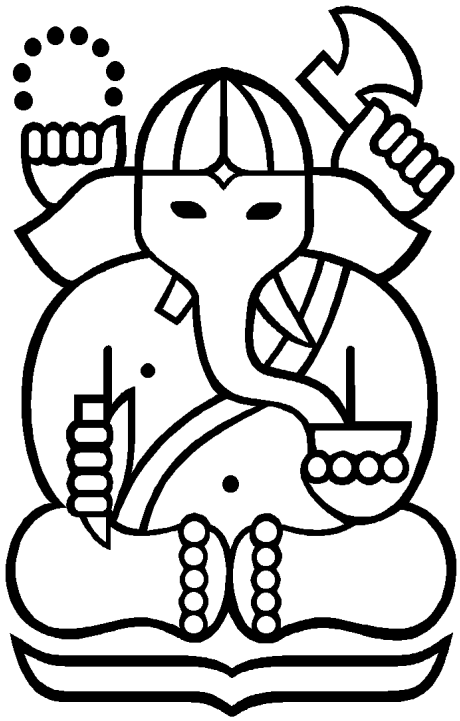
LAPORAN TUGAS KECIL 3
IF 2211 STRATEGI ALGORITMA
PENYELESAIAN PUZZLE RUSH HOUR MENGGUNAKAN ALGORITMA PATHFINDING



Disusun oleh :

13523043 Najwa ahani Fathima

13523080 Diyah Susan Nugrahani

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2025

DAFTAR ISI

A. Deskripsi Masalah

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

Komponen penting dari permainan Rush Hour terdiri dari:

1. Papan – *Papan* merupakan tempat permainan dimainkan.
   Papan terdiri atas *cell*, yaitu sebuah *singular point* dari papan. Sebuah *piece* akan menempati *cell-cell* pada papan. Ketika permainan dimulai, semua *piece* telah diletakkan di dalam papan dengan konfigurasi tertentu berupa lokasi piece dan *orientasi*, antara *horizontal* atau *vertikal*.
   Hanya *primary piece* yang dapat digerakkan keluar papan melewati *pintu keluar*. *Piece* yang bukan *primary piece* tidak dapat digerakkan keluar papan. Papan memiliki satu *pintu keluar* yang pasti berada di *dinding papan* dan sejajar dengan orientasi *primary piece*.

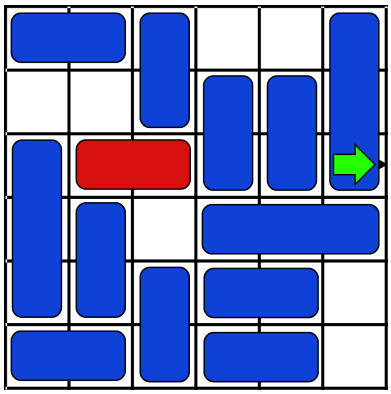
2. Piece – *Piece* adalah sebuah kendaraan di dalam papan. Setiap *piece* memiliki *posisi*, *ukuran*, dan *orientasi*. *Orientasi* sebuah *piece* hanya dapat berupa horizontal atau vertikal–tidak mungkin diagonal. *Piece* dapat memiliki beragam *ukuran*, yaitu jumlah *cell* yang ditempati oleh *piece*.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

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

4. Pintu Keluar – *Pintu keluar* adalah tempat *primary piece* dapat digerakkan keluar untuk menyelesaikan permainan

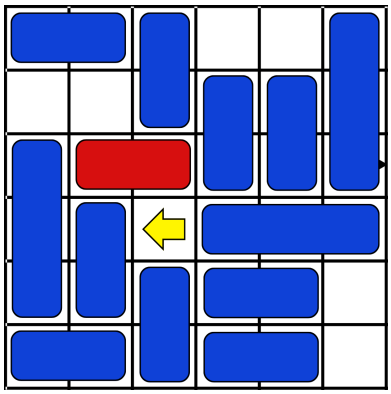
Ilustrasi kasus :

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

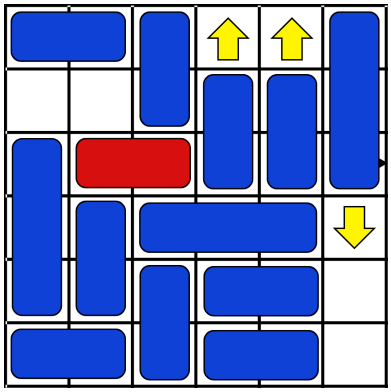


Gambar 1. Awal Permainan Game Rush Hour

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

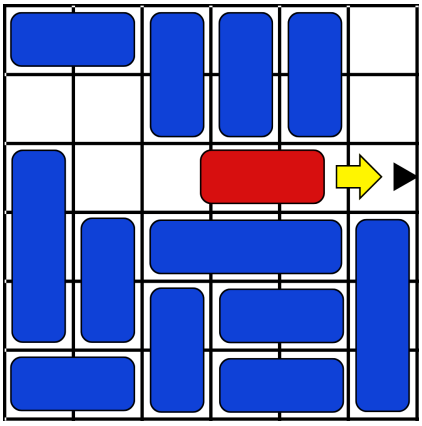


Gambar 2. Gerakan Pertama Game Rush Hour



Gambar 3. Gerakan Kedua Game Rush Hour

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



Gambar 4. Pemain Menyelesaikan Permainan

B. Penjelasan Umum Program

Program ini diimplementasikan menggunakan bahasa Java dan dengan kakas GUI *(Graphical User Interface)* JavaFX. Kakas untuk JavaFX terdapat pada folder lib. Folder src beberapa folder implementasi. Folder algorithms berisi program utama *(main logic)* dan implementasi dari algoritma-algoritma pencarian jalur. Folder utils berisi implementasi kelas yang digunakan untuk keberjalanan program. Sedangkan folder gui berisi implementasi GUI. dan Berikut adalah struktur folder dari program ini Rush Hour Solver ini.

```
├── .vscode/
├── bin/
├── doc/
├── lib/
│   └── javafx-sdk-24.0.1/
├── src/
│   ├── algorithms/
│   │   ├── Algorithm.java
│   │   ├── AStarSolver.java
│   │   ├── BeamSearchSolver.java
│   │   ├── GreedyBFSSolver.java
│   │   ├── Heuristic.java
│   │   └── UCSSolver.java
│   ├── gui/
│   │   ├── bg.png
│   │   ├── Block.java
│   │   ├── LandingPage.java
│   │   ├── MainApp.java
│   │   ├── MainBoardPage.java
│   │   └── style.css
│   ├── utils/
│   │   ├── Position.java
│   │   ├── RushHourGame.java
│   │   ├── SaveLoad.java
│   │   ├── State.java
│   │   ├── Vehicle.java
│   │   └── Main.java
├── test/
```

Pada bagian ini, akan dijelaskan juga mengenai implementasi kelas-kelas pada utils. Kelas Position digunakan untuk merepresentasikan posisi baris dan kolom pada papan. Kelas Vehicle digunakan untuk merepresentasikan blok pada papan, dengan satu blok direpresentasikan oleh satu Vehicle. Kelas State merepresentasikan kondisi papan saat ini, hal ini termasuk kondisi (letak dan posisi) dari masing-masing Vehicle, biaya yang digunakan untuk mencapai *state* tersebut, pergerakan yang dilakukan, dan *parent state* (kondisi sebelum *state* saat ini). Kemudian, kelas SaveLoad digunakan untuk membaca *file* input dan menuliskan hasil jalur ke *file* output.

Kelas RushHourGame mengimplementasikan logika keberjalanan game, mulai dari *parsing* input, mengkonstruksi papan dan *state* awal, menggerakkan Vehicle, menampilkan papan, serta memanggil algoritma dan heuristik sesuai input.

Kelas Algoritma merupakan kelas induk *(parent)* dari seluruh implementasi kelas algoritma. Kelas ini berisi implementasi umum seperti pengecekan *goal state* dan pencarian *state* anak untuk eksplorasi.

C. Algoritma UCS

Algoritma Uniform Cost Search (UCS) merupakan algoritma pencarian jalur yang termasuk dalam kategori *blind search* atau *uninformed search*. Algoritma ini digunakan untuk mencari biaya *(cost)* terendah dalam suatu jalur. Algoritma ini merupakan variansi dari algoritma Djikstra dan sangat berguna untuk pencarian jalur terbaik dengan biaya paling rendah dari simpul asal ke simpul tujuan, terutama untuk graf berbobot.

Dalam persoalan ini, bobot atau biaya untuk mencapai tujuan ditentukan oleh banyaknya langkah yang diambil untuk mencapai kondisi *(state/node)* tersebut. Pada persoalan Rush Hour, simpul adalah *state* dari papan yang mencakup lokasi dan letak masing-masing blok.

Beberapa konsep kunci pada algoritma ini adalah sebagai berikut:

1. Priority queue.

   Algoritma ini menggunakan *priority queue* untuk menyimpan simpul yang telah dicek. Simpul pada queue diurutkan berdasarkan biaya paling rendah. Sehingga, dalam setiap kali iterasi, akan dicek simpul dengan biaya paling rendah, memungkinkan pencarian jalur yang paling cepat dengan pergerakan minimum.

2. Biaya atau *cost*

   Seperti yang telah dijelaskan di atas, biaya untuk setiap simpul adalah jumlah langkah yang diambil untuk mencapai *state* simpul tersebut.

3. Eksplorasi

Pada setiap iterasi atau pengecekan, akan dilakukan pencarian jalur anak pada simpul pertama dari queue (simpul dengan biaya terendah). Untuk setiap pergerakan blok yang memungkinkan akan dibuat *state* atau simpul baru, lalu dimasukkan ke dalam priority queue dengan biaya + 1. Saat pengecekan simpul, jika *state* telah dieksplor, maka simpul akan dilewati (tidak dilakukan eksplorasi kembali).

4. Terminasi

   Pencarian dengan algoritma ini berhenti ketika suatu *goal state* telah dicapai atau ketika semua kemungkinan *state* telah dicek (tidak ada solusi).

```
function solve():
    nodeCount ← 0
    unexplored ← priority queue ordered by cost (lowest cost first)
    explored ← empty set

    start ← new State(clone of vehicle map, cost 0, no parent, empty move)
    add start to unexplored

    while unexplored is not empty:
        current ← remove element with lowest cost from unexplored
        stateString ← string representation of current state

        if stateString is in explored:
            continue

        add stateString to explored
        nodeCount ← nodeCount + 1

        if current is goal state:
            print "Total nodes explored: " + nodeCount
            return path from start to current

        neighbours ← get list of neighbours of current
        add all neighbours to unexplored

    print "Total nodes explored: " + nodeCount
    return null
```

D. Algoritma Greedy Best First Search

Algoritma Greedy Best First Search merupakan algoritma pencarian jalur dengan heuristik *(informed search)*. Ide dasar dari algoritma ini adalah menggunakan fungsi evaluasi f(n) untuk setiap simpul. Dalam persoalan Rush Hour, f(n) adalah heuristik

(h(n)) yang digunakan sebagai estimasi biaya dari *n* ke tujuan. Algoritma ini mengekspansi jalur yang terlihat atau nampak lebih dekat dengan *state* tujuan.

Implementasi untuk algoritma ini pada persoalan Rush Hour secara garis besar sama dengan yang ada pada UCS (seperti bagian sebelumnya). Bagian yang membedakan adalah variabel penentu urutan *prioriy queue* yang dalam hal ini, algoritma Greedy Best First Search menggunakan kalkulasi heuristik (dapat dilihat pada SubBab G. Fungsi Heuristik).

```
function solve():
    nodeCount ← 0
    unexplored ← priority queue ordered by heuristic value (lowest first)
    explored ← empty set

    start ← new State (clone of vehicle map, cost 0, no parent, empty move)
    add start to unexplored

    while unexplored is not empty:
        current ← remove element with lowest heuristic from unexplored
        stateString ← string representation of current state

        if stateString is in explored:
            continue

        add stateString to explored
        nodeCount ← nodeCount + 1

        if current is a goal state:
            print "Total nodes explored: " + nodeCount
            return path from start to current

        neighbours ← get list of neighbour states from current
        sort neighbours by heuristic value (lowest first)
        add all neighbours to unexplored

    print "Total nodes explored: " + nodeCount
    return null
```

E. Algoritma A*

Algoritma A* adalah algoritma yang digunakan untuk pencarian jalur di dalam suatu graf. Algoritma ini dapat digunakan untuk mencari jalur terdekat dari suatu simpul awal ke simpul grad. Algoritma ini termasuk dalam algoritma pencarian jalur bertipe *informed (heuristic search)*. Pencarian bertipe *informed search* menggunakan penerapan nilai

heuristik sehingga dapat memberikan jalur yang lebih efisien dibandingkan dengan *Breadth First Search (BFS) atau Depth First Search (DFS)*.

Algoritma A* merupakan gabungan dari dua algoritma pencarian jalur algoritma *Uniform Cost Search (UCS)* dan algoritma *Greedy Best-First Search.* Oleh karena itu, fungsi yang digunakan oleh algoritma A* adalah

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

Dengan g(n) adalah fungsi untuk mencari panjang lintasan dari simpul asal ke simpul n dan h(n) adalah fungsi untuk menaksir ongkos dari simpul n ke simpul tujuan. Algoritma ini akan membangkitkan simpul dengan nilai f(n) terkecil untuk setiap simpul aktif yang belum pernah ditelusuri. Karena menggunakan dua pendekatan untuk mencari nilai f(n) terkecil, algoritma ini relatif lebih lama dalam mengeksekusi program. Namun disisi lain, algoritma ini dijamin selalu menghasilkan solusi yang optimum.

Beberapa konsep kunci pada algoritma ini adalah sebagai berikut :

1. Priority queue

   Algoritma ini menggunakan *priority queue* untuk menyimpan simpul yang telah dicek. Simpul-simpul tersebut akan diurutkan berdasarkan nilai f(n) yang paling rendah. Simpul yang memiliki biaya paling rendah akan dibangkitkan dan dimasukkan ke jalur pencarian.

2. Biaya

   Algoritma ini menggunakan pembanding biaya dengan mempertimbangkan nilai dari heuristic dan cost untuk mencapai kondisi saat ini. Semakin rendah biayanya, langkah tersebut akan semakin diprioritaskan.

3. Eksplorasi

   Pencarian akan dimulai dengan membuat *priority queue* yang mengurutkan state berdasarkan nilai f(n). Kemudian menyimpan state mana saja yang sudah pernah diperiksa. Untuk seluruh state yang ada akan dicek, apabila state tersebut adalah

goal maka akan selesai. Jika state sudah pernah diperiksa maka akan diabaikan dan dilanjutkan ke state berikutnya. Kemudian state ditambahkan ke state yang telah diperiksa. Untuk setiap tetangga dari state ini, akan dilakukan pengecekan serupa, namun jika ditemukan jalur yang lebih baik maka akan diupdate jalurnya.

4. Terminasi

   Algoritma akan berhenti apabila goal state telah tercapai atau semua kemungkinan telah dicoba namun tidak memiliki solusi.

```
function AStar(game):
    nodeCount = 0
    gScores = new HashMap()
    openSet = new PriorityQueue(ordered by f(n) = g(n) + h(n))
    closedSet = new HashSet()

    startState = CreateInitialState(game)
    gScores[startState] = 0
    openSet.add(startState)

    while openSet is not empty:
        currentState = openSet.poll()

        if IsGoal(currentState, game) then
            print("Total nodes explored: " + nodeCount)
            return ConstructPath(currentState)

        if closedSet.contains(currentState) then
            continue

        nodeCount++
        closedSet.add(currentState)

        for each nextState in GetNeighbours(currentState):
            if closedSet.contains(nextState):
                continue

            tentativeGScore = currentState.cost + 1

            if nextState not in gScores OR tentativeGScore <
gScores[nextState] then
                gScores[nextState] = tentativeGScore

                //hapus state jika sudah ada di disana
                openSet.removeIf(state => state equals nextState)

                // updated state to openSet
                openSet.add(nextState)
```

```
print("Total nodes explored: " + nodeCount)
return null  // no solution found
```

F. Algoritma Beam Search

Algoritma *beam search* adalah algoritma pencarian jalur dengan pendekatan heuristik namun dengan membatasi jumlah node yang disimpan dalam queue dengan cara menyimpan hanya node terbaik di setiap state nya. Algoritma ini memiliki pendekatan yang hampir sama dengan *best first search* dengan mempertimbangkan nilai f(n) = h(n). Algoritma ini memiliki performa yang lebih baik jika dibandingkan dengan *best first search* karena node yang dibangkitkan dibatasi dan terjamin memiliki peluang yang lebih baik untuk mendapatkan jalur yang lebih optimum.

Beberapa konsep kunci pada algoritma ini adalah sebagai berikut :

1. Priority Queue

   Pada algoritma ini, pencarian dibatasi dengan cara membatasi jumlah state yang akan dicek. Hanya n state terbaik yang akan ditambahkan

2. Biaya

   Biaya yang diperhitungkan untuk algoritma ini adalah nilai heuristik nya yaitu f(n) = h(n). State yang akan ditambahkan untuk dicek adalah yang memiliki biaya terendah.

3. Eksplorasi

   Pencarian dimulai dengan membuat *priority queue* dan mengurutkannya berdasarkan nilai heuristic terkecil. Setelah itu membuat set yang berisi state yang belum dieksplor dan set yang berisi state yang telah dieksplor. Selama ada state yang belum dieksplor, maka cek state tersebut. Jika ternyata sudah dimasukkan ke state yang telah dieksplor maka lanjut ke state berikutnya, jika belum maka tambahkan ke set yang telah dieksplor. Urutkan seluruh tetangga dari state saat ini dan tambahkan maksimal sebanyak batas beamWidth ke set yang akan dieksplor.

