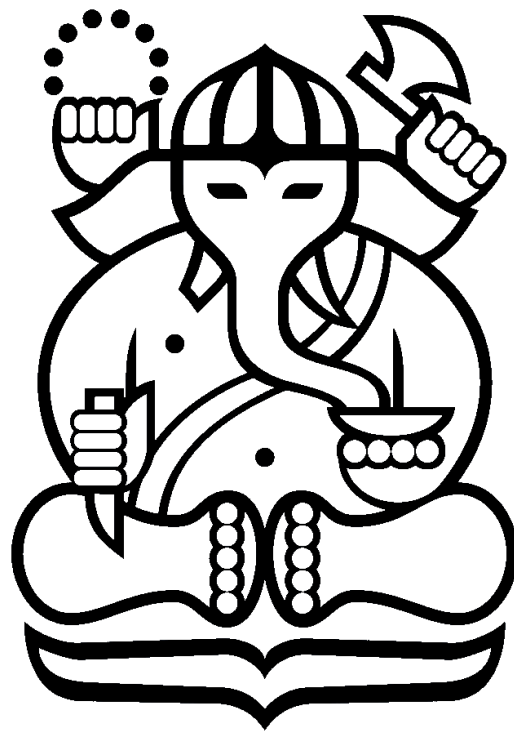


Laporan Tugas Kecil

Penyelesaian Rush Hour Dengan

Algoritma Pencarian Rute

**Tugas Kecil 3 - IF2211 Strategi Algoritma Semester II Tahun
2024/2025**



Disusun oleh

Naufarrel Zhafif Abhista - 13523149

I Made Wiweka Putera - 13523160

Daftar Isi

Daftar Isi.....	1
1. Pendahuluan.....	2
1.1. Rush Hour.....	2
1.2. Algoritma Pencarian Rute.....	4
2. Heuristik.....	5
2.1. Distance To Exit.....	5
2.2. Blocking Pieces.....	5
2.3. Manhattan Distance.....	6
2.4. Pieces Density.....	6
2.5. Combined.....	6
3. Algoritma Pencarian Rute.....	7
3.1. Konsep Dasar Algoritma Pencarian.....	7
3.2. Fungsi Evaluasi: $f(n) = g(n) + h(n)$	7
3.3. Uniform Cost Search (UCS).....	8
3.4. Greedy Best-First Search.....	9
3.5. A* Search.....	10
3.6. (Bonus) Branch And Bound.....	11
4. Source Code.....	13
4.1. Struktur Proyek.....	13
4.2. Dependency.....	15
4.3. RushHourSolverApp.java.....	15
5. Uji Coba.....	16
5.1. Input.....	16
5.2. Uniform Cost Search.....	17
5.3. Greedy Best-First Search.....	19
5.4. A* Search.....	21
5.5. Branch and Bound.....	23
6. Analisis.....	27
6.1. Analisis Heuristik.....	27
6.2. Analisis Pencarian.....	27
7. Lampiran.....	28

1. Pendahuluan

1.1. *Rush Hour*



Gambar 1. Rush Hour Puzzle

(Sumber:

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

Rush Hour adalah permainan teka-teki yang diciptakan oleh Nob Yoshigahara pada tahun 1970-an. Permainan ini dimainkan pada papan berukuran 6x6 dengan beberapa kendaraan (blok) yang harus digeser untuk membuka jalan bagi kendaraan utama (biasanya berwarna merah) agar dapat keluar dari papan. Setiap kendaraan hanya dapat bergerak maju atau mundur sesuai orientasinya (horizontal atau vertikal) tanpa dapat berputar. Tantangan dalam permainan ini terletak pada bagaimana menemukan urutan pergeseran kendaraan yang tepat dengan jumlah langkah minimal untuk mengeluarkan kendaraan utama.

Komponen penting dari permainan Rush Hour terdiri dari:

1. Papan Permainan

Papan menjadi arena tempat permainan berlangsung. Terdiri dari kisi-kisi berbentuk sel yang menjadi titik koordinat penempatan kendaraan. Setiap sel dapat ditempati oleh bagian dari kendaraan. Saat permainan dimulai, semua kendaraan sudah berada pada posisi awal tertentu dengan konfigurasi spesifik yang mencakup letak dan arah (horizontal atau vertikal).

2. Kendaraan (Piece)

Kendaraan merupakan objek yang dapat dipindahkan pada papan permainan. Setiap kendaraan memiliki karakteristik berupa:

- Posisi pada papan
- Ukuran (jumlah sel yang ditempati)
- Orientasi yang hanya bisa horizontal atau vertikal (tidak mungkin diagonal)
- Ukuran kendaraan bervariasi, umumnya terdiri dari kendaraan yang menempati 2 sel atau 3 sel. Penting untuk diketahui bahwa kendaraan tidak dapat melintasi atau menembus kendaraan lain saat bergerak.

3. Kendaraan Utama (Primary Piece)

Kendaraan utama adalah objek khusus yang menjadi fokus untuk dikeluarkan dari papan (biasanya direpresentasikan dengan warna merah). Dalam satu permainan, hanya boleh ada satu kendaraan utama yang harus dibebaskan.

4. Pintu Keluar

Pintu keluar merupakan titik pada tepi papan yang menjadi sasaran bagi kendaraan utama untuk keluar. Ketika kendaraan utama mencapai pintu keluar, permainan dinyatakan selesai.

5. Pergerakan

Pergerakan dalam permainan terbatas pada pergeseran lurus sesuai orientasi kendaraan:

- Kendaraan vertikal hanya dapat bergerak ke atas atau ke bawah
- Kendaraan horizontal hanya dapat bergerak ke kiri atau ke kanan

- Setiap kendaraan memiliki keterbatasan bahwa tidak dapat melewati atau menembus kendaraan lain, sehingga jalur pergerakannya harus bebas hambatan.

1.2. Algoritma Pencarian Rute

Algoritma pencarian rute adalah algoritma/teknik pemecahan masalah yang digunakan untuk menemukan jalur optimal dari suatu keadaan awal menuju keadaan tujuan dalam ruang masalah. Beberapa algoritma pencarian rute yang diimplementasikan dalam proyek ini:

- Uniform Cost Search (UCS): Algoritma yang mengeksplorasi keadaan berdasarkan biaya terkecil (jumlah langkah) dari keadaan awal.
- A Search*: Algoritma yang menggunakan fungsi heuristik untuk mengestimasi jarak ke tujuan, sehingga pencarian lebih terarah dibandingkan UCS.
- Greedy Best-First Search: Algoritma yang memilih keadaan berikutnya hanya berdasarkan nilai heuristik, tanpa mempertimbangkan biaya yang telah ditempuh.
- Branch and Bound: Algoritma yang memelihara batas atas pada biaya solusi dan memangkas cabang-cabang yang tidak potensial.

Algoritma-algoritma tersebut memiliki kekuatan dan kelemahan masing-masing dari segi waktu eksekusi, memori yang digunakan, dan optimalitas solusi yang dihasilkan.

2. Heuristik

Heuristik adalah fungsi estimasi yang digunakan dalam algoritma pencarian terinformasi untuk memperkirakan jarak atau biaya dari suatu keadaan ke keadaan tujuan. Dalam permainan Rush Hour, heuristik berperan penting untuk mengarahkan pencarian dengan memprioritaskan keadaan yang lebih menjanjikan, sehingga algoritma dapat menemukan solusi lebih cepat tanpa harus mengeksplorasi seluruh ruang pencarian.

Beberapa heuristik dapat dikatakan *admissible*. Sebuah heuristik dikatakan *admissible* jika untuk setiap simpul n , $h(n) \leq h^*(n)$, dimana $h^*(n)$ adalah biaya sebenarnya untuk mencapai tujuan. Dengan kata lain, sebuah heuristik bersifat *admissible* bila heuristik tersebut tidak pernah *overestimate* jarak sebenarnya ke tujuan. Heuristik yang *admissible* selalu memberikan estimasi yang optimistik (sama dengan atau lebih kecil dari) biaya sebenarnya untuk mencapai tujuan.

Dalam permainan Rush Hour:

- Heuristik *admissible* akan memberikan perkiraan jumlah langkah minimum yang dibutuhkan untuk mencapai solusi
- Estimasi ini tidak boleh melebihi jumlah langkah yang sebenarnya dibutuhkan dalam kondisi optimal
- Heuristik yang baik akan mendekati biaya sebenarnya tanpa melebihinya

Sebaliknya, heuristik yang tidak *admissible* dapat menghasilkan solusi yang tidak optimal ketika digunakan, karena algoritma mungkin "terburu-buru" menuju solusi yang tampak menjanjikan berdasarkan estimasi yang berlebihan.

Berikut penjelasan berbagai heuristik yang diimplementasikan:

2.1. *Distance To Exit*

Heuristik ini mengukur seberapa jauh kendaraan utama dari pintu keluar jika tidak ada kendaraan lain yang menghalangi. Perhitungannya sederhana:

- Untuk kendaraan utama horizontal: menghitung jumlah sel dari posisi kendaraan utama hingga tepi papan tempat pintu keluar berada
- Untuk kendaraan utama vertikal: menghitung jumlah sel dari posisi kendaraan utama hingga tepi papan tempat pintu keluar berada

Semakin kecil nilai heuristik ini, semakin dekat kendaraan utama dengan pintu keluar. Heuristik ini efektif untuk permainan sederhana tetapi tidak memperhitungkan kendaraan penghalang, yang artinya merupakan batas bawah

dari jumlah langkah yang diperlukan (optimistis). Maka, dapat kita katakan bahwa heuristik ini bersifat *admissible*.

2.2. *Blocking Pieces*

Heuristik ini menghitung jumlah kendaraan yang secara langsung menghalangi jalur kendaraan utama ke pintu keluar. Setiap kendaraan yang berada di antara kendaraan utama dan pintu keluar dihitung sebagai satu penghalang. Logikanya adalah:

- Semakin banyak kendaraan penghalang, semakin banyak langkah yang diperlukan untuk memindahkan mereka
- Ketika tidak ada kendaraan penghalang, kendaraan utama dapat langsung mencapai pintu keluar

Heuristik ini hanya menghitung jumlah kendaraan penghalang, tanpa mempertimbangkan:

- Berapa langkah yang diperlukan untuk memindahkan setiap kendaraan
- Fakta bahwa menggeser satu kendaraan mungkin memerlukan penggeseran kendaraan lainnya terlebih dahulu

Hal ini dapat menyebabkan *underestimate* yang signifikan, tetapi dalam beberapa kasus juga dapat *overestimate* jika beberapa kendaraan bisa dipindahkan dengan satu gerakan yang sama. Sehingga, heuristik ini bersifat *Non-Admissible*

2.3. *Manhattan Distance*

Heuristik ini mengukur total jarak perpindahan yang diperlukan oleh semua kendaraan penghalang untuk membuka jalan bagi kendaraan utama. Untuk setiap kendaraan penghalang, dihitung jarak minimum (dalam jumlah sel) yang diperlukan untuk memindahkannya agar tidak menghalangi jalur kendaraan utama. Contohnya:

- Untuk kendaraan horizontal yang menghalangi, dihitung berapa langkah minimum untuk memindahkannya ke atas atau ke bawah
- Untuk kendaraan vertikal yang menghalangi, dihitung berapa langkah minimum untuk memindahkannya ke kiri atau ke kanan

Heuristik ini menghitung jumlah minimal langkah yang diperlukan untuk memindahkan setiap kendaraan penghalang dan ditambah 1 untuk gerakan *primary piece*. Namun, implementasinya mengasumsikan setiap kendaraan dapat langsung dipindahkan tanpa hambatan, padahal:

- Mungkin ada kendaraan lain yang menghalangi pergerakan kendaraan penghalang
- Beberapa kendaraan mungkin harus bergerak lebih jauh dari yang diperkirakan karena konfigurasi papan

Maka, dapat kita katakan bahwa heuristik ini bersifat *Non-Admissible*.

