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

Oleh

Jonathan Kenan Budianto (13523139)

Bryan P. Hutagalung (18222130)

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika – Institut Teknologi Bandung

Jl. Ganesha 10, Bandung 40132

2025

Daftar Isi

Daftar Tabel

Daftar Gambar

Bab I

Pendahuluan

1.1 Deskripsi Tugas

Gambar 1. Rush Hour Puzzle

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

Komponen penting dari permainan Rush Hour terdiri dari:

1. **Papan**

   Papan merupakan tempat permainan dimainkan. Papan terdiri atas cell, yaitu

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

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

2. **Piece**

   Piece adalah sebuah kendaraan di dalam papan. Setiap piece memiliki posisi, ukuran, dan orientasi. Orientasi sebuah piece hanya dapat berupa horizontal atau vertikal–tidak mungkin diagonal. Piece dapat memiliki beragam ukuran, yaitu jumlah cell yang ditempati oleh piece. Secara standar, variasi ukuran sebuah piece adalah 2–piece (menempati 2 cell) atau 3–piece (menempati 3 cell). Suatu piece tidak dapat digerakkan melewati/menembus piece yang lain.

3. **Primary Piece**

   Primary piece adalah kendaraan utama yang harus dikeluarkan dari papan (biasanya berwarna merah). Hanya boleh terdapat satu primary piece.

4. **Pintu Keluar**

   Pintu keluar adalah tempat primary piece dapat digerakkan keluar untuk menyelesaikan permainan

5. **Gerakan**

   Gerakan yang dimaksudkan adalah pergeseran piece di dalam permainan. Piece hanya dapat bergerak/bergeser lurus sesuai orientasinya (atas-bawah jika vertikal dan kiri-kanan jika horizontal). Suatu piece tidak dapat digerakkan melewati/menembus piece yang lain.

1.2 Ilustrasi Kasus

Permainan dimulai dengan sebuah papan berukuran 6×6 yang berisi total 12 kendaraan (*piece*), salah satunya adalah kendaraan utama (*primary piece*) yang harus dikeluarkan dari papan. Setiap kendaraan diletakkan pada posisi tertentu dengan orientasi horizontal atau vertikal. Pada awal permainan, kendaraan-kendaraan tersebut membentuk kondisi macet sehingga *primary piece* tidak dapat langsung keluar. Pemain harus menggeser kendaraan lain terlebih dahulu agar dapat membuka jalur menuju pintu keluar yang terletak di dinding papan dan sejajar dengan arah *primary piece*.

Sebagai contoh, *primary piece* ditempatkan secara horizontal di tengah papan, dengan kendaraan lain menghalangi jalur keluarnya. Pemain dapat menggeser satu per satu kendaraan lain untuk membentuk jalan lurus. Setelah beberapa langkah, jalur menuju pintu keluar menjadi kosong, dan *primary piece* dapat digeser ke luar papan. Permainan dianggap selesai saat *primary piece* berhasil keluar melalui pintu tersebut.

Bab II
Landasan Teori Algoritma Pathfinding

2.1 Uniform Cost Search (UCS)

2.1.1 Penjelasan Algoritma UCS

Algoritma **Uniform Cost Search** atau **UCS** merupakan salah satu metode dalam route planning yang digunakan untuk menentukan rute tanpa memanfaatkan informasi tambahan mengenai lokasi tujuan (termasuk dalam kategori Uninformed Search atau Blind Search). Proses pencariannya berfokus pada nilai biaya atau cost yang diperlukan untuk mencapai suatu simpul dalam graf. Biaya ini dilambangkan sebagai g(n), di mana n menyatakan simpul dalam graf pencarian. Dalam pendekatan ini, simpul dengan cost terendah akan diprioritaskan untuk diekspansi terlebih dahulu hingga ditemukan solusi.

Karena algoritma ini tidak menggunakan informasi tentang letak tujuan, maka UCS digolongkan sebagai algoritma pencarian buta (Uninformed Search). Dalam praktiknya, simpul dengan nilai cost paling kecil akan diproses lebih dulu. Oleh karena itu, algoritma ini umumnya diimplementasikan menggunakan struktur data Priority Queue, sehingga simpul yang memiliki prioritas tertinggi (yakni cost terendah) akan berada di urutan terdepan untuk diproses.

Penentuan prioritas dalam UCS hanya bergantung pada besar biaya yang telah dikeluarkan untuk mencapai simpul tersebut. Maka, fungsi evaluasi yang digunakan dapat dirumuskan sebagai:

$$f(n) = g(n)$$

di mana f(n) merupakan fungsi evaluasi yang menentukan prioritas simpul, sedangkan g(n) adalah nilai biaya atau cost dari simpul tersebut.

2.1.2 Analisis Algoritma UCS

Langkah :

1. Buat sebuah priority queue yang akan menyimpan semua simpul yang akan dikunjungi, diurutkan berdasarkan total biaya dari simpul awal ke simpul tersebut, yang disebut sebagai g(n).
2. Tambahkan simpul awal ke dalam priority queue dengan nilai g(n) sama dengan nol karena belum ada langkah yang diambil.
3. Siapkan sebuah struktur data seperti set untuk mencatat semua simpul yang telah dikunjungi agar tidak dikunjungi ulang dan mencegah pencarian berputar-putar.
4. Selama priority queue masih memiliki elemen:
    - Ambil simpul dengan nilai g(n) paling kecil dari antrian, yaitu simpul dengan total biaya terkecil sejauh ini.
    - Periksa apakah simpul tersebut merupakan simpul tujuan. Jika ya, berarti jalur ditemukan dan pencarian dapat dihentikan dengan mengembalikan solusi.
    - Tandai simpul tersebut sebagai telah dikunjungi untuk menghindari eksplorasi ulang.
    - Bangkitkan semua simpul tetangga yang dapat dicapai dari simpul saat ini dengan memproses setiap kemungkinan gerakan yang valid.
    - Untuk setiap simpul tetangga yang dihasilkan, hitung total biaya g(n) dari simpul awal ke simpul tetangga tersebut dengan menambahkan biaya gerakan. Jika simpul tetangga belum pernah dikunjungi atau jalur yang ditemukan memiliki g(n) yang lebih kecil dari sebelumnya, tambahkan simpul tersebut ke priority queue agar dapat diproses lebih lanjut.
5. Jika antrian kosong dan simpul tujuan tidak pernah ditemukan selama pencarian, maka disimpulkan bahwa tidak ada solusi yang dapat dicapai dari konfigurasi awal tersebut.

Analisis :

1. Definisi f(n) dan g(n), f(n) = g(n) . g(n) adalah total biaya dari simpul awal hingga simpul n. Dalam konteks Rush Hour, ini dapat dihitung sebagai jumlah pergeseran kendaraan yang dilakukan sejak awal permainan. Karena UCS tidak menggunakan heuristik, f(n) hanya mempertimbangkan cost aktual tanpa prediksi ke depan.
2. Completeness, UCS selalu complete, artinya akan selalu menemukan solusi jika ada, karena secara sistematis mengeksplorasi semua kemungkinan berdasarkan urutan cost terkecil.
3. Optimality, UCS selalu optimal jika semua langkah memiliki bobot yang sama (seperti dalam Rush Hour), karena ia selalu memilih simpul dengan cost terkecil terlebih dahulu.
4. Time Complexity O(b^d), di mana b adalah branching factor (rata-rata jumlah langkah per simpul), d adalah kedalaman solusi. Jika banyak kendaraan yang bisa bergerak, nilai b menjadi besar.
5. Space Complexity O(b^d), karena menyimpan semua simpul dalam frontier.
6. UCS tidak sama dengan BFS dalam Rush Hour. Walaupun pada Rush Hour setiap langkah memiliki cost yang sama (1 langkah), UCS dan BFS bisa saja menghasilkan jalur yang sama dalam hal panjang path. Namun, urutan simpul yang diekspansi bisa berbeda, karena UCS mengevaluasi berdasarkan g(n) sedangkan BFS berdasarkan kedalaman level saja.
7. Kelebihan dan kekurangan. Kelebihannya adalah mudah diimplementasikan, menjamin solusi optimal. Kekurangan adalah memakan memori dan waktu yang besar pada pencarian dengan banyak simpul.

2.2 Greedy Best-First Search (GBFS)

2.2.1 Penjelasan Algoritma GBFS

Greedy Best First Search merupakan salah satu algoritma dalam perencanaan rute (route planning) yang memanfaatkan informasi tambahan

mengenai posisi tujuan pencarian, sehingga termasuk ke dalam kelompok Informed Search. Proses pencarian dalam algoritma ini didasarkan pada nilai **heuristik** dari sebuah simpul, yang biasanya dilambangkan dengan h(n). Nilai heuristik ini merepresentasikan estimasi jarak atau biaya dari suatu simpul n menuju simpul tujuan. Contohnya, dalam konteks pencarian jarak antar lokasi, nilai heuristik dapat berupa jarak garis lurus (Euclidean) antara posisi saat ini dan tujuan akhir.

Prinsip utama dari algoritma ini adalah memilih dan mengembangkan simpul yang memiliki nilai heuristik paling kecil terlebih dahulu, karena diasumsikan simpul tersebut paling dekat dengan solusi. Oleh karena itu, struktur data yang umum digunakan dalam implementasinya adalah Priority Queue, di mana simpul dengan nilai heuristik terendah akan berada di urutan paling depan dan dieksekusi terlebih dahulu.

Karena prioritas hanya ditentukan berdasarkan estimasi jarak ke tujuan (tanpa mempertimbangkan biaya dari simpul awal), maka fungsi evaluasi dalam algoritma ini dirumuskan sebagai:

$$f(n) = h(n)$$

dengan f(n) sebagai nilai evaluasi yang menentukan prioritas simpul, dan h(n) sebagai nilai heuristik atau estimasi ongkos dari simpul tersebut menuju tujuan akhir.

2.2.2 Analisis Algoritma GBFS

Langkah

1. Buat sebuah priority queue yang menyimpan simpul-simpul berdasarkan nilai heuristik h(n), yaitu estimasi jarak atau biaya dari simpul saat ini ke simpul tujuan.
2. Hitung nilai heuristik h(n) untuk simpul awal dan tambahkan simpul awal ke dalam priority queue berdasarkan nilai tersebut.
3. Siapkan sebuah struktur data tambahan seperti set untuk mencatat simpul-simpul yang telah dikunjungi agar tidak terjadi kunjungan berulang.

4. Selama antrian tidak kosong:
    - Ambil simpul dengan nilai h(n) paling kecil dari priority queue, karena diasumsikan simpul tersebut paling dekat dengan tujuan.
    - Periksa apakah simpul tersebut adalah simpul tujuan. Jika benar, pencarian selesai dan solusi dikembalikan.
    - Tandai simpul sebagai telah dikunjungi.
    - Bangkitkan semua simpul tetangga yang bisa dicapai dari simpul saat ini dengan memproses semua gerakan yang memungkinkan.
    - Untuk setiap simpul tetangga yang dihasilkan.
    - Hitung nilai heuristik h(n) dari simpul tetangga menggunakan fungsi estimasi yang sesuai.
    - Jika simpul tetangga belum pernah dikunjungi, tambahkan ke priority queue dengan nilai h(n) sebagai prioritasnya agar dapat dieksekusi pada iterasi selanjutnya.
5. Jika pencarian telah mengunjungi semua simpul yang mungkin dan tidak pernah mencapai simpul tujuan, maka solusi tidak tersedia dari kondisi awal yang diberikan.

Analisis

1. Definisi f(n) dan g(n) . f(n) = h(n) tidak menggunakan g(n) sama sekali. h(n) adalah nilai heuristik yang memperkirakan jarak dari simpul n ke simpul tujuan. Dalam Rush Hour, contoh heuristiknya adalah jumlah langkah minimum yang dibutuhkan primary piece untuk mencapai pintu keluar tanpa mempertimbangkan hambatan.
2. Completeness, tidak selalu complete. Jika heuristiknya menyesatkan atau ruang pencarian sangat besar, pencarian bisa gagal menemukan solusi.

3. Optimality, tidak optimal. GBFS hanya mengejar simpul yang "tampak paling dekat ke tujuan" berdasarkan h(n), tanpa memperhatikan apakah jalur tersebut mahal dari segi langkah atau cost.
4. Time Complexity O(b^d) dalam kasus terburuk, tapi dalam praktik bisa lebih cepat dibanding UCS jika heuristiknya sangat akurat.
5. Space Complexity O(b^d), karena tetap menyimpan banyak simpul dalam frontier.
6. GBFS tidak menjamin solusi optimal untuk Rush Hour. Karena hanya mengandalkan estimasi ke tujuan dan tidak mempertimbangkan cost dari awal, solusi yang ditemukan bisa saja bukan yang terpendek.
7. Kelebihan dan kekurangan. Kelebihannya adalah cepat dalam banyak kasus, cocok jika hanya ingin solusi cepat tanpa perlu optimal. Kekurangannya adalah tidak menjamin solusi optimal, sangat bergantung pada kualitas heuristik.

2.3 A* (A Star) Search (A*S)

2.3.1 Penjelasan Algoritma A*S

Algoritma A* (dibaca A-star) merupakan salah satu metode dalam penentuan rute atau route planning yang termasuk ke dalam kategori Informed Search, karena menggunakan informasi tambahan mengenai lokasi tujuan dalam proses pencariannya. Algoritma ini merupakan pengembangan dari Uniform Cost Search dan Greedy Best First Search, dengan tujuan untuk menggabungkan kelebihan keduanya. Ide utamanya adalah menghindari pengembangan simpul atau jalur yang memiliki biaya tinggi, sekaligus tetap mempertimbangkan estimasi jarak ke tujuan melalui nilai heuristik.

Pencarian solusi dalam algoritma A* dilakukan dengan memanfaatkan fungsi evaluasi yang dilambangkan dengan f(n). Fungsi ini merupakan hasil penjumlahan antara g(n), yaitu biaya aktual yang telah dikeluarkan untuk mencapai simpul tersebut dari simpul awal, dan h(n), yaitu estimasi biaya dari simpul tersebut menuju simpul tujuan. Simpul yang memiliki nilai f(n) paling

kecil akan diprioritaskan untuk dikembangkan terlebih dahulu, karena dianggap memiliki potensi terbesar untuk mencapai solusi secara optimal. Oleh karena itu, struktur data yang umum digunakan untuk mengatur urutan simpul adalah Priority Queue, yang akan menempatkan simpul dengan nilai f(n) terendah di urutan paling depan.

Fungsi evaluasi dalam algoritma A* dirumuskan sebagai berikut:

$$f(n) = g(n) + h(n)$$

dengan f(n) sebagai nilai evaluasi total suatu simpul, g(n) sebagai biaya nyata dari simpul awal menuju simpul tersebut, dan h(n) sebagai estimasi biaya dari simpul tersebut menuju tujuan akhir berdasarkan heuristik.

Salah satu konsep penting dalam algoritma A* adalah heuristik yang bersifat admissible, yaitu fungsi heuristik h(n) selalu kurang dari atau sama dengan biaya sebenarnya dari simpul tersebut ke tujuan. Jika heuristik yang digunakan memenuhi sifat admissible, maka algoritma A* dapat menjamin bahwa solusi yang ditemukan adalah solusi yang paling optimal secara biaya.

2.3.2 Analisis Algoritma A*S

Langkah

1. Buat sebuah priority queue yang menyimpan simpul-simpul berdasarkan nilai f(n), yaitu fungsi evaluasi yang merupakan penjumlahan dari g(n) dan h(n). Di mana g(n) adalah biaya dari simpul awal ke simpul saat ini, dan h(n) adalah estimasi biaya dari simpul tersebut ke simpul tujuan.
2. Hitung nilai g(n) dan h(n) untuk simpul awal, lalu tambahkan simpul awal ke priority queue dengan f(n) = g(n) + h(n).
3. Siapkan sebuah struktur data seperti set untuk mencatat simpul-simpul yang telah dikunjungi agar tidak diproses dua kali.
4. Selama priority queue tidak kosong:
    - Ambil simpul dengan nilai f(n) terkecil dari antrian karena dianggap sebagai kandidat terbaik menuju solusi.

- Periksa apakah simpul tersebut merupakan simpul tujuan. Jika iya, kembalikan jalur yang telah ditemukan sebagai solusi akhir.
- Tandai simpul tersebut sebagai telah dikunjungi.
- Bangkitkan semua simpul tetangga yang dapat dicapai dari simpul tersebut melalui gerakan yang valid.
- Untuk setiap simpul tetangga yang dihasilkan, hitung nilai g(n) sebagai biaya total dari simpul awal ke simpul tetangga tersebut.Hitung nilai h(n) sebagai estimasi biaya dari simpul tetangga ke tujuan.Hitung nilai f(n) sebagai hasil penjumlahan dari g(n) dan h(n).Jika simpul tetangga belum pernah dikunjungi, atau ditemukan jalur baru dengan f(n) yang lebih kecil dari sebelumnya, maka tambahkan atau perbarui simpul tersebut dalam priority queue.

5. Jika semua simpul yang memungkinkan telah diproses dan tidak ditemukan jalur menuju simpul tujuan, maka algoritma menyimpulkan bahwa tidak ada solusi dari konfigurasi awal yang diberikan.

Analisis

1. Definisi f(n) dan g(n), f(n) = g(n) + h(n). g(n) adalah biaya aktual dari simpul awal ke simpul n. h(n) adalah estimasi biaya dari simpul n ke tujuan. Dalam Rush Hour, dapat berupa estimasi langkah tersisa menuju pintu keluar.
2. Completeness A* adalah algoritma yang complete, selama heuristik yang digunakan tidak negatif.
3. Optimality A* optimal jika dan hanya jika heuristik h(n) adalah admissible.
4. Jika h(n) tidak pernah melebihi biaya sebenarnya dari simpul n ke tujuan, maka heuristik disebut admissible. Contohnya, jika h(n) hanya menghitung jumlah langkah lurus ke pintu keluar tanpa memperhitungkan halangan, maka h(n) adalah admissible karena selalu kurang atau sama dengan biaya sebenarnya.

5. Time Complexity O(b^d) pada kasus terburuk, namun umumnya lebih cepat dari UCS karena menggunakan heuristik untuk mempersempit ruang pencarian.
6. Space Complexity O(b^d), karena menyimpan banyak simpul dan informasi tambahan (seperti cost, parent).
7. Secara teoritis, A* lebih efisien dari UCS dalam Rush Hour. Dengan heuristik yang baik, A* akan memprioritaskan jalur yang tidak hanya murah dari awal, tapi juga dekat ke tujuan, sehingga mengurangi jumlah simpul yang perlu dieksplorasi.
8. Kelebihan dan kekurangan. Kelebihannya adalah menggabungkan kelebihan UCS dan GBFS, cepat dan optimal jika heuristik admissible. Kekurangannya adalah konsumsi memori bisa tinggi, performa sangat tergantung kualitas heuristik.

2.4 Iterative Deepening A* (A Star) Search (IDA*S)

2.4.1 Penjelasan Algoritma IDA*S

Algoritma Iterative Deepening A* atau IDA* merupakan varian dari algoritma A* yang dirancang untuk mengatasi keterbatasan memori saat melakukan pencarian jalur dalam ruang pencarian yang besar. Algoritma ini menggabungkan keunggulan dari algoritma pencarian mendalam terbatas (Iterative Deepening Depth First Search) dengan prinsip evaluasi biaya dari A*. Sama seperti A*, algoritma ini menggunakan fungsi evaluasi f(n) yang terdiri dari dua komponen, yaitu g(n) sebagai biaya aktual dari simpul awal ke simpul n, dan h(n) sebagai estimasi biaya dari simpul n ke simpul tujuan. Nilai f(n) = g(n) + h(n) digunakan sebagai batas untuk menentukan apakah sebuah simpul layak untuk dikembangkan pada iterasi tertentu.

Proses pencarian IDA* dilakukan dalam beberapa iterasi. Setiap iterasi menggunakan batas nilai f(n) tertentu, dan hanya akan mengembangkan simpul-simpul yang memiliki nilai f(n) kurang dari atau sama dengan batas tersebut. Jika pada suatu iterasi solusi belum ditemukan,

maka algoritma akan meningkatkan batas pencarian berdasarkan nilai f(n) terkecil yang melebihi batas sebelumnya dan melakukan pencarian ulang. Dengan demikian, IDA* melakukan eksplorasi ruang pencarian secara mendalam seperti Depth First Search, namun tetap mempertahankan keoptimalan solusi seperti A*.

Keunggulan utama dari IDA* terletak pada efisiensi penggunaan memorinya. Karena algoritma ini menggunakan pendekatan rekursif seperti DFS, ia tidak membutuhkan struktur data seperti Priority Queue atau penyimpanan semua simpul terbuka secara eksplisit, sehingga konsumsi memorinya jauh lebih kecil dibandingkan A*. Meskipun waktu pencariannya dapat menjadi lebih lambat akibat pengulangan eksplorasi simpul, IDA* sangat cocok digunakan untuk permasalahan besar di mana efisiensi memori menjadi pertimbangan utama. Untuk menjamin keoptimalan solusi, heuristik yang digunakan dalam IDA* juga harus memenuhi sifat admissible seperti pada algoritma A*.

