LAPORAN TUGAS KECIL #2
PENYELESAIAN PUZZLE RUSH HOUR MENGGUNAKAN ALGORITMA PATHFINDING

IF2211– Strategi Algoritma

Dosen:

Dr. Ir. Rinaldi, M.T.

Anggota Kelompok:

Raudhah Yahya Kuddah 13122003

Juan Sohuturon Arauna Siagian 18222086

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2024

Daftar Isi

Rumusan Masalah

Diberikan sebuah papan permainan logika *Rush Hour* berbasis *grid* berukuran 6x6 yang terdiri dari sekumpulan *piece* atau kendaraan dengan posisi, ukuran, dan orientasi tertentu, di mana salah satunya merupakan *primary piece* yang harus dikeluarkan dari papan melalui sebuah pintu keluar. Setiap *piece* hanya dapat bergerak lurus sesuai orientasinya (horizontal atau vertikal), dan tidak dapat saling menembus satu sama lain maupun keluar dari batas papan dengan pengecualian ketika *primary piece* melalui pintu keluar.

Tujuan dari permainan adalah untuk menemukan urutan gerakan yang optimal atau jumlah langkah sesedikit mungkin agar *primary piece* dapat keluar dari papan melalui pintu keluar. Oleh karena itu, diperlukan suatu pendekatan algoritmik untuk menelusuri ruang kemungkinan konfigurasi papan dan menentukan solusi optimal.

Komponen utama dari permasalahan ini adalah papan permainan sebagai input yang direpresentasikan sebagai matriks dua dimensi berisi informasi posisi dan orientasi masing-masing *piece*. Setiap konfigurasi papan merupakan *state*, dan setiap gerakan menghasilkan transisi antar *state*. Untuk menyelesaikan permasalahan ini, digunakan pendekatan algoritmik seperti *Uniform Cost Search, Greedy Best First Search (BFS), A* Search*, atau metode pencarian lainnya untuk menelusuri solusi dengan jumlah langkah minimum agar *primary piece* mencapai pintu keluar.

Tujuan

1. Memahami dan Mengimplementasikan algoritma pencarian jalur (*pathfindin*g) yaitu *Greedy Best First Search, Uniform Cost Search (UCS),* dan *A** dalam menyelesaikan permainan logika *Rush Hour.*

2. Menganalisis performa masing-masing algoritma dalam menemukan solusi optimal berdasarkan jumlah langkah, waktu komputasi, serta penggunaan heuristik.

3. Mendesain dan menerapkan fungsi heuristik yang sesuai, minimal satu, untuk membantu algoritma pencarian dalam mengestimasi jarak atau langkah menuju solusi.

Penjelasan Algoritma

1. Penjelasan Teori Algoritma *Uniform Cost Search*

Uniform Cost Search (UCS) adalah algoritma pencarian jalur yang bertujuan untuk menemukan solusi dengan biaya total paling rendah dari titik awal ke titik tujuan. UCS menggunakan struktur data antrian prioritas untuk selalu memilih *node* dengan biaya kumulatif terkecil terlebih dahulu. Hal ini menjadikannya algoritma yang lengkap dan optimal, selama semua biaya antar langkah bernilai non-negatif. UCS sangat cocok digunakan untuk kasus pencarian dengan biaya langkah yang seragam, seperti pada permainan *Rush Hour*.

Dalam konteks permainan *Rush Hour*, setiap konfigurasi papan dianggap sebagai sebuah *state*, dan setiap gerakan kendaraan membentuk transisi menuju *state* baru dengan biaya tambahan satu langkah. UCS akan menelusuri seluruh kemungkinan gerakan kendaraan dan menyimpan semua *state* yang dihasilkan ke dalam antrian prioritas, lalu memilih *state* dengan jumlah langkah terkecil untuk dievaluasi berikutnya. Proses ini akan terus dilakukan hingga ditemukan konfigurasi di mana *primary piece* dapat mencapai pintu keluar.

Keunggulan UCS terletak pada kemampuannya menemukan solusi dengan jumlah langkah paling sedikit, tanpa perlu memperkirakan seberapa dekat suatu state dengan tujuan. Namun, karena UCS tidak menggunakan *heuristic*, pencarian dapat menjadi lebih luas dan memakan waktu di ruang pencarian yang besar. Meski begitu, UCS tetap menjadi pendekatan dasar yang kuat dan dapat dijadikan pembanding untuk algoritma lain seperti *A** yang memanfaatkan *heuristic* tambahan.

2. Penjelasan Teori Algoritma *Greedy Best First Search*

Greedy Best First Search (GBFS) adalah algoritma pencarian jalur yang memilih jalur berdasarkan prediksi jarak terdekat ke tujuan, bukan total biaya dari titik awal. GBFS menggunakan *heuristic function* (h(n)) untuk memperkirakan seberapa dekat sebuah *state* dengan tujuan, lalu selalu memilih state dengan nilai *heuristic* terkecil. Berbeda dengan UCS yang mempertimbangkan biaya kumulatif, GBFS hanya fokus pada arah pencarian menuju tujuan, sehingga prosesnya bisa lebih cepat meskipun tidak selalu optimal.

Dalam permainan *Rush Hour,* setiap konfigurasi papan dianggap sebagai sebuah *state*, dan *primary piece* harus digerakkan ke pintu keluar. GBFS akan menggunakan *heuristic*, misalnya jumlah kendaraan yang menghalangi jalur *primary piece* ke pintu keluar, sebagai dasar pemilihan state selanjutnya. Algoritma ini kemudian mengevaluasi *state* dengan nilai *heuristic* paling kecil, berharap dapat segera mencapai solusi dengan eksplorasi minimum.

Kelebihan utama GBFS adalah kecepatannya dalam menemukan solusi, terutama pada ruang pencarian yang besar. Namun, karena hanya mempertimbangkan *heuristic* tanpa memperhitungkan biaya dari awal, algoritma ini tidak menjamin solusi optimal. GBFS dapat tersesat jika *heuristic* yang digunakan tidak akurat atau tidak informatif. Meski begitu, GBFS tetap menjadi pendekatan yang efisien untuk kasus-kasus di mana kecepatan pencarian lebih penting daripada optimalitas solusi.

3. Penjelasan Teori Algoritma *A**

*A** adalah algoritma pencarian jalur yang menggabungkan keunggulan dari *Uniform Cost Search* (UCS) dan *Greedy Best First Search* (GBFS). *A** menggunakan dua komponen utama dalam proses pencarian, yaitu biaya dari titik awal ke state saat ini *g(n)* dan perkiraan biaya dari *state* tersebut ke tujuan *h(n)*. Total biaya ini dituliskan sebagai *f(n) = g(n) + h(n)*. Dengan mempertimbangkan biaya aktual dan estimasi jarak ke tujuan, *A** mampu melakukan pencarian secara efisien sekaligus optimal.

Dalam konteks permainan *Rush Hour, A** menganggap setiap konfigurasi papan sebagai *state*, dan setiap gerakan kendaraan sebagai langkah menuju *state* baru. Fungsi *g(n)* merepresentasikan jumlah langkah yang telah dilakukan dari awal permainan, sedangkan *h(n)* dapat berupa jumlah kendaraan yang menghalangi *primary piece*, atau jarak langsung dari *primary piece* ke pintu keluar. Dengan menyeimbangkan kedua nilai ini, *A** akan mengevaluasi *state* yang paling menjanjikan berdasarkan kombinasi antara kemajuan yang sudah dicapai dan estimasi sisa langkah.

Kelebihan utama *A** adalah kemampuannya menemukan solusi optimal dengan eksplorasi yang efisien, asalkan *heuristic* yang digunakan bersifat *admissible* atau tidak melebih-lebihkan estimasi biaya ke tujuan. Dalam permainan seperti *Rush Hour, A** sering menjadi pilihan paling tepat karena mampu menavigasi ruang pencarian yang kompleks dan menghasilkan solusi dengan jumlah langkah minimum. Namun, *A** tetap memiliki tantangan

dalam efisiensi memori karena banyaknya *state* yang harus disimpan dan dievaluasi, terutama dalam konfigurasi papan yang sangat padat.

4. Analisis Algoritma pada Program

4.1 *Cost* dan Hubungannya dengan algoritma pencarian

Dalam algoritma pencarian, fungsi evaluasi biaya sangat penting untuk menentukan urutan ekspansi simpul. Secara umum, biaya total yang digunakan untuk mengevaluasi suatu simpul disebut sebagai *f(n)*. Nilai ini bervariasi tergantung pada algoritma: untuk UCS, *f(n) = g(n)*, yaitu biaya kumulatif dari simpul awal ke simpul saat ini; untuk *Greedy Best First Search, f(n) = h(n)*, yaitu estimasi jarak dari simpul saat ini ke tujuan; dan untuk *A*, f(n) = g(n) + h(n)*, yang merupakan kombinasi keduanya. Biaya ini menjadi dasar pengurutan antrian prioritas dalam setiap algoritma pencarian. Dalam konteks permainan Rush Hour, biaya biasanya dihitung sebagai jumlah langkah yang telah dilakukan, sehingga *g(n)* = jumlah langkah dari konfigurasi awal ke konfigurasi saat ini. Heuristic *h(n)* bisa berupa jumlah kendaraan yang menghalangi jalur *primary piece* ke pintu keluar atau jarak langsung ke pintu. Fungsi *f(n)* pada *A** memberikan pendekatan seimbang antara pencarian optimal dan efisien. Oleh karena itu, pemahaman terhadap struktur biaya sangat penting dalam merancang solusi cerdas pada permainan berbasis *grid* seperti *Rush Hour*.

4.2 *Uniform Cost Search* dalam konteks *Rush Hour*

Uniform Cost Search (UCS) adalah algoritma yang menjamin solusi optimal dengan menelusuri simpul berdasarkan total biaya terkecil dari awal hingga simpul tersebut, atau *f(n) = g(n)*. Dalam permainan *Rush Hour*, setiap pergerakan kendaraan dianggap memiliki biaya tetap seperti satu langkah, sehingga UCS akan mengevaluasi semua konfigurasi papan berdasarkan total jumlah langkah yang diperlukan untuk mencapainya. UCS identik dengan *Breadth-First Search* (BFS) ketika semua langkah memiliki biaya yang sama, seperti pada *Rush Hour*, sehingga urutan *node* yang diekspansi dan path yang ditemukan akan sama. Keunggulan UCS adalah jaminan optimalitas dalam menemukan solusi dengan langkah paling sedikit, namun algoritma ini bisa menjadi lambat jika ruang pencarian sangat luas karena ia mengeksplorasi banyak *state* tanpa panduan arah ke tujuan. Dalam implementasinya, UCS akan terus memilih konfigurasi dengan

g(n) terendah dari antrian prioritas sampai mencapai konfigurasi ketika *primary piece* dapat keluar dari papan.

4.3 *Greedy Best First Search* dalam konteks *Rush Hour*

Greedy Best First Search (GBFS) adalah algoritma pencarian yang hanya mempertimbangkan estimasi jarak ke tujuan, yaitu *f(n) = h(n)*. Dalam permainan *Rush Hour*, heuristic h(n) bisa berupa jumlah kendaraan yang menghalangi jalan *primary piece* menuju pintu keluar, atau jarak langsung dari posisi akhir *primary piece* ke pintu keluar. Dalam implementasi program ini, heuristik seperti LAZY digunakan untuk mengukur jarak sisa antara ujung primary piece dan pintu keluar secara langsung, tanpa mempertimbangkan hambatan. Dengan hanya mengandalkan nilai h(n), GBFS cenderung lebih cepat karena tidak memperhitungkan biaya dari titik awal (g(n)), tetapi kelemahannya adalah ia tidak menjamin solusi yang optimal. Hal ini disebabkan oleh sifatnya yang serakah, yaitu selalu memilih *state* yang tampak paling dekat ke tujuan, bahkan jika itu bukan jalur terbaik secara keseluruhan. Oleh karena itu, GBFS tidak menjamin solusi optimal dalam permainan *Rush Hour*. Algoritma ini cocok digunakan ketika kecepatan lebih diprioritaskan daripada kualitas solusi, namun pengguna harus berhati-hati terhadap kemungkinan "tersesat" di jalur suboptimal.

