan ke Next Fringe dan menentukan batas f minimum berikutnya. Setelah Current Fringe kosong,

batas dinaikkan ke nilai kecil berikutnya, dan Next Fringe menjadi Current Fringe untuk iterasi selanjutnya.

Keunggulan Fringe Search adalah penggunaan memori yang jauh lebih hemat dibanding A*, karena tidak perlu mempertahankan satu frontier tunggal yang membesar, hanya dua daftar berukuran relatif kecil per iterasi. Selain itu, dengan pola iterative-deepening pada nilai f, ia tetap terinformasi secara heuristik dan menjamin solusi optimal ketika heuristik admissible.

Dalam praktiknya, Fringe Search mengekspansi lebih sedikit simpul daripada IDA* tradisional (yang hanya mendasarkan pada $g + h$) dan memerlukan lebih sedikit memori daripada A*, sehingga cocok untuk teka-teki seperti Rush Hour dengan ruang status besar. Namun, kompleksitas waktunya tetap eksponensial dalam kedalaman solusi, yakni $O(b^d)$ dalam kasus terburuk, meskipun heuristik yang baik dapat memangkas cabang pencarian. Dari segi ruang, ia menggunakan memori $O(b^m)$ di mana m adalah kedalaman maksimum yang dicapai pada iterasi tertentu, jauh lebih kecil daripada $O(b^d)$ Frontier A*.

```
function FringeSearch(initialBoard, initialPieces, heuristicFunc):
    # 1. Inisialisasi
    MAX_COST ← boardWidth * boardHeight * 50
    startTime ← current time
    nodesVisited ← 0

    # Hybrid heuristic untuk blocking/combined
    useHybrid ← (heuristicFunc is blockingVehicles or combined)

    define getH(board, pieces):
        if useHybrid:
            # ambil estimasi konservatif antara Manhattan dan blocking
            return max(1, min(manhattan(board, pieces), heuristicFunc(board, pieces)))
        else:
            return heuristicFunc(board, pieces)

    gValues ← empty map<string, number> # g(n) terbaik per stateString

    # Buat state awal
    initialState:
```

```

board ← initialBoard
pieces ← initialPieces
stateString ← stringify(initialBoard)
cost ← 0
heuristic ← getH(initialBoard, initialPieces)
f ← cost + heuristic
moves ← []

gValues[stateString] ← 0

now ← [initialState] # states dengan f ≤ fLimit saat ini
later ← [] # states dengan f > fLimit
fLimit ← initialState.f

# 2. Loop utama
while now not empty or later not empty:
    # Jika semua state low-f sudah habis, naikkan fLimit
    if now is empty:
        if later is empty:
            break
        nextF ← min(state.f for state in later)
        now ← filter(later, state.f == nextF)
        later ← filter(later, state.f > nextF)
        fLimit ← nextF
    if fLimit ≥ MAX_COST:
        break

    # Ambil satu state untuk diexpand
    state ← now.shift()
    nodesVisited ← nodesVisited + 1

    # Jika ini goal, return
    if isSolved(state.pieces):
        elapsed ← current time – startTime
        return SolutionResult(
            solved = true,
            moves = state.moves,
            nodesVisited = nodesVisited,
            executionTime = elapsed
        )

    # Skip jika cost terlalu tinggi
    if state.cost ≥ MAX_COST:
        continue

```

```

# 3. Expand semua valid moves
for each move in getValidMoves(state.board, state.pieces):
    nextBoard, nextPieces ← applyMove(state.board, state.pieces, move)
    nextKey ← stringify(nextBoard)
    newG ← state.cost + 1

    # Prune jika g tidak lebih baik
    if gValues[nextKey] exists and gValues[nextKey] ≤ newG:
        continue
    gValues[nextKey] ← newG

    newH ← getH(nextBoard, nextPieces)
    newF ← newG + newH
    newMoves ← state.moves + [move]

    nextState:
        board ← nextBoard
        pieces ← nextPieces
        stateString ← nextKey
        cost ← newG
        heuristic ← newH
        f ← newF
        moves ← newMoves

    # Klasifikasikan ke now atau later
    if newF ≤ fLimit:
        now.unshift(nextState) # prioritaskan eksplorasi kedalaman
    else:
        later.push(nextState) # tunggu pass berikutnya

# 4. Gagal menemukan solusi
elapsed ← current time – startTime
return SolutionResult(
    solved = false,
    moves = [],
    nodesVisited = nodesVisited,
    executionTime = elapsed
)

```

BAB II

ANALISIS PEMECAHAN MASALAH

3.1 Analisis Penyelesaian Puzzle Rush Hour

Dalam menyelesaikan puzzle Rush Hour, langkah awal yang kami lakukan adalah memilih representasi state yang tepat, dengan merepresentasikan setiap konfigurasi papan sebagai string atau struktur data yang dapat memuat setiap posisi kendaraan (piece). Dengan cara tersebut, kami dapat membuat setiap operasi move untuk menciptakan state baru yang masih mudah dibedakan dan dibandingkan setiap pergerakannya. Setelah representasi konfigurasi papan sudah siap, kami menggunakan algoritma *pathfinding* (UCS, Greedy Best-First, A*, Dijkstra, atau Fringe) yang akan menelusuri ruang state tersebut demi menemukan urutan langkah yang membawa mobil utama keluar melalui pintu keluar.

Inti dari setiap strategi muncul dari cara masing-masing algoritma memprioritaskan state mana yang akan dieksplorasi lebih dahulu. Dengan menggunakan heuristik Manhattan distance, kita juga memusatkan perhatian pada seberapa jauh secara garis lurus posisi ujung mobil utama ke pintu keluar. Pendekatan ini sederhana dan cepat dihitung, namun tidak memperhitungkan adanya kendaraan lain yang menghalangi jalan. Sebagai hasilnya, Greedy Best-First Search dengan heuristik ini cenderung cepat “mendekati” solusi tetapi mudah terjebak pada konfigurasi macet yang perlu gerakan memutar-balikan banyak kendaraan.

Untuk mengatasinya, kita menambahkan heuristik blocking vehicles, yaitu menghitung berapa banyak kendaraan yang langsung menghalangi jalur lurus si mobil utama. Dengan menekankan pengurangan jumlah penghalang sebelum mendekat pada Manhattan distance, A*, Greedy, atau Fringe dapat dipaksa untuk lebih sering memindahkan kendaraan samping yang sesungguhnya krusial untuk membuka jalan utama. Heuristik ini lebih mahal dihitung karena perlu memeriksa setiap sel sepanjang jalur, tetapi signifikan menurunkan jumlah node yang dieksplorasi pada puzzle berskala menengah hingga besar.

Terakhir, kombinasi kedua heuristik, yaitu combined memberi keseimbangan antara jarak fisik (Manhattan) dan kesulitan orientasi (blocking). Pendekatan gabungan ini sangat berguna

ketika satu heuristik saja terlalu optimistik (terlalu rendah) atau terlalu pesimistik (terlalu tinggi), sehingga A*, Greedy, atau Fringe dengan heuristik ini mampu mengelola trade-off antara “seberapa dekat” dan “seberapa bebas” jalur utama.

Dengan strategi heuristik di atas, algoritma pencarian akan lebih efisien mengarahkan eksplorasi ke state yang benar-benar menjanjikan dengan membuka jalur utama, kemudian memajukan mobil utama ke pintu keluar. Pada akhirnya, pemilihan heuristik yang informatif dan perancangan state representation yang efisien menjadi dua pilar utama dalam menyelesaikan puzzle Rush Hour dengan performa yang baik.

3.2 Fungsi Evaluasi $f(n)$ dan $g(n)$

Dalam semua algoritma pencarian jalur, digunakan suatu fungsi evaluasi $f(n)$ untuk menentukan prioritas eksplorasi node. Fungsi $g(n)$ menyatakan biaya dari node awal hingga node saat ini. Perbedaan utama antar algoritma terletak pada bagaimana $f(n)$ didefinisikan:

1. UCS: $f(n) = g(n)$. UCS hanya mempertimbangkan biaya kumulatif dari awal hingga node saat ini, tanpa informasi estimasi ke tujuan.
2. Greedy Best-First Search: $f(n) = h(n)$. $g(n)$ tidak digunakan. Algoritma ini hanya menggunakan $h(n)$, yaitu estimasi biaya dari node saat ini ke tujuan. Karena mengabaikan $g(n)$, jalur yang dilalui bisa jadi sangat mahal meskipun tampak menjanjikan secara heuristik.
3. A*: $f(n) = g(n) + h(n)$. Kombinasi antara biaya nyata dari awal ($g(n)$) dan estimasi ke tujuan ($h(n)$). A* menyeimbangkan eksplorasi jalur yang sudah ditempuh dengan prediksi sisa perjalanan.
4. Dijkstra: $f(n) = g(n)$. Tidak menggunakan fungsi heuristik, hanya memperhitungkan jarak aktual dari awal ke node n .
5. Fringe: $f(n) = g(n) + h(n)$. Sama seperti A* namun dengan konsumsi memori yang lebih rendah dan tanpa heap besar.

Dengan definisi tersebut, terlihat bahwa UCS, A*, Dijkstra, Fringe menggunakan informasi biaya nyata ($g(n)$), sementara Greedy tidak. Secara umum, $g(n)$ adalah biaya riil

dari start ke node n, sedangkan $f(n)$ adalah fungsi evaluasi yang dipakai untuk memprioritaskan node dalam frontier.

3.3 Keadmissible-an Heuristik pada Algoritma A*

Heuristik pada algoritma A* disebut admissible jika tidak pernah melebih-lebihkan biaya sebenarnya dari node saat ini ke node tujuan (optimistic).

$$h(n) \leq h^*(n) \text{ untuk semua node } n,$$

di mana $h(n)$ adalah estimasi biaya dari node n ke tujuan dan $h^*(n)$ adalah biaya sebenarnya dari node n ke tujuan. Dalam pengerjaan tugas kecil kami, terdapat tiga heuristik yang kami gunakan, yaitu manhattan, blockingVehicles, dan combined. Heuristik manhattan yang mengukur jarak dari kendaraan utama ke pintu keluar tanpa mempertimbangkan penghalang tergolong admissible karena selalu menghasilkan estimasi yang tidak lebih besar dari biaya riil. Heuristik blockingVehicles, yang menghitung jumlah kendaraan yang menghalangi jalur keluar, juga admissible karena hanya menghitung jumlah penghalang, bukan biaya pergeserannya. Karena setiap kendaraan penghalang minimal memerlukan satu gerakan untuk digeser, nilai $h(n) = \text{jumlah blocking}$ tidak akan melebih-lebihkan jumlah langkah sebenarnya yang dibutuhkan untuk menghilangkan semua penghalang tersebut. Dengan kata lain, untuk setiap kendaraan penghalang, pasti diperlukan setidaknya satu langkah, tidak mungkin memerlukan nol langkah, sehingga $h_{\text{blocking}}(n) \leq h^*(n)$. Namun, jika kita menggunakan heuristik combined ($h_{\text{combined}}(n) = \text{Manhattan}(n) + 3 \times \text{blockingVehicles}(n)$) pengali tiga pada blockingVehicles bisa membuat $h_{\text{combined}}(n)$ melebihi biaya aktual, misalnya saat ada dua kendaraan penghalang yang sebenarnya hanya membutuhkan dua langkah total, tetapi $3 \times 2 = 6 > 2$, sehingga melanggar admissibility.

3.4 Perbandingan UCS dan BFS pada Penyelesaian Rush Hour

Dalam konteks penyelesaian Rush Hour, di mana setiap pergerakan kendaraan memiliki biaya yang seragam, algoritma Uniform Cost Search (UCS) dan Breadth-First Search (BFS) akan menghasilkan path solusi yang memiliki panjang sama dan akan mengeksplorasi node berdasarkan jumlah langkah yang meningkat secara bertahap. Hal ini karena UCS akan

memperluas simpul berdasarkan total biaya dari awal ($g(n)$), yang nilainya identik dengan kedalaman simpul pada BFS ketika semua langkah bernilai 1.

Dengan demikian, path yang dihasilkan UCS dan BFS akan sama dalam hal jumlah langkah dan optimalitas solusi. Selain itu, karena urutan perluasan dalam kedua algoritma mengikuti peningkatan nilai $g(n)$, maka urutan simpul yang dibangkitkan pun akan sama jika tidak ada tie-breaking yang membedakan antara dua simpul dengan biaya sama.

Namun, dalam implementasi praktis, UCS menggunakan priority queue yang bisa memiliki kebijakan tie-breaking berbeda, seperti urutan berdasarkan posisi dalam memori atau urutan penyisipan, sehingga urutan pengembangan simpul bisa saja berbeda dalam detailnya, meskipun kedalaman dan hasil akhir tetap sama.

3.5 Efisiensi A* dibanding UCS pada Rush Hour

Secara teoritis, A* akan lebih efisien daripada UCS dalam menyelesaikan permainan Rush Hour karena A* memanfaatkan informasi tambahan dari heuristik untuk menuntun pencarian langsung menuju solusi, sedangkan UCS hanya mengeksplorasi berdasarkan biaya sejauh ini tanpa panduan ke tujuan.

Dengan heuristik yang admissible dan konsisten, misalnya menghitung jarak Manhattan atau jumlah kendaraan penghalang, A* dapat memangkas sebagian besar cabang pencarian yang jelas-jelas tidak menjanjikan, sehingga jumlah simpul yang dibangkitkan jauh lebih sedikit dibandingkan UCS yang harus memeriksa semua jalur dengan biaya yang sama hingga kedalaman tertentu.

Meski keduanya sama-sama menjamin solusi optimal ketika heuristik A* admissible, A* lebih cepat menemukan solusi karena setiap pengembangan simpul dihitung berdasarkan estimasi total biaya $g(n)+h(n)$, bukan semata $g(n)$. Akibatnya, ruang pencarian yang dikerjakan A* untuk puzzle berukuran besar seperti Rush Hour secara signifikan lebih kecil, membuatnya secara teoritis lebih efisien dari segi jumlah node yang diperiksa dan waktu eksekusi.

3.6 Apakah Greedy Best-First Search Menjamin Solusi Optimal?

Secara teoritis, Greedy Best-First Search tidak pernah menjamin bahwa solusi yang ditemukannya adalah yang paling optimal dalam jumlah langkah saat menyelesaikan puzzle Rush Hour. Hal ini disebabkan karena algoritma ini hanya mengutamakan nilai heuristik $h(n)$, yakni sejauh apa posisi kendaraan utama tampak mendekati pintu keluar, tanpa memperhitungkan biaya yang sudah ditempuh ($g(n)$). Akibatnya, ia bisa saja memilih urutan gerakan yang “terlihat” paling cepat mencapai goal menurut heuristik, tetapi sebenarnya membutuhkan lebih banyak langkah secara keseluruhan.

Karena Greedy BFS tidak memadukan estimasi $h(n)$ dengan akumulasi biaya sebenarnya, dan tidak memiliki mekanisme untuk merevisi pilihan jalur berdasarkan total cost, ia rentan terjebak pada jalur sub-optimal atau plateau lokal yang memblokir jalan menuju solusi terbaik. Oleh karena itu, meski Greedy Best-First seringkali cepat menemukan suatu solusi, ia tidak dapat dijadikan jaminan untuk menemukan solusi dengan jumlah langkah minimum. Misalnya, jika dua konfigurasi berbeda sama-sama menghasilkan $h=3$, Greedy dapat terus berganti antara keduanya, tanpa tahu mana yang sudah dicapai dengan lebih sedikit langkah sejauh ini, hingga menemukan dead-end dan harus backtrack secara ekstensif.