2.4.2 Analisis Algoritma IDA*S

Langkah

1. Hitung nilai f(n) awal dengan menggunakan fungsi evaluasi f(n) = g(n) + h(n), di mana g(n) adalah biaya dari simpul awal ke simpul saat ini, dan h(n) adalah estimasi biaya dari simpul tersebut ke tujuan berdasarkan fungsi heuristik.
2. Inisialisasi batas awal pencarian atau threshold dengan nilai f(n) dari simpul awal.
3. Lakukan pencarian secara rekursif menggunakan pendekatan depth-first search, tetapi hanya menjelajahi simpul yang memiliki nilai f(n) kurang dari atau sama dengan nilai threshold saat ini.
4. Pada setiap simpul yang dikunjungi dalam pencarian:
    - Hitung nilai f(n) dari simpul tersebut.

- Jika nilai f(n) melebihi nilai threshold saat ini, jangan lanjutkan eksplorasi dari simpul ini. Simpan nilai f(n) tersebut sebagai kandidat batas threshold berikutnya.
- Jika simpul merupakan simpul tujuan, maka pencarian selesai dan solusi dikembalikan.
- Jika bukan simpul tujuan dan f(n) masih dalam batas threshold, bangkitkan semua simpul tetangga dari simpul saat ini yang merupakan hasil dari langkah-langkah valid pada permainan.
- Untuk setiap simpul tetangga, hitung g(n) baru dengan menambahkan 1 ke biaya langkah dari parent-nya. Hitung h(n) baru berdasarkan heuristik.Hitung f(n) sebagai penjumlahan dari g(n) dan h(n).Lanjutkan pencarian rekursif terhadap simpul tetangga tersebut, selama nilai f(n) tidak melebihi threshold saat ini.

5. Jika semua simpul pada iterasi tersebut telah diperiksa dan tidak ada solusi ditemukan, perbarui threshold menjadi nilai f(n) terkecil yang melebihi batas threshold sebelumnya.
6. Ulangi proses pencarian dari simpul awal menggunakan threshold baru yang diperoleh.
7. Lanjutkan iterasi ini sampai:
    - Simpul tujuan ditemukan dan solusi dikembalikan, atau
    - Tidak ada lagi simpul yang dapat dikunjungi dan semua nilai f(n) melebihi threshold tanpa menghasilkan solusi, yang berarti tidak ada jalur ke simpul tujuan.

Analisis

1. Definisi f(n) dan g(n), IDA* menggunakan fungsi evaluasi f(n) = g(n) + h(n), di mana g(n) adalah biaya aktual dari simpul awal ke simpul n, dan h(n) adalah estimasi biaya dari simpul n ke tujuan. Fungsi ini digunakan sebagai batas pencarian yang akan dinaikkan secara bertahap.

2. Completeness, algoritma ini bersifat complete, artinya akan selalu menemukan solusi jika solusi memang tersedia dalam ruang pencarian.
3. Optimality, IDA* menjamin solusi optimal selama heuristik yang digunakan bersifat admissible, yaitu tidak pernah melebihi biaya minimum sebenarnya dari simpul ke tujuan.
4. IDA* memiliki heuristik admissible, jika menggunakan heuristik seperti jarak lurus minimum primary piece ke pintu keluar tanpa mempertimbangkan hambatan, maka nilai h(n) tidak melebihi biaya sebenarnya, sehingga heuristik tersebut admissible.
5. Time Complexity O(b^d), dengan b sebagai branching factor dan d sebagai kedalaman solusi. Karena menggunakan iterasi dengan threshold yang terus naik, simpul bisa dikunjungi berulang kali, menyebabkan waktu eksekusi lebih lama.
6. Space Complexity O(d), karena IDA* menggunakan pendekatan depth-first dan hanya menyimpan jalur saat ini, sehingga sangat efisien secara memori.
7. Efisiensi dibandingkan A* dalam Rush Hour, IDA* lebih hemat memori dibanding A*, namun bisa lebih lambat secara waktu karena pencarian berulang dalam setiap iterasi. Efektif untuk ruang pencarian besar dengan batasan memori.
8. Kelebihan dan kekurangan, kelebihannya adalah Hemat memori karena hanya menyimpan jalur aktif. Tetap menjamin solusi optimal dengan heuristik admissible. Kekurangannya adalah Waktu eksekusi bisa lebih lama karena simpul dikunjungi ulang. Sensitif terhadap kenaikan threshold, apalagi jika banyak nilai f(n) yang melampaui sedikit demi sedikit

Bab III

Source Program

3.1 Struktur Direktori dan File

```
├── bin
├── src
│   ├── algorithm
│   │   ├── AS.java
│   │   ├── GBFS.java
│   │   ├── IDAS.java
│   │   ├── PathFinder.java
│   │   └── UCS.java
│   ├── core
│   │   ├── Board.java
│   │   ├── FileParser.java
│   │   ├── GameState.java
│   │   ├── Move.java
│   │   └── Piece.java
│   ├── heuristic
│   │   ├── BP.java
│   │   ├── DB.java
│   │   ├── Heuristic.java
│   │   └── MD.java
│   ├── ui
│   │   ├── Animation.java
│   │   ├── BoardPanel.java
│   │   ├── CLI.java
│   │   └── GUI.java
│   └── Main.java
└── test
```

```
├── README.md
```

3.2 Source Code

3.2.1 Algorithm

PathFinder.java

```
package algorithm;

import core.GameState;
import heuristic.Heuristic;

/**
 * Abstract base for pathfinders.
 */
public abstract class PathFinder {
  protected final Heuristic heuristic;
  protected int nodesVisited;
  protected long executionTimeMillis;

  public PathFinder(Heuristic heuristic) {
    this.heuristic = heuristic;
  }

  // Find path to solution.
  public abstract GameState findPath(GameState initialState);

  // Get algorithm name.
  public abstract String getName();

  // Get nodes visited.
  public int getNodesVisited() {
    return nodesVisited;
```

```
  }

  // Get execution time (ms).
  public long getExecutionTime() {
    return executionTimeMillis;
  }

  // Start timing search.
  protected void startTimer() {
    this.nodesVisited = 0;
    this.executionTimeMillis = System.currentTimeMillis();
  }

  // Stop timing search.
  protected void stopTimer() {
    this.executionTimeMillis = System.currentTimeMillis() - this.executionTimeMillis;
  }
}
```

AS.java

```
package algorithm;

import core.GameState;
import core.Move;
import heuristic.Heuristic;
import java.util.Comparator;
import java.util.HashSet;
import java.util.PriorityQueue;
import java.util.Set;

/**
 * A* Search algorithm.
```

```
 * Uses g(n) + h(n).
 */
public class AS extends PathFinder {

  public AS(Heuristic heuristic) {
    super(heuristic);
  }

  @Override
  public GameState findPath(GameState initialState) {
    startTimer();

    PriorityQueue<GameState> openSet = new PriorityQueue<>(
      Comparator.comparingInt(s -> s.getCost() + heuristic.evaluate(s))
    );
    Set<String> closedSet = new HashSet<>(); // Visited board configurations

    openSet.add(initialState);

    while (!openSet.isEmpty()) {
      GameState current = openSet.poll();
      nodesVisited++;

      if (current.getBoard().isWin()) {
        stopTimer();
        return current; // Solution found
      }

      String boardKey = current.getBoard().toString();
      if (closedSet.contains(boardKey)) continue;
      closedSet.add(boardKey);

      for (Move move : current.getPossibleMoves()) {
```

```
        GameState nextState = current.applyMove(move);
        if (!closedSet.contains(nextState.getBoard().toString())) {
          openSet.add(nextState);
        }
      }
    }
    stopTimer();
    return null; // No solution
  }

  @Override
  public String getName() {
    return "A* Search";
  }
}
```

GBFS.java

```
package algorithm;

import core.GameState;
import core.Move;
import heuristic.Heuristic;
import java.util.Comparator;
import java.util.HashSet;
import java.util.PriorityQueue;
import java.util.Set;

/**
 * Greedy Best-First Search.
 * Uses h(n) only.
 */
```

```
public class GBFS extends PathFinder {

  public GBFS(Heuristic heuristic) {
    super(heuristic);
  }

  @Override
  public GameState findPath(GameState initialState) {
    startTimer();

    PriorityQueue<GameState> openSet = new PriorityQueue<>(
      Comparator.comparingInt(heuristic::evaluate)
    );
    Set<String> closedSet = new HashSet<>();

    openSet.add(initialState);

    while (!openSet.isEmpty()) {
      GameState current = openSet.poll();
      nodesVisited++;

      if (current.getBoard().isWin()) {
        stopTimer();
        return current;
      }

      String boardKey = current.getBoard().toString();
      if (closedSet.contains(boardKey)) continue;
      closedSet.add(boardKey);

      for (Move move : current.getPossibleMoves()) {
        GameState nextState = current.applyMove(move);
        if (!closedSet.contains(nextState.getBoard().toString())) {
```

```
            openSet.add(nextState);
          }
        }
      }
      stopTimer();
      return null;
    }

    @Override
    public String getName() {
      return "Greedy Best-First Search";
    }
}
```

IDAS.java

```
package algorithm;

import core.GameState;
import core.Move;
import heuristic.Heuristic;

/**
 * Iterative Deepening A*.
 * Depth-first with threshold.
 */
public class IDAS extends PathFinder {
  private GameState solutionState;

  public IDAS(Heuristic heuristic) {
    super(heuristic);
  }
```

```
  @Override
  public GameState findPath(GameState initialState) {
    startTimer();
    solutionState = null;

    int threshold = heuristic.evaluate(initialState);

    while (solutionState == null) {
      SearchResult result = searchRecursive(initialState, 0, threshold);
      nodesVisited += result.nodesThisIteration;

      if (result.status == SearchStatus.FOUND) {
        stopTimer();
        return solutionState;
      }
      if (result.status == SearchStatus.NOT_FOUND_EXHAUSTED) {
        stopTimer();
        return null;
      }
      threshold = result.nextThreshold;
    }
    stopTimer();
    return solutionState;
  }

        private SearchResult searchRecursive(GameState state, int gCost, int
currentThreshold) {
    int fCost = gCost + heuristic.evaluate(state);
    int nodesThisIteration = 1;

    if (fCost > currentThreshold) {
        return new SearchResult(SearchStatus.NOT_FOUND_WITHIN_THRESHOLD,
fCost, nodesThisIteration);
```

```
    }

    if (state.getBoard().isWin()) {
      solutionState = state;
      return new SearchResult(SearchStatus.FOUND, fCost, nodesThisIteration);
    }

    int minNextThreshold = Integer.MAX_VALUE;

    for (Move move : state.getPossibleMoves()) {
      GameState nextState = state.applyMove(move);
                                                if (state.getParent() != null &&
nextState.getBoard().toString().equals(state.getParent().getBoard().toString())) {
        continue;
      }

                    SearchResult result = searchRecursive(nextState, gCost + 1,
currentThreshold);
      nodesThisIteration += result.nodesThisIteration;

      if (result.status == SearchStatus.FOUND) {
        return new SearchResult(SearchStatus.FOUND, fCost, nodesThisIteration);
      }
      if (result.nextThreshold < minNextThreshold) {
        minNextThreshold = result.nextThreshold;
      }
    }
       return new SearchResult(SearchStatus.NOT_FOUND_WITHIN_THRESHOLD,
minNextThreshold, nodesThisIteration);
  }

      private enum SearchStatus { FOUND, NOT_FOUND_WITHIN_THRESHOLD,
NOT_FOUND_EXHAUSTED }
```

```
  private static class SearchResult {
    final SearchStatus status;
    final int nextThreshold;
    final int nodesThisIteration;

    SearchResult(SearchStatus status, int nextThreshold, int nodesThisIteration) {
      this.status = status;
      this.nextThreshold = nextThreshold;
      this.nodesThisIteration = nodesThisIteration;
    }
  }

  @Override
  public String getName() {
    return "Iterative Deepening A*";
  }
}
```

UCS.java

```
package algorithm;

import core.GameState;
import core.Move;
import java.util.Comparator;
import java.util.HashSet;
import java.util.PriorityQueue;
import java.util.Set;

/**
 * Uniform Cost Search.
 * Finds shortest path.
```

```
 */
public class UCS {
  private int nodesVisited;
  private long executionTimeMillis;

  public UCS() {}

  public GameState solve(GameState initialState) {
    long startTime = System.currentTimeMillis();
    nodesVisited = 0;

                                    PriorityQueue<GameState>  frontier  =  new
PriorityQueue<>(Comparator.comparingInt(GameState::getCost));
    Set<String> visitedStates = new HashSet<>();
    frontier.add(initialState);

    while (!frontier.isEmpty()) {
      GameState current = frontier.poll();
      nodesVisited++;

      if (current.getBoard().isWin()) {
        executionTimeMillis = System.currentTimeMillis() - startTime;
        return current;
      }

      String currentBoardKey = current.getBoard().toString();
      if (visitedStates.contains(currentBoardKey)) continue;
      visitedStates.add(currentBoardKey);

      for (Move move : current.getPossibleMoves()) {
        GameState next = current.applyMove(move);
        if (!visitedStates.contains(next.getBoard().toString())) {
          frontier.add(next);
```

```
        }
      }
    }
    executionTimeMillis = System.currentTimeMillis() - startTime;
    return null;
  }

  // Get nodes visited.
  public int getNodesVisited() {
    return nodesVisited;
  }

  // Get execution time (ms).
  public long getExecutionTime() {
    return executionTimeMillis;
  }

  // Get algorithm name.
  public String getName() {
    return "Uniform Cost Search";
  }
}
```

3.2.2 Core

Board.java

```
package core;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
```

```
import java.util.List;
import java.util.Map;

/**
 * Represents the game board.
 */
public class Board {
  private final int rows;
  private final int cols;
  private final char[][] grid;
  private final List<Piece> pieces;
  private int exitRow;
  private int exitCol;
  private final boolean[] isSpaceOnlyRow;
  private final boolean[] isSpaceOnlyCol;

  public Board(int rows, int cols) {
    this.rows = rows;
    this.cols = cols;
    this.grid = new char[rows][cols];
    this.pieces = new ArrayList<>();
    this.exitRow = -1;
    this.exitCol = -1;
    this.isSpaceOnlyRow = new boolean[rows];
    this.isSpaceOnlyCol = new boolean[cols];

    for (char[] rowGrid : this.grid) {
      Arrays.fill(rowGrid, '.');
    }
  }

  // Set exit location.
  public void setExit(int r, int c) {
```

```
    this.exitRow = r;
    this.exitCol = c;
  }

  // Initialize from configuration.
  public void initialize(char[][] originalConfiguration) {
    Map<Character, List<int[]>> piecePositions = new HashMap<>();

    for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
        char c = originalConfiguration[i][j];
        if (c == ' ' || c == '\0') {
          this.grid[i][j] = '.';
        } else if (c != '.') {
          this.grid[i][j] = c;
           piecePositions.computeIfAbsent(c, _ -> new ArrayList<>()).add(new int[]{i,
j});
        } else {
          this.grid[i][j] = '.';
        }
      }
    }

    for (Map.Entry<Character, List<int[]>> entry : piecePositions.entrySet()) {
      char id = entry.getKey();
      List<int[]> positions = entry.getValue();
      if (positions.isEmpty()) continue;

      positions.sort((a, b) -> (a[0] != b[0]) ? a[0] - b[0] : a[1] - b[1]);

      int size = positions.size();
      int startRow = positions.get(0)[0];
      int startCol = positions.get(0)[1];
```

```
    boolean isHorizontal = (size > 1) && (positions.get(0)[0] == positions.get(1)[0]);

    pieces.add(new Piece(id, (id == 'P'), isHorizontal, size, startRow, startCol));
  }

  for (int i = 0; i < rows; i++) {
    isSpaceOnlyRow[i] = true;
    for (int j = 0; j < cols; j++) {
      if (originalConfiguration[i][j] != ' ') {
        isSpaceOnlyRow[i] = false;
        break;
      }
    }
  }
  for (int j = 0; j < cols; j++) {
    isSpaceOnlyCol[j] = true;
    for (int i = 0; i < rows; i++) {
      if (originalConfiguration[i][j] != ' ') {
        isSpaceOnlyCol[j] = false;
        break;
      }
    }
  }

  if (this.exitRow == -1 && this.exitCol == -1) {
    Piece primary = getPrimaryPiece();
    if (primary != null) {
      if (primary.isHorizontal()) {
        exitRow = primary.getRow();
        exitCol = (primary.getCol() + primary.getSize() / 2.0 > cols / 2.0) ? cols : -1;
      } else {
        exitCol = primary.getCol();
            exitRow = (primary.getRow() + primary.getSize() / 2.0 > rows / 2.0) ? rows :
```

```
-1;
        }
      }
    }
    updateGrid();
  }

      public boolean isRowSpaceOnly(int r) { return (r >= 0 && r < rows) &&
isSpaceOnlyRow[r]; }

      public boolean isColSpaceOnly(int c) { return (c >= 0 && c < cols) &&
isSpaceOnlyCol[c]; }

  // Get the primary piece.
  public Piece getPrimaryPiece() {
    for (Piece piece : pieces) {
      if (piece.isPrimary()) return piece;
    }
    return null;
  }

  // Get all pieces.
  public List<Piece> getPieces() {
    return new ArrayList<>(pieces);
  }

  public int getRows() { return rows; }
  public int getCols() { return cols; }
  public int getExitRow() { return exitRow; }
  public int getExitCol() { return exitCol; }

  // Update logical grid.
  public void updateGrid() {
```

```
    for (char[] rowGrid : this.grid) {
      Arrays.fill(rowGrid, '.');
    }
    for (Piece piece : pieces) {
      int r = piece.getRow();
      int c = piece.getCol();
      for (int s = 0; s < piece.getSize(); s++) {
        if (piece.isHorizontal()) {
                    if (c + s >= 0 && c + s < cols && r >=0 && r < rows) grid[r][c + s] =
piece.getId();
        } else {
                    if (r + s >= 0 && r + s < rows && c >=0 && c < cols) grid[r + s][c] =
piece.getId();
        }
      }
    }
  }

  // Get current grid state.
  public char[][] getGrid() {
    char[][] currentGrid = new char[rows][cols];
    for(int i=0; i<rows; i++) {
      System.arraycopy(this.grid[i], 0, currentGrid[i], 0, cols);
    }
    return currentGrid;
  }

  // Create a deep copy.
  public Board copy() {
    Board newBoard = new Board(rows, cols);
    newBoard.exitRow = this.exitRow;
    newBoard.exitCol = this.exitCol;
```

```
    for (Piece piece : this.pieces) {
      newBoard.pieces.add(piece.copy());
    }

    for(int i=0; i<rows; i++) {
      System.arraycopy(this.grid[i], 0, newBoard.grid[i], 0, cols);
    }
        System.arraycopy(this.isSpaceOnlyRow, 0, newBoard.isSpaceOnlyRow, 0,
rows);
         System.arraycopy(this.isSpaceOnlyCol, 0, newBoard.isSpaceOnlyCol, 0,
cols);

    return newBoard;
  }

  // Check win condition.
  public boolean isWin() {
    Piece primary = getPrimaryPiece();
    if (primary == null) return false;

    int pRow = primary.getRow();
    int pCol = primary.getCol();
    int pSize = primary.getSize();

    // For horizontal pieces with exit on the right
    if (primary.isHorizontal() && exitCol == cols) {
      if (pRow != exitRow) return false;

      return pCol + pSize - 1 == cols - 1;
    }

    // For horizontal pieces with exit on the left
    if (primary.isHorizontal() && exitCol == -1) {
```

```
    if (pRow != exitRow) return false;

    return pCol == 0;
  }

  // For vertical pieces with exit at the bottom
  if (!primary.isHorizontal() && exitRow == rows) {
    if (pCol != exitCol) return false;

    return pRow + pSize - 1 == rows - 1;
  }

  // For vertical pieces with exit at the top
  if (!primary.isHorizontal() && exitRow == -1) {
    if (pCol != exitCol) return false;

    return pRow == 0;
  }

  // For exits within the grid (less common case)
  if (exitRow >= 0 && exitRow < rows && exitCol >= 0 && exitCol < cols) {
    if (primary.isHorizontal()) {
      return pRow == exitRow && pCol <= exitCol && pCol + pSize - 1 >= exitCol;
    } else {
      return pCol == exitCol && pRow <= exitRow && pRow + pSize - 1 >= exitRow;
    }
  }

  return false;
}

@Override
public String toString() {
```

```
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
        sb.append(grid[i][j]);
      }
      if (i < rows - 1) sb.append("\n");
    }
    return sb.toString();
  }
}
```