2.4. *Pieces Density*

Heuristik ini mengukur kepadatan kendaraan di area antara kendaraan utama dan pintu keluar. Semakin padat area tersebut, semakin sulit untuk mencapai solusi. Perhitungannya:

- Tentukan "koridor" dari kendaraan utama ke pintu keluar
- Hitung persentase sel yang ditempati kendaraan di koridor tersebut

Heuristik ini tidak secara langsung memetakan ke jumlah langkah yang diperlukan. Persentase kepadatan tidak menjamin korelasi langsung dengan biaya sebenarnya. Area yang padat tidak selalu berarti memerlukan lebih banyak langkah, dan sebaliknya. Karena bisa saja terjadi *overestimate*, heuristik ini *Non-Admissible*.

2.5. *Combined*

Heuristik gabungan mengkombinasikan beberapa heuristik di atas dengan memberikan bobot tertentu pada masing-masing. Formulasnya:

$$h_{combined} = (w_1 \times h_{distance}) + (w_2 \times h_{blocking}) + (w_3 \times h_{manhattan}) + (w_4 \times h_{density})$$

Dimana w_1 , w_2 , w_3 , dan w_4 adalah bobot yang ditetapkan untuk masing-masing heuristik.

Keunggulan heuristik gabungan adalah dapat mengkompensasi kelemahan masing-masing heuristik individu. Misalnya, Distance To Exit mungkin terlalu optimistik, sedangkan Blocking Pieces mungkin tidak memperhitungkan jarak perpindahan yang diperlukan. Dengan menggabungkan keduanya, estimasi yang dihasilkan bisa lebih realistis.

Namun, karena beberapa heuristik yang digabungkan bersifat *non-admissible*, kombinasinya (heuristik ini) juga *non-admissible*.

Pemilihan heuristik yang tepat sangat memengaruhi kinerja algoritma pencarian seperti A* dan Greedy Best-First Search. Heuristik yang akurat akan mengarahkan pencarian ke solusi dengan lebih efisien, sedangkan heuristik yang kurang akurat dapat menyebabkan algoritma mengeksplorasi jalur yang tidak perlu.

3. Algoritma Pencarian Rute

Algoritma pencarian rute adalah teknik yang digunakan untuk menemukan jalur optimal dari keadaan awal menuju keadaan tujuan dalam suatu ruang masalah. Pada dasarnya, algoritma pencarian rute dapat dibagi menjadi dua kategori utama: pencarian tidak terinformasi (*uninformed search*) dan pencarian terinformasi (*informed search*).

3.1. Konsep Dasar Algoritma Pencarian

Algoritma pencarian rute bekerja dengan melakukan eksplorasi sistematis terhadap ruang keadaan (*state space*). Ruang keadaan ini dapat direpresentasikan sebagai graf, di mana:

- Simpul (*node*) mewakili keadaan dalam masalah
- Sisi (*edge*) mewakili transisi atau aksi yang mengubah satu keadaan menjadi keadaan lain
- Biaya (*cost*) sisi mewakili usaha atau sumber daya yang dibutuhkan untuk transisi

Tujuan algoritma adalah menemukan jalur dari simpul awal ke simpul tujuan dengan kriteria tertentu, biasanya memiliki biaya total minimum.

3.2. Fungsi Evaluasi: $f(n) = g(n) + h(n)$

Fungsi evaluasi $f(n)$ adalah elemen kunci dalam algoritma pencarian terinformasi seperti A* Search. Fungsi ini membantu algoritma menentukan node mana yang akan dieksplorasi selanjutnya:

- $g(n)$: Merepresentasikan biaya sebenarnya dari simpul awal ke simpul n . Ini adalah biaya yang telah diketahui dan akumulasi dari semua langkah yang telah diambil untuk mencapai simpul n dari simpul awal.
- $h(n)$: Merepresentasikan estimasi biaya (heuristik) dari simpul n ke simpul tujuan. Estimasi ini memberikan "panduan" kepada algoritma mengenai seberapa jauh lagi untuk mencapai tujuan.
- $f(n)$: Merupakan jumlah dari $g(n)$ dan $h(n)$, yang mewakili estimasi total biaya jalur dari simpul awal ke simpul tujuan yang melalui simpul n .

3.3. Uniform Cost Search (UCS)

Uniform Cost Search adalah algoritma pencarian tak terinformasi (*uninformed search*) yang bersifat sistematis dan menjamin solusi optimal. Algoritma ini bekerja dengan mengeksplorasi node-node dalam graf berdasarkan biaya kumulatif terkecil dari node awal.

Cara Kerja UCS:

- Algoritma memulai pencarian dari node awal dan secara bertahap menjelajahi node-node tetangga
- Node diurutkan dalam sebuah priority queue berdasarkan biaya kumulatif (jumlah langkah) yang dibutuhkan untuk mencapainya dari node awal
- Selalu memilih node dengan biaya kumulatif terkecil untuk dieksplorasi selanjutnya
- Pencarian berlanjut hingga mencapai node tujuan atau semua kemungkinan node telah dieksplorasi

Atau dalam pseudocode,

```
function UCS(initialState)
    // Initialize
    frontier ← PriorityQueue() ordered by path cost
    frontier.add(node(initialState, cost=0))
    explored ← empty set

    while frontier is not empty do
        node ← frontier.removeMin()

        if isGoalState(node.state) then
            return buildSolution(node)

        if node.state not in explored then
            add node.state to explored

            for each action in getPossibleActions(node.state) do
                childState ← result(node.state, action)
                childCost ← node.cost + 1 // Cost of one step

                if childState not in explored then
                    parentNode ← node(childState, cost=childCost,
parent=node)
                    frontier.add(childNode)

    return failure // No solution found
```

UCS tidak menggunakan informasi tambahan (heuristik, $h(n)$) tentang jarak ke tujuan, namun hanya menggunakan informasi biaya jalur saat ini ($g(n)$). Meskipun menjamin solusi optimal, UCS cenderung tidak efisien pada ruang pencarian besar karena mungkin mengeksplorasi banyak jalur yang tidak mengarah ke tujuan.

Dalam Rush Hour, UCS akan mempertimbangkan semua kemungkinan pergerakan kendaraan pada setiap langkah dan memilih jalur dengan jumlah langkah paling sedikit. Jika diperhatikan, algoritma Uniform Cost Search (UCS) berperilaku secara identik dengan Breadth-First Search (BFS). Hal ini disebabkan oleh karakteristik unik dari permainan Rush Hour. Setiap pergerakan kendaraan memiliki biaya yang sama, yaitu satu langkah. Ketika semua transisi antar keadaan memiliki bobot yang seragam, UCS secara efektif akan menjelajahi ruang keadaan *level by level*, seperti yang dilakukan oleh BFS. Keduanya akan selalu mengeksplorasi keadaan dengan jumlah langkah terkecil terlebih dahulu sebelum mempertimbangkan keadaan yang membutuhkan langkah lebih banyak.

Perbedaan implementasi hanya terletak pada struktur data yang digunakan: BFS menggunakan antrian sederhana (FIFO), sedangkan UCS menggunakan priority queue berdasarkan biaya path. Namun, karena semua biaya langkah bernilai 1, priority queue akan mengeluarkan node dalam urutan yang sama dengan antrian biasa. Kedua algoritma dijamin menemukan solusi dengan jumlah langkah minimum, menjadikan UCS setara dengan BFS dalam menyelesaikan puzzle Rush Hour.

3.4. Greedy Best-First Search

Greedy Best-First Search adalah algoritma pencarian terinformasi (*informed search*) yang menggunakan fungsi heuristik untuk mengestimasi jarak ke tujuan. Algoritma ini selalu memilih node yang tampak paling menjanjikan berdasarkan estimasi heuristik, tanpa mempertimbangkan biaya yang sudah ditempuh.

Cara Kerja Greedy Best-First Search:

- Algoritma menyimpan node-node yang akan dieksplorasi dalam priority queue
- Node diurutkan berdasarkan HANYA nilai heuristik (estimasi jarak ke tujuan)
- Pada setiap langkah, algoritma memilih node dengan nilai heuristik terendah
- Node-node baru (successor) dihasilkan dan ditambahkan ke queue

- Proses berlanjut hingga menemukan node tujuan atau kehabisan node untuk dieksplorasi

Atau dalam *pseudocode*,

```
function GreedyBestFirstSearch(initialState, heuristic)
    // Initialize
    frontier ← PriorityQueue() ordered by heuristic value
    initialNode ← node(initialState, cost=0,
hValue=heuristic(initialState))
    frontier.add(initialNode)
    explored ← empty set

    while frontier is not empty do
        node ← frontier.removeMin()

        if isGoalState(node.state) then
            return buildSolution(node)

        if node.state not in explored then
            add node.state to explored

            for each action in getPossibleActions(node.state) do
                childState ← result(node.state, action)

                if childState not in explored then
                    childHValue ← heuristic(childState)
                    childNode ← node(childState, cost=node.cost+1,
hValue=childHValue, parent=node)
                    frontier.add(childNode) // Ordered only by hValue

    return failure // No solution found
```

Greedy Best-First Search dapat menemukan solusi dengan cepat, namun tidak menjamin solusi optimal karena mengabaikan biaya yang sudah ditempuh. Algoritma ini sangat bergantung pada kualitas fungsi heuristik yang digunakan.

Dalam Rush Hour, algoritma Greedy Best-First Search memprioritaskan eksplorasi keadaan berdasarkan nilai heuristik semata, tanpa mempertimbangkan jumlah langkah yang telah ditempuh. Algoritma ini selalu memilih keadaan yang tampak paling menjanjikan menurut fungsi heuristiknya—misalnya, keadaan dengan sedikit kendaraan penghalang atau kendaraan utama yang lebih dekat ke pintu keluar. Karena tidak memperhitungkan biaya perjalanan sejauh ini, Greedy Best-First Search tidak menjamin solusi optimal dalam hal jumlah langkah minimum. Meskipun algoritma ini dapat menemukan solusi dengan cepat, solusi yang dihasilkan hanya bersifat optimal lokal, artinya mungkin bukan lintasan terpendek menuju keadaan tujuan. Efisiensi Greedy Best-First Search sangat

bergantung pada kualitas heuristik yang digunakan; heuristik yang tepat dapat mempercepat penemuan solusi, namun tetap tidak mengubah sifat dasarnya yang mengabaikan jumlah langkah yang diperlukan untuk mencapai keadaan tersebut.