4.4 *A** dalam konteks *Rush Hour*

*A** merupakan algoritma pencarian yang menggabungkan kelebihan UCS dan GBFS dengan menggunakan fungsi evaluasi *f(n) = g(n) + h(n),* di mana g(n) adalah total langkah dari awal dan h(n) adalah estimasi jarak ke tujuan. Dalam permainan *Rush Hour*, g(n) adalah jumlah langkah dari konfigurasi awal ke konfigurasi saat ini, sedangkan h(n) dapat berupa jumlah kendaraan penghalang atau jarak ke pintu keluar. Dalam program ini, *A** menggunakan *heuristic* bernama DUMBASS, yaitu sebuah fungsi yang menghitung jumlah kendaraan vertikal yang secara langsung menghalangi jalur primary piece ke pintu keluar pada baris yang sama. Semakin banyak kendaraan penghalang, semakin besar nilai *heuristic*-nya. Dengan pendekatan ini, *A** cenderung lebih efisien daripada UCS karena dapat "mengarah" ke solusi menggunakan informasi heuristik, namun tetap menjamin solusi optimal jika heuristic yang digunakan adalah *admissible*, yaitu tidak pernah melebih-lebihkan biaya ke tujuan $h(n) \leq h^*(n)$. Heuristic disebut *admissible* jika selalu optimis, dan dalam *Rush Hour*, heuristic seperti jumlah kendaraan penghalang umumnya *admissible* karena tidak lebih besar dari jumlah langkah minimum sebenarnya yang

dibutuhkan. Oleh karena itu, secara teoritis, *A** merupakan algoritma yang efisien dan optimal dalam menyelesaikan *Rush Hour* selama heuristic yang digunakan memenuhi syarat *admissible*.

4.5 *Heuristic*

Dalam implementasi program ini, digunakan dua jenis heuristic, yaitu fungsi berbasis jarak *primary piece (*LAZY*)* ke target dan fungsi berbasis blok pembatas (DUMBASS), yang dirancang khusus untuk permainan *Rush Hour*. fungsi berbasis jarak *primary piece* ke target menghitung jarak dalam satuan kolom dari ujung *primary piece* ke sisi kanan papan, tempat pintu keluar berada. Nilai ini memberikan estimasi kasar mengenai seberapa jauh *primary piece* perlu bergerak untuk keluar dari papan. Namun, *heuristic* ini tidak memperhitungkan adanya kendaraan lain yang mungkin menghalangi jalur tersebut, sehingga bisa menghasilkan estimasi yang terlalu optimis. Oleh karena itu, fungsi ini cocok digunakan pada algoritma yang tidak menuntut jaminan optimalitas, seperti *Greedy Best First Search.*

Sementara itu, *heuristic* yang berbasis blok pembatas berfungsi dengan menghitung jumlah kendaraan vertikal yang menghalangi jalur *primary piece* menuju pintu keluar. Dalam permainan *Rush Hour,* kendaraan vertikal yang berada di baris yang sama dan di antara primary piece dan pintu keluar dapat menghambat pergerakan *primary piece*. Dengan menghitung jumlah kendaraan penghalang ini, fungsi berbasis blok pembatas memberikan estimasi minimum tentang berapa banyak kendaraan yang perlu dipindahkan agar solusi tercapai. *Heuristic* ini bersifat *admissible*, karena tidak pernah melebih-lebihkan biaya sebenarnya yang dibutuhkan untuk menyelesaikan permainan. Hal ini menjadikannya cocok untuk digunakan pada algoritma seperti *A**, yang memerlukan *heuristic* yang aman untuk menjamin solusi optimal.

Isi Program

1. main.cpp

```
#include "class/Board/Board.hpp"
#include "class/Algo/Heuristics/Heuristics.hpp"
#include "class/Algo/Solver/Solver.hpp"

#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <limits>
#include <chrono>

int main(int argc, char *argv[])
{
    std::string filename;
    std::string algorithm;
    std::string heuristic;

    if (argc >= 3)
    {
        filename = argv[1];
        algorithm = argv[2];
        if (algorithm != "UCS")
            heuristic = argv[3];
    }
    else
    {
        std::cout << "[INPUT] ENTER PATH TO PROBLEM FOLDER:
";
        std::cin >> filename;
        std::cout << "[INPUT] ALGORITHM: (UCS / GBFS / A*/
IDA*)";
        std::cin >> algorithm;
        if (algorithm != "UCS")
        {
```

```
            std::cout << "[INPUT] SELECT HEURISTIC (DUMBASS / LAZY) : ";
            std::cin >> heuristic;
        }
    }

    // read testcase
    std::ifstream file(filename + "/problem.txt");
    if (!file)
    {
        std::cerr << "Cannot open file.\n";
        return 1;
    }

    int N, M, P;
    std::vector<std::string> INPUT;

    file >> N >> M;
    file >> P;
    file.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // newline

    std::string line;
    while (std::getline(file, line))
        INPUT.push_back(line);
    Board board(INPUT, N, M);
    Heuristics H((algorithm == "UCS") ? "UCS" : heuristic);
    Solver solver(board, H);

    std::vector<Board> solutions;
    auto start = std::chrono::high_resolution_clock::now();
    if (algorithm == "UCS" || algorithm == "A*")
        solutions = solver.SolveComplete();
    else if (algorithm == "GBFS")
        solutions = solver.SolveGreedy();
    else
        solutions = solver.SolveLowMemory();
    auto end = std::chrono::high_resolution_clock::now();
```

```
    auto duration =
std::chrono::duration_cast<std::chrono::milliseconds>(end -
start);

    std::ofstream outfile(filename + "/solutions.txt");
    if (!outfile)
    {
        std::cerr << "Cannot create solutions.txt in
folder.\n";
        return 1;
    }

    outfile << duration.count() << "\n";
    outfile << solver.VISITED_NODES << "\n";
    outfile << solutions.size() << "\n";
    outfile << solutions[0].N << " " << solutions[0].M <<
"\n"; // possibly rotated
    for (Board b : solutions)
        b.PrintBoard(outfile);
}
```

2. Board.cpp

```
#include "Board.hpp"

#include <functional>
#include <iostream>

Board::Board(std::vector<std::string> INPUT, int N, int M)
{
    this->N = N;
    this->M = M;
    std::vector<std::vector<char>> DATA(N,
std::vector<char>(M));

    // Generate DATA[][]
    // Recursive rotate function: rotates DATA 'times' times
90 deg clockwise
```

```
    std::function<void(int)> rotateDATA = [&](int times)
    {
        if (times <= 0)
            return;

        int n = DATA.size();
        int m = DATA[0].size();

        std::vector<std::vector<char>> rotated(m,
std::vector<char>(n));
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < m; j++)
            {
                rotated[j][n - i - 1] = DATA[i][j];
            }
        }
        DATA = rotated;
        std::swap(this->N, this->M);

        rotateDATA(times - 1);
    };

    // Case 1 : exit is at the top / bottom
    if (INPUT.size() > static_cast<std::size_t>(N))
    {
        // Case 1.1 : exit is at the top
        if (INPUT[0].find('K') != std::string::npos)
        {
            for (int i = 1; i <= N; i++)
            {
                for (int j = 0; j < M; j++)
                {
                    DATA[i - 1][j] = INPUT[i][j];
                }
            }
            // rotate 90 degree clockwise
            rotateDATA(1);
        }
```

```
        // Case 2.2 : exit is at the bottom
        else
        {
            for (int i = 0; i < N; i++)
            {
                for (int j = 0; j < M; j++)
                {
                    DATA[i][j] = INPUT[i][j];
                }
            }
            // rotate 270 degree clockwise
            rotateDATA(3);
        }
    }
    // Case 2 : exit is at the left / right
    else
    {
        // Case 2.1 : exit is at the left
        if (INPUT[0].size() > static_cast<std::size_t>(M))
        {
            for (int i = 0; i < N; i++)
            {
                for (int j = 1; j <= M; j++)
                {
                    DATA[i][j - 1] = INPUT[i][j];
                }
            }
            rotateDATA(2);
        }
        // Case 2.2 (default) : exit is at the right
        else
        {
            for (int i = 0; i < N; i++)
            {
                for (int j = 0; j < M; j++)
                {
                    DATA[i][j] = INPUT[i][j];
                }
            }
```

```
        }
    }

    // Turn DATA[][] into map<char, Piece>
    std::vector<std::vector<bool>> checked(N,
std::vector<bool>(M));
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < M; j++)
        {
            if (DATA[i][j] == '.' || checked[i][j])
                continue;
            checked[i][j] = true;

            char id = DATA[i][j];
            int length = 1;
            std::pair<int, int> pos = {i, j};
            bool isVertical = (i + 1 < N && DATA[i + 1][j] ==
id);

            if (isVertical)
            {
                while (i + length < N && DATA[i + length][j]
== id)
                    checked[i + length++][j] = true;
            }
            else
            {
                while (j + length < M && DATA[i][j + length]
== id)
                    checked[i][j + length++] = true;
            }

            this->Pieces.insert({id, Piece(id, length,
isVertical, pos)});
        }
    }
}
```

```
std::vector<std::shared_ptr<Board>>
Board::GenerateSuccessors() const
{
    // Reconstruct 2D board
    std::vector<std::vector<char>> board(this->N,
std::vector<char>(this->M, '.'));
    for (const auto &[id, piece] : this->Pieces)
    {
        for (int i = 0; i < piece.length; i++)
        {
            if (piece.isVertical)
                board[piece.pos.first + i][piece.pos.second]
= id;
            else
                board[piece.pos.first][piece.pos.second + i]
= id;
        }
    }

    std::vector<std::shared_ptr<Board>> successors;

    for (const auto &[id, piece] : this->Pieces)
    {
        if (piece.isVertical)
        {
            for (int i = piece.pos.first - 1; i >= 0; i--)
            {
                if (board[i][piece.pos.second] != '.')
                    break;

                auto newBoard =
std::make_shared<Board>(*this);
                newBoard->Pieces.at(id).pos.first = i;
                successors.push_back(newBoard);
            }
            for (int i = piece.pos.first + piece.length; i <
N; i++)
            {
                if (board[i][piece.pos.second] != '.')
```

```
                    break;

                auto newBoard =
std::make_shared<Board>(*this);
                newBoard->Pieces.at(id).pos.first = i -
piece.length + 1;
                successors.push_back(newBoard);
            }
        }
        else
        {
            for (int j = piece.pos.second - 1; j >= 0; j--)
            {
                if (board[piece.pos.first][j] != '.')
                    break;

                auto newBoard =
std::make_shared<Board>(*this);
                newBoard->Pieces.at(id).pos.second = j;
                successors.push_back(newBoard);
            }
            for (int j = piece.pos.second + piece.length; j <
M; j++)
            {
                if (board[piece.pos.first][j] != '.')
                    break;

                auto newBoard =
std::make_shared<Board>(*this);
                newBoard->Pieces.at(id).pos.second = j -
piece.length + 1;
                successors.push_back(newBoard);
            }
        }
    }

    return successors;
}
```

```
std::string Board::Serialize() const
{
    std::string key;
    for (const auto &[id, piece] : this->Pieces)
    {
        const Piece &p = piece;
        key += p.id;
        key += std::to_string(p.pos.first) + "," +
std::to_string(p.pos.second) + ",";
        key += (p.isVertical ? "V" : "H") +
std::to_string(p.length) + ";";
    }
    return key;
}

bool Board::IsSolved()
{
    Piece primary = this->Pieces.at('P');
    return primary.pos.second + primary.length == this->M;
}

void Board::PrintBoard(std::ostream &out) const
{
    std::vector<std::vector<char>> board(this->N,
std::vector<char>(this->M, '.'));

    for (const auto &[id, piece] : this->Pieces)
    {
        for (int i = 0; i < piece.length; i++)
        {
            if (piece.isVertical)
                board[piece.pos.first + i][piece.pos.second]
= id;
            else
                board[piece.pos.first][piece.pos.second + i]
= id;
        }
    }
```

```
    for (int i = 0; i < this->N; i++)
    {
        for (int j = 0; j < this->M; j++)
            out << board[i][j];
        out << '\n';
    }
}
```