FileParser.java

```
package core;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

/**
 * Parses puzzle input files.
 */
public class FileParser {
  private final String filePath;

  public FileParser(String filePath) {
    this.filePath = filePath;
  }

  // Parse the puzzle file.
```

```
public Board parseFile() throws IOException {
  List<String> lines = readAllLines();

  int lineIdx = 0;
  String dimLine = getNextNonCommentLine(lines, lineIdx);
  lineIdx = lines.indexOf(dimLine) + 1;
  String[] dimensions = dimLine.split(" ");
  int declaredRows = Integer.parseInt(dimensions[0]);
  int declaredCols = Integer.parseInt(dimensions[1]);

  String pieceCountLine = getNextNonCommentLine(lines, lineIdx);
  lineIdx = lines.indexOf(pieceCountLine) + 1;

  // Read potential grid lines
  List<String> potentialGridLines = new ArrayList<>();
  for (int i = lineIdx; i < lines.size(); i++) {
    if (!lines.get(i).trim().startsWith("//") && !lines.get(i).trim().isEmpty()) {
      potentialGridLines.add(lines.get(i));
    }
  }

  // Find exit positions and actual grid content
  int topExitCol = -1;
  int bottomExitCol = -1;
  boolean leftExitFound = false;
  boolean rightExitFound = false;
  List<String> actualGridLines = new ArrayList<>();

  for (int r = 0; r < potentialGridLines.size(); r++) {
    String line = potentialGridLines.get(r);
    String trimmedLine = line.trim();

    // Check for standalone 'K' line
```

```
    if (trimmedLine.equals("K")) {
      int kCol = line.indexOf('K');
      if (actualGridLines.isEmpty()) {
        // K is above the grid
        topExitCol = kCol;
      } else {
        // K is below the grid
        bottomExitCol = kCol;
      }
      continue;
    }

    actualGridLines.add(line);
  }

  // Create board and configuration array
  Board board = new Board(declaredRows, declaredCols);
  char[][] configuration = new char[declaredRows][declaredCols];
  for (char[] row : configuration) {
    Arrays.fill(row, '.');
  }

  // Process grid lines to fill configuration
  for (int r = 0; r < Math.min(actualGridLines.size(), declaredRows); r++) {
    String line = actualGridLines.get(r);

    // Check for K at beginning of line
    if (line.startsWith("K")) {
      board.setExit(r, -1);
      leftExitFound = true;
      line = line.substring(1);
    }
```

```
    // Skip leading spaces
    int gridColIndex = 0;
    for (int c = 0; c < line.length() && gridColIndex < declaredCols; c++) {
      char currentChar = line.charAt(c);

      // Skip leading spaces
      if (currentChar == ' ') {
        continue;
      }

      // Handle K within grid (unusual case)
      if (currentChar == 'K') {
        board.setExit(r, gridColIndex);
      } else {
        configuration[r][gridColIndex] = currentChar;
      }

      gridColIndex++;
    }

    // Check for K at end of lin
    if (line.endsWith("K") && !line.trim().equals("K")) {
      board.setExit(r, declaredCols);
      rightExitFound = true;
    }
  }

  // Set top/bottom exits if detected
  if (!leftExitFound && !rightExitFound) {
    if (topExitCol != -1) {
      board.setExit(-1, topExitCol);
    } else if (bottomExitCol != -1) {
      board.setExit(declaredRows, bottomExitCol);
```

```
      }
    }


    board.initialize(configuration);
    return board;
  }

  // Read all lines from file.
  private List<String> readAllLines() throws IOException {
    List<String> lines = new ArrayList<>();
    try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
      String line;
      while ((line = reader.readLine()) != null) {
        lines.add(line);
      }
    }
    return lines;
  }

  // Get next non-comment/empty line.
  private String getNextNonCommentLine(List<String> lines, int startIndex) throws
IOException {
    for (int i = startIndex; i < lines.size(); i++) {
      String line = lines.get(i).trim();
      if (!line.startsWith("//") && !line.isEmpty()) {
        return line;
      }
    }
    throw new IOException("Expected more content in file.");
  }
}
```

GameState.java

```
package core;

import java.util.ArrayList;
import java.util.List;
import java.util.Objects;

/**
 * Represents a game state.
 */
public class GameState {
  private final Board board;
  private final GameState parent;
  private final Move lastMove;
  private final int cost;

  public GameState(Board board) {
    this(board, null, null, 0);
  }

  public GameState(Board board, GameState parent, Move lastMove) {
    this(board, parent, lastMove, (parent != null ? parent.getCost() + 1 : 0));
  }

  private GameState(Board board, GameState parent, Move lastMove, int cost) {
    this.board = board;
    this.parent = parent;
    this.lastMove = lastMove;
    this.cost = cost;
  }

  public Board getBoard() { return board; }
  public GameState getParent() { return parent; }
```

```
  public Move getLastMove() { return lastMove; }
  public int getCost() { return cost; }

  public List<Move> getPossibleMoves() {
    List<Move> moves = new ArrayList<>();
    char[][] grid = board.getGrid(); // Use a copy
    int rows = board.getRows();
    int cols = board.getCols();

    for (Piece piece : board.getPieces()) {
      int r = piece.getRow();
      int c = piece.getCol();
      int size = piece.getSize();

      if (piece.isHorizontal()) {
        // Check LEFT (within grid)
        int maxStepsLeftInGrid = 0;
        for (int i = c - 1; i >= 0 && grid[r][i] == '.'; i--) maxStepsLeftInGrid++;
                if (maxStepsLeftInGrid > 0) moves.add(new Move(piece, Move.LEFT,
maxStepsLeftInGrid));

        // Check LEFT (to exit at col = -1)
        if (c == 0 && board.getExitCol() == -1 && board.getExitRow() == r) {
          moves.add(new Move(piece, Move.LEFT, 1));
        }

        // Check RIGHT (within grid)
        int maxStepsRightInGrid = 0;
        for (int i = c + size; i < cols && grid[r][i] == '.'; i++) maxStepsRightInGrid++;
              if (maxStepsRightInGrid > 0) moves.add(new Move(piece, Move.RIGHT,
maxStepsRightInGrid));

        // Check RIGHT (to exit at col = cols)
```

```
                    if ((c + size -1) == (cols - 1) && board.getExitCol() == cols &&
board.getExitRow() == r){
          moves.add(new Move(piece, Move.RIGHT, 1));
        }

      } else { // Vertical
        // Check UP (within grid)
        int maxStepsUpInGrid = 0;
        for (int i = r - 1; i >= 0 && grid[i][c] == '.'; i--) maxStepsUpInGrid++;
                    if (maxStepsUpInGrid > 0) moves.add(new Move(piece, Move.UP,
maxStepsUpInGrid));

        // Check UP (to exit at row = -1)
        if (r == 0 && board.getExitRow() == -1 && board.getExitCol() == c) {
          moves.add(new Move(piece, Move.UP, 1));
        }

        // Check DOWN (within grid)
        int maxStepsDownInGrid = 0;
        for (int i = r + size; i < rows && grid[i][c] == '.'; i++) maxStepsDownInGrid++;
            if (maxStepsDownInGrid > 0) moves.add(new Move(piece, Move.DOWN,
maxStepsDownInGrid));

        // Check DOWN (to exit at row = rows)
                    if ((r + size - 1) == (rows - 1) && board.getExitRow() == rows &&
board.getExitCol() == c) {
          moves.add(new Move(piece, Move.DOWN, 1));
        }
      }
    }
    return moves;
  }
```

```
  // Apply move, return new state.
  public GameState applyMove(Move move) {
    Board newBoard = board.copy();
    Piece movingPiece = null;
    for (Piece p : newBoard.getPieces()) {
      if (p.getId() == move.getPiece().getId()) {
        movingPiece = p;
        break;
      }
    }

    if (movingPiece != null) {
      switch (move.getDirection()) {
                  case Move.UP    -> movingPiece.setRow(movingPiece.getRow() -
move.getSteps());
                   case Move.RIGHT -> movingPiece.setCol(movingPiece.getCol() +
move.getSteps());
                case Move.DOWN  -> movingPiece.setRow(movingPiece.getRow() +
move.getSteps());
                    case Move.LEFT  -> movingPiece.setCol(movingPiece.getCol() -
move.getSteps());
      }
      newBoard.updateGrid();
    }
    return new GameState(newBoard, this, move);
  }

  // Reconstruct solution path.
  public List<GameState> getSolutionPath() {
    List<GameState> path = new ArrayList<>();
    GameState current = this;
    while (current != null) {
      path.add(0, current);
```

```
      current = current.getParent();
    }
    return path;
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    GameState gameState = (GameState) o;
    return Objects.equals(board.toString(), gameState.board.toString());
  }

  @Override
  public int hashCode() {
    return Objects.hash(board.toString());
  }
}
```

Move.java

```
package core;

/**
 * Represents a piece movement.
 */
public class Move {
  public static final int UP = 0;
  public static final int RIGHT = 1;
  public static final int DOWN = 2;
  public static final int LEFT = 3;

  private final Piece piece;
```

```
private final int direction;
private final int steps;

public Move(Piece piece, int direction, int steps) {
  this.piece = piece;
  this.direction = direction;
  this.steps = steps;
}

// Get moved piece.
public Piece getPiece() {
  return piece;
}

// Get move direction.
public int getDirection() {
  return direction;
}

// Get move steps.
public int getSteps() {
  return steps;
}

// Get direction as string.
public String getDirectionString() {
  return switch (direction) {
    case UP -> "atas";
    case RIGHT -> "kanan";
    case DOWN -> "bawah";
    case LEFT -> "kiri";
    default -> "unknown";
  };
```

```
  }

  @Override
  public String toString() {
    return piece.getId() + "-" + getDirectionString();
  }
}
```

Piece.java

```
package core;

/**
 * Represents a vehicle piece.
 */
public class Piece {
  private final char id;
  private final boolean isPrimary;
  private final boolean isHorizontal;
  private final int size;
  private int row;
  private int col;

   public Piece(char id, boolean isPrimary, boolean isHorizontal, int size, int row, int
col) {
    this.id = id;
    this.isPrimary = isPrimary;
    this.isHorizontal = isHorizontal;
    this.size = size;
    this.row = row;
    this.col = col;
  }
```

```
// Get piece identifier.
public char getId() {
  return id;
}

// Is primary piece?
public boolean isPrimary() {
  return isPrimary;
}

// Is piece horizontal?
public boolean isHorizontal() {
  return isHorizontal;
}

// Get piece size.
public int getSize() {
  return size;
}

// Get piece row.
public int getRow() {
  return row;
}

// Get piece column.
public int getCol() {
  return col;
}

// Set piece row.
public void setRow(int row) {
  this.row = row;
```

```
  }

  // Set piece column.
  public void setCol(int col) {
    this.col = col;
  }

  // Create a copy.
  public Piece copy() {
    return new Piece(id, isPrimary, isHorizontal, size, row, col);
  }

  @Override
  public String toString() {
    return String.valueOf(id);
  }
}
```

3.2.3 Heuristic

Heuristic.java

```
package heuristic;

import core.GameState;

/**
 * Abstract base for heuristics.
 */
public abstract class Heuristic {
  public abstract int evaluate(GameState state);

  public abstract String getName();
```

```
}
```

BP.java

```
package heuristic;

import core.Board;
import core.GameState;
import core.Piece;

/**
 * Blocking Pieces heuristic.
 * Counts pieces in path.
 */
public class BP extends Heuristic {

  @Override
  public int evaluate(GameState state) {
    Board board = state.getBoard();
    Piece primary = board.getPrimaryPiece();
    char[][] grid = board.getGrid();

    if (primary == null) return Integer.MAX_VALUE;

    int blockingCount = 0;
    int exitR = board.getExitRow();
    int exitC = board.getExitCol();

    if (primary.isHorizontal()) {
      int r = primary.getRow();
      if (exitC >= primary.getCol() + primary.getSize() || exitC == board.getCols()) {
        for (int c = primary.getCol() + primary.getSize(); c < board.getCols(); c++) {
          if (c == exitC && r == exitR) break;
```

```
        if (grid[r][c] != '.') blockingCount++;
      }
    }

    else if (exitC < primary.getCol() || exitC == -1) {
      for (int c = primary.getCol() - 1; c >= 0; c--) {
        if (c == exitC && r == exitR) break;
        if (grid[r][c] != '.') blockingCount++;
      }
    }
  } else {
    int c = primary.getCol();

    if (exitR >= primary.getRow() + primary.getSize() || exitR == board.getRows()) {
      for (int r = primary.getRow() + primary.getSize(); r < board.getRows(); r++) {
        if (r == exitR && c == exitC) break;
        if (grid[r][c] != '.') blockingCount++;
      }
    }

    else if (exitR < primary.getRow() || exitR == -1) {
      for (int r = primary.getRow() - 1; r >= 0; r--) {
        if (r == exitR && c == exitC) break;
        if (grid[r][c] != '.') blockingCount++;
      }
    }
  }
  return blockingCount;
}

@Override
public String getName() {
  return "Blocking Pieces";
```

```
  }
}
```

DB.java

```
package heuristic;

import core.GameState;

/**
 * Distance + Blocking heuristic.
 * Combines MD and BP.
 */
public class DB extends Heuristic {
  private final MD mdHeuristic;
  private final BP bpHeuristic;

  public DB() {
    this.mdHeuristic = new MD();
    this.bpHeuristic = new BP();
  }

  @Override
  public int evaluate(GameState state) {
    return mdHeuristic.evaluate(state) + bpHeuristic.evaluate(state);
  }

  @Override
  public String getName() {
    return "Distance + Blocking";
  }
}
```

MD.java

```
package heuristic;

import core.Board;
import core.GameState;
import core.Piece;

/**
 * Manhattan Distance heuristic.
 * Estimates moves to exit.
 */
public class MD extends Heuristic {

  @Override
  public int evaluate(GameState state) {
    Board board = state.getBoard();
    Piece primary = board.getPrimaryPiece();

    if (primary == null) return Integer.MAX_VALUE;

    int exitR = board.getExitRow();
    int exitC = board.getExitCol();

    if (primary.isHorizontal()) {
      int targetCol = (exitC == board.getCols() || exitC == -1) ? exitC : exitC;
      if (targetCol == board.getCols()) {
        return Math.max(0, targetCol - (primary.getCol() + primary.getSize()));
      } else if (targetCol == -1) {
        return Math.max(0, primary.getCol() - targetCol);
      } else {
        return Math.abs(primary.getCol() - targetCol);
      }
    } else {
```

```
        int targetRow = (exitR == board.getRows() || exitR == -1) ? exitR : exitR;
        if (targetRow == board.getRows()) {
          return Math.max(0, targetRow - (primary.getRow() + primary.getSize()));
        } else if (targetRow == -1) {
          return Math.max(0, primary.getRow() - targetRow);
        } else {
          return Math.abs(primary.getRow() - targetRow);
        }
      }
    }

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

3.2.4 UI

Animation.java

```
package ui;

import core.GameState;
import java.awt.FlowLayout;
import java.util.List;
import java.util.concurrent.atomic.AtomicInteger;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JSlider;
import javax.swing.JTextArea;
import javax.swing.Timer;
```

```
/**
 * Animates solution playback.
 */
public class Animation {
  private final BoardPanel boardPanel;
  private final List<GameState> solutionPath;
  private final JTextArea logArea;
  private int animationDelayMillis;
  private Timer animationTimer;
  private final AtomicInteger currentStep = new AtomicInteger(0);

  private JLabel stepLabel;
  private JButton playButton, pauseButton, resetButton;
  private JSlider speedSlider;

      public Animation(BoardPanel boardPanel, List<GameState> solutionPath,
JTextArea logArea) {
    this.boardPanel = boardPanel;
    this.solutionPath = solutionPath;
    this.logArea = logArea;
    this.animationDelayMillis = 1000;
  }

  // Start animation playback.
  public void start() {
    if (playButton != null && playButton.isEnabled()) {
      playButton.doClick();
    } else if (solutionPath != null && !solutionPath.isEmpty()){
      setupAndPlayAnimation();
    }
  }
```

```
// Create animation controls.
public JPanel createControlPanel() {
  JPanel panel = new JPanel(new FlowLayout(FlowLayout.LEFT));
  int totalSteps = !solutionPath.isEmpty() ? solutionPath.size() - 1 : 0;
  stepLabel = new JLabel("Step: 0/" + totalSteps);
  playButton = new JButton("Play");
  pauseButton = new JButton("Pause");
  resetButton = new JButton("Reset");
  speedSlider = new JSlider(100, 2000, 2100 - animationDelayMillis);

  pauseButton.setEnabled(false);
  playButton.setEnabled(totalSteps > 0);
  resetButton.setEnabled(false);

  playButton.addActionListener(_ -> setupAndPlayAnimation());
  pauseButton.addActionListener(_ -> pauseAnimation());
  resetButton.addActionListener(_ -> resetAnimation());
  speedSlider.addChangeListener(_ -> {
    animationDelayMillis = 2100 - speedSlider.getValue();
    if (animationTimer != null && animationTimer.isRunning()) {
      animationTimer.setDelay(animationDelayMillis);
    }
  });

  panel.add(stepLabel);
  panel.add(playButton);
  panel.add(pauseButton);
  panel.add(resetButton);
  panel.add(new JLabel("Speed:"));
  panel.add(speedSlider);
  return panel;
}
```

```
// Setup and play/resume.
private void setupAndPlayAnimation() {
  if (solutionPath == null || solutionPath.isEmpty()) return;

  playButton.setEnabled(false);
  pauseButton.setEnabled(true);
  resetButton.setEnabled(true);

  if (animationTimer != null && animationTimer.isRunning()) animationTimer.stop();

  animationTimer = new Timer(animationDelayMillis, _ -> {
    int step = currentStep.get();
    if (step < solutionPath.size() -1) {
      step = currentStep.incrementAndGet();
      GameState state = solutionPath.get(step);
      boardPanel.updateBoard(state.getBoard());
      stepLabel.setText("Step: " + step + "/" + (solutionPath.size() - 1));
      logArea.append("Langkah " + step + ": " + state.getLastMove() + "\n");

      if (step == solutionPath.size() - 1) {
        animationTimer.stop();
        playButton.setEnabled(false);
        pauseButton.setEnabled(false);
      }
    } else {
       animationTimer.stop();
       playButton.setEnabled(false);
       pauseButton.setEnabled(false);
    }
  });

  if (currentStep.get() == 0 && !solutionPath.isEmpty()) {
```

```
      boardPanel.updateBoard(solutionPath.get(0).getBoard());
      logArea.append("Papan Awal (Langkah 0)\n");
    }
    animationTimer.start();
  }

  // Pause the animation.
  private void pauseAnimation() {
    if (animationTimer != null && animationTimer.isRunning()) animationTimer.stop();
    playButton.setEnabled(currentStep.get() < solutionPath.size() -1);
    pauseButton.setEnabled(false);
  }

  // Reset animation to start.
  private void resetAnimation() {
    if (animationTimer != null && animationTimer.isRunning()) animationTimer.stop();
    currentStep.set(0);
    if (solutionPath != null && !solutionPath.isEmpty()) {
      boardPanel.updateBoard(solutionPath.get(0).getBoard());
      stepLabel.setText("Step: 0/" + (solutionPath.size() - 1));
    } else {
      stepLabel.setText("Step: 0/0");
    }
    logArea.setText("");
    playButton.setEnabled(solutionPath != null && solutionPath.size() > 1);
    pauseButton.setEnabled(false);
    resetButton.setEnabled(false);
  }
}
```

BoardPanel.java

```
package ui;
```

```
import core.Board;
import core.Piece;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.util.HashMap;
import java.util.Map;
import java.util.Random;
import javax.swing.JPanel;

/**
 * Renders the game board.
 */
public class BoardPanel extends JPanel {
  private Board board;
  private final int cellSize = 60;
  private final Map<Character, Color> pieceColors;
  private final int margin = cellSize;
  private final Random random = new Random();

  public BoardPanel(Board board) {
    this.board = board;
    this.pieceColors = new HashMap<>();
    pieceColors.put('P', Color.RED);

    assignPieceColors();
    updatePreferredSizeAndRevalidate();
  }
```

```
  // Assigns colors to pieces.
  private void assignPieceColors() {
    if (this.board != null && this.board.getPieces() != null) {
      for (Piece piece : this.board.getPieces()) {
        char id = piece.getId();
        if (id != 'P' && !pieceColors.containsKey(id)) {
          pieceColors.put(id, generateRandomColor());
        }
      }
    }
  }

  // Updates preferred size and revalidates.
  private void updatePreferredSizeAndRevalidate() {
      int cols = (this.board != null && this.board.getCols() > 0) ? this.board.getCols() :
6;
    int rows = (this.board != null && this.board.getRows() > 0) ? this.board.getRows()
: 6;

    int panelWidth = cols * this.cellSize + 2 * this.margin;
    int panelHeight = rows * this.cellSize + 2 * this.margin;

    setPreferredSize(new Dimension(panelWidth, panelHeight));
    revalidate();
  }

  // Get current board.
  public Board getBoard() {
    return board;
  }

  // Update displayed board.
  public void updateBoard(Board newBoard) {
```

```
    this.board = newBoard;
    assignPieceColors();
    updatePreferredSizeAndRevalidate();
    repaint();
  }

  // Generate random piece color.
  private Color generateRandomColor() {
                        return new Color(random.nextInt(180), random.nextInt(180),
random.nextInt(180));
  }

  @Override
  protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    if (board == null) return;

    Graphics2D g2d = (Graphics2D) g.create();
    g2d.translate(margin, margin);

    int rows = board.getRows();
    int cols = board.getCols();
    char[][] grid = board.getGrid();

    // Draw grid cells
    for (int r = 0; r < rows; r++) {
      for (int c = 0; c < cols; c++) {
        // Check if this is an empty cell
        if (grid[r][c] == '.') {
          g2d.setColor(Color.LIGHT_GRAY);
          g2d.drawRect(c * cellSize, r * cellSize, cellSize, cellSize);
        }
      }
```

