

**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\*, dan Dijkstra. 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 pelaksanaan 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

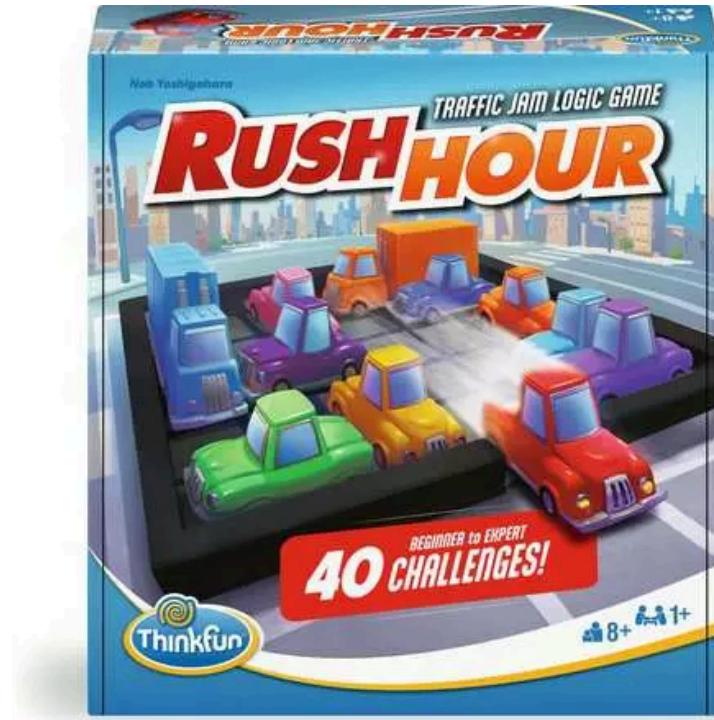
<b>KATA PENGANTAR</b>	<b>2</b>
<b>DAFTAR ISI</b>	<b>3</b>
<b>DAFTAR GAMBAR</b>	<b>5</b>
<b>BAB I</b>	<b>6</b>
<b>PENJELASAN ALGORITMA</b>	<b>6</b>
2.1 Deskripsi Tugas	6
2.2 Algoritma Uniform-Cost Search (UCS)	9
2.3 Algoritma Greedy Best First Search (GBFS)	12
2.4 Algoritma A*	15
2.5 Algoritma Dijkstra	18
<b>BAB II</b>	<b>21</b>
<b>ANALISIS PEMECAHAN MASALAH</b>	<b>21</b>
3.1 Analisis Penyelesaian Puzzle Rush Hour	21
3.2 Fungsi Evaluasi $f(n)$ dan $g(n)$	22
3.3 Keadmissible-an Heuristik pada Algoritma A*	23
3.4 Perbandingan UCS dan BFS pada Penyelesaian Rush Hour	23
3.5 Efisiensi A* dibanding UCS pada Rush Hour	24
3.6 Apakah Greedy Best-First Search Menjamin Solusi Optimal?	25
<b>BAB III</b>	<b>26</b>
<b>SOURCE PROGRAM</b>	<b>26</b>
4.1 Type	26
4.2 Algoritma	28
4.3 Lainnya	33
<b>BAB IV</b>	<b>47</b>
<b>PENGUJIAN</b>	<b>47</b>
5.1 Hasil Pengujian	47
5.2 Analisis Hasil Pengujian	56
5.2.1 Uniform Cost Search (UCS)	56
5.2.2 Greedy Best-First Search (GBFS)	56
5.2.3 A* Search	57
5.2.4 Algoritma Dijkstra	57
5.2.5 Analisis Hasil Pengujian	57
<b>BAB V</b>	<b>59</b>
<b>PENJELASAN BONUS</b>	<b>59</b>

<b>BAB VI</b>	<b>61</b>
<b>KESIMPULAN</b>	<b>61</b>
<b>BAB VIII</b>	<b>62</b>
<b>LAMPIRAN</b>	<b>62</b>
6.1 Daftar Pustaka	62
6.2 Tautan Repository	62
6.3 Tautan Deployment	62
6.4 Tabel Checklist	62

# 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

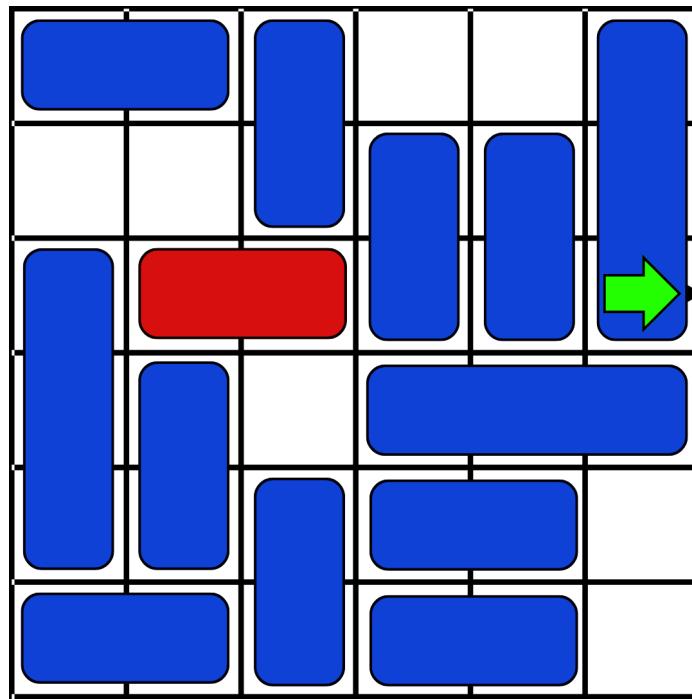
*horizontal* atau *vertikal*.

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.

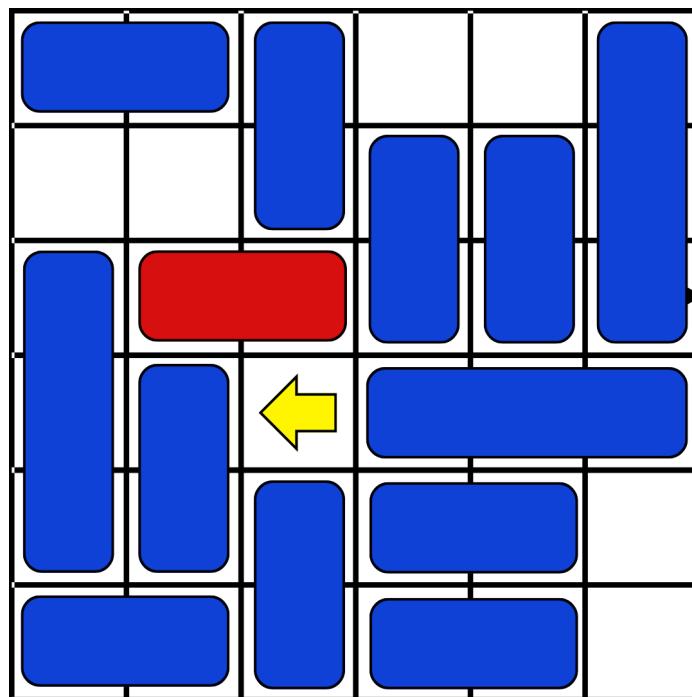
Ilustrasi kasus :

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

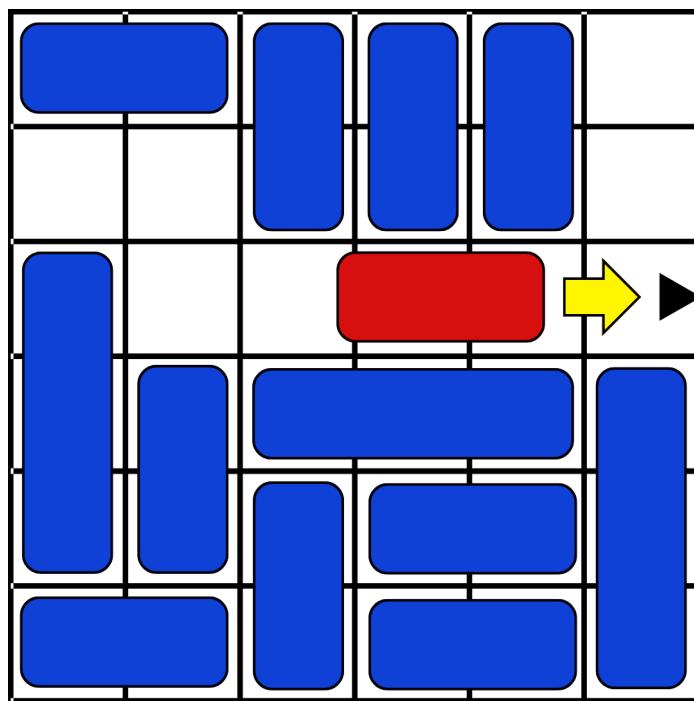
Pemain dapat menggeser-geser *piece* (termasuk *primary piece*) untuk membentuk jalan lurus antara *primary piece* dan *pintu keluar*.