3. Piece.hpp

```
#ifndef __PIECE__
#define __PIECE__

#include <utility>

class Piece
{

public:
    char id;                    // identifier
    int length;                 // length of this piece
    bool isVertical;            // vertical / horizontal
    std::pair<int, int> pos; // top-left index

    Piece(char id, int length, bool isVertical,
std::pair<int, int> pos) : id(id), length(length),
isVertical(isVertical), pos(pos) {};
};

#endif
```

4. Heuristics.cpp

```
#include "Heuristics.hpp"
```

```
int Heuristics::DUMBASS(Board &board)
{
    if (board.IsSolved())
        return 0;

    const auto &pieces = board.Pieces;
    const Piece &primary = pieces.at('P');

    int row = primary.pos.first;
    int colStart = primary.pos.second;
    int colEnd = colStart + primary.length - 1;

    int blockingCells = 0;
    // Check for vertical pieces that block the path to the
exit
    for (const auto &[id, piece] : pieces)
    {
        if (id == 'P')
            continue;
        if (piece.isVertical)
        {
            int col = piece.pos.second;
            int rowStart = piece.pos.first;
            int rowEnd = rowStart + piece.length - 1;

            if (col > colEnd && col <= board.M - 1 &&
rowStart <= row && rowEnd >= row)
                blockingCells++;
        }
    }

    return blockingCells;
}

int Heuristics::LAZY(Board &board)
{
    Piece primary = board.Pieces.at('P');
    return board.M - (primary.pos.second + primary.length) -
1;
```

```
}

int Heuristics::calculate(Board &board)
{
    if (this->type == "DUMBASS")
        return DUMBASS(board);
    if (this->type == "LAZY")
        return LAZY(board);
    if (this->type == "UCS")
        return 0;

    return DUMBASS(board);
}
```

5. Solver.cpp

```
#include "Solver.hpp"

#include <queue>
#include <vector>
#include <memory>
#include <unordered_set>
#include <algorithm>
#include <iostream>
#include <functional>

std::vector<Board> Solver::SolveComplete()
{
    auto cmp = [](const std::shared_ptr<Node> &a, const
std::shared_ptr<Node> &b)
    {
        return a->f > b->f; // Min-heap
    };

    std::priority_queue<
        std::shared_ptr<Node>,
        std::vector<std::shared_ptr<Node>>,
```

```
        decltype(cmp)>
        openSet(cmp);

    std::unordered_set<std::string> visited;

    auto startNode = std::make_shared<Node>(INITIAL, 0,
H.calculate(INITIAL));
    openSet.push(startNode);
    while (!openSet.empty())
    {
        auto current = openSet.top();
        openSet.pop();

        std::string stateKey = current->board.Serialize();
        if (visited.count(stateKey))
            continue;
        visited.insert(stateKey);
        this->VISITED_NODES++;

        if (current->board.IsSolved())
        {
            std::vector<Board> path;
            for (auto node = current; node != nullptr; node =
node->parent)
            {
                path.push_back(node->board);
            }
            std::reverse(path.begin(), path.end());
            return path;
        }

        auto successors =
current->board.GenerateSuccessors();
        for (auto &succ : successors)
        {
            std::string succKey = succ->Serialize();
            if (visited.count(succKey))
                continue;
```

```
            int g = current->g + 1;
            int h = H.calculate(*succ);
            auto nextNode = std::make_shared<Node>(*succ, g,
h, current);
            openSet.push(nextNode);
        }
    }

    return {}; // No solution found
}

std::vector<Board> Solver::SolveGreedy()
{
    auto cmp = [](const std::shared_ptr<Node> &a, const
std::shared_ptr<Node> &b)
    {
        return a->h > b->h; // Min-heap based only on
heuristic
    };

    std::priority_queue<
        std::shared_ptr<Node>,
        std::vector<std::shared_ptr<Node>>,
        decltype(cmp)>
        openSet(cmp);

    std::unordered_set<std::string> visited;

    int h = H.calculate(INITIAL);
    auto startNode = std::make_shared<Node>(INITIAL, 0, h);
// g=0, h=heuristic
    openSet.push(startNode);

    while (!openSet.empty())
    {
        auto current = openSet.top();
        openSet.pop();

        std::string stateKey = current->board.Serialize();
```

```
        if (visited.count(stateKey))
            continue;
        visited.insert(stateKey);
        this->VISITED_NODES++;

        if (current->board.IsSolved())
        {
            std::vector<Board> path;
            for (auto node = current; node != nullptr; node =
node->parent)
            {
                path.push_back(node->board);
            }
            std::reverse(path.begin(), path.end());
            return path;
        }

        auto successors =
current->board.GenerateSuccessors();
        std::sort(successors.begin(), successors.end(),
[](const auto &a, const auto &b)
                  {
                      return a->Serialize() < b->Serialize();
// Ensure consistent expansion
                  });

        for (auto &succ : successors)
        {
            std::string succKey = succ->Serialize();
            if (visited.count(succKey))
                continue;

            int h = H.calculate(*succ);
            auto nextNode = std::make_shared<Node>(*succ, 0,
h, current); // g is 0
            openSet.push(nextNode);
        }
    }
```

```
    return {}; // No solution found
}

std::vector<Board> Solver::SolveLowMemory()
{
    struct SearchResult
    {
        std::shared_ptr<Node> result;
        int nextThreshold;
    };

    std::unordered_map<std::string, int> globalF; // Global
visited map: stateKey -> min f-cost

    std::function<SearchResult(std::shared_ptr<Node>, int,
std::unordered_set<std::string> &)> dfs;
    dfs = [&](std::shared_ptr<Node> current, int threshold,
std::unordered_set<std::string> &pathVisited) -> SearchResult
    {
        std::string stateKey = current->board.Serialize();
        int f = current->g + current->h;

        if (f > threshold)
            return {nullptr, f};

        // Global pruning: skip if already visited with equal
or better f
        auto it = globalF.find(stateKey);
        if (it != globalF.end() && it->second <= f)
            return {nullptr, INT_MAX};
        globalF[stateKey] = f;

        // Cycle avoidance on current path
        if (pathVisited.count(stateKey))
            return {nullptr, INT_MAX};

        if (current->board.IsSolved())
            return {current, threshold};
```

```
        pathVisited.insert(stateKey);
        this->VISITED_NODES++;

        int minThreshold = INT_MAX;
        auto successors =
current->board.GenerateSuccessors();

        // Cache heuristic for each successor before sorting
        std::vector<std::pair<std::shared_ptr<Board>, int>>
successorsWithH;
        for (auto &succ : successors)
        {
            int h = H.calculate(*succ);
            successorsWithH.emplace_back(succ, h);
        }

        // Sort by f = g + h ascending
        std::sort(successorsWithH.begin(),
successorsWithH.end(),
                  [&](const auto &a, const auto &b)
                  {
                      int fa = current->g + 1 + a.second;
                      int fb = current->g + 1 + b.second;
                      return fa < fb;
                  });

        for (auto &[succ, h] : successorsWithH)
        {
            std::string succKey = succ->Serialize();
            if (pathVisited.count(succKey))
                continue;

            int g = current->g + 1;
            auto nextNode = std::make_shared<Node>(*succ, g,
h, current);

            auto result = dfs(nextNode, threshold,
pathVisited);
            if (result.result != nullptr)
```

```
            {
                pathVisited.erase(stateKey); // backtrack
                return result;
            }

            minThreshold = std::min(minThreshold,
result.nextThreshold);
        }

        pathVisited.erase(stateKey); // backtrack
        return {nullptr, minThreshold};
    };

    int h0 = H.calculate(INITIAL);
    auto root = std::make_shared<Node>(INITIAL, 0, h0);

    int threshold = h0;

    while (true)
    {
        globalF.clear(); // Clear global visited at each
iteration
        std::unordered_set<std::string> pathVisited;

        auto result = dfs(root, threshold, pathVisited);

        if (result.result != nullptr)
        {
            // Reconstruct path from result node
            std::vector<Board> path;
            for (auto node = result.result; node != nullptr;
node = node->parent)
                path.push_back(node->board);
            std::reverse(path.begin(), path.end());
            return path;
        }

        if (result.nextThreshold == INT_MAX)
        {
```

```
            break;
        }

        threshold = result.nextThreshold;
    }

    return {}; // No solution found
}
```

6. GUI.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.BufferedReader;
import java.io.File;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Map;
import java.util.List;

public class GUI {

    private int height, width;
    private char[][] gridData;
    private JPanel gridPanel;
    private Character currentChar = null;

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new
GUI().createAndShowGUI());
    }

    private void createAndShowGUI() {
        JFrame frame = new JFrame("Rush Hour Puzzle
Generator");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        frame.setSize(700, 700);
        frame.setLayout(new BorderLayout());

        // Top input panel
        JPanel inputPanel = new JPanel(new FlowLayout());
        JLabel heightLabel = new JLabel("Height:");
        JTextField heightField = new JTextField(5);
        JLabel widthLabel = new JLabel("Width:");
        JTextField widthField = new JTextField(5);
        JButton renderButton = new JButton("Render");
        JButton putPieceButton = new JButton("Put Piece");
        JButton saveButton = new JButton("Save");
        JButton solveButton = new JButton("Solve");

        inputPanel.add(heightLabel);
        inputPanel.add(heightField);
        inputPanel.add(widthLabel);
        inputPanel.add(widthField);
        inputPanel.add(renderButton);
        inputPanel.add(putPieceButton);
        inputPanel.add(saveButton);
        inputPanel.add(solveButton);

        gridPanel = new JPanel();
        JScrollPane scrollPane = new JScrollPane(gridPanel);
        frame.add(inputPanel, BorderLayout.NORTH);
        frame.add(scrollPane, BorderLayout.CENTER);

        renderButton.addActionListener(_ -> {
            try {
                height =
Integer.parseInt(heightField.getText());
                width =
Integer.parseInt(widthField.getText());
                if (height <= 0 || width <= 0)
                    throw new NumberFormatException();

                int fullHeight = height + 2;
                int fullWidth = width + 2;
```

```
                gridData = new char[fullHeight][fullWidth];
                gridPanel.removeAll();
                gridPanel.setLayout(new
GridLayout(fullHeight, fullWidth, 2, 2));

                for (int row = 0; row < fullHeight; row++) {
                    for (int col = 0; col < fullWidth; col++)
{
                        JPanel cell = new JPanel();
                        boolean isOuter = (row == height + 1
|| col == width + 1 || row == 0 || col == 0);
                        cell.setBackground(isOuter ?
Color.DARK_GRAY : Color.LIGHT_GRAY);

cell.setBorder(BorderFactory.createLineBorder(Color.BLACK));

                        int finalRow = row;
                        int finalCol = col;