```
    }

    // Draw pieces
    g2d.setFont(new Font("Arial", Font.BOLD, cellSize / 3));
    FontMetrics fm = g2d.getFontMetrics();

    for (Piece piece : board.getPieces()) {
      int r = piece.getRow();
      int c = piece.getCol();

      // Skip pieces that are completely off-board
      if (r < -piece.getSize() || r >= rows + piece.getSize() ||
        c < -piece.getSize() || c >= cols + piece.getSize()){
        continue;
      }

      g2d.setColor(pieceColors.getOrDefault(piece.getId(), Color.DARK_GRAY));
      int pieceWidth = piece.isHorizontal() ? piece.getSize() * cellSize : cellSize;
      int pieceHeight = piece.isHorizontal() ? cellSize : piece.getSize() * cellSize;
      g2d.fillRect(c * cellSize + 2, r * cellSize + 2, pieceWidth - 4, pieceHeight - 4);

      g2d.setColor(Color.WHITE);
      String label = String.valueOf(piece.getId());
      int textX = c * cellSize + (pieceWidth - fm.stringWidth(label)) / 2;
      int textY = r * cellSize + (pieceHeight - fm.getHeight()) / 2 + fm.getAscent();
      g2d.drawString(label, textX, textY);
    }

    // Draw exit 'K'
    drawExit(g2d, rows, cols, fm);
    g2d.dispose();
  }
```

```
  // Helper to draw exit.
  private void drawExit(Graphics2D g2d, int rows, int cols, FontMetrics fm) {
    int exitR = board.getExitRow();
    int exitC = board.getExitCol();
            if (exitR == -1 && exitC == -1 && (board.getPrimaryPiece() == null ||
!board.getPrimaryPiece().isPrimary())) return;



    g2d.setColor(Color.GREEN);
    String kLabel = "K";
    int textWidth = fm.stringWidth(kLabel);
    int textHeight = fm.getAscent();
    int kRectX, kRectY, kRectW = cellSize, kRectH = cellSize;

    // Determine K position
    if (exitC == cols) { kRectX = cols * cellSize; kRectY = exitR * cellSize; } // Right
    else if (exitC == -1) { kRectX = -cellSize; kRectY = exitR * cellSize; }   // Left
       else if (exitR == rows) { kRectX = exitC * cellSize; kRectY = rows * cellSize; } //
Bottom
    else if (exitR == -1) { kRectX = exitC * cellSize; kRectY = -cellSize; }   // Top
    else if (exitR >= 0 && exitR < rows && exitC >= 0 && exitC < cols) { // Within grid
      kRectX = exitC * cellSize; kRectY = exitR * cellSize;
    } else return;

    g2d.fillRect(kRectX + 2, kRectY + 2, kRectW - 4, kRectH - 4);
    g2d.setColor(Color.BLACK);
    g2d.drawString(kLabel,
            kRectX + (kRectW - textWidth) / 2,
            kRectY + (kRectH - fm.getHeight()) / 2 + textHeight);
  }
}
```

CLI.java

```
package ui;

import algorithm.AS;
import algorithm.GBFS;
import algorithm.IDAS;
import algorithm.PathFinder;
import algorithm.UCS;
import core.Board;
import core.FileParser;
import core.GameState;
import core.Move;
import core.Piece;
import heuristic.BP;
import heuristic.DB;
import heuristic.Heuristic;
import heuristic.MD;
import java.io.File;
import java.io.IOException;
import java.util.List;
import java.util.Scanner;

/**
 * Command-line interface.
 */
public class CLI {
  private static final String ANSI_RESET = "\u001B[0m";
  private static final String ANSI_RED   = "\u001B[31m";
  private static final String ANSI_GREEN = "\u001B[32m";
  private static final String ANSI_YELLOW = "\u001B[33m";

  // Main entry for CLI.
  public static void main(String[] args) {
    CLI.run();
```

```
  }

  // Run the CLI application.
  public static void run() {
    try (Scanner scanner = new Scanner(System.in)) {
      System.out.print("Enter puzzle file name: ");
      String fileName = scanner.nextLine().trim();
      String filePath = resolveFilePath(fileName);

      Board board = new FileParser(filePath).parseFile();
      GameState initialState = new GameState(board);

      System.out.println("\nSelect algorithm:");
      System.out.println("1. UCS 2. GBFS 3. A* 4. IDAS");
      System.out.print("Choice: ");
      int algoChoice = scanner.nextInt();

      Heuristic heuristic = null;
      if (algoChoice > 1 && algoChoice <= 4) {
        System.out.println("\nSelect heuristic:");
                    System.out.println("1. Manhattan Distance  2. Blocking Pieces  3.
Distance+Blocking");
        System.out.print("Choice: ");
        heuristic = switch (scanner.nextInt()) {
          case 2 -> new BP();
          case 3 -> new DB();
          default -> new MD();
        };
      }

      System.out.println("\nInitial Board:");
      printBoard(board, '\0');
```

```
        GameState solutionState;
        int nodes;
        long timeMs;
        String algoName;

        if (algoChoice == 1) {
          UCS solver = new UCS();
          solutionState = solver.solve(initialState);
          nodes = solver.getNodesVisited();
          timeMs = solver.getExecutionTime();
          algoName = solver.getName();
        } else if (algoChoice >= 2 && algoChoice <= 4) {
          PathFinder solver = switch (algoChoice) {
            case 2 -> new GBFS(heuristic);
            case 3 -> new AS(heuristic);
            case 4 -> new IDAS(heuristic);
                    default -> throw new IllegalArgumentException("Invalid algorithm
choice.");
          };
          solutionState = solver.findPath(initialState);
          nodes = solver.getNodesVisited();
          timeMs = solver.getExecutionTime();
          algoName = solver.getName();
        } else {
          System.out.println("Invalid algorithm choice.");
          return;
        }

        if (solutionState != null) {
          List<GameState> path = solutionState.getSolutionPath();
          System.out.println("\nSolution: " + (path.size() - 1) + " moves.");
          for (int i = 1; i < path.size(); i++) {
            GameState state = path.get(i);
```

```
            Move move = state.getLastMove();
            System.out.println("\nMove " + i + ": " + move);
            printBoard(state.getBoard(), move.getPiece().getId());
          }
          System.out.println("\n--- Statistics ---");
          System.out.println("Algorithm: " + algoName);
          if (heuristic != null) System.out.println("Heuristic: " + heuristic.getName());
          System.out.println("Nodes visited: " + nodes);
          System.out.println("Time: " + timeMs + " ms");
        } else {
          System.out.println("\nNo solution found.");
           System.out.println("Nodes visited: " + nodes);
          System.out.println("Time: " + timeMs + " ms");
        }

      } catch (IOException e) {
        System.err.println("File Error: " + e.getMessage());
      } catch (Exception e) {
        System.err.println("Error: " + e.getMessage());
      }
    }

    // Resolve file path.
    private static String resolveFilePath(String fileName) {
      if (fileName.contains(File.separator)) return fileName;
      if (!fileName.toLowerCase().endsWith(".txt")) fileName += ".txt";

      File fileInTestDir = new File("test", fileName);
      if (fileInTestDir.exists()) return fileInTestDir.getAbsolutePath();

        File fileInCurrentTestDir = new File(System.getProperty("user.dir"), "test/input"
+ File.separator + fileName);
       if (fileInCurrentTestDir.exists()) return fileInCurrentTestDir.getAbsolutePath();
```

```
    return fileName;
  }

  // Print board with colors.
  private static void printBoard(Board board, char movedPieceId) {
    char[][] grid = board.getGrid();
    int rows = board.getRows(), cols = board.getCols();
    Piece primary = board.getPrimaryPiece();

    for (int i = 0; i < rows; i++) {
      // Skip printing if this is a space-only row
      if (board.isRowSpaceOnly(i)) {
        continue;
      }

      for (int j = 0; j < cols; j++) {
        // Skip printing if this is a space-only column
        if (board.isColSpaceOnly(j)) {
          continue;
        }

        char c = grid[i][j];
        if (c == '.') System.out.print('.');
        else if (primary != null && c == primary.getId()) System.out.print(ANSI_RED + c
+ ANSI_RESET);
                              else if (movedPieceId != '\0' && c == movedPieceId)
System.out.print(ANSI_YELLOW + c + ANSI_RESET);
        else System.out.print(c);
      }

      if (board.getExitRow() == i && board.getExitCol() == cols) {
        System.out.print(" " + ANSI_GREEN + 'K' + ANSI_RESET);
```

```
      }
      System.out.println();
    }
    if (board.getExitRow() == rows) {
      for(int j=0; j<board.getExitCol(); ++j) System.out.print(" ");
      System.out.println(ANSI_GREEN + 'K' + ANSI_RESET);
    }
  }
}
```

GUI.java

```
package ui;

import algorithm.AS;
import algorithm.GBFS;
import algorithm.IDAS;
import algorithm.PathFinder;
import algorithm.UCS;
import core.Board;
import core.FileParser;
import core.GameState;
import heuristic.BP;
import heuristic.DB;
import heuristic.Heuristic;
import heuristic.MD;
import java.awt.AWTException;
import java.awt.BorderLayout;
import java.awt.Component;
import java.awt.FlowLayout;
import java.awt.Robot;
import java.awt.image.BufferedImage;
import java.io.File;
```

```
import java.io.IOException;
import java.util.List;
import javax.imageio.ImageIO;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.SwingConstants;
import javax.swing.SwingUtilities;
import javax.swing.filechooser.FileNameExtensionFilter;

/**
 * Main graphical user interface.
 */
public class GUI extends JFrame {
  private BoardPanel boardPanel;
  private JTextArea logArea;
  private JPanel solutionDisplayArea;
  private JButton solveButton;
  private JComboBox<String> algoSelector;
  private JComboBox<String> heuristicSelector;
  private JLabel currentFileLabel;
  private Board currentBoard;
  private String currentPuzzleName; // Store current puzzle name for screenshots

  public GUI() {
    setTitle("Rush Hour Solver");
```

```
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setLayout(new BorderLayout(5, 5));

    setupControlPanel();
    setupSolutionDisplayArea();
    setupLogArea();

    pack();
    setLocationRelativeTo(null);
    setVisible(true);
  }

  // Setup top control panel.
  private void setupControlPanel() {
    JPanel panel = new JPanel(new FlowLayout(FlowLayout.LEFT, 5, 5));
    panel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));

    JButton fileButton = new JButton("Select Puzzle File");
    currentFileLabel = new JLabel("No file selected");
    fileButton.addActionListener(_ -> loadPuzzleFile());

    algoSelector = new JComboBox<>(new String[]{
        "Uniform Cost Search", "Greedy Best-First", "A* Search", "Iterative Deepening
A*"
    });
    heuristicSelector = new JComboBox<>(new String[]{
      "Manhattan Distance", "Blocking Pieces", "Distance + Blocking"
    });
                                                algoSelector.addActionListener(_      ->
heuristicSelector.setEnabled(algoSelector.getSelectedIndex() != 0));
    heuristicSelector.setEnabled(algoSelector.getSelectedIndex() != 0);

    solveButton = new JButton("Solve");
```

```
    solveButton.addActionListener(_ -> solvePuzzle());
    solveButton.setEnabled(false);

    panel.add(fileButton);
    panel.add(currentFileLabel);
    panel.add(new JLabel("Algorithm:"));
    panel.add(algoSelector);
    panel.add(new JLabel("Heuristic:"));
    panel.add(heuristicSelector);
    panel.add(solveButton);
    add(panel, BorderLayout.NORTH);
  }

  // Setup main board display.
  private void setupSolutionDisplayArea() {
    solutionDisplayArea = new JPanel(new BorderLayout());
    boardPanel = new BoardPanel(null);
    solutionDisplayArea.add(boardPanel, BorderLayout.CENTER);
    add(solutionDisplayArea, BorderLayout.CENTER);
  }

  // Setup bottom log area.
  private void setupLogArea() {
    logArea = new JTextArea(8, 40);
    logArea.setEditable(false);
    logArea.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));
    JScrollPane scrollPane = new JScrollPane(logArea);

    JPanel bottomPanel = new JPanel(new BorderLayout());
    bottomPanel.add(scrollPane, BorderLayout.CENTER);
          JLabel footer = new JLabel("© 2025 IF2211_TK3_13523139_18222130",
SwingConstants.CENTER);
    footer.setBorder(BorderFactory.createEmptyBorder(5, 0, 5, 0));
```

```
    bottomPanel.add(footer, BorderLayout.SOUTH);
    add(bottomPanel, BorderLayout.SOUTH);
  }

  // Load puzzle from file.
  private void loadPuzzleFile() {
    File testDir = new File(System.getProperty("user.dir"), "test/input");
    JFileChooser chooser = new JFileChooser(testDir.exists() ? testDir : null);
    chooser.setFileFilter(new FileNameExtensionFilter("Text Files (*.txt)", "txt"));

    if (chooser.showOpenDialog(this) == JFileChooser.APPROVE_OPTION) {
      File file = chooser.getSelectedFile();
      String fileName = file.getName();
      currentFileLabel.setText(fileName);
      currentPuzzleName = fileName.replace(".txt", "");

      try {
        currentBoard = new FileParser(file.getAbsolutePath()).parseFile();
        displayBoard(currentBoard);
        logArea.setText("Puzzle loaded: " + fileName + "\n");
        solveButton.setEnabled(true);

        // Clear previous solution animation controls
        if (solutionDisplayArea.getComponentCount() > 1) {
                                                    Component lastComp =
solutionDisplayArea.getComponent(solutionDisplayArea.getComponentCount()
-1);
          if (!(lastComp instanceof BoardPanel)) {
            solutionDisplayArea.remove(lastComp);
          }
        }
        // After removing old controls, revalidate and repaint the container
        solutionDisplayArea.revalidate();
```

```
        solutionDisplayArea.repaint();

        // Take screenshot of initial state
        SwingUtilities.invokeLater(() -> {
          try {
            // Small delay to ensure UI is fully drawn
            Thread.sleep(200);
            takeScreenshot("initial");
          } catch (InterruptedException ex) {
                        logArea.append("Failed to capture initial state screenshot: " +
ex.getMessage() + "\n");
          }
        });
      } catch (IOException ex) {
            JOptionPane.showMessageDialog(this, "Error: " + ex.getMessage(), "File
Load Error", JOptionPane.ERROR_MESSAGE);
        logArea.setText("Error loading " + fileName + ": " + ex.getMessage() + "\n");
        currentBoard = null;
        solveButton.setEnabled(false);
      }
    }
  }

  // Update board display.
  private void displayBoard(Board boardToDisplay) {
    if (boardPanel == null) {
      boardPanel = new BoardPanel(boardToDisplay);
      solutionDisplayArea.add(boardPanel, BorderLayout.CENTER);
    } else {
      boardPanel.updateBoard(boardToDisplay);
    }
    pack();
  }
```

```
  // Solve the current puzzle.
  private void solvePuzzle() {
    if (currentBoard == null) {
                    JOptionPane.showMessageDialog(this, "Load puzzle first.", "Error",
JOptionPane.ERROR_MESSAGE);
      return;
    }
    solveButton.setEnabled(false);
    logArea.setText("Solving puzzle...\n");

    int algoIdx = algoSelector.getSelectedIndex();
    Heuristic selectedHeuristic = switch (heuristicSelector.getSelectedIndex()) {
      case 1 -> new BP();
      case 2 -> new DB();
      default -> new MD();
    };

    new Thread(() -> {
      GameState initial = new GameState(currentBoard.copy());
      GameState solutionState;
      int nodes;
      long timeMs;
      String algoName = algoSelector.getSelectedItem().toString();
      String heurName = heuristicSelector.getSelectedItem().toString();

      try {
        if (algoIdx == 0) {
          UCS solver = new UCS();
          solutionState = solver.solve(initial);
          nodes = solver.getNodesVisited();
          timeMs = solver.getExecutionTime();
        } else {
```

```
        PathFinder solver = switch (algoIdx) {
          case 1 -> new GBFS(selectedHeuristic);
          case 2 -> new AS(selectedHeuristic);
          case 3 -> new IDAS(selectedHeuristic);
          default -> throw new IllegalStateException("Invalid algorithm index.");
        };
        solutionState = solver.findPath(initial);
        nodes = solver.getNodesVisited();
        timeMs = solver.getExecutionTime();
      }
    } catch (IllegalStateException ex) {
      final String errorMsg = "Solver error: " + ex.getMessage();
      SwingUtilities.invokeLater(() -> {
        logArea.append(errorMsg + "\n");
        solveButton.setEnabled(true);
      });
      return;
    }

    final GameState finalSolution = solutionState;
    final int finalNodes = nodes;
    final long finalTimeMs = timeMs;
    final String finalHeurName = (algoIdx != 0) ? heurName : null;
    final String finalAlgoName = algoName.replaceAll("\\s+", "");

    SwingUtilities.invokeLater(() -> {
      if (finalSolution != null) {
        List<GameState> path = finalSolution.getSolutionPath();
        displaySolutionAnimation(path);
        logArea.append("Solution: " + (path.size() - 1) + " moves.\n");

        // Take screenshot of final state
        new Thread(() -> {
```

```
            try {
              // Jump to the last state in the animation
              boardPanel.updateBoard(path.get(path.size() - 1).getBoard());
              Thread.sleep(500);
              takeScreenshot("final_" + finalAlgoName);
            } catch (InterruptedException ex) {
              SwingUtilities.invokeLater(() ->
                      logArea.append("Failed to capture final state screenshot: " +
ex.getMessage() + "\n")
              );
            }
          }).start();
        } else {
          logArea.append("No solution found.\n");
        }
        logArea.append("Algorithm: " + finalAlgoName + "\n");
            if (finalHeurName != null) logArea.append("Heuristic: " + finalHeurName +
"\n");
        logArea.append("Nodes visited: " + finalNodes + "\n");
        logArea.append("Time: " + finalTimeMs + " ms\n");
        solveButton.setEnabled(true);
      });
    }).start();
  }

  // Display solution animation.
  private void displaySolutionAnimation(List<GameState> solutionPath) {
    if (solutionPath == null || solutionPath.isEmpty()) return;

    boardPanel.updateBoard(solutionPath.get(0).getBoard());
    Animation animation = new Animation(boardPanel, solutionPath, logArea);
    JPanel animControls = animation.createControlPanel();
```

```
    if (solutionDisplayArea.getComponentCount() > 1) {
                                                    Component lastComp =
solutionDisplayArea.getComponent(solutionDisplayArea.getComponentCount()
-1);
       if (!(lastComp instanceof BoardPanel)) {
         solutionDisplayArea.remove(lastComp);
       }
    }
    solutionDisplayArea.add(animControls, BorderLayout.SOUTH);
    solutionDisplayArea.revalidate();
    solutionDisplayArea.repaint();
    pack();
    animation.start();
  }

  // Take screenshot of the application window
  private void takeScreenshot(String state) {
    try {
      // Get current algorithm and heuristic names
      String algoShortName;
      String heurShortName;
      String baseDir = "test" + File.separator + "output";
      String targetDir;

      if (state.contains("final")) {
        // Get algorithm short name
        int algoIdx = algoSelector.getSelectedIndex();
        switch (algoIdx) {
          case 0 -> algoShortName = "UCS";
          case 1 -> algoShortName = "GBFS";
          case 2 -> algoShortName = "AS";
          case 3 -> algoShortName = "IDAS";
          default -> algoShortName = "Unknown";
```

```
    }

    // Get heuristic short name
    if (algoIdx != 0) {
      int heurIdx = heuristicSelector.getSelectedIndex();
      switch (heurIdx) {
        case 0 -> heurShortName = "MD";
        case 1 -> heurShortName = "BP";
        case 2 -> heurShortName = "DB";
        default -> heurShortName = "Unknown";
      }

      // For final state
      targetDir = baseDir + File.separator + "final" +
            File.separator + algoShortName +
            File.separator + heurShortName;
    } else {
      // For UCS with no heuristic
      targetDir = baseDir + File.separator + "final" +
            File.separator + algoShortName;
    }
  } else {
    // For initial state
    targetDir = baseDir + File.separator + "initial";
  }

  // Create the directories if they don't exist
  File outputDir = new File(targetDir);
  if (!outputDir.exists()) {
    outputDir.mkdirs();
  }

  String filename = targetDir + File.separator + currentPuzzleName + ".png";
```

```
      // Capture only the board panel
      BufferedImage screenshot;
      if (boardPanel != null) {
        screenshot = new BufferedImage(
          boardPanel.getWidth(),
          boardPanel.getHeight(),
          BufferedImage.TYPE_INT_ARGB
        );

        boardPanel.paint(screenshot.getGraphics());
      } else {
        // Fallback
        screenshot = new Robot().createScreenCapture(getBounds());
      }

      // Save to file
      File outputFile = new File(filename);
      ImageIO.write(screenshot, "png", outputFile);
      logArea.append("Board screenshot saved: " + outputFile.getPath() + "\n");

    } catch (AWTException | IOException e) {
      logArea.append("Error taking screenshot: " + e.getMessage() + "\n");
    }
  }

  public static void main(String[] args) {
    SwingUtilities.invokeLater(GUI::new);
  }
}
```

3.2.5 Main

Main.java

```
import javax.swing.SwingUtilities;
import ui.CLI;
import ui.GUI;