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  $\varepsilon$  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 * 35
    startTime ← current time
    nodesVisited ← 0

    visitedStates ← empty set      # menyimpan state yang sudah diproses
    openList ← empty min-heap / priority queue ordered by cost

    initialState:
        board ← initialBoard
        pieces ← initialPieces
```

```

stateString ← stringify(initialBoard)
cost ← 0           # g = angka langkah dari start
moves ← []         # list langkah yang telah diambil

openList.push(initialState)

# 2. Cari solusi terpendek
while openList is not empty:
    nodesVisited ← nodesVisited + 1

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

    if current.stateString in visitedStates:
        continue      # lewati jika sudah diproses

    visitedStates.add(current.stateString)

    if isSolved(current.pieces):
        # ketemu solusi optimal pertama kali → ini langsung optimal
        endTime ← current.time
        return Solution(
            solved = true,
            moves = current.moves,
            nodesVisited = nodesVisited,
            executionTime = endTime - startTime
        )

    if current.cost > MAX_COST:
        continue      # biaya terlalu besar, abaikan cabang ini

    # 3. Generate tetangga (valid moves)
    for each move in getValidMoves(current.board, current.pieces):
        nextBoard, nextPieces ← applyMove(current.board, current.pieces, move)
        nextStateString ← stringify(nextBoard)

        if nextStateString in visitedStates:
            continue      # sudah diproses, skip

        newCost ← current.cost + 1 # tiap move dihitung 1 langkah
        newMoves ← current.moves + [move]

        nextState:
            board ← nextBoard
            pieces ← nextPieces

```

```

stateString ← nextStateString
cost ← newCost
moves ← newMoves

openList.push(nextState) # enqueue untuk diproses nanti

# 4. Jika openList habis tanpa menemukan solusi
endTime ← current time
return Solution(
    solved = false,
    moves = [],
    nodesVisited = nodesVisited,
    executionTime = endTime – startTime
)

```

### 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 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 mengejar node yang kelihatannya paling menjanjikan tanpa mempertimbangkan jalur sebelumnya.

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 * 35
    startTime ← current time
    nodesVisited ← 0

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

    initialState:
        board ← initialBoard
        pieces ← initialPieces
        stateString ← stringify(initialBoard)
        heuristic ← heuristicFunc(initialBoard, initialPieces) # nilai h awal
        cost ← 0           # g = 0 langkah
        moves ← []         # jalur kosong dari start

    openList.push(initialState)

    # 2. Loop utama pencarian
    while openList is not empty:
```

```

nodesVisited ← nodesVisited + 1

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

if current.stateString in visitedStates:
    continue      # skip state yang sudah diproses

visitedStates.add(current.stateString)

if isSolved(current.pieces):
    # Ketemu goal pertama → tidak menjamin optimal cost,
    # tapi solusi ditemukan sesuai heuristik greedy
    endTime ← current.time
    return SolutionResult(
        solved = true,
        moves = current.moves,
        nodesVisited = nodesVisited,
        executionTime = endTime - startTime
    )

if current.cost > MAX_COST:
    continue      # cegah looping tak terbatas

# 3. Generate dan enqueue tetangga
for each move in getValidMoves(current.board, current.pieces):
    nextBoard, nextPieces ← applyMove(current.board, current.pieces, move)
    nextStateString ← stringify(nextBoard)

    if nextStateString in visitedStates:
        continue      # skip yang sudah dikunjungi

    newCost ← current.cost + 1
    newHeuristic ← heuristicFunc(nextBoard, nextPieces)

    if newCost ≤ MAX_COST:
        nextState:
            board ← nextBoard
            pieces ← nextPieces
            stateString ← nextStateString
            heuristic ← newHeuristic
            cost ← newCost
            moves ← current.moves + [move]
            openList.push(nextState)

```

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

## 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 * 35
    startTime ← current time
    nodesVisited ← 0

    visitedStates ← empty set      # untuk mengecek node yang sudah diproses
    openList ← empty min-heap ordered by f (g + h)

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

    openList.push(initialState)

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

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

        if current.stateString in visitedStates:
            continue      # skip jika sudah diproses

        visitedStates.add(current.stateString)

        if isSolved(current.pieces):
            endTime ← current time
            return SolutionResult(
                solved = true,
                moves = current.moves,
                nodesVisited = nodesVisited,
                executionTime = endTime - startTime
            )

        if current.cost > MAX_COST:
```

```

        continue          # guard untuk kasus loop tak berujung

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

    if nextStateString in visitedStates:
        continue          # skip yang sudah diproses

    newCost ← current.cost + 1  # g + 1 per langkah
    if newCost > MAX_COST:
        continue          # skip jika melebihi batas

    newHeuristic ← heuristicFunc(nextBoard, nextPieces) # h untuk state baru
    newF ← newCost + newHeuristic

    nextState:
        board ← nextBoard
        pieces ← nextPieces
        stateString ← nextStateString
        cost ← newCost
        heuristic ← newHeuristic
        f ← newF
        moves ← current.moves + [move]

    openList.push(nextState)    # enqueue untuk dieksplorasi nanti

# 4. Jika openList habis tanpa solusi
endTime ← current time
return SolutionResult(
    solved = false,
    moves = [],
    nodesVisited = nodesVisited,
    executionTime = endTime – startTime
)

```

## 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 ( $\infty$ ) 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 * 35
    startTime ← current time
    nodesVisited ← 0

    distanceMap ← empty map<string, number> # menyimpan biaya terendah untuk
    setiap stateString
    openList ← empty min-heap ordered by cost

    initialState:
        board ← initialBoard
        pieces ← initialPieces
        stateString ← stringify(initialBoard)
        cost ← 0                      # jarak/g dari start
        moves ← []                      # jalur kosong

    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 kita sudah menemukan jalur lebih baik ke state ini, skip
        if current.cost > distanceMap[currKey]:
            continue

        # Cek apakah sudah mencapai goal
        if isSolved(current.pieces):
            endTime ← current.time
            return SolutionResult(
                solved = true,
                moves = current.moves,
                nodesVisited = nodesVisited,
                executionTime = endTime - startTime
            )

    # Guard untuk mencegah eksplorasi tak terduga
```

```

if current.cost > MAX_COST:
    continue

# 3. Generate dan proses tetangga
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 cost = 1

    # Jika jalur ini lebih pendek dari yang pernah kita catat
    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 solusi
endTime ← current time
return SolutionResult(
    solved = false,
    moves = [],
    nodesVisited = nodesVisited,
    executionTime = endTime – startTime
)

```

## **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 terkласifikasi yang memuat posisi setiap kendaraan. Dengan cara tersebut, kami dapat membuat setiap operasi move (geser kendaraan satu satuan) untuk menciptakan state baru yang masih mudah dibedakan dan dibandingkan. Setelah representasi konfigurasi papan sudah siap, kami menggunakan algoritma *pathfinding* (UCS, Greedy Best-First, A\*, atau Dijkstra) yang akan menelusuri ruang state tersebut demi menemukan urutan langkah yang membawa mobil utama keluar melalui pintu keluar.

Inti perbedaan 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\* atau Greedy 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\* atau UCS 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.

Dengan definisi tersebut, terlihat bahwa UCS, A\*, Dijkstra 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 penggerjaan 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 penyelesaian Rush Hour, di mana setiap gerakan kendaraan memiliki biaya seragam, yaitu satu langkah per pergerakan, Uniform Cost Search (UCS) pada dasarnya akan mengeksplorasi status papan berdasarkan kenaikan jumlah langkah persis seperti Breadth-First Search (BFS). Artinya, baik UCS maupun BFS akan menemukan solusi dengan jumlah langkah minimal yang sama, karena keduanya memperluas simpul “level demi level” jika setiap edge dianggap memiliki bobot 1.

Namun, secara konseptual dan dalam hal implementasi yang telah kami lakukan, UCS tidak identik dengan BFS. Pada BFS, simpul baru selalu dimasukkan dan diambil dari antrian FIFO (first-in, first-out) tanpa mempertimbangkan nilai biaya, sehingga eksplorasi benar-benar mengikuti urutan kedatangan. Sementara itu, UCS menggunakan priority queue yang mengurutkan status berdasarkan nilai  $g(n)$ , dalam kasus ini sama dengan kedalaman simpul dan ketika dua status memiliki biaya yang sama, urutan pengembangannya akan bergantung pada mekanisme tie-breaking dari implementasi PriorityQueue. Akibatnya, meski panjang jalur output dan optimalitas yang dijamin oleh kedua algoritma sama, urutan internal simpul yang dibangkitkan oleh UCS bisa berbeda dari urutan BFS murni. Dengan demikian, dalam konteks Rush Hour UCS dapat dikatakan “berperilaku seperti” BFS dalam hal hasil akhir dan kedalaman eksplorasi, tetapi secara struktural dan teknis tetap berbeda.

Karena UCS menyimpan frontier dalam priority queue menurut  $g(n)$ , ia bisa saja memproses ulang dua state pada kedalaman yang sama dalam urutan berbeda dibanding BFS.