                        cell.addMouseListener(new
MouseAdapter() {
                            @Override
                            public void
mouseClicked(MouseEvent e) {
                                if (currentChar != null) {

gridData[finalRow][finalCol] = currentChar;

cell.setBackground(getColorForChar(currentChar));
                                    cell.removeAll();
                                    cell.add(new
JLabel(currentChar.toString()));
                                    cell.revalidate();
                                    cell.repaint();
                                }
                            }
                        });
```

```
                        gridPanel.add(cell);
                    }
                }
                putPieceButton.setVisible(true);
                gridPanel.revalidate();
                gridPanel.repaint();

            } catch (NumberFormatException ex) {
                JOptionPane.showMessageDialog(frame, "Please
enter valid positive integers for height and width.");
            }
        });

        putPieceButton.addActionListener(_ -> {
            if (gridData == null) {
                JOptionPane.showMessageDialog(null, "No grid.
Please render first.");
                return;
            }
            String input = JOptionPane.showInputDialog(frame,
"Enter character A-Z for the piece:");
            if (input != null && input.length() == 1) {
                char ch =
Character.toUpperCase(input.charAt(0));
                if (ch >= 'A' && ch <= 'Z') {
                    currentChar = ch;
                } else {
                    JOptionPane.showMessageDialog(frame,
"Please enter a letter A-Z.");
                }
            }
        });

        saveButton.addActionListener(_ -> {
            if (gridData == null) {
                JOptionPane.showMessageDialog(null, "No grid
to save. Please render and add pieces first.");
                return;
            }
```

```
            // Predetermined directory
            java.io.File dir = new java.io.File("test");
            if (!dir.exists()) {
                dir.mkdirs(); // Create directory if it
doesn't exist
            }

            while (true) {
                String folderName =
JOptionPane.showInputDialog(null, "Enter testcase name
(folder):");
                if (folderName == null ||
folderName.trim().isEmpty()) {
                    return;
                }

                File testcaseDir = new File(dir, folderName);
                File problemFile = new File(testcaseDir,
"problem.txt");

                if (testcaseDir.exists()) {
                    int choice =
JOptionPane.showConfirmDialog(null,
                            "Testcase name already taken.
Overwrite?",
                            "Testcase name is used!",
                            JOptionPane.YES_NO_OPTION);

                    if (choice != JOptionPane.YES_OPTION) {
                        continue; // Reprompt
                    }
                }

                // Create or overwrite directory
                testcaseDir.mkdirs();

                try (java.io.PrintWriter writer = new
java.io.PrintWriter(problemFile)) {
```

```
                    // 1. Write height and width
                    writer.println(height + " " + width);

                    String exitPosition = null;
                    // 2. Count distinct characters
                    java.util.Set<Character> pieces = new
java.util.HashSet<>();
                    for (int i = 0; i <= height + 1; i++) {
                        for (int j = 0; j <= width + 1; j++)
{
                            char c = gridData[i][j];
                            if (c == 'K') {
                                if (i == 0)
                                    exitPosition = "UP";
                                if (j == 0)
                                    exitPosition = "LEFT";
                                if (j == width + 1)
                                    exitPosition = "RIGHT";
                                if (i == height + 1)
                                    exitPosition = "DOWN";
                            } else if (c != '\0' && c != 'P')
{
                                pieces.add(c);
                            }
                        }
                    }
                    writer.println(pieces.size());
                    // 3. Write grid rows
                    if (exitPosition == null)
                        throw new Exception();
                    if (exitPosition == "UP") {
                        // print top-most layer
                        for (int j = 1; j <= width; j++) {
                            char c = gridData[0][j];
                            writer.print((c == '\0' ? ' ' :
c));

                        }
                        writer.println();
                        for (int i = 1; i <= height; i++) {
```

```
                                for (int j = 1; j <= width; j++)
{
                                    char c = gridData[i][j];
                                    writer.print((c == '\0' ? '.'
: c));
                                }
                                if (i < height)
                                    writer.println();
                            }
                        } else if (exitPosition == "DOWN") {
                            for (int i = 1; i <= height; i++) {
                                for (int j = 1; j <= width; j++)
{
                                    char c = gridData[i][j];
                                    writer.print((c == '\0' ? '.'
: c));
                                }
                                writer.println();
                            }
                            // print bottom-most layer
                            for (int j = 1; j <= width; j++) {
                                char c = gridData[height + 1][j];
                                writer.print((c == '\0' ? ' ' :
c));
                            }
                        } else if (exitPosition == "RIGHT") {
                            // print all the way to the right
                            for (int i = 1; i <= height; i++) {
                                for (int j = 1; j <= width; j++)
{
                                    char c = gridData[i][j];
                                    writer.print((c == '\0' ? '.'
: c));
                                }
                                if (gridData[i][width + 1] ==
'K')
                                    writer.print('K');
                                if (i < height)
                                    writer.println();
```

```
                        }
                    } else {
                        // print all the way to the left
                        for (int i = 1; i <= height; i++) {
                            for (int j = 0; j <= width; j++)
{
                                if (j > 0) {
                                    char c = gridData[i][j];
                                    writer.print((c == '\0' ?
'.' : c));
                                } else

writer.print((gridData[i][j] == 'K') ? "K" : ' ');
                                }
                                if (i < height)
                                    writer.println();
                            }
                        }

                        JOptionPane.showMessageDialog(null,
"Puzzle saved to " + problemFile.getAbsolutePath());
                        putPieceButton.setVisible(false);
                        solveButton.setVisible(true);
                    } catch (Exception ex) {
                        JOptionPane.showMessageDialog(null,
"Error saving file: " + ex.getMessage());
                    }

                    break; // Exit loop once saved
                }
            });

            solveButton.addActionListener(_ -> {
                // 1. Prompt for directory
                JFileChooser chooser = new JFileChooser();
                chooser.setDialogTitle("Select Problem Folder");

chooser.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
                chooser.setAcceptAllFileFilterUsed(false);
```

```
            int result = chooser.showOpenDialog(null);
            if (result != JFileChooser.APPROVE_OPTION) {
                return; // Cancelled
            }

            File selectedFolder = chooser.getSelectedFile();

            // 2. Prompt for solving method
            String[] methods = { "UCS", "A*", "GBFS", "IDA*"
};
            String method = (String)
JOptionPane.showInputDialog(
                    null,
                    "Choose solving method:",
                    "Solve Puzzle",
                    JOptionPane.PLAIN_MESSAGE,
                    null,
                    methods,
                    methods[0]);

            if (method == null)
                return; // Cancelled

            String heuristic = null;

            // 3. Prompt for heuristic if not UCS
            if (!method.equals("UCS")) {
                String[] heuristics = { "DUMBASS", "LAZY" };
                heuristic = (String)
JOptionPane.showInputDialog(
                        null,
                        "Choose heuristic:",
                        "Select Heuristic",
                        JOptionPane.PLAIN_MESSAGE,
                        null,
                        heuristics,
                        heuristics[0]);
```

```
                if (heuristic == null)
                    return; // Cancelled
            }

            // 4. Construct command
            File problemFile = selectedFolder;

            List<String> command = new ArrayList<>();
            command.add("./bin/main.exe");
            command.add(problemFile.getAbsolutePath());
            command.add(method);
            if (heuristic != null) {
                command.add(heuristic);
            }

            try {
                ProcessBuilder pb = new
ProcessBuilder(command);
                pb.redirectErrorStream(true);
                Process process = pb.start();

                BufferedReader reader = new
BufferedReader(new
InputStreamReader(process.getInputStream()));
                StringBuilder output = new StringBuilder();
                String line;
                while ((line = reader.readLine()) != null) {
                    output.append(line).append("\n");
                }

                process.waitFor();

                // 6. Read solutions.txt
                File solutionsFile = new File(problemFile,
"solutions.txt");
                if (!solutionsFile.exists()) {
                    JOptionPane.showMessageDialog(null,
"solutions.txt not found.");
                    return;
```

```
                }

                List<char[][]> steps = new ArrayList<>();
                int runtime, visited, numSteps, h, w;

                try (BufferedReader solReader = new
BufferedReader(new java.io.FileReader(solutionsFile))) {
                    runtime =
Integer.parseInt(solReader.readLine().trim());
                    visited =
Integer.parseInt(solReader.readLine().trim());
                    numSteps =
Integer.parseInt(solReader.readLine().trim());

                    String[] dims =
solReader.readLine().trim().split("\\s+");
                    h = Integer.parseInt(dims[0]);
                    w = Integer.parseInt(dims[1]);

                    for (int s = 0; s < numSteps; s++) {
                        char[][] board = new char[h][w];
                        for (int i = 0; i < h; i++) {
                            String row =
solReader.readLine();
                            for (int j = 0; j < w; j++) {
                                board[i][j] = row.charAt(j);
                            }
                        }
                        steps.add(board);
                    }
                }

                // 7. Show metadata popup
                JOptionPane.showMessageDialog(null,
                        "Runtime: " + runtime + " ms\n" +
                                "Visited Nodes: " + visited +
"\n" +
                                "Number of Steps: " +
numSteps);
```

```
                // 8. Show solution viewer
                JFrame solutionFrame = new JFrame("Solution
Viewer");
                solutionFrame.setSize(600, 600);
                solutionFrame.setLayout(new BorderLayout());

                JPanel boardPanel = new JPanel();
                boardPanel.setLayout(new GridLayout(h, w, 2,
2));
                solutionFrame.add(boardPanel,
BorderLayout.CENTER);

                JButton prevButton = new JButton("Prev");
                JButton nextButton = new JButton("Next");
                JLabel stepLabel = new JLabel(); // Step
display

                JPanel navPanel = new JPanel();
                navPanel.add(prevButton);
                navPanel.add(stepLabel);
                navPanel.add(nextButton);
                solutionFrame.add(navPanel,
BorderLayout.SOUTH);

                final int[] stepIndex = { 0 };

                Runnable renderStep = () -> {
                    boardPanel.removeAll();
                    char[][] board = steps.get(stepIndex[0]);

                    for (int i = 0; i < h; i++) {
                        for (int j = 0; j < w; j++) {
                            JPanel cell = new JPanel();
                            cell.setPreferredSize(new
Dimension(40, 40));

cell.setBorder(BorderFactory.createLineBorder(Color.BLACK));
                            char c = board[i][j];
```

```
                            cell.setBackground(c == '.' ?
Color.white : getColorForChar(c));
                            if (c != '.')
                                cell.add(new
JLabel(Character.toString(c)));
                            boardPanel.add(cell);
                        }
                    }

                    stepLabel.setText("STEP " + (stepIndex[0]
+ 1) + " / " + steps.size());

                    boardPanel.revalidate();
                    boardPanel.repaint();
                };

                renderStep.run();

                prevButton.addActionListener(_ -> {
                    if (stepIndex[0] > 0) {
                        stepIndex[0]--;
                        renderStep.run();
                    }
                });

                nextButton.addActionListener(_ -> {
                    if (stepIndex[0] < steps.size() - 1) {
                        stepIndex[0]++;
                        renderStep.run();
                    }
                });

                solutionFrame.setVisible(true);

            } catch (Exception ex) {
                ex.printStackTrace();
                JOptionPane.showMessageDialog(null,
                        "Failed to run solver or read output:
" + ex.getMessage());
```