4. Terminasi

   Pencarian berakhir ketika state saat ini adalah state goals atau ketika seluruh kemungkinan telah dicek namun tetap tidak menghasilkan solusi.

```
function BeamSearch(game, heuristic, beamWidth):
    nodeCount = 0
    unexplored = new PriorityQueue(ordered by heuristic value)
    explored = new HashSet()

    startState = CreateInitialState(game)
    unexplored.add(startState)

    while unexplored is not empty:
        currentState = unexplored.poll()

        if explored.contains(currentState):
            continue

        explored.add(currentState)
        nodeCount++

        if IsGoal(currentState, game):
            print("Total nodes explored: " + nodeCount)
            return ConstructPath(currentState)

        neighbours = GetNeighbours(currentState)

        // Sort neighbours by heuristic value (lowest first)
        Sort(neighbours, by heuristic value)

        // Add at most beamWidth best neighbours to unexplored
        for i = 0 to min(beamWidth, neighbours.size) - 1:
            if not explored.contains(neighbours[i]):
                unexplored.add(neighbours[i])

        // Limit unexplored queue to beamWidth
        if unexplored.size > beamWidth:
            newQueue = new PriorityQueue(ordered by heuristic value)

            for i = 0 to beamWidth - 1:
                if unexplored is not empty:
                    newQueue.add(unexplored.poll())

            unexplored = newQueue
```

```
    print("Total nodes explored: " + nodeCount)
    return null  // No solution found
```

G. Fungsi Heuristik

Fungsi heuristik digunakan untuk mempertimbangkan jalur mana yang akan dibangkitkan dengan cara memperhitungkan *cost* yang paling optimal di antara kemungkinan yang ada. Dalam tugas kecil ini, digunakan dua fungsi heuristik.

1. Banyak blok yang menghalangi pintu keluar

   Nilai heuristik ini mengukur berapa banyak blok yang menghalangi blok berwarna merah untuk mencapai pintu keluar. Nilai heuristik ini *admissible* karena untuk mencapai pintu keluar jarak minimum yang harus ditempuh sebanding dengan banyak blok yang menghalangi pintu keluar. Semakin banyak blok yang menghalangi maka blok target akan semakin sulit untuk mencapai pintu keluar.

2. Jarak blok target merah ke pintu keluar

   Nilai heuristik ini mengukur berapa jarak yang dibutuhkan oleh blok berwarna merah untuk mencapai pintu keluar. Nilai heuristik ini adalah nilai yang *admissible* karena untuk mencapai pintu keluar jarak minimum yang harus ditempuh adalah jarak blok berwarna merah ke pintu keluar.

H. Analisis Algoritma

f(n) merupakan fungsi yang digunakan untuk mengevaluasi setiap node dengan tujuan mendapatkan node yang memiliki biaya optimal untuk mencapai *goal state.* Nilai f(n) untuk setiap algoritma berbeda-beda. Untuk algoritma UCS nilai f(n) = g(n), untuk algoritma BFS nilai f(n) = h(n), untuk algoritma A* nilai f(n) = h(n) + g(n), dan untuk algoritma *beam search* nilai f(n) = h(n).

Nilai h(n) merepresentasikan nilai heuristic yang merupakan biaya estimasi yang dibutuhkan untuk mencapai *goal state* pada state saat ini. Pendekatan heuristik dapat dilakukan dengan beberapa cara, untuk tugas kecil ini digunakan beberapa pendekatan heuristik berupa jumlah blok penghalang dan jarak dari target ke pintu keluar. Sedangkan untuk nilai g(n) adalah biaya yang diperlukan dari *root* (posisi awal) hingga mencapai state saat ini.

Fungsi heuristik yang *admissible* memiliki syarat dimana untuk setiap node n, h(n) <= h*(n) dimana h*(n) adalah biaya sebenarnya yang harus dikeluarkan untuk mencapai ke *goal state.* Untuk heuristic pertama yang menggunakan jumlah blok yang menghalangi, heuristik ini tidak dijamin admissible karena ada situasi dimana beberapa kendaraan yang menghalangi memerlukan lebih dari satu langkah untuk dipindahkan. Hal ini dapat terjadi ketika kendaraan tersebut terkunci oleh kendaraan lain sehingga memerlukan multiple langkah sebelum bisa digeser. Hal ini dapat membuat perhitungan menjadi *over estimated.*

Untuk pendekatan dengan heuristik jarak antar target ke pintu keluar,heuristik ini *admissible* karena dalam kasus ini posisi kendaraan target dan pintu keluar selalu sejajar sehingga jaraknya adalah h*(n) itu sendiri. Karena heuristic ini tidak mempertimbangkan hambatan sehingga nilainya pasti selalu kurang atau sama dengan biaya aktual untuk mencapai ke pintu keluar. Namun, meskipun heuristik ini *admissible,* masih ada kemungkinan bahwa perhitungan tidak konsisten pada beberapa kasus.

Pada persoalan ini, algoritma UCS dan BFS (Breadth First Search) cukup sama. Hal ini karena pembobotan yang digunakan untuk *cost* UCS pada setiap langkah adalah satu, sehingga sama dengan pencarian BFS yang menggunakan kedalaman. UCS mempertimbangkan bobot atau biaya yang diperlukan untuk mencapai suatu *state* dan akan melakukan eksplorasi pada simpul atau *state* dengan biaya terendah terlebih dahulu. Artinya, urutan node yang dibangkitkan oleh UCS sama dengan BFS yang mencari simpul pada kedalaman pertama terlebih dahulu, begitu pula dengan *path* yang dihasilkan.

Jika dibandingkan dengan Greedy Best First Search, algoritma UCS memiliki perbedaan pada variabel yang digunakan untuk penentuan prioritas pada priority queue. UCS mempertimbangkan biaya, sedangkan Greedy Best First Search menggunakan fungsi heuristik sebagai fungsi pengurut dalam priority queue. Artinya, urutan *state* yang dibangkitkan dan *path* yang dihasillkan akan berbeda.

Secara teori, apabila fungsi heuristik yang digunakan untuk algoritma A* terjamin *admissible* maka A* akan lebih efisien dibandingkan UCS. Hal ini karena algoritma ini akan mengurangi eksplorasi jalur yang tidak relevan dan memprioritaskan jalur yang lebih mungkin menuju solusi lebih cepat. Sedangkan untuk UCS akan mengeksplorasi semua jalur dengan biaya terendah tanpa menggunakan pertimbangan lain.

Secara teori, algoritma Greedy pada umumnya tidak menjamin solusi yang optimal. Begitu pula dengan algoritma Greedy Best First Search untuk menyelesaikan persoalan Rush Hour ini. Selain itu, terdapat beberapa masalah dengan penggunaan algoritma ini untuk menyelesaikan persoalan, yakni: 1) Pencarian bisa saja tidak lengkap *(incomplete)*, 2) Terhambat dengan lokal minimum, dan 3) *Irrevocable* (tidak dapat diubah atau dibalikkan).

I. Source Program

1. Folder `algorithms`

Algorithm.java

```
package algorithms;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;
import utils.Position;
import utils.RushHourGame;
import utils.State;
import utils.Vehicle;
public class Algorithm {
```

```
    public static final String ANSI_RESET = "\u001B[0m";
    public static final String ANSI_YELLOW = "\u001B[43m";  // kuning
    public static final String ANSI_RED = "\u001B[41m";      // merah
    public static final String ANSI_GREEN = "\u001B[32m";  // hijau
    protected static RushHourGame game;
    public Algorithm(RushHourGame game) {
        this.game = game;
    }
  protected boolean isGoal(State state, RushHourGame game) {
        Vehicle target = state.getVehicle().get('P');
        Position exitPosition = game.getExitPosition();
        if (target.isHorizontal()) {
            int row = target.getRow();
            int tailCol = target.getCol();
            int headCol = target.getCol() + target.getLength() - 1;
            return ((headCol + 1 == exitPosition.getCol() || tailCol - 1 ==
exitPosition.getCol())
                    && row == exitPosition.getRow());
        } else {
            int col = target.getCol();
            int tailRow = target.getRow();
            int headRow = target.getRow() + target.getLength() - 1;

            return (headRow + 1 == exitPosition.getRow() || tailRow - 1 ==
exitPosition.getRow())
                    && col == exitPosition.getCol();
        }
    }
    protected boolean canExitHorizontal(Vehicle target, Position exit, Map<Character,
Vehicle> vehicles) {
        int row = target.getRow();
        int startCol, endCol;

        if (exit.getCol() < target.getCol()) {
            // exit is on the left side
            startCol = exit.getCol() + 1;
            endCol = target.getCol() - 1;
        } else {
            // exit is on the right side
            startCol = target.getCol() + target.getLength();
            endCol = exit.getCol();
        }

        for (int col = startCol; col <= endCol; col++) {
            for (Vehicle v : vehicles.values()) {
                if (v.getId() == target.getId()) continue;

                if (v.isHorizontal()) {
                    if (v.getRow() == row && col >= v.getCol() && col < v.getCol() +
v.getLength()) {
```

```
                        return false;
                    }
                } else {
                    if (col == v.getCol() && row >= v.getRow() && row < v.getRow() +
v.getLength()) {
                        return false;
                    }
                }
            }
        }

        return true;
    }

protected boolean canExitVertical(Vehicle target, Position exit, Map<Character,
Vehicle> vehicles) {
    int col = target.getCol();
    int startRow, endRow;

    if (exit.getRow() < target.getRow()) {
        // exit is on the top side
        startRow = exit.getRow() + 1;
        endRow = target.getRow() - 1;

        // Khusus untuk kasus pintu keluar di atas dan kendaraan di baris 0
        if (target.getRow() == 0) {
            // Kendaraan sudah di baris paling atas, langsung cek apakah kendaraan
target
            // bisa bergerak keluar (tidak ada yang menghalangi di sepanjang lintasan
keluar)
            if (exit.getRow() == -1 && exit.getCol() == col) {
                // Pastikan pintu keluar sejajar dengan kendaraan
                return true;
            } else {
                return false;
            }
        }
    } else {
        // exit is on the bottom side
        startRow = target.getRow() + target.getLength();
        endRow = exit.getRow();
    }

    // Jika interval invalid (startRow > endRow), langsung kembalikan false
    if (startRow > endRow) {
        return false;
    }

    // Periksa apakah ada kendaraan lain menghalangi jalur keluar
    for (int row = startRow; row <= endRow; row++) {
```

```
        for (Vehicle v : vehicles.values()) {
            if (v.getId() == target.getId()) continue;

            if (v.isHorizontal()) {
                if (row == v.getRow() && col >= v.getCol() && col < v.getCol() +
v.getLength()) {
                    return false;
                }
            } else {
                if (col == v.getCol() && row >= v.getRow() && row < v.getRow() +
v.getLength()) {
                    return false;
                }
            }
        }
    }

    return true;
}
    protected List<State> getNeighbours(State state) {
        List<State> neighbours = new ArrayList<>();
        char[][] board = buildBoardFromVehicles(state.vehicles);

        for (Vehicle v : state.vehicles.values()) {
            for (int direction : new int[]{-1,1}) {
                if (v.canMove(direction, board)) {
                    Map<Character, Vehicle> newVehicles =
cloneVehicleMap(state.vehicles);
                    Vehicle movedVehicle = newVehicles.get(v.getId());
                    movedVehicle.move(direction);

                    String dirString = getDirectionString(v, direction);
                    String moveDesc = v.getId() + " moves " + dirString;

                    State newState = new State(newVehicles, state.cost+1, state,
moveDesc);
                    neighbours.add(newState);
                }
            }
        }
        return neighbours;
    }
    protected String getDirectionString(Vehicle v, int direction) {
        if (v.isHorizontal()) {
            return direction == 1 ? "right" : "left";
        } else {
            return direction == 1 ? "down" : "up";
        }
    }
```

```
    protected Map<Character, Vehicle> cloneVehicleMap(Map<Character, Vehicle> original)
{
        Map<Character, Vehicle> copy = new HashMap<>();
        for (Map.Entry<Character, Vehicle> entry : original.entrySet()) {
            copy.put(entry.getKey(), new Vehicle(entry.getValue()));
        }
        return copy;
    }

    protected char[][] buildBoardFromVehicles(Map<Character, Vehicle> vehicles) {
        int rows = game.getBoard().length;
        int cols = game.getBoard()[0].length;
        char[][] board = new char[rows][cols];

        for (int i = 0; i < rows; i++) {
            Arrays.fill(board[i], '.');
        }

        for (Vehicle v : vehicles.values()) {
            for (int i = 0; i < v.getLength(); i++) {
                int r = v.getRow() + (v.isHorizontal() ? 0 : i);
                int c = v.getCol() + (v.isHorizontal() ? i : 0);
                if (r >= 0 && r < rows && c >= 0 && c < cols) {
                    board[r][c] = v.getId();
                }
            }
        }

        return board;
    }

    protected List<State> constructPath(State goal) {
        List<State> path = new ArrayList<>();
        while (goal != null) {
            path.add(goal);
            goal = goal.parent;
        }
        Collections.reverse(path);
        return path;
    }

    public int displaySolution(List<State> path) {
        if (path == null || path.isEmpty()) return 0;

        int step = 0;
        System.out.println("Step " + step++ + " - Initial state");
        printBoard(path.get(0).vehicles, '-');
        System.out.println("------------------------");

        if (path.size() == 1) return 0;
```

```
        char currentVehicle = '-';
        State lastState = path.get(0);
        String lastMove = "";

        for (int i = 1; i < path.size(); i++) {
            State state = path.get(i);
            char vehicle = state.move.charAt(0);

            if (vehicle != currentVehicle) {
                // cetak gerakan sebelumnya jika ada
                if (!lastMove.isEmpty()) {
                    System.out.println("Step " + step++);
                    System.out.println("Move: " + lastMove);
                    printBoard(lastState.vehicles, currentVehicle);
                    System.out.println("------------------------");
                }
                currentVehicle = vehicle;
            }

            lastState = state;
            lastMove = state.move;
        }

        // cetak gerakan terakhir
        if (!lastMove.isEmpty()) {
            System.out.println("Step " + step);
            System.out.println("Move: " + lastMove);
            printBoard(lastState.vehicles, currentVehicle);
            System.out.println("------------------------");
        }

        return step;
    }

    protected void printBoard(Map<Character, Vehicle> vehicles, char id) {
        int rows = game.getBoard().length;
        int cols = game.getBoard()[0].length;
        char[][] board = new char[rows][cols];

        for (int i = 0; i < rows; i++) {
            Arrays.fill(board[i], '.');
        }

        for (Vehicle v : vehicles.values()) {
            for (int i = 0; i < v.getLength(); i++) {
                int r = v.getRow() + (v.isHorizontal() ? 0 : i);
                int c = v.getCol() + (v.isHorizontal() ? i : 0);
                if (r >= 0 && r < rows && c >= 0 && c < cols) {
                    board[r][c] = v.getId();
```

```
            }
        }
    }

    Position exit = game.getExitPosition();

    // cetak 'K' di luar papan jika exit position negatif
    if (exit.getRow() == -1) { // exit di atas papan
        for (int j = 0; j < exit.getCol(); j++) {
            System.out.print("  ");
        }
        System.out.println(ANSI_GREEN + "K" + ANSI_RESET + " ");
    } else if (exit.getCol() == -1) { // exit di kiri papan
        // System.out.print(ANSI_GREEN + "K" + ANSI_RESET + " ");
    }

    // cetak papan
    for (int i = 0; i < rows; i++) {
        //  'K' di kiri jika exit di baris ini
        if (exit.getCol() == -1 && i == exit.getRow()) {
            System.out.print(ANSI_GREEN + "K" + ANSI_RESET + " ");
        }
        else if(exit.getCol() == -1) { //iya?
            System.out.print("  ");

        }

        for (int j = 0; j < cols; j++) {
            //  'K' di dalam papan jika posisi exit valid
            if (i == exit.getRow() && j == exit.getCol() &&
                exit.getRow() >= 0 && exit.getCol() >= 0) {
                System.out.print(ANSI_GREEN + "K" + ANSI_RESET + " ");
                continue;
            }

            // Cetak kendaraan dengan warna
            if (board[i][j] == id) {
                System.out.print(ANSI_YELLOW + board[i][j] + ANSI_RESET + " ");
            } else if (board[i][j] == 'P') {
                System.out.print(ANSI_RED + board[i][j] + ANSI_RESET + " ");
            } else {
                System.out.print(board[i][j] + " ");
            }
        }

        //  'K' di kanan jika exit di baris ini
        if (exit.getCol() == cols && i == exit.getRow()) {
            System.out.print(ANSI_GREEN + "K" + ANSI_RESET + " ");
        }
        System.out.println();
```

```
        }

        //  'K' di bawah papan jika exit position di bawah
        if (exit.getRow() == rows) {
            for (int j = 0; j < exit.getCol(); j++) {
                System.out.print("  ");
            }
            System.out.println(ANSI_GREEN + "K" + ANSI_RESET + " ");
        }
    }

    public static void writeSolutionToFile(String filename, List<State> path, int
steps, int nodeCount, long duration) {
        try (FileWriter writer = new FileWriter(filename)) {
            if (path == null || path.isEmpty()) {
                writer.write("No solution found.\n");
                return;
            }

            int step = 0;
            writer.write("Step " + step++ + " - Initial state\n");
            writer.write(boardToString(path.get(0).vehicles, '-') + "\n");
            writer.write("------------------------\n");

            if (path.size() == 1) return;

            char currentVehicle = '-';
            State lastState = path.get(0);
            String lastMove = "";

            for (int i = 1; i < path.size(); i++) {
                State state = path.get(i);
                char vehicle = state.move.charAt(0);

                if (vehicle != currentVehicle) {
                    if (!lastMove.isEmpty()) {
                        writer.write("Step " + step++ + "\n");
                        writer.write("Move: " + lastMove + "\n");
                        writer.write(boardToString(lastState.vehicles, currentVehicle)
+ "\n");
                        writer.write("------------------------\n");
                    }
                    currentVehicle = vehicle;
                }

                lastState = state;
                lastMove = state.move;
            }

            if (!lastMove.isEmpty()) {
```

```
                writer.write("Step " + step + "\n");
                writer.write("Move: " + lastMove + "\n");
                writer.write(boardToString(lastState.vehicles, currentVehicle) + "\n");
                writer.write("-----------------------\n");
            }

            writer.write("\nSummary:\n");
            writer.write("Total steps: " + steps + "\n");
            writer.write("Total nodes explored: " + nodeCount + "\n");
            writer.write("Solution found in " + duration + " ms\n");
        } catch (IOException e) {
            System.err.println("Error writing to file: " + e.getMessage());
        }
    }

    private static String boardToString(Map<Character, Vehicle> vehicles, char id) {
        StringBuilder sb = new StringBuilder();
        int rows = game.getBoard().length;
        int cols = game.getBoard()[0].length;
        char[][] board = new char[rows][cols];

        for (int i = 0; i < rows; i++) {
            Arrays.fill(board[i], '.');
        }

        for (Vehicle v : vehicles.values()) {
            for (int i = 0; i < v.getLength(); i++) {
                int r = v.getRow() + (v.isHorizontal() ? 0 : i);
                int c = v.getCol() + (v.isHorizontal() ? i : 0);
                if (r >= 0 && r < rows && c >= 0 && c < cols) {
                    board[r][c] = v.getId();
                }
            }
        }

        Position exit = game.getExitPosition();

        if (exit.getRow() == -1) {
            for (int j = 0; j < exit.getCol(); j++) {
                sb.append("  ");
            }
            sb.append("K\n");
        }

        for (int i = 0; i < rows; i++) {
            if (exit.getCol() == -1 && i == exit.getRow()) {
                sb.append("K ");
            }
            else if(exit.getCol() == -1) {
                sb.append("  ");
```

```
            }

            for (int j = 0; j < cols; j++) {
                if (i == exit.getRow() && j == exit.getCol() &&
                    exit.getRow() >= 0 && exit.getCol() >= 0) {
                    sb.append("K ");
                    continue;
                }

                if (board[i][j] == id) {
                    sb.append(board[i][j]).append(" ");
                } else if (board[i][j] == 'P') {
                    sb.append(board[i][j]).append(" ");
                } else {
                    sb.append(board[i][j]).append(" ");
                }
            }

            if (exit.getCol() == cols && i == exit.getRow()) {
                sb.append("K");
            }
            sb.append("\n");
        }

        if (exit.getRow() == rows) {
            for (int j = 0; j < exit.getCol(); j++) {
                sb.append("  ");
            }
            sb.append("K\n");
        }

        return sb.toString();
    }

    protected void debugUnexplored(PriorityQueue<State> unexplored) {
        for (State s : unexplored) {
            System.out.println("move: " + s.move + " --- h(n): " +
Heuristic.calculateHeuristicGreedy(s, game, 1) + " --- f(n): " +
            (s.cost + Heuristic.calculateHeuristicGreedy(s, game, 1)));
        }
        System.out.println();
    }
}
```