3.5. A* Search

A* Search adalah algoritma pencarian terinformasi yang menggabungkan kelebihan UCS dan Greedy Best-First Search. Algoritma ini memperhitungkan baik biaya yang sudah ditempuh maupun estimasi biaya menuju tujuan.

Cara Kerja A* Search:

- Node-node disimpan dalam priority queue berdasarkan nilai $f(n)$ terendah
- Algoritma secara iteratif memilih node dengan nilai $f(n)$ terendah untuk dieksplorasi
- Untuk setiap node yang dieksplorasi, dihasilkan node-node successor dan ditambahkan ke queue
- Proses berlanjut hingga menemukan node tujuan atau kehabisan node untuk dieksplorasi

Atau dalam *pseudocode*,

```
function AStarSearch(initialState, heuristic)
    // Initialize
    frontier ← PriorityQueue() ordered by  $f = g + h$ 
    initialHValue ← heuristic(initialState)
    initialNode ← node(initialState, cost=0, hValue=initialHValue)
    frontier.add(initialNode)
    explored ← empty set

    while frontier is not empty do
        node ← frontier.removeMin() // Node with lowest  $f$  value

        if isGoalState(node.state) then
            return buildSolution(node)

        if node.state not in explored then
            add node.state to explored

        for each action in getPossibleActions(node.state) do
            childState ← result(node.state, action)
            childGValue ← node.cost + 1 // Cost of one step

            if childState not in explored then
                childHValue ← heuristic(childState)
                childNode ← node(
                    childState,
                    cost=childGValue,
```

```
        hValue=childHValue,  
        fValue=childGValue + childHValue,  
        parent=node  
    )  
    frontier.add(childNode) // Ordered by fValue  
  
    return failure // No solution found
```

A* Search menjamin solusi optimal jika fungsi heuristik yang digunakan bersifat *admissible* (tidak pernah meng-*overestimate* biaya sebenarnya ke tujuan) dan *consistent* (memenuhi ketidaksamaan segitiga).

Secara teoritis, algoritma A* memang lebih efisien dibandingkan UCS dalam menyelesaikan puzzle Rush Hour. Keunggulan utama A* terletak pada pemanfaatan fungsi heuristik yang dapat "mengarahkan" pencarian ke arah yang lebih menjanjikan, sementara UCS menjelajahi keadaan secara merata berdasarkan jumlah langkah.

Dalam Rush Hour, A* dengan heuristik yang baik (seperti Distance to Exit) memungkinkan A* untuk menemukan solusi dengan mengeksplorasi jumlah keadaan yang lebih sedikit dibandingkan UCS, yang harus menjelajahi seluruh ruang keadaan pada kedalaman yang sama sebelum bergerak lebih dalam.

Sebagai contoh, ketika menghadapi papan Rush Hour yang kompleks, UCS akan mengeksplorasi semua kemungkinan konfigurasi papan setelah 1 langkah, lalu 2 langkah, dan seterusnya. Pendekatan ini tidak efisien jika solusi memerlukan banyak langkah. Sebaliknya, A* dapat "melompati" beberapa konfigurasi yang tampak kurang menjanjikan meskipun memiliki jumlah langkah yang lebih sedikit.

Namun, keuntungan efisiensi ini sangat bergantung pada kualitas heuristik yang digunakan. Dengan heuristik yang lemah atau tidak informatif, A* dapat kehilangan keunggulannya dan berperilaku serupa dengan UCS. Selain itu, komputasi nilai heuristik itu sendiri membutuhkan overhead tambahan yang perlu diperhitungkan dalam evaluasi efisiensi keseluruhan algoritma.

3.6. (Bonus) Branch And Bound

Branch and Bound adalah algoritma pencarian yang memelihara batas atas (*upper bound*) pada nilai solusi optimal dan menggunakan batas ini untuk memangkas cabang-cabang yang tidak menjanjikan dalam ruang pencarian.

Cara Kerja Branch and Bound:

- Algoritma menjelajahi ruang pencarian dengan pendekatan seperti UCS, namun dengan fitur pemangkasan
- Saat menemukan solusi lengkap, nilai solusi tersebut digunakan sebagai batas atas untuk solusi optimal
- Jika suatu node memiliki batas bawah (lower bound) yang melebihi batas atas saat ini, node tersebut dipangkas
- Batas bawah diperoleh dari biaya yang sudah ditempuh ditambah estimasi biaya menuju tujuan (mirip dengan A^*)
- Proses berlanjut hingga seluruh ruang pencarian telah dieksplorasi atau dipangkas

```
function BranchAndBound(initialState, heuristic)
    // Initialize
    frontier ← PriorityQueue() ordered by lower bound (cost + heuristic)
    initialHValue ← heuristic(initialState)
    initialNode ← node(initialState, cost=0, hValue=initialHValue)
    frontier.add(initialNode)
    explored ← empty set
    upperBound ← infinity
    bestSolution ← null

    while frontier is not empty do
        node ← frontier.removeMin()
        lowerBound ← node.cost + node.hValue

        // Pruning: Skip if this path can't be better than what we have
        if lowerBound ≥ upperBound then
            continue

        if isGoalState(node.state) then
            // Update the upper bound if we found a better solution
            if node.cost < upperBound then
                upperBound ← node.cost
                bestSolution ← node
            continue // Continue searching for potentially better
solutions

        if node.state not in explored then
            add node.state to explored

        for each action in getPossibleActions(node.state) do
            childState ← result(node.state, action)
            childCost ← node.cost + 1

            if childState not in explored then
                childHValue ← heuristic(childState)
                childLowerBound ← childCost + childHValue
```

```

solution          // Only consider paths that could improve the
                  if childLowerBound < upperBound then
                    childNode ← node(
                      childState,
                      cost=childCost,
                      hValue=childHValue,
                      parent=node
                    )
                    frontier.add(childNode)

return bestSolution // Return the best solution found

```

Branch and Bound menjamin solusi optimal dan dapat lebih efisien daripada UCS karena kemampuannya memangkas cabang yang tidak menjanjikan.

Dalam Rush Hour, Branch and Bound menawarkan keunggulan potensial dibanding algoritma lain melalui mekanisme pemangkasannya yang agresif. Algoritma ini menggabungkan pendekatan pencarian sistematis dengan kemampuan untuk "melompati" seluruh bagian dari ruang pencarian yang terbukti tidak akan menghasilkan solusi lebih baik dari yang sudah ditemukan.

Sebagai contoh, jika Branch and Bound telah menemukan solusi dengan 15 langkah, algoritma ini akan secara otomatis menghindari eksplorasi jalur-jalur yang sudah dipastikan membutuhkan 16 langkah atau lebih berdasarkan nilai batas bawahnya. Strategi ini sangat berguna pada puzzle kompleks dengan banyak jalan buntu atau jalur suboptimal.

Dibandingkan dengan A*, Branch and Bound kadang dapat menghasilkan performa lebih baik ketika solusi ditemukan relatif cepat, karena setelah menemukan solusi awal, kemampuan pemangkasannya semakin efektif. Namun, seperti halnya A*, efektivitas Branch and Bound sangat bergantung pada kualitas fungsi heuristik yang digunakan untuk menghitung batas bawah. Heuristik yang lemah dapat menyebabkan sedikit pemangkasan, sehingga algoritma akan berperilaku seperti UCS dengan overhead tambahan.

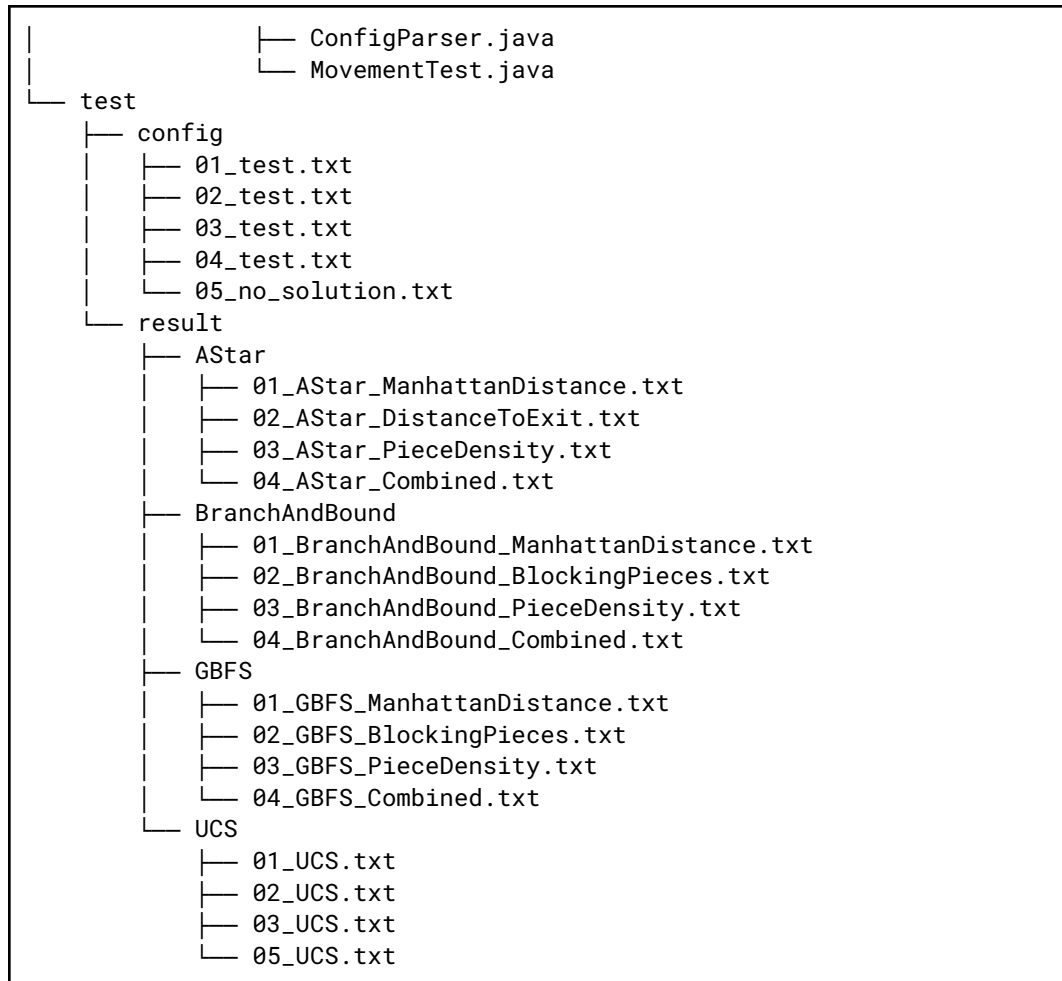
4. Source Code

Pada bagian ini, akan diperjelas secara singkat mengenai struktur proyek dan juga modularitas dari setiap kelas. *Penjelasan kode dari suatu file hanya akan dilakukan per bagian, bukan keseluruhan file itu sendiri.* Untuk melihat isi suatu file secara penuh, dapat dilihat secara langsung repositorinya, yang berada di Bab 6 (Lampiran).