1. Pada level 2, terdapat dua state, A dan B, keduanya dengan  $g(n)=2$ .
2. Dalam BFS murni, simpul dimasukkan ke antrean secara FIFO, sehingga jika A di-enqueue sebelum B, maka BFS akan memproses  $A \rightarrow B$ .
3. Sedangkan UCS akan mempop node dengan  $g=2$ , tapi antara A dan B, jika tie-breaking heap-nya menganggap B terletak di posisi internal yang lebih dulu, sehingga UCS dapat memproses  $B \rightarrow A$ , meski cost-nya 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&lt;string, PieceInfo&gt;;</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";     ...   </pre>
Heuristic	<pre>     ... export type Heuristic = "manhattan"   "blockingVehicles"   "combined";     ...   </pre>

## 4.2 Algoritma

Algoritma	Tampilan Kode
-----------	---------------

## UCS

```
    /**
     * Implementasi untuk status pencarian dalam algoritma UCS
     */
    interface SearchState {
        board: string[]; // Konfigurasi papan saat ini
        pieces: PiecesMap; // Inferensi tentang posisi Kondom pada papan
        stateString: string; // Representasi string untuk satu status
        cost: number; // Biaya total solusi (jumlah langkah)
        moves: Move[]; // Daftar langkah untuk mencapai solusi
    }

    /**
     * Antonioku untuk status pencarian dalam algoritma UCS
     */
    interface Solution {
        cost: number; // Biaya total solusi (jumlah langkah)
        moves: Move[]; // Daftar langkah untuk mencapai solusi
    }

    /**
     * Implementasi algoritma Uniform Cost Search (UCS) untuk menemukan solusi optimal
     * UCS memiliki node dengan biaya terendah untuk dieksplorasi, memjamin selusi optimal
     * @param initialBoard Konfigurasi papan awal
     * @param initialPieces Inferensi kondom awal pada papan
     */
    const ucs = (initialBoard: string[], initialPieces: PiecesMap): SolutionResult => {
        // Definisi batas maksimum biaya untuk menengah loop tak terhingga
        // Buktikan berdasarkan ukuran papan (misalnya 8x8 = 64)
        const MAX_COST = initialBoard.length + initialBoard[0].length * 25;

        // Catat waktu mulai untuk mengetahui durasi eksekusi
        const start = performance.now();

        // Set untuk mengetahui status yang sudah dikunjungi
        const visitedStates = new Set<string>();

        // Menggunakan prioritas queue untuk menilai status yang efisien
        // Biaya ditambahkan dengan jumlah langkah
        const queueList = new PriorityQueue<SearchState>((a, b) => a.cost + b.cost);

        // Menghitung jumlah simpul yang dikunjungi
        let nodesVisited = 0;

        // Inisialisasi status awal
        const initialState: SearchState = {
            board: initialBoard,
            pieces: initialPieces,
            stateString: getBoardStateString(initialBoard), // Mengonversi board menjadi string unik
            cost: 0, // Jumlah langkah min
            moves: []
        };

        // Simpan status awal ke dalam antrean
        openList.push(initialState);

        // Variabel untuk menyimpan solusi terbaik yang ditemukan
        let bestSolution: Solution = {
            cost: Infinity, // Solusi terbaik dengan biaya tak terhingga
            moves: []
        };

        // Loop utama pencarian
        while (!openList.isEmpty()) {
            nodesVisited++;

            // Ambil status dengan biaya terendah
            const currentState = openList.pop();

            // Lakukan jika status ini sudah dikunjungi sebelumnya
            if (visitedStates.has(currentState.stateString)) continue;
            visitedStates.add(currentState.stateString);

            // Periksa apakah status cost keadaan solusi
            if (isSolved(currentState.pieces)) {
                // Persetuju solusi terbaik jika ini adalah solusi dengan biaya lebih rendah
                if (!bestSolution || currentState.cost < bestSolution.cost) {
                    bestSolution = currentState;
                    bestSolution.cost = currentState.cost;
                    bestSolution.moves = [...currentState.moves];
                }
            }

            // Hitung waktu eksekusi dan kombinasi solusi
            const end = performance.now();
            return {
                solved: true,
                moves: currentState.moves, // Daftar langkah menuju solusi
                nodesVisited, // Jumlah node yang diperiksa
                executionTime: end - start, // Waktu eksekusi dalam milidik
                cost: currentState.cost
            };
        }

        // Lakukan jika biaya sudah melebihi batas maksimum
        if (currentState.cost > MAX_COST) continue;

        // Dapatkan semua langkah yang valid dari status saat ini
        const validMoves = getValidMoves(currentState.board, currentState.pieces);

        // Iterasi melalui semua langkah yang mungkin
        for (const move of validMoves) {
            // Tampilkan status dan daftar langkah saat ini
            const result = applyMove(currentState.board, currentState.pieces, move);
            const newStateString = getBoardStateString(result.board);

            // Lakukan status yang sudah dibentuk sebelumnya
            if (!visitedStates.has(newStateString)) continue;

            // Buat status baru dengan biaya yang diperbarui
            const nextState: SearchState = {
                board: result.board,
                pieces: result.pieces,
                stateString: newStateString,
                cost: currentState.cost + 1, // Biaya bertambah 1 untuk setiap langkah
                moves: [...currentState.moves], // Tambahan langkah baru ke daftar langkah
                moves: [...currentState.moves, move]
            };

            // Masukkan status baru ke dalam antrean
            openList.push(nextState);
        }
    };

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

export default ucs;
```

## Greedy

```
***  
import { applyMove, getBoardStateString, getValidMoves, isSolved } from "../helpers";  
import { PriorityQueue } from "./PriorityQueue";  
import type { Move, PieceMap, SolutionResult } from "./types";  
  
// status pencarian dalam algoritma greedy  
interface SearchState {  
    board: string[12]; // Konfigurasi papan saat ini  
    pieces: PieceMap; // Informasi tentang posisi kendaraan pada papan  
    stateString: string; // Representasi string unik dari status papan (untuk pengecekan status  
    validitas dan konsistensi)  
    heuristic: number; // Nilai heuristik untuk status saat ini  
    moves: Move[]; // Daftar langkah yang telah diambil untuk mencapai status ini  
    cost: number; // Biaya/jumlah langkah untuk mencapai status ini  
}  
  
/*  
 * Implementasi algoritma Greedy Best-First Search untuk menyelesaikan puzzle Rush Hour.  
 * Algoritma ini selalu memilih status dengan nilai heuristik terendah untuk dieksplorasi  
 * selanjutnya.  
 *  
 * @param initialBoard Konfigurasi papan awal.  
 * @param initialPieces Informasi kendaraan awal pada papan.  
 * @param heuristicFunc Fungsi heuristik yang digunakan untuk memilih jalur ke solusi  
 */  
const greedy = (initialBoard: string[], initialPieces: PieceMap, heuristicFunc: (board:  
string[12], pieces: PieceMap) => number): SolutionResult => {  
    const start = performance.now();  
    let moves: Move[] = [];  
    let cost: number = 0;  
    let heuristic: number = heuristicFunc(initialBoard, initialPieces);  
    let end: number = 0;  
    let openList: PriorityQueue<SearchState> = new PriorityQueue();  
    let visitedStates: Set<string> = new Set();  
    let nodeVisited = 0;  
  
    // Inisialisasi status awal  
    const initialState: SearchState = {  
        board: initialBoard,  
        pieces: initialPieces,  
        stateString: getBoardStateString(initialBoard), // Mengonversi board menjadi string unik  
        heuristic: heuristicFunc(initialBoard, initialPieces), // Menghitung nilai heuristik awal  
        moves: [], // Belum ada langkah yang diambil  
        cost: 0, // Biaya awal adalah 0  
    };  
  
    // Masukkan status awal ke dalam antrean  
    openList.push(initialState);  
  
    // Loop utama pencarian  
    while (!openList.isEmpty()) {  
        const currentState = openList.pop();  
        if (!nodeVisited) {  
            console.log(`Node ${nodeVisited} visited`);  
        }  
  
        // Cek jika status ini sudah dikunjungi sebelumnya  
        if (!visitedStates.has(currentState.stateString)) {  
            visitedStates.add(currentState.stateString);  
  
            // Periksa apakah status saat ini adalah solusi  
            if (isSolved(currentState.pieces)) {  
                const end = performance.now();  
                return {  
                    solved: true,  
                    moves: currentState.moves, // Daftar langkah menuju solusi  
                    nodesVisited: nodeVisited, // Jumlah node yang diperiksa  
                    executionTime: end - start, // Waktu eksekusi dalam milidetik  
                };  
            }  
  
            // Lebih jika biaya sudah melebihi batas maksimum  
            if (currentState.cost > MAX_COST) {  
                continue;  
            }  
  
            // Dapatkan semua langkah yang valid dari status saat ini  
            const validMoves = getValidMoves(currentState.board, currentState.pieces);  
  
            // Iterasi melalui semua langkah yang mungkin  
            for (const move of validMoves) {  
                // Periksa apakah langkah tersebut belum pernah dilakukan sebelumnya  
                const { board: nextBoard, pieces: newPieces } = applyMove(currentState.board,  
                currentState.pieces, move);  
                const newStateString = getBoardStateString(nextBoard);  
  
                // Lebih jika biaya sudah melebihi batas maksimum  
                if (newCost <= MAX_COST) {  
                    const newSearchSete = {  
                        board: nextBoard,  
                        pieces: newPieces,  
                        stateString: newStateString,  
                        heuristic: heuristicFunc(nextBoard, newPieces),  
                        moves: [...currentState.moves, move], // Tambahkan langkah baru ke daftar langkah  
                        cost: currentState.cost + 1  
                    };  
  
                    // Masukkan status baru ke dalam antrean  
                    openList.push(newState);  
                }  
            }  
        }  
        // Cetak antrean ketika dan tidak ditemukan solusi  
        const end = performance.now();  
        return {  
            solved: false,  
            moves: [],  
            nodesVisited: nodeVisited,  
            executionTime: end - start,  
        };  
    };  
    export default greedy;
```