AStarSolver.java

```
package algorithms;
import java.util.*;
import utils.RushHourGame;
import utils.State;
```

```java
public class AStarSolver extends Algorithm {
    private int nodeCount = 0;
    private int heuristic = 1;

    public AStarSolver(RushHourGame game, int heu) {
        super(game);
        this.nodeCount = 0;
        this.heuristic = heu;
    }

    public List<State> solve() {
        nodeCount = 0;
        Map<String, Integer> gScores = new HashMap<>();
        PriorityQueue<State> openSet = new PriorityQueue<>((s1, s2) -> {
            int f1 = s1.getCost() + Heuristic.calculateHeuristicAstar(s1, game,
heuristic);
            int f2 = s2.getCost() + Heuristic.calculateHeuristicAstar(s2, game,
heuristic);
            return Integer.compare(f1, f2);
        });

        Set<String> closedSet = new HashSet<>();

        State startState = new State(cloneVehicleMap(game.getVehicles()), 0, null, "");
        String startStateKey = startState.getStateString();
        gScores.put(startStateKey, 0);
        openSet.add(startState);

        while (!openSet.isEmpty()) {
            //state with lowest f score
            State currentState = openSet.poll();
            String currentStateKey = currentState.getStateString();
            if (isGoal(currentState, game)) {
                System.out.println("Total nodes explored: " + nodeCount);
                return constructPath(currentState);
            }

            if (closedSet.contains(currentStateKey)) {
                continue;
            }
            nodeCount++;

            closedSet.add(currentStateKey);

            for (State nextState : getNeighbours(currentState)) {
                String nextStateKey = nextState.getStateString();

                if (closedSet.contains(nextStateKey)) {
                    continue;
                }
```

```
                int tentativeGScore = currentState.getCost() + 1;

                if (!gScores.containsKey(nextStateKey) || tentativeGScore <
gScores.get(nextStateKey)) {
                    // update best known cost to this state
                    gScores.put(nextStateKey, tentativeGScore);


                    openSet.removeIf(s -> s.getStateString().equals(nextStateKey));

                    openSet.add(nextState);
                }
            }
        }
        System.out.println("Total nodes explored: " + nodeCount);
        return null;
    }

    public int getNodeCount() {
        return nodeCount;
    }
}
```

BeamSeaerchSolver.java

```
package algorithms;

import java.util.*;
import utils.RushHourGame;
import utils.State;

public class BeamSearchSolver extends Algorithm {
    private int nodeCount = 0;
    private int heuristic = 1;
    private int beamWidth;

    public BeamSearchSolver(RushHourGame game, int heuristic, int beamWidth) {
        super(game);
        this.nodeCount = 0;
        this.heuristic = heuristic;
        this.beamWidth = beamWidth;
    }

    public List<State> solve() {
        nodeCount = 0;
        PriorityQueue<State> unexplored = new PriorityQueue<>(Comparator.comparingInt(s
-> Heuristic.calculateHeuristicGreedy(s, game, heuristic)));
        Set<String> explored = new HashSet<>();
```

```
        State start = new State(cloneVehicleMap(game.getVehicles()), 0, null, "");
        unexplored.add(start);

        while (!unexplored.isEmpty()) {
            State current = unexplored.poll();
            String stateString = current.getStateString();

            if (explored.contains(stateString)) continue;
            explored.add(stateString);
            nodeCount++;

            if (isGoal(current, game)) {
                System.out.println("Total nodes explored: " + nodeCount);
                return constructPath(current);
            }

            List<State> neighbours = getNeighbours(current);

            neighbours.sort(Comparator.comparingInt(s ->
Heuristic.calculateHeuristicGreedy(s, game, heuristic)));

            for (int i = 0; i < Math.min(beamWidth, neighbours.size()); i++) {
                if (!explored.contains(neighbours.get(i).getStateString())) {
                    unexplored.add(neighbours.get(i));
                }
            }

            if (unexplored.size() > beamWidth) {
                PriorityQueue<State> newQueue = new
PriorityQueue<>(Comparator.comparingInt(s ->
                    Heuristic.calculateHeuristicGreedy(s, game, heuristic)));

                for (int i = 0; i < beamWidth; i++) {
                    if (!unexplored.isEmpty()) {
                        newQueue.add(unexplored.poll());
                    }
                }

                unexplored = newQueue;
            }
        }

        System.out.println("Total nodes explored: " + nodeCount);
        return null;
    }

    public void setBeamWidth(int beamWidth) {
        this.beamWidth = beamWidth;
    }
```

```
    public int getNodeCount() {
        return nodeCount;
    }
}
```

GreedyBFSSolver.java

```
package algorithms;
import java.util.*;
import utils.RushHourGame;
import utils.State;

public class GreedyBFSSolver extends Algorithm {
    private int nodeCount = 0;
    private int heuristic = 1;

    public GreedyBFSSolver(RushHourGame game, int heu) {
        super(game);
        this.nodeCount = 0;
        this.heuristic = heu;
    }

    public List<State> solve() {
        nodeCount = 0;
        PriorityQueue<State> unexplored = new PriorityQueue<>(Comparator.comparingInt(s
-> Heuristic.calculateHeuristicGreedy(s, game, heuristic)));
        Set<String> explored = new HashSet<>();

        State start = new State(cloneVehicleMap(game.getVehicles()), 0, null, "");
        unexplored.add(start);

        while (!unexplored.isEmpty()) {
            State current = unexplored.poll();
            String stateString = current.getStateString();

            if (explored.contains(stateString)) continue;
            explored.add(stateString);
            nodeCount++;

            if (isGoal(current, game)) {
                System.out.println("Total nodes explored: " + nodeCount);
                return constructPath(current);
            }

            List<State> neighbours = getNeighbours(current);
            neighbours.sort(Comparator.comparingInt(s ->
Heuristic.calculateHeuristicGreedy(s, game, heuristic)));
            unexplored.addAll(neighbours);
        }
```

```
        System.out.println("Total nodes explored: " + nodeCount);
        return null;
    }

    public int getNodeCount() {
        return nodeCount;
    }
}
```

Heuristic.java

```
package algorithms;
import utils.Position;
import utils.RushHourGame;
import utils.State;
import utils.Vehicle;

public class Heuristic {
     public static int manhattanDistance(State state, RushHourGame game) {
        Vehicle playerVehicle = state.getVehicle().get('P');
        Position exitPosition = game.getExitPosition();

        int playerEndRow = playerVehicle.getRow();
        int playerEndCol = playerVehicle.getCol() + playerVehicle.getLength() - 1;

        int horizontalDistance = Math.abs(exitPosition.getCol() - playerEndCol);
        int verticalDistance = Math.abs(exitPosition.getRow() - playerEndRow);

        int manhattanDistance = horizontalDistance + verticalDistance;

        return manhattanDistance;
    }

    public static int blockingCount(State state, RushHourGame game) {
        Vehicle playerVehicle = state.getVehicle().get('P');
        int playerEndCol = playerVehicle.getCol() + playerVehicle.getLength() - 1;

        int blockingCount = 0;
        int playerRow = playerVehicle.getRow();

        for (Vehicle vehicle : state.getVehicle().values()) {
            if (vehicle.getId() == 'P') continue;

            if (vehicle.isHorizontal()) {
                // horizontal
                if (vehicle.getRow() == playerRow &&
                    vehicle.getCol() > playerEndCol) {
                    blockingCount++;
                }
            } else {
```

```
                // vertical
                int vehicleStartRow = vehicle.getRow();
                int vehicleEndRow = vehicleStartRow + vehicle.getLength() - 1;

                if (vehicle.getCol() > playerEndCol &&
                    vehicleStartRow <= playerRow && vehicleEndRow >= playerRow) {
                    blockingCount++;
                }
            }
        }

        return blockingCount;
    }

    public static int calculateHeuristicGreedy(State state, RushHourGame game, int
opsi) {
        if (opsi == 1) {
            return blockingCount(state, game);
        } else {
            return manhattanDistance(state, game);
        }
    }

    public static int calculateHeuristicAstar(State state, RushHourGame game, int opsi)
{
        Vehicle playerVehicle = state.getVehicle().get('P');
        Position exitPosition = game.getExitPosition();

        int playerEndCol = playerVehicle.getCol() + playerVehicle.getLength() - 1;
        int distanceToExit = Math.max(0, exitPosition.getCol() - playerEndCol);

        if (opsi == 1) {
            return distanceToExit + blockingCount(state, game);
        } else {
            return distanceToExit + manhattanDistance(state, game);
        }
    }

}
```

UCSSolver.java

```
package algorithms;
import java.util.*;
import utils.*;

public class UCSSolver extends Algorithm {
    private int nodeCount = 0;
    public UCSSolver(RushHourGame game) {
        super(game);
```

```
        this.nodeCount = 0;
    }

    public List<State> solve() {
        nodeCount = 0;
        PriorityQueue<State> unexplored = new PriorityQueue<>(Comparator.comparingInt(n
-> n.cost));
        Set<String> explored = new HashSet<>();

        State start = new State(cloneVehicleMap(game.getVehicles()), 0, null, "");
        unexplored.add(start);

        while (!unexplored.isEmpty()) {
            State current = unexplored.poll();
            String stateString = current.getStateString();

            if (explored.contains(stateString)) continue;
            explored.add(stateString);
            nodeCount++;

            if (isGoal(current, game)) {
                System.out.println("Total nodes explored: " + nodeCount);
                return constructPath(current);
            }

            unexplored.addAll(getNeighbours(current));

        }
        System.out.println("Total nodes explored: " + nodeCount);
        return null;
    }

    public int getNodeCount() {
        return nodeCount;
    }
}
```

2. Folder `utils`

Position.java

```
package utils;
public class Position {
    public int row;
    public int col;

    public Position(int row, int col) {
        this.row = row;
```

```
        this.col = col;
    }

    public Position(Position other) {
        this.row = other.row;
        this.col = other.col;
    }

    public void displayPosition() {
    System.out.println("Row: " + row + ", Col: " + col);
    }

    public int getRow(){
        return row;
    }
    public int getCol(){
        return col;
    }
}
```

RushHourGame.java

```
package utils;

import algorithms.AStarSolver;
import algorithms.BeamSearchSolver;
import algorithms.GreedyBFSSolver;
import algorithms.UCSSolver;
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class RushHourGame {
    // warna ANSI untuk terminal
    public static final String ANSI_RESET = "\u001B[0m";
    public static final String ANSI_YELLOW = "\u001B[43m";  // kuning
    public static final String ANSI_RED = "\u001B[41m";      // merah

    private int rows;
    private int cols;
    private int numVehicles;
    private char[][] board;
    private Map<Character, Vehicle> vehicles;
    private Vehicle targetVehicle;
    private Position exitPosition;

    public long startTime;
    public long endTime;
    public List<State> solution;
```

```
    public int steps;
    public int nodes;



    public RushHourGame(String input) {
        parseInput(input);
        identifyVehicles();
    }

    public RushHourGame(RushHourGame other) {
        this.rows = other.rows;
        this.cols = other.cols;
        this.numVehicles = other.numVehicles;
        this.board = new char[rows][cols];
        for (int i = 0; i < rows; i++) {
            System.arraycopy(other.board[i], 0, this.board[i], 0, cols);
        }
        this.vehicles = new HashMap<>();
        for (Map.Entry<Character, Vehicle> entry : other.vehicles.entrySet()) {
            this.vehicles.put(entry.getKey(), new Vehicle(entry.getValue()));
        }
        this.targetVehicle = new Vehicle(other.targetVehicle);
        this.exitPosition = new Position(other.exitPosition);
    }

    private void parseInput(String input) {
        String[] lines = input.split("\\r?\\n"); // Support Windows + Unix line endings
        String[] dimensions = lines[0].trim().split("\\s+");
        rows = Integer.parseInt(dimensions[0]);
        cols = Integer.parseInt(dimensions[1]);
        numVehicles = Integer.parseInt(lines[1].trim());
        board = new char[rows][cols];

        for (char[] row : board) {
            Arrays.fill(row, ' ');
        }



        // cek baris di atas papan
        if (lines.length > 2) {
            String topLine = lines[2];
            for (int j = 0; j < topLine.length(); j++) {
                if (topLine.charAt(j) == 'K') {
                    // K di atas papan
                    exitPosition = new Position(-1, j);
                    System.out.println("K ditemukan di atas papan: exitPosition=-1," +
j);

                    break;
                }
```

```
            }
        }

        // cek baris di bawah papan
        if (lines.length > rows + 2) {
            String bottomLine = lines[rows + 2];
            System.out.println(bottomLine);
            for (int j = 0; j < bottomLine.length(); j++) {
                if (bottomLine.charAt(j) == 'K') {
                    // K di bawah papan
                    exitPosition = new Position(rows, j);
                    System.out.println("K ditemukan di bawah papan: exitPosition=" +
rows + "," + j);
                    break;
                }
            }
        }

        // isi papan dengan karakter non-K
        int boardStartLine = 0;

        if (lines.length > 2 && lines[2].contains("K")) {
            boardStartLine = 1;
        }

        for (int i = 0; i < rows; i++) {
            int lineIndex = i + boardStartLine + 2;
            if (lineIndex >= lines.length) break;

            String line = lines[lineIndex];
            int boardCol = 0;

            for (int j = 0; j < line.length(); j++) {
                char c = line.charAt(j);
                if (c == ' ') continue;

                if (c == 'K') continue;

                if ((c >= 'A' && c <= 'Z') || c == '.') {
                    if (boardCol < cols) {
                        board[i][boardCol++] = c;
                    }
                }
            }

            // K di kiri dan kanan
            if (line.length() > 0 && line.charAt(0) == 'K') {
                exitPosition = new Position(i, -1);
                System.out.println("K ditemukan di sisi kiri: exitPosition=" + i +
",-1");
```

```
            }

            if (boardCol == cols && line.length() > cols && line.charAt(cols) == 'K') {
                exitPosition = new Position(i, cols);
                System.out.println("K ditemukan di sisi kanan: exitPosition=" + i + ","
+ cols);
            }
        }

        int lastBoardRow = Math.min(rows - 1, lines.length - 3);
        if (lastBoardRow >= 0) {
            String lastLine = lines[lastBoardRow + 2];
            for (int j = 0; j < lastLine.length(); j++) {
                if (lastLine.charAt(j) == 'K' && (j >= cols || j == lastLine.length() -
1)) {
                    exitPosition = new Position(lastBoardRow, cols);
                    System.out.println("K ditemukan di sisi kanan terakhir:
exitPosition=" + lastBoardRow + "," + cols);
                    break;
                }
            }
        }

        if (exitPosition == null) {
            throw new IllegalArgumentException("Pintu keluar (K) tidak ditemukan dalam
input");
        }
    }

    private void identifyVehicles() {
    vehicles = new HashMap<>();

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            char current = board[i][j];
            if (current != '.' && current != 'K' && !vehicles.containsKey(current)) {
                // identify direction (horizontal or vertical)
                boolean isHorizontal = (j + 1 < cols && board[i][j + 1] == current);

                // starting position of the vehicle
                int length = 1;
                int startRow = i;
                int startCol = j;

                if (isHorizontal) {
                    int col = j + 1;
                    while (col < cols && board[i][col] == current) {
                        length++;
                        col++;
                    }
```

```
                } else {
                    // vertical
                    int row = i + 1;
                    while (row < rows && board[row][j] == current) {
                        length++;
                        row++;
                    }
                }

                Vehicle vehicle = new Vehicle(current, startRow, startCol, length,
isHorizontal);
                if (current == 'P') {
                    targetVehicle = vehicle;
                }

                vehicles.put(current, vehicle);
            }
        }
    }
}


    public Position getExitPosition() {
        return exitPosition;
    }

    public char[][] getBoard(){
        return this.board;
    }

    public Vehicle getTargetVehicle() {
        return targetVehicle;
    }
    public Map<Character, Vehicle> getVehicles() {
        return vehicles;
    }

    public void displayBoard() {
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[i].length; j++) {
                char cell = board[i][j];
                if (cell == '.') {
                    System.out.print(". ");
                } else {
                    // cek apakah kendaraan ini baru bergerak atau 'P'
                    if (cell == lastMovedVehicle) {
                        System.out.print(ANSI_YELLOW + cell + ANSI_RESET + " ");  //
kuning
                    } else if (cell == 'P') {
                        System.out.print(ANSI_RED + cell + ANSI_RESET + " ");     //
```

```
merah
                    } else {
                        System.out.print(cell + " ");
                    }
                }
            }
            System.out.println();
        }
        System.out.println();
    }



    private char lastMovedVehicle = '\0';  // menyimpan ID kendaraan terakhir yang
bergerak

    public void moveVehicle(char vehicleId, int direction) {
        Vehicle vehicle = vehicles.get(vehicleId);
        int row = vehicle.getRow();
        int col = vehicle.getCol();
        int length = vehicle.getLength();

        // clear current position
        if (vehicle.isHorizontal()) {
            for (int i = 0; i < length; i++) {
                board[row][col + i] = '.';
            }
        } else {
            for (int i = 0; i < length; i++) {
                board[row + i][col] = '.';
            }
        }

        // update position
        if (vehicle.isHorizontal()) {
            vehicle.setCol(direction < 0 ? col - 1 : col + 1);
        } else {
            vehicle.setRow(direction < 0 ? row - 1 : row + 1);
        }

        // set new position
        row = vehicle.getRow();
        col = vehicle.getCol();
        if (vehicle.isHorizontal()) {
            for (int i = 0; i < length; i++) {
                board[row][col + i] = vehicleId;
            }
        } else {
            for (int i = 0; i < length; i++) {
                board[row + i][col] = vehicleId;
```

```
            }
        }

        lastMovedVehicle = vehicleId;
        if (vehicleId == 'P') {
            targetVehicle = new Vehicle(vehicle);
        }
    }

    public int getCols(){
        return cols;
    }

    public int getRows(){
        return rows;
    }

    public char getChar(int row, int col) {
        return board[row][col];
    }

    public void solveGame(int algorithm, int heuristic, int beamWidth) {

        System.out.println("algo: " + algorithm);
        System.out.println("beam: " + beamWidth);
        if (!isPossible()) {
            solution = null;
            startTime = 0;
            endTime = 0;
            steps = 0;
            nodes = 0;
            return;
        }

        // start game
        startTime = System.currentTimeMillis();
        switch (algorithm) {
            case 3:
                {
                    AStarSolver solver = new AStarSolver(this, heuristic);
                    solution = solver.solve();
                    steps = solver.displaySolution(solution);
                    nodes = solver.getNodeCount();
                    break;
                }
            case 2:
                {
                    GreedyBFSSolver solver = new GreedyBFSSolver(this, heuristic);
                    solution = solver.solve();
                    steps = solver.displaySolution(solution);
```

```
                    nodes = solver.getNodeCount();
                    break;
                }
            case 4:
                {
                    BeamSearchSolver solver = new BeamSearchSolver(this, heuristic,
beamWidth);
                    solution = solver.solve();
                    steps = solver.displaySolution(solution);
                    nodes = solver.getNodeCount();
                    break;
                }
            default:
                {
                    UCSSolver solver = new UCSSolver(this);
                    solution = solver.solve();
                    steps = solver.displaySolution(solution);
                    nodes = solver.getNodeCount();
                    break;
                }
        }

        endTime = System.currentTimeMillis();
    }

    private boolean isPossible() {
        int exitRow = exitPosition.getRow();
        int exitCol = exitPosition.getCol();

        if (targetVehicle.isHorizontal()) {
            if (targetVehicle.getRow() != exitRow) {
                return false;
            }
        } else {
            if (targetVehicle.getCol() != exitCol) {
                return false;
            }
        }
        return true;
    }
}
```