/**
 * Main application entry point.
 * Launches CLI or GUI.
 */
public class Main {
  public static void main(String[] args) {
    boolean useCLI = false;
    if (args != null) {
      for (String arg : args) {
        if ("--cli".equalsIgnoreCase(arg) || "-cli".equalsIgnoreCase(arg)) {
          useCLI = true;
          break;
        }
      }
    }

    if (useCLI) {
      CLI.run();
    } else {
      SwingUtilities.invokeLater(GUI::new);
    }
  }
}
```

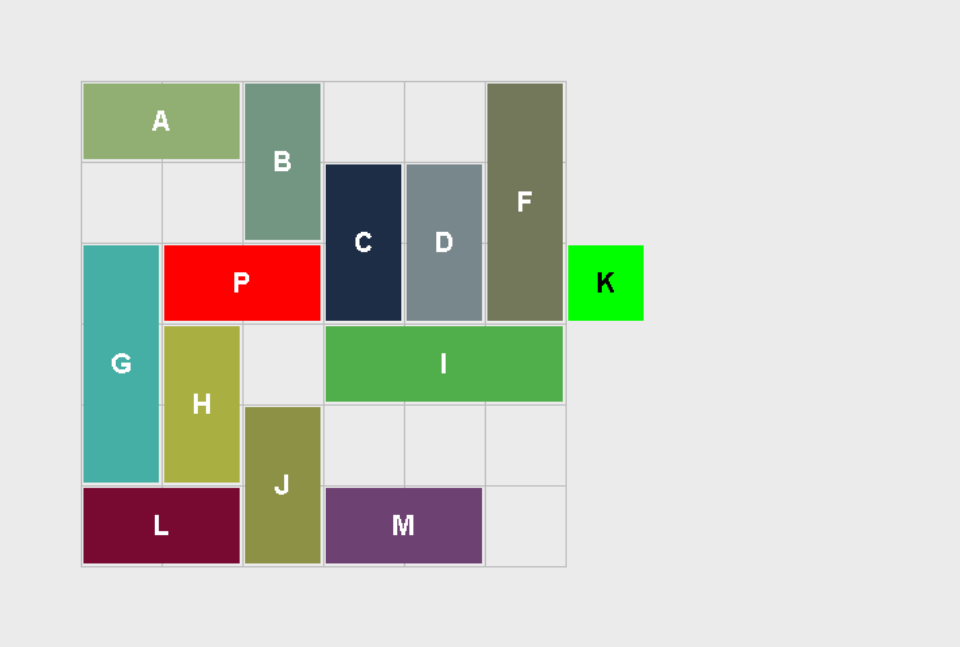
Bab IV

Hasil dan Pembahasan

4.1 Hasil Eksperimen

4.1.1 Test Case 1

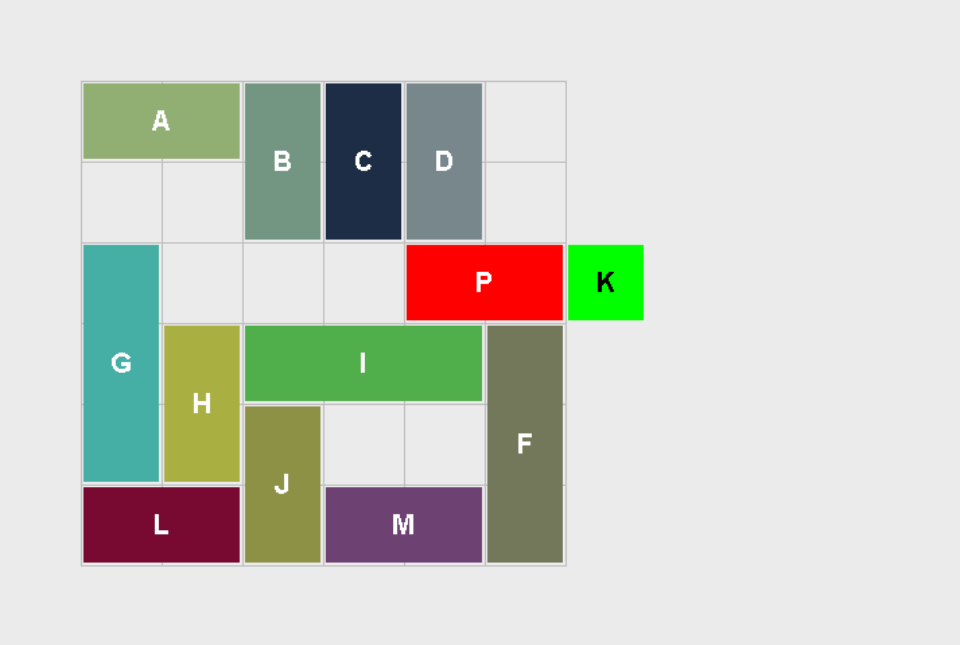
a. Input



Gambar 2. Initial State Test Case 1

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

b. Output (UCS)



Gambar 3. Final State Test Case 1 dengan UCS

```
Initial Board:
AAB..F
..BCDF
GPPCDF K
GH.III
GHJ...
LLJMM.

Solution: 5 moves.

Move 1: D-atas
AAB.DF
..BCDF
GPPC.F K
GH.III
GHJ...
LLJMM.
```

Move 2: I-kiri
AAB.DF
..BCDF
GPPC.F K
GHIII.
GHJ...
LLJMM.

Move 3: C-atas
AABCDF
..BCDF
GPP..F K
GHIII.
GHJ...
LLJMM.

Move 4: F-bawah
AABCD.
..BCD.
GPP... K
GHIIIF
GHJ..F
LLJMMF

Move 5: P-kanan
AABCD.
..BCD.
G...PP K
GHIIIF
GHJ..F
LLJMMF

--- Statistics ---

```
Algorithm: Uniform Cost Search
Nodes visited: 424
Time: 24 ms
```

c. Output (GBFS)

1. Manhattan Distance

```
Initial Board:
AAB..F
..BCDF
GPPCDF K
GH.III
GHJ...
LLJMM.

Solution: 58 moves.

Move 1: C-atas
AABC.F
..BCDF
GPP.DF K
GH.III
GHJ...
LLJMM.

Move 2: P-kanan
AABC.F
..BCDF
G.PPDF K
GH.III
GHJ...
LLJMM.
```

Move 3: D-atas

AABCDF

..BCDF

G.PP.F K

GH.III

GHJ...

LLJMM.

Move 4: P-kanan

AABCDF

..BCDF

G..PPF K

GH.III

GHJ...

LLJMM.

Move 5: G-atas

AABCDF

G.BCDF

G..PPF K

GH.III

.HJ...

LLJMM.

Move 6: J-atas

AABCDF

G.BCDF

G.JPPF K

GHJIII

.H....

LL.MM.

Move 7: M-kanan

AABCDF
G.BCDF
G.JPPF K
GHJIII
.H....
LL..MM

Move 8: L-kanan
AABCDF
G.BCDF
G.JPPF K
GHJIII
.H....
..LLMM

Move 9: J-bawah
AABCDF
G.BCDF
G..PPF K
GHJIII
.HJ...
..LLMM

Move 10: L-kiri
AABCDF
G.BCDF
G..PPF K
GHJIII
.HJ...
LL..MM

Move 11: H-atas
AABCDF

GHBCDF
GH.PPF K
G.JIII
..J...
LL..MM

Move 12: M-kiri
AABCDF
GHBCDF
GH.PPF K
G.JIII
..J...
LLMM..

Move 13: J-atas
AABCDF
GHBCDF
GHJPPF K
G.JIII
.....
LLMM..

Move 14: G-bawah
AABCDF
.HBCDF
GHJPPF K
G.JIII
G.....
LLMM..

Move 15: M-kanan
AABCDF
.HBCDF

GHJPPF K
G.JIII
G.....
LL..MM

Move 16: L-kanan
AABCDF
.HBCDF
GHJPPF K
G.JIII
G.....
..LLMM

Move 17: J-bawah
AABCDF
.HBCDF
GH.PPF K
G.JIII
G.J...
..LLMM

Move 18: L-kiri
AABCDF
.HBCDF
GH.PPF K
G.JIII
G.J...
LL..MM

Move 19: B-bawah
AA.CDF
.HBCDF
GHBPPF K

G.JIII
G.J...
LL..MM

Move 20: M-kiri
AA.CDF
.HBCDF
GHBPPF K
G.JIII
G.J...
LLMM..

Move 21: H-bawah
AA.CDF
..BCDF
G.BPPF K
GHJIII
GHJ...
LLMM..

Move 22: M-kanan
AA.CDF
..BCDF
G.BPPF K
GHJIII
GHJ...
LL..MM

Move 23: L-kanan
AA.CDF
..BCDF
G.BPPF K
GHJIII

GHJ...
..LLMM

Move 24: H-atas
AA.CDF
.HBCDF
GHBPPF K
G.JIII
G.J...
..LLMM

Move 25: G-atas
AA.CDF
GHBCDF
GHBPPF K
G.JIII
..J...
..LLMM

Move 26: L-kiri
AA.CDF
GHBCDF
GHBPPF K
G.JIII
..J...
LL..MM

Move 27: M-kiri
AA.CDF
GHBCDF
GHBPPF K
G.JIII
..J...

LLMM..

Move 28: H-bawah
AA.CDF
G.BCDF
G.BPPF K
GHJIII
.HJ...
LLMM..

Move 29: M-kanan
AA.CDF
G.BCDF
G.BPPF K
GHJIII
.HJ...
LL..MM

Move 30: L-kanan
AA.CDF
G.BCDF
G.BPPF K
GHJIII
.HJ...
..LLMM

Move 31: H-bawah
AA.CDF
G.BCDF
G.BPPF K
G.JIII
.HJ...
.HLLMM

Move 32: A-kanan

.AACDF

G.BCDF

G.BPPF K

G.JIII

.HJ...

.HLLMM

Move 33: H-atas

.AACDF

GHBCDF

GHBPPF K

G.JIII

..J...

..LLMM

Move 34: L-kiri

.AACDF

GHBCDF

GHBPPF K

G.JIII

..J...

LL..MM

Move 35: J-bawah

.AACDF

GHBCDF

GHBPPF K

G..III

..J...

LLJ.MM

Move 36: I-kiri
.AACDF
GHBCDF
GHBPPF K
GIII..
..J...
LLJ.MM

Move 37: M-kiri
.AACDF
GHBCDF
GHBPPF K
GIII..
..J...
LLJMM.

Move 38: G-atas
GAACDF
GHBCDF
GHBPPF K
.III..
..J...
LLJMM.

Move 39: I-kiri
GAACDF
GHBCDF
GHBPPF K
III...
..J...
LLJMM.

Move 40: M-kanan

GAACDF
GHBCDF
GHBPPF K
III...
..J...
LLJ.MM

Move 41: F-bawah
GAACD.
GHBCD.
GHBPPF K
III..F
..J..F
LLJ.MM

Move 42: M-kiri
GAACD.
GHBCD.
GHBPPF K
III..F
..J..F
LLJMM.

Move 43: I-kanan
GAACD.
GHBCD.
GHBPPF K
..IIIF
..J..F
LLJMM.

Move 44: M-kanan
GAACD.

GHBCD.
GHBPPF K
..IIIF
..J..F
LLJ.MM

Move 45: H-bawah
GAACD.
G.BCD.
G.BPPF K
.HIIIF
.HJ..F
LLJ.MM

Move 46: M-kiri
GAACD.
G.BCD.
G.BPPF K
.HIIIF
.HJ..F
LLJMM.

Move 47: G-bawah
.AACD.
..BCD.
G.BPPF K
GHIIIF
GHJ..F
LLJMM.

Move 48: M-kanan
.AACD.
..BCD.

G.BPPF K
GHIIIF
GHJ..F
LLJ.MM

Move 49: H-atas
.AACD.
.HBCD.
GHBPPF K
G.IIIF
G.J..F
LLJ.MM

Move 50: F-atas
.AACDF
.HBCDF
GHBPPF K
G.III.
G.J...
LLJ.MM

Move 51: M-kiri
.AACDF
.HBCDF
GHBPPF K
G.III.
G.J...
LLJMM.

Move 52: H-bawah
.AACDF
..BCDF
G.BPPF K

GHIII.
GHJ...
LLJMM.

Move 53: M-kanan
.AACDF
..BCDF
G.BPPF K
GHIII.
GHJ...
LLJ.MM

Move 54: G-atas
GAACDF
G.BCDF
G.BPPF K
.HIII.
.HJ...
LLJ.MM

Move 55: H-atas
GAACDF
GHBCDF
GHBPPF K
..III.
..J...
LLJ.MM

Move 56: M-kiri
GAACDF
GHBCDF
GHBPPF K
..III.

```
..J...
LLJMM.

Move 57: F-bawah
GAACD.
GHBCD.
GHBPP. K
..IIIF
..J..F
LLJMMF

Move 58: P-kanan
GAACD.
GHBCD.
GHB.PP K
..IIIF
..J..F
LLJMMF

--- Statistics ---
Algorithm: Greedy Best-First Search
Heuristic: Manhattan Distance
Nodes visited: 199
Time: 20 ms
```

2. Blocking Piece

```
Initial Board:
AAB..F
..BCDF
GPPCDF K
GH.III
GHJ...
```

LLJMM.

Solution: 6 moves.

Move 1: C-atas
AABC.F
..BCDF
GPP.DF K
GH.III
GHJ...
LLJMM.

Move 2: D-atas
AABCDF
..BCDF
GPP..F K
GH.III
GHJ...
LLJMM.

Move 3: P-kanan
AABCDF
..BCDF
G..PPF K
GH.III
GHJ...
LLJMM.

Move 4: I-kiri
AABCDF
..BCDF
G..PPF K
GHIII.

```
GHJ...
LLJMM.

Move 5: F-bawah
AABCD.
..BCD.
G..PP. K
GHIIIF
GHJ..F
LLJMMF

Move 6: P-kanan
AABCD.
..BCD.
G...PP K
GHIIIF
GHJ..F
LLJMMF

--- Statistics ---
Algorithm: Greedy Best-First Search
Heuristic: Blocking Pieces
Nodes visited: 13
Time: 9 ms
```

3. Distance + Blocking

```
Initial Board:
AAB..F
..BCDF
GPPCDF K
GH.III
GHJ...
```

LLJMM.

Solution: 8 moves.

Move 1: C-atas
AABC.F
..BCDF
GPP.DF K
GH.III
GHJ...
LLJMM.

Move 2: P-kanan
AABC.F
..BCDF
G.PPDF K
GH.III
GHJ...
LLJMM.

Move 3: D-atas
AABCDF
..BCDF
G.PP.F K
GH.III
GHJ...
LLJMM.

Move 4: P-kanan
AABCDF
..BCDF
G..PPF K
GH.III

GHJ...
LLJMM.

Move 5: G-atas
AABCDF
G.BCDF
G..PPF K
GH.III
.HJ...
LLJMM.

Move 6: I-kiri
AABCDF
G.BCDF
G..PPF K
GHIII.
.HJ...
LLJMM.

Move 7: F-bawah
AABCD.
G.BCD.
G..PP. K
GHIIIF
.HJ..F
LLJMMF

Move 8: P-kanan
AABCD.
G.BCD.
G...PP K
GHIIIF
.HJ..F

```
LLJMMF


--- Statistics ---
Algorithm: Greedy Best-First Search
Heuristic: Distance + Blocking
Nodes visited: 14
Time: 9 ms
```

d. Output (A*S)

1. Manhattan Distance

```
Initial Board:
AAB..F
..BCDF
GPPCDF K
GH.III
GHJ...
LLJMM.

Solution: 6 moves.

Move 1: C-atas
AABC.F
..BCDF
GPP.DF K
GH.III
GHJ...
LLJMM.

Move 2: D-atas
AABCDF
..BCDF
GPP..F K
```

GH.III
GHJ...
LLJMM.

Move 3: P-kanan
AABCDF
..BCDF
G..PPF K
GH.III
GHJ...
LLJMM.

Move 4: I-kiri
AABCDF
..BCDF
G..PPF K
GHIII.
GHJ...
LLJMM.

Move 5: F-bawah
AABCD.
..BCD.
G..PP. K
GHIIIF
GHJ..F
LLJMMF

Move 6: P-kanan
AABCD.
..BCD.
G...PP K
GHIIIF

```
GHJ..F
LLJMMF

--- Statistics ---
Algorithm: A* Search
Heuristic: Manhattan Distance
Nodes visited: 142
Time: 17 ms
```

2. Blocking Piece

```
Initial Board:
AAB..F
..BCDF
GPPCDF K
GH.III
GHJ...
LLJMM.

Solution: 5 moves.

Move 1: C-atas
AABC.F
..BCDF
GPP.DF K
GH.III
GHJ...
LLJMM.

Move 2: D-atas
AABCDF
..BCDF
GPP..F K
```

GH.III
GHJ...
LLJMM.

Move 3: I-kiri
AABCDF
..BCDF
GPP..F K
GHIII.
GHJ...
LLJMM.

Move 4: F-bawah
AABCD.
..BCD.
GPP... K
GHIIIF
GHJ..F
LLJMMF

Move 5: P-kanan
AABCD.
..BCD.
G...PP K
GHIIIF
GHJ..F
LLJMMF

--- Statistics ---
Algorithm: A* Search
Heuristic: Blocking Pieces
Nodes visited: 92
Time: 17 ms

3. **Distance + Blocking**

Initial Board:
AAB..F
..BCDF
GPPCDF K
GH.III
GHJ...
LLJMM.

Solution: 6 moves.

Move 1: D-atas
AAB.DF
..BCDF
GPPC.F K
GH.III
GHJ...
LLJMM.

Move 2: C-atas
AABCDF
..BCDF
GPP..F K
GH.III
GHJ...
LLJMM.

Move 3: P-kanan
AABCDF
..BCDF
G..PPF K
GH.III

GHJ...
LLJMM.

Move 4: I-kiri
AABCDF
..BCDF
G..PPF K
GHIII.
GHJ...
LLJMM.

Move 5: F-bawah
AABCD.
..BCD.
G..PP. K
GHIIIF
GHJ..F
LLJMMF

Move 6: P-kanan
AABCD.
..BCD.
G...PP K
GHIIIF
GHJ..F
LLJMMF

--- Statistics ---
Algorithm: A* Search
Heuristic: Distance + Blocking
Nodes visited: 16
Time: 10 ms

e. Output (IDA*S)

1. Manhattan Distance

```
Initial Board:
AAB..F
..BCDF
GPPCDF K
GH.III
GHJ...
LLJMM.

Solution: 6 moves.

Move 1: C-atas
AABC.F
..BCDF
GPP.DF K
GH.III
GHJ...
LLJMM.

Move 2: D-atas
AABCDF
..BCDF
GPP..F K
GH.III
GHJ...
LLJMM.

Move 3: P-kanan
AABCDF
..BCDF
G..PPF K
GH.III
```

GHJ...
LLJMM.

Move 4: I-kiri
AABCDF
..BCDF
G..PPF K
GHIII.
GHJ...
LLJMM.

Move 5: F-bawah
AABCD.
..BCD.
G..PP. K
GHIIIF
GHJ..F
LLJMMF

Move 6: P-kanan
AABCD.
..BCD.
G...PP K
GHIIIF
GHJ..F
LLJMMF

--- Statistics ---
Algorithm: Iterative Deepening A*
Heuristic: Manhattan Distance
Nodes visited: 889
Time: 14 ms

2. Blocking Piece

Initial Board:

AAB..F

..BCDF

GPPCDF K

GH.III

GHJ...

LLJMM.

Solution: 5 moves.

Move 1: C-atas

AABC.F

..BCDF

GPP.DF K

GH.III

GHJ...

LLJMM.

Move 2: D-atas

AABCDF

..BCDF

GPP..F K

GH.III

GHJ...

LLJMM.

Move 3: I-kiri

AABCDF

..BCDF

GPP..F K

GHIII.

GHJ...

```
LLJMM.