BAB III

SOURCE PROGRAM

4.1 Type

Type	Tampilan Kode
Direction	<pre>••• export type Direction = "atas" "bawah" "kiri" "kanan";</pre>
PieceOrientation	<pre>••• export type PieceOrientation = "horizontal" "vertical" "unknown";</pre>
Position	<pre>••• export interface Position { row: number; col: number; }</pre>
PieceInfo	<pre>••• export interface PieceInfo { positions: Position[]; symbol: string; size?: number; orientation?: PieceOrientation; primaryPosition?: Position; isPrimary?: boolean; isExit?: boolean; }</pre>

PiecesMap	<pre>● ● ● export type PiecesMap = Record<string, PieceInfo>;</pre>
BoardConfig	<pre>● ● ● export interface BoardConfig { dimensions: { A: number; B: number }; numPieces: number; boardConfig: string[]; finishLocation: Position; }</pre>
Move	<pre>● ● ● export interface Move { piece: string; direction: Direction; }</pre>
SolutionResult	<pre>● ● ● export interface SolutionResult { solved: boolean; moves: Move[]; nodesVisited: number; executionTime: number; }</pre>

Stats	<pre> ... export interface Stats { nodesVisited: number; executionTime: string; moves: number; } </pre>
Algorithm	<pre> ... export type Algorithm = "greedy" "ucs" "astar" "dijkstra" "fringe"; </pre>
Heuristic	<pre> ... export type Heuristic = "manhattan" "blockingVehicles" "combined"; </pre>

4.2 Algoritma

Algoritma	Tampilan Kode

UCS

```
...
import applyMove from "../helpers/applyMove";
import getBoardStateString from "../helpers/getBoardStateString";
import getValidMoves from "../helpers/getValidMoves";
import isSolved from "../helpers/isSolved";
import { PriorityQueue } from "../priorityQueue";
import type { Move, PiecesMap, SolutionResult } from "../types";

interface SearchState {
  board: string[][];
  pieces: PiecesMap;
  stateString: string;
  cost: number;
  moves: Move[];
}

interface Solution {
  cost: number;
  moves: Move[];
}

const ucs = (initialBoard: string[][], initialPieces: PiecesMap): SolutionResult => {
  const MAX_COST = initialBoard.length * initialBoard[0].length * 58;
  const start = performance.now();
  const visitedStates = new Set<string>();
  const openList = new PriorityQueue<SearchState>((a, b) => a.cost - b.cost);
  let nodesVisited = 0;

  const initialState: SearchState = {
    board: initialBoard,
    pieces: initialPieces,
    stateString: getBoardStateString(initialBoard),
    cost: 0,
    moves: [],
  };

  openList.push(initialState);

  let bestSolution: Solution = {
    cost: Infinity,
    moves: [],
  };

  while (!openList.isEmpty()) {
    nodesVisited++;
    const currentState = openList.pop()!;

    if (visitedStates.has(currentState.stateString)) continue;
    visitedStates.add(currentState.stateString);

    if (isSolved(currentState.pieces)) {
      if (!bestSolution || currentState.cost < bestSolution.cost) {
        bestSolution = {
          cost: currentState.cost,
          moves: [...currentState.moves],
        };
      }
      const end = performance.now();
      return {
        solved: true,
        moves: currentState.moves,
        nodesVisited,
        executionTime: end - start,
      };
    }

    if (currentState.cost > MAX_COST) continue;

    const validMoves = getValidMoves(currentState.board, currentState.pieces);

    for (const move of validMoves) {
      const result = applyMove(currentState.board, currentState.pieces, move);
      const newStateString = getBoardStateString(result.board);

      if (visitedStates.has(newStateString)) continue;

      const newState: SearchState = {
        board: result.board,
        pieces: result.pieces,
        stateString: newStateString,
        cost: currentState.cost + 1,
        moves: [...currentState.moves, move],
      };

      openList.push(newState);
    }
  }

  const end = performance.now();
  return {
    solved: bestSolution.cost !== Infinity,
    moves: bestSolution.moves,
    nodesVisited,
    executionTime: end - start,
  };
};

export default ucs;
```

Greedy

```
...
import applyMove from "../helpers/applyMove";
import getBoardStateString from "../helpers/getBoardStateString";
import getValidMoves from "../helpers/getValidMoves";
import isSolved from "../helpers/isSolved";
import { PriorityQueue } from "../priorityQueue";
import type { Move, PiecesMap, SolutionResult } from "../types";

interface SearchState {
  board: string[][];
  pieces: PiecesMap;
  stateString: string;
  heuristic: number;
  moves: Move[];
  cost: number;
}

const greedy = (
  initialBoard: string[][][],
  initialPieces: PiecesMap,
  heuristicFunc: (board: string[][][], pieces: PiecesMap) => number
): SolutionResult => {
  const MAX_COST = initialBoard.length * initialBoard[0].length * 50;
  const start = performance.now();
  const visitedStates = new Set<string>();
  const openList = new PriorityQueue<SearchState>((a, b) => a.heuristic - b.heuristic);
  let nodesVisited = 0;

  const initialState: SearchState = {
    board: initialBoard,
    pieces: initialPieces,
    stateString: getBoardStateString(initialBoard),
    heuristic: heuristicFunc(initialBoard, initialPieces),
    moves: [],
    cost: 0,
  };

  openList.push(initialState);

  while (!openList.isEmpty()) {
    nodesVisited++;
    const currentState = openList.pop()!;

    if (visitedStates.has(currentState.stateString)) continue;
    visitedStates.add(currentState.stateString);

    if (isSolved(currentState.pieces)) {
      const end = performance.now();
      return {
        solved: true,
        moves: currentState.moves,
        nodesVisited,
        executionTime: end - start,
      };
    }

    if (currentState.cost > MAX_COST) continue;

    const validMoves = getValidMoves(currentState.board, currentState.pieces);

    for (const move of validMoves) {
      const { board: newBoard, pieces: newPieces } = applyMove(currentState.board, currentState.pieces, move);
      const newStateString = getBoardStateString(newBoard);

      if (visitedStates.has(newStateString)) continue;

      const newHeuristic = heuristicFunc(newBoard, newPieces);
      const newCost = currentState.cost + 1;

      if (newCost <= MAX_COST) {
        const newState: SearchState = {
          board: newBoard,
          pieces: newPieces,
          stateString: newStateString,
          heuristic: newHeuristic,
          moves: [...currentState.moves, move],
          cost: newCost,
        };
        openList.push(newState);
      }
    }
  }

  const end = performance.now();
  return {
    solved: false,
    moves: [],
    nodesVisited,
    executionTime: end - start,
  };
};

export default greedy;
```

A*

```
import applyMove from "../helpers/applyMove";
import getBoardStateString from "../helpers/getBoardStateString";
import getValidMoves from "../helpers/getValidMoves";
import isSolved from "../helpers/isSolved";
import { PriorityQueue } from "./priorityQueue";
import type { Move, PiecesMap, SolutionResult } from "../types";

interface SearchState {
  board: string[][];
  pieces: PiecesMap;
  stateString: string;
  cost: number;
  heuristic: number;
  f: number;
  moves: Move[];
}

const astar = (
  initialBoard: string[][][],
  initialPieces: PiecesMap,
  heuristicFunc: (board: string[][][], pieces: PiecesMap) => number
): SolutionResult => {
  const MAX_COST = initialBoard.length * initialBoard[0].length * 50;
  const start = performance.now();
  const visitedStates = new Set<string>();
  const openList = new PriorityQueue<SearchState>((a, b) => a.f - b.f);
  let nodesVisited = 0;

  const initialState: SearchState = {
    board: initialBoard,
    pieces: initialPieces,
    stateString: getBoardStateString(initialBoard),
    cost: 0,
    heuristic: heuristicFunc(initialBoard, initialPieces),
    f: heuristicFunc(initialBoard, initialPieces),
    moves: []
  };

  openList.push(initialState);

  while (!openList.isEmpty()) {
    nodesVisited++;
    const currentState = openList.pop()!;
    if (visitedStates.has(currentState.stateString)) continue;
    visitedStates.add(currentState.stateString);

    if (isSolved(currentState.pieces)) {
      const end = performance.now();
      return {
        solved: true,
        moves: currentState.moves,
        nodesVisited,
        executionTime: end - start,
      };
    }

    if (currentState.cost > MAX_COST) continue;

    const validMoves = getValidMoves(currentState.board, currentState.pieces);

    for (const move of validMoves) {
      const { board: newBoard, pieces: newPieces } = applyMove(
        currentState.board,
        currentState.pieces,
        move
      );
      const newStateString = getBoardStateString(newBoard);
      if (visitedStates.has(newStateString)) continue;

      const newHeuristic = heuristicFunc(newBoard, newPieces);
      const newCost = currentState.cost + 1;
      if (newCost > MAX_COST) continue;

      const newState: SearchState = {
        board: newBoard,
        pieces: newPieces,
        stateString: newStateString,
        cost: newCost,
        heuristic: newHeuristic,
        f: newCost + newHeuristic,
        moves: [...currentState.moves, move],
      };

      openList.push(newState);
    }
  }

  const end = performance.now();
  return {
    solved: false,
    moves: [],
    nodesVisited,
    executionTime: end - start,
  };
};

export default astar;
```

Dijkstra

```
...
import applyMove from "../helpers/applyMove";
import getBoardStateString from "../helpers/getBoardStateString";
import getValidMoves from "../helpers/getValidMoves";
import isSolved from "../helpers/isSolved";
import { PriorityQueue } from "../priorityQueue";
import type { Move, PiecesMap, SolutionResult } from "../types";

interface SearchState {
  board: string[10];
  pieces: PiecesMap;
  stateString: string;
  cost: number;
  moves: Move[];
}

const dijkstra = (initialBoard: string[][], initialPieces: PiecesMap): SolutionResult => {
  const MAX_COST = initialBoard.length * initialBoard[0].length * 50;
  const start = performance.now();
  const distanceMap = new Map<string, number>();
  const queue = new PriorityQueue<SearchState>((a, b) => a.cost - b.cost);
  let nodesVisited = 0;

  const initialState: SearchState = {
    board: initialBoard,
    pieces: initialPieces,
    stateString: getBoardStateString(initialBoard),
    cost: 0,
    moves: []
  };

  queue.push(initialState);
  distanceMap.set(initialState.stateString, 0);

  while (!queue.isEmpty()) {
    nodesVisited++;
    const currentState = queue.pop()!;
    const currentStateString = currentState.stateString;

    if (currentState.cost > (distanceMap.get(currentStateString) || Infinity)) {
      continue;
    }

    if (isSolved(currentState.pieces)) {
      const end = performance.now();
      return {
        solved: true,
        moves: currentState.moves,
        nodesVisited,
        executionTime: end - start,
      };
    }

    if (currentState.cost > MAX_COST) {
      continue;
    }

    const validMoves = getValidMoves(currentState.board, currentState.pieces);

    for (const move of validMoves) {
      const { board: newBoard, pieces: newPieces } = applyMove(currentState.board,
        currentState.pieces, move);
      const newStateString = getBoardStateString(newBoard);
      const newCost = currentState.cost + 1;

      if (newCost < (distanceMap.get(newStateString) || Infinity)) {
        distanceMap.set(newStateString, newCost);
        const newState: SearchState = {
          board: newBoard,
          pieces: newPieces,
          stateString: newStateString,
          cost: newCost,
          moves: [...currentState.moves, move],
        };
        queue.push(newState);
      }
    }
  }

  const end = performance.now();
  return {
    solved: false,
    moves: [],
    nodesVisited,
    executionTime: end - start,
  };
};

export default dijkstra;
```

Fringe

```
● ● ●

import applyMove from "../helpers/applyMove";
import getBoardStateString from "../helpers/getBoardStateString";
import getValidMoves from "../helpers/getValidMoves";
import isSolved from "../helpers/isSolved";
import { heuristics } from "../heuristics";
import type { Move, PiecesMap, SolutionResult } from "../../types";

interface SearchState {
    board: string[][];
    pieces: PiecesMap;
    stateString: string;
    cost: number;
    heuristic: number;
    f: number;
    moves: Move[];
}

const fringe = [
    initialBoard: string[][],
    initialPieces: PiecesMap,
    heuristicFunc: (board: string[][], pieces: PiecesMap) => number
]: SolutionResult => {
    const MAX_COST = initialBoard.length * initialBoard[0].length * 50;
    const start = performance.now();
    let nodesVisited = 0;

    const useHybridHeuristic = heuristicFunc === heuristics.blockingVehicles || heuristicFunc === heuristics.combined;

    const getF = (board: string[][], pieces: PiecesMap) => {
        if (useHybridHeuristic) {
            return Math.max(1, Math.min(heuristics.manhattan(board, pieces), heuristicFunc(board, pieces)));
        }
        return heuristicFunc(board, pieces);
    };

    const qValues = new Map<string, number>();
    const initialState: SearchState = {
        board: initialBoard,
        pieces: initialPieces,
        stateString: getBoardStateString(initialBoard),
        cost: 0,
        heuristic: getHeuristic(initialBoard, initialPieces),
        f: getF(initialBoard, initialPieces),
        moves: []
    };
    qValues.set(initialState.stateString, 0);

    let now: SearchState[] = [initialState];
    let later: SearchState[] = [];
    let fLimit = initialState.f;

    while (now.length > 0 || later.length > 0) {
        if (now.length === 0) {
            if (later.length === 0) break;
            let nextF = Infinity;
            for (const state of later) {
                nextF = Math.min(nextF, state.f);
            }
            now = later.filter(state => state.f === nextF);
            later = later.filter(state => state.f > nextF);
        }

        fLimit = nextF;
        if (fLimit >= MAX_COST) break;
    }

    const state = now.shift();
    nodesVisited++;

    if (isSolved(state.pieces)) {
        const end = performance.now();
        return {
            solved: true,
            moves: state.moves,
            nodesVisited,
            executionTime: end - start,
            board: state.board
        };
    }

    if (state.cost === MAX_COST) continue;

    const validMoves = getValidMoves(state.board, state.pieces);
    for (const move of validMoves) {
        const newBoard: string[][] = applyMove(state.board, state.pieces, move);
        const newStateString = getBoardStateString(newBoard);
        const newCost = state.cost + 1;

        const existingG = qValues.get(newStateString);
        if (existingG === undefined || existingG < newCost) continue;
        qValues.set(newStateString, newCost);

        const newH = getHeuristic(newStateString, newCost);
        const newF = newCost + newH;
        const newState: SearchState = {
            board: newBoard,
            pieces: newPieces,
            stateString: newStateString,
            cost: newCost,
            heuristic: newH,
            f: newF,
            moves: [...state.moves, move]
        };

        if (newF <= fLimit) {
            now.unshift(newState);
        } else {
            later.push(newState);
        }
    }

    const end = performance.now();
    return {
        solved: false,
        moves: [],
        nodesVisited,
        executionTime: end - start,
        board: state.board
    };
};

export default fringe;
```

4.3 Lainnya

Lainnya	Tampilan Kode
PriorityQueue	<pre>... export class PriorityQueue<T> { private items: T[] = []; private readonly comparator: (a: T, b: T) => number; constructor(comparator: (a: T, b: T) => number) { this.comparator = comparator; } push(item: T): void { let index = this.items.length; this.items.push(item); while (index > 0) { const parentIndex = Math.floor((index - 1) / 2); if (this.comparator(this.items[parentIndex], this.items[index]) <= 0) break; [this.items[parentIndex], this.items[index]] = [this.items[index], this.items[parentIndex]]; index = parentIndex; } } pop(): T undefined { if (this.isEmpty()) return undefined; const result = this.items[0]; const last = this.items.pop()!; if (this.items.length > 0) { this.items[0] = last; this.heapify(0); } return result; } isEmpty(): boolean { return this.items.length === 0; } size(): number { return this.items.length; } private heapify(index: number): void { const left = 2 * index + 1; const right = 2 * index + 2; let smallest = index; if (left < this.items.length && this.comparator(this.items[left], this.items[smallest]) < 0) { smallest = left; } if (right < this.items.length && this.comparator(this.items[right], this.items[smallest]) < 0) { smallest = right; } if (smallest !== index) { [this.items[index], this.items[smallest]] = [this.items[smallest], this.items[index]]; this.heapify(smallest); } } }</pre>