4.1. Struktur Proyek

Proyek ini memiliki struktur direktori dan file sebagai berikut:

```
Tucil3_13523148_13523160
├── README.md
├── bin
│   └── rush-hour-solver-1.0-SNAPSHOT.jar
├── doc
│   └── Tucil3-Stima-2025.pdf
├── pom.xml
├── src
│   ├── main
│   │   ├── java
│   │   │   ├── RushHourSolverApp.java
│   │   │   ├── gui
│   │   │   │   ├── BoardDrawingUtil.java
│   │   │   │   ├── PieceColorManager.java
│   │   │   │   └── PlayPuzzle.java
│   │   │   ├── logic
│   │   │   │   ├── Board.java
│   │   │   │   ├── GameLogic.java
│   │   │   │   ├── GameManager.java
│   │   │   │   ├── GameState.java
│   │   │   │   ├── Node.java
│   │   │   │   ├── Piece.java
│   │   │   │   └── PrimaryPiece.java
│   │   │   └── solver
│   │   │       ├── algorithm
│   │   │       │   ├── AStarSolver.java
│   │   │       │   ├── BestFSolver.java
│   │   │       │   ├── BranchAndBoundSolver.java
│   │   │       │   ├── InformedSolver.java
│   │   │       │   ├── Solver.java
│   │   │       │   └── UCSolver.java
│   │   │       └── heuristic
│   │   │           ├── BlockingPiecesHeuristic.java
│   │   │           ├── CombinedHeuristic.java
│   │   │           ├── DistanceToExitHeuristic.java
│   │   │           ├── Heuristic.java
│   │   │           ├── ManhattanDistanceHeuristic.java
│   │   │           └── PieceDensityHeuristic.java
│   └── util
```



Penjelasan masing-masing direktori dan file:

- **bin/**: Berisi file executable untuk menjalankan program. Program dapat dijalankan pada sistem yang memiliki JVM.
- **doc/**: Direktori untuk dokumentasi proyek dalam bentuk **.pdf**.
- **src/main/java/**: Berisi kode sumber utama proyek.
 - **RushHourSolverApp.java**: File utama yang berisi kelas aplikasi utama dan berfungsi sebagai entry point program.
 - **gui/**: Paket yang berisi komponen-komponen antarmuka grafis untuk visualisasi dan interaksi.
 - **logic/**: Paket yang menangani logika permainan dan representasi keadaan (GameState).
 - **solver/algorithm/**: Paket yang berisi implementasi algoritma pencarian solusi.
 - **solver/heuristic/**: Paket yang berisi implementasi berbagai fungsi heuristik untuk algoritma pencarian terinformasi.
 - **util/**: Paket utilitas yang berisi kelas-kelas pembantu.

- **test/**: Direktori untuk file uji coba dalam bentuk gambar dan hasil dalam bentuk gambar dengan format file yang sama dalam folder output.
- **pom.xml**: Bertanggung jawab untuk melakukan kompilasi program dengan Maven sehingga menghasilkan executable yang dapat dijalankan.
- **README.md**: File readme yang mencakup deskripsi, persyaratan, instalasi, dan cara penggunaan aplikasi.

4.2. *Dependency*

Kompilasi program ini menggunakan Maven. Jika menggunakan Linux, dapat menggunakan,

```
sudo apt install Maven
```

Pastikan mempunyai Java Development Kit (JDK) 21 agar program dapat dijalankan.

4.3. **RushHourSolverApp.java**

RushHourSolverApp.java merupakan awal mula dari program saat dijalankan pengguna.

```
/**
 * Konstruktor utama aplikasi Rush Hour Solver.
 * Menginisialisasi komponen UI dan mengatur jendela utama
 * aplikasi.
 */
public RushHourSolverApp() extends JFrame {
    super("Rush Hour Solver");

    // Inisialisasi komponen UI
    initializeUI();
    initializeTimer();

    // Atur jendela utama
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setSize(900, 700);
    setLocationRelativeTo(null);
    setVisible(true);
}

/**
 * Menginisialisasi antarmuka pengguna utama dengan menyusun
```

```

panel-panel.
*/
private void initializeUI() {
    // Implementasi penyusunan panel utama, statistik, dan
    visualisasi
}

/**
 * Menginisialisasi timer untuk animasi langkah-langkah solusi.
 */
private void initializeTimer() {
    // Implementasi pengaturan timer animasi
}

/**
 * Membuat panel untuk kontrol solver.
 *
 * @return JPanel berisi kontrol-kontrol untuk mengonfigurasi
    dan menjalankan solver
 */
private JPanel createSolverPanel() {
    // Implementasi pembuatan panel dengan kontrol solver
}

/**
 * Membuat panel untuk menampilkan visualisasi solusi.
 *
 * @return JPanel berisi visualisasi papan dan kontrol-kontrol
    navigasi
 */
private JPanel createVisualizerPanel() {
    // Implementasi pembuatan panel visualisasi
}

/**
 * Membuat panel untuk menampilkan statistik pencarian.
 *
 * @return JPanel berisi label-label statistik solver
 */
private JPanel createStatsPanel() {
    // Implementasi pembuatan panel statistik
}

/**
 * Membuka dialog pemilihan file konfigurasi dan memuat file
    yang dipilih.
 */
private void browseConfigFile() {

```

```

        // Implementasi pemilihan dan pemuatan file konfigurasi
    }

    /**
     * Menjalankan proses pencarian solusi dengan algoritma dan
     * heuristik yang dipilih.
     */
    private void solvePuzzle() {
        // Implementasi algoritma pencarian solusi
    }

    /**
     * Memvisualisasikan hasil pencarian dari solver.
     *
     * @param solver Solver yang telah menyelesaikan pencarian
     */
    private void visualizeSolverResults(Solver solver) {
        // Implementasi visualisasi hasil pencarian
    }

    /**
     * Menampilkan langkah sebelumnya dalam solusi.
     */
    private void showPreviousStep() {
        // Implementasi navigasi ke langkah sebelumnya
    }

    /**
     * Menampilkan langkah berikutnya dalam solusi.
     */
    private void showNextStep() {
        // Implementasi navigasi ke langkah berikutnya
    }

    /**
     * Memperbarui visualisasi berdasarkan langkah saat ini.
     */
    private void updateVisualization() {
        // Implementasi pembaruan tampilan visualisasi
    }

    /**
     * Mengatur ulang visualisasi ke langkah awal.
     */
    private void restartVisualization() {
        // Implementasi pengaturan ulang visualisasi
    }

```

```

/**
 * Beralih antara status play dan pause pada animasi.
 */
private void togglePlayPause() {
    // Implementasi pengalihan status animasi
}

/**
 * Memulai pemutaran otomatis langkah-langkah solusi.
 */
private void startAnimation() {
    // Implementasi mulai animasi
}

/**
 * Menghentikan pemutaran otomatis langkah-langkah solusi.
 */
private void stopAnimation() {
    // Implementasi berhenti animasi
}

/**
 * Memperbarui kecepatan animasi berdasarkan nilai slider.
 */
private void updateAnimationSpeed() {
    // Implementasi pengaturan kecepatan animasi
}

/**
 * Membuka jendela permainan interaktif dengan konfigurasi saat ini.
 */
private void openGameWindow() {
    // Implementasi pembukaan jendela permainan
}

/**
 * Menyimpan solusi lengkap ke file teks.
 */
private void saveSolutionToFile() {
    // Implementasi penyimpanan solusi ke file
}

/**
 * Menghasilkan laporan teks lengkap tentang solusi.
 *
 * @return String berisi laporan solusi terformat
 */

```

```

private String generateSolutionReport() {
    // Implementasi pembuatan laporan solusi
}

/**
 * Titik masuk aplikasi.
 *
 * @param args Parameter baris perintah (tidak digunakan)
 */
public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
        @Override
        public void run() {
            RushHourSolverApp mainApp = new RushHourSolverApp();
            mainApp.setVisible(true);
        }
    });
}

```

4.4. Graphical User Interface (Bonus)

4.4.1. BoardDrawingUtil.java

```

package gui;

import java.awt.Component;
import java.awt.Graphics;

/**
 * Utilitas untuk visualisasi papan permainan.
 * Kelas ini menyediakan fungsi-fungsi untuk menggambar papan
 * permainan Rush Hour
 */
public class BoardDrawingUtil {

    /**
     * Metode ini menggambar grid papan, piece-piece, dan pintu
     keluar
     *
     * @param g Konteks grafis untuk menggambar
     * @param grid Data grid papan (karakter yang mewakili
     piece)
     * @param rows Jumlah baris papan
     * @param cols Jumlah kolom papan
     * @param exitX Koordinat X pintu keluar
     */
}

```

```

        * @param exitY Koordinat Y pintu keluar
        * @param exitSide Sisi pintu keluar (0=atas, 1=kanan,
        2=bawah, 3=kiri)
        * @param targetComponent Komponen yang digunakan untuk
        menentukan ukuran
        */
        public static void drawBoard(Graphics g, char[][] grid, int
        rows, int cols, int exitX, int exitY, int exitSide, Component
        targetComponent) {
            // Implementasi penggambaran papan permainan
        }
    }
}

```

4.4.2. PieceColorManager.java

```

package gui;

import java.awt.Color;
import java.util.Map;

/**
 * Kelas utilitas untuk manajemen warna piece.
 */
public class PieceColorManager {
    private static final Map<Character, Color> pieceColors = new
    HashMap<>();

    /**
     * Static initializer yang mengisi peta warna dengan nilai
     default.
     * Piece utama (P) berwarna merah, dan piece lainnya
     memiliki warna yang telah ditentukan.
     */
    static {
        // Inisialisasi warna-warna standar untuk berbagai jenis
        piece
    }

    /**
     * Mendapatkan warna untuk label piece tertentu.
     * Jika piece belum memiliki warna yang ditetapkan, akan
     dibuat warna baru secara otomatis.
     *
     * @param pieceLabel Karakter label piece
     * @return Warna untuk piece tersebut
     */
}

```



```

        public static Color getColorForPiece(char pieceLabel) {
            // Implementasi pengambilan atau pembuatan warna untuk
piece
        }

        /**
         * Mendapatkan seluruh peta warna.
         *
         * @return Salinan dari peta warna piece
         */
        public static Map<Character, Color> getColorMap() {
            // Implementasi pengembalian salinan peta warna
        }
    }
}

```

4.4.3. PlayPuzzle.java

```

package gui;

import java.awt.Graphics;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JComboBox;
import javax.swing.JFileChooser;

/**
 * Kelas untuk mode permainan interaktif Rush Hour.
 * Memungkinkan pengguna bermain puzzle Rush Hour secara manual
dengan
 * menggerakkan piece menggunakan kontrol GUI.
 */
public class PlayPuzzle extends JFrame {

    /**
     * Konstruktor yang membuat jendela permainan interaktif.
     * Menginisialisasi komponen UI dan mengatur tata letak
jendela.
     */
    public PlayPuzzle() {
        // Implementasi inisialisasi UI permainan
    }

    /**
     * Menginisialisasi antarmuka pengguna untuk mode permainan.