SaveLoad.java

```
package utils;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
```

```java
import java.util.Scanner;

public class SaveLoad{
    public String Load(){
        // input file
        Scanner scanner = new Scanner(System.in);
        System.out.println("Masukkan path folder: ");
        String filePath = scanner.nextLine();

        // convert ke string
        StringBuilder content = new StringBuilder();
        try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
            String line;
            while ((line = reader.readLine()) != null) {
                content.append(line).append("\n");
            }
        } catch (IOException e) {
            System.out.println("Kesalahan saat membaca file: " + e.getMessage());
        }
        return content.toString();
    }

    public static void saveMatrixToTxt(char[][] matrix, String filename) {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(filename))) {
            for (int i = 0; i < matrix.length; i++) {
                for (int j = 0; j < matrix[i].length; j++) {
                    writer.write(String.valueOf(matrix[i][j]));

                }
                writer.newLine(); // Baris baru untuk setiap baris matrix
            }
            System.out.println("Matrix berhasil disimpan ke " + filename);
        } catch (IOException e) {
            System.err.println("Error saat menyimpan matrix: " + e.getMessage());
        }
    }
}
```

State.java

```java
package utils;
import java.util.*;
public class State implements Comparable<State> {
    public Map<Character, Vehicle> vehicles;
    public int cost;
    public State parent;
    public String move;
    public State(Map<Character, Vehicle> vehicles, int cost, State parent, String move)
{
        this.vehicles = vehicles;
```

```
        this.cost = cost;
        this.parent = parent;
        this.move = move;
    }

    public int getCost(){
        return cost;
    }

    @Override
    public int compareTo(State other) {
        return Integer.compare(this.cost, other.cost);
    }

    public String getStateString() {
        List<String> ids = new ArrayList<>();
        for (Vehicle v : vehicles.values()) {
            ids.add(v.getId() + ":" + v.getRow() + "," + v.getCol());
        }
        Collections.sort(ids);
        return String.join("|", ids);
    }

    public Map<Character, Vehicle> getVehicle() {
        return vehicles;
    }
}
```

Vehicle.java

```
package utils;

public class Vehicle {
    private char id;
    private int row;
    private int col;
    private int length;
    private boolean isHorizontal;

    public Vehicle(char id, int row, int col, int length, boolean isHorizontal) {
        this.id = id;
        this.row = row;
        this.col = col;
        this.length = length;
        this.isHorizontal = isHorizontal;
    }

    public Vehicle(Vehicle other) {
        this.id = other.id;
        this.row = other.row;
```

```
        this.col = other.col;
        this.length = other.length;
        this.isHorizontal = other.isHorizontal;
    }

    public boolean canMove(int direction, char[][] board) {
        // direction: 1 = maju (kanan/bawah), -1 = mundur (kiri/atas)
        if (isHorizontal) {
            if (direction < 0) {
                // cek gerakan ke kiri
                int newCol = col - 1;
                return newCol >= 0 && board[row][newCol] == '.';
            } else {
                // cek gerakan ke kanan
                int newCol = col + length;
                return newCol < board[0].length && board[row][newCol] == '.';
            }
        } else {
            // vertikal
            if (direction < 0) {
                // cek gerakan ke atas
                int newRow = row - 1;
                return newRow >= 0 && board[newRow][col] == '.';
            } else {
                // cek gerakan ke bawah
                int newRow = row + length;
                return newRow < board.length && board[newRow][col] == '.';
            }
        }
    }

    public void move(int direction) {
        if (isHorizontal) {
            col += direction;
        } else {
            row += direction;
        }
    }

    public char getId() {
        return id;
    }

    public int getRow() {
        return row;
    }

    public void setRow(int row) {
        this.row = row;
    }
```

```
    public void setCol(int col) {
        this.col = col;
    }

    public int getCol() {
        return col;
    }

    public int getLength() {
        return length;
    }

    public boolean isHorizontal() {
        return isHorizontal;
    }

    @Override
    public String toString() {
        return id + " at (" + row + "," + col + ") length=" + length + " " +
(isHorizontal ? "horizontal" : "vertical");
    }
}
```

3. Folder gui

LandingPage.java

```
package gui;
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import javafx.scene.control.Label;
import javafx.geometry.Pos;
import javafx.scene.control.Alert;
import javafx.scene.control.Button;
import javafx.scene.image.Image;
import javafx.scene.layout.Background;
import javafx.scene.layout.BackgroundImage;
import javafx.scene.layout.BackgroundPosition;
import javafx.scene.layout.BackgroundRepeat;
import javafx.scene.layout.BackgroundSize;
import javafx.scene.layout.VBox;
import javafx.stage.FileChooser;
import utils.RushHourGame;
public class LandingPage {
    private VBox layout;
```

```
    public LandingPage(MainApp app) {

        layout = new VBox(20);

        // set bg
        Image bgImage = new
Image(getClass().getResource("/gui/bg.png").toExternalForm());

        BackgroundImage backgroundImage = new BackgroundImage(
            bgImage,
            BackgroundRepeat.NO_REPEAT,
            BackgroundRepeat.NO_REPEAT,
            BackgroundPosition.CENTER,
            new BackgroundSize(100, 100, true, true, true, false)
        );
        layout.setBackground(new Background(backgroundImage));
        layout.setPrefSize(600, 600);

        // set button
        Label intro = new Label("Up load your configuration!");
        Button startButtonUpload = new Button("Start Upload");

        startButtonUpload.setOnAction(e -> {
            FileChooser fileChooser = new FileChooser();
            fileChooser.setTitle("Upload Puzzle File");
            fileChooser.getExtensionFilters().add(
                new FileChooser.ExtensionFilter("Text Files", "*.txt")
            );

            File file = fileChooser.showOpenDialog(layout.getScene().getWindow());
            if (file != null) {
                try {
                    String content = new String(Files.readAllBytes(file.toPath()));
                    System.out.println("Uploaded file content:\n" + content);

                    // create new game
                    RushHourGame game = new RushHourGame(content);
                    Alert successAlert = new Alert(Alert.AlertType.INFORMATION);
                    successAlert.setTitle("Load Successful");
                    successAlert.setHeaderText(null);
                    successAlert.setContentText("File loaded successfully!");
                    successAlert.showAndWait();

                    // pass to board scene
                    app.showMainBoard(game);


                } catch (IOException ex) {
                    ex.printStackTrace();
                }
```

```
            }
        });

        Label credit = new Label("This program is made by Najwa K. F. (13523043) and
Diyah S. N. (13523080)");

        layout.setAlignment(Pos.CENTER);
        // layout.getChildren().add(startButtonManual);
        layout.getChildren().add(intro);
        layout.getChildren().add(startButtonUpload);
        layout.getChildren().add(credit);

        credit.getStyleClass().add("credit");
        intro.getStyleClass().add("intro");
        startButtonUpload.getStyleClass().add("button-start");
        layout.getStyleClass().add("landing");
    }

    public VBox getLayout() {
        return layout;
    }
}
```

MainApp.java

```
package gui;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import utils.RushHourGame;
public class MainApp extends Application {
    private Stage primaryStage;

    @Override
    public void start(Stage primaryStage) {
        this.primaryStage = primaryStage;
        showLandingPage();
    }

    public void showLandingPage() {
        LandingPage landingPage = new LandingPage(this);
        Scene scene = new Scene(landingPage.getLayout(), 600, 600);

scene.getStylesheets().add(getClass().getResource("style.css").toExternalForm());
        primaryStage.setTitle("Rush Hour Game Solver");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public void showMainBoard(RushHourGame game) {
```

```
        MainBoardPage mainBoard = new MainBoardPage(game, this);
        Scene scene = new Scene(mainBoard.getLayout(), 600, 600);

scene.getStylesheets().add(getClass().getResource("style.css").toExternalForm());
        primaryStage.setTitle("Rush Hour Game Solver");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

MainBoardPage.java

```
package gui;

import javafx.animation.KeyFrame;
import javafx.animation.PauseTransition;
import javafx.animation.Timeline;
import javafx.application.Platform;
import javafx.geometry.Pos;
import javafx.scene.control.Alert;
import javafx.scene.control.Button;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.control.TextInputDialog;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.StackPane;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Font;
import javafx.scene.control.TextField;
import javafx.scene.text.Text;
import javafx.util.Duration;
import utils.RushHourGame;
import utils.State;
import utils.Vehicle;
import utils.Position;

import java.util.*;

import algorithms.Algorithm;

public class MainBoardPage {
    private RushHourGame game;
```

```
    private BorderPane layout;
    private final Integer CELL_SIZE = 60;
    private final List<String> algorithms = Arrays.asList("Uniform Cost Search",
"Greedy Best First Search", "A* Algorithm", "Beam Search");
    private final List<String> heuristics = Arrays.asList("Blocking Vehicles",
"Manhattan Distance");
    private final List<Color> otherColors = Arrays.asList(
            Color.LIGHTBLUE, Color.ORANGE, Color.PURPLE, Color.DARKCYAN,
            Color.GOLD, Color.DARKMAGENTA, Color.DARKORANGE, Color.DEEPSKYBLUE,
            Color.MEDIUMVIOLETRED, Color.DARKOLIVEGREEN
    );
    private String chosenAlgorithm = algorithms.get(0);
    private String chosenHeuristic = heuristics.get(0);
    private int beamWidth = 0;

    public MainBoardPage(RushHourGame game, MainApp app) {
        this.game = game;
        layout = new BorderPane();

        // SET BOTTOM
        Label lblDuration = new Label("Duration: -");
        Label lblSteps = new Label("Steps: -");
        Label lblNodes = new Label("Nodes: -");
        Button backButton = new Button("Back or Upload");

        backButton.setOnAction(e -> {
            app.showLandingPage();
        });

        HBox statsBox = new HBox(15, lblDuration, lblSteps, lblNodes, backButton);
        statsBox.setAlignment(Pos.CENTER);
        layout.setBottom(statsBox);
        statsBox.getStyleClass().add("stats-bar");

        // SET TOP
        TextField beamWidthInput = new TextField();
        beamWidthInput.setPromptText("input beam width here");
        beamWidthInput.setVisible(false);
        beamWidthInput.setManaged(false);

        ComboBox<String> algorithmOptions = new ComboBox<>();
        algorithmOptions.getItems().addAll(algorithms);
        algorithmOptions.setValue(algorithms.get(0));
        algorithmOptions.setOnAction(e -> {
            chosenAlgorithm = algorithmOptions.getValue();
            beamWidthInput.setVisible(chosenAlgorithm.equals("Beam Search"));
            beamWidthInput.setManaged(chosenAlgorithm.equals("Beam Search"));
        });

        ComboBox<String> heuristicOptions = new ComboBox<>();
```

```
        heuristicOptions.getItems().addAll(heuristics);
        heuristicOptions.setValue(heuristics.get(0));
        heuristicOptions.setOnAction(e -> {
            chosenHeuristic = heuristicOptions.getValue();
        });

        Button solveButton = new Button("Solve");

        HBox top = new HBox(15, algorithmOptions, heuristicOptions, solveButton);
        top.getChildren().add(beamWidthInput);
        top.setAlignment(Pos.CENTER);

        solveButton.setOnAction(e -> {
            int algo = 1, heu = 1;
            if (chosenAlgorithm.equals(algorithms.get(0))) algo = 1;
            if (chosenAlgorithm.equals(algorithms.get(1))) algo = 2;
            if (chosenAlgorithm.equals(algorithms.get(2))) algo = 3;
            if (chosenAlgorithm.equals(algorithms.get(3))) {
                algo = 4;
                beamWidth = Integer.parseInt(beamWidthInput.getText().trim());
            }

            if (chosenHeuristic.equals(heuristics.get(0))) heu = 1;
            if (chosenHeuristic.equals(heuristics.get(1))) heu = 2;

            game.solveGame(algo, heu, beamWidth);

            if (game.solution != null) {
                // update stats labels
                lblDuration.setText("Duration: " + ( game.endTime - game.startTime) + "
ms");

                lblSteps.setText("Steps: " + game.steps);
                lblNodes.setText("Nodes: " + game.nodes);

                animateSolution(game.solution);
            } else {
                new Alert(Alert.AlertType.ERROR, "No Solution Found.").showAndWait();
            }
        });

        Label header = new Label("Choose the algorithm and heuristic");
        header.getStyleClass().add("game-header");
        VBox option = new VBox(15, header, top);

        option.setAlignment(Pos.CENTER);
        layout.setTop(option);

        // SET CENTER
        layout.setCenter(setBoard());
    }
```

```
private GridPane setBoard() {
    char[][] board = new char[game.getRows()][game.getCols()];
    for (int i = 0; i < game.getRows(); i++) {
        for (int j = 0; j < game.getCols(); j++) {
            board[i][j] = game.getChar(i, j);
        }
    }
    return buildGridFromCharBoard(board);
}

private char[][] generateBoardFromState(State state) {
    int rows = game.getRows();
    int cols = game.getCols();
    char[][] board = new char[rows][cols];
    for (int i = 0; i < rows; i++) {
        Arrays.fill(board[i], '.');
    }

    for (Map.Entry<Character, Vehicle> entry : state.vehicles.entrySet()) {
        char id = entry.getKey();
        Vehicle v = entry.getValue();
        int r = v.getRow();
        int c = v.getCol();

        for (int i = 0; i < v.getLength(); i++) {
            if (v.isHorizontal()) {
                board[r][c + i] = id;
            } else {
                board[r + i][c] = id;
            }
        }
    }

    return board;
}

private GridPane buildGridFromCharBoard(char[][] board) {
    int rows = board.length;
    int cols = board[0].length;

    Set<Character> uniqueChars = new HashSet<>();
    for (char[] row : board) {
        for (char ch : row) {
            if (ch != '.') {
                uniqueChars.add(ch);
            }
        }
    }
```

```
        Map<Character, Color> colorMap = new HashMap<>();
        colorMap.put('P', Color.RED);
        colorMap.put('K', Color.LIMEGREEN);

        int colorIndex = 0;
        for (char ch : uniqueChars) {
            if (ch != 'P' && ch != 'K') {
                colorMap.put(ch, otherColors.get(colorIndex % otherColors.size()));
                colorIndex++;
            }
        }

        GridPane grid = new GridPane();
        grid.setAlignment(Pos.CENTER);