Heuristics

```
import type { Heuristic, PiecesMap } from "./types";

export const heuristics: Record<Heuristic, (board: string[][], pieces: PiecesMap) => number> = {
    manhattan: (_board, pieces) => {
        const primaryPiece = Object.values(pieces).find((p) => p.isPrimary)!;
        const exitPos = pieces["K"].positions[0];

        if (primaryPiece.orientation === "horizontal") {
            if (exitPos.col < primaryPiece.positions[0].col) {
                const leftPos = primaryPiece.positions[0];
                return Math.abs(leftPos.row - exitPos.row) + Math.abs(leftPos.col - exitPos.col - 1);
            } else {
                const rightPos = primaryPiece.positions[primaryPiece.positions.length - 1];
                return Math.abs(rightPos.row - exitPos.row) + Math.abs(rightPos.col + 1 - exitPos.col);
            }
        } else {
            const distances = primaryPiece.positions.map((pos) => Math.abs(pos.row - exitPos.row) + Math.abs(pos.col - exitPos.col));
            return Math.min(...distances);
        }
    },
    blockingVehicles: (board, pieces) => {
        const primaryPiece = Object.values(pieces).find((p) => p.isPrimary)!;
        const exitPos = pieces["K"].positions[0];

        if (primaryPiece.orientation === "horizontal") {
            const row = primaryPiece.positions[0].row;

            if (exitPos.col < primaryPiece.positions[0].col) {
                const leftmostCol = primaryPiece.positions[0].col;
                let count = 0;
                for (let col = exitPos.col + 1; col < leftmostCol; col++) {
                    if (board[row][col] !== "." && board[row][col] !== "K") {
                        count++;
                    }
                }
                return count;
            } else {
                const rightmostCol = primaryPiece.positions[primaryPiece.positions.length - 1].col;
                let count = 0;
                for (let col = rightmostCol + 1; col <= exitPos.col; col++) {
                    if (board[row][col] !== "." && board[row][col] !== "K") {
                        count++;
                    }
                }
                return count;
            }
        }

        const col = primaryPiece.positions[0].col;
        if (exitPos.row < primaryPiece.positions[0].row) {
            const topmostRow = primaryPiece.positions[0].row;
            let count = 0;
            for (let row = exitPos.row + 1; row < topmostRow; row++) {
                if (board[row][col] !== "." && board[row][col] !== "K") {
                    count++;
                }
            }
            return count;
        } else {
            const bottommostRow = primaryPiece.positions[primaryPiece.positions.length - 1].row;
            let count = 0;
            for (let row = bottommostRow + 1; row <= exitPos.row; row++) {
                if (board[row][col] !== "." && board[row][col] !== "K") {
                    count++;
                }
            }
            return count;
        }
    },
    combined: (board, pieces) => {
        return heuristics.manhattan(board, pieces) + 3 * heuristics.blockingVehicles(board, pieces);
    },
};
```

Constant

```
import type { Direction } from "./types";

export const DIRECTIONS: Record<"UP" | "DOWN" | "LEFT" | "RIGHT", Direction> = {
    UP: "atas",
    DOWN: "bawah",
    LEFT: "kiri",
    RIGHT: "kanan",
};

export const pieceColors: Record<string, string> = {
    A: "bg-blue-500",
    B: "bg-purple-500",
    C: "bg-pink-500",
    D: "bg-indigo-500",
    E: "bg-teal-500",
    F: "bg-cyan-500",
    G: "bg-orange-500",
    H: "bg-lime-500",
    I: "bg-amber-500",
    J: "bg-emerald-500",
    L: "bg-rose-500",
    M: "bg-sky-500",
    N: "bg-violet-500",
    O: "bg-fuchsia-500",
};
```

ParseInputFile

IsSolved

```
import type { PiecesMap } from "../types";

const isSolved = (pieces: PiecesMap): boolean => {
  const primaryPiece = Object.values(pieces).find((p) => p.isPrimary);
  if (!primaryPiece) return false;
  const exitPiece = pieces["K"];
  if (!exitPiece) return false;
  const exitPos = exitPiece.positions[0];

  if (primaryPiece.orientation === "horizontal") {
    const leftmostPos = primaryPiece.positions[0];
    if (exitPos.col < leftmostPos.col) {
      return leftmostPos.row === exitPos.row && leftmostPos.col === exitPos.col + 1;
    } else if (exitPos.col > primaryPiece.positions[primaryPiece.positions.length - 1].col) {
      const rightmostPos = primaryPiece.positions[primaryPiece.positions.length - 1];
      return rightmostPos.row === exitPos.row && rightmostPos.col + 1 === exitPos.col;
    }
  } else if (primaryPiece.orientation === "vertical") {
    const topmostPos = primaryPiece.positions[0];
    const bottommostPos = primaryPiece.positions[primaryPiece.positions.length - 1];
    if (exitPos.row < topmostPos.row) {
      return topmostPos.col === exitPos.col && topmostPos.row === exitPos.row + 1;
    } else if (exitPos.row > bottommostPos.row) {
      return bottommostPos.col === exitPos.col && bottommostPos.row + 1 === exitPos.row;
    }
  }

  return primaryPiece.positions.some((pos) => pos.row === exitPos.row && pos.col === exitPos.col);
};

export default isSolved;
```

GetValidMoves

```

    const getValidMoves = (piece: string[], pieces: PiecesMap): Move[] => {
      const moves: Move[] = [];
      const boardHeight = board[0].length;
      const boardWidth = board[0].length;

      // Find exit location
      const exitRow = pieces[KEY];
      const exitCol = pieces[KEY].split(",");
      const exitRowIndex = exitRow.indexOf(KEY);
      const exitColIndex = exitCol.indexOf(KEY);
      const exitRowHeight = exitRow[exitRowIndex] === KEY ? boardHeight : exitRow[exitRowIndex] - 1;
      const exitColWidth = exitCol[exitColIndex] === KEY ? boardWidth : exitCol[exitColIndex] + 1;

      Object.keys(piece).forEach((pieceKey) => {
        const [row, col] = piece[pieceKey];
        const orientation = piece[pieceKey][0];
        const symbol = piece[pieceKey][1];
        if (orientation === "horizontal") {
          const leftPosIndex = piece.positions[0];
          const rightPosIndex = piece.positions[1];

          if (leftPosIndex === exitRowIndex) {
            let stepLeft = 1;
            let canMove = true;

            while (canMove && leftPosIndex < exitRowIndex) {
              const leftCell = board[leftPosIndex][row][rightPosIndex.col - stepLeft];
              if (leftCell === ".") {
                stepLeft++;
                moves.push({
                  row: row,
                  col: col,
                  piece: symbol,
                  direction: DIRECTIONS.LEFT,
                  steps: stepLeft,
                });
              } else if (leftCell === "K" && piece.isPrimary) {
                stepLeft++;
                moves.push({
                  row: row,
                  col: col,
                  piece: symbol,
                  direction: DIRECTIONS.LEFT,
                  steps: stepLeft,
                });
                canMove = false;
              } else {
                canMove = false;
              }
            }
          }

          // Check right moves
          const rightPosIndex = piece.positions[1];
          const rightRowIndex = rightPosIndex === exitRowIndex ? exitRowIndex : rightPosIndex.col + 1;
          const rightCell = board[rightRowIndex][row][rightPosIndex.col + stepRight + 1];
          if (rightCell === ".") {
            let stepRight = 1;
            let canMove = true;

            while (canMove && rightRowIndex < exitRowIndex) {
              const rightCell = board[rightRowIndex][row][rightPosIndex.col + stepRight];
              if (rightCell === ".") {
                stepRight++;
                moves.push({
                  row: row,
                  col: col,
                  piece: symbol,
                  direction: DIRECTIONS.RIGHT,
                  steps: stepRight,
                });
              } else if (rightCell === "K" && piece.isPrimary) {
                stepRight++;
                moves.push({
                  row: row,
                  col: col,
                  piece: symbol,
                  direction: DIRECTIONS.RIGHT,
                  steps: stepRight,
                });
                canMove = false;
              } else {
                canMove = false;
              }
            }
          }
        } else if (orientation === "vertical") {
          const topPosIndex = piece.positions[0];
          const bottomPosIndex = piece.positions[1];

          if (topPosIndex === exitColIndex) {
            let stepUp = 1;
            let canMove = true;

            while (canMove && topPosIndex < exitColIndex) {
              const topCell = board[topPosIndex][row][topPosIndex.col];
              if (topCell === ".") {
                stepUp++;
                moves.push({
                  row: row,
                  col: col,
                  piece: symbol,
                  direction: DIRECTIONS.UP,
                  steps: stepUp,
                });
              } else if (topCell === "K" && piece.isPrimary) {
                stepUp++;
                moves.push({
                  row: row,
                  col: col,
                  piece: symbol,
                  direction: DIRECTIONS.UP,
                  steps: stepUp,
                });
                canMove = false;
              } else {
                canMove = false;
              }
            }
          }

          // Check down moves
          const bottomPosIndex = piece.positions[1];
          const topPosIndex = piece.positions[0];
          const bottomRowIndex = bottomPosIndex === exitColIndex ? exitColIndex : bottomPosIndex.col - 1;
          const bottomCell = board[bottomRowIndex][row][bottomPosIndex.col];
          if (bottomCell === ".") {
            let stepDown = 1;
            let canMove = true;

            while (canMove && bottomRowIndex < exitColIndex) {
              const bottomCell = board[bottomRowIndex][row][stepDown + 1];
              if (bottomCell === ".") {
                stepDown++;
                moves.push({
                  row: row,
                  col: col,
                  piece: symbol,
                  direction: DIRECTIONS.DOWN,
                  steps: stepDown,
                });
              } else if (bottomCell === "K" && piece.isPrimary) {
                stepDown++;
                moves.push({
                  row: row,
                  col: col,
                  piece: symbol,
                  direction: DIRECTIONS.DOWN,
                  steps: stepDown,
                });
                canMove = false;
              } else {
                canMove = false;
              }
            }
          }
        }
      });

      return moves;
    };
  };

  export default getValidMoves;

```

GetPiecesInfo

```
•••
import type { PiecesMap } from "./types";
const getPiecesInfo = (boardConfig: string[]): PiecesMap => {
  const pieces: PiecesMap = {};
  const boardMatrix = boardConfig.map((row) => row.split(""));
  const visited = new Set<string>();
  const usedSymbols = new Set<string>();
  let pieceCount = 0;
  const MAX_PIECES = 24;

  const isIgnoredCell = (cell: string) => cell === "." || cell === " ";
  const isSamePiece = (cell1: string, cell2: string) => {
    if (isIgnoredCell(cell1) || isIgnoredCell(cell2)) return false;
    const isAlphaChar = (char: string) => /^[A-Za-z]$/.test(char);
    if (isAlphaChar(cell1) && isAlphaChar(cell2)) {
      return cell1 === cell2;
    }
    return false;
  };
  for (let i = 0; i < boardMatrix.length; i++) {
    for (let j = 0; j < boardMatrix[i].length; j++) {
      const cell = boardMatrix[i][j];
      const posKey = `${i},${j}`;
      if (visited.has(posKey)) continue;
      if (!isIgnoredCell(cell)) {
        visited.add(posKey);
        continue;
      }
      if (cell !== "K") {
        if (usedSymbols.has(cell)) {
          throw new Error(`Duplicate piece symbol "${cell}" detected. Each letter can only be used once. `);
        }
        if (pieceCount >= MAX_PIECES) {
          throw new Error(`Maximum number of pieces ${MAX_PIECES} exceeded.`);
        }
        const contiguousCells: { row: number; col: number }[] = [];
        const queue: { row: number; col: number }[] = [{ row: i, col: j }];
        const cellVisited = new Set<string>();
        cellVisited.add(posKey);

        while (queue.length > 0) {
          const current = queue.shift();
          contiguousCells.push(current);
          visited.add(`${current.row},${current.col}`);
          directions = [
            [1, 0],
            [-1, 0],
            [0, 1],
            [0, -1],
          ];
          for (const [dx, dy] of directions) {
            const newRow = current.row + dx;
            const newCol = current.col + dy;
            const newPosKey = `${newRow},${newCol}`;
            if (newRow >= 0 && newRow < boardMatrix.length && newCol >= 0 && newCol < boardMatrix[newRow].length && isSamePiece(boardMatrix[newRow][newCol], cell) && !cellVisited.has(newPosKey)) {
              queue.push({ row: newRow, col: newCol });
              cellVisited.add(newPosKey);
            }
          }
        }
        pieces[cell] = {
          positions: contiguousCells,
          symbol: cell,
        };
        usedSymbols.add(cell);
        pieceCount++;
      } else if (cell === "K") {
        pieces["K"] = { positions: [{ row: i, col: j }], symbol: "K", isExit: true };
        visited.add(posKey);
      }
    }
  }
  Object.values(pieces).forEach((piece) => {
    if (piece.isExit) return;
    const { position } = piece;
    piece.size = positions.length;
    if (positions.length > 1) {
      const sameRow = positions.every((pos) => pos.row === positions[0].row);
      piece.orientation = sameRow ? "horizontal" : "vertical";
      if (piece.orientation === "horizontal") {
        positions.sort((a, b) => a.col - b.col);
        for (let i = 1; i < positions.length; i++) {
          if (positions[i].col != positions[i - 1].col + 1) {
            throw new Error(`Piece ${piece.symbol} has non-contiguous horizontal positions. Each piece must form a continuous line. `);
          }
        }
      } else {
        positions.sort((a, b) => a.row - b.row);
        for (let i = 1; i < positions.length; i++) {
          if (positions[i].row != positions[i - 1].row + 1) {
            throw new Error(`Piece ${piece.symbol} has non-contiguous vertical positions. Each piece must form a continuous line. `);
          }
        }
      } else {
        piece.orientation = "unknown";
      }
      piece.primaryPosition = positions[0];
      piece.isPrimary = piece.symbol === "P";
    }
  });
  return pieces;
};

export default getPiecesInfo;
```

GetBoardStateString

```
const getBoardStateString = (board: string[][]): string => {
  return board.map((row) => row.join("")).join("");
};

export default getBoardStateString;
```

DeepCopy

```
const deepCopy = <T>(arr: T[][]): T[][] => arr.map((row) => [...row]);

export default deepCopy;
```

ApplyMove

```
import type { Direction, Move, PiecesMap, Position } from "../types";
import deepCopy from "./deepCopy";

const applyMove = (board: string[][][], pieces: PiecesMap, move: Move): { board: string[][]; pieces: PiecesMap } => {
    const { piece, direction, steps = 1 } = move;
    const newBoard = deepCopy(board);
    const newPieces = { ...pieces };
    const pieceInfo = newPieces[piece];

    if (!pieceInfo) {
        throw new Error(`Piece ${piece} not found`);
    }

    const positions = [...pieceInfo.positions];

    for (const pos of positions) {
        newBoard[pos.row][pos.col] = ".";
    }

    let newPositions: Position[] = [];

    for (let i = 0; i < steps; i++) {
        if (i === 0) {
            newPositions = calculateNewPositions(positions, direction);
        } else {
            newPositions = calculateNewPositions(newPositions, direction);
        }

        if (!arePositionsValid(newPositions, newBoard)) {
            if (i > 0) {
                break;
            } else {
                newPositions = positions;
                break;
            }
        }
    }

    newPieces[piece] = {
        ...pieceInfo,
        positions: newPositions,
    };

    for (const pos of newPositions) {
        newBoard[pos.row][pos.col] = piece;
    }

    return { board: newBoard, pieces: newPieces };
};

function calculateNewPositions(positions: Position[], direction: Direction): Position[] {
    return positions.map((pos) => {
        const newPos = { ...pos };

        switch (direction) {
            case "atas":
                newPos.row -= 1;
                break;
            case "bawah":
                newPos.row += 1;
                break;
            case "kiri":
                newPos.col -= 1;
                break;
            case "kanan":
                newPos.col += 1;
                break;
        }

        return newPos;
    });
}

function arePositionsValid(positions: Position[], board: string[][]): boolean {
    return positions.every((pos) => {
        if (pos.row < 0 || pos.row >= board.length || pos.col < 0 || pos.col >= board[0].length) {
            return false;
        }
        return board[pos.row][pos.col] === "." || board[pos.row][pos.col] === "K";
    });
}

export default applyMove;
```