```

```

        * Membuat panel, tombol, dan kontrol lainnya yang
        diperlukan.
    */
    private void initializeGUI() {
        // Implementasi pembuatan komponen UI
    }

    /**
     * Memuat GameState ke dalam mode permainan.
     *
     * @param state GameState yang akan dimuat
     */
    public void loadGameState(GameState state) {
        // Implementasi pemuatan state permainan
    }

    /**
     * Memuat konfigurasi puzzle dari file.
     * Membuka dialog pemilihan file dan memuat konfigurasi yang
     dipilih.
     */
    private void loadConfig() {
        // Implementasi pemuatan konfigurasi dari file
    }

    /**
     * Menggerakkan piece yang dipilih.
     *
     * @param forward true untuk bergerak maju, false untuk
     mundur
     */
    private void movePiece(boolean forward) {
        // Implementasi pergerakan piece
    }

    /**
     * Menggambar papan permainan dan piece-piecenya.
     *
     * @param g Konteks grafis untuk menggambar
     */
    private void drawBoard(Graphics g) {
        // Implementasi penggambaran papan permainan
    }

    /**
     * Memperbarui pemilih piece dengan piece yang tersedia.
     * Mengisi dropdown dengan label piece yang ada di papan.
     */

```

```

private void updatePieceSelector() {
    // Implementasi pembaruan dropdown pemilih piece
}

/**
 * Memeriksa apakah permainan telah diselesaikan.
 *
 * @return true jika puzzle telah diselesaikan
 */
private boolean checkWinCondition() {
    // Implementasi pemeriksaan kondisi kemenangan
}
}

```

4.5. Logic

```

package logic;

/**
 * Kelas Board: Representasi papan permainan Rush Hour.
 */
public class Board {
    // Konstruktor untuk membuat papan baru dengan ukuran dan
    // posisi exit tertentu
    public Board(int rows, int cols, int exitX, int exitY, int
    exitSide) {}

    // Menampilkan representasi papan ke konsol
    public void printBoard() {}

    // Mendapatkan array koordinat exit [x,y]
    public int[] getOutCoord() {}

    // Mendapatkan koordinat X dari exit
    public int getOutCoordX() {}

    // Mendapatkan koordinat Y dari exit
    public int getOutCoordY() {}

    // Mendapatkan grid papan permainan
    public char[][] getGrid() {}

    // Mendapatkan sisi papan tempat exit berada (0=atas,
    // 1=kanan, 2=bawah, 3=kiri)
    public int getExitSide() {}
}

```

```

        // Mengatur grid papan permainan
        public void setGrid(char[][] newGrid) {}
    }

    /**
     * Kelas GameState: Menyimpan keadaan permainan dan posisi semua
     * bidak.
     */
    public class GameState {
        // Konstruktor untuk membuat state permainan baru
        public GameState(Board board, PrimaryPiece primaryPiece,
            Map<Character, Piece> pieces) {}

        // Mendapatkan papan permainan
        public Board getBoard() {}

        // Mendapatkan state bidak utama
        public PieceState getPrimaryPieceState() {}

        // Mendapatkan semua bidak non-utama
        public Map<Character, PieceState> getPieces() {}

        // Kelas dalam untuk menyimpan state satu bidak
        public static class PieceState {
            // Mendapatkan bidak
            public Piece getPiece() {}

            // Mendapatkan koordinat X bidak
            public int getX() {}

            // Mendapatkan koordinat Y bidak
            public int getY() {}
        }
    }

    /**
     * Kelas GameLogic: Mengatur operasi logika permainan Rush Hour.
     */
    public class GameLogic {
        // Membuat salinan dari GameState
        public static GameState copyGameState(GameState gameState)
        {}

        // Menggerakkan bidak ke arah tertentu (0=kanan, 1=bawah,
        2=kiri, 3=atas)
        public static GameState movePiece(GameState gameState, char
        pieceLabel, int direction) {}
    }

```

```

        // Memperbarui grid papan berdasarkan posisi bidak
        public static void updateBoardGrid(GameState gameState) {}

        // Menghasilkan semua kemungkinan langkah selanjutnya
        public static List<Node> generateSuccessors(Node
currentNode, solver.heuristic.Heuristic heuristic) {}

        // Menempatkan bidak pada grid
        private static void placePieceOnGrid(char[][] grid,
GameState.PieceState pieceState) {}

        // Memeriksa apakah terjadi tabrakan antara bidak
        private static boolean wouldCollide(GameState gameState,
char pieceLabel, int newX, int newY) {}

        // Membuat GameState baru dengan posisi bidak yang
diperbarui
        private static GameState
createNewStateWithMovedPiece(GameState gameState, char
pieceLabel, int newX, int newY) {}
    }

/**
 * Kelas Piece: Bidak dasar pada permainan Rush Hour.
 */
public class Piece {
    // Konstruktor untuk membuat bidak baru
    public Piece(char label, int size, boolean isHorizontal) {}

    // Mendapatkan label bidak
    public char getLabel() {}

    // Mendapatkan ukuran bidak
    public int getSize() {}

    // Memeriksa apakah bidak horizontal atau vertikal
    public boolean isHorizontal() {}
}

/**
 * Kelas PrimaryPiece: Bidak utama yang harus mencapai exit.
 */
public class PrimaryPiece extends Piece {
    // Konstruktor untuk membuat bidak utama
    public PrimaryPiece(char label, int size, boolean
isHorizontal) {}

```

```

        // Konstruktor dengan opsi highlight
        public PrimaryPiece(char label, int size, boolean
isHorizontal, boolean isHighlighted) {}

        // Memeriksa apakah bidak utama di-highlight
        public boolean isHighlighted() {}
    }

/**
 * Kelas Node: Representasi simpul dalam pencarian solusi.
 */
public class Node implements Comparable<Node> {
    // Konstruktor untuk node awal
    public Node(GameState state) {}

    // Konstruktor untuk node turunan
    public Node(GameState state, Node parent, int cost, int
heuristicValue, String moveMade) {}

    // Memeriksa apakah node ini mencapai keadaan tujuan
    public boolean isGoalState() {}

    // Mendapatkan state permainan dari node ini
    public GameState getState() {}

    // Mendapatkan node induk
    public Node getParent() {}

    // Mendapatkan biaya dari akar ke node ini
    public int getCost() {}

    // Mendapatkan nilai heuristik
    public int getHeuristicValue() {}

    // Mendapatkan total cost (f = g + h untuk A*)
    public int getTotalCost() {}

    // Mendapatkan deskripsi gerakan yang menghasilkan node ini
    public String getMoveMade() {}

    // Membandingkan node berdasarkan biaya
    @Override
    public int compareTo(Node other) {}
}

/**
 * Kelas GameManager: Mengatur operasi dasar permainan Rush
Hour.

```

```

*/
public class GameManager {
    // Memuat state permainan dari input pengguna
    public static GameState loadGameState() {}

    // Memuat state permainan dari path file tertentu
    public static GameState loadGameStateFromFile(String
configPath) {}

    // Mendapatkan path absolut ke resource dalam classpath
    public static String getResourcePath(String relativePath) {}

    // Menampilkan menu utama permainan
    public static void showMainMenu() {}

    // Menangani interaksi menu permainan
    public static void handleMenuChoice(int choice) {}

    // Mengecek apakah permainan telah diselesaikan
    public static boolean isGameSolved(GameState state) {}
}

```

4.6. Solver

4.6.1. Algorithm

```

package solver.algorithm;

/**
 * Kelas Solver: Kelas abstrak dasar untuk semua algoritma
pencarian.
 */
public abstract class Solver {
    // Menjalankan algoritma pencarian dari keadaan awal
    public abstract Node solve(GameState initialState);

    // Mendapatkan jalur solusi dari node tujuan ke node awal
    protected List<Node> buildPath(Node goalNode) {}

    // Mendapatkan jalur solusi yang sudah ditemukan
    public List<Node> getSolutionPath() {}

    // Mendapatkan jumlah node yang dieksplorasi selama
pencarian
    public int getNodesExplored() {}
}

```

```

        // Mendapatkan waktu eksekusi algoritma dalam milidetik
        public long getExecutionTimeMs() {}

        // Mendapatkan ukuran maksimum frontier selama pencarian
        public int getMaxQueueSize() {}

        // Mendapatkan nama algoritma pencarian
        public abstract String getAlgorithmName();
    }

    /**
     * Kelas InformedSolver: Kelas abstrak untuk algoritma pencarian
     * berbasis heuristik.
     */
    public abstract class InformedSolver extends Solver {
        // Konstruktor yang menerima fungsi heuristik
        public InformedSolver(Heuristic heuristic) {}

        // Mendapatkan fungsi heuristik yang digunakan
        public Heuristic getHeuristic() {}

        // Mendapatkan nama algoritma pencarian
        @Override
        public abstract String getAlgorithmName();
    }

    /**
     * Kelas UCSolver: Implementasi algoritma Uniform Cost Search.
     */
    public class UCSolver extends Solver {
        // Menjalankan algoritma UCS dari keadaan awal
        @Override
        public Node solve(GameState initialState) {}

        // Mendapatkan nama algoritma pencarian
        @Override
        public String getAlgorithmName() { return "Uniform Cost
Search"; }
    }

    /**
     * Kelas BestFSolver: Implementasi algoritma Greedy Best-First
     * Search.
     */
    public class BestFSolver extends InformedSolver {
        // Konstruktor yang menerima fungsi heuristik
        public BestFSolver(Heuristic heuristic) {}
    }

```



```

        // Menjalankan algoritma GBFS dari keadaan awal
        @Override
        public Node solve(GameState initialState) {}

        // Mendapatkan nama algoritma pencarian
        @Override
        public String getAlgorithmName() { return "Greedy Best-First
Search"; }
    }

/**
 * Kelas AStarSolver: Implementasi algoritma A* Search.
 */
public class AStarSolver extends InformedSolver {
    // Konstruktor yang menerima fungsi heuristik
    public AStarSolver(Heuristic heuristic) {}

    // Konstruktor default yang menggunakan
BlockingPiecesHeuristic
    public AStarSolver() {}

    // Menjalankan algoritma A* dari keadaan awal
    @Override
    public Node solve(GameState initialState) {}

    // Mendapatkan nama algoritma pencarian
    @Override
    public String getAlgorithmName() { return "A* Search"; }
}

/**
 * Kelas BranchAndBoundSolver: Implementasi algoritma Branch and
Bound.
 */
public class BranchAndBoundSolver extends InformedSolver {
    // Konstruktor yang menerima fungsi heuristik
    public BranchAndBoundSolver(Heuristic heuristic) {}

    // Konstruktor default yang menggunakan
BlockingPiecesHeuristic
    public BranchAndBoundSolver() {}

    // Menjalankan algoritma Branch and Bound dari keadaan awal
    @Override
    public Node solve(GameState initialState) {}

    // Mendapatkan nama algoritma pencarian

```

```
        @Override
        public String getAlgorithmName() { return "Branch and
Bound"; }
    }
```