        Position exit = game.getExitPosition();

        for (int r = -1; r <= rows; r++) {
            for (int c = -1; c <= cols; c++) {
                char ch;

                boolean isExit = (exit != null && r == exit.getRow() && c ==
exit.getCol());

                if (r >= 0 && r < rows && c >= 0 && c < cols) {
                    ch = board[r][c];
                } else {
                    ch = '.';
                }

                Rectangle rect = new Rectangle(CELL_SIZE, CELL_SIZE);

                if (isExit) {
                    rect.setFill(Color.LIMEGREEN);
                } else if (r == -1 || c == -1 || r == rows || c == cols) {
                    rect.setFill(Color.BLACK);
                } else if (ch == '.') {
                    rect.setFill(Color.WHITE);
                } else {
                    rect.setFill(colorMap.getOrDefault(ch, Color.GRAY));
                }

                rect.setStroke(Color.BLACK);

                Text label = new Text(isExit ? "K" : (ch == '.' ? "" :
String.valueOf(ch)));
                label.setFont(Font.font(20));
                label.setFill(Color.BLACK);

                StackPane cell = new StackPane(rect, label);
```

```
                grid.add(cell, c + 1, r + 1);
            }
        }

        return grid;
    }


    public void animateSolution(List<State> path) {
        if (path == null || path.isEmpty()) return;

        Timeline timeline = new Timeline();
        timeline.setCycleCount(1);

        for (int i = 0; i < path.size(); i++) {
            final int index = i;
            KeyFrame frame = new KeyFrame(Duration.seconds(index * 0.5), e -> {
                char[][] board = generateBoardFromState(path.get(index));
                layout.setCenter(buildGridFromCharBoard(board));
            });
            timeline.getKeyFrames().add(frame);
        }

        timeline.setOnFinished(e -> {
            // Delay 1 second after animation ends
            PauseTransition delay = new PauseTransition(Duration.seconds(1));
            delay.setOnFinished(ev -> {
                Platform.runLater(() -> {
                    TextInputDialog dialog = new TextInputDialog("solusi.txt");
                    dialog.setTitle("Save Solution");
                    dialog.setHeaderText("Rush Hour Solved!");
                    dialog.setContentText("Enter filename to save steps (with
extension):");

                    Optional<String> result = dialog.showAndWait();
                    result.ifPresent(filename -> {
                        String outputFilename = "test/" + filename;
                        Algorithm.writeSolutionToFile(
                            outputFilename,
                            game.solution,
                            game.steps,
                            game.nodes,
                            (game.endTime - game.startTime)
                        );
                    });
                });
            });
            delay.play();
        });
```

```
        timeline.play();
    }

    public BorderPane getLayout() {
        return this.layout;
    }
}
```

style.css

```
.root {
    -fx-font-family: "Segoe UI", sans-serif;
}

.button-start {
    -fx-font-size: 16px;
    -fx-background-color: #ffe600;
    -fx-text-fill: rgb(0, 0, 0);
    -fx-font-weight: bold;
    -fx-padding: 10 20;
    -fx-background-radius: 10;
    -fx-effect: dropshadow(gaussian, rgba(0,0,0,0.75), 8, 0.5, 2, 2);
}

.button:hover {
    -fx-background-color: #d6b317;
}

.landing {
    -fx-padding: 50;
}

.credit {
    -fx-font-size: 12px;
    -fx-text-fill: #ffffff;
    -fx-font-style: italic;
    -fx-padding: 5 10 5 10;
    -fx-alignment: center;
}

.intro {
    -fx-font-size: 13px;
    -fx-text-fill: #ffffff;
    -fx-font-weight: bold;
    -fx-padding: 5 5 5 5;
    -fx-alignment: center;
    -fx-text-alignment: center;
}
```

```
.game-header {
    -fx-text-fill: black;
    -fx-font-size: 16px;
    -fx-font-weight: bold;
    -fx-font-family: "Verdana", "Arial Black", sans-serif;
    -fx-alignment: center;
}

.stats-bar {
    -fx-padding: 10px 20px;
    -fx-spacing: 30px;
    -fx-alignment: center-left;
    -fx-border-color: #6a0000;
    -fx-border-width: 1px;
    -fx-border-radius: 10px;
    -fx-background-radius: 10px;
}
```

4. Main.java

```
import algorithms.Algorithm;
import java.util.Scanner;
import utils.RushHourGame;
import utils.SaveLoad;

public class Main {
    public static void main(String[] args) {
        SaveLoad system = new SaveLoad();
        String input = system.Load();
        RushHourGame game = new RushHourGame(input);
        int algorithm;
        int heuristic;
        int beamWidth = 0;

        Scanner scanner = new Scanner(System.in);

        // Main program
        while (true) {
            System.out.println("===== Choose algorithm =====");
            System.out.println(" 1. Uniform Cost Search");
            System.out.println(" 2. Greedy Best First Search");
            System.out.println(" 3. A* Algorithm");
            System.out.println(" 4. Beam Search");

            System.out.print("Your choice (number): ");
            algorithm = scanner.nextInt();
            scanner.nextLine();
            if (algorithm > 4 || algorithm < 1) {
```

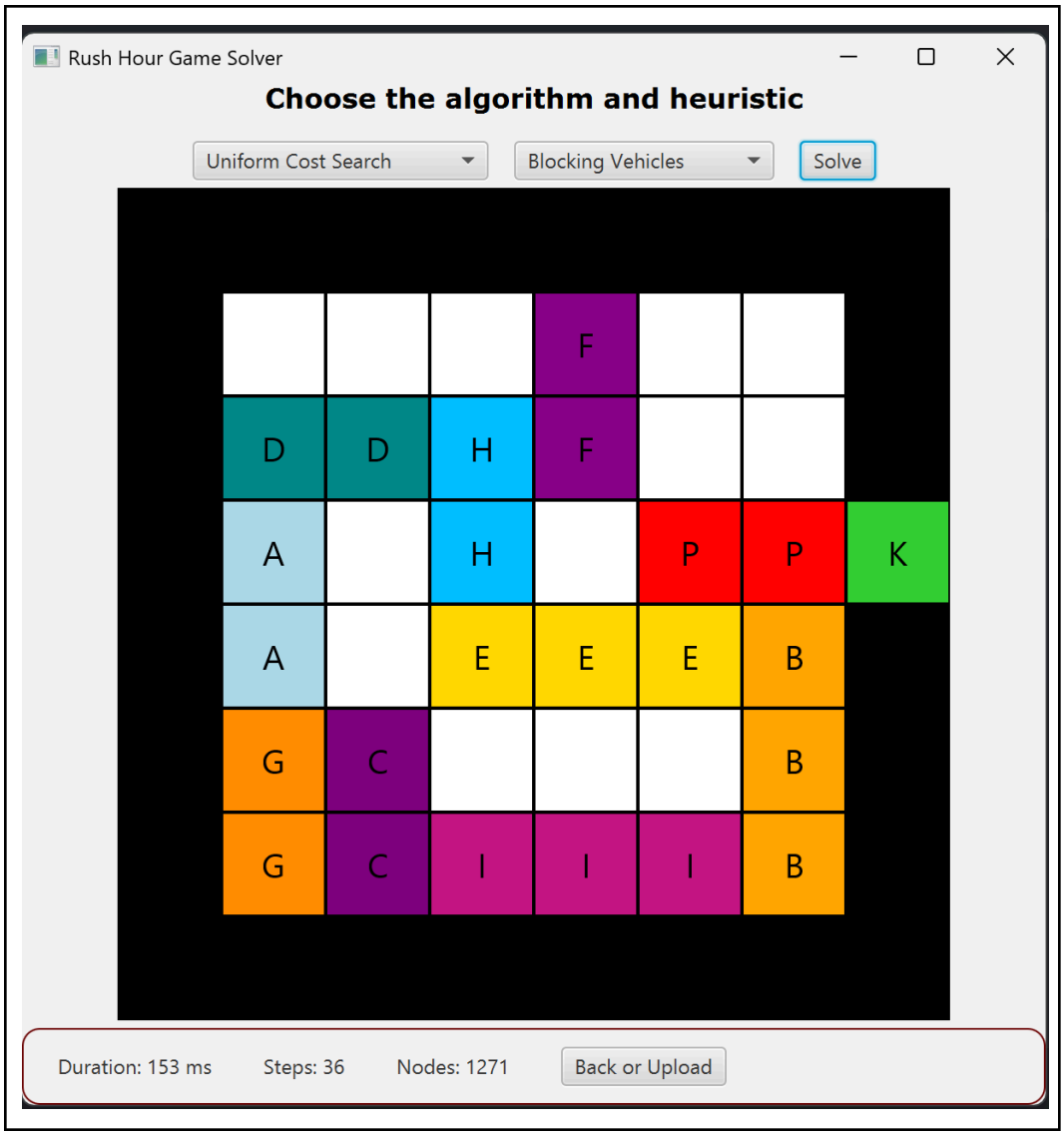
```
                System.out.println();
                continue;
            }

            if (algorithm == 4) {
                System.out.print("Enter beam width: ");
                beamWidth = scanner.nextInt();
                scanner.nextLine();
            }

            System.out.println("===== Choose heuristic =====");
            System.out.println(" 1. Blocking Vehicles");
            System.out.println(" 2. Manhattan Distance");
            System.out.print("Your choice (number): ");
            heuristic = scanner.nextInt();
            scanner.nextLine();
            if (heuristic > 2 || heuristic < 1) {
                System.out.println();
                continue;
            }
            break;
        }

        game.displayBoard();

        // solve game based on heuristic
        game.solveGame(algorithm, heuristic, beamWidth);

        // Output ke console
        if (game.solution != null) {
            // ouput file
            System.out.println("Masukkan output filename (without .txt): ");
            String filePath = scanner.nextLine();
            System.out.println("Total steps : " + game.steps);
            System.out.println("Total nodes explored: " + game.nodes);
            System.out.println("Solution found in " + (game.endTime - game.startTime) +
" ms");

            // Output ke file
            String outputFilename = "test/" + filePath + ".txt";
            Algorithm.writeSolutionToFile(
                outputFilename,
                game.solution,
                game.steps,
                game.nodes,
                (game.endTime - game.startTime)
            );
            System.out.println("Solution saved in " + outputFilename);
        } else {
            System.out.println("No solution found.");
```

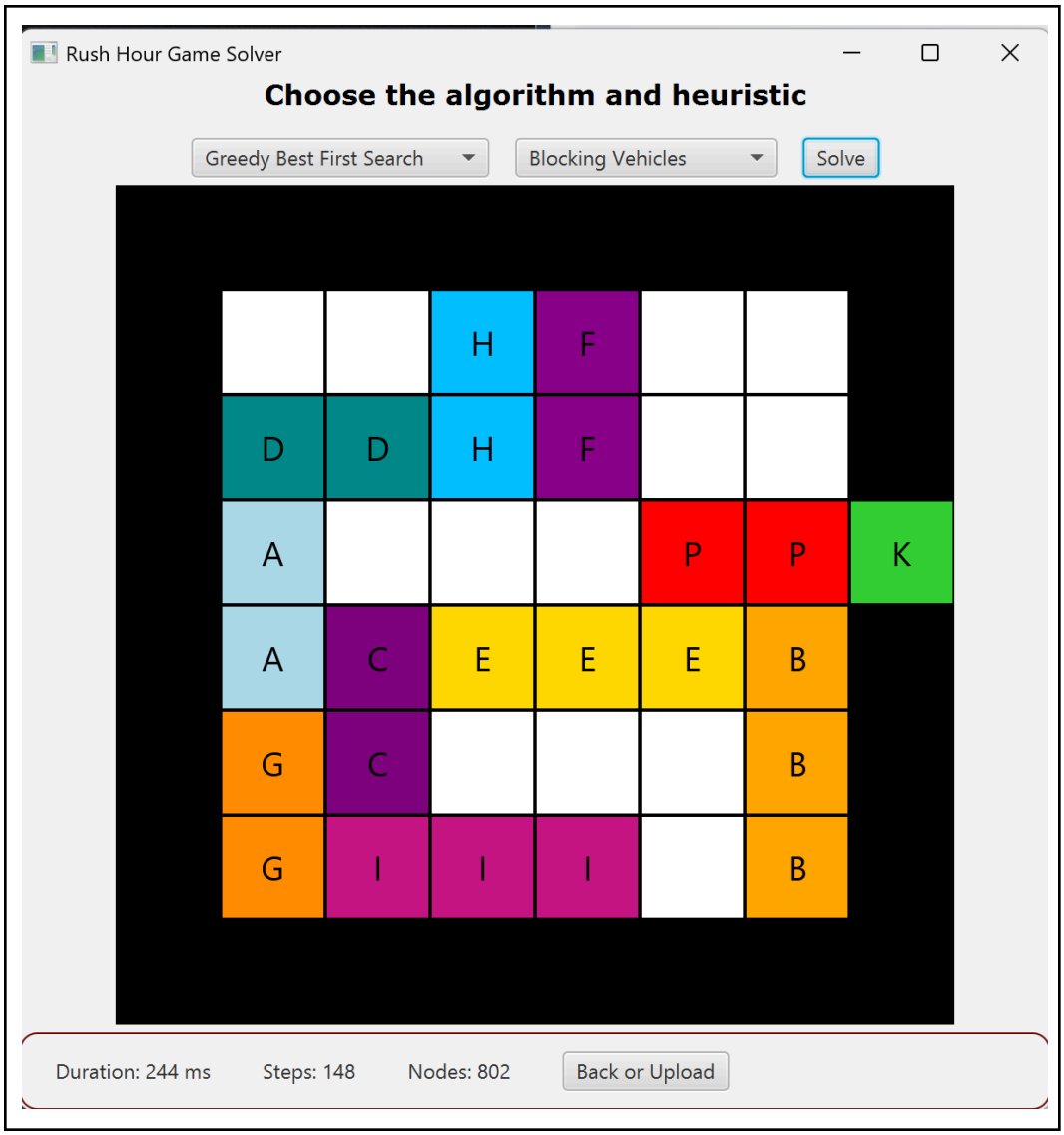
```
        }