```
            }
        });

        frame.setVisible(true);
    }

    private static final Map<Character, Color> fixedColorMap
= Map.ofEntries(
            Map.entry('A', new Color(0x3CB44B)), // Green
            Map.entry('B', new Color(0x4363D8)), // Blue
            Map.entry('C', new Color(0xF58231)), // Orange
            Map.entry('D', new Color(0x911EB4)), // Purple
            Map.entry('E', new Color(0x46F0F0)), // Cyan
            Map.entry('F', new Color(0xF032E6)), // Magenta
            Map.entry('G', new Color(0xBCF60C)), // Lime
            Map.entry('H', new Color(0xFABEBE)), // Pink
            Map.entry('I', new Color(0x008080)), // Teal
            Map.entry('J', new Color(0xFFD8B1)), // Apricot
            Map.entry('K', new Color(0x000075)), // Very dark
blue
            Map.entry('L', new Color(0x808000)), // Olive
            Map.entry('M', new Color(0xAA6E28)), // Brown
            Map.entry('N', new Color(0xA9A9A9)), // Dark gray
            Map.entry('O', new Color(0xDCBEFF)), // Lavender
            Map.entry('P', new Color(0xE6194B)), // Red
            Map.entry('Q', new Color(0x9A6324)), // Earth
brown
            Map.entry('R', new Color(0xFFE119)), // Yellow
            Map.entry('S', new Color(0xBCEBCB)), // Mint
            Map.entry('T', new Color(0xFFB3BA)), // Soft pink
            Map.entry('U', new Color(0xC9C9FF)), // Light
purple
            Map.entry('V', new Color(0xB9FBC0)), // Light
green
            Map.entry('W', new Color(0x8B0000)), // Dark red
            Map.entry('X', new Color(0x40E0D0)), // Turquoise
            Map.entry('Y', new Color(0xFF69B4)), // Hot pink
            Map.entry('Z', new Color(0x5D3FD3)) // Deep
purple
```

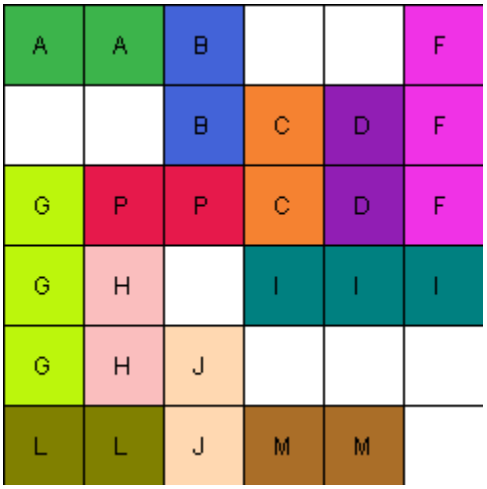
```
    );