4.6.2. Heuristic

```
package solver.heuristic;

/**
 * Heuristic: Interface dasar yang harus diimplementasikan oleh
 * semua fungsi heuristik.
 */
public interface Heuristic {
    // Menghitung nilai heuristik untuk state permainan tertentu
    int calculate(GameState state);

    // Mendapatkan nama heuristik untuk tujuan pelaporan
    String getName();
}

/**
 * BlockingPiecesHeuristic: Menghitung jumlah bidak yang
 * menghalangi jalur bidak utama ke exit.
 */
public class BlockingPiecesHeuristic implements Heuristic {
    // Menghitung jumlah bidak penghalang di jalur menuju exit
    @Override
    public int calculate(GameState state) {}

    // Mendapatkan nama heuristik
    @Override
    public String getName() { return "Blocking Pieces"; }
}

/**
 * ManhattanDistanceHeuristic: Menggunakan jarak Manhattan dari
 * bidak utama ke exit.
 */
public class ManhattanDistanceHeuristic implements Heuristic {
    // Menghitung jarak Manhattan ke exit
    @Override
    public int calculate(GameState state) {}

    // Mendapatkan nama heuristik
    @Override
```

```

        public String getName() { return "Manhattan Distance"; }
    }

    /**
     * DistanceToExitHeuristic: Menghitung jarak khusus dari bidak
     * utama ke exit.
     */
    public class DistanceToExitHeuristic implements Heuristic {
        // Menghitung jarak dari bidak utama ke exit berdasarkan
        // arah exit
        @Override
        public int calculate(GameState state) {}

        // Mendapatkan nama heuristik
        @Override
        public String getName() { return "Distance To Exit"; }
    }

    /**
     * PieceDensityHeuristic: Mengukur kepadatan bidak di sekitar
     * jalur ke exit.
     */
    public class PieceDensityHeuristic implements Heuristic {
        // Menghitung kepadatan bidak di sekitar jalur menuju exit
        @Override
        public int calculate(GameState state) {}

        // Mendapatkan nama heuristik
        @Override
        public String getName() { return "Piece Density"; }
    }

    /**
     * CombinedHeuristic: Menggabungkan beberapa heuristik untuk
     * hasil yang lebih akurat.
     */
    public class CombinedHeuristic implements Heuristic {
        // Konstruktor yang menginisialisasi semua heuristik yang
        // akan digabungkan
        public CombinedHeuristic() {}

        // Menghitung nilai gabungan dari beberapa heuristik
        @Override
        public int calculate(GameState state) {}

        // Mendapatkan nama heuristik
        @Override
        public String getName() { return "Combined"; }
    }

```

```
}
```

4.7. Util

```
package util;

/**
 * Kelas ConfigParser: Utilitas untuk membaca file konfigurasi
 * puzzle Rush Hour.
 */
public class ConfigParser {
    // Membaca file konfigurasi dan membuat GameState dengan
    // papan dan bidak yang sesuai
    public static GameState parseConfig(String filePath) throws
    IOException {}

    // Mencari posisi exit berdasarkan label khusus ('K') di
    // papan
    private static int[] findExitPosition(List<String>
    boardLines) {}

    // Menentukan sisi exit berdasarkan posisinya (atas, kanan,
    // bawah, kiri)
    private static int determineExitSide(int exitX, int exitY,
    int rows, int cols) {}

    // Mendeteksi dan membuat objek bidak dari representasi teks
    // papan
    private static Map<Character, Piece>
    detectPieces(List<String> boardLines, char primaryLabel) {}

    // Menghitung ukuran bidak berdasarkan kemunculan labelnya
    // di papan
    private static int calculatePieceSize(List<String>
    boardLines, char label) {}

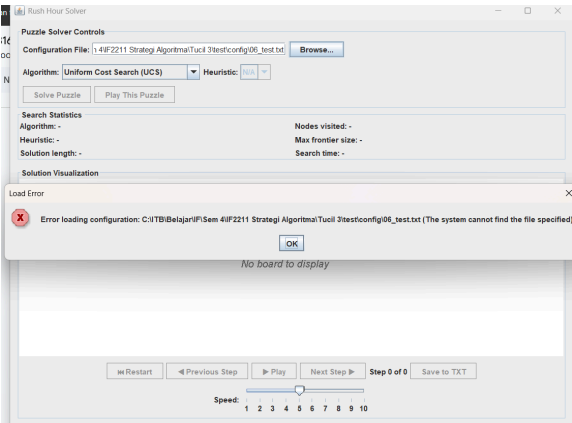
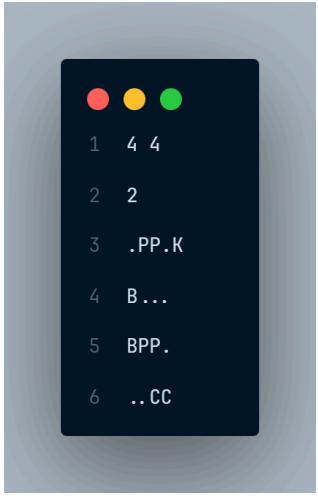
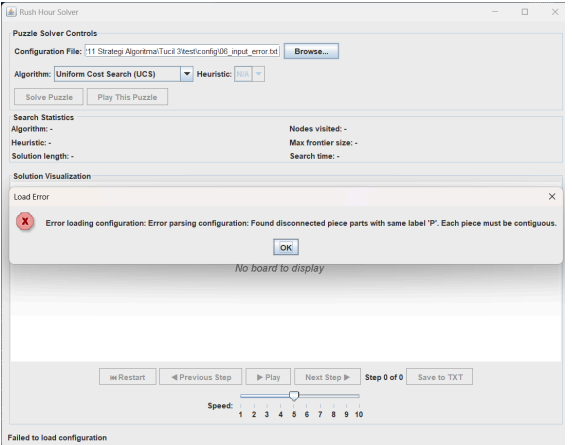
    // Menentukan apakah bidak horizontal atau vertikal
    // berdasarkan posisinya
    private static boolean isPieceHorizontal(List<String>
    boardLines, char label) {}


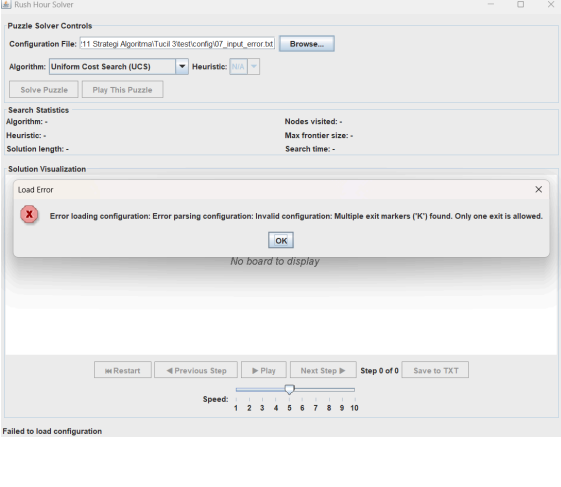
    // Mencari koordinat awal dari bidak dengan label tertentu
    private static int[] findPieceStart(List<String> boardLines,
    char label) {}
}
```


5. Uji Coba



Program diuji coba dengan menggunakan file config berformat **.txt**. Sample dan hasilnya dapat dilihat pada folder /test.

5.1. Input

Input	Output	Keterangan
<nama file tidak ditemukan>		Melempar <i>exception</i>
		Blok tidak boleh terpisah jika labelnya sama

		<p>Hanya boleh ada satu exit</p>
---	--	----------------------------------

5.2. Uniform Cost Search

Board Config	Hasil (Final Board State)	Keterangan
		<p>Board size: 6x6 Exit position: (6,2) Exit side: Right</p> <p>Algorithm: Uniform Cost Search Heuristic: None Solution: Found Solution length: 5 moves Nodes visited: 530 Maximum frontier size: 603 Execution time: 126 ms</p>

Board size: 6x6
Exit position: (-1,2)
Exit side: Left

Algorithm: Uniform Cost Search
Heuristic: None
Solution: Found
Solution length: 4 moves
Nodes visited: 244
Maximum frontier size: 352
Execution time: 132 ms

Algorithm: Uniform Cost Search (UCS) Heuristic: N/A

Solve Puzzle Play This Puzzle

Search Statistics

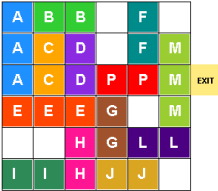
Algorithm: Uniform Cost Search	Nodes visited: 334
Heuristic: None	Max frontier size: 406
Solution length: 4 moves	Search time: 47ms

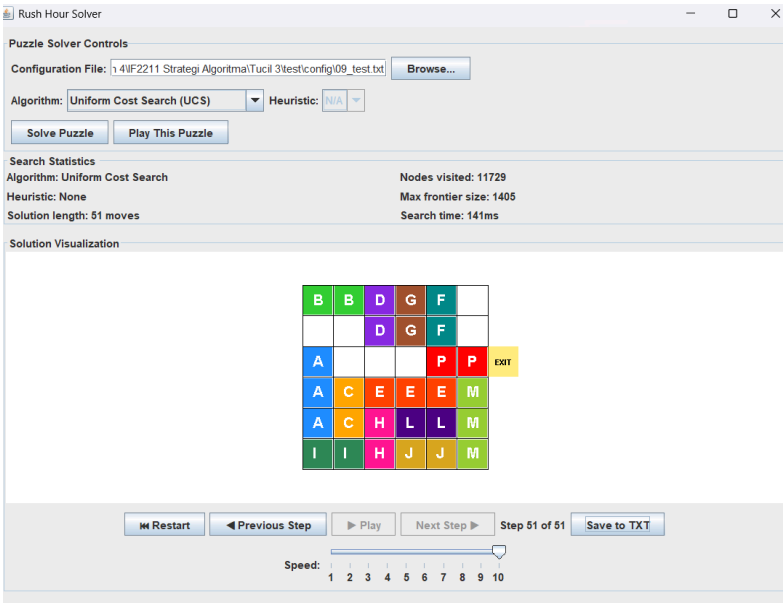
Solution Visualization

EXIT

Restart Previous Step Play Next Step Step 4 of 4 Save to TXT

Speed: 1 2 3 4 5 6 7 8 9 10





Rush Hour Solver

Puzzle Solver Controls

Configuration File: h 4VF2211 Strategi AlgoritmaTucil 3testconfig09_test.txt Browse...