        scanner.close();
    }
}
```

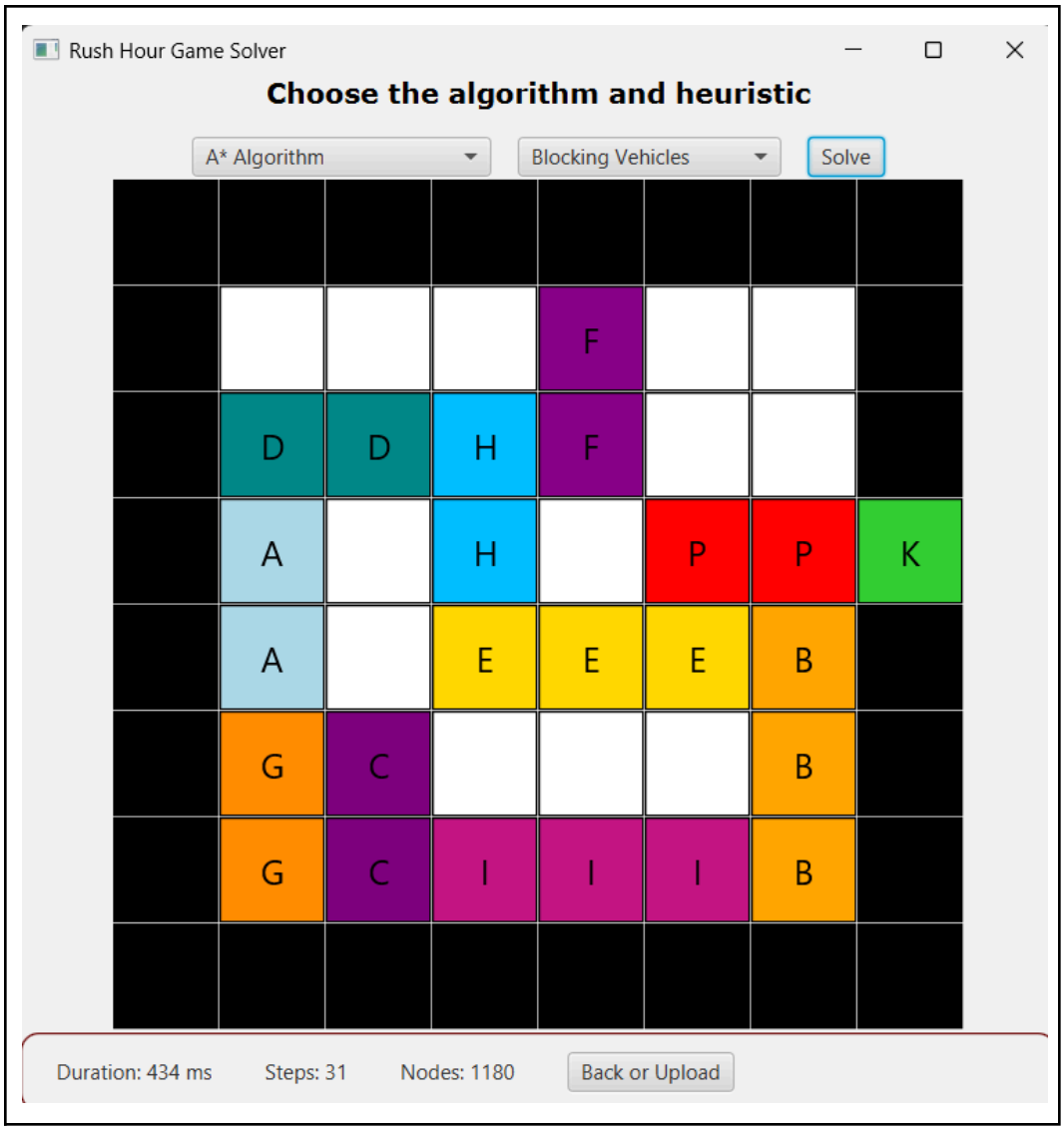
J. Test Case

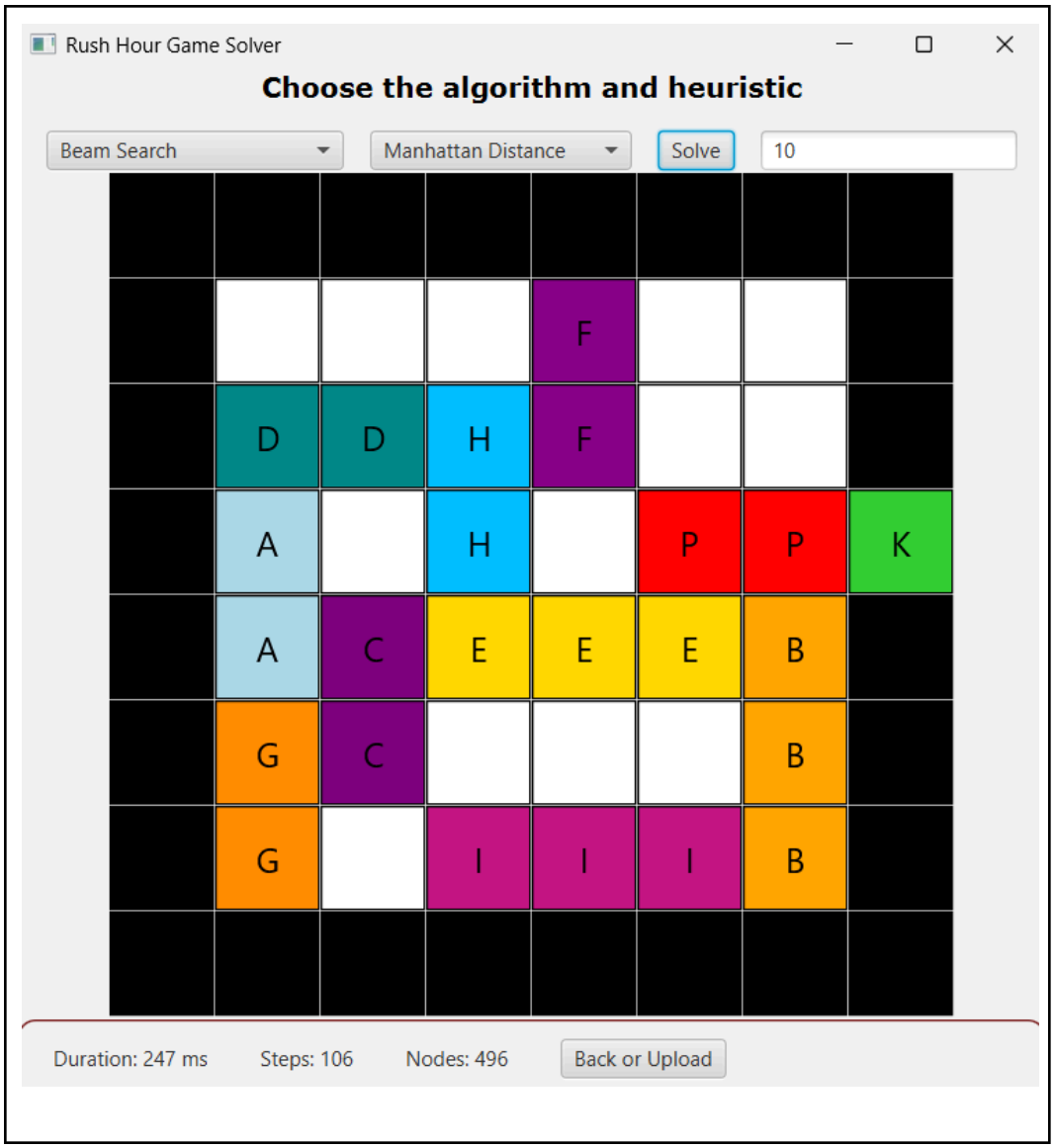
1. Test case 1

Input
6 6 10 A....B ACDD.B .CPP.BK EEEF.. G.HF.. G.HIII
Output


Rush Hour Game Solver
Choose the algorithm and heuristic
Uniform Cost Search
Blocking Vehicles
Solve
F
D D H F
A H P P K
A E E E B
G C B
G C I I I B
Duration: 153 ms
Steps: 36
Nodes: 1271
Back or Upload


Rush Hour Game Solver
Choose the algorithm and heuristic
Greedy Best First Search
Blocking Vehicles
Solve
H F
D D H F
A P P K
A C E E E B
G C B
G I I I B
Duration: 244 ms
Steps: 148
Nodes: 802
Back or Upload

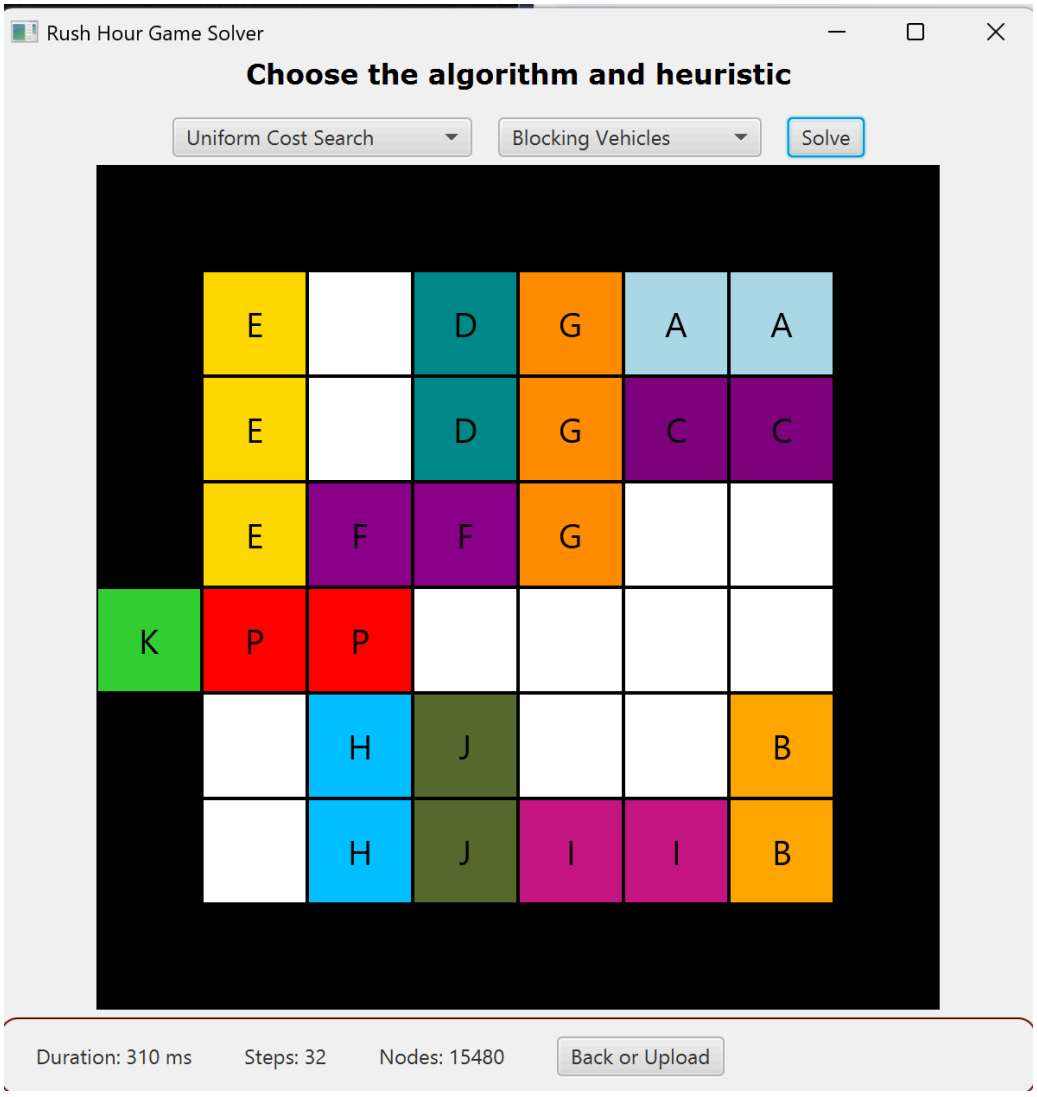

Rush Hour Game Solver
Choose the algorithm and heuristic
A* Algorithm
Blocking Vehicles
Solve
F
D D H F
A H P P K
A E E E B
G C B
G C I I I B
Duration: 434 ms
Steps: 31
Nodes: 1180
Back or Upload



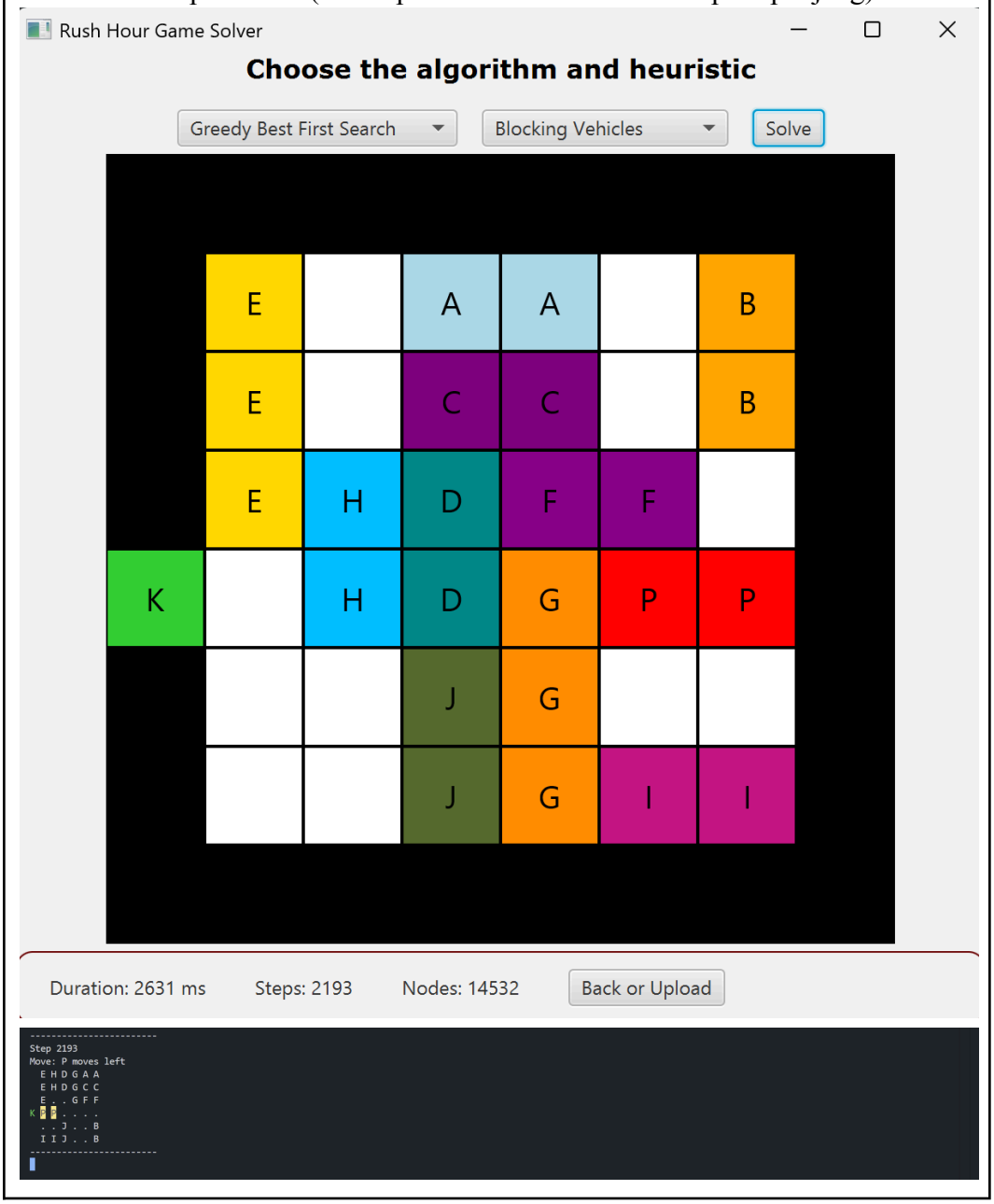
2. Test case 2

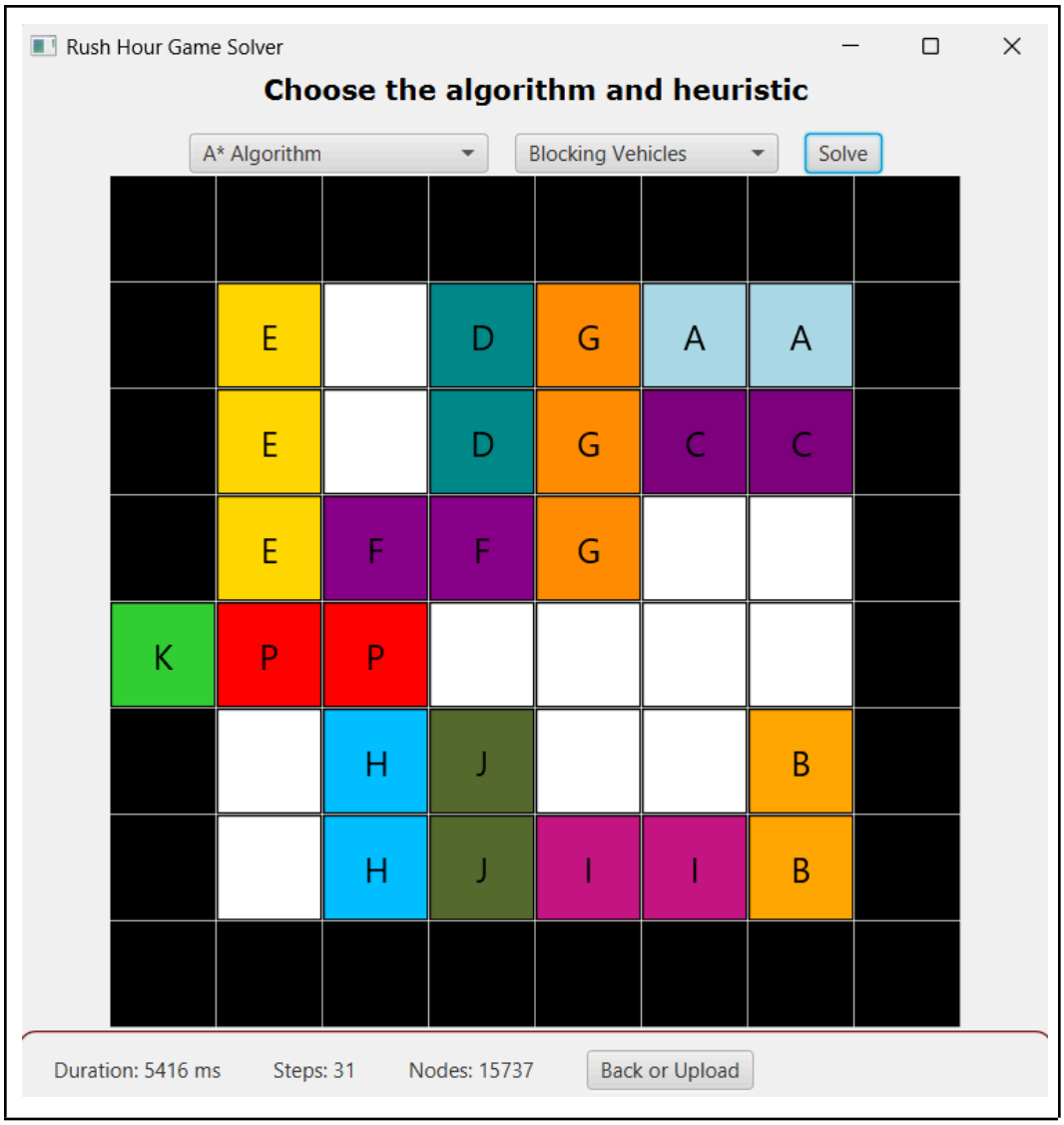
Input
6 6 11 ...AAB E.DCCB E.DGFF KEHJGPP .HJG.. ...II.

Output


Rush Hour Game Solver
Choose the algorithm and heuristic
Uniform Cost Search
Blocking Vehicles
Solve
E D G A A
E D G C C
E F F G
K P P
H J B
H J I I B
Duration: 310 ms
Steps: 32
Nodes: 15480
Back or Upload

Solusi terlihat pada CLI (solusi pada GUI dilewati karena path panjang).

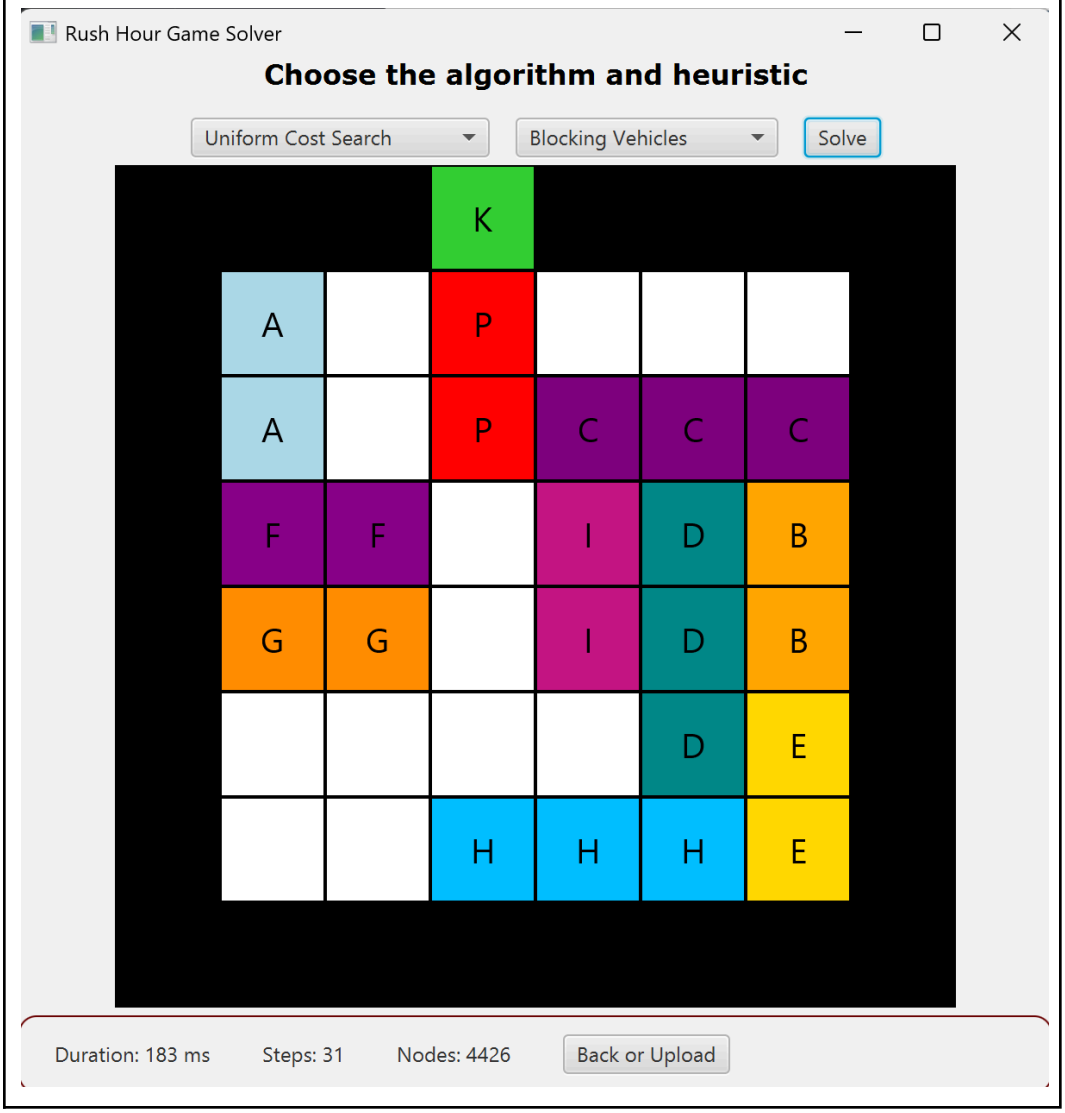


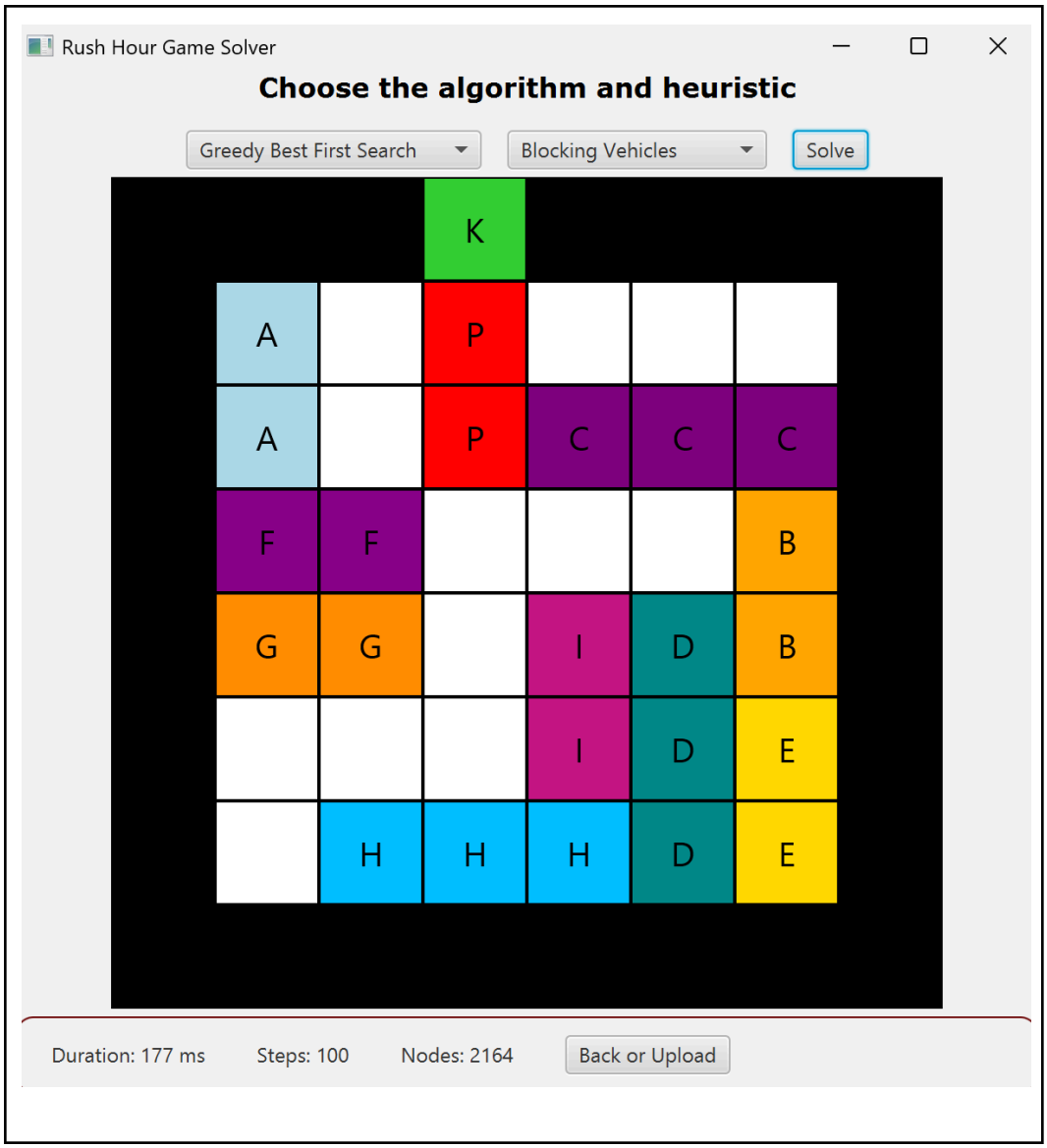


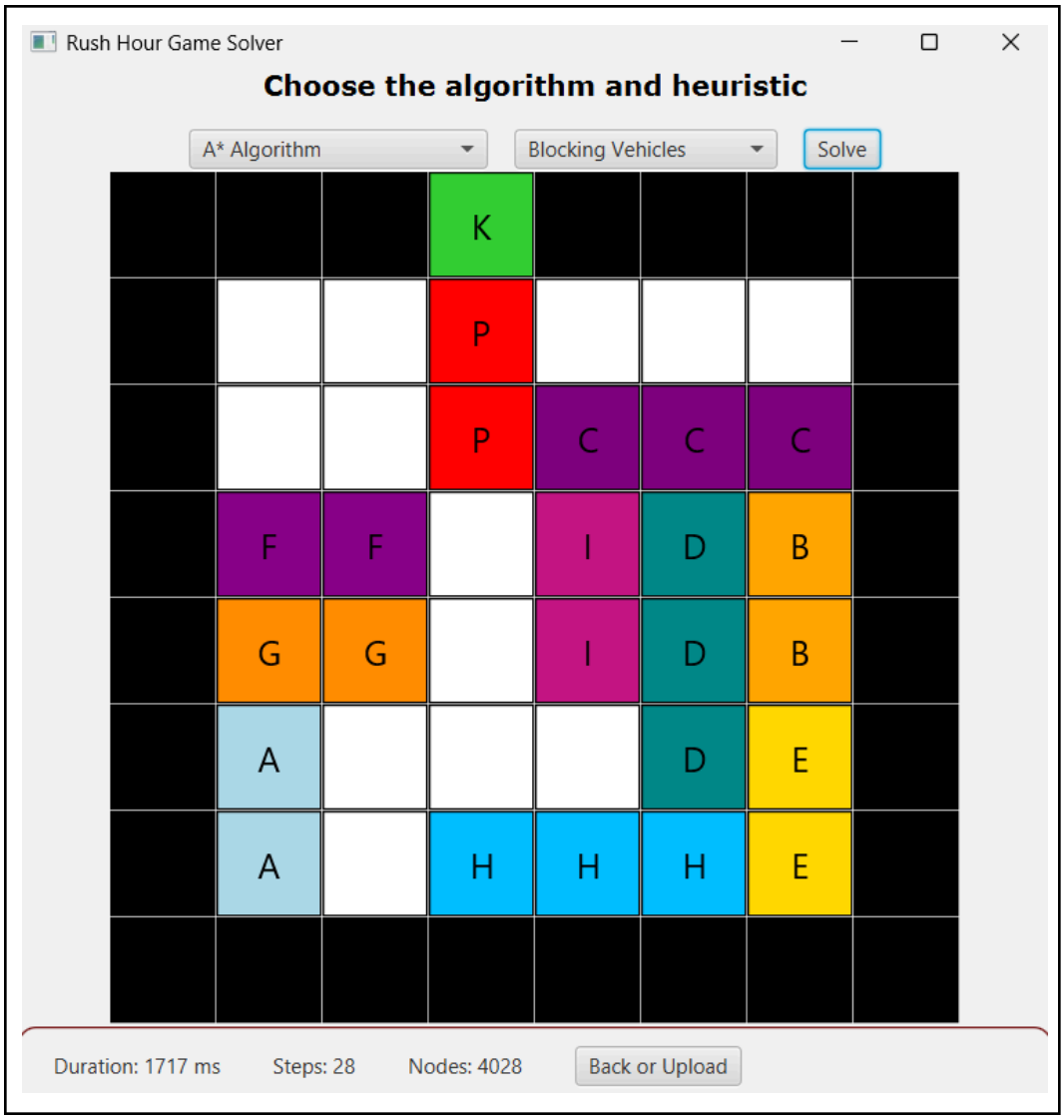
3. Test case 3

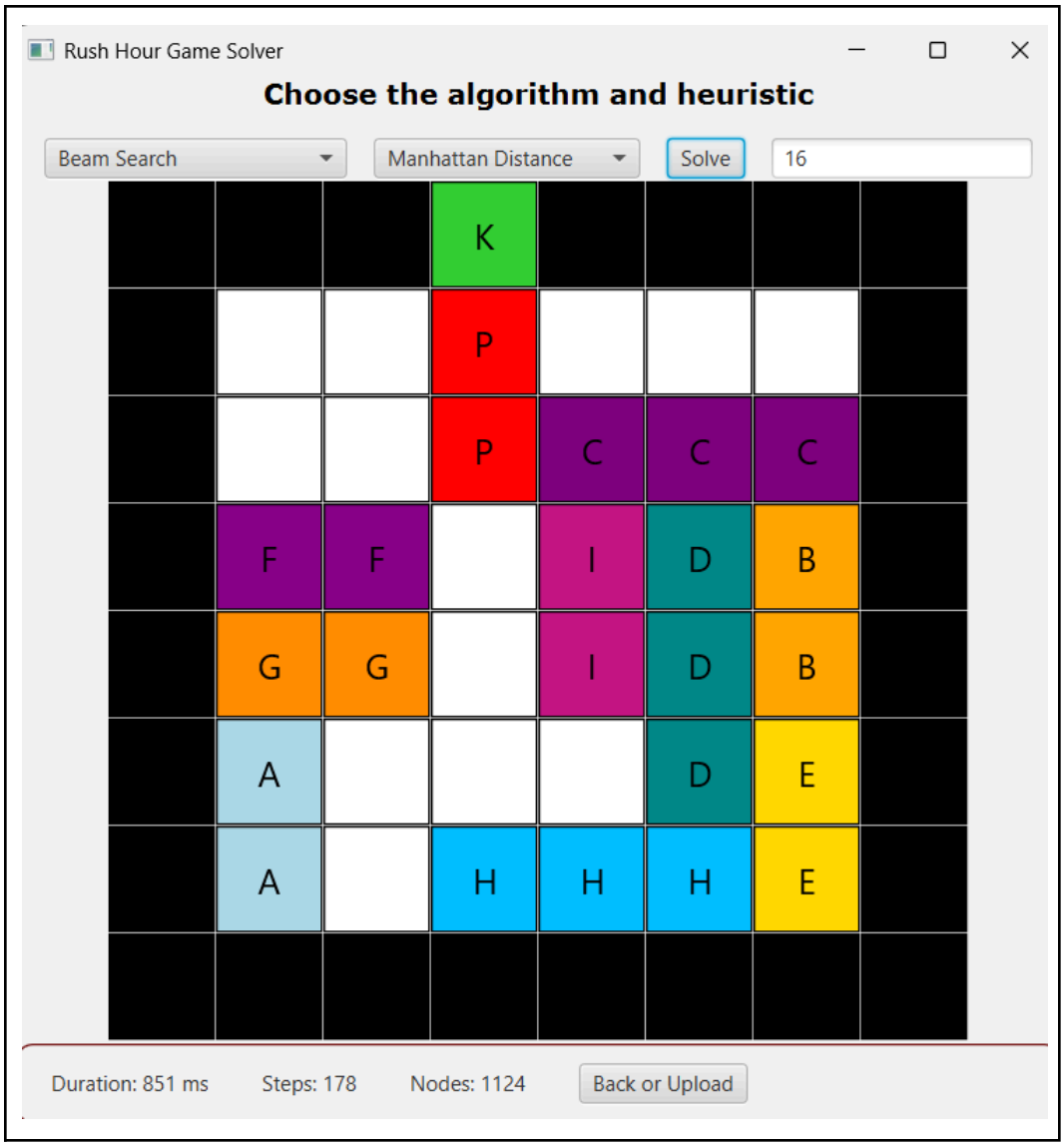
Input
6 6 10   K A...D. ACCCD. FFP.DB ..PGGB ...I.E HHHI.E

Output


Rush Hour Game Solver
Choose the algorithm and heuristic
Uniform Cost Search
Blocking Vehicles
Solve
K
A
P
A
P
C
C
C
F
F
I
D
B
G
G
I
D
B
D
E
H
H
H
E
Duration: 183 ms
Steps: 31
Nodes: 4426
Back or Upload


Rush Hour Game Solver
Choose the algorithm and heuristic
Greedy Best First Search
Blocking Vehicles
Solve
K
A
P
A
P
C
C
C
F
F
B
G
G
I
D
B
I
D
E
H
H
H
D
E
Duration: 177 ms
Steps: 100
Nodes: 2164
Back or Upload

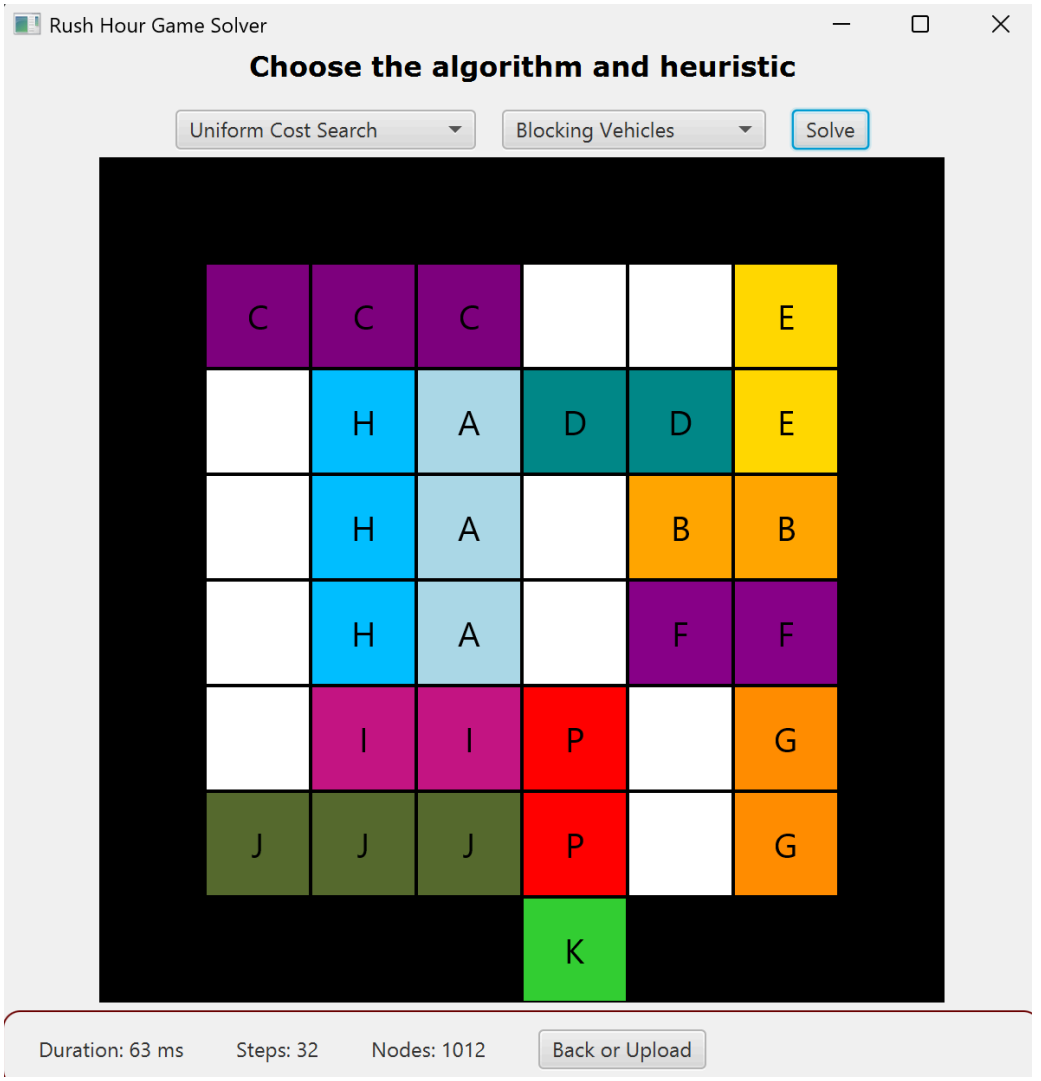

Rush Hour Game Solver
Choose the algorithm and heuristic
A* Algorithm
Blocking Vehicles
Solve
K
P
P
C
C
C
F
F
I
D
B
G
G
I
D
B
A
D
E
A
H
H
H
E
Duration: 1717 ms
Steps: 28
Nodes: 4028
Back or Upload

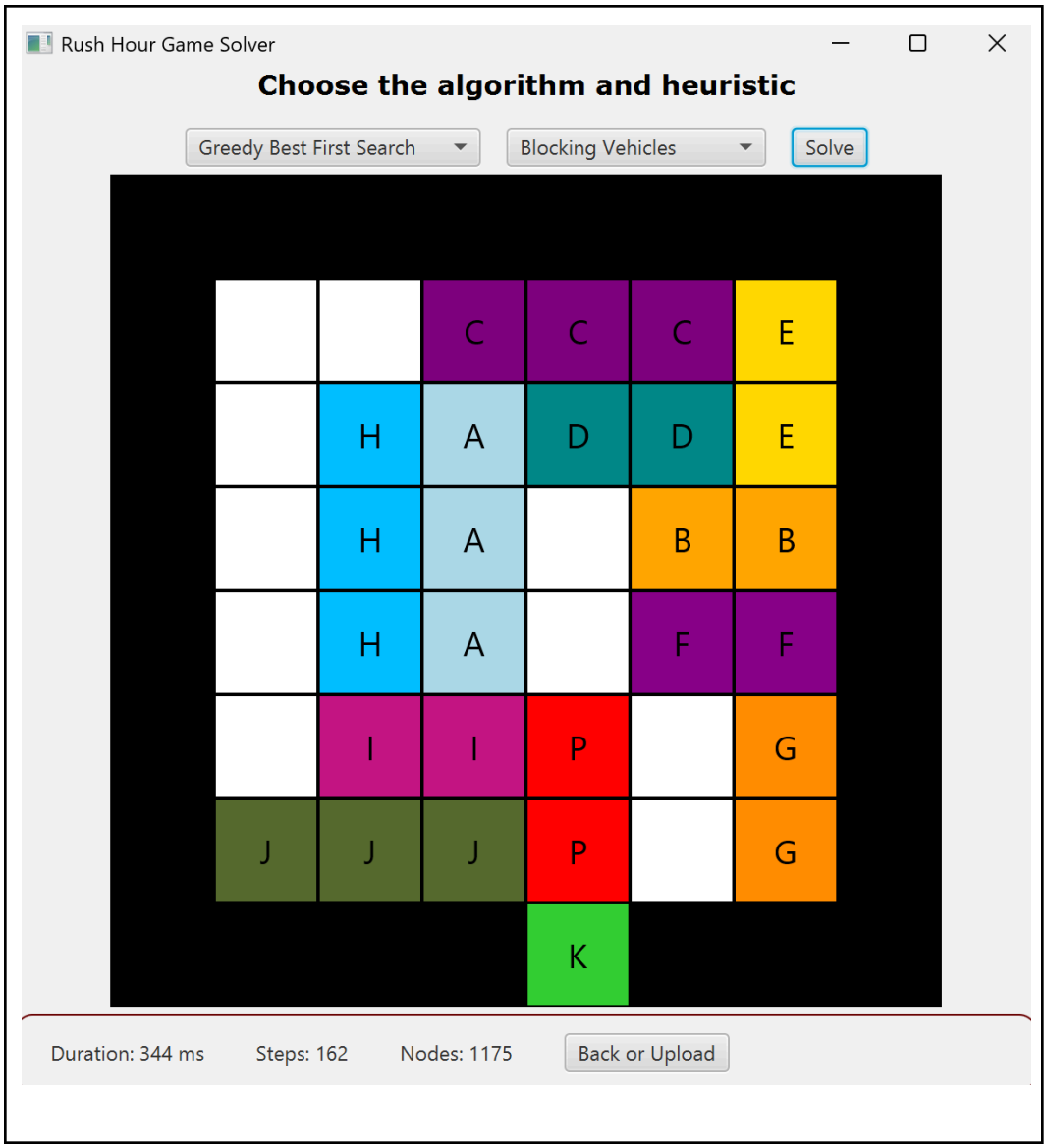


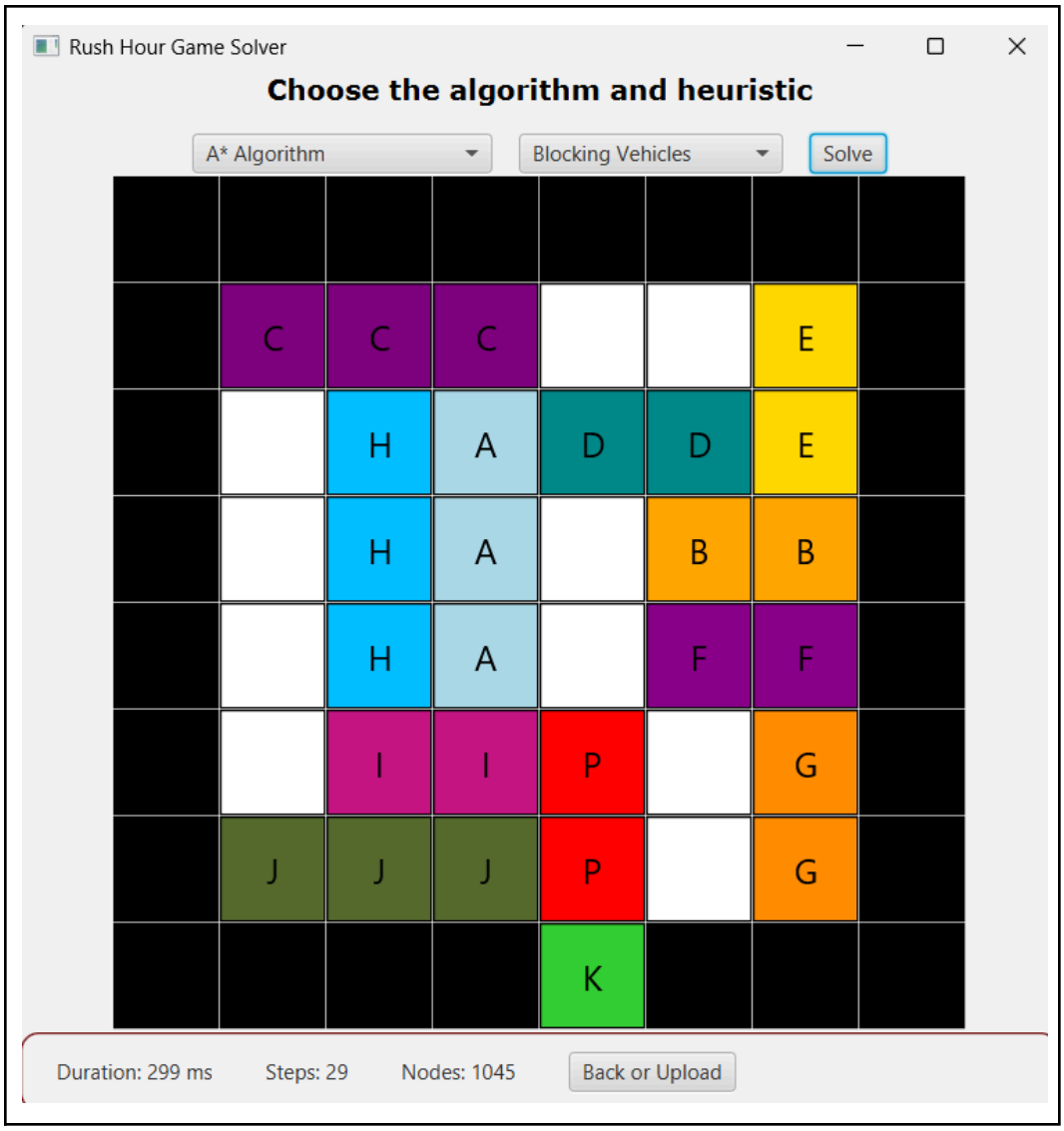
4. Test case 4

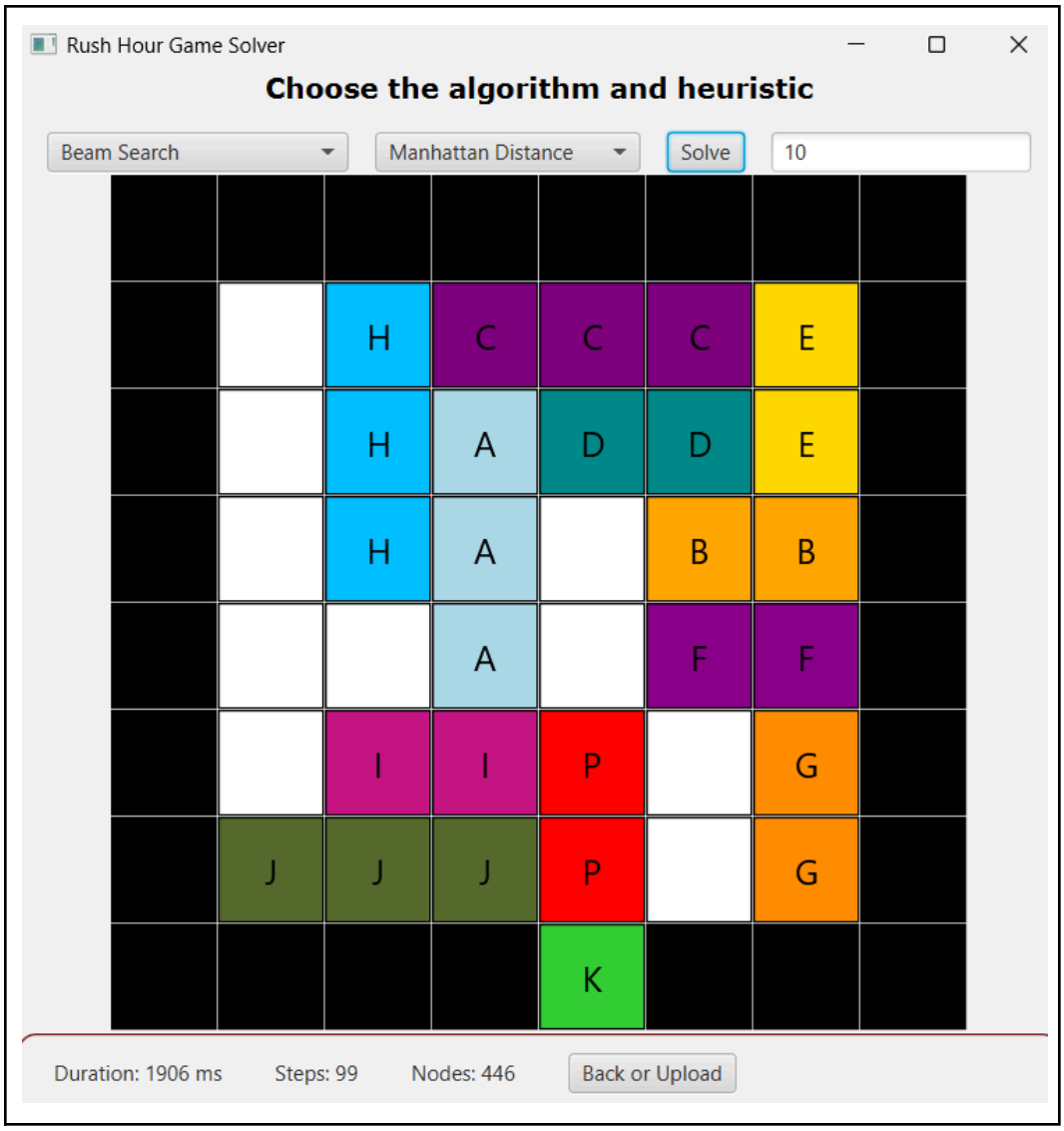
Input
6 6 11 ..ACCC ..ADDE BBAP.E .HPPFF .HII.G .HJJJG K

Output


Rush Hour Game Solver
Choose the algorithm and heuristic
Uniform Cost Search
Blocking Vehicles
Solve
C C C
E
H A D D E
H A B B
H A F F
I I P G
J J J P G
K
Duration: 63 ms
Steps: 32
Nodes: 1012
Back or Upload


Rush Hour Game Solver
Choose the algorithm and heuristic
Greedy Best First Search
Blocking Vehicles
Solve
C C C E
H A D D E
H A B B
H A F F
I I P G
J J J P G
K
Duration: 344 ms
Steps: 162
Nodes: 1175
Back or Upload


Rush Hour Game Solver
Choose the algorithm and heuristic