    public static Color getColorForChar(char ch) {
        return fixedColorMap.getOrDefault(ch, Color.GRAY);
    }
}
```

Implementasi dan Pengujian

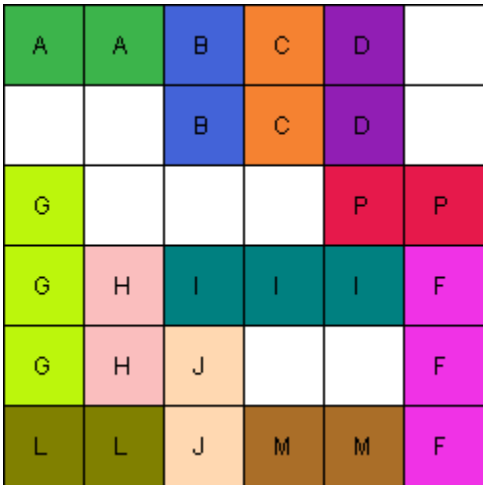
I. UCS

1. Testcase 1

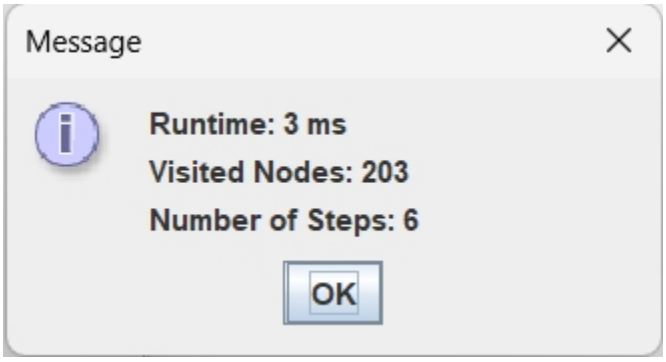
- Initial :



- Akhir :

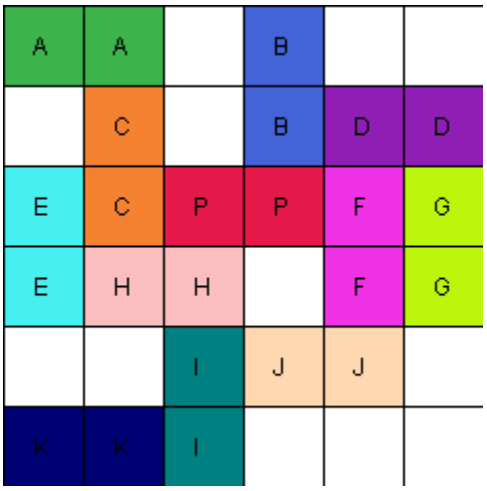


- Info :

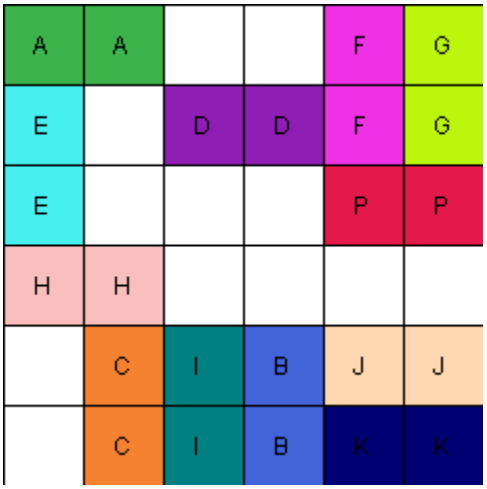


2. Testcase 2

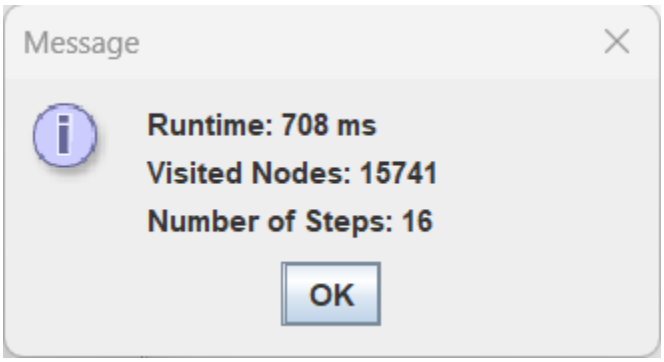
- Initial :



- Akhir :

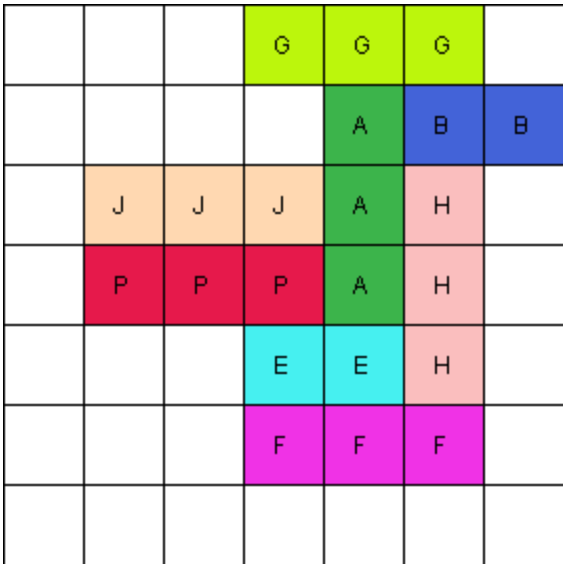


- Info

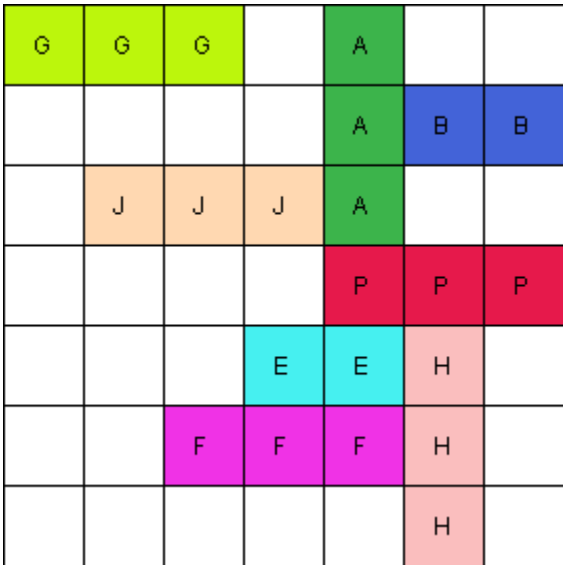


3. Testcase 3

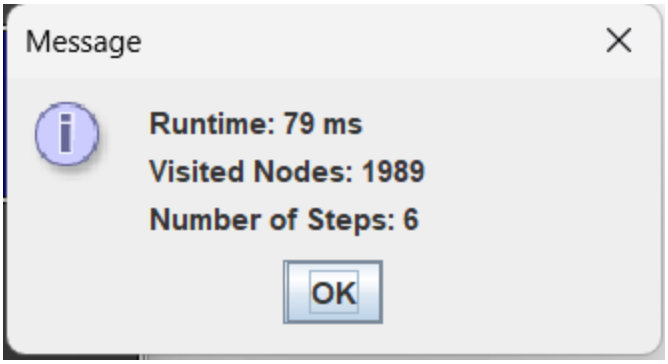
- Initial :



- Akhir :

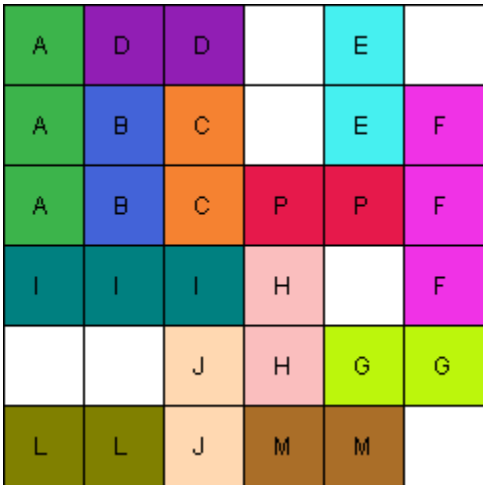


- Info

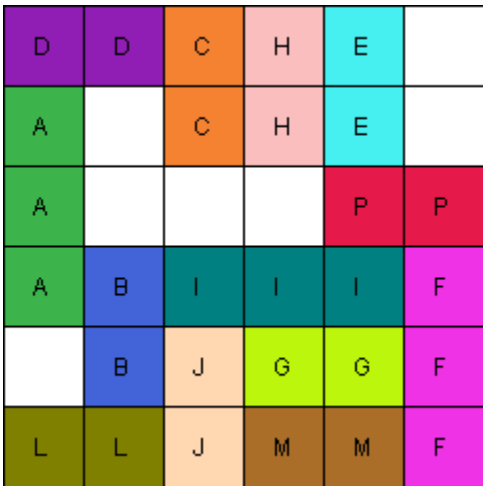


4. Testcase 4

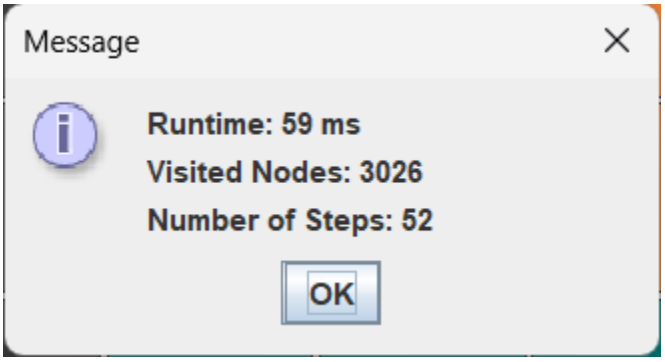
- Initial :



- Akhir :



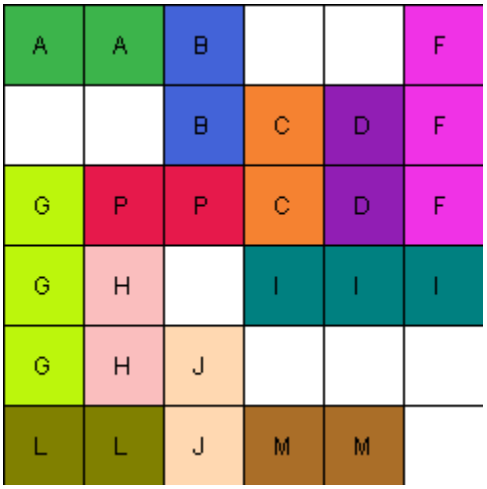
- Info :



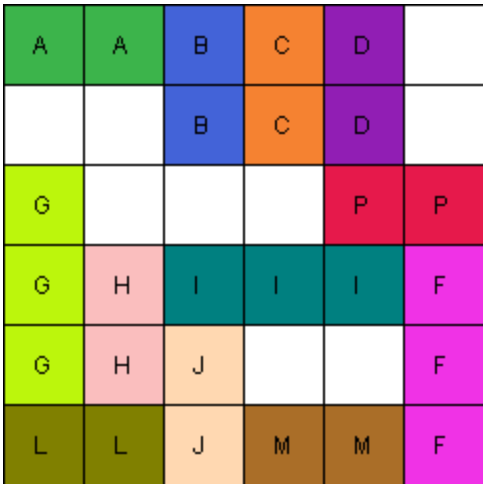
II. A*

1. Testcase 1

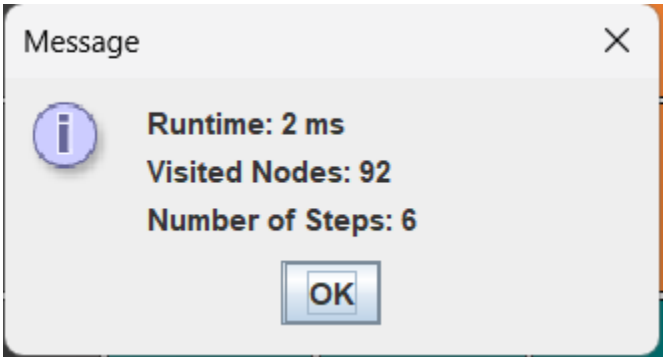
- Initial :



- Akhir :

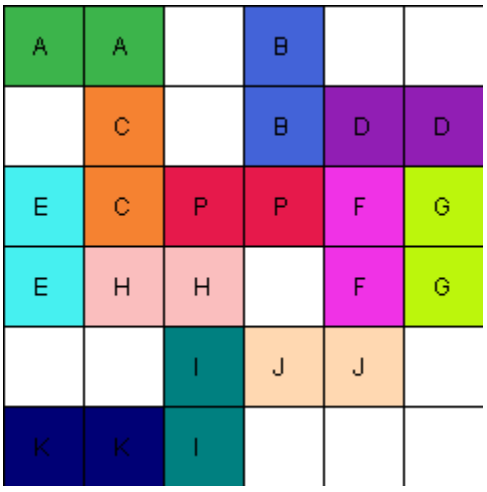


- Info :

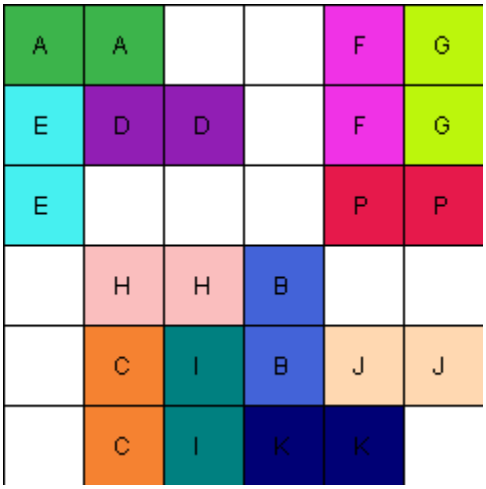


2. Testcase 2

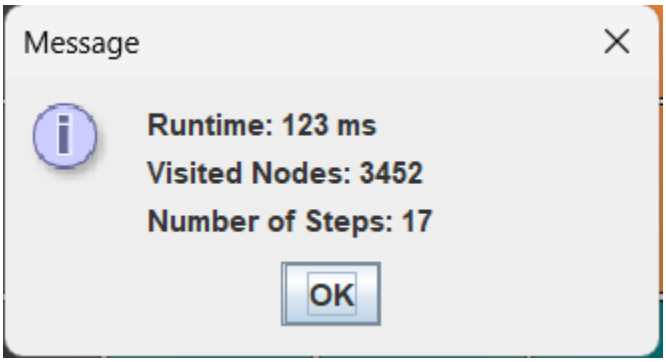
- Initial :



- Akhir :

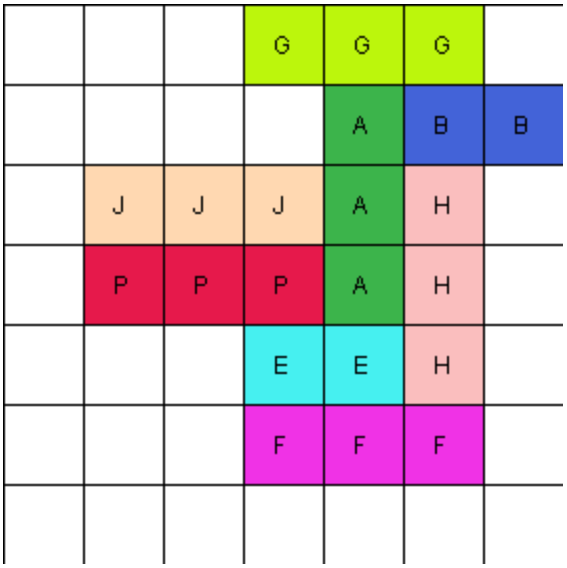


- Info :

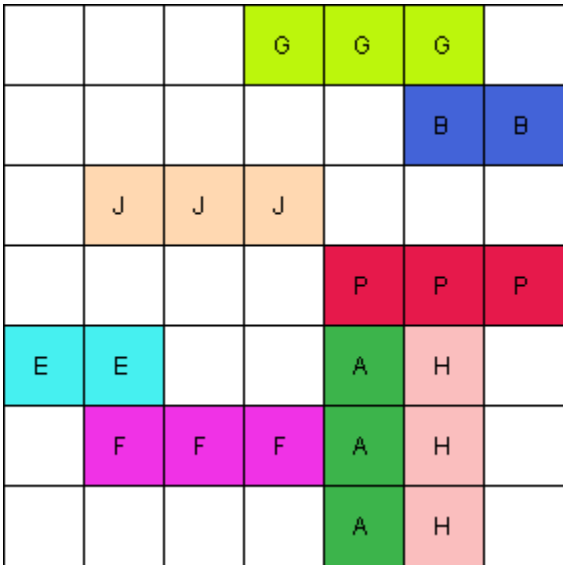


3. Testcase 3

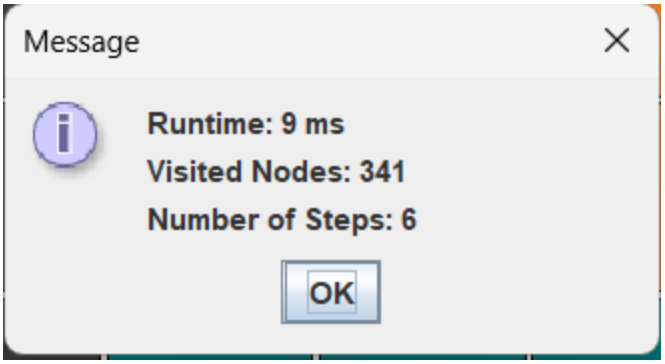
- Initial :



- Akhir :

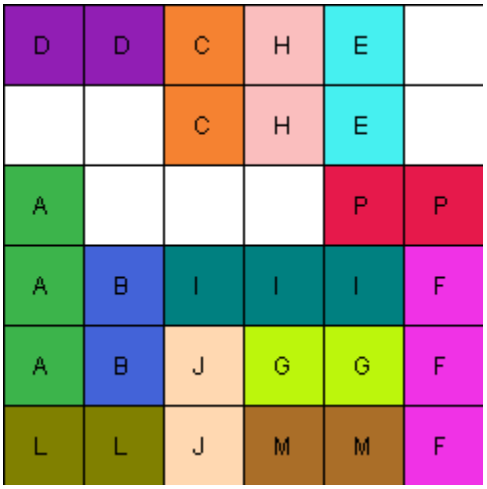


- Info

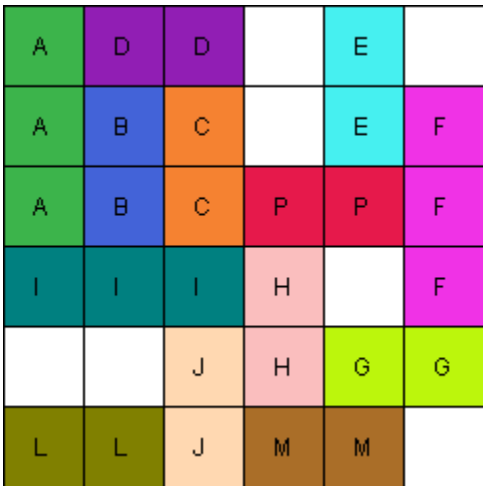


4. Testcase 4

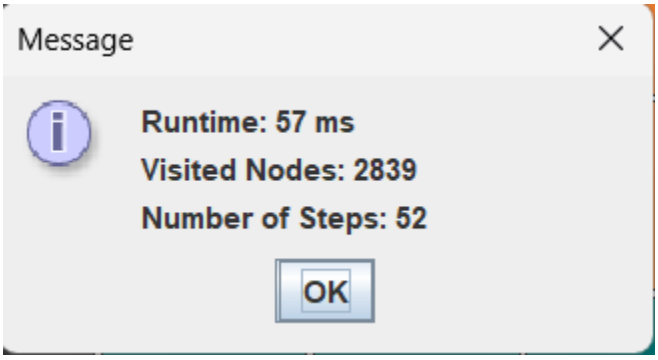
- Initial :



- Akhir :



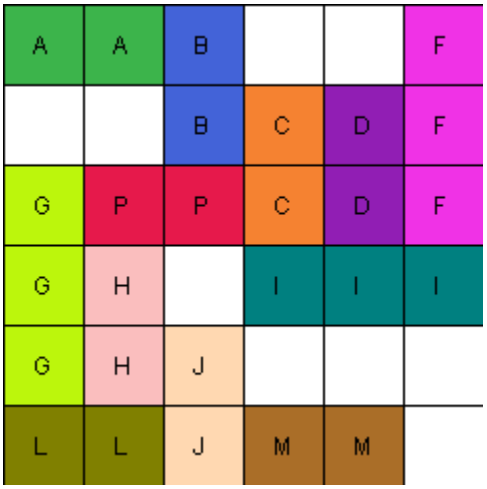
- Info :



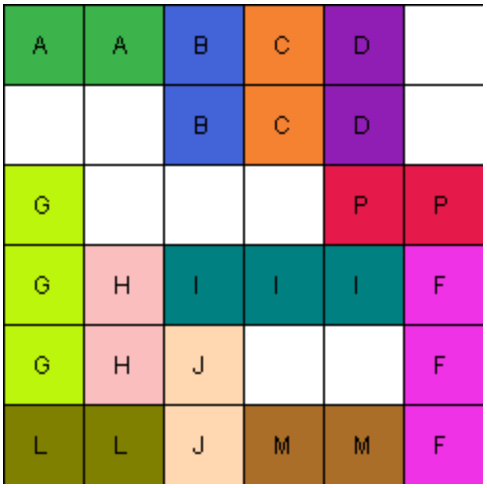
III. Greedy Best First Search

1. Testcase 1

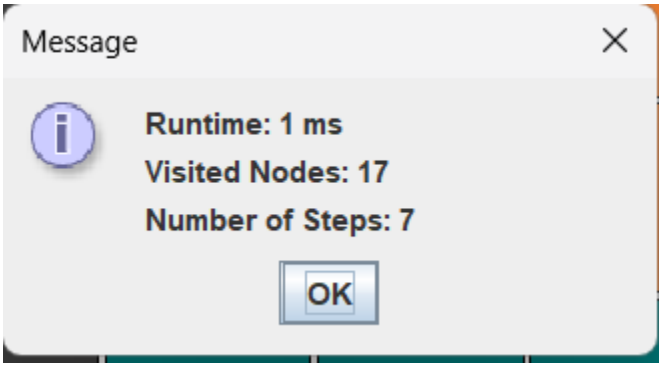
- Initial :



- Akhir :

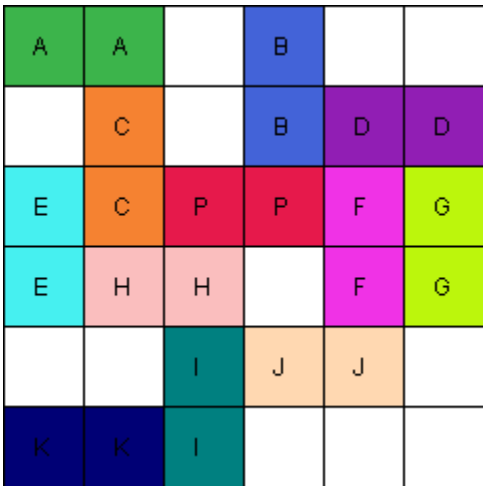


- Info :

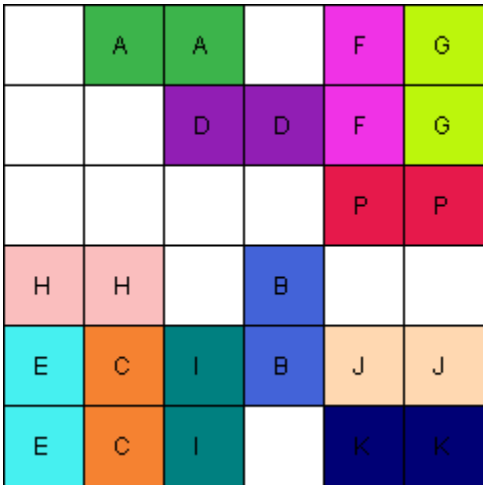


2. Testcase 2

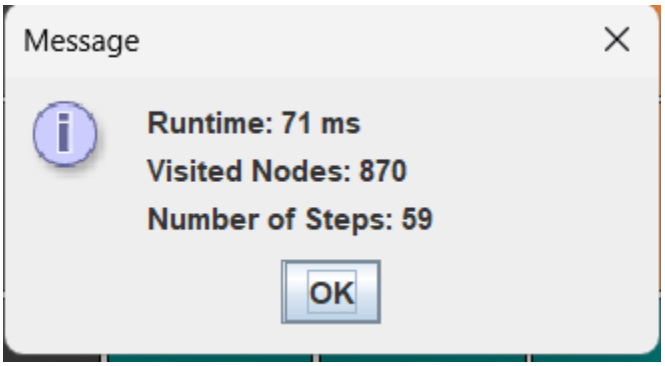
- Initial :



- Akhir :

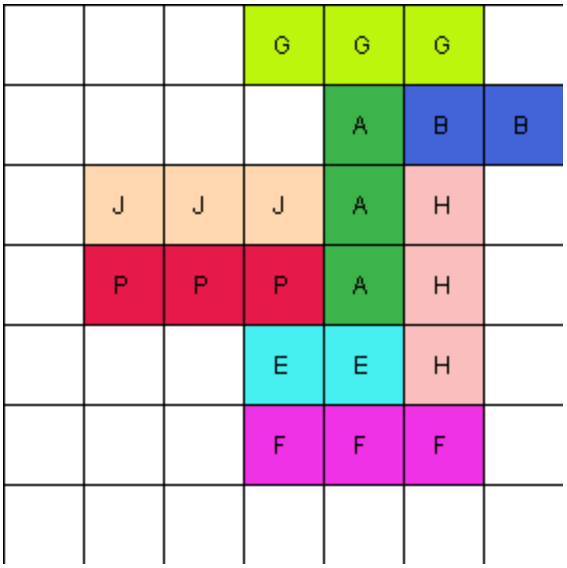


- Info :

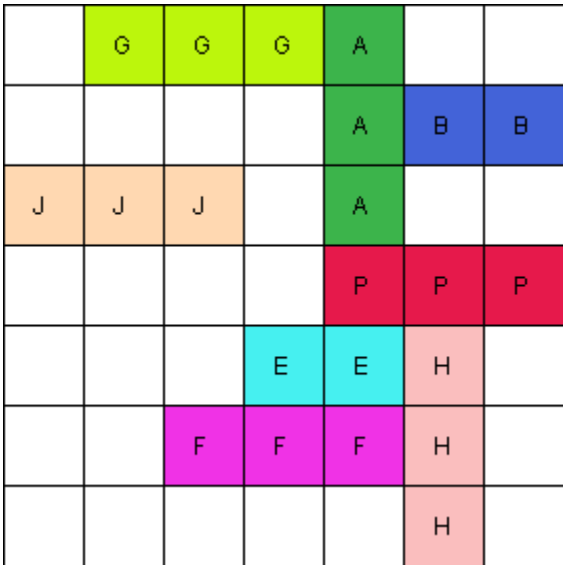