Algorithm: Uniform Cost Search (UCS) Heuristic: N/A

Solve Puzzle Play This Puzzle

Search Statistics

Algorithm: Uniform Cost Search

Heuristic: None

Solution length: 51 moves

Nodes visited: 11729

Max frontier size: 1405

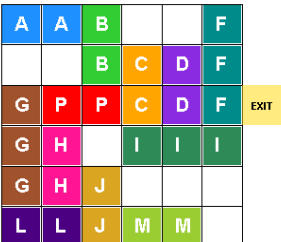
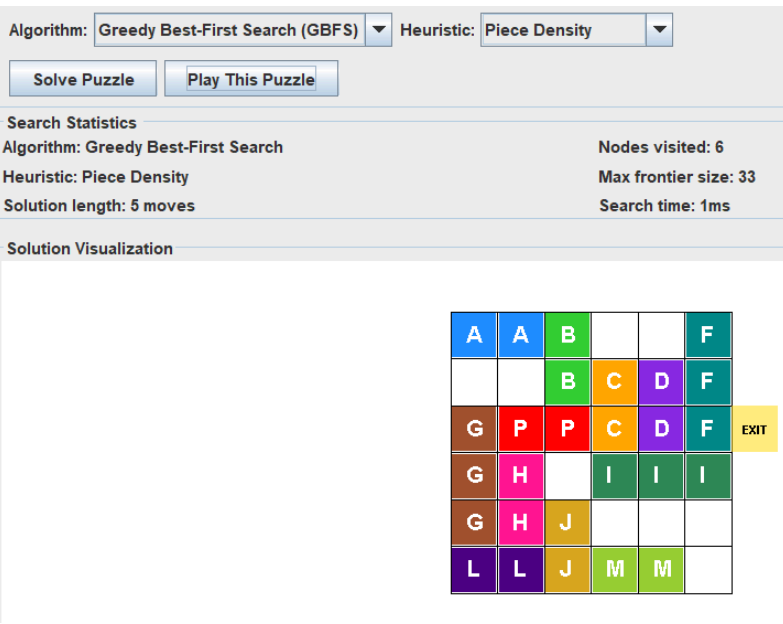
Search time: 141ms

Solution Visualization

Restart Previous Step Play Next Step Step 51 of 51 Save to TXT

Speed: 1 2 3 4 5 6 7 8 9 10

5.3. Greedy Best-First Search

Board Config	Hasil (Final Board State)	Keterangan
		

A	A	B			F
		B	C	D	F
G	H	H	C	D	F
G	P		I	I	I
G	P	J			
L	L	J	M	M	

EXIT

Algorithm: Greedy Best-First Search (GBFS) Heuristic: Distance To Exit

Solve Puzzle Play This Puzzle

Search Statistics

Algorithm: Greedy Best-First Search Nodes visited: 17

Heuristic: Distance To Exit Max frontier size: 7

Solution length: 9 moves Search time: 1ms

Solution Visualization

A	A	B	C		F
		B	C	D	F
G	H	H		D	F
G		J	I	I	I
G	P	J			
	P	L	L	M	M

EXIT

A	B	B		F	
A	C	D		F	M
A	C	D	P	P	M
E	E	E	G		M
		H	G	L	L
I	I	H	J	J	

EXIT

Algorithm: Greedy Best-First Search (GBFS) Heuristic: Manhattan Distance

Solve Puzzle Play This Puzzle

Search Statistics

Algorithm: Greedy Best-First Search Nodes visited: 12945

Heuristic: Manhattan Distance Max frontier size: 5625

Solution length: 187 moves Search time: 54ms

Solution Visualization

B	B	D	G	F	
A	C	D	G	F	
A	C			P	P
A	E	E	E		M
L	L	H			M
I	I	H	J	J	M

EXIT

Algorithm: Greedy Best-First Search (GBFS) ▾

Heuristic: Piece Density ▾

Solve Puzzle

Play This Puzzle

Search Statistics

Algorithm: Greedy Best-First Search

Heuristic: Piece Density

Solution length: 90 moves

Nodes visited: 4518


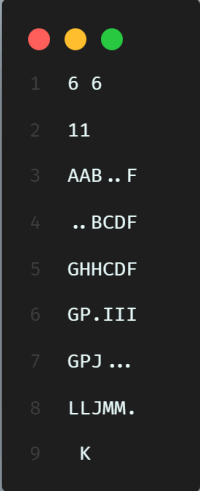
Max frontier size: 967

Search time: 26ms

Solution Visualization

5.4. A* Search

Board Config	Hasil (Final Board State)	Keterangan
		<p>Board size: 6x6 Exit position: (6,2) Exit side: Right</p> <p>Algorithm: A* Search Heuristic: Manhattan Distance Solution: Found Solution length: 5 moves Nodes visited: 1311 Maximum frontier size: 1028 Execution time: 89 ms</p>

 <pre> 1 6 6 2 11 3 AAB .. F 4 .. BCDF 5 KGPPCDF 6 GH.III 7 GHJ ... 8 LLJMM. </pre>	 <pre> 1 AAB .. F 2 .. BCDF 3 KPP.CDF 4 GHJIII 5 GHJ ... 6 GLLMM. </pre>	<p>Board size: 6x6 Exit position: (-1,2) Exit side: Left</p> <p>Algorithm: A* Search Heuristic: Distance to Exit Solution: Found Solution length: 4 moves Nodes visited: 81 Maximum frontier size: 177 Execution time: 9 ms</p>
 <pre> 1 6 6 2 11 3 AAB .. F 4 .. BCDF 5 GHHCDF 6 GP.III 7 GPJ ... 8 LLJMM. 9 K </pre>	 <pre> 1 AAB .. F 2 .. BCDF 3 GHHCDF 4 G.JIII 5 GPJ ... 6 .PLLMM 7 K </pre>	<p>Board size: 6x6 Exit position: (1,6) Exit side: Bottom</p> <p>Algorithm: A* Search Heuristic: Piece Density Solution: Found Solution length: 4 moves Nodes visited: 9 Maximum frontier size: 43 Execution time: 1 ms</p>

Rush Hour Solver

Puzzle Solver Controls

Configuration File: Browse...

Algorithm: A* Search Heuristic: Distance To Exit

Solve Puzzle Play This Puzzle

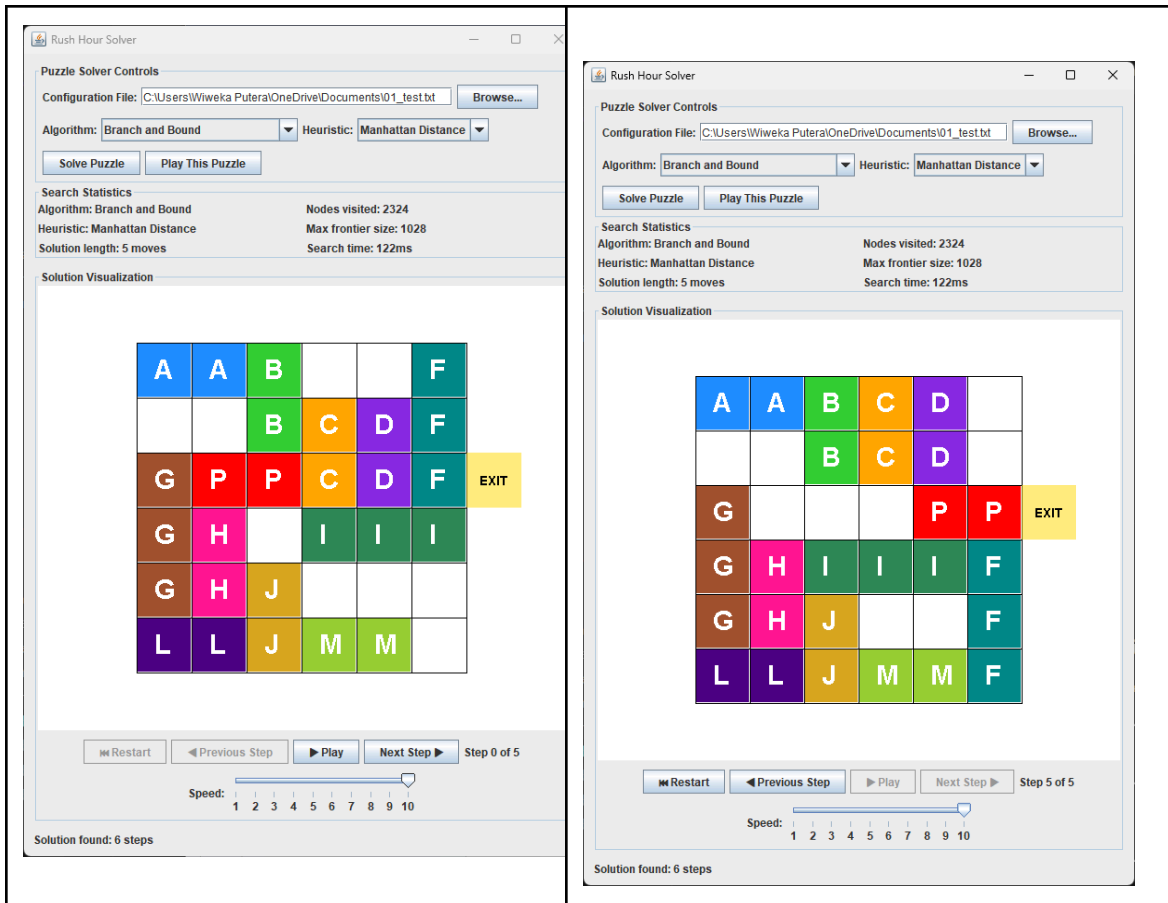
Search Statistics

Algorithm: A* Search	Nodes visited: 10109
Heuristic: Distance to Exit	Max frontier size: 1608
Solution length: 51 moves	Search time: 51ms

Solution Visualization

5.5. Branch and Bound

Board Config	Hasil (Final Board State)
--------------	---------------------------



Rush Hour Solver

Puzzle Solver Controls
Configuration File: C:\Users\Wiweka Putera\OneDrive\Documents\02_test.txt [Browse...](#)
Algorithm: Branch and Bound Heuristic: Blocking Pieces
[Solve Puzzle](#) [Play This Puzzle](#)

Search Statistics
Algorithm: Branch and Bound Nodes visited: 368
Heuristic: Blocking Pieces Max frontier size: 229
Solution length: 4 moves Search time: 17ms

Solution Visualization

	A	A	B			F
			B	C	D	F
EXIT	G	P	P	C	D	F
	G	H		I	I	I
	G	H	J			
	L	L	J	M	M	

[Restart](#) [Previous Step](#) [Play](#) [Next Step](#) Step 0 of 4
Speed: 1 2 3 4 5 6 7 8 9 10

Solution found: 5 steps

Rush Hour Solver

Puzzle Solver Controls
Configuration File: C:\Users\Wiweka Putera\OneDrive\Documents\02_test.txt [Browse...](#)
Algorithm: Branch and Bound Heuristic: Blocking Pieces
[Solve Puzzle](#) [Play This Puzzle](#)

Search Statistics
Algorithm: Branch and Bound Nodes visited: 368
Heuristic: Blocking Pieces Max frontier size: 229
Solution length: 4 moves Search time: 17ms

Solution Visualization

	A	A	B			F
			B	C	D	F
EXIT	P	P		C	D	F
	G	H	J	I	I	I
	G	H	J			
	G	L	L	M	M	

[Restart](#) [Previous Step](#) [Play](#) [Next Step](#) Step 4 of 4
Speed: 1 2 3 4 5 6 7 8 9 10

Solution found: 5 steps

A	B	B		F	
A	C	D		F	M
A	C	D	P	P	M
E	E	E	G		M
		H	G	L	L
I	I	H	J	J	

EXIT

Rush Hour Solver

Puzzle Solver Controls
Configuration File: \4\F2211 Strategi Algoritma\Tucil 3\test\config\09_test.txt [Browse...](#)
Algorithm: Branch and Bound Heuristic: Combined
[Solve Puzzle](#) [Play This Puzzle](#)

Search Statistics
Algorithm: Branch and Bound Nodes visited: 6907
Heuristic: Combined Heuristic Max frontier size: 1089
Solution length: 53 moves Search time: 41ms

Solution Visualization

A	B	B		F	
A	C	D		F	M
A	C	D	P	P	M
E	E	E	G		M
		H	G	L	L
I	I	H	J	J	

EXIT

A	B	B		F	
A	C	D		F	M
A	C	D	P	P	M
E	E	E	G		M
		H	G	L	L
I	I	H	J	J	

EXIT

Rush Hour Solver

Puzzle Solver Controls

Configuration File:

Algorithm: Heuristic:

Search Statistics

Algorithm: Branch and Bound

Heuristic: Distance to Exit

Solution length: 51 moves

Nodes visited: 11541

Max frontier size: 1608

Search time: 52ms

Solution Visualization

B	B	D	G	F	
		D	G	F	
A				P	P
A	C	E	E	E	M
A	C	H	L	L	M
I	I	H	J	J	M

EXIT

6. Analisis Kompleksitas

6.1. Analisis Heuristik

Distance To Exit Heuristic

Kompleksitas Waktu $T(n)$: $O(1)$

Heuristik ini hanya melakukan perhitungan aritmatika sederhana berdasarkan posisi kendaraan utama dan lokasi pintu keluar. Semua operasi dalam heuristik ini (pengurangan, penjumlahan, perbandingan) dilakukan dalam waktu konstan, tanpa tergantung ukuran papan atau jumlah kendaraan.