Move 4: F-bawah
AABCD.
..BCD.
GPP... K
GHIIIF
GHJ..F
LLJMMF

Move 5: P-kanan
AABCD.
..BCD.
G...PP K
GHIIIF
GHJ..F
LLJMMF

--- Statistics ---
Algorithm: Iterative Deepening A*
Heuristic: Blocking Pieces
Nodes visited: 674
Time: 12 ms
```

3. Distance + Blocking

```
Initial Board:
AAB..F
..BCDF
GPPCDF K
GH.III
GHJ...
LLJMM.
```

Solution: 6 moves.

Move 1: C-atas

AABC.F

..BCDF

GPP.DF K

GH.III

GHJ...

LLJMM.

Move 2: D-atas

AABCDF

..BCDF

GPP..F K

GH.III

GHJ...

LLJMM.

Move 3: P-kanan

AABCDF

..BCDF

G..PPF K

GH.III

GHJ...

LLJMM.

Move 4: I-kiri

AABCDF

..BCDF

G..PPF K

GHIII.

GHJ...

LLJMM.

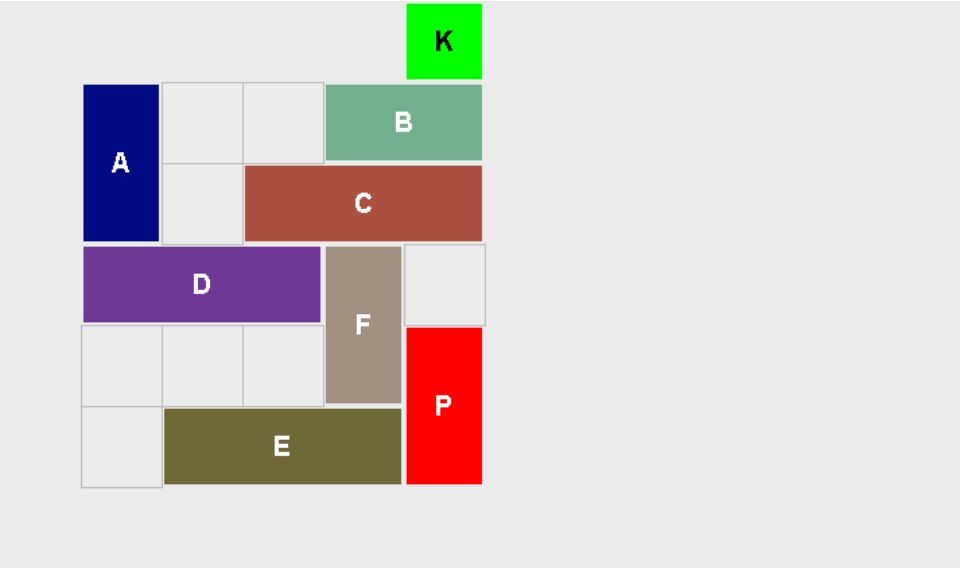
Move 5: F-bawah
AABCD.
..BCD.
G..PP. K
GHIIIF
GHJ..F
LLJMMF

Move 6: P-kanan
AABCD.
..BCD.
G...PP K
GHIIIF
GHJ..F
LLJMMF

--- Statistics ---
Algorithm: Iterative Deepening A*
Heuristic: Distance + Blocking
Nodes visited: 50
Time: 4 ms

4.1.2 Test Case 2

a. Input



Gambar 4. Initial State Test Case 2

```
5 5
6
  K
A..BB
A.CCC
DDDF.
...FP
.EEEP
```

b. Output (UCS)

```
Initial Board:
A..BB
A.CCC
DDDF.
...FP
.EEEP
```

Solution: 3 moves.

Move 1: B-kiri
ABB..
A.CCC
DDDF.
...FP
.EEEP

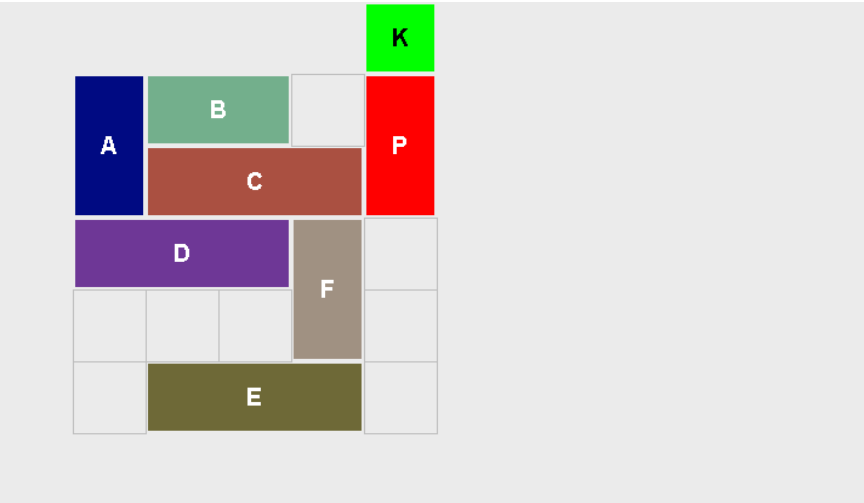
Move 2: C-kiri
ABB..
ACCC.
DDDF.
...FP
.EEEP

Move 3: P-atas
ABB.P
ACCCP
DDDF.
...F.
.EEE.

--- Statistics ---
Algorithm: Uniform Cost Search
Nodes visited: 35
Time: 7 ms

c. Output (GBFS)

1. **Manhattan Distance**



Gambar 5. Final State Test Case 2 dengan GBFS + MD

```
Initial Board:
A..BB
A.CCC
DDDF.
...FP
.EEEP

Solution: 5 moves.

Move 1: P-atas
A..BB
A.CCC
DDDFP
...FP
.EEE.

Move 2: C-kiri
A..BB
ACCC.
DDDFP
```

```
...FP
.EEE.

Move 3: P-atas
A..BB
ACCCP
DDDFP
...F.
.EEE.

Move 4: B-kiri
ABB..
ACCCP
DDDFP
...F.
.EEE.

Move 5: P-atas
ABB.P
ACCCP
DDDF.
...F.
.EEE.

--- Statistics ---
Algorithm: Greedy Best-First Search
Heuristic: Manhattan Distance
Nodes visited: 8
Time: 4 ms
```

2. Blocking Piece

```
Initial Board:
```

A..BB
A.CCC
DDDF.
...FP
.EEEP

Solution: 3 moves.

Move 1: B-kiri
ABB..
A.CCC
DDDF.
...FP
.EEEP

Move 2: C-kiri
ABB..
ACCC.
DDDF.
...FP
.EEEP

Move 3: P-atas
ABB.P
ACCCP
DDDF.
...F.
.EEE.

--- Statistics ---
Algorithm: Greedy Best-First Search
Heuristic: Blocking Pieces
Nodes visited: 4

Time: 6 ms

3. Distance + Blocking

Initial Board:
A..BB
A.CCC
DDDF.
...FP
.EEEP

Solution: 4 moves.

Move 1: P-atas
A..BB
A.CCC
DDDFP
...FP
.EEE.

Move 2: B-kiri
ABB..
A.CCC
DDDFP
...FP
.EEE.

Move 3: C-kiri
ABB..
ACCC.
DDDFP
...FP
.EEE.

```
Move 4: P-atas
ABB.P
ACCCP
DDDF.
...F.
.EEE.

--- Statistics ---
Algorithm: Greedy Best-First Search
Heuristic: Distance + Blocking
Nodes visited: 5
Time: 5 ms
```

d. Output (A*S)

1. Manhattan Distance

```
Initial Board:
A..BB
A.CCC
DDDF.
...FP
.EEEP

Solution: 4 moves.

Move 1: C-kiri
A..BB
ACCC.
DDDF.
...FP
.EEEP
```

```
Move 2: P-atas
A..BB
ACCCP
DDDFP
...F.
.EEE.

Move 3: B-kiri
ABB..
ACCCP
DDDFP
...F.
.EEE.

Move 4: P-atas
ABB.P
ACCCP
DDDF.
...F.
.EEE.

--- Statistics ---
Algorithm: A* Search
Heuristic: Manhattan Distance
Nodes visited: 13
Time: 5 ms
```

2. Blocking Piece

```
Initial Board:
A..BB
A.CCC
DDDF.
```

```
...FP
.EEEP

Solution: 3 moves.

Move 1: B-kiri
ABB..
A.CCC
DDDF.
...FP
.EEEP

Move 2: C-kiri
ABB..
ACCC.
DDDF.
...FP
.EEEP

Move 3: P-atas
ABB.P
ACCCP
DDDF.
...F.
.EEE.

--- Statistics ---
Algorithm: A* Search
Heuristic: Blocking Pieces
Nodes visited: 7
Time: 6 ms
```

3. **Distance + Blocking**

Initial Board:
A..BB
A.CCC
DDDF.
...FP
.EEEP

Solution: 3 moves.

Move 1: B-kiri
ABB..
A.CCC
DDDF.
...FP
.EEEP

Move 2: C-kiri
ABB..
ACCC.
DDDF.
...FP
.EEEP

Move 3: P-atas
ABB.P
ACCCP
DDDF.
...F.
.EEE.

--- Statistics ---
Algorithm: A* Search
Heuristic: Distance + Blocking

Nodes visited: 7

Time: 5 ms

e. Output (IDA*S)

1. Manhattan Distance

Initial Board:

A..BB

A.CCC

DDDF.

...FP

.EEEP

Solution: 4 moves.

Move 1: C-kiri

A..BB

ACCC.

DDDF.

...FP

.EEEP

Move 2: P-atas

A..BB

ACCCP

DDDFP

...F.

.EEE.

Move 3: B-kiri

ABB..

ACCCP

DDDFP

```
...F.
.EEE.

Move 4: P-atas
ABB.P
ACCCP
DDDF.
...F.
.EEE.

--- Statistics ---
Algorithm: Iterative Deepening A*
Heuristic: Manhattan Distance
Nodes visited: 48
Time: 4 ms
```

2. Blocking Piece

```
Initial Board:
A..BB
A.CCC
DDDF.
...FP
.EEEP

Solution: 3 moves.

Move 1: B-kiri
ABB..
A.CCC
DDDF.
...FP
.EEEP
```

Move 2: C-kiri
ABB..
ACCC.
DDDF.
...FP
.EEEP

Move 3: P-atas
ABB.P
ACCCP
DDDF.
...F.
.EEE.

--- Statistics ---
Algorithm: Iterative Deepening A*
Heuristic: Blocking Pieces
Nodes visited: 55
Time: 4 ms

3. Distance + Blocking

Initial Board:
A..BB
A.CCC
DDDF.
...FP
.EEEP

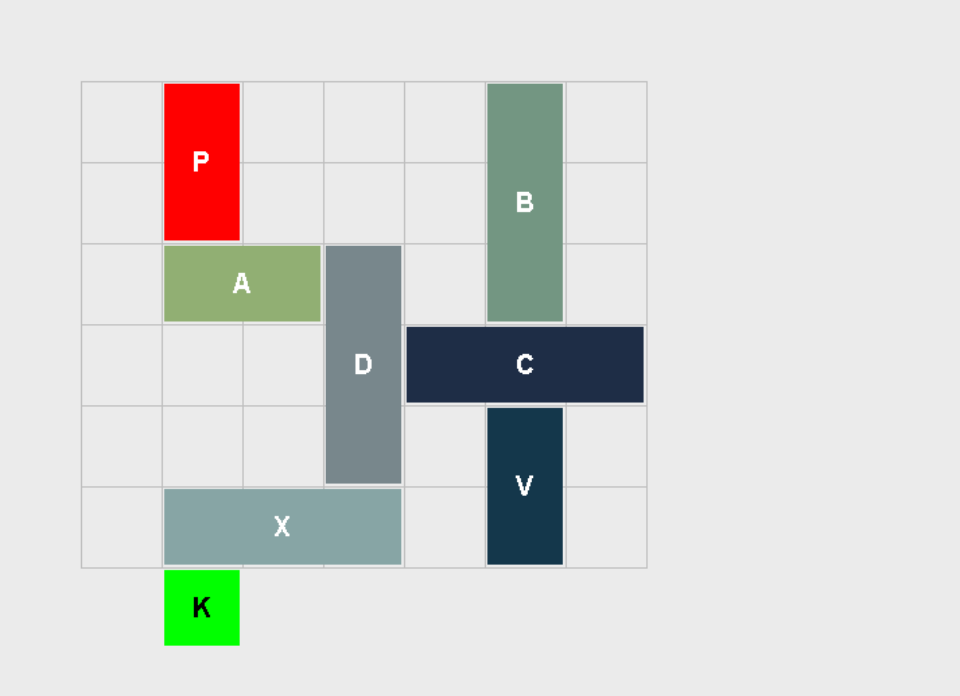
Solution: 4 moves.

Move 1: P-atas

```
A..BB
A.CCC
DDDFP
...FP
.EEE.

Move 2: B-kiri
ABB..
A.CCC
DDDFP
...FP
.EEE.

Move 3: C-kiri
ABB..
ACCC.
DDDFP
...FP
.EEE.

Move 4: P-atas
ABB.P
ACCCP
DDDF.
...F.
.EEE.

--- Statistics ---
Algorithm: Iterative Deepening A*
Heuristic: Distance + Blocking
Nodes visited: 6
Time: 2 ms
```

4.1.3 Test Case 3

a. Input



Gambar 6. Initial State Test Case 3

```
6 7
6
.P...B.
.P...B.
.AAD.B.
...DCCC
...D.V.
.XXX.V.
K
```

b. Output (UCS)

```
Initial Board:
.P...B.
.P...B.
```

.AAD.B.
...DCCC
...D.V.
.XXX.V.
K

Solution: 8 moves.

Move 1: X-kiri
.P...B.
.P...B.
.AAD.B.
...DCCC
...D.V.
XXX..V.
K

Move 2: D-bawah
.P...B.
.P...B.
.AA..B.
...DCCC
...D.V.
XXXD.V.
K

Move 3: A-kanan
.P...B.
.P...B.
...AAB.
...DCCC
...D.V.
XXXD.V.

K

Move 4: P-bawah

.....B.

.....B.

...AAB.

.P.DCCC

.P.D.V.

XXXD.V.

K

Move 5: A-kiri

.....B.

.....B.

AA...B.

.P.DCCC

.P.D.V.

XXXD.V.

K

Move 6: D-atas

...D.B.

...D.B.

AA.D.B.

.P..CCC

.P...V.

XXX..V.

K

Move 7: X-kanan

...D.B.

...D.B.

AA.D.B.

```
.P..CCC
.P...V.
..XXXV.
 K

Move 8: P-bawah
...D.B.
...D.B.
AA.D.B.
....CCC
.P...V.
.PXXXV.
 K

--- Statistics ---
Algorithm: Uniform Cost Search
Nodes visited: 250
Time: 13 ms
```

c. Output (GBFS)

1. Manhattan Distance

```
Initial Board:
.P...B.
.P...B.
.AAD.B.
...DCCC
...D.V.
.XXX.V.
 K

Solution: 11 moves.
```

Move 1: A-kiri

.P...B.

.P...B.

AA.D.B.

...DCCC

...D.V.

.XXX.V.

K

Move 2: X-kanan

.P...B.

.P...B.

AA.D.B.

...DCCC

...D.V.

..XXXV.

K

Move 3: X-kiri

.P...B.

.P...B.

AA.D.B.

...DCCC

...D.V.

XXX..V.

K

Move 4: D-bawah

.P...B.

.P...B.

AA...B.

...DCCC

...D.V.

XXXD.V.
K

Move 5: A-kanan
.P...B.
.P...B.
...AAB.
...DCCC
...D.V.
XXXD.V.
K

Move 6: P-bawah
.....B.
.....B.
...AAB.
.P.DCCC
.P.D.V.
XXXD.V.
K

Move 7: A-kiri
.....B.
.....B.
AA...B.
.P.DCCC
.P.D.V.
XXXD.V.
K

Move 8: D-atas
...D.B.
...D.B.

AA.D.B.
.P..CCC
.P...V.
XXX..V.
K

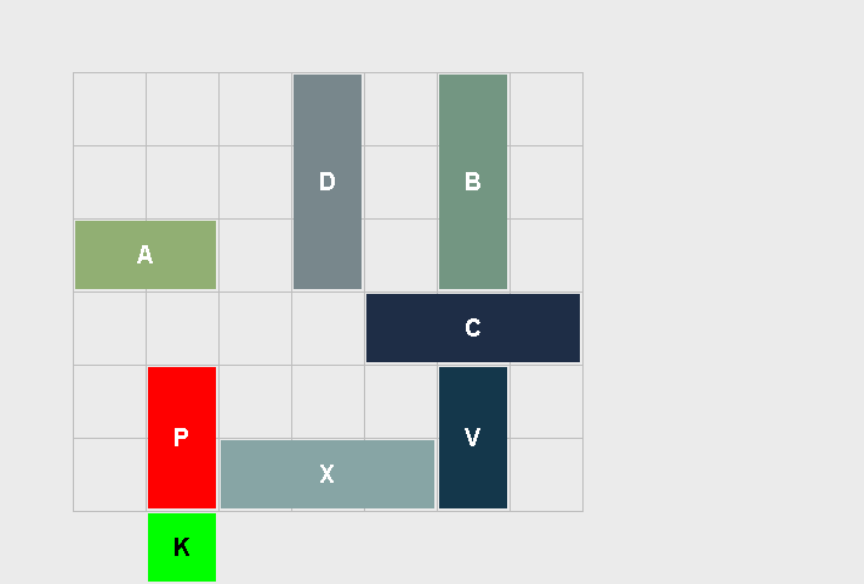
Move 9: A-kanan
...D.B.
...D.B.
.AAD.B.
.P..CCC
.P...V.
XXX..V.
K

Move 10: X-kanan
...D.B.
...D.B.
.AAD.B.
.P..CCC
.P...V.
..XXXV.
K

Move 11: P-bawah
...D.B.
...D.B.
.AAD.B.
....CCC
.P...V.
.PXXXV.
K

```
--- Statistics ---
Algorithm: Greedy Best-First Search
Heuristic: Manhattan Distance
Nodes visited: 197
Time: 18 ms
```

2. Blocking Piece



Gambar 7. Final State Test Case 3 dengan A*S + BP

```
Initial Board:
.P...B.
.P...B.
.AAD.B.
...DCCC
...D.V.
.XXX.V.
K

Solution: 11 moves.

Move 1: X-kanan
```

.P...B.
.P...B.
.AAD.B.
...DCCC
...D.V.
..XXXV.
K

Move 2: A-kiri
.P...B.
.P...B.
AA.D.B.
...DCCC
...D.V.
..XXXV.
K

Move 3: D-atas
.P.D.B.
.P.D.B.
AA.D.B.
....CCC
.....V.
..XXXV.
K

Move 4: X-kiri
.P.D.B.
.P.D.B.
AA.D.B.
....CCC
.....V.
XXX..V.

K

Move 5: D-bawah

.P...B.

.P...B.

AA...B.

...DCCC

...D.V.

XXXD.V.

K

Move 6: A-kanan

.P...B.

.P...B.

...AAB.

...DCCC

...D.V.

XXXD.V.

K

Move 7: P-bawah

.....B.

.....B.

...AAB.

.P.DCCC

.P.D.V.

XXXD.V.

K

Move 8: A-kiri

.....B.

.....B.

AA...B.

.P.DCCC
.P.D.V.
XXXD.V.
K

Move 9: D-atas
...D.B.
...D.B.
AA.D.B.
.P..CCC
.P...V.
XXX..V.
K

Move 10: X-kanan
...D.B.
...D.B.
AA.D.B.
.P..CCC
.P...V.
..XXXV.
K

Move 11: P-bawah
...D.B.
...D.B.
AA.D.B.
....CCC
.P...V.
.PXXXV.
K

--- Statistics ---

Algorithm: Greedy Best-First Search

Heuristic: Blocking Pieces

Nodes visited: 15

Time: 6 ms

3. Distance + Blocking

Initial Board:

.P...B.

.P...B.

.AAD.B.

...DCCC

...D.V.

.XXX.V.

K

Solution: 11 moves.

Move 1: X-kanan

.P...B.

.P...B.

.AAD.B.

...DCCC

...D.V.

..XXXV.

K

Move 2: A-kiri

.P...B.

.P...B.

AA.D.B.

...DCCC

...D.V.

..XXXV.
K

Move 3: D-atas
.P.D.B.
.P.D.B.
AA.D.B.
....CCC
.....V.
..XXXV.
K

Move 4: X-kiri
.P.D.B.
.P.D.B.
AA.D.B.
....CCC
.....V.
XXX..V.
K

Move 5: D-bawah
.P...B.
.P...B.
AA...B.
...DCCC
...D.V.
XXXD.V.
K

Move 6: A-kanan
.P...B.
.P...B.

...AAB.
...DCCC
...D.V.
XXXD.V.
K

Move 7: P-bawah
.....B.
.....B.
...AAB.
.P.DCCC
.P.D.V.
XXXD.V.
K

Move 8: A-kiri
.....B.
.....B.
AA...B.
.P.DCCC
.P.D.V.
XXXD.V.
K

Move 9: D-atas
...D.B.
...D.B.
AA.D.B.
.P..CCC
.P...V.
XXX..V.
K

```
Move 10: X-kanan
...D.B.
...D.B.
AA.D.B.
.P..CCC
.P...V.
..XXXV.
 K

Move 11: P-bawah
...D.B.
...D.B.
AA.D.B.
....CCC
.P...V.
.PXXXV.
 K

--- Statistics ---
Algorithm: Greedy Best-First Search
Heuristic: Distance + Blocking
Nodes visited: 15
Time: 7 ms
```

d. Output (A*S)

1. Manhattan Distance

```
Initial Board:
.P...B.
.P...B.
.AAD.B.
...DCCC
...D.V.
```

.XXX.V.
K

Solution: 8 moves.

Move 1: X-kiri
.P...B.
.P...B.
.AAD.B.
...DCCC
...D.V.
XXX..V.
K

Move 2: D-bawah
.P...B.
.P...B.
.AA..B.
...DCCC
...D.V.
XXXD.V.
K

Move 3: A-kanan
.P...B.
.P...B.
...AAB.
...DCCC
...D.V.
XXXD.V.
K

Move 4: P-bawah

.....B.
.....B.
...AAB.
.P.DCCC
.P.D.V.
XXXD.V.
K

Move 5: A-kiri
.....B.
.....B.
AA...B.
.P.DCCC
.P.D.V.
XXXD.V.
K

Move 6: D-atas
...D.B.
...D.B.
AA.D.B.
.P..CCC
.P...V.
XXX..V.
K

Move 7: X-kanan
...D.B.
...D.B.
AA.D.B.
.P..CCC
.P...V.
..XXXV.