## A\*

```
/// static implementasi algoritma A*
function searchAStar() {
    board: string[]; // Konfigurasi papan saat ini
    pieces: PieceMap; // Representasi string unik dari status papan
    cost: number; // Biaya/jumlah langkah yang telah diambil (nilai g)
    heuristic: number; // Nilai heuristik untuk status saat ini (nilai h)
    f: number; // Nilai total f = g + h, yang digunakan untuk prioritas dalam pencarian
    moves: Move[]; // Daftar langkah yang telah dimulai untuk mencapai status ini
}

/*
 * Implementasi algoritma A* Search untuk menyelesaikan puzzle Rush Hour
 * & menggunakan kombinasi biaya sejauh ini (g) dan estimasinya biaya ke tujuan (h)
 * & menggunakan fungsi heuristik yang dikenal sebagai fungsi Manhattan
 * & awalan initialBoard menginisiasi papan awal
 * & param initialPieces informasi kendaraan awal pada papan
 * & param heuristicFunc fungsi heuristik yang digunakan untuk menilai jarak ke solusi
 */
const astar = (initialBoard: string[][], initialPieces: PieceMap, heuristicFunc: (board: string[][], pieces: PieceMap) => number): SolutionResult | null => {
    const start = performance.now();
    const MAX_COST = initialBoard.length * initialBoard[0].length * 25;
    const openList = new PriorityQueue<SearchState>();
    const startNode = openList.push(initialState);
    const visitedStates = new Set<string>();
    const nodesVisited = 0;

    let currentState: SearchState;
    const initialState: SearchState = {
        board: initialBoard,
        pieces: initialPieces,
        stateString: getBoardStateString(initialBoard), // Mengkonversi board menjadi string unik
        cost: 0, // Biaya awal adalah 0
        heuristic: heuristicFunc(initialBoard, initialPieces), // Menghitung nilai heuristik awal
        f: heuristicFunc(initialBoard, initialPieces), // Nilai f awal sama dengan heuristik
        moves: [], // Belum ada langkah yang dimiliki
        solved: false
    };

    openList.push(initialState);

    while (!openList.isEmpty()) {
        nodesVisited++;
        const currentCost = openList.pop();
        const currentMove = currentCost.moves;
        const currentString = currentCost.stateString;

        if (visitedStates.has(currentString)) continue;
        visitedStates.add(currentString);

        if (currentString === goalString) {
            return {
                solved: true,
                moves: currentMove, // Daftar langkah menuju solusi
                nodesVisited,
                executionTime: end - start, // Waktu eksekusi dalam milidetik
                board: []
            };
        }

        for (const move of validMoves) {
            const newBoard = cloneBoard(currentState.board);
            const newPieces = clonePieceMap(currentState.pieces);
            const newCost = currentCost.cost + 1; // Biaya bertambah 1 untuk setiap langkah
            const newStateString = getBoardStateString(newBoard);
            const newMoveString = getBoardStateString(newState);

            if (visitedStates.has(newStateString)) continue;

            const newHeuristic = heuristicFunc(newState, newPieces);
            const newCost += newHeuristic; // Jumlah biaya yang diperlukan
            const newState = {
                board: newBoard,
                pieces: newPieces,
                stateString: newStateString,
                cost: newCost,
                heuristic: newHeuristic,
                f: newCost + newHeuristic,
                moves: [...currentState.moves, move]
            };

            if (newCost > MAX_COST) continue;

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

            openList.push(newState);
        }
    }

    return {
        solved: false,
        moves: [],
        nodesVisited,
        executionTime: end - start,
    };
}

export default astar;
```

## Dijkstra

```
● ● ●  
import { applyMove, getBoardStateString, getValidMoves, isSolved } from "../helpers";  
import { PriorityQueue } from "../priorityQueue";  
import type { Move, PieceType, SolutionResult } from "../types";  
  
// status penerapan dalam algoritma Dijkstra  
interface SearchState {  
    board: string[]; // Konfigurasi papan saat ini  
    pieces: PiecesMap; // Informasi tentang semua kendaraan pada papan  
    stateString: string; // Representasi string unik dari status papan  
    cost: number; // Biaya/jarak dari node awal ke node ini  
    moves: Move[]; // Daftar langkah yang telah diambil untuk mencapai status ini  
}  
  
/*  
 * Implementasi algoritma Dijkstra untuk menyelesaikan puzzle Rush Hour  
 * Dijkstra mencari jalur terpendek dengan memprioritaskan node dengan biaya terendah  
 * Misip dengan UCS, tetapi menggunakan pendekatan yang berbeda untuk penerapan node yang dikunjungi  
 *   
 * @param initialBoard Konfigurasi papan awal  
 * @param initialPlaces Informasi kendaraan awal pada papan  
 */  
const dijkstra = (initialBoard: string[], initialPlaces: PiecesMap): SolutionResult => {  
    // Batasan maksimum biaya untuk mencegah loop tak terhingga  
    const MAX_COST = initialBoard.length * initialBoard[0].length * 25;  
  
    // Tabel waktu awal untuk persukuran kinerja  
    const start = performance.now();  
  
    // Map untuk menyimpan biaya terendah yang diketahui untuk setiap state  
    // Berbeda dengan UCS yang menggunakan set sederhana untuk status yang dikunjungi  
    const distanceMap = new Map<string, number>();  
  
    // Priority queue untuk menyimpan state yang akan dieksplorasi  
    // State dengan biaya terendah akan diproses terlebih dahulu  
    const queue = new PriorityQueue<SearchState>((a, b) => a.cost - b.cost);  
  
    // Hitung jumlah node yang dikunjungi untuk statistik  
    let nodesVisited = 0;  
  
    // Inisialisasi state awal  
    const initialState: SearchState = {  
        board: initialBoard,  
        pieces: initialPlaces,  
        stateString: getBoardStateString(initialBoard),  
        cost: 0,  
        moves: []  
    };  
  
    // Tambahkan state awal ke queue dan distance map  
    queue.push(initialState);  
    distanceMap.set(initialState.stateString, 0);  
  
    // Loop penerapan utama - proses state hingga queue kosong  
    while (!queue.isEmpty()) {  
        nodesVisited++;  
  
        // Ambil state dengan biaya terendah  
        const currentState = queue.pop();  
        const currentStateString = currentState.stateString;  
  
        // Jika biaya state ini lebih tinggi dari yang sudah diketahui, lewati  
        // ini adalah optimasi wajib dalam algoritma Dijkstra  
        if (currentState.cost > (distanceMap.get(currentStateString) || Infinity)) {  
            continue;  
        }  
  
        // Periksa apakah state ini adalah solusi  
        if (isSolved(currentState.pieces)) {  
            const end = performance.now();  
            return {  
                solved: true,  
                moves: currentState.moves,  
                nodesVisited,  
                executionTime: end - start,  
            };
        }  
  
        // Lewati jika biaya terlalu tinggi (batasan keamanan)  
        if (currentState.cost > MAX_COST) {  
            continue;  
        }  
  
        // Dapatkan semua langkah yang valid  
        const validMoves = getValidMoves(currentState.board, currentState.pieces);  
  
        // Eksplorasi semua langkah yang valid  
        for (const move of validMoves) {  
            // Tambahkan langkah untuk mendapatkan state baru  
            const [board, newPlace, pieces] = applyMove(currentState.board,  
                currentState.pieces, move);  
            const newStateString = getBoardStateString(board);  
  
            // Hitung biaya baru untuk mencapai state ini  
            const newCost = currentState.cost + 1;  
  
            // Jika ditemukan jalur yang lebih pendek atau state baru belum dikunjungi  
            if (newCost < (distanceMap.get(newStateString) || Infinity)) {  
                // Pindah jalur terpanjang ke state ini  
                distanceMap.set(newStateString, newCost);  
  
                // Buat state baru dan tambahkan ke queue  
                const newState: SearchState = {  
                    board: newStateString,  
                    pieces: newPlace,  
                    stateString: newStateString,  
                    cost: newCost,  
                    moves: [...currentState.moves, move],  
                };  
  
                queue.push(newState);  
            }
        }
    }
    // Jika queue kosong dan tidak ditemukan solusi  
    const end = performance.now();
    return {
        solved: false,
        moves: [],
        nodesVisited,
        executionTime: end - start,
    };
};

export default dijkstra;
```