App

```
import { lazy } from "react";
import { createBrowserRouter, RouterProvider } from "react-router";

const router = createBrowserRouter([
  {
    path: "/",
    Component: lazy(() => import("./pages/Home")),
  },
  {
    path: "/about",
    Component: lazy(() => import("./pages/About")),
  },
]);

function App() {
  return <RouterProvider router={router} />;
}

export default App;
```

About

```
const [name, setName] = useState("Yonatan Nyo");
const [email, setEmail] = useState("yonatan.nyo@ui.ac.id");
const [password, setPassword] = useState("");
const [confirmPassword, setConfirmPassword] = useState("");
const [isEmailValid, setIsEmailValid] = useState(true);
const [isPasswordMatch, setIsPasswordMatch] = useState(true);

const handleNameChange = (e) => {
  setName(e.target.value);
};

const handleEmailChange = (e) => {
  setEmail(e.target.value);
};

const handlePasswordChange = (e) => {
  setPassword(e.target.value);
};

const handleConfirmPasswordChange = (e) => {
  setConfirmPassword(e.target.value);
};

const validateEmail = (email) => {
  const re =
    /^(([^<>()\[\]\\.,;:\s@"]+(\.[^<>()\[\]\\.,;:\s@"]+)*)|(".+"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\])|(([a-zA-Z]+\.)+[a-zA-Z]{2,}))$/;
  return re.test(String(email).toLowerCase());
};

const validatePasswords = (password, confirmPassword) => {
  if (password !== confirmPassword) {
    setIsPasswordMatch(false);
  } else {
    setIsPasswordMatch(true);
  }
};

const handleSubmit = (e) => {
  e.preventDefault();
  if (!validateEmail(email)) {
    setIsEmailValid(false);
  } else {
    setIsEmailValid(true);
  }
  validatePasswords(password, confirmPassword);
  if (isEmailValid && isPasswordMatch) {
    // Perform registration logic here
    // ...
    alert("Registration successful!");
  }
};

return (
  <div>
    <form>
      <div>
        <label>Name:</label>
        <input type="text" value={name} onChange={handleNameChange}>
      </div>
      <div>
        <label>Email:</label>
        <input type="text" value={email} onChange={handleEmailChange}>
      </div>
      <div>
        <label>Password:</label>
        <input type="password" value={password} onChange={handlePasswordChange}>
      </div>
      <div>
        <label>Confirm Password:</label>
        <input type="password" value={confirmPassword} onChange={handleConfirmPasswordChange}>
      </div>
      <div>
        <button type="submit" onClick={handleSubmit}>Register</button>
      </div>
    </form>
  </div>
);

```

Home

Code terlalu panjang
https://github.com/yonatan-nyo/Tucil3_13523008_13523036/blob/main/src/src/pages/Home.tsx

HomeHead

```
const HomeHead = ({ isDarkMode, toggleDarkMode }: { isDarkMode: boolean; toggleDarkMode: () => void }) => {
  return (
    <header className="mb-8">
      <div className="flex justify-between items-center">
        <h1 className="text-3xl md:text-4xl font-bold">
          <span className="text-blue-500">Rush Hour</span> Game Solver
        </h1>
        <button onClick={toggleDarkMode} className="p-2 rounded-full ${isDarkMode ? "bg-yellow-400 text-gray-900" : "bg-gray-800 text-white"}>
          {isDarkMode ? "☀️" : "🌙"}
        </button>
      </div>
      <p className={`${mt-2 ${isDarkMode ? "text-gray-300" : "text-gray-600"}`}>A pathfinding solution for the classic Rush Hour puzzle game</p>
    );
};

export default HomeHead;
```

InputSection

ResultStat

```
import { Clock, HardDrive, MoveHorizontal } from "lucide-react";
import type { Stats } from "../../lib/types";

const ResultStat = ({ stats, isDarkMode }: { stats: Stats; isDarkMode: boolean }) => {
  return (
    <div className={`${mt-6 ${isDarkMode ? "bg-blue-900/30 border-blue-800" : "bg-blue-50 border-blue-200"}; border p-4 rounded-md`}>
      <h3 className="font-bold text-lg mb-2">Results</h3>

      <div className="grid grid-cols-1 sm:grid-cols-3 gap-4">
        <div className="flex items-center">
          <HardDrive className={`${isDarkMode ? "text-blue-400" : "text-blue-600"} mr-2`} size={20} />
          <div>
            <div className="text-sm opacity-75">Nodes Visited</div>
            <div className="font-bold text-xl">{stats.nodesVisited.toLocaleString()}</div>
          </div>
        </div>
        <div className="flex items-center">
          <Clock className={`${isDarkMode ? "text-blue-400" : "text-blue-600"} mr-2`} size={20} />
          <div>
            <div className="text-sm opacity-75">Execution Time</div>
            <div className="font-bold text-xl">{stats.executionTime} ms</div>
          </div>
        </div>
        <div className="flex items-center">
          <MoveHorizontal className={`${isDarkMode ? "text-blue-400" : "text-blue-600"} mr-2`} size={20} />
          <div>
            <div className="text-sm opacity-75">Solution Moves</div>
            <div className="font-bold text-xl">{stats.moves}</div>
          </div>
        </div>
      </div>
    );
};

export default ResultStat;
```

SolutionControl

Navbar

```
import React from 'react';
import { Link } from 'react-router';

const Navbar: React.FC = () => {
  return (
    <nav className="bg-blue-700 p-4">
      <div className="container mx-auto flex justify-between">
        <div className="text-white text-lg font-bold">
          <Link to="/">Rush Hour Solver</Link>
        </div>
        <div className="space-x-4">
          <Link to="/" className="text-white hover:text-blue-200">Home</Link>
          <Link to="/about" className="text-white hover:text-blue-200">About</Link>
        </div>
      </div>
    </nav>
  );
};

export default Navbar;
```

Footer

```
import React from "react";
import { Link } from "react-router";

const Footer: React.FC = () => {
  return (
    <footer className="bg-gray-800 text-white shadow-inner">
      <div className="container mx-auto px-4 py-6">
        <div className="flex flex-col md:flex-row justify-between items-center space-y-4 md:space-y-0">
          <div className="flex flex-col items-center md:items-start">
            <p className="text-sm md:text-base font-medium">© {new Date().getFullYear()} Rush
            Hour Game Solver</p>
            <p className="text-xs text-gray-400 mt-1">Created for Strategi dan Algoritma Tucil
            3 ITB</p>
            <div className="flex space-x-2 mt-2">
              <span className="text-xs text-gray-300">13523008</span>
              <span className="text-xs text-gray-300">13523036</span>
            </div>
          </div>
          <div className="flex space-x-4">
            <Link to="/about" className="text-gray-400 hover:text-white transition-colors duration-300 text-sm md:text-base">
              About Us
            </Link>
            <a href="https://github.com/yonatan-nyo/Tucil3_13523008_13523036"
               target="_blank"
               rel="noopener noreferrer"
               className="text-gray-400 hover:text-white transition-colors duration-300 text-sm md:text-base">
              GitHub
            </a>
          </div>
        </div>
      </div>
    </footer>
  );
};

export default Footer;
```

Layout

```
import React from "react";
import Navbar from "./Navbar";
import Footer from "./Footer";

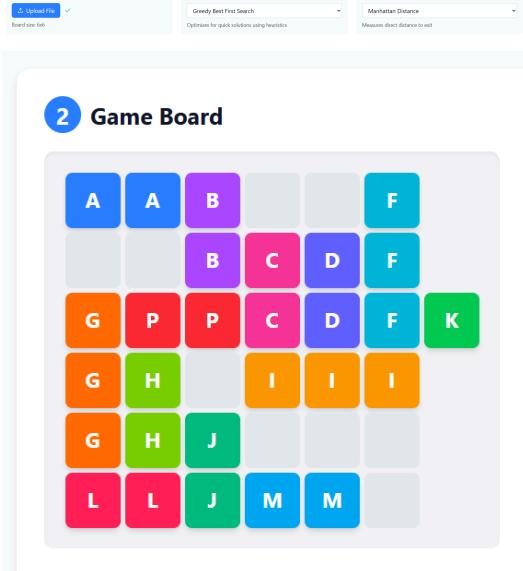
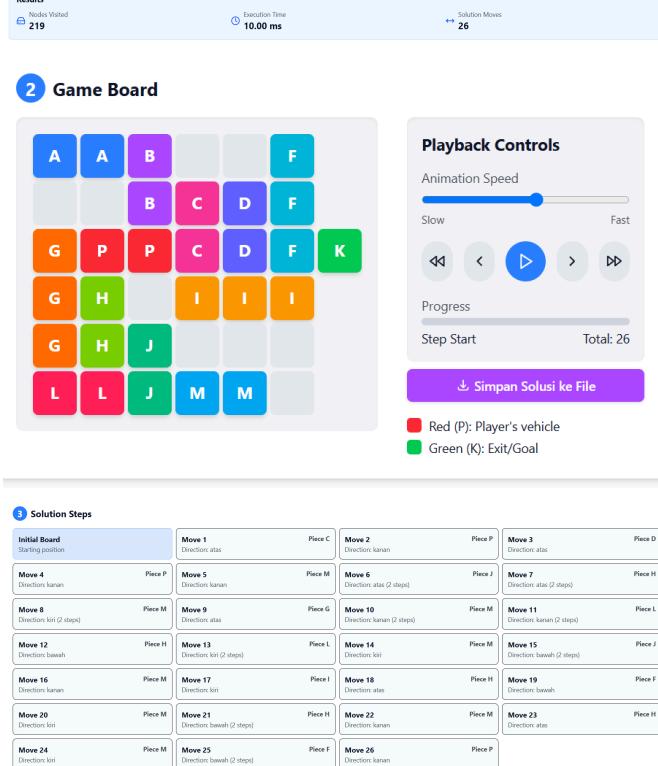
const Layout: React.FC<{ children: React.ReactNode }> = ({ children }) => {
  return (
    <div className="flex flex-col min-h-screen">
      <Navbar />
      <main className="flex-grow">{children}</main>
      <Footer />
    </div>
  );
};

export default Layout;
```

BAB IV

PENGUJIAN

5.1 Hasil Pengujian

<p style="text-align: center;">Test Case A:</p> <p style="text-align: center;">6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.</p>																																																								
<p>Input</p> <p>Test Case A menggunakan algoritma Greedy Best First Search dengan heuristic Manhattan Distance</p> 	<p>Output</p>  <table border="1" style="width: 100%; border-collapse: collapse; font-size: small;"> <thead> <tr> <th>Initial Board Starting position</th> <th>Move 1 Direction: atas</th> <th>Piece C</th> <th>Move 2 Direction: kanan</th> <th>Piece P</th> <th>Move 3 Direction: atas</th> <th>Piece D</th> </tr> </thead> <tbody> <tr> <td>Move 4 Direction: kanan</td> <td>Piece P</td> <td>Move 5 Direction: kanan</td> <td>Piece M</td> <td>Move 6 Direction: atas (2 steps)</td> <td>Piece J</td> <td>Move 7 Direction: atas (2 steps)</td> <td>Piece H</td> </tr> <tr> <td>Move 8 Direction: kiri (2 steps)</td> <td>Piece M</td> <td>Move 9 Direction: atas</td> <td>Piece G</td> <td>Move 10 Direction: kanan (2 steps)</td> <td>Piece M</td> <td>Move 11 Direction: kanan (2 steps)</td> <td>Piece L</td> </tr> <tr> <td>Move 12 Direction: bawah</td> <td>Piece H</td> <td>Move 13 Direction: kiri (2 steps)</td> <td>Piece L</td> <td>Move 14 Direction: kiri</td> <td>Piece M</td> <td>Move 15 Direction: bawah (2 steps)</td> <td>Piece J</td> </tr> <tr> <td>Move 16 Direction: kanan</td> <td>Piece M</td> <td>Move 17 Direction: kiri</td> <td>Piece I</td> <td>Move 18 Direction: atas</td> <td>Piece H</td> <td>Move 19 Direction: bawah</td> <td>Piece F</td> </tr> <tr> <td>Move 20 Direction: kiri</td> <td>Piece M</td> <td>Move 21 Direction: bawah (2 steps)</td> <td>Piece H</td> <td>Move 22 Direction: kanan</td> <td>Piece M</td> <td>Move 23 Direction: atas</td> <td>Piece H</td> </tr> <tr> <td>Move 24 Direction: kiri</td> <td>Piece M</td> <td>Move 25 Direction: bawah (2 steps)</td> <td>Piece F</td> <td>Move 26 Direction: kanan</td> <td>Piece P</td> <td></td> <td></td> </tr> </tbody> </table>	Initial Board Starting position	Move 1 Direction: atas	Piece C	Move 2 Direction: kanan	Piece P	Move 3 Direction: atas	Piece D	Move 4 Direction: kanan	Piece P	Move 5 Direction: kanan	Piece M	Move 6 Direction: atas (2 steps)	Piece J	Move 7 Direction: atas (2 steps)	Piece H	Move 8 Direction: kiri (2 steps)	Piece M	Move 9 Direction: atas	Piece G	Move 10 Direction: kanan (2 steps)	Piece M	Move 11 Direction: kanan (2 steps)	Piece L	Move 12 Direction: bawah	Piece H	Move 13 Direction: kiri (2 steps)	Piece L	Move 14 Direction: kiri	Piece M	Move 15 Direction: bawah (2 steps)	Piece J	Move 16 Direction: kanan	Piece M	Move 17 Direction: kiri	Piece I	Move 18 Direction: atas	Piece H	Move 19 Direction: bawah	Piece F	Move 20 Direction: kiri	Piece M	Move 21 Direction: bawah (2 steps)	Piece H	Move 22 Direction: kanan	Piece M	Move 23 Direction: atas	Piece H	Move 24 Direction: kiri	Piece M	Move 25 Direction: bawah (2 steps)	Piece F	Move 26 Direction: kanan	Piece P		
Initial Board Starting position	Move 1 Direction: atas	Piece C	Move 2 Direction: kanan	Piece P	Move 3 Direction: atas	Piece D																																																		
Move 4 Direction: kanan	Piece P	Move 5 Direction: kanan	Piece M	Move 6 Direction: atas (2 steps)	Piece J	Move 7 Direction: atas (2 steps)	Piece H																																																	
Move 8 Direction: kiri (2 steps)	Piece M	Move 9 Direction: atas	Piece G	Move 10 Direction: kanan (2 steps)	Piece M	Move 11 Direction: kanan (2 steps)	Piece L																																																	
Move 12 Direction: bawah	Piece H	Move 13 Direction: kiri (2 steps)	Piece L	Move 14 Direction: kiri	Piece M	Move 15 Direction: bawah (2 steps)	Piece J																																																	
Move 16 Direction: kanan	Piece M	Move 17 Direction: kiri	Piece I	Move 18 Direction: atas	Piece H	Move 19 Direction: bawah	Piece F																																																	
Move 20 Direction: kiri	Piece M	Move 21 Direction: bawah (2 steps)	Piece H	Move 22 Direction: kanan	Piece M	Move 23 Direction: atas	Piece H																																																	
Move 24 Direction: kiri	Piece M	Move 25 Direction: bawah (2 steps)	Piece F	Move 26 Direction: kanan	Piece P																																																			

Test Case A menggunakan algoritma Greedy Best First Search dengan heuristic Blocking Vehicles

1 Configure Solver

Board Configuration: **Untitled File**

Algorithm: Greedy Best First Search
Optimizes for quick solutions using heuristics

Heuristic: Blocking Vehicles
Counts vehicles blocking the path

2 Game Board

The board shows a 7x7 grid with various colored blocks representing vehicles and obstacles. The colors correspond to the vehicle types and goals defined in the solver setup.