```
 K

Move 8: P-bawah
...D.B.
...D.B.
AA.D.B.
....CCC
.P...V.
.PXXXV.
 K

--- Statistics ---
Algorithm: A* Search
Heuristic: Manhattan Distance
Nodes visited: 64
Time: 10 ms
```

2. Blocking Piece

```
Initial Board:
.P...B.
.P...B.
.AAD.B.
...DCCC
...D.V.
.XXX.V.
 K

Solution: 8 moves.

Move 1: X-kiri
.P...B.
.P...B.
```

.AAD.B.
...DCCC
...D.V.
XXX..V.
K

Move 2: D-bawah
.P...B.
.P...B.
.AA..B.
...DCCC
...D.V.
XXXD.V.
K

Move 3: A-kanan
.P...B.
.P...B.
...AAB.
...DCCC
...D.V.
XXXD.V.
K

Move 4: P-bawah
.....B.
.....B.
...AAB.
.P.DCCC
.P.D.V.
XXXD.V.
K

Move 5: A-kiri

.....B.

.....B.

AA...B.

.P.DCCC

.P.D.V.

XXXD.V.

K

Move 6: D-atas

...D.B.

...D.B.

AA.D.B.

.P..CCC

.P...V.

XXX..V.

K

Move 7: X-kanan

...D.B.

...D.B.

AA.D.B.

.P..CCC

.P...V.

..XXXV.

K

Move 8: P-bawah

...D.B.

...D.B.

AA.D.B.

....CCC

.P...V.

```
.PXXXV.
 K

--- Statistics ---
Algorithm: A* Search
Heuristic: Blocking Pieces
Nodes visited: 128
Time: 11 ms
```

3. Distance + Blocking

```
Initial Board:
.P...B.
.P...B.
.AAD.B.
...DCCC
...D.V.
.XXX.V.
 K

Solution: 8 moves.

Move 1: X-kiri
.P...B.
.P...B.
.AAD.B.
...DCCC
...D.V.
XXX..V.
 K

Move 2: D-bawah
.P...B.
```

.P...B.
.AA..B.
...DCCC
...D.V.
XXXD.V.
K

Move 3: A-kanan
.P...B.
.P...B.
...AAB.
...DCCC
...D.V.
XXXD.V.
K

Move 4: P-bawah
.....B.
.....B.
...AAB.
.P.DCCC
.P.D.V.
XXXD.V.
K

Move 5: A-kiri
.....B.
.....B.
AA...B.
.P.DCCC
.P.D.V.
XXXD.V.
K

Move 6: D-atas

...D.B.

...D.B.

AA.D.B.

.P..CCC

.P...V.

XXX..V.

K

Move 7: X-kanan

...D.B.

...D.B.

AA.D.B.

.P..CCC

.P...V.

..XXXV.

K

Move 8: P-bawah

...D.B.

...D.B.

AA.D.B.

....CCC

.P...V.

.PXXXV.

K

--- Statistics ---

Algorithm: A* Search

Heuristic: Distance + Blocking

Nodes visited: 25

Time: 8 ms

e. **Output (IDA*S)**

1. **Manhattan Distance**

```
Initial Board:
.P...B.
.P...B.
.AAD.B.
...DCCC
...D.V.
.XXX.V.
 K

Solution: 8 moves.

Move 1: X-kiri
.P...B.
.P...B.
.AAD.B.
...DCCC
...D.V.
XXX..V.
 K

Move 2: D-bawah
.P...B.
.P...B.
.AA..B.
...DCCC
...D.V.
XXXD.V.
 K
```

Move 3: A-kanan

.P...B.

.P...B.

...AAB.

...DCCC

...D.V.

XXXD.V.

K

Move 4: P-bawah

.....B.

.....B.

...AAB.

.P.DCCC

.P.D.V.

XXXD.V.

K

Move 5: A-kiri

.....B.

.....B.

AA...B.

.P.DCCC

.P.D.V.

XXXD.V.

K

Move 6: D-atas

...D.B.

...D.B.

AA.D.B.

.P..CCC

```
.P...V.
XXX..V.
 K

Move 7: X-kanan
...D.B.
...D.B.
AA.D.B.
.P..CCC
.P...V.
..XXXV.
 K

Move 8: P-bawah
...D.B.
...D.B.
AA.D.B.
....CCC
.P...V.
.PXXXV.
 K

--- Statistics ---
Algorithm: Iterative Deepening A*
Heuristic: Manhattan Distance
Nodes visited: 1078
Time: 17 ms
```

2. Blocking Piece

```
Initial Board:
.P...B.
.P...B.
```

.AAD.B.
...DCCC
...D.V.
.XXX.V.
K

Solution: 8 moves.

Move 1: X-kiri
.P...B.
.P...B.
.AAD.B.
...DCCC
...D.V.
XXX..V.
K

Move 2: D-bawah
.P...B.
.P...B.
.AA..B.
...DCCC
...D.V.
XXXD.V.
K

Move 3: A-kanan
.P...B.
.P...B.
...AAB.
...DCCC
...D.V.
XXXD.V.

K

Move 4: P-bawah

.....B.

.....B.

...AAB.

.P.DCCC

.P.D.V.

XXXD.V.

K

Move 5: A-kiri

.....B.

.....B.

AA...B.

.P.DCCC

.P.D.V.

XXXD.V.

K

Move 6: D-atas

...D.B.

...D.B.

AA.D.B.

.P..CCC

.P...V.

XXX..V.

K

Move 7: X-kanan

...D.B.

...D.B.

AA.D.B.

```
.P..CCC
.P...V.
..XXXV.
K

Move 8: P-bawah
...D.B.
...D.B.
AA.D.B.
....CCC
.P...V.
.PXXXV.
K

--- Statistics ---
Algorithm: Iterative Deepening A*
Heuristic: Blocking Pieces
Nodes visited: 14633
Time: 74 ms
```

3. **Distance + Blocking**

```
Initial Board:
.P...B.
.P...B.
.AAD.B.
...DCCC
...D.V.
.XXX.V.
K

Solution: 8 moves.
```

Move 1: X-kiri

.P...B.

.P...B.

.AAD.B.

...DCCC

...D.V.

XXX..V.

K

Move 2: D-bawah

.P...B.

.P...B.

.AA..B.

...DCCC

...D.V.

XXXD.V.

K

Move 3: A-kanan

.P...B.

.P...B.

...AAB.

...DCCC

...D.V.

XXXD.V.

K

Move 4: P-bawah

.....B.

.....B.

...AAB.

.P.DCCC

.P.D.V.

XXXD.V.
K

Move 5: A-kiri
.....B.
.....B.
AA...B.
.P.DCCC
.P.D.V.
XXXD.V.
K

Move 6: D-atas
...D.B.
...D.B.
AA.D.B.
.P..CCC
.P...V.
XXX..V.
K

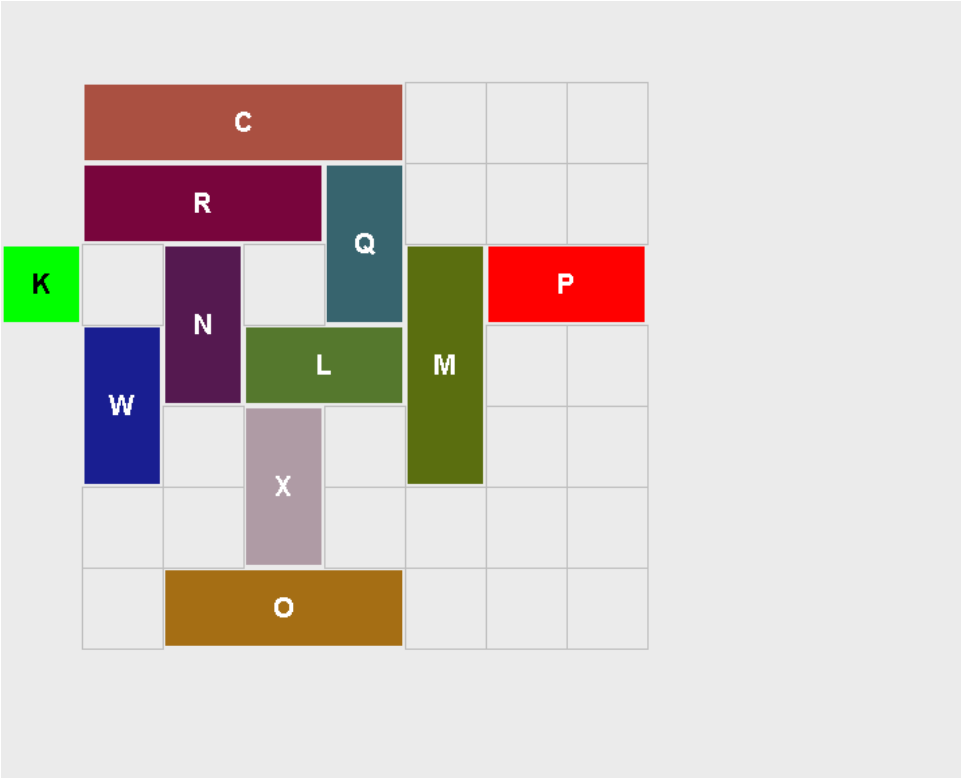
Move 7: X-kanan
...D.B.
...D.B.
AA.D.B.
.P..CCC
.P...V.
..XXXV.
K

Move 8: P-bawah
...D.B.
...D.B.

```
AA.D.B.
....CCC
.P...V.
.PXXXV.
K

--- Statistics ---
Algorithm: Iterative Deepening A*
Heuristic: Distance + Blocking
Nodes visited: 122
Time: 8 ms
```

4.1.4 Test Case 4

a. Input



Gambar 8. Initial State Test Case 4

```
7 7
9
 CCCC...
 RRRQ...
K.N.QMPP
 WNLLM..
 W.X.M..
 ..X....
 .OOO...
```

b. Output (UCS)

```
Initial Board:
CCCC...
RRRQ...
.N.QMPP
WNLLM..
W.X.M..
..X....
.OOO...

Solution: 5 moves.

Move 1: M-bawah
CCCC...
RRRQ...
.N.Q.PP
WNLL...
W.X.M..
..X.M..
.OOOM..
```

Move 2: N-bawah

CCCC...

RRRQ...

...Q.PP

W.LL...

WNX.M..

.NX.M..

.OOOM..

Move 3: L-kiri

CCCC...

RRRQ...

...Q.PP

WLL....

WNX.M..

.NX.M..

.OOOM..

Move 4: Q-bawah

CCCC...

RRR....

.....PP

WLL....

WNXQM..

.NXQM..

.OOOM..

Move 5: P-kiri

CCCC...

RRR....

PP.....

WLL....

WNXQM..

```
.NXQM..
.OOOM..

--- Statistics ---
Algorithm: Uniform Cost Search
Nodes visited: 977
Time: 42 ms
```

c. Output (GBFS)

1. Manhattan Distance

```
Initial Board:
CCCC...
RRRQ...
.N.QMPP
WNLLM..
W.X.M..
..X....
.OOO...

Solution: 17 moves.

Move 1: O-kanan
CCCC...
RRRQ...
.N.QMPP
WNLLM..
W.X.M..
..X....
....OOO

Move 2: O-kiri
CCCC...
```

RRRQ...
.N.QMPP
WNLLM..
W.X.M..
..X....
OOO....

Move 3: N-bawah
CCCC...
RRRQ...
...QMPP
W.LLM..
WNX.M..
.NX....
OOO....

Move 4: O-kanan
CCCC...
RRRQ...
...QMPP
W.LLM..
WNX.M..
.NX....
....OOO

Move 5: N-bawah
CCCC...
RRRQ...
...QMPP
W.LLM..
W.X.M..
.NX....
.N..OOO

Move 6: O-kiri

CCCC...

RRRQ...

...QMPP

W.LLM..

W.X.M..

.NX....

.NOOO..

Move 7: N-atas

CCCC...

RRRQ...

.N.QMPP

WNLLM..

W.X.M..

..X....

..OOO..

Move 8: M-bawah

CCCC...

RRRQ...

.N.Q.PP

WNLLM..

W.X.M..

..X.M..

..OOO..

Move 9: P-kiri

CCCC...

RRRQ...

.N.QPP.

WNLLM..

W.X.M..
..X.M..
..OOO..

Move 10: O-kanan
CCCC...
RRRQ...
.N.QPP.
WNLLM..
W.X.M..
..X.M..
....OOO

Move 11: X-bawah
CCCC...
RRRQ...
.N.QPP.
WNLLM..
W...M..
..X.M..
..X.OOO

Move 12: O-kiri
CCCC...
RRRQ...
.N.QPP.
WNLLM..
W...M..
..X.M..
..XOOO.

Move 13: C-kanan
...CCCC

RRRQ...
.N.QPP.
WNLLM..
W...M..
..X.M..
..XOOO.

Move 14: N-bawah
...CCCC
RRRQ...
...QPP.
W.LLM..
W...M..
.NX.M..
.NXOOO.

Move 15: L-kiri
...CCCC
RRRQ...
...QPP.
WLL.M..
W...M..
.NX.M..
.NXOOO.

Move 16: Q-bawah
...CCCC
RRR....
....PP.
WLL.M..
W..QM..
.NXQM..
.NXOOO.

```
Move 17: P-kiri
...CCCC
RRR....
PP.....
WLL.M..
W..QM..
.NXQM..
.NXOOO.

--- Statistics ---
Algorithm: Greedy Best-First Search
Heuristic: Manhattan Distance
Nodes visited: 72
Time: 14 ms
```

2. Blocking Piece

```
Initial Board:
CCCC...
RRRQ...
.N.QMPP
WNLLM..
W.X.M..
..X....
.OOO...

Solution: 5 moves.

Move 1: M-bawah
CCCC...
RRRQ...
.N.Q.PP
```

WNLL...
W.X.M..
..X.M..
.OOOM..

Move 2: N-bawah
CCCC...
RRRQ...
...Q.PP
W.LL...
WNX.M..
.NX.M..
.OOOM..

Move 3: L-kiri
CCCC...
RRRQ...
...Q.PP
WLL....
WNX.M..
.NX.M..
.OOOM..

Move 4: Q-bawah
CCCC...
RRR....
.....PP
WLL....
WNXQM..
.NXQM..
.OOOM..

Move 5: P-kiri

```
CCCC...
RRR....
PP.....
WLL....
WNXQM..
.NXQM..
.OOOM..

--- Statistics ---
Algorithm: Greedy Best-First Search
Heuristic: Blocking Pieces
Nodes visited: 9
Time: 5 ms
```

3. **Distance + Blocking**

```
Initial Board:
CCCC...
RRRQ...
.N.QMPP
WNLLM..
W.X.M..
..X....
.OOO...

Solution: 7 moves.

Move 1: M-bawah
CCCC...
RRRQ...
.N.Q.PP
WNLL...
W.X.M..
```

..X.M..
.OOOM..

Move 2: P-kiri
CCCC...
RRRQ...
.N.QPP.
WNLL...
W.X.M..
..X.M..
.OOOM..

Move 3: N-bawah
CCCC...
RRRQ...
...QPP.
W.LL...
WNX.M..
.NX.M..
.OOOM..

Move 4: C-kanan
...CCCC
RRRQ...
...QPP.
W.LL...
WNX.M..
.NX.M..
.OOOM..

Move 5: L-kanan
...CCCC
RRRQ...

```
...QPP.
W....LL
WNX.M..
.NX.M..
.OOOM..

Move 6: Q-bawah
...CCCC
RRR....
....PP.
W....LL
WNXQM..
.NXQM..
.OOOM..

Move 7: P-kiri
...CCCC
RRR....
PP.....
W....LL
WNXQM..
.NXQM..
.OOOM..

--- Statistics ---
Algorithm: Greedy Best-First Search
Heuristic: Distance + Blocking
Nodes visited: 9
Time: 8 ms
```

d. Output (A*S)

1. Manhattan Distance

Initial Board:

CCCC...

RRRQ...

.N.QMPP

WNLLM..

W.X.M..

..X....

.OOO...

Solution: 6 moves.

Move 1: M-bawah

CCCC...

RRRQ...

.N.Q.PP

WNLL...

W.X.M..

..X.M..

.OOOM..

Move 2: L-kanan

CCCC...

RRRQ...

.N.Q.PP

WN...LL

W.X.M..

..X.M..

.OOOM..

Move 3: Q-bawah

CCCC...

RRR....

.N...PP

```
WN...LL
W.XQM..
..XQM..
.OOOM..

Move 4: P-kiri
CCCC...
RRR....
.NPP...
WN...LL
W.XQM..
..XQM..
.OOOM..

Move 5: N-bawah
CCCC...
RRR....
..PP...
W....LL
WNXQM..
.NXQM..
.OOOM..

Move 6: P-kiri
CCCC...
RRR....
PP.....
W....LL
WNXQM..
.NXQM..
.OOOM..

--- Statistics ---
```

Algorithm: A* Search

Heuristic: Manhattan Distance

Nodes visited: 116

Time: 12 ms

2. Blocking Piece

Initial Board:

CCCC...

RRRQ...

.N.QMPP

WNLLM..

W.X.M..

..X....

.OOO...

Solution: 5 moves.

Move 1: N-bawah

CCCC...

RRRQ...

...QMPP

W.LLM..

WNX.M..

.NX....

.OOO...

Move 2: L-kiri

CCCC...

RRRQ...

...QMPP

WLL.M..

WNX.M..

```
.NX....
.OOO...

Move 3: Q-bawah
CCCC...
RRR....
....MPP
WLL.M..
WNXQM..
.NXQ...
.OOO...

Move 4: M-bawah
CCCC...
RRR....
.....PP
WLL....
WNXQM..
.NXQM..
.OOOM..

Move 5: P-kiri
CCCC...
RRR....
PP.....
WLL....
WNXQM..
.NXQM..
.OOOM..

--- Statistics ---
Algorithm: A* Search
Heuristic: Blocking Pieces
```

```
Nodes visited: 70
Time: 15 ms
```

3. Distance + Blocking

```
Initial Board:
CCCC...
RRRQ...
.N.QMPP
WNLLM..
W.X.M..
..X....
.OOO...

Solution: 5 moves.

Move 1: N-bawah
CCCC...
RRRQ...
...QMPP
W.LLM..
WNX.M..
.NX....
.OOO...

Move 2: M-bawah
CCCC...
RRRQ...
...Q.PP
W.LL...
WNX.M..
.NX.M..
.OOOM..
```

Move 3: L-kiri
CCCC...
RRRQ...
...Q.PP
WLL....
WNX.M..
.NX.M..
.OOOM..

Move 4: Q-bawah
CCCC...
RRR....
.....PP
WLL....
WNXQM..
.NXQM..
.OOOM..

Move 5: P-kiri
CCCC...
RRR....
PP.....
WLL....
WNXQM..
.NXQM..
.OOOM..

--- Statistics ---
Algorithm: A* Search
Heuristic: Distance + Blocking
Nodes visited: 21
Time: 10 ms

e. Output (IDA*S)

1. Manhattan Distance

```
Initial Board:
CCCC...
RRRQ...
.N.QMPP
WNLLM..
W.X.M..
..X....
.OOO...

Solution: 7 moves.

Move 1: M-bawah
CCCC...
RRRQ...
.N.Q.PP
WNLL...
W.X.M..
..X.M..
.OOOM..

Move 2: P-kiri
CCCC...
RRRQ...
.N.QPP.
WNLL...
W.X.M..
..X.M..
.OOOM..

Move 3: L-kanan
```

CCCC...
RRRQ...
.N.QPP.
WN...LL
W.X.M..
..X.M..
.OOOM..

Move 4: Q-bawah
CCCC...
RRR....
.N..PP.
WN...LL
W.XQM..
..XQM..
.OOOM..

Move 5: P-kiri
CCCC...
RRR....
.NPP...
WN...LL
W.XQM..
..XQM..
.OOOM..

Move 6: N-bawah
CCCC...
RRR....
..PP...
W....LL
WNXQM..
.NXQM..

```
.OOOM..

Move 7: P-kiri
CCCC...
RRR....
PP.....
W....LL
WNXQM..
.NXQM..
.OOOM..

--- Statistics ---
Algorithm: Iterative Deepening A*
Heuristic: Manhattan Distance
Nodes visited: 2713
Time: 22 ms
```

2. Blocking Piece

```
Initial Board:
CCCC...
RRRQ...
.N.QMPP
WNLLM..
W.X.M..
..X....
.OOO...

Solution: 5 moves.