A* Algorithm
Blocking Vehicles
Solve
C C C E
H A D D E
H A B B
H A F F
I I P G
J J J P G
K
Duration: 299 ms
Steps: 29
Nodes: 1045
Back or Upload



K. Hasil Analisis *PathFinding*

Berdasarkan hasil test case, algoritma UCS memiliki performa yang cukup baik jika dibanding algoritma lainnya. Hal ini dapat dilihat pada banyaknya step yang dilalui untuk mencapai tujuan. Dalam kasus terburuk, kompleksitas dari algoritma UCS adalah O(b^c) dengan b adalah *branching factor* atau rata-rata jumlah cabang per simpul dan c adalah biaya minimum solusi optimal, yang dalam hal ini adalah jumlah langkah dari *state* awal ke *state* tujuan. Pada operasi utama, poll() dari priority queue memiliki kompleksitas O(log n), pengecekan goal O(1), eksplorasi O(b), dan penambahan ke priority queue O(b log n). Sehingga kompleksitas waktu total adalah O(b^c x c x log b).

Greedy Best First Search mengeksplorasi lebih banyak simpul dan langkah dibanding UCS. Dalam kasus terburuk, algoritma ini memiliki kompleksitas O(b^d) dengan d adalah kedalaman solusi. Selain itu poll() dari priority queue memiliki kompleksitas O(log n), eksplorasi O(b), pengurutan tetangga O(b log b), dan penambahan ke priority queue O(b log n). Sehingga kompleksitas total adalah O(b^d x d x log b).

Algoritma A* bergantung pada dua hal, yaitu f(n) = h(n) + g(n) sehingga algoritma ini bergantung pada biaya estimasi dari heuristik dan cost untuk mencapai state saat ini. Oleh karena itu, kompleksitas waktu nya adalah sebesar O(b^d). Dalam kasus terbaik, jika heuristik yang digunakan akurat maka algoritma ini dapat memiliki kompleksitas O(d).

Algoritma *beam search* menggunakan pendekatan bfs dengan membatasi penggunaan memori dengan cara membatasi jumlah node yang ditelusuri. Hal ini membuat kompleksitas waktu nya menjadi sebesar O(b*k*d) dengan b adalah jumlah rata-rata suksesor node, k jumlah node maksimal yang disimpan, d kedalaman solusi yang ditemukan.

Berdasarkan hasil analisis terhadap empat test case yang telah dicoba, terlihat bahwa algoritma yang menghasilkan jalur paling optimal adalah algoritma A*. Hal ini secara teori dapat dijelaskan karena nilai f(n) untuk A* mempertimbangkan dua hal yaitu cost dan nilai heuristik sehingga algoritma ini akan dapat memilih dengan lebih bijak node mana saja yang perlu untuk dibangkitkan.

Jika ditinjau dari durasi pencarian, maka algoritma yang paling bagus adalah algoritma UCS. Hal ini terjadi karena algoritma ini tidak menggunakan heuristik dalam pengambilan keputusannya sehingga proses pencarian akan relatif lebih cepat dibandingkan algoritma lainnya.

Jadi dapat disimpulkan bahwa dari segi jumlah langkah yang diambil, algoritma A* adalah yang paling optimum. Sedangkan jika ditinjau dari durasi pencarian, algoritma UCS adalah yang paling optimum. Algoritma *best first search* dan *beam search* memiliki performa yang kurang baik karena algoritma ini memungkinkan memilih langkah yang optimum lokal namun tidak optimum global sehingga langkahnya relatif lebih banyak dibanding algoritma lain. Jika dibandingkan, pada kasus tertentu, *beam search*