Results
Nodes Visited: 22
Execution Time: 1.20 ms
Solution Moves: 8

2 Game Board

The solved board state is shown, with the vehicle's path highlighted in blue. The vehicle (P) has reached the green exit goal (K).

Playback Controls
Animation Speed: Slow to Fast
Progress: Step Start Total: 8
Save Solution: Simpan Solusi ke File

3 Solution Steps

Initial Board	Move 1	Move 2	Move 3	Move 4	Move 5	Move 6	Move 7	Move 8
Starting position	Piece C Direction: atas	Piece C Direction: atas	Piece D Direction: atas	Piece G Direction: kanan	Piece I Direction: kiri	Piece I Direction: kiri	Piece M Direction: bawah (3 steps)	Piece P Direction: kanan (3 steps)

Test Case A menggunakan algoritma Greedy Best First Search dengan heuristic Combined

1 Configure Solver

Board Configuration: **Untitled File**

Algorithm: Greedy Best First Search
Optimizes for quick solutions using heuristics

Heuristic: Combined Heuristic
Combines multiple metrics for better estimates

2 Game Board

The board shows a 7x7 grid with various colored blocks representing vehicles and obstacles. The colors correspond to the vehicle types and goals defined in the solver setup.

Results
Nodes Visited: 18
Execution Time: 1.50 ms
Solution Moves: 9

2 Game Board

The solved board state is shown, with the vehicle's path highlighted in blue. The vehicle (P) has reached the green exit goal (K).

Playback Controls
Animation Speed: Slow to Fast
Progress: Step Start Total: 9
Save Solution: Simpan Solusi ke File

3 Solution Steps

Initial Board	Move 1	Move 2	Move 3	Move 4	Move 5	Move 6	Move 7	Move 8
Starting position	Piece C Direction: atas	Piece C Direction: atas	Piece D Direction: atas	Piece G Direction: bawah	Piece M Direction: kanan	Piece I Direction: kiri	Piece P Direction: kanan (2 steps)	Piece F Direction: bawah (3 steps)

Test Case A menggunakan algoritma UCS

Results
Nodes Visited: 426
Execution Time: 9.10 ms
Solution Moves: 5

1 Configure Solver

Board Configuration
Upload File ✓
Board size: 6x6

Algorithm
Uniform Cost Search (UCS)
Finds the shortest path by exploring all options

Solve Puzzle > Reset

2 Game Board

The board consists of 36 squares arranged in a 6x6 grid. The colors of the blocks are: Row 1: Blue, Blue, Purple, Grey, Grey, Teal. Row 2: Grey, Grey, Grey, Grey, Grey, Grey. Row 3: Orange, Red, Red, Pink, Blue, Teal. Row 4: Orange, Green, Green, Grey, Grey, Grey. Row 5: Orange, Green, Green, Grey, Grey, Grey. Row 6: Red, Red, Green, Teal, Teal, Grey. Block 'P' is red and block 'K' is green.

Test Case A menggunakan algoritma A* dengan heuristic Manhattan Distance

1 Configure Solver

Board Configuration
Upload File ✓
Board size: 6x6

Algorithm
A* Search
Combines heuristic with path cost for efficiency

Heuristic
Blocking Vehicles
Counts vehicles blocking the path

2 Game Board

The board consists of 36 squares arranged in a 6x6 grid. The colors of the blocks are: Row 1: Blue, Blue, Purple, Grey, Grey, Teal. Row 2: Grey, Grey, Grey, Grey, Grey, Grey. Row 3: Orange, Red, Red, Pink, Blue, Teal. Row 4: Orange, Green, Green, Grey, Grey, Grey. Row 5: Orange, Green, Green, Grey, Grey, Grey. Row 6: Red, Red, Green, Teal, Teal, Grey. Block 'P' is red and block 'K' is green.

Test Case A menggunakan algoritma A* dengan heuristic Blocking Vehicles

1 Configure Solver

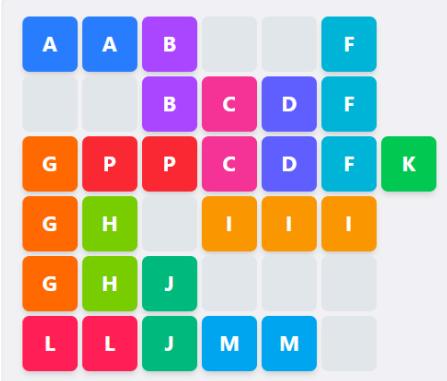
Board Configuration
Upload File ✓
Board size: 6x6

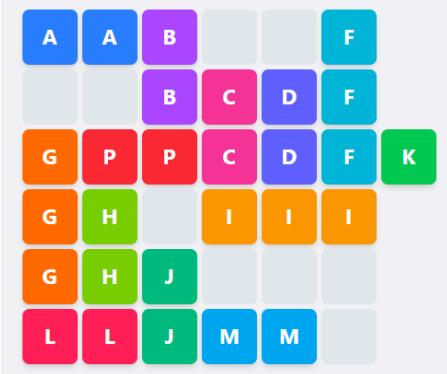
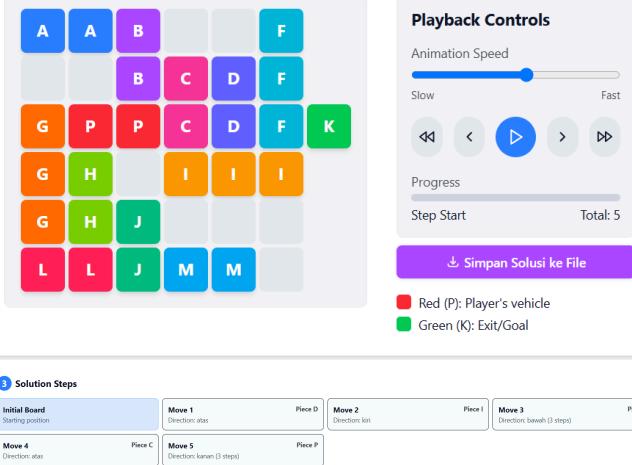
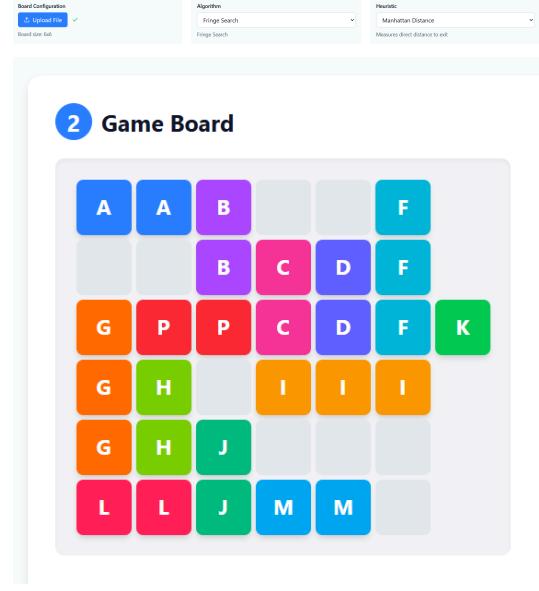
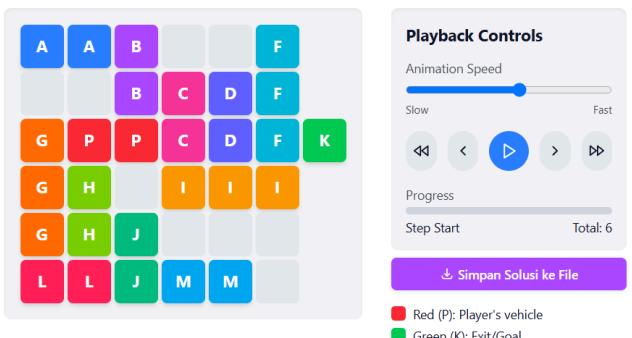
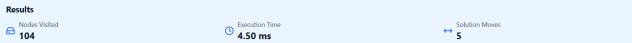
Algorithm
A* Search
Combines heuristic with path cost for efficiency

Heuristic
Blocking Vehicles
Counts vehicles blocking the path

2 Game Board

The board consists of 36 squares arranged in a 6x6 grid. The colors of the blocks are: Row 1: Blue, Blue, Purple, Grey, Grey, Teal. Row 2: Grey, Grey, Grey, Grey, Grey, Grey. Row 3: Orange, Red, Red, Pink, Blue, Teal. Row 4: Orange, Green, Green, Grey, Grey, Grey. Row 5: Orange, Green, Green, Grey, Grey, Grey. Row 6: Red, Red, Green, Teal, Teal, Grey. Block 'P' is red and block 'K' is green.

<p>2 Game Board</p> 	<p>2 Game Board</p>  <p>Playback Controls</p> <p>Animation Speed: Slow to Fast</p> <p>Progress: Step Start Total: 5</p> <p>Simpan Solusi ke File</p> <p>Red (P): Player's vehicle Green (K): Exit/Goal</p> <p>3 Solution Steps</p> <table border="1"> <tr> <td>Initial Board Starting position</td> <td>Move 1 Direction: atas Piece C</td> <td>Move 2 Direction: atas Piece D</td> <td>Move 3 Direction: kiri Piece I</td> </tr> <tr> <td>Move 4 Direction: bawah (3 steps) Piece F</td> <td>Move 5 Direction: kanan (3 steps) Piece P</td> <td>Move 6 Direction: kanan Piece K</td> <td></td> </tr> </table>	Initial Board Starting position	Move 1 Direction: atas Piece C	Move 2 Direction: atas Piece D	Move 3 Direction: kiri Piece I	Move 4 Direction: bawah (3 steps) Piece F	Move 5 Direction: kanan (3 steps) Piece P	Move 6 Direction: kanan Piece K	
Initial Board Starting position	Move 1 Direction: atas Piece C	Move 2 Direction: atas Piece D	Move 3 Direction: kiri Piece I						
Move 4 Direction: bawah (3 steps) Piece F	Move 5 Direction: kanan (3 steps) Piece P	Move 6 Direction: kanan Piece K							
<p>Test Case A menggunakan algoritma A* dengan heuristic Combined</p> <p>1 Configure Solver</p> <p>Board Configuration: <input type="button" value="Upload File"/> Board size: 6x6</p> <p>Algorithm: A* Search</p> <p>Heuristic: Combined Heuristic</p> <p>Combined heuristics with path cost for better estimate</p> <p>Combined multiple metrics for better estimate</p> <p>2 Game Board</p> 	<p>2 Game Board</p>  <p>Playback Controls</p> <p>Animation Speed: Slow to Fast</p> <p>Progress: Step Start Total: 6</p> <p>Simpan Solusi ke File</p> <p>Red (P): Player's vehicle Green (K): Exit/Goal</p> <p>3 Solution Steps</p> <table border="1"> <tr> <td>Initial Board Starting position</td> <td>Move 1 Direction: atas Piece C</td> <td>Move 2 Direction: atas Piece D</td> <td>Move 3 Direction: kanan (2 steps) Piece I</td> </tr> <tr> <td>Move 4 Direction: kiri Piece F</td> <td>Move 5 Direction: bawah (3 steps) Piece P</td> <td>Move 6 Direction: kanan Piece K</td> <td></td> </tr> </table>	Initial Board Starting position	Move 1 Direction: atas Piece C	Move 2 Direction: atas Piece D	Move 3 Direction: kanan (2 steps) Piece I	Move 4 Direction: kiri Piece F	Move 5 Direction: bawah (3 steps) Piece P	Move 6 Direction: kanan Piece K	
Initial Board Starting position	Move 1 Direction: atas Piece C	Move 2 Direction: atas Piece D	Move 3 Direction: kanan (2 steps) Piece I						
Move 4 Direction: kiri Piece F	Move 5 Direction: bawah (3 steps) Piece P	Move 6 Direction: kanan Piece K							
<p>Test Case A menggunakan algoritma Dijkstra</p> <p>1 Configure Solver</p> <p>Board Configuration: <input type="button" value="Upload File"/> Board size: 6x6</p> <p>Algorithm: Dijkstra's Algorithm</p> <p>Finds the shortest path in weighted graphs</p>	<p>Results</p> <p>Nodes Visited: 156</p> <p>Execution Time: 5.80 ms</p> <p>Solution Moves: 5</p>								

<p>2 Game Board</p> 	<p>2 Game Board</p>  <table border="1" data-bbox="773 671 1405 741"> <thead> <tr> <th>Initial Board</th> <th>Move 1 Direction: atas Piece P</th> <th>Move 2 Direction: atas Piece D</th> <th>Move 3 Direction: bawah (3 steps) Piece I</th> <th>Move 4 Direction: atas Piece C</th> <th>Move 5 Direction: kanan (3 steps) Piece F</th> <th>Move 6 Direction: kanan (2 steps) Piece K</th> </tr> </thead> </table>	Initial Board	Move 1 Direction: atas Piece P	Move 2 Direction: atas Piece D	Move 3 Direction: bawah (3 steps) Piece I	Move 4 Direction: atas Piece C	Move 5 Direction: kanan (3 steps) Piece F	Move 6 Direction: kanan (2 steps) Piece K
Initial Board	Move 1 Direction: atas Piece P	Move 2 Direction: atas Piece D	Move 3 Direction: bawah (3 steps) Piece I	Move 4 Direction: atas Piece C	Move 5 Direction: kanan (3 steps) Piece F	Move 6 Direction: kanan (2 steps) Piece K		
<p>Test Case A menggunakan algoritma Fringe dengan heuristik Manhattan Distance</p> 	<p>2 Game Board</p>  <table border="1" data-bbox="773 1305 1405 1374"> <thead> <tr> <th>Initial Board</th> <th>Move 1 Direction: atas Piece P</th> <th>Move 2 Direction: atas Piece D</th> <th>Move 3 Direction: kanan (2 steps) Piece I</th> <th>Move 4 Direction: kanan Piece C</th> <th>Move 5 Direction: bawah (3 steps) Piece F</th> <th>Move 6 Direction: kanan Piece K</th> </tr> </thead> </table>	Initial Board	Move 1 Direction: atas Piece P	Move 2 Direction: atas Piece D	Move 3 Direction: kanan (2 steps) Piece I	Move 4 Direction: kanan Piece C	Move 5 Direction: bawah (3 steps) Piece F	Move 6 Direction: kanan Piece K
Initial Board	Move 1 Direction: atas Piece P	Move 2 Direction: atas Piece D	Move 3 Direction: kanan (2 steps) Piece I	Move 4 Direction: kanan Piece C	Move 5 Direction: bawah (3 steps) Piece F	Move 6 Direction: kanan Piece K		
<p>Test Case A menggunakan algoritma Fringe dengan heuristic Blocking Vehicles</p> 	<p>Results</p> 							

2 Game Board

2 Game Board

Playback Controls

Animation Speed

Slow
<<
<
>
>>
Fast

Progress

Step Start
Total: 5

↓ Simpan Solusi ke File

Red (P): Player's vehicle
Green (K): Exit/Goal

Test Case A menggunakan algoritma Fringe dengan heuristic Combined

Configure Solver

Board Configuration: Board Upadte file ✓
Board Size: 8x8

Algorithms: Fringe Search
Fringe Search

Heuristics: Combined Heuristic
Combined multiple metrics for better estimation

Results

Nodes Visited: 155
Execution Time: 8.10 ms
Solution Moves: 5

2 Game Board

Playback Controls

Animation Speed

Slow
<<
<
>
>>
Fast

Progress

Step Start
Total: 5

↓ Simpan Solusi ke File

Red (P): Player's vehicle
Green (K): Exit/Goal

Test Case D:

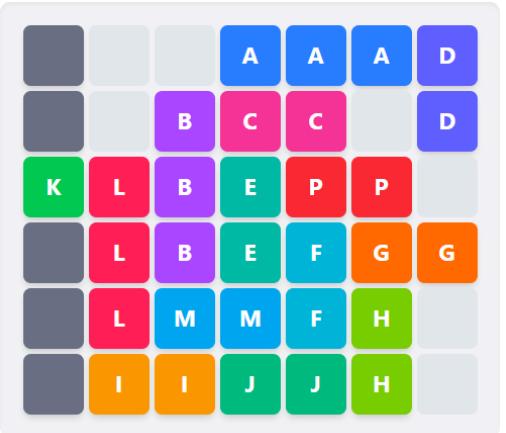
6 6

12

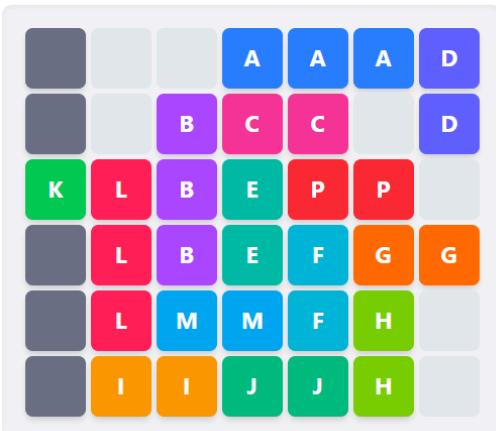
..AAAD

.BCC.D

KLBEPP.
 LBEFGG
 LMMFH.
 IIJH.