Move 1: M-bawah
CCCC...
RRRQ...
```

.N.Q.PP
WNLL...
W.X.M..
..X.M..
.OOOM..

Move 2: L-kanan
CCCC...
RRRQ...
.N.Q.PP
WN...LL
W.X.M..
..X.M..
.OOOM..

Move 3: Q-bawah
CCCC...
RRR....
.N...PP
WN...LL
W.XQM..
..XQM..
.OOOM..

Move 4: N-bawah
CCCC...
RRR....
.....PP
W....LL
WNXQM..
.NXQM..
.OOOM..

Move 5: P-kiri

CCCC...

RRR....

PP.....

W....LL

WNXQM..

.NXQM..

.OOOM..

--- Statistics ---

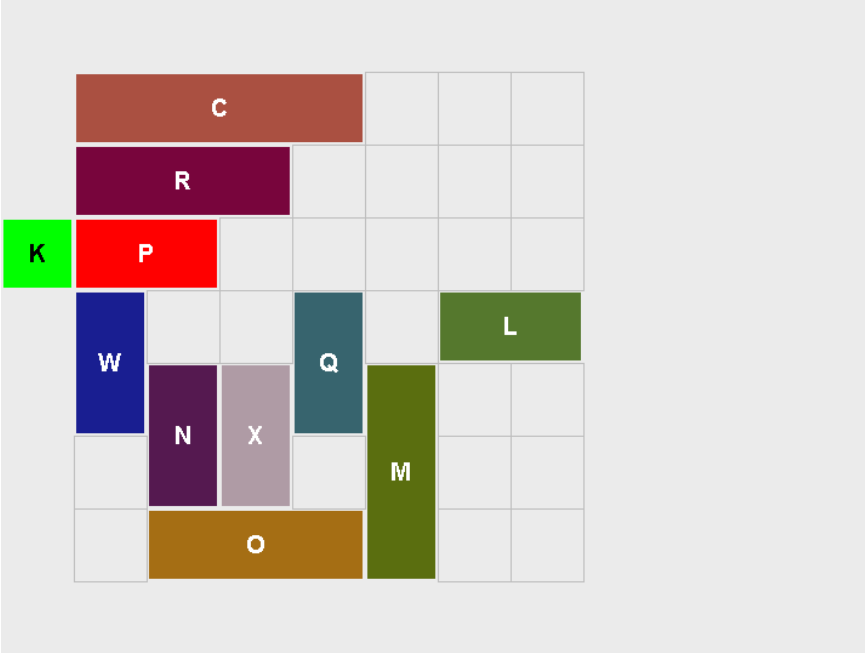
Algorithm: Iterative Deepening A*

Heuristic: Blocking Pieces

Nodes visited: 2980

Time: 31 ms

3. Distance + Blocking



Gambar 9. Final State Test Case 4 dengan IDA*S + DB

Initial Board:

CCCC...
RRRQ...
.N.QMPP
WNLLM..
W.X.M..
..X....
.OOO...

Solution: 8 moves.

Move 1: M-bawah
CCCC...
RRRQ...
.N.Q.PP
WNLL...
W.X.M..
..X.M..
.OOOM..

Move 2: P-kiri
CCCC...
RRRQ...
.N.QPP.
WNLL...
W.X.M..
..X.M..
.OOOM..

Move 3: L-kanan
CCCC...
RRRQ...
.N.QPP.
WN...LL

W.X.M..
..X.M..
.OOOM..

Move 4: Q-bawah
CCCC...
RRR....
.N..PP.
WN...LL
W.XQM..
..XQM..
.OOOM..

Move 5: P-kiri
CCCC...
RRR....
.NPP...
WN...LL
W.XQM..
..XQM..
.OOOM..

Move 6: Q-atas
CCCC...
RRR....
.NPP...
WN.Q.LL
W.XQM..
..X.M..
.OOOM..

Move 7: N-bawah
CCCC...

```
RRR....
..PP...
W..Q.LL
WNXQM..
.NX.M..
.OOOM..

Move 8: P-kiri
CCCC...
RRR....
PP.....
W..Q.LL
WNXQM..
.NX.M..
.OOOM..

--- Statistics ---
Algorithm: Iterative Deepening A*
Heuristic: Distance + Blocking
Nodes visited: 319
Time: 10 ms
```

4.2 Analisis dan Pembahasan

4.2.1 Analisis Kinerja Algoritma (Kompleksitas)

Kompleksitas Waktu

Kompleksitas waktu suatu algoritma sangat bergantung pada karakteristik struktur masalah dan cara algoritma tersebut mengeksplorasi ruang solusi. Pada permainan *Rush Hour*, kompleksitas waktu dari algoritma pencarian dipengaruhi oleh berbagai faktor, seperti jumlah state yang perlu dieksplorasi dan cara algoritma tersebut mengatur pencariannya. Berikut

adalah analisis kompleksitas waktu untuk masing-masing algoritma yang diterapkan pada masalah ini:

1. **Uniform Cost Search (UCS)**

   UCS bekerja dengan cara mengeksplorasi state berdasarkan biaya kumulatif yang terendah (g(n)), dengan menggunakan antrian prioritas. Proses eksplorasi dilakukan secara sistematis dan sangat bergantung pada faktor percabangan dan kedalaman solusi.

   Kompleksitas waktu UCS adalah O(b^d), dengan keterangan:

   - *b* adalah faktor percabangan rata-rata, yaitu jumlah langkah yang mungkin dari setiap state.
   - *d* adalah kedalaman dari solusi terdekat.

   Pada implementasi ini, UCS harus memproses semua langkah valid pada setiap state, yang membutuhkan waktu O(n), di mana *n* adalah jumlah mobil yang terlibat dalam permainan. Selain itu, operasi pada antrian prioritas memerlukan waktu O(log m), di mana *m* adalah jumlah node dalam antrian tersebut. HashSet digunakan untuk menyimpan state yang telah dikunjungi, yang memungkinkan pengecekan keberadaan state dalam waktu O(1).

   Meskipun UCS menjamin solusi optimal, algoritma ini bisa sangat tidak efisien pada puzzle dengan ruang state yang besar karena harus mengeksplorasi banyak node sebelum menemukan solusi.

2. **Greedy Best-First Search (GBFS)**

   Berbeda dengan UCS, GBFS memilih jalur pencarian berdasarkan fungsi heuristik (h(n)) yang mengestimasi seberapa dekat sebuah state menuju solusi. GBFS tidak mempertimbangkan biaya kumulatif, hanya berfokus pada penurunan nilai heuristik untuk memperkirakan langkah selanjutnya.

   Kompleksitas waktu GBFS adalah O(b^m), dengan keterangan:

   - *b* adalah faktor percabangan.
   - *m* adalah kedalaman maksimum dari ruang pencarian.

Keunggulan utama dari GBFS adalah kecepatannya, terutama jika heuristik yang digunakan sangat informatif. Namun, GBFS tidak selalu memberikan solusi optimal karena hanya mengandalkan heuristik dan mengabaikan biaya kumulatif yang dibutuhkan untuk mencapai solusi. Pada beberapa kasus, jika heuristiknya tidak cukup baik atau hampir tidak informatif, GBFS dapat berperilaku hampir acak, dan mengunjungi banyak state sebelum akhirnya menemukan solusi.

3. **A Star Search (A*S)**

   A* menggabungkan dua elemen utama: biaya kumulatif dari langkah sebelumnya (g(n)) dan estimasi biaya sisa menuju tujuan (h(n)). Dengan demikian, A* menggunakan fungsi penilaian f(n) = g(n) + h(n) untuk menentukan langkah selanjutnya dalam pencarian.

   Kompleksitas waktu A*S adalah O(b^d), dengan keterangan:

   - *b* adalah faktor percabangan.
   - *d* adalah kedalaman solusi yang dicapai.

   A* dijamin untuk menemukan solusi optimal jika heuristik yang digunakan admissible (tidak pernah melebih-lebihkan biaya sebenarnya). Dengan heuristik yang baik, A* dapat menghindari eksplorasi banyak state yang kurang menjanjikan, membuatnya lebih efisien dibandingkan UCS dalam hal pencarian solusi optimal. Namun, A* juga menghadapi kendala penggunaan memori yang tinggi, seperti halnya UCS.

4. **Iterative Deepening A Star Search (IDA*S)**

   IDAS mengkombinasikan pencarian A* dengan teknik iterative deepening untuk menghemat penggunaan memori. Pendekatannya mengurangi ruang memori dengan mengevaluasi kembali state dalam setiap iterasi dengan kedalaman yang semakin meningkat.

   Kompleksitas waktu IDA*S adalah O(b^d), dengan keterangan:

   - *b* adalah faktor percabangan.
   - *d* adalah kedalaman solusi.

Meskipun IDAS mengurangi penggunaan memori secara signifikan dibandingkan A*, waktu komputasinya seringkali lebih lama karena perlu mengevaluasi ulang banyak state pada setiap iterasi. Terutama pada puzzle yang lebih kompleks, IDAS bisa memerlukan lebih banyak waktu untuk menyelesaikan pencarian, meskipun penggunaan memori yang lebih efisien dapat menjadi keuntungan utama.

Kompleksitas Ruang

1. **UCS**

   O(b^d) – Menyimpan semua node yang dieksplorasi dalam memori.

2. **GBFS**

   O(b^d) – Sama seperti UCS dalam hal penggunaan memori, menyimpan semua state yang dijelajahi.

3. **A*S**

   O(b^d) – Seperti UCS, A* juga memerlukan memori untuk menyimpan semua node yang dijelajahi.

4. **IDA*S**

   O(d) – Penggunaan memori yang efisien, hanya menyimpan jalur saat ini yang sedang dieksplorasi.

Hasil Eksperimen

1. **UCS**

   Algoritma ini menjelajahi jumlah node yang sangat banyak pada puzzle yang lebih kompleks, yang sesuai dengan prediksi teoretis.

2. **GBFS**

   Meskipun sangat cepat, sering kali menghasilkan solusi suboptimal, terutama pada puzzle yang lebih rumit.

3. **A*S**

   Algoritma ini dengan heuristik Distance + Blocking (BD) memberikan keseimbangan terbaik antara kecepatan pencarian dan kualitas solusi.

4. **IDA*S**

   Penggunaan memori yang minimal tetapi lebih lambat dibandingkan

dengan A*, terutama pada puzzle yang membutuhkan banyak langkah untuk diselesaikan.

4.2.2 Pengaruh Heuristik terhadap Kinerja Algoritma

Pemilihan heuristik memainkan peran yang sangat penting dalam kinerja algoritma pencarian, terutama untuk GBFS dan A*S. Dalam implementasi Rush Hour Solver, tiga jenis heuristik diujikan:

1. **Manhattan Distance (MD)**

   Heuristik ini mengukur jarak Manhattan antara posisi mobil utama dan pintu keluar. Meskipun komputasi sangat cepat, MD tidak memperhitungkan mobil penghalang, sehingga sering kali tidak cukup informatif, terutama pada puzzle dengan banyak hambatan.

   - **Kelebihan**

     Sangat cepat (O(1)) dan admissible.
   - **Kekurangan**

     Tidak memperhitungkan penghalang dan tidak informatif pada puzzle yang padat.

2. **Blocking Pieces (BP)**

   BP menghitung jumlah mobil penghalang yang harus dipindahkan agar mobil utama bisa mencapai pintu keluar. Heuristik ini sangat informatif dalam puzzle dengan banyak penghalang.

   - **Kelebihan**

     Memahami kemacetan dan hambatan yang ada, memberikan wawasan tentang kompleksitas puzzle.
   - **Kekurangan**

     Tidak memperhitungkan jarak ke tujuan dan bisa terlalu menyederhanakan masalah.

3. **Distance + Blocking (DB)**

   DB menggabungkan keunggulan MD dan BP, memberikan gambaran yang lebih lengkap tentang jarak dan penghalang dalam puzzle.

- **Kelebihan**

  Kombinasi informasi yang lebih baik antara jarak dan penghalang.
- **Kekurangan**

  Lebih mahal dalam komputasi dan bisa menjadi non-admissible jika penjumlahan heuristik menyebabkan overestimasi.

Admisibilitas Heuristik

1. **MD**

   Admissible, karena selalu memberikan estimasi yang tidak lebih besar dari biaya sebenarnya.
2. **BP**

   Admissible dalam beberapa kondisi, namun bisa non-admissible pada kasus tertentu jika penghalang memerlukan lebih dari satu langkah untuk dipindahkan.
3. **DB**

   Biasanya non-admissible karena gabungan dari dua heuristik yang salah satunya dapat menyebabkan overestimasi.

Hasil Eksperimental

1. **MD**

   Efektif pada puzzle sederhana, tetapi kurang efisien pada puzzle dengan banyak penghalang.
2. **BP**

   Sangat efektif pada puzzle dengan penghalang yang padat, tetapi tidak selalu memberikan solusi terbaik pada puzzle dengan penghalang yang lebih tersebar.
3. **DB**

   Secara konsisten memberikan hasil terbaik, mengurangi jumlah node yang dikunjungi dan sering kali memberikan solusi optimal.

4.2.3 Perbandingan Kinerja Antar Algoritma

Dari berbagai eksperimen, berikut adalah kesimpulan terkait perbandingan kinerja algoritma pencarian dalam menyelesaikan puzzle Rush Hour:

1. **Optimalitas Solusi**
    - **UCS**

      Selalu menemukan solusi optimal.
    - **GBFS**

      Cenderung menghasilkan solusi suboptimal.
    - **A*S**

      Dengan heuristik admissible, A* selalu menemukan solusi optimal. Heuristik DB bahkan sering menghasilkan solusi optimal meskipun tidak admissible.
    - **IDA*S**

      Menemukan solusi optimal, meskipun lebih lambat pada puzzle yang lebih kompleks.
2. **Efisiensi Komputasi**
    - **UCS**

      Sangat tidak efisien pada puzzle besar.
    - **GBFS**

      Algoritma tercepat tetapi dengan kemungkinan solusi suboptimal.
    - **A*S**

      Menawarkan keseimbangan yang baik antara efisiensi dan optimalitas.
    - **IDA*S**

      Menggunakan memori secara efisien, tetapi lebih lambat pada puzzle yang rumit.
3. **Penggunaan Memori**

- **UCS, GBFS, dan A*S**
  Memerlukan banyak memori untuk menyimpan state yang telah dijelajahi.
- **IDA*S**
  Menggunakan memori lebih efisien, hanya menyimpan jalur yang sedang dieksplorasi.

Hasil Pengujian Spesifik

1. **Puzzle sederhana**
   Semua algoritma menemukan solusi optimal dengan cepat, dengan GBFS yang seringkali tercepat.
2. **Puzzle kompleks**
   UCS menjelajahi banyak node, A* dengan DB sangat efisien, dan IDAS efektif meski lebih lambat. GBFS sangat cepat tetapi solusi sering suboptimal.

4.2.4 Visualisasi Solusi pada GUI

Implementasi antarmuka grafis (GUI) dalam Rush Hour Solver menyediakan beberapa elemen visual yang penting untuk memudahkan pemahaman solusi dan proses pencarian:

1. **Tampilan Papan**
   Kelas *BoardPanel* menyajikan visualisasi yang jelas dari papan permainan, dengan mobil yang diberi warna berbeda untuk memudahkan identifikasi.
2. **Animasi Solusi**
   Kelas *Animation* memungkinkan pengguna untuk melihat langkah demi langkah solusi dengan kontrol pemutaran, pengatur kecepatan, dan pelacakan langkah.

Manfaat Visualisasi

1. **Validasi Solusi**
   Pengguna dapat dengan mudah memverifikasi solusi yang ditemukan.

2. **Pemahaman Strategi**

   Menyediakan wawasan mengenai strategi pencarian yang digunakan.

3. **Debugging**

   Membantu mengidentifikasi potensi kelemahan dalam algoritma atau heuristik.

4. **Pengambilan Screenshot**

   Memudahkan dokumentasi solusi untuk analisis lebih lanjut.

Secara keseluruhan, komponen visualisasi dalam GUI tidak hanya meningkatkan pengalaman pengguna tetapi juga memberikan alat penting dalam menganalisis kinerja algoritma dan heuristik yang digunakan dalam puzzle Rush Hour.

Bab V
Penutup

5.1 Kesimpulan

Dari analisis berbagai algoritma pencarian untuk menyelesaikan permainan Rush Hour, dapat disimpulkan sebagai berikut:

1. **Perbandingan Algoritma Pencarian**
    a. **UCS (Uniform Cost Search)**:
    UCS selalu menemukan solusi optimal (jalur terpendek), namun sangat tidak efisien dalam hal waktu dan memori pada puzzle kompleks. Cocok untuk masalah di mana optimalitas solusi menjadi prioritas, tetapi dengan ruang state terbatas.
    b. **GBFS (Greedy Best-First Search)**:
    GBFS cepat dan efisien dalam mengunjungi node, tetapi sering menghasilkan solusi suboptimal. Cocok untuk situasi yang memprioritaskan kecepatan pencarian daripada optimalitas solusi.
    c. **A*S (A Star Search)**
    A*S memberikan keseimbangan terbaik antara kecepatan dan optimalitas, terutama dengan heuristik DB (Distance + Blocking). Algoritma ini lebih efisien dibandingkan UCS dan lebih optimal daripada GBFS.
    d. **IDA*S (Iterative Deepening A Star Search)**
    IDA*S sangat efisien dalam penggunaan memori tetapi lebih lambat karena mengevaluasi ulang banyak state pada setiap iterasi. Cocok digunakan ketika memori terbatas.
2. **Pengaruh Heuristik**
    a. **MD (Manhattan Distance)**
    Efektif untuk puzzle sederhana, tetapi kurang informatif pada puzzle kompleks dengan banyak penghalang.

b. **BP (Blocking Pieces)**

   Berguna untuk puzzle dengan banyak penghalang, tetapi tidak mempertimbangkan jarak ke tujuan.

c. **DB (Distance + Blocking)**

   Kombinasi terbaik yang memberikan keseimbangan antara informasi jarak dan penghalang, meskipun terkadang non-admissible.

3. **Visualisasi dan Representasi**
   - GUI yang disertakan memudahkan pengguna dalam memverifikasi solusi dan memahami proses pencarian algoritma melalui animasi dan visualisasi.
   - Penggunaan warna berbeda untuk setiap mobil memperjelas pergerakan mobil dalam solusi.
4. **Rush Hour sebagai Domain Pencarian**

   Rush Hour adalah platform ideal untuk menguji algoritma pencarian karena ruang state terbatas namun kompleks, serta memungkinkan penerapan heuristik yang intuitif. Variasi tingkat kesulitan puzzle memberikan tantangan yang berbeda, dari yang sederhana hingga sangat kompleks.

5.2 Saran

1. **Pengembangan Algoritma dan Heuristik**
   - **Heuristik Hibrid yang Adaptif**

     Mengembangkan heuristik yang menyesuaikan dengan karakteristik puzzle, seperti kepadatan penghalang atau posisi mobil utama.
   - **Penelusuran Bidirectional**

     Menerapkan pencarian dari dua arah (awal dan tujuan) untuk mengurangi node yang dijelajahi.
   - **Heuristik Pembelajaran Mesin**

     Menggunakan teknik pembelajaran mesin untuk mengembangkan heuristik yang lebih akurat.
2. **Optimasi Kinerja**

- **Penyimpanan State yang Lebih Efisien**

  Menggunakan bitboard untuk mengurangi memori dan mempercepat pemeriksaan state.
- **Paralelisasi**

  Mengimplementasikan algoritma pencarian secara paralel untuk mempercepat proses pencarian.

3. **Peningkatan Antarmuka Pengguna**
   - **Editor Puzzle**

     Menambahkan fitur untuk membuat dan mengedit puzzle secara langsung.
   - **Visualisasi Proses Pencarian**

     Menampilkan proses pencarian algoritma secara real-time.

   - **Statistik Perbandingan Interaktif**

     Menyediakan tampilan yang memungkinkan pengguna membandingkan kinerja berbagai algoritma dan heuristik.
4. **Perluasan Domain Aplikasi**
   - **Varian Permainan Rush Hour**

     Menyelidiki varian dengan aturan tambahan dan mengadaptasi solver untuk menangani varian ini.
   - **Aplikasi untuk Platform Mobile**

     Mengembangkan aplikasi untuk perangkat mobile agar pengguna dapat menyelesaikan puzzle di mana saja.
5. **Penelitian Lanjutan**
   - **Analisis Kompleksitas Teoritis**

     Mempelajari lebih lanjut struktur solusi optimal dan tantangan yang dihadapi algoritma pencarian.
   - **Pattern Database**

     Mengeksplorasi penggunaan pattern database untuk meningkatkan akurasi heuristik pada A*.

Secara keseluruhan, Rush Hour Solver berhasil menunjukkan efektivitas berbagai algoritma pencarian. Pengembangan lebih lanjut dapat berfokus pada mengoptimalkan kinerja, meningkatkan antarmuka pengguna, serta memperluas aplikasi pada berbagai platform.

Referensi

LaValle, S. M. (2006). *Planning algorithms*. Cambridge University Press.

Russell, S., & Norvig, P. (2010). *Artificial intelligence: A modern approach* (3rd ed.). Pearson Education.

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence, 27*(1), 97–109. https://doi.org/10.1016/0004-3702(85)90005-1

Rush Hour Solver. (n.d.). *Rush Hour solver and puzzle explanation*. Retrieved May 5, 2025, from https://www.rushhourpuzzle.com

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik, 1*(1), 269–271. https://doi.org/10.1007/BF01386390

Shannon, C. E. (1956). The synthesis of two-terminal switching circuits. *Bell System Technical Journal, 35*(1), 1–38. https://doi.org/10.1002/j.1538-7305.1956.tb03932.x

Lampiran

Link Repository : https://github.com/nathangalung/Tucil3_13523139_18222130

Tabel 1. Penilaian Kelengkapan Program

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

,