menghasilkan solusi yang lebih optimal daripada *best first search* karena pembatasan node yang ditelusuri. Namun pada kasus kasus kompleks, pemilihan angka *beamwidth* yang terlalu kecil membuat solusi tidak dapat ditemukan karena ada kemungkinan jalur solusi terpangkas.

L. Implementasi Bonus

1. Fungsi Heuristik Alternatif

   Pada tugas kecil ini, digunakan dua pendekatan fungsi heuristik yaitu jumlah blok penghalang dan jarak blok merah ke pintu keluar. Untuk heuristik jumlah blok penghalang, semakin sedikit penghalang yang menghalangi jalur maka semakin baik niali f(n) nya sehingga state tersebut akan menempati urutan yang lebih awal di *priority queue.* Fungsi heuristik tersebut merupakan fungsi heuristic utama dalam algoritma ini. Sedangkan untuk heuristik alternatif digunakan pendekatan dengan menggunakan *manhattan distance.* Namun, karena kendaraan P dan pintu keluar selalu sejajar maka pendekatan ini sama saja dengan menghitung jarak antara target ke pintu keluar.

2. Algoritma Alternatif

   Pada tugas kecil ini, algoritma alternatif yang digunakan adalah *beam searching.* Algoritma ini hampir mirip pendekatannya dengan algoritma *best first search* yang menggunakan nilai heuristik sebagai pembanding sehingga f(n) = h(n). Namun yang membedakan adalah, algoritma ini membatasi jumlah node yang dapat dicek sehingga dapat berpeluang untuk mempercepat pencarian dengan membuang hal-hal yang tidak perlu dicek. Akan tetapi, disisi lain algoritma ini memiliki kemungkinan tidak menemukan solusi ketika batas yang dimasukkan terlalu kecil. Hal tersebut terjadi karena pemangkasan node yang dapat dicek dilakukan secara drastis sehingga memungkinkan solusi tidak ditemukan.

3. Graphical User Interface (GUI)

   GUI pada program ini diaplikasikan dengan JavaFX. JavaFX dipilih karena memiliki kemampuan untuk membuat antarmuka pengguna yang kaya dan visual,

fleksibilitas lintas platform, dan dukungan untuk animasi dan efek visual. Kakas JavaFX terdapat pada repositori di folder lib.

Seluruh implementasi GUI ada pada folder gui. Folder ini terdiri dari beberapa file. File style.css merupakan file CSS untuk tampilan program. File bg.png adalah file gambar background yang digunakan saat program pertama kali dibuka. File MainApp.java adalah file utama untuk GUI. Pada file ini, diimplementasikan metode-metode untuk menampilkan halaman pada program. File LandingPage.java berisi implementasi kode untuk menampilkan halaman utama program dan unggah file. File MainBoardPage.java berisi implementasi penampilan papan, input pengguna, dan animasi.

M. Lampiran

Pranala *repository* : https://github.com/najwakahanifatima/Tucil3_13523043_13523080

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