		Input	Output																																																															
		<p>Test Case D menggunakan algoritma Greedy Best First Search dengan heuristic Manhattan Distance</p>  <p>2 Game Board</p> 	<p>Results</p> <ul style="list-style-type: none"> Nodes Visited: 47,499 Execution Time: 918.40 ms Solution Moves: 459 <p>2 Game Board</p>  <p>Playback Controls</p> <ul style="list-style-type: none"> Animation Speed: Slow to Fast Progress: Step Start Total: 459 Simpan Solusi ke File <p>Legend: Red (P): Player's vehicle, Green (K): Exit/Goal</p> <p>3 Solution Steps</p> <table border="1"> <thead> <tr> <th>Initial Board Starting position</th> <th>Move 1 Direction: kiri</th> <th>Piece A</th> <th>Move 2 Direction: bawah</th> <th>Piece D</th> <th>Move 3 Direction: atas (2 step)</th> <th>Piece L</th> </tr> </thead> <tbody> <tr> <td>Move 4 Direction: kanan</td> <td>Piece C</td> <td>Move 5 Direction: kiri</td> <td>Piece M</td> <td>Move 6 Direction: kiri</td> <td>Piece C</td> <td>Move 7 Direction: atas</td> <td>Piece D</td> </tr> <tr> <td>Move 8 Direction: bawah</td> <td>Piece E</td> <td>Move 9 Direction: kanan</td> <td>Piece P</td> <td>Move 10 Direction: atas</td> <td>Piece F</td> <td>Move 11 Direction: atas</td> <td>Piece E</td> </tr> <tr> <td>Move 12 Direction: kanan (2 steps)</td> <td>Piece M</td> <td>Move 13 Direction: kiri</td> <td>Piece M</td> <td>Move 14 Direction: bawah</td> <td>Piece F</td> <td>Move 15 Direction: kiri</td> <td>Piece M</td> </tr> <tr> <td>Move 16 Direction: bawah</td> <td>Piece L</td> <td>Move 17 Direction: kanan</td> <td>Piece M</td> <td>Move 18 Direction: atas</td> <td>Piece F</td> <td>Move 19 Direction: kanan</td> <td>Piece M</td> </tr> <tr> <td>Move 20 Direction: kanan (2 step)</td> <td>Piece M</td> <td>Move 21 Direction: bawah</td> <td>Piece E</td> <td>Move 22 Direction: bawah</td> <td>Piece F</td> <td>Move 23 Direction: kanan</td> <td>Piece A</td> </tr> <tr> <td>Move 24 Direction: atas</td> <td>Piece F</td> <td>Move 25 Direction: atas</td> <td>Piece E</td> <td>Move 26 Direction: kanan (2 steps)</td> <td>Piece M</td> <td>Move 27 Direction: kiri</td> <td>Piece M</td> </tr> <tr> <td>Move 28</td> <td>Piece F</td> <td>Move 29</td> <td>Piece M</td> <td>Move 30</td> <td>Piece L</td> <td>Move 31</td> <td>Piece M</td> </tr> </tbody> </table>	Initial Board Starting position	Move 1 Direction: kiri	Piece A	Move 2 Direction: bawah	Piece D	Move 3 Direction: atas (2 step)	Piece L	Move 4 Direction: kanan	Piece C	Move 5 Direction: kiri	Piece M	Move 6 Direction: kiri	Piece C	Move 7 Direction: atas	Piece D	Move 8 Direction: bawah	Piece E	Move 9 Direction: kanan	Piece P	Move 10 Direction: atas	Piece F	Move 11 Direction: atas	Piece E	Move 12 Direction: kanan (2 steps)	Piece M	Move 13 Direction: kiri	Piece M	Move 14 Direction: bawah	Piece F	Move 15 Direction: kiri	Piece M	Move 16 Direction: bawah	Piece L	Move 17 Direction: kanan	Piece M	Move 18 Direction: atas	Piece F	Move 19 Direction: kanan	Piece M	Move 20 Direction: kanan (2 step)	Piece M	Move 21 Direction: bawah	Piece E	Move 22 Direction: bawah	Piece F	Move 23 Direction: kanan	Piece A	Move 24 Direction: atas	Piece F	Move 25 Direction: atas	Piece E	Move 26 Direction: kanan (2 steps)	Piece M	Move 27 Direction: kiri	Piece M	Move 28	Piece F	Move 29	Piece M	Move 30	Piece L	Move 31	Piece M
Initial Board Starting position	Move 1 Direction: kiri	Piece A	Move 2 Direction: bawah	Piece D	Move 3 Direction: atas (2 step)	Piece L																																																												
Move 4 Direction: kanan	Piece C	Move 5 Direction: kiri	Piece M	Move 6 Direction: kiri	Piece C	Move 7 Direction: atas	Piece D																																																											
Move 8 Direction: bawah	Piece E	Move 9 Direction: kanan	Piece P	Move 10 Direction: atas	Piece F	Move 11 Direction: atas	Piece E																																																											
Move 12 Direction: kanan (2 steps)	Piece M	Move 13 Direction: kiri	Piece M	Move 14 Direction: bawah	Piece F	Move 15 Direction: kiri	Piece M																																																											
Move 16 Direction: bawah	Piece L	Move 17 Direction: kanan	Piece M	Move 18 Direction: atas	Piece F	Move 19 Direction: kanan	Piece M																																																											
Move 20 Direction: kanan (2 step)	Piece M	Move 21 Direction: bawah	Piece E	Move 22 Direction: bawah	Piece F	Move 23 Direction: kanan	Piece A																																																											
Move 24 Direction: atas	Piece F	Move 25 Direction: atas	Piece E	Move 26 Direction: kanan (2 steps)	Piece M	Move 27 Direction: kiri	Piece M																																																											
Move 28	Piece F	Move 29	Piece M	Move 30	Piece L	Move 31	Piece M																																																											
		<p>Test Case D menggunakan algoritma Greedy Best First Search dengan heuristic Blocking Vehicles</p>  <p>2 Game Board</p> 	<p>Results</p> <ul style="list-style-type: none"> Nodes Visited: 24,991 Execution Time: 505.70 ms Solution Moves: 337 <p>2 Game Board</p>  <p>Playback Controls</p> <ul style="list-style-type: none"> Animation Speed: Slow to Fast Progress: Step Start Total: 337 Simpan Solusi ke File <p>Legend: Red (P): Player's vehicle, Green (K): Exit/Goal</p>																																																															

2 Game Board



Solution Steps

Initial Board Starting position	Move 1 Direction: atas Piece L	Move 2 Direction: kanan Piece E	Move 3 Direction: kanan Piece P	Move 4 Direction: atas Piece L	Move 5 Direction: atas Piece E	Move 6 Direction: kanan Piece L	Move 7 Direction: bawah Piece L
Move 8 Direction: bawah Piece L	Move 9 Direction: atas (2 steps) Piece E	Move 10 Direction: kanan Piece M	Move 11 Direction: kanan Piece C	Move 12 Direction: bawah Piece L	Move 13 Direction: kanan Piece M	Move 14 Direction: bawah Piece B	Move 15 Direction: kanan Piece M
Move 16 Direction: atas Piece L	Move 17 Direction: kanan Piece M	Move 18 Direction: kanan Piece P	Move 19 Direction: kanan Piece M	Move 20 Direction: kanan Piece A	Move 21 Direction: kanan Piece M	Move 22 Direction: atas Piece M	Move 23 Direction: kanan Piece M
Move 24 Direction: bawah (2 steps) Piece M	Move 25 Direction: bawah Piece M	Move 26 Direction: kanan (2 steps) Piece L	Move 27 Direction: kanan Piece M	Move 28 Direction: bawah Piece L	Move 29 Direction: kanan Piece M	Move 30 Direction: atas Piece C	Move 31 Direction: kanan Piece M
Move 30 Direction: atas Piece L	Move 31 Direction: kanan Piece M						

Test Case D menggunakan algoritma Greedy Best First Search dengan heuristic Combined

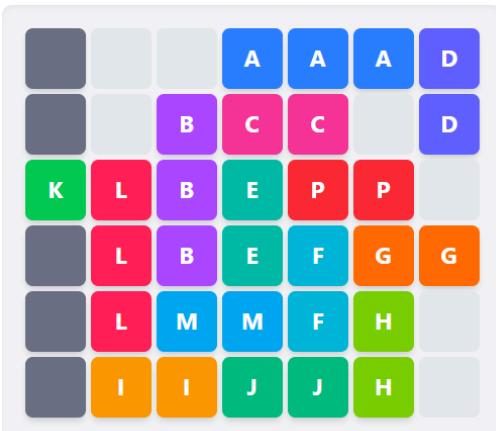
Board Configuration: Export File

Algorithm: Greedy Best First Search

Heuristic: Combined Heuristic

Optimizes for quick solutions using heuristics. Combines multiple metrics for better estimates.

2 Game Board



Results

Nodes Visited: 38,450 Execution Time: 474.20 ms Solution Moves: 305

2 Game Board

Playback Controls

Animation Speed:

Slow << < > >> Fast

Progress:

Step Start Total: 305

↓ Simpan Solusi ke File

Red (P): Player's vehicle
Green (K): Exit/Goal

Solution Steps

Initial Board Starting position	Move 1 Direction: atas Piece B	Move 2 Direction: atas (2 steps) Piece L	Move 3 Direction: kanan Piece M	Move 4 Direction: bawah Piece E	Move 5 Direction: bawah Piece B	Move 6 Direction: atas Piece B	Move 7 Direction: kanan Piece A
Move 8 Direction: bawah Piece P	Move 9 Direction: atas (2 steps) Piece M	Move 10 Direction: bawah (2 steps) Piece L	Move 11 Direction: atas Piece F	Move 12 Direction: kanan Piece M	Move 13 Direction: atas (2 steps) Piece L	Move 14 Direction: bawah (2 steps) Piece M	Move 15 Direction: kanan Piece A
Move 16 Direction: kanan Piece M	Move 17 Direction: kanan Piece M	Move 18 Direction: bawah (2 steps) Piece L	Move 19 Direction: kanan Piece M	Move 20 Direction: atas Piece L	Move 21 Direction: kanan (2 steps) Piece M	Move 22 Direction: kanan Piece M	Move 23 Direction: kanan Piece C
Move 24 Direction: kanan Piece M	Move 25 Direction: kanan (2 steps) Piece M	Move 26 Direction: atas Piece L	Move 27 Direction: kanan (2 steps) Piece M	Move 28 Direction: bawah Piece M	Move 29 Direction: kanan Piece L	Move 30 Direction: atas Piece H	Move 31 Direction: kanan Piece L
Move 30 Direction: atas Piece L	Move 31 Direction: kanan Piece M						

Test Case D menggunakan algoritma UCS

Nodes Visited: 67,907 Execution Time: 504.00 ms Solution Moves: 44

1 Configure Solver

Board Configuration
Upload File ✓
Board size: 6x7

Algorithm
Uniform Cost Search (UCS)
Finds the shortest path by exploring all options

Solve Puzzle ➔ Reset

2 Game Board

3 Solution Steps

Initial Board Starting position	Move 1 Direction: kiri (2 steps)	Move 2 Direction: kanan	Move 3 Direction: atas	Move 4 Direction: kanan	Move 5 Direction: bawah	Move 6 Direction: kiri (2 steps)	Move 7 Direction: atas (2 steps)	Move 8 Direction: kiri	Move 9 Direction: kiri	Move 10 Direction: bawah (4 steps)	Move 11 Direction: kanan	Move 12 Direction: kanan	Move 13 Direction: bawah (2 steps)	Move 14 Direction: kanan (3 steps)	Move 15 Direction: kanan (4 steps)	Move 16 Direction: atas (2 steps)	Move 17 Direction: atas	Move 18 Direction: atas (2 steps)	Move 19 Direction: atas	Move 20 Direction: kiri (4 steps)	Move 21 Direction: atas	Move 22 Direction: atas	Move 23 Direction: kiri (2 steps)	Move 24 Direction: kanan (2 steps)	Move 25 Direction: bawah (3 steps)	Move 26 Direction: bawah (3 steps)	Move 27 Direction: kanan (2 steps)	Move 28 Direction: bawah (2 steps)	Move 29 Direction: atas	Move 30 Direction: atas	Move 31 Direction: atas		
Piece A	Piece P	Piece F	Piece F	Piece M	Piece B	Piece G	Piece C	Piece P	Piece G	Piece D	Piece G	Piece F	Piece F	Piece E	Piece A	Piece B	Piece B	Piece C	Piece L	Piece E	Piece D	Piece H	Piece C	Piece I	Piece J	Piece H	Piece I	Piece J	Piece H	Piece B	Piece A	Piece C	Piece P

Playback Controls
Animation Speed: Slow ⏪ Fast ⏩ Progress: Step Start Total: 44
Simpan Solusi ke File

Red (P): Player's vehicle
Green (K): Exit/Goal

Test Case D menggunakan algoritma A* dengan heuristic Manhattan Distance

1 Configure Solver

Board Configuration
Upload File ✓
Board size: 6x7

Algorithm
A* Search
Manhattan Distance
Combines UCS with heuristics for efficient pathfinding

Solve Puzzle ➔ Reset

2 Game Board

3 Solution Steps

Initial Board Starting position	Move 1 Direction: kanan	Move 2 Direction: atas	Move 3 Direction: kanan	Move 4 Direction: bawah	Move 5 Direction: kiri	Move 6 Direction: kiri (2 steps)	Move 7 Direction: atas (2 steps)	Move 8 Direction: kiri	Move 9 Direction: kiri	Move 10 Direction: bawah (4 steps)	Move 11 Direction: kanan	Move 12 Direction: kanan	Move 13 Direction: bawah (2 steps)	Move 14 Direction: kanan (3 steps)	Move 15 Direction: kanan (4 steps)	Move 16 Direction: atas (2 steps)	Move 17 Direction: atas	Move 18 Direction: atas (2 steps)	Move 19 Direction: atas	Move 20 Direction: kiri (4 steps)	Move 21 Direction: atas	Move 22 Direction: atas	Move 23 Direction: kiri (2 steps)	Move 24 Direction: kanan (2 steps)	Move 25 Direction: bawah (3 steps)	Move 26 Direction: bawah (3 steps)	Move 27 Direction: kanan (2 steps)	Move 28 Direction: bawah (2 steps)	Move 29 Direction: atas	Move 30 Direction: atas	Move 31 Direction: atas		
Piece A	Piece P	Piece F	Piece M	Piece B	Piece C	Piece G	Piece D	Piece F	Piece G	Piece D	Piece G	Piece F	Piece F	Piece E	Piece A	Piece B	Piece B	Piece C	Piece L	Piece E	Piece D	Piece H	Piece C	Piece I	Piece J	Piece H	Piece I	Piece J	Piece H	Piece B	Piece A	Piece C	Piece P

Results
Nodes Visited: 61,767 Execution Time: 493.20 ms Solution Moves: 45

Playback Controls
Animation Speed: Slow ⏪ Fast ⏩ Progress: Step Start Total: 45
Simpan Solusi ke File

Red (P): Player's vehicle
Green (K): Exit/Goal

Test Case D menggunakan algoritma A* dengan heuristic Blocking Vehicles

The screenshot shows the 'Configure Solver' interface with the following settings:

- Board Configuration:** Board size: 6x7, Upload File (selected), Board case: 67.
- Algorithm:** A* Search.
- Heuristic:** Manhattan Distance, Combined UCS with heuristic for efficient pathfinding.