### 4.3 Lainnya

Lainnya	Tampilan Kode
PriorityQueue	<pre>... export class PriorityQueue&lt;T&gt; {     private items: T[] = [];     private readonly comparator: (a: T, b: T) =&gt; number;      constructor(comparator: (a: T, b: T) =&gt; number) {         this.comparator = comparator;     }      push(item: T): void {         let index = this.items.length;         this.items.push(item);          while (index &gt; 0) {             const parentIndex = Math.floor((index - 1) / 2);             if (this.comparator(this.items[parentIndex], this.items[index]) &lt;= 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 &gt; 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 &lt; this.items.length &amp;&amp; this.comparator(this.items[left], this.items[smallest]) &lt;             0) {             smallest = left;         }          if (right &lt; this.items.length &amp;&amp; this.comparator(this.items[right], this.items[smallest])             &lt; 0) {             smallest = right;         }          if (smallest !== index) {             [this.items[index], this.items[smallest]] = [this.items[smallest], this.items[index]];             this.heapify(smallest);         }     } }</pre>

## Heuristics

```
...
// Enhanced heuristic functions for Rush Hour puzzle
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];

    // For horizontal primary piece (common in Rush Hour)
    if (primaryPiece.orientation === "horizontal") {
      // If exit is on the left, use leftmost position of primary piece
      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);
      }
      // If exit is on the right, use rightmost position
      else {
        const rightPos = primaryPiece.positions[primaryPiece.positions.length - 1];
        return Math.abs(rightPos.row - exitPos.row) + Math.abs(rightPos.col + 1 - exitPos.col);
      }
    }
    // For vertical primary piece
    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];

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

      // Exit on the left side
      if (exitPos.col < primaryPiece.positions[0].col) {
        const leftmostCol = primaryPiece.positions[0].col;
        let count = 0;
        // Count pieces between primary piece and exit
        for (let col = exitPos.col + 1; col < leftmostCol; col++) {
          if (board[row][col] !== "." && board[row][col] !== "K") {
            count++;
          }
        }
        return count;
      }
      // Exit on the right side (original logic)
      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;
      }
    }
    // For vertical primary piece
    const col = primaryPiece.positions[0].col;
    // Exit above primary piece
    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;
    }
    // Exit below primary piece
    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) => {
    // Using a weighted combination of both heuristics
    // Increasing the weight of blocking vehicles which is more impactful
    return heuristics.manhattan(board, pieces) + 5 * heuristics.blockingVehicles(board, pieces);
  },
};


```

## Helpers

```
...  
import { DIRECTIONS } from "./constant";  
import type { BoardConfig, Move, PiecesMap } from "./types";  
  
export const deepCopy = <T>(arr: T[][]): T[][] => arr.map((row) => [...row]);  
  
export const parseInputFile = (content: string): BoardConfig => {  
  const lines = content.replace(/\r/g, "").trim().split("\n");  
  if (lines.length < 2) {  
    throw new Error("Input file must have at least 2 lines (dimensions and numPieces).");  
  }  
  
  const [dimensions, numPieces, ...boardConfig] = lines;  
  const [width, height] = dimensions.split(" ").map(Number);  
  const N = parseInt(numPieces);  
  
  if (!Number.isInteger(width) || !Number.isInteger(height) || width <= 0 || height <= 0) {  
    throw new Error("Invalid board dimensions.");  
  }  
  if (!Number.isInteger(N) || N < 0) {  
    throw new Error("Invalid number of pieces.");  
  }  
  const kLocation = { col: -1, row: -1 };  
  for (let i = 0; i < boardConfig.length; i++) {  
    const row = boardConfig[i];  
    if (row.includes("K")) {  
      if (kLocation.col !== -1) {  
        throw new Error("There should be only one K in the board.");  
      }  
      kLocation.row = i;  
      kLocation.col = row.indexOf("K");  
    }  
  }  
  if (kLocation.col === -1 || kLocation.row === -1) {  
    throw new Error("There should be one K in the board.");  
  }  
  const isKOnTheLeft = kLocation.col === 0 && kLocation.col < height;  
  const isKOnTheTop = kLocation.col <= width && kLocation.row === 0;  
  const isKOnTheRight = kLocation.col === width && kLocation.row < height;  
  const isKOnTheBottom = kLocation.col < width && kLocation.row === height;  
  
  if (!isKOnTheLeft && !isKOnTheTop && !isKOnTheRight && !isKOnTheBottom) {  
    throw new Error("K shouldn't be in the middle of the board.");  
  }  
  
  // Validate boardConfig  
  if (boardConfig.length !== height + (isKOnTheTop || isKOnTheBottom ? 1 : 0)) {  
    throw new Error(`Expected ${height} rows in boardConfig, got ${boardConfig.length}.`);  
  }  
  boardConfig.forEach((row, idx) => {  
    if (isKOnTheLeft) {  
      if (row.length !== width + 1) {  
        throw new Error(`Each row must have ${width + 1} columns.`);  
      }  
    } else if (isKOnTheRight) {  
      if (row.includes("K")) {  
        if (row.length !== width + 1) {  
          throw new Error(`Each row must have ${width + 1} columns.`);  
        }  
      } else {  
        if (row.length !== width && row.length !== width + 1) {  
          throw new Error(`Each row must have ${width} or ${width + 1} columns.`);  
        }  
      }  
    } else if (isKOnTheTop) {  
      if (idx !== 0) {  
        if (row.length !== width) {  
          throw new Error(`Each row must have ${width} columns.`);  
        }  
      }  
    } else if (isKOnTheBottom) {  
      if (idx !== height) {  
        if (row.length !== width) {  
          throw new Error(`Each row must have ${width} columns.`);  
        }  
      }  
    } else {  
      throw new Error(`Unknown K location.`);  
    }  
  });  
  
  return {  
    dimensions: { A: width, B: height },  
    numPieces: N,  
    boardConfig,  
    finishLocation: { col: kLocation.col, row: kLocation.row },  
  };  
};
```

```

const getPlacesInfo = (boardConfig: string[][]) : PiecesMap => {
  const pieces: PieceMap = {};
  const boardMatrix = boardConfig.map((row) => row.split(""));
  const visitors = new Set<string>();
  const usedSymbols = new Set<string>();
  let pieceCount = 0;

  const MAX_PIECES = 24;

  // Helper function to check if a cell should be ignored
  const isIgnoredCell = (cell: string) => cell === " " || cell === "+";
  const isSamePiece = (cell1: string, cell2: string) => {
    if (cell1 === cell2) return true;
    if (isIgnoredCell(cell1) || isIgnoredCell(cell2)) return false;
  };

  // Only compares alaphabetic characters
  const isAlphaChar = (char: string) => /\w[A-Za-z]/.test(char);

  if (!isAlphaChar(cell1) && !isAlphaChar(cell2)) {
    return cell1 === cell2;
  }

  return false;
};

// Find all places and their positions
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;

    // Rule: no piece and no char
    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 curRow = (row: number; col: number) => [row, i, col, j];
    const curVisited = new Set<string>();
    const curVisited.add(posKey);

    while (true) {
      const current = curRow.shift();
      contiguousCells.push(current);
      visited.add(`${current.row},${current.col}`);

      const 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 &&
          !isIgnoredCell(boardMatrix[newRow][newCol]) &&
          !curVisited.has(newPosKey)
        ) {
          curVisited.add(newPosKey);
        }
      }
    }

    pieces[cell] = {
      positions: contiguousCells,
      symbol: cell,
    };
    usedSymbols.add(cell);
    pieceCount++;
  }
}

// Rest of the function remains the same
Object.values(pieces).forEach((piece) => {
  if (piece.isExit) return;
  const { positions } = piece;
  piece.size = positions.length;

  // Intersecting orientation
  if (positions.length > 1) {
    const sameRow = positions.every((pos) => pos.row === piece.positions[0].row);
    piece.orientation = sameRow ? "horizontal" : "vertical";

    // Additional check to ensure the piece is truly continuous
    if (piece.orientation === "horizontal") {
      positions.sort((a) => 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 '$piece.symbol' has non-contiguous horizontal positions. Each piece must form a continuous line.`);
        }
      }
    } else {
      positions.sort((a) => a.row - b.row);
      for (let i = 1; i < positions.length; i++) {
        if (positions[i].row !== positions[i - 1].row + 1) {
          throw new Error(`Piece '$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 const getBoardStateString = (board: string[][]): string => {
  return board.map((row) => row.join("")).join("\n");
};