3. Testcase 3

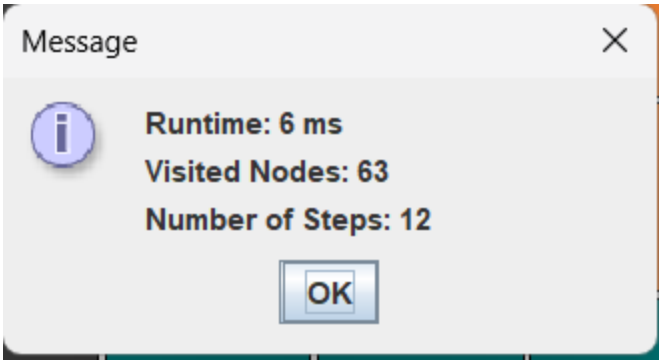
- Initial :



- Akhir :

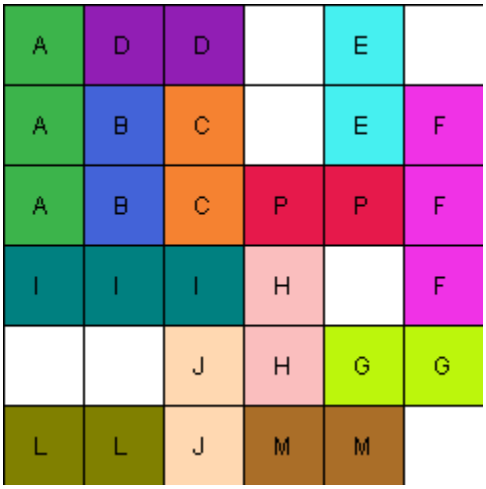


- Info :

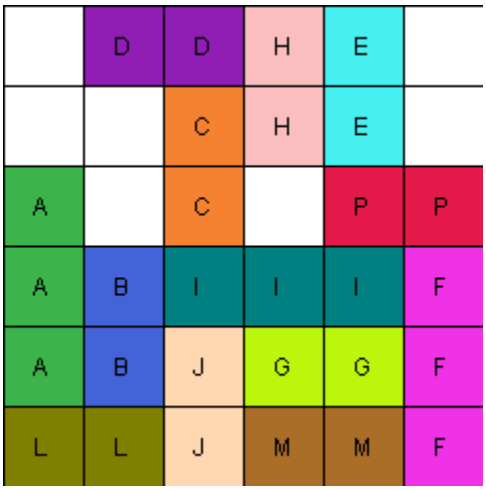


4. Testcase 4

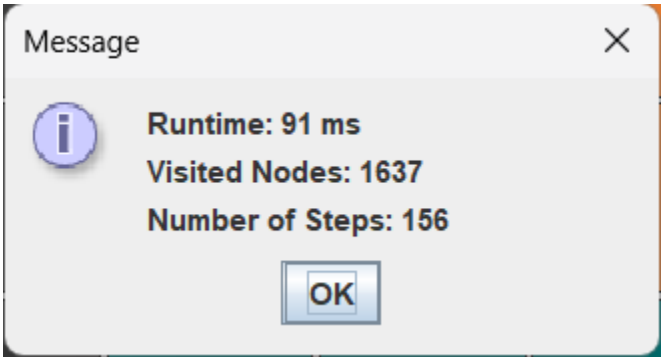
- Initial :



- Akhir :



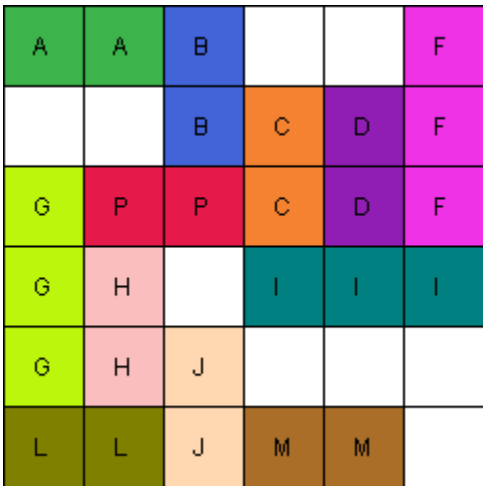
- Info :



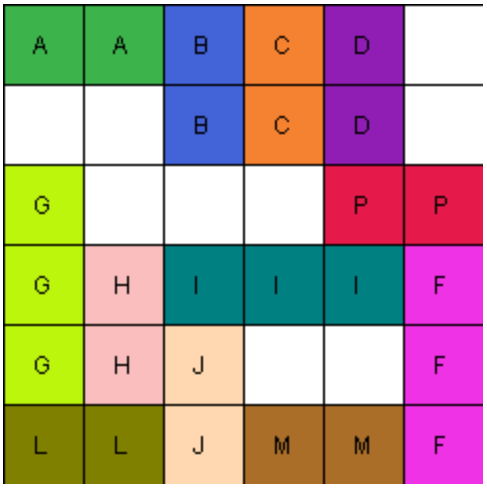
IV. IDA*

1. Testcase 1

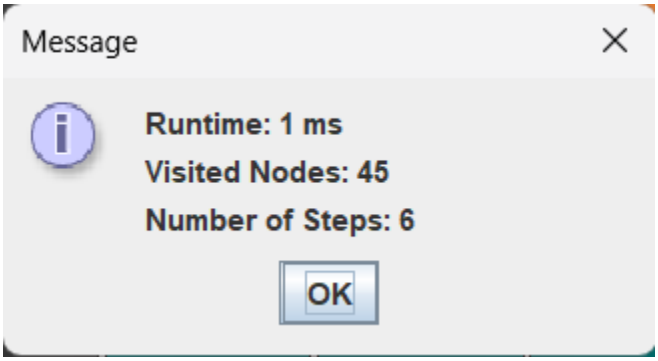
- Initial :



- Akhir :

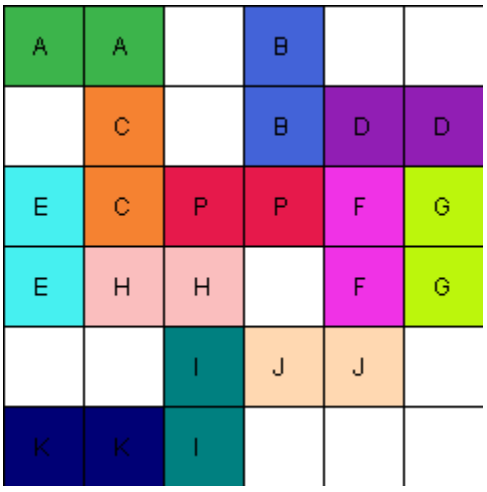


- Info :

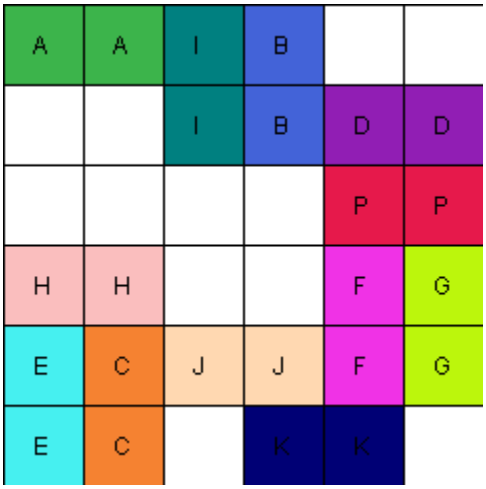


2. Testcase 2

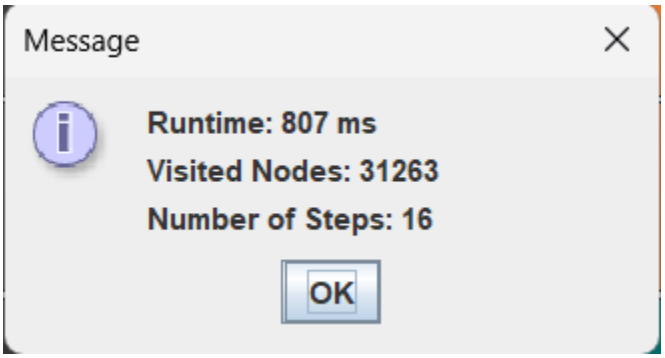
- Initial :



- Akhir :

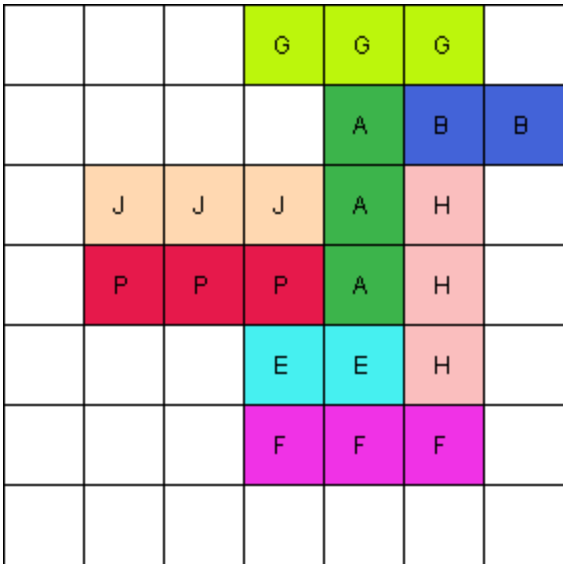


- Info :

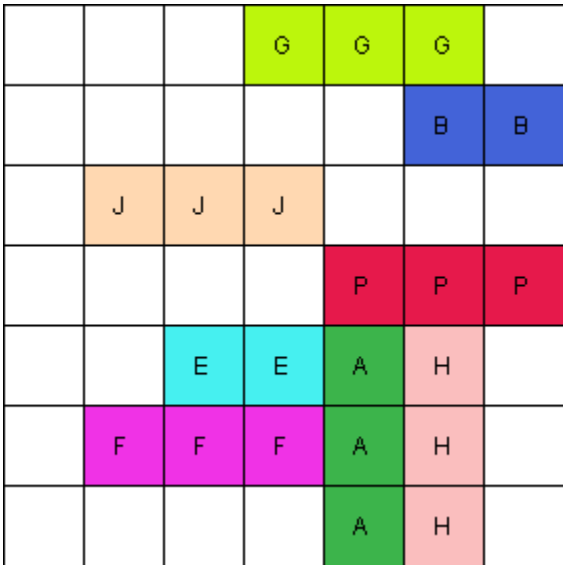


3. Testcase 3

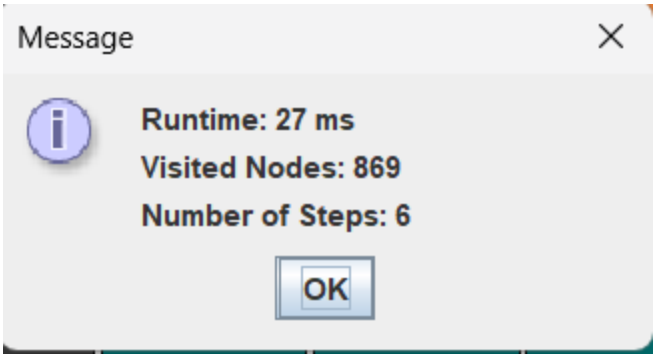
- Initial :



- Akhir :

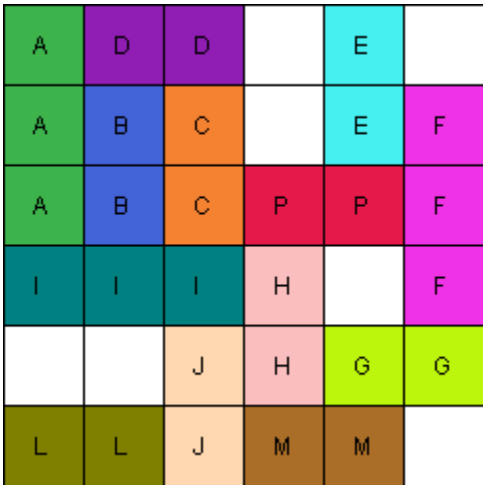


- Info :

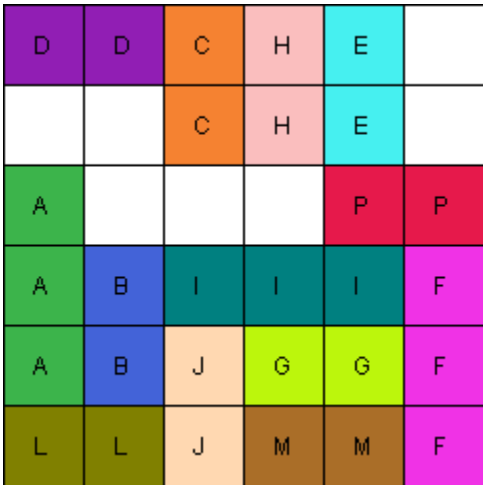


4. Testcase 4

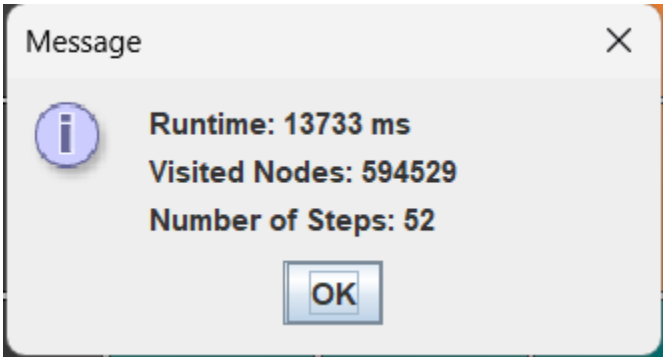
- Initial :



- Akhir :



- Info :



Analisis Hasil Pengujian

Pengujian dilakukan menggunakan empat buah *test case* yang merepresentasikan konfigurasi *puzzle Rush Hou*r dengan kompleksitas berbeda-beda. Setiap *test case* diuji dengan empat algoritma pencarian jalur, yaitu *Uniform Cost Searc*h (UCS), *Greedy Best First Search* (GBFS), *A**, dan *Iterative Deepening A** (IDA*). Evaluasi dilakukan terhadap performa waktu pencarian, efisiensi jalur, dan akurasi solusi yang dihasilkan.Terdapat beberapa perbedaan performa di antara algoritma tersebut:

Algoritma A* menunjukkan performa yang sangat baik dengan menghasilkan solusi optimal dalam waktu yang relatif cepat. A* memanfaatkan fungsi heuristik yang dikombinasikan dengan biaya perjalanan sejauh ini, sehingga mampu menyeimbangkan antara eksplorasi dan eksploitasi. Meskipun jumlah node yang dikunjungi cukup besar, solusi yang diperoleh selalu merupakan solusi dengan langkah minimal. Hal ini menunjukkan keunggulan A* dalam menemukan solusi yang optimal secara konsisten.