2 Game Board:

A 6x7 grid representing the game board. The grid contains various colored blocks labeled with letters A through P and K. The 'K' blocks are green and represent the exit/goal. The 'P' blocks are red and represent the player's vehicle. Some blocks are grey, indicating they are obstacles or part of the wall.

Results
Nodes Visited: 58,879
Execution Time: 361.90 ms
Solution Moves: 44

2 Game Board

The game board shows the final state after 44 moves. The pieces have moved from their initial positions to reach the goal area. The 'P' blocks (red) are now positioned near the 'K' blocks (green) at the bottom right.

Playback Controls:
Animation Speed: Slow to Fast
Progress: Step Start, Total: 44
Simpan Solusi ke File

3 Solution Steps

Initial Board	Move 1	Move 2	Move 3
Starting position	Piece A (Direction: kiri (2 steps))	Piece A (Direction: kanan)	Piece P (Direction: atas)
Move 4 (Direction: kanan)	Piece M (Direction: kiri)	Piece B (Direction: bawah)	Piece C (Direction: kanan)
Move 8 (Direction: kiri)	Piece P (Direction: kiri)	Piece G (Direction: bawah)	Piece D (Direction: kanan)
Move 12 (Direction: kanan)	Piece P (Direction: kiri)	Piece F (Direction: bawah (2 steps))	Piece C (Direction: kanan)
Move 16 (Direction: kanan (2 steps))	Piece A (Direction: kiri)	Piece B (Direction: atas (2 steps))	Piece L (Direction: atas)
Move 20 (Direction: kiri (4 steps))	Piece G (Direction: kiri (2 steps))	Piece M (Direction: bawah (2 steps))	Piece E (Direction: bawah (2 steps))
Move 24 (Direction: kiri (2 steps))	Piece P (Direction: kiri)	Piece C (Direction: bawah (2 steps))	Piece D (Direction: atas (4 steps))
Move 28	Piece J (Direction: kiri)	Piece F (Direction: atas)	Piece I (Direction: atas)
	Piece J (Direction: bawah)	Piece F (Direction: bawah)	Piece H (Direction: atas (3 steps))
	Piece J (Direction: kanan)	Piece F (Direction: kanan)	Piece G (Direction: atas)
	Piece J (Direction: kanan)	Piece F (Direction: kanan)	Piece G (Direction: atas)

Test Case D menggunakan algoritma A* dengan heuristic Combined

The screenshot shows the 'Configure Solver' interface with the following settings:

- Board Configuration:** Board size: 6x7, Upload File (selected), Board case: 67.
- Algorithm:** A* Search.
- Heuristic:** Combined Heuristic, Combines heuristics with path cost for efficiency.

2 Game Board:

A 6x7 grid representing the game board. The grid contains various colored blocks labeled with letters A through P and K. The 'K' blocks are green and represent the exit/goal. The 'P' blocks are red and represent the player's vehicle. Some blocks are grey, indicating they are obstacles or part of the wall.

Results
Nodes Visited: 58,518
Execution Time: 454.90 ms
Solution Moves: 47

2 Game Board

The game board shows the final state after 47 moves. The pieces have moved from their initial positions to reach the goal area. The 'P' blocks (red) are now positioned near the 'K' blocks (green) at the bottom right.

Playback Controls:
Animation Speed: Slow to Fast
Progress: Step Start, Total: 47
Simpan Solusi ke File

3 Solution Steps

Initial Board	Move 1	Move 2	Move 3
Starting position	Piece A (Direction: kiri (2 steps))	Piece A (Direction: kanan)	Piece P (Direction: atas)
Move 4 (Direction: kanan)	Piece M (Direction: kiri)	Piece B (Direction: bawah)	Piece C (Direction: kanan)
Move 8 (Direction: kiri)	Piece P (Direction: kiri)	Piece G (Direction: bawah)	Piece D (Direction: kanan)
Move 12 (Direction: kanan)	Piece P (Direction: kiri)	Piece F (Direction: bawah (2 steps))	Piece C (Direction: kanan)
Move 16 (Direction: kanan (2 steps))	Piece A (Direction: kiri)	Piece B (Direction: atas (2 steps))	Piece L (Direction: atas)
Move 20 (Direction: kiri (4 steps))	Piece G (Direction: kiri (2 steps))	Piece M (Direction: bawah (2 steps))	Piece E (Direction: bawah (2 steps))
Move 24 (Direction: kiri (2 steps))	Piece P (Direction: kiri)	Piece C (Direction: bawah (2 steps))	Piece D (Direction: atas (4 steps))
Move 28	Piece J (Direction: kiri)	Piece F (Direction: atas)	Piece I (Direction: atas)
	Piece J (Direction: bawah)	Piece F (Direction: bawah)	Piece H (Direction: atas (3 steps))
	Piece J (Direction: kanan)	Piece F (Direction: kanan)	Piece G (Direction: atas)
	Piece J (Direction: kanan)	Piece F (Direction: kanan)	Piece G (Direction: atas)

3 Solution Steps					
Initial Board Starting position	Piece P	Move 1 Direction: kanan	Piece P	Move 2 Direction: atas	Piece F
Move 4 Direction: bawah	Piece B	Move 5 Direction: kiri (2 steps)	Piece C	Move 6 Direction: kiri (2 steps)	Piece A
Move 8 Direction: kiri	Piece P	Move 9 Direction: kiri	Piece G	Move 10 Direction: bawah (4 steps)	Piece D
Move 12 Direction: kanan	Piece P	Move 13 Direction: bawah (2 steps)	Piece F	Move 14 Direction: kanan (4 steps)	Piece C
Move 16 Direction: atas (2 steps)	Piece B	Move 17 Direction: atas (2 steps)	Piece L	Move 18 Direction: atas (2 steps)	Piece B
Move 20 Direction: atas	Piece F	Move 21 Direction: kiri (4 steps)	Piece G	Move 22 Direction: bawah (2 steps)	Piece F
Move 24 Direction: atas	Piece H	Move 25 Direction: kanan (2 steps)	Piece J	Move 26 Direction: bawah (4 steps)	Piece E
Move 28	Piece F	Move 29 Direction: kanan (2 steps)	Piece G	Move 30 Direction: atas (2 steps)	Piece C
					Move 31 Direction: kiri (2 steps)

Test Case D menggunakan algoritma Dijkstra

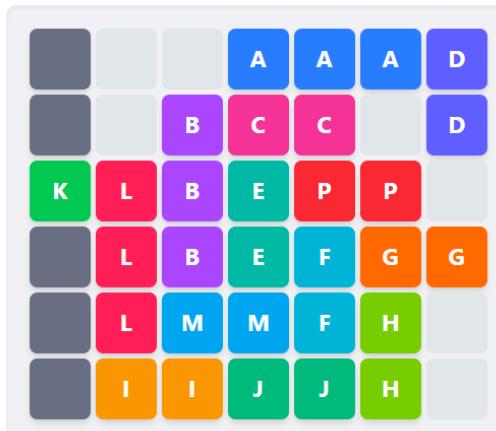
1 Configure Solver

Board Configuration
Upload File ✓
Board size: 6x7

Algorithm
Dijkstra's Algorithm
Combines UCS with heuristics for efficient pathfinding

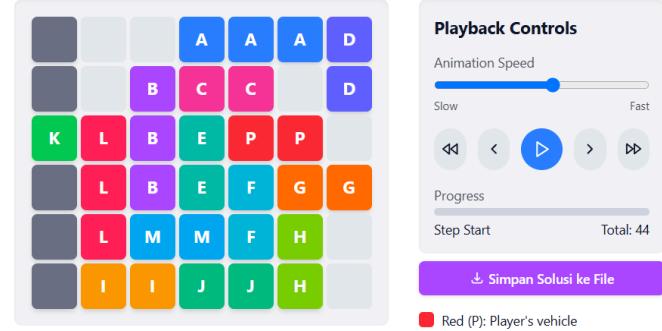
Solve Puzzle > Reset

2 Game Board



Results					
Nodes Visited 14,437	Execution Time 340.20 ms	Solution Moves 44			

3 Game Board



3 Solution Steps					
Initial Board Starting position	Piece P	Move 1 Direction: kanan	Piece P	Move 2 Direction: atas	Piece F
Move 4 Direction: bawah	Piece B	Move 5 Direction: kiri (2 steps)	Piece C	Move 6 Direction: kiri (2 steps)	Piece A
Move 8 Direction: kiri	Piece P	Move 9 Direction: kiri	Piece G	Move 10 Direction: bawah (4 steps)	Piece D
Move 12 Direction: kanan	Piece P	Move 13 Direction: bawah (2 steps)	Piece F	Move 14 Direction: kanan (4 steps)	Piece C
Move 16 Direction: atas (2 steps)	Piece B	Move 17 Direction: atas (2 steps)	Piece L	Move 18 Direction: atas (2 steps)	Piece B
Move 20 Direction: kiri (4 steps)	Piece G	Move 21 Direction: atas	Piece D	Move 22 Direction: atas	Piece H
Move 24 Direction: kiri (2 steps)	Piece M	Move 25 Direction: bawah (3 steps)	Piece F	Move 26 Direction: bawah (3 steps)	Piece E
Move 28	Piece F	Move 29 Direction: kanan (2 steps)	Piece P	Move 30 Direction: atas (2 steps)	Piece C
					Move 31 Direction: kiri (2 steps)

Test Case D menggunakan algoritma Fringe dengan heuristik Manhattan Distance

Board Configuration
Upload File ✓
Board size: 6x6

Algorithm
Fringe Search
Fringe Search

Heuristic
Manhattan Distance
Measures direct distance to exit

Results					
Nodes Visited 14,324	Execution Time 404.80 ms	Solution Moves 46			

2 Game Board

2 Game Board

Playback Controls

- Animation Speed: Slow to Fast
- Progress: Step Start to Total: 46
- Save Solution to File

3 Solution Steps

Initial Board Starting position	Move 1 Direction: kiri (2 steps)	Piece A	Move 2 Direction: kanan	Piece P	Move 3 Direction: atas	Piece F	
Move 4 Direction: kanan	Piece M	Move 5 Direction: bawah	Piece B	Move 6 Direction: kiri	Piece C	Move 7 Direction: atas (2 steps)	Piece F
Move 8 Direction: kiri	Piece P	Move 9 Direction: kanan	Piece G	Move 10 Direction: bawah (4 steps)	Piece D	Move 11 Direction: kanan	Piece G
Move 12 Direction: kanan	Piece P	Move 13 Direction: bawah (2 steps)	Piece F	Move 14 Direction: kanan (2 steps)	Piece A	Move 15 Direction: kanan (3 steps)	Piece C
Move 16 Direction: atas (2 steps)	Piece B	Move 17 Direction: atas (2 steps)	Piece L	Move 18 Direction: atas	Piece E	Move 19 Direction: atas	Piece F
Move 20 Direction: kiri (4 steps)	Piece G	Move 21 Direction: atas	Piece D	Move 22 Direction: kiri (2 steps)	Piece M	Move 23 Direction: atas	Piece H
Move 24 Direction: kanan (2 steps)	Piece J	Move 25 Direction: bawah (3 steps)	Piece E	Move 26 Direction: bawah (3 steps)	Piece F	Move 27 Direction: kiri (2 steps)	Piece P
Move 28 Direction: kanan (2 steps)	Piece G	Move 29 Direction: bawah	Piece B	Move 30 Direction: kiri	Piece A	Move 31 Direction: kiri (2 steps)	Piece C

Legend: Red (P): Player's vehicle; Green (K): Exit/Goal

Test Case D menggunakan algoritma Fringe dengan heuristic Blocking Vehicles

Board Configuration: Board State file ✓
Algorithm: Fringe Search
Heuristic: Blocking Vehicles

2 Game Board



Results: Nodes Visited: 13,363, Execution Time: 292.00 ms, Solution Moves: 44

2 Game Board



Playback Controls

Animation Speed: Slow to Fast

Progress: Step Start to Total: 44

Save Solution to File

Red (P): Player's vehicle; Green (K): Exit/Goal

3 Solution Steps

Initial Board Starting position	Move 1 Direction: kiri (2 steps)	Piece A	Move 2 Direction: kanan	Piece P	Move 3 Direction: atas	Piece F	
Move 4 Direction: kanan	Piece M	Move 5 Direction: bawah	Piece B	Move 6 Direction: kiri	Piece C	Move 7 Direction: atas (2 steps)	Piece F
Move 8 Direction: kiri	Piece P	Move 9 Direction: kanan	Piece G	Move 10 Direction: bawah (4 steps)	Piece D	Move 11 Direction: kanan	Piece P
Move 12 Direction: kanan	Piece P	Move 13 Direction: bawah (2 steps)	Piece F	Move 14 Direction: kanan (2 steps)	Piece A	Move 15 Direction: kanan (3 steps)	Piece C
Move 16 Direction: atas (2 steps)	Piece B	Move 17 Direction: atas (2 steps)	Piece L	Move 18 Direction: atas	Piece E	Move 19 Direction: atas	Piece F
Move 20 Direction: kiri (4 steps)	Piece G	Move 21 Direction: atas	Piece D	Move 22 Direction: kiri (2 steps)	Piece M	Move 23 Direction: atas	Piece H
Move 24 Direction: kanan (2 steps)	Piece J	Move 25 Direction: bawah (3 steps)	Piece E	Move 26 Direction: bawah (3 steps)	Piece F	Move 27 Direction: kiri (2 steps)	Piece C
Move 28 Direction: kiri (2 steps)	Piece G	Move 29 Direction: bawah	Piece B	Move 30 Direction: kiri	Piece A	Move 31 Direction: bawah	Piece B

Test Case D menggunakan algoritma Fringe dengan heuristic Combined

1 Configure Solver

Board Configuration: Upload File Board Size 6x6

Algorithms: Fringe Search Greedy Best First Search

Heuristic: Combined Heuristic Combines multiple metrics for better estimates

Results

Nodes Visited: 14,337 Execution Time: 315.90 ms Solution Moves: 45

2 Game Board

3 Solution Steps

Initial Board Starting position	Move 1 Direction: kiri (2 steps)	Piece A	Move 2 Direction: kanan	Piece P	Move 3 Direction: atas	Piece F
Move 4 Direction: kanan	Move 5 Direction: bawah	Piece B	Move 6 Direction: kiri	Piece C	Move 7 Direction: atas (2 steps)	Piece I
Move 8 Direction: kiri	Move 9 Direction: kiri	Piece G	Move 10 Direction: bawah (4 steps)	Piece D	Move 11 Direction: kanan	Piece G
Move 12 Direction: kanan	Move 13 Direction: bawah (2 steps)	Piece F	Move 14 Direction: kanan (2 steps)	Piece A	Move 15 Direction: kanan (3 steps)	Piece C
Move 16 Direction: atas (2 steps)	Move 17 Direction: atas (2 steps)	Piece L	Move 18 Direction: atas	Piece E	Move 19 Direction: atas	Piece F
Move 20 Direction: kiri (4 steps)	Move 21 Direction: atas	Piece D	Move 22 Direction: kiri (2 steps)	Piece M	Move 23 Direction: atas	Piece H
Move 24 Direction: kanan (2 steps)	Move 25 Direction: bawah (3 steps)	Piece I	Move 26 Direction: bawah (3 steps)	Piece F	Move 27 Direction: kiri (2 steps)	Piece P
Move 28 Direction: kanan (2 steps)	Move 29 Direction: bawah	Piece B	Move 30 Direction: kiri	Piece A	Move 31 Direction: kiri (2 steps)	Piece C

Playback Controls

Animation Speed: Slow Fast

<< < > >>

Progress: Step Start Total: 45

Simpan Solusi ke File

Red (P): Player's vehicle
Green (K): Exit/Goal

Test Case F (kosong):

Input	Output
Test Case F	Error Error parsing file: Input file must have at least 2 lines (dimensions and numPieces);

Test Case G (No board):

6 6
11

Input	Output
Test Case G	Error Error parsing file: There should be one K in the board.