```

```

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

  // Find exit location
  const exitPiece = pieces["K"];
  const exitPos = exitPiece?.positions[0];

  const isExitOnLeft = exitPos?.col === 0 && exitPos.row < boardHeight;
  const isExitOnTop = exitPos?.row === 0 && exitPos.col < boardWidth;
  const isExitOnRight = exitPos?.col === boardWidth - 1 && exitPos.row < boardHeight;
  const isExitOnBottom = exitPos?.row === boardHeight - 1 && exitPos.col < boardWidth;

  Object.values(pieces).forEach((piece) => {
    if (piece.isExit) return;
    const { orientation, symbol } = piece;
    if (!orientation) return;

    if (orientation === "horizontal") {
      const leftmostPos = piece.positions[0];

      // Check left move - special case for primary piece and exit on left
      if (leftmostPos.col > 0 && !(isExitOnLeft && leftmostPos.col === 1 && !piece.isPrimary)) {
        const leftCell = board[leftmostPos.row][leftmostPos.col - 1];
        if (leftCell === ".") {
          moves.push({ piece: symbol, direction: DIRECTIONS.LEFT });
        } else if (leftCell === "K" && piece.isPrimary) {
          // Allow primary piece to move left into exit
          moves.push({ piece: symbol, direction: DIRECTIONS.LEFT });
        }
      }

      // Check right move
      const rightmostPos = piece.positions[piece.positions.length - 1];
      const rightPos = rightmostPos.col + 1;
      if (rightPos < boardWidth && !(isExitOnRight && rightPos === boardWidth - 1 && !piece.isPrimary)) {
        const rightCell = board[rightmostPos.row][rightPos];
        if (rightCell === ".") {
          moves.push({ piece: symbol, direction: DIRECTIONS.RIGHT });
        } else if (rightCell === "K" && piece.isPrimary) {
          // Allow primary piece to move right into exit
          moves.push({ piece: symbol, direction: DIRECTIONS.RIGHT });
        }
      }
    } else if (orientation === "vertical") {
      const topmostPos = piece.positions[0];

      // Check up move
      if (topmostPos.row > 0 && !(isExitOnTop && topmostPos.row === 1 && !piece.isPrimary)) {
        const topCell = board[topmostPos.row - 1][topmostPos.col];
        if (topCell === ".") {
          moves.push({ piece: symbol, direction: DIRECTIONS.UP });
        } else if (topCell === "K" && piece.isPrimary) {
          // Allow primary piece to move up into exit if exit is at the top
          moves.push({ piece: symbol, direction: DIRECTIONS.UP });
        }
      }

      // Check down move
      const bottommostPos = piece.positions[piece.positions.length - 1];
      if (bottommostPos.row + 1 < boardHeight && !(isExitOnBottom && bottommostPos.row === boardHeight - 2 && !piece.isPrimary)) {
        const bottomCell = board[bottommostPos.row + 1][bottommostPos.col];
        if (bottomCell === ".") {
          moves.push({ piece: symbol, direction: DIRECTIONS.DOWN });
        } else if (bottomCell === "K" && piece.isPrimary) {
          // Allow primary piece to move down into exit if exit is at the bottom
          moves.push({ piece: symbol, direction: DIRECTIONS.DOWN });
        }
      }
    });
  });

  return moves;
};

```

```

    . . .

export const applyMove = (board: string[][][], pieces: PiecesMap, move: Move): { board: string[][]
[]; pieces: PiecesMap } => {
  const newBoard = deepCopy(board);
  const { piece: pieceSymbol, direction } = move;
  const piece = pieces[pieceSymbol];
  if (!piece.orientation) return { board: newBoard, pieces: { ...pieces } };
  const { orientation, positions } = piece;
  const newPositions = [...positions];

  if (orientation === "horizontal") {
    if (direction === DIRECTIONS.LEFT) {
      const rightmostPos = positions[positions.length - 1];
      newBoard[rightmostPos.row][rightmostPos.col] = ".";
      const newLeftPos = { row: positions[0].row, col: positions[0].col - 1 };
      newBoard[newLeftPos.row][newLeftPos.col] = pieceSymbol;
      newPositions.pop();
      newPositions.unshift(newLeftPos);
    } else if (direction === DIRECTIONS.RIGHT) {
      const leftmostPos = positions[0];
      newBoard[leftmostPos.row][leftmostPos.col] = ".";
      const newRightPos = {
        row: positions[positions.length - 1].row,
        col: positions[positions.length - 1].col + 1,
      };
      if (newBoard[newRightPos.row][newRightPos.col] === "K") {
        // keep K visible
      } else {
        newBoard[newRightPos.row][newRightPos.col] = pieceSymbol;
      }
      newPositions.shift();
      newPositions.push(newRightPos);
    }
  } else if (orientation === "vertical") {
    if (direction === DIRECTIONS.UP) {
      const bottommostPos = positions[positions.length - 1];
      newBoard[bottommostPos.row][bottommostPos.col] = ".";
      const newTopPos = { row: positions[0].row - 1, col: positions[0].col };
      newBoard[newTopPos.row][newTopPos.col] = pieceSymbol;
      newPositions.pop();
      newPositions.unshift(newTopPos);
    } else if (direction === DIRECTIONS.DOWN) {
      const topmostPos = positions[0];
      newBoard[topmostPos.row][topmostPos.col] = ".";
      const newBottomPos = {
        row: positions[positions.length - 1].row + 1,
        col: positions[positions.length - 1].col,
      };
      newBoard[newBottomPos.row][newBottomPos.col] = pieceSymbol;
      newPositions.shift();
      newPositions.push(newBottomPos);
    }
  }

  const newPieces = { ...pieces };
  newPieces[pieceSymbol] = {
    ...piece,
    positions: newPositions,
    primaryPosition: newPositions[0],
  };

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

// Fixed isSolved function for Rush Hour puzzles
export 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];

  // For a typical Rush Hour puzzle with horizontal primary piece:
  if (primaryPiece.orientation === "horizontal") {
    // Get leftmost position of primary piece
    const leftmostPos = primaryPiece.positions[0];

    // If exit is on the left side, primary piece should be adjacent to exit
    if (exitPos.col < leftmostPos.col) {
      return leftmostPos.row === exitPos.row && leftmostPos.col === exitPos.col + 1;
    }
    // If exit is on the right side
    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;
    }
  }
  // For vertical primary piece (less common but possible)
  else if (primaryPiece.orientation === "vertical") {
    const topmostPos = primaryPiece.positions[0];
    const bottommostPos = primaryPiece.positions[primaryPiece.positions.length - 1];

    // Exit at the top
    if (exitPos.row < topmostPos.row) {
      return topmostPos.col === exitPos.col && topmostPos.row === exitPos.row + 1;
    }
    // Exit at the bottom
    else if (exitPos.row > bottommostPos.row) {
      return bottommostPos.col === exitPos.col && bottommostPos.row + 1 === exitPos.row;
    }
  }

  // Special case: If the primary piece is already at the exit position
  // (this could happen in some puzzle variations)
  return primaryPiece.positions.some((pos) => pos.row === exitPos.row && pos.col === exitPos.col);
};

```

## 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",
};
```

## 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

Home

Code terlalu panjang  
[https://github.com/yonatan-nyo/Tucil3\\_13523008\\_13523036/blob/main/src/src/pages/Home.tsx](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";

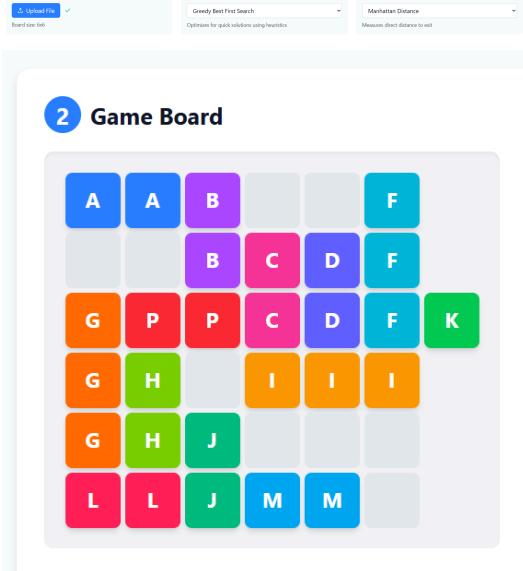
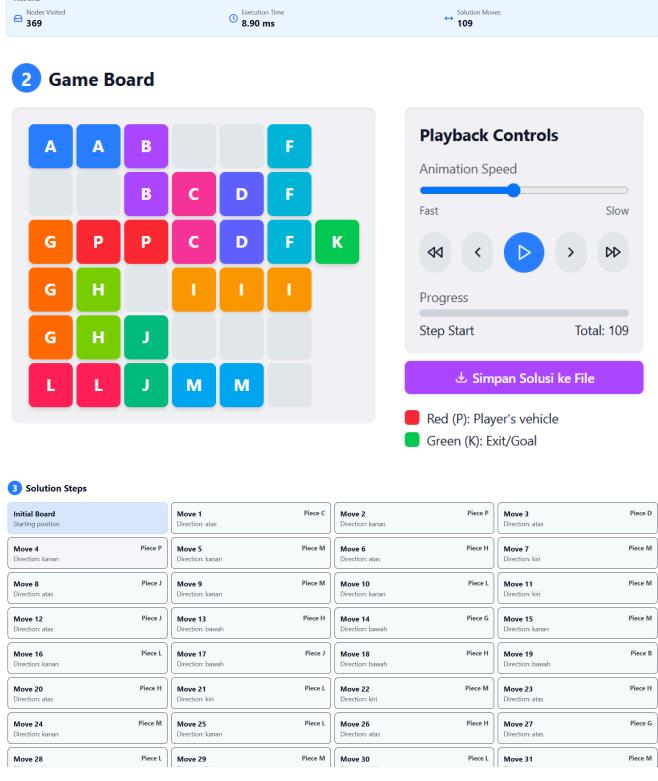
const Layout: React.FC<{ children: React.ReactNode }> = ({ children }) => {
  return (
    <div className="flex flex-col min-h-screen">
      <Navbar />
      <main className="flex-grow">{children}</main>
      <Footer />
    </div>
  );
};

export default Layout;
```