Kompleksitas Ruang $S(n)$: $O(1)$

Heuristik ini hanya menggunakan sejumlah tetap variabel untuk menyimpan informasi posisi. Jumlah variabel yang digunakan tidak tergantung pada ukuran input (papan atau jumlah kendaraan).

Blocking Pieces Heuristic

Kompleksitas Waktu $T(n)$: $O(n)$, dengan n adalah panjang papan (untuk kasus terburuk)

Heuristik ini melakukan traversal dari posisi kendaraan utama hingga pintu keluar untuk menghitung jumlah kendaraan penghalang.

- Untuk pintu keluar di kanan/kiri: $O(\text{lebar papan})$
- Untuk pintu keluar di atas/bawah: $O(\text{tinggi papan})$
- Dalam kasus terburuk, algoritma mungkin memeriksa seluruh baris atau kolom papan

Kompleksitas Ruang $S(n)$: $O(1)$

Tidak ada struktur data tambahan yang ukurannya bergantung pada input.

Manhattan Distance Heuristic

Kompleksitas Waktu $T(n)$: $O(n + k)$ dengan n adalah dimensi papan dan k adalah jumlah kendaraan penghalang

Heuristik ini melakukan dua operasi utama:

- Mengidentifikasi kendaraan penghalang pada jalur ke pintu keluar: $O(n)$
- Untuk setiap kendaraan penghalang, menghitung jarak minimum untuk memindahkannya: $O(1)$ per kendaraan
- $T(n) = O(n + k)$ dengan n adalah dimensi papan dan k adalah jumlah kendaraan penghalang

Dalam praktiknya, $k \leq n$, sehingga $T(n) = O(n)$

Kompleksitas Ruang $S(n)$: $O(k)$ dengan k adalah jumlah kendaraan penghalang

Ukuran map bergantung pada jumlah kendaraan penghalang yang ditemukan (maksimal sebanyak jumlah kendaraan di papan).

Piece Density Heuristic

Kompleksitas Waktu $T(n)$: $O(n^2)$

Heuristik ini memeriksa semua sel dalam "koridor" antara kendaraan utama dan pintu keluar.

- Dalam kasus terburuk (koridor lebar mencakup sebagian besar papan), kita memeriksa $O(\text{lebar} \times \text{tinggi})$ sel
- Dimensi papan adalah $n \times n$, sehingga $T(n) = O(n^2)$

Kompleksitas Ruang $S(n)$: $O(1)$

Tidak ada struktur data tambahan yang ukurannya bergantung pada input.

Combined Heuristic

Kompleksitas Waktu $T(n)$: $O(n^2)$

Heuristik ini menggabungkan keempat heuristik sebelumnya.

- Distance To Exit: $O(1)$
- Blocking Pieces: $O(n)$
- Manhattan Distance: $O(n)$
- Piece Density: $O(n^2)$

Total: $O(1 + n + n + n^2) = O(n^2)$

Kompleksitas Ruang $S(n)$: $O(k)$ dengan k adalah jumlah kendaraan penghalang

Kompleksitas ruang didominasi oleh Manhattan Distance Heuristic yang memerlukan map untuk menyimpan informasi kendaraan penghalang.

Kesimpulan

Dari kelima heuristik yang diimplementasikan:

- Distance To Exit adalah yang paling efisien dengan kompleksitas waktu dan ruang konstan $O(1)$.
- Blocking Pieces dan Manhattan Distance memiliki kompleksitas waktu linear $O(n)$ terhadap dimensi papan, dengan Manhattan Distance menggunakan ruang tambahan $O(k)$ untuk menyimpan informasi kendaraan penghalang.
- Piece Density memiliki kompleksitas waktu kuadratik $O(n^2)$ dalam kasus terburuk karena perlu memeriksa area koridor dua dimensi.
- Combined Heuristic mewarisi kompleksitas terburuk dari keempat heuristik yang digabungkan, yaitu $O(n^2)$ untuk waktu dan $O(k)$ untuk ruang.

Dalam prakteknya, karena ukuran papan Rush Hour biasanya kecil dan tetap (6x6), perbedaan performa antar heuristik mungkin tidak signifikan. Namun, untuk papan yang lebih besar, Distance To Exit dan Blocking Pieces akan memiliki keunggulan performa dibandingkan Piece Density dan Combined Heuristic.

6.2. Analisis Kerja Pencarian

Uniform Cost Search (UCS)

Kompleksitas Waktu $T(n)$: $O(b^d)$ di mana b adalah faktor percabangan dan d adalah kedalaman solusi. Dihitung dengan:

1. UCS akan mengeksplorasi semua node dengan biaya kurang dari biaya solusi
2. Jumlah node hingga kedalaman d adalah $O(b^d)$
3. Dalam Rush Hour, b adalah jumlah rata-rata kemungkinan gerakan yang valid (pergeseran kendaraan)
4. Setiap node diproses tepat sekali karena UCS selalu memilih node dengan biaya terendah

Kompleksitas Ruang $S(n)$: $O(b^d)$, dihitung dengan:

1. UCS menyimpan semua node yang telah dieksplorasi dalam set visited
2. Selain itu juga memelihara priority queue berisi node yang belum dieksplorasi
3. Dalam kasus terburuk, hampir semua node dalam ruang pencarian hingga kedalaman d harus disimpan dalam memori

Greedy Best-First Search

Kompleksitas Waktu $T(n)$: $O(b^m)$, di mana m adalah kedalaman maksimum ruang pencarian.

- Greedy Best-First Search tidak menjamin eksplorasi node berdasarkan kedalaman atau biaya path
- Dalam kasus terburuk, algoritma mungkin mengeksplorasi seluruh ruang pencarian
- Namun, jika heuristik sangat baik, kompleksitas waktu bisa jauh lebih baik

Kompleksitas Ruang $S(n)$: $O(b^m)$, dihitung dengan:

1. Algoritma menyimpan semua node yang dieksplorasi dalam set visited
2. Juga memelihara priority queue berdasarkan nilai heuristik
3. Dalam kasus terburuk, algoritma menyimpan hampir semua node dalam ruang pencarian

A* Search

*Kompleksitas Waktu $T(n)$: $O(b^d)^{**}$ di mana d adalah kedalaman solusi optimal. Dihitung dengan:*

1. Jika heuristik yang digunakan admissible dan konsisten, A* tidak akan mengeksplorasi node dengan $f(n) > C^*$ (C^* = biaya solusi optimal)
2. Jumlah node dengan $f(n) \leq C^*$ adalah eksponensial terhadap kedalaman solusi
3. Dalam implementasi Rush Hour, setiap node dieksplorasi sekali (karena set visited)

Kompleksitas Ruang $S(n)$: $O(b^d)$, dihitung dengan:

1. A* menyimpan semua node yang telah dieksplorasi dalam set visited
2. Juga memelihara priority queue berdasarkan nilai $f(n) = g(n) + h(n)$
3. Dalam kasus terburuk, algoritma perlu menyimpan semua node hingga kedalaman solusi

Branch and Bound

Kompleksitas Waktu $T(n)$: $O(b^d)$ dalam kasus terburuk, tetapi sering kali lebih baik karena pemangkasan. Dihitung dengan:

1. Branch and Bound serupa dengan A* namun dengan pemangkasan eksplisit
2. Setelah solusi pertama ditemukan, batas atas digunakan untuk memangkas cabang yang tidak menjanjikan
3. Jika solusi ditemukan cepat, kompleksitas bisa jauh lebih baik dari $O(b^d)$
4. Tetapi dalam kasus terburuk, pemangkasan minimal dan kompleksitas sama dengan A*

Kompleksitas Ruang $S(n)$: $O(b^d)$, dihitung dengan:

1. Menyimpan node yang sudah dieksplorasi dalam set visited
2. Memelihara priority queue yang diurut berdasarkan lower bound ($g(n) + h(n)$)
3. Dalam kasus terburuk, kompleksitas ruangnya sama dengan A*

Dalam praktiknya, untuk masalah Rush Hour:

- Ukuran ruang pencarian dan faktor percabangan (b) dipengaruhi oleh ukuran papan dan jumlah kendaraan
- Kedalaman solusi (d) bergantung pada kesulitan puzzle
- Kinerja algoritma heuristik (A*, Greedy, Branch and Bound) sangat bergantung pada efektivitas fungsi heuristik yang digunakan

Meskipun kompleksitas teoretis menunjukkan pertumbuhan eksponensial untuk semua algoritma, optimasi implementasi dan karakteristik domain Rush Hour (misalnya, jumlah gerakan yang valid terbatas) dapat menghasilkan kinerja yang lebih baik dalam praktik.

6.3. Analisis Hasil Pencarian

A Search* secara konsisten menunjukkan kinerja terbaik. Algoritma ini menemukan solusi optimal dengan jumlah eksplorasi node yang jauh lebih sedikit dibandingkan UCS, terutama pada puzzle kompleks. Keunggulannya semakin terlihat pada konfigurasi puzzle yang memerlukan banyak langkah untuk diselesaikan. Ketika menggunakan heuristik yang tepat, yakni *Distance to Exit*, A* mampu "mengarahkan" pencarian dengan sangat efektif.

Branch and Bound menempati posisi kedua dengan performa yang hampir menyamai A* pada banyak kasus. Namun, overhead komputasi untuk evaluasi pemangkasan kadang mengurangi kecepatannya secara keseluruhan.

Greedy Best-First Search menunjukkan waktu eksekusi tercepat untuk menemukan solusi, tetapi solusinya seringkali tidak optimal (memerlukan lebih banyak langkah). Algoritma ini sangat berguna jika tujuan utama adalah menemukan solusi yang "cukup baik" dengan cepat, bukan solusi optimal.

UCS (Breadth-First Search) menunjukkan performa terburuk untuk puzzle kompleks karena mengeksplorasi terlalu banyak state yang tidak relevan. Meskipun selalu menemukan solusi optimal, algoritma ini menghabiskan waktu dan memori yang jauh lebih besar dibanding algoritma berbasis heuristik.

7. Lampiran

Repositori: https://github.com/reletz/Tucil3_13523149_13523160

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	✓	