Sementara itu, algoritma IDA* juga mampu menemukan solusi optimal, namun dengan waktu pencarian yang cenderung lebih lama dibandingkan A*. Hal ini disebabkan oleh sifat iteratif dari IDA* yang melakukan pencarian berulang dengan batas biaya yang meningkat secara bertahap. Meskipun penggunaan memori IDA* lebih hemat, jumlah iterasi yang dilakukan membuatnya kurang efisien pada puzzle dengan kompleksitas tinggi.

Uniform Cost Search memiliki keunggulan dalam menjamin optimalitas solusi, tetapi kinerjanya sangat dipengaruhi oleh banyaknya kemungkinan langkah yang harus dieksplorasi. UCS cenderung mengunjungi lebih banyak node dibandingkan A*, karena tidak memanfaatkan

informasi heuristik. Akibatnya, waktu eksekusi UCS dalam eksperimen ini relatif lebih lama, terutama pada papan puzzle yang besar dan kompleks.

Greedy Best First Search menjadi algoritma tercepat dalam beberapa kasus karena hanya mempertimbangkan estimasi jarak ke tujuan. Namun, kecepatan ini mengorbankan optimalitas solusi. Pada eksperimen ini, GBFS sering kali menghasilkan solusi yang tidak minimal dan bahkan gagal menemukan solusi pada konfigurasi tertentu karena terlalu cepat mempersempit ruang pencarian. Hal ini menunjukkan bahwa meskipun cepat, GBFS kurang andal dalam menemukan solusi terbaik.

Eksperimen kali ini juga mengimplementasikan bonus, yaitu **fungsi heuristik**, **algoritma pencarian tambahan**, dan **GUI**. Fungsi heuristik yang digunakan adalah fungsi berbasis blok pembatas, dan fungsi berbasis jarak *primary piece* ke target. Fungsi heuristik tidak terlalu berpengaruh kepada algoritma A*, IDA*, dan Uniform Cost Search karena *completeness* dari ketiga algoritma tersebut. *Sample space* yang cukup kecil juga menyebabkan Fungsi heuristik cukup aman digunakan pada pendekatan *Greedy Best First Search* pada kasus-kasus sederhana.
Implementasi struktur data dan akses memori yang baik adalah kunci utama dari program ini. **Program ini dapat menyelesaikan Puzzle tersulit menurut *Fogleman* dalam waktu 