## BAB IV

### PENGUJIAN

#### 5.1 Hasil Pengujian

<p style="text-align: center;"><b>Test Case A:</b></p> <p style="text-align: center;">6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.</p>																																																													
<p><b>Input</b></p> <p>Test Case A menggunakan algoritma Greedy Best First Search dengan heuristic Manhattan Distance</p> 	<p><b>Output</b></p>  <table border="1" style="width: 100%; border-collapse: collapse; font-size: small;"> <thead> <tr> <th colspan="6">3 Solution Steps</th> </tr> </thead> <tbody> <tr> <td>Initial Board Starting position</td> <td>Move 1 Direction: atas</td> <td>Piece C</td> <td>Move 2 Direction: kanan</td> <td>Piece P</td> <td>Move 3 Direction: atas</td> </tr> <tr> <td>Move 4 Direction: kanan</td> <td>Piece P</td> <td>Piece M</td> <td>Move 5 Direction: kanan</td> <td>Piece H</td> <td>Move 6 Direction: atas</td> </tr> <tr> <td>Move 8 Direction: atas</td> <td>Piece J</td> <td>Piece M</td> <td>Move 9 Direction: kanan</td> <td>Piece L</td> <td>Move 10 Direction: kanan</td> </tr> <tr> <td>Move 12 Direction: atas</td> <td>Piece J</td> <td>Piece H</td> <td>Move 13 Direction: bawah</td> <td>Piece G</td> <td>Move 11 Direction: kanan</td> </tr> <tr> <td>Move 16 Direction: kanan</td> <td>Piece L</td> <td>Piece J</td> <td>Move 17 Direction: bawah</td> <td>Piece H</td> <td>Move 18 Direction: bawah</td> </tr> <tr> <td>Move 20 Direction: atas</td> <td>Piece H</td> <td>Piece L</td> <td>Move 21 Direction: kanan</td> <td>Piece M</td> <td>Move 19 Direction: bawah</td> </tr> <tr> <td>Move 24 Direction: kanan</td> <td>Piece M</td> <td>Piece L</td> <td>Move 25 Direction: kanan</td> <td>Piece L</td> <td>Move 26 Direction: atas</td> </tr> <tr> <td>Move 28</td> <td>Piece L</td> <td>Piece M</td> <td>Move 29</td> <td>Piece M</td> <td>Move 30</td> </tr> <tr> <td></td> <td></td> <td></td> <td></td> <td></td> <td>Move 31</td> </tr> </tbody> </table>	3 Solution Steps						Initial Board Starting position	Move 1 Direction: atas	Piece C	Move 2 Direction: kanan	Piece P	Move 3 Direction: atas	Move 4 Direction: kanan	Piece P	Piece M	Move 5 Direction: kanan	Piece H	Move 6 Direction: atas	Move 8 Direction: atas	Piece J	Piece M	Move 9 Direction: kanan	Piece L	Move 10 Direction: kanan	Move 12 Direction: atas	Piece J	Piece H	Move 13 Direction: bawah	Piece G	Move 11 Direction: kanan	Move 16 Direction: kanan	Piece L	Piece J	Move 17 Direction: bawah	Piece H	Move 18 Direction: bawah	Move 20 Direction: atas	Piece H	Piece L	Move 21 Direction: kanan	Piece M	Move 19 Direction: bawah	Move 24 Direction: kanan	Piece M	Piece L	Move 25 Direction: kanan	Piece L	Move 26 Direction: atas	Move 28	Piece L	Piece M	Move 29	Piece M	Move 30						Move 31
3 Solution Steps																																																													
Initial Board Starting position	Move 1 Direction: atas	Piece C	Move 2 Direction: kanan	Piece P	Move 3 Direction: atas																																																								
Move 4 Direction: kanan	Piece P	Piece M	Move 5 Direction: kanan	Piece H	Move 6 Direction: atas																																																								
Move 8 Direction: atas	Piece J	Piece M	Move 9 Direction: kanan	Piece L	Move 10 Direction: kanan																																																								
Move 12 Direction: atas	Piece J	Piece H	Move 13 Direction: bawah	Piece G	Move 11 Direction: kanan																																																								
Move 16 Direction: kanan	Piece L	Piece J	Move 17 Direction: bawah	Piece H	Move 18 Direction: bawah																																																								
Move 20 Direction: atas	Piece H	Piece L	Move 21 Direction: kanan	Piece M	Move 19 Direction: bawah																																																								
Move 24 Direction: kanan	Piece M	Piece L	Move 25 Direction: kanan	Piece L	Move 26 Direction: atas																																																								
Move 28	Piece L	Piece M	Move 29	Piece M	Move 30																																																								
					Move 31																																																								
<p>Test Case A menggunakan algoritma Greedy Best First Search dengan</p>																																																													

### heuristic Blocking Vehicles

**1 Configure Solver**

Board Configuration  
Upload File ✓  
Board size: 6x6

Algorithm  
Greedy Best First Search  
Optimizes for quick solutions using heuristics

Heuristic  
Blocking Vehicle  
Counts vehicles blocking the path

**2 Game Board**

### 2 Game Board



#### Playback Controls

Animation Speed  
Fast □ Slow □

Progress  
Step Start Total: 10

↓ Simpan Solusi ke File

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

### 3 Solution Steps

Initial Board	Move 1	Move 2	Move 3
Starting position	Direction: atas	Direction: atas	Direction: atas
Move 4	Piece I	Piece C	Piece G
Direction: kanan	Move 5	Move 6	Move 7
	Direction: bawah	Direction: bawah	Direction: bawah
Move 8	Piece P	Piece F	Piece F
Direction: kanan	Move 9	Move 10	Move 10
	Direction: kanan	Direction: kanan	Direction: kanan

### Results

Nodes Visited  
1,908

Execution Time  
17.80 ms

Solution Moves  
9

### Test Case A menggunakan algoritma UCS

**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**

### 2 Game Board



#### Playback Controls

Animation Speed  
Fast □ Slow □

Progress  
Step Start Total: 9

↓ Simpan Solusi ke File

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

### 3 Solution Steps

Initial Board	Move 1	Move 2	Move 3
Starting position	Direction: kiri	Direction: bawah	Direction: atas
Move 4	Piece I	Piece F	Piece C
Direction: kanan	Move 5	Move 6	Move 7
	Direction: bawah	Direction: atas	Direction: bawah
Move 8	Piece P	Move 9	Piece F
Direction: kanan	Move 10	Direction: kanan	Direction: kanan

### Test Case A menggunakan algoritma A\* dengan heuristic Manhattan Distance

Results  
Nodes Visited  
1,290

Execution Time  
21.60 ms

Solution Moves  
9

**1 Configure Solver**

Board Configuration: Board size: 6x6  
Algorithm: A\* Search  
Heuristic: Blocking Vehicles

**2 Game Board**

**Game Board**

**Playback Controls**

Animation Speed: Fast to Slow  
Progress: Step Start to Total: 9  
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	Move 9
Starting position	Direction kiri Piece F	Direction atas Piece I	Direction atas Piece D	Direction bawah Piece F	Direction bawah Piece F	Direction atas Piece C	Direction bawah Piece F	Direction bawah Piece E	Direction kanan Piece P

Test Case A menggunakan algoritma A\* dengan heuristic Blocking Vehicles

**1 Configure Solver**

Board Configuration: Board size: 6x6  
Algorithm: A\* Search  
Heuristic: Blocking Vehicles

**2 Game Board**

**Game Board**

**Playback Controls**

Animation Speed: Fast to Slow  
Progress: Step Start to Total: 9  
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	Move 9
Starting position	Direction kiri Piece F	Direction atas Piece I	Direction atas Piece D	Direction bawah Piece F	Direction bawah Piece F	Direction atas Piece C	Direction bawah Piece F	Direction bawah Piece E	Direction kanan Piece P

Test Case A menggunakan algoritma Dijkstra

**1 Configure Solver**

Board Configuration: Board size: 6x6  
Algorithm: Dijkstra's Algorithm  
Find the shortest path in weighted graphs

**2 Game Board**

**Game Board**

**Playback Controls**

Animation Speed: Fast to Slow  
Progress: Step Start to Total: 9  
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	Move 9
Starting position	Direction kiri Piece F	Direction atas Piece I	Direction atas Piece D	Direction bawah Piece F	Direction bawah Piece F	Direction atas Piece C	Direction bawah Piece F	Direction bawah Piece E	Direction kanan Piece P

**2 Game Board**

**2 Game Board**

**Playback Controls**

Animation Speed

Fast

Slow

<<
<
>
>>

Progress

Step Start
Total: 9

↓ Simpan Solusi ke File

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

3 Solution Steps

Initial Board Starting position	Move 1 Direction: atas Piece F	Move 2 Direction: kiri Piece D	Move 3 Direction: atas Piece C
Move 4 Direction: bawah	Move 5 Direction: kanan Piece P	Move 6 Direction: bawah Piece P	Move 7 Direction: bawah Piece F
Move 8 Direction: kanan	Move 9 Direction: kanan Piece P		

### Test Case D:

6 6

12

..AAAD

.BCC.D

KLBEPP.