--	--

Test Case H (Mismatch puzzle piece count):

6 6
 11
 AAB..F
 ..BCDF
 GPPCDFK
 GH....
 GHJ...
 LLJMM.

Input	Output
<p>Test Case H</p>	<p>Solve Puzzle ></p> <p>Error Error parsing file: Mismatch in piece count: Input specifies 11 pieces, but 10 unique pieces found on board.</p>

Test Case J ($K>1$):

6 6
 10
 K
 PAB..F
 PABCDF
 GGGCDF
 .H....
 .HJ...
 LLJMM.K

Input	Output
<p>Test Case J</p>	<p>Error Error parsing file: There should be only one K in the board.</p>

Test Case K (K tidak sejajar dengan P):

6 6
 10

<p style="text-align: center;"> K PAB..F PABCDF GGGCDF .H.... .HJ... LLJMM. </p>	
Input	Output
Test Case K 	<div style="background-color: #f0f0f0; padding: 5px;"> Error Error parsing file: Exit (K) must be aligned with the primary piece (P). </div>

5.2 Analisis Hasil Pengujian

5.2.1 Algoritma Uniform Cost Search (UCS)

Uniform Cost Search mengekspansi simpul berdasarkan biaya kumulatif $g(n)$, dan dalam kasus Rush Hour, di mana setiap gerakan memiliki biaya seragam sebesar 1, ia menjamin menemukan solusi dengan jumlah gerakan minimum. Karena UCS menggunakan priority queue yang diurutkan oleh $g(n)$, ia mengeksplorasi semua konfigurasi papan hingga kedalaman solusi optimal tercapai. Dari segi kompleksitas, UCS memiliki waktu dan ruang $O(b^d)$, di mana b adalah faktor percabangan (rata-rata gerakan valid per keadaan) dan d adalah kedalaman solusi. Implementasi kami juga mencatat jumlah simpul yang dikunjungi dan menggunakan visitedStates untuk menghindari siklus, namun meski lengkap dan optimal, UCS menjadi sangat mahal untuk puzzle dengan ruang status besar karena frontier tumbuh eksponensial.

5.2.2 Algoritma Greedy Best-First Search (GBFS)

Greedy Best-First Search memilih simpul berikutnya hanya berdasarkan nilai heuristik $h(n)$, tanpa mempertimbangkan biaya sejauh ini, sehingga ia menavigasi papan

seolah-olah “greedy” menuju solusi. Dalam implementasi Rush Hour, GBFS sangat cepat dan hemat memori, runtime dan ruang tipikalnya jauh lebih kecil daripada batas terburuk $O(b^m)$ (m = kedalaman maksimum yang dicapai) karena heuristik yang baik memotong sebagian cabang yang tidak relevan. Namun, secara teoretis GBFS tidak menjamin solusi optimal, karena ia dapat terjebak pada jalur yang tampak menjanjikan menurut heuristik tetapi memerlukan lebih banyak langkah secara keseluruhan.

5.2.3 Algoritma A* Search

A* Search memadukan UCS dan GBFS melalui fungsi evaluasi $f(n) = g(n) + h(n)$. Dengan heuristik admissible, seperti manhattan atau blockingVehicles, A* tidak hanya tetap menjamin solusi optimal, tetapi juga lebih efisien daripada UCS dalam eksplorasi. Secara teoretis, kompleksitas A* adalah $O(b^d)$, namun empiris menunjukkan ia mengunjungi jauh lebih sedikit simpul karena heuristik memandu pencarian. Ruang yang digunakan sebanding dengan simpul yang dikunjungi, dan runtime keseluruhan umumnya setengah atau lebih kecil dari UCS pada puzzle ukuran menengah.

5.2.4 Algoritma Dijkstra

Dijkstra berperilaku hampir identik dengan UCS ketika semua biaya edge sama, tetapi secara konsep ia menyimpan peta jarak (distance map) ke setiap simpul dan terus memperbarui jarak terpendeknya hingga seluruh graf dianalisis. Dengan implementasi min-heap, kompleksitas Dijkstra adalah $O((V+E) \log V)$, di mana V jumlah keadaan dan E jumlah transisi. Dalam penyelesaian Rush Hour, karena kita berhenti begitu menemukan solusi, Dijkstra sedikit kurang efisien dibanding UCS murni, sebab ia lebih cenderung tetap memperbarui semua status yang belum diekspansi. Meskipun demikian, ia tetap menjamin solusi optimal dan lengkap.

5.2.5 Algoritma Fringe Search

Fringe Search sama seperti A* dengan fungsi evaluasinya adalah $f(n)=g(n)+h(n)$ yang meminimalkan penggunaan memori dan overhead heap. Alih-alih menyimpan semua frontier dalam priority queue, ia memisahkan node menjadi dua daftar, “now” untuk semua state dengan $f(n)$ di bawah batas saat ini dan “later” untuk sisanya, lalu

mengekspansi “now” secara tuntas sebelum menaikkan ambang batas ke nilai f terkecil di “later.” Pendekatan ini menghindari re-expansion berlebihan yang terjadi pada IDA* dan mengurangi footprint memori dibanding A*, karena hanya dua daftar linear yang dikelola. Dari segi kompleksitas, Fringe Search mendekati $O(b^d)$ tergantung seberapa informatif heuristik h , dengan b branching factor dan d kedalaman solusi. Dalam konteks Rush Hour, dengan heuristik admissible seperti Manhattan atau blockingVehicles, Fringe Search tetap menjamin solusi optimal dan biasanya mengunjungi lebih sedikit node daripada UCS atau A*, sambil menggunakan memori jauh lebih sedikit dibanding A*.

5.2.6 Analisis Hasil Pengujian

Berdasarkan Test Case A, A* dengan heuristik Combined secara signifikan mendapatkan solusi lebih baik dengan waktu eksekusi hanya sekitar 0,70 ms dengan 13 node dibangkitkan, dan berhasil menemukan jalur optimal sepanjang 5 langkah. Sebaliknya, UCS, yang hanya mengandalkan biaya kumulatif menghasilkan 426 node, yang menjadi tertinggi di antara algoritma lainnya, meski jalur yang ditemukan tetap optimal. Greedy Best-First Search dengan heuristik Manhattan menunjukkan performa paling buruk pada Test Case A, yaitu 10,00 ms, karena ia mengejar target berdasarkan estimasi jarak saja tanpa menjamin biaya total terendah sehingga sering tersesat ke jalur panjang (solusi tidak optimal). Greedy dengan heuristik Combined sedikit lebih baik dalam hal waktu, namun tetap suboptimal karena sama-sama mengabaikan biaya sejauh ini.

Pada Test Case D, algoritma Fringe Search mendapatkan solusi paling cepat dan node paling sedikit, yaitu dengan waktu 292,00 ms dan 13.363 count. Namun, UCS juga menghasilkan node terbanyak, yaitu 67.907 karena eksplorasi frontier yang luas tanpa arahan heuristik. Greedy Best-First Search dengan heuristik Manhattan Distance mendapatkan solusi paling lambat dengan waktu 918,40 ms dan, seperti pada Test Case A, gagal mencapai solusi optimal sebanyak 44 langkah. A* dengan heuristik Manhattan memang lebih cepat daripada UCS/Dijkstra, tetapi karena heuristiknya kurang akurat cenderung melewatkannya jalur terbaik sehingga solusi akhirnya tidak optimal.

Secara umum, perbedaan ini mencerminkan trade-off mendasar antara eksplorasi dan optimalitas:

1. UCS/Dijkstra menjamin jalur biaya terendah (optimal), tetapi harus menjelajahi banyak node sehingga memakan waktu dan memori lebih besar.
2. Greedy Best-First sangat cepat bila heuristik “mengarah” dengan baik, namun karena tidak mempertimbangkan biaya sejauh ini, ia rentan memberikan jalur suboptimal dan dalam kasus heuristik yang membuat “tersesat”, bahkan menjadi yang paling lambat.
3. A*/Fringe Search memadukan keduanya dengan heuristik admissible yang semakin akurat terutama Blocking Vehicle dan Combined, keduanya meminimalkan eksplorasi sambil tetap menjamin optimalitas.

Oleh karena itu, untuk puzzle Rush Hour yang sangat rumit, Fringe Search dengan heuristik Blocking Vehicles menjadi salah satu opsi paling menjanjikan, karena mengombinasikan efisiensi eksplorasi, penggunaan memori yang moderat, dan garansi solusi optimal.

BAB V

PENJELASAN BONUS

Pertama, selain Uniform Cost Search, Greedy Best-First Search, dan A*, kami juga mengimplementasikan algoritma Dijkstra dan Fringe Search sebagai alternatif pathfinding. Dalam implementasi ini, kami menggunakan struktur data khusus berupa distanceMap yang memetakan setiap status papan permainan ke biaya terendah yang diketahui untuk mencapai status tersebut. Dijkstra menggunakan priority queue untuk mengeksplorasi node-node berdasarkan biaya terendah, mirip dengan UCS, tetapi dengan pendekatan berbeda dalam melacak node yang dikunjungi. Ketika menemukan jalur lebih pendek ke suatu status, algoritma memperbarui peta jarak dan menambahkan status baru ke antrian prioritas. Dengan pendekatan ini, Dijkstra menjamin solusi optimal dengan kompleksitas waktu $O((V+E)\log V)$ dimana V adalah jumlah status dan E adalah transisi yang mungkin.

Kami juga menerapkan Fringe Search, algoritma seperti A* yang menghindari penggunaan priority queue besar dan alih-alih bekerja dalam “batch” berdasarkan ambang nilai $f(n)=g(n)+h(n)$. Pada setiap iterasi, semua node dengan $f(n) \leq f\text{Limit}$ diekspansi secara tuntas dengan disimpan dalam daftar now, sedangkan sisanya ditunda ke daftar later. Ketika daftar now habis, ambang fLimit ditingkatkan menjadi nilai f terkecil pada daftar later, kemudian node yang tepat memenuhi cutoff dipindahkan ke now, dan proses dilanjutkan. Dengan struktur dua daftar ini, Fringe Search meminimalkan overhead heap, menggunakan memori lebih sedikit daripada A*, tetapi tetap mengutamakan node berbiaya rendah plus heuristik. Pendekatan ini menjamin solusi optimal (jika heuristik admissible) dengan eksplorasi yang seringkali lebih sedikit dan footprint memori lebih moderat, sehingga cocok untuk puzzle Rush Hour yang kompleks.

Kedua, kami mengimplementasikan beberapa fungsi heuristik yang dapat dipilih pengguna untuk algoritma informed search (A*, Greedy BFS, Fringe Search). Fungsi-fungsi ini mencakup heuristik yang menghitung jumlah kendaraan yang menghalangi mobil target ke pintu keluar, heuristik yang mempertimbangkan jarak kendaraan utama ke pintu keluar, dan heuristik yang menggabungkan keduanya dengan pembobotan tertentu. Semua fungsi heuristik dirancang untuk memberikan perkiraan biaya menuju status tujuan, dengan fungsi pertama dan kedua bersifat

admissible (tidak akan melebih-lebihkan biaya sebenarnya), sementara fungsi gabungan memberikan panduan pencarian yang lebih agresif namun mungkin tidak menjamin optimalitas pada A*. Pengguna dapat memilih fungsi heuristik sesuai preferensi untuk menyeimbangkan kecepatan dan optimalitas.

Terakhir, kami mengembangkan antarmuka pengguna grafis menggunakan React dan Vite, memungkinkan interaksi visual dengan puzzle Rush Hour. GUI kami terdiri dari beberapa bagian utama, yaitu HomeHead untuk judul dan informasi umum, InputSection untuk menerima input konfigurasi papan permainan, bagian untuk menampilkan papan permainan, SolutionControl untuk mengatur solusi, dan ResultStat untuk menampilkan statistik kinerja algoritma. Pengguna dapat memilih algoritma (UCS, A*, Greedy, Dijkstra, Fringe Search) dan fungsi heuristik (jika menggunakan algoritma terinformasi) melalui dropdown menu. Setelah algoritma berjalan, solusi divisualisasikan langkah demi langkah, dengan metrik kinerja seperti jumlah node dikunjungi dan waktu eksekusi ditampilkan secara real-time.

BAB VI

KESIMPULAN

Melalui pengerjaan Tugas Kecil 3 ini, kami telah berhasil mengembangkan sebuah aplikasi berbasis web untuk memecahkan masalah penyelesaian permainan Rush Hour dengan menerapkan algoritma Uniform Cost Search (UCS), Greedy Best-First Search, A*, Dijkstra, dan Fringe Search beserta heuristic, seperti manhattan distance, blocking vehicle, dan combine kedua heuristic tersebut. Hasil pengujian yang kami lakukan menunjukkan bahwa aplikasi dapat menyelesaikan penyelesaian permainan untuk mendapatkan solusi (jika ada) dengan waktu eksekusi dan jumlah node yang ditelusuri ditampilkan kepada pengguna serta langkah visualisasi solusi.

BAB VIII

LAMPIRAN

6.1 Daftar Pustaka

- Maulidevi, Nur Ulfa 2025. “Penentuan Rute (Route/Path Planning) Bagian 1: BFS, DFS, UCS, Greedy Best First Search” ([Penentuan Rute \(Route/Path Planning\)](#)), diakses 16 Mei 2025).
- Munir, Rinaldi dan Maulidevi, N. U. 2025. “Penentuan Rute (Route/Path Planning) Bagian 2: Algoritma A*” ([Penentuan Rute \(Route/Path Planning\)](#)), diakses 16 Mei 2025).
- Trivusi. 2022. “Apa Itu Algoritma Uniform-Cost Search?” Trivusi. (<https://www.trivusi.web.id/2022/10/apa-itu-algoritma-uniform-cost-search.html>, diakses 16 Mei 2025).
- GeeksforGeeks. 2024. “Greedy Best-First Search Algorithm.” GeeksforGeeks. (<https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/>, diakses 16 Mei 2025).
- Wikipedia. 2024. “Fringe Search” Wikipedia. (https://en.wikipedia.org/wiki/Fringe_search, diakses 18 Mei 2025).

6.2 Tautan Repository

Link repository dari Tugas Kecil 3 IF2211 Strategi Algoritma adalah sebagai berikut:

https://github.com/yonatan-nyo/Tucil3_13523008_13523036

6.3 Tautan Deployment

Link website dari Tugas Kecil 3 IF2211 Strategi Algoritma adalah sebagai berikut:

<https://rushhour-solver.vercel.app/>

6.4**Tabel Checklist**

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	✓	