LBEFGG

LMMFH.

IJJH.

Input	Output
Test Case D menggunakan algoritma Greedy Best First Search dengan heuristic Manhattan Distance <div style="margin-top: 10px;"> <b>Configure Solver</b> <div style="display: flex; justify-content: space-between;"> <div style="width: 33%;">           Board Configuration           <div style="display: flex; justify-content: space-between;"> <span>Import File</span> <span>Export File</span> </div> <div style="display: flex; justify-content: space-between;"> <span>Insert icon</span> <span>Remove icon</span> </div> </div> <div style="width: 33%;">           Algorithm           <div style="display: flex; justify-content: space-between;"> <span>Greedy Best First Search</span> <span>Optimizes quick solutions using heuristics</span> </div> </div> <div style="width: 33%;">           Heuristic           <div style="display: flex; justify-content: space-between;"> <span>Manhattan Distance</span> <span>Measures direct distance to exit</span> </div> </div> </div> <div style="display: flex; justify-content: space-between; margin-top: 5px;"> <span>Solve Puzzle</span> <span>Reset</span> </div> </div>	<div style="display: flex; justify-content: space-between; margin-bottom: 10px;"> <span>Results</span> <span>Nodes Visited: 40,742</span> <span>Execution Time: 521.10 ms</span> <span>Solution Moves: 517</span> </div>

**2 Game Board**

**2 Game Board**

**Playback Controls**

Animation SpeedFast
Slow

<<
<
>
>>

Progress

Step Start
Total: 517

↓ Simpan Solusi ke File

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

**3 Solution Steps**

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

**Results**

Nodes Visited: 18,822   Execution Time: 240.70 ms   Solution Moves: 520

**2 Game Board**

**2 Game Board**

**Playback Controls**

Animation SpeedSlow
Fast

<<
<
>
>>

Progress

Step Start
Total: 520

↓ Simpan Solusi ke File

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

**3 Solution Steps**

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

## Test Case D menggunakan algoritma UCS

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

## Test Case D menggunakan algoritma A\* dengan heuristic Manhattan Distance

**1 Configure Solver**

Board Configuration  
Upload File Board size: 6x7

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

Solve Puzzle > Reset

**2 Game Board**

**3 Solution Steps**

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

1 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: kanan	Piece C	Move 6 Direction: kiri	Piece A
Move 8 Direction: kiri	Piece A	Move 9 Direction: atas	Piece F	Move 10 Direction: kiri	Piece P
Move 12 Direction: kiri	Piece G	Move 13 Direction: bawah	Piece D	Move 14 Direction: bawah	Piece D
Move 16 Direction: kanan	Piece G	Move 17 Direction: kanan	Piece P	Move 18 Direction: bawah	Piece F
Move 20 Direction: kanan	Piece C	Move 21 Direction: kanan	Piece C	Move 22 Direction: kanan	Piece C
Move 24 Direction: kanan	Piece A	Move 25 Direction: kiri	Piece G	Move 26 Direction: atas	Piece B
Move 28 Direction: atas	Piece B	Move 29 Direction: atas	Piece E	Move 30 Direction: kiri	Piece G
					Move 31 Direction: kiri

Test Case D menggunakan algoritma A\* dengan heuristic Blocking Vehicles

1 Configure Solver

Board Configuration:  ✓  
Board size: 6x7

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

Solve Puzzle > Reset

## 2 Game Board



Results					
Nodes Visited: 68,025		Execution Time: 460.00 ms		Solution Moves: 85	

## 2 Game Board



### Playback Controls

Animation Speed:

Fast << < > >> Slow

Progress:

Step Start: Total: 85

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

## 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	Piece C	Move 6 Direction: kiri	Piece A
Move 8 Direction: kiri	Piece A	Move 9 Direction: atas	Piece F	Move 10 Direction: kiri	Piece P
Move 12 Direction: kiri	Piece G	Move 13 Direction: bawah	Piece D	Move 14 Direction: bawah	Piece D
Move 16 Direction: kanan	Piece G	Move 17 Direction: kanan	Piece P	Move 18 Direction: bawah	Piece F
Move 20 Direction: kanan	Piece C	Move 21 Direction: kanan	Piece C	Move 22 Direction: kanan	Piece C
Move 24 Direction: kanan	Piece A	Move 25 Direction: kiri	Piece G	Move 26 Direction: atas	Piece B
Move 28 Direction: atas	Piece B	Move 29 Direction: atas	Piece E	Move 30 Direction: kiri	Piece G
					Move 31 Direction: kiri

Test Case D menggunakan algoritma Dijkstra

1 Configure Solver

Board Configuration:  ✓  
Board size: 6x7

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

Solve Puzzle > Reset

Results					
Nodes Visited: 19,332		Execution Time: 319.40 ms		Solution Moves: 85	

**2 Game Board**

**2 Game Board**

**Playback Controls**

- Animation Speed: Fast (blue slider) to Slow (grey slider)
- Progress: Step Start (grey button) to Total: 85 (grey button)
- Simpan Solusi ke File (purple button)

**3 Solution Steps**

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

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
Test Case H	<div style="display: flex; align-items: center;"> <span style="margin-right: 10px;">Configure Solver</span> <span style="border: 1px solid #ccc; padding: 2px 5px; border-radius: 3px;">Upload File</span> <span style="margin-left: auto;"> <a href="#" style="color: green; font-weight: bold;">Solve Puzzle &gt;</a> </span> </div> <p style="font-size: small; color: red; margin-top: 5px;">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
Test Case J	<div style="display: flex; align-items: center;"> <span style="margin-right: 10px;">Configure Solver</span> <span style="border: 1px solid #ccc; padding: 2px 5px; border-radius: 3px;">Upload File</span> <span style="margin-left: auto;"> <a href="#" style="color: red; font-weight: bold;">Solve Puzzle &gt;</a> </span> </div> <p style="font-size: small; color: red; margin-top: 5px;">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 K PAB..F

	PABCDF GGGCDF .H.... .HJ... LLJMM.
Input	Output
Test Case K 	<b>Error</b> Error parsing file: Exit (K) must be aligned with the primary piece (P).

## 5.2 Analisis Hasil Pengujian

### 5.2.1 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 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. Dalam pengujian yang dilakukan pada test case D, dapat dilihat bahwa ketika menggunakan algoritma GBFS 517 moves dibandingkan algoritma lainnya 85 moves.

### 5.2.3 A\* Search

A\* Search memadukan UCS dan GBFS melalui fungsi evaluasi  $f(n) = g(n) + h(n)$ . Dengan heuristik admissible, seperti manhattan atau blocking Vehicles, 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 Analisis Hasil Pengujian

Dengan menggunakan Test Case A, kita dapat lihat bahwa ketika menggunakan algoritma Greedy Best First Search dengan heuristic Greedy Best First Search dengan heuristik Blocking Vehicles mendapatkan solusi dengan waktu tercepat dan node count paling sedikit, yaitu 2.40ms dan 41 dan dengan menggunakan A\* dan heuristik

Manhattan Distance mendapatkan solusi dengan waktu terlama, yaitu 21.60ms dan dengan menggunakan UCS mendapatkan solusi dengan node count paling banyak yaitu 1908. Walaupun Greedy Best First Search mendapatkan solusi dengan waktu paling cepat dan node count paling sedikit tetapi solusi yang ditemukan tidak optimal tidak seperti algoritma lain yang mendapatkan solusi untuk 9 pergerakan sedangkan dengan GBFS 10 gerakan.

## **BAB V**

### **PENJELASAN BONUS**

Pertama, selain Uniform Cost Search, Greedy Best-First Search, dan A\*, kami juga mengimplementasikan algoritma Dijkstra 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.

Kedua, kami mengimplementasikan beberapa fungsi heuristik yang dapat dipilih pengguna untuk algoritma informed search (A\* dan Greedy BFS). 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) 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\*, dan Dijkstra 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. n.d. “Greedy Best-First Search Algorithm.” GeeksforGeeks. (<https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/>, diakses 16 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](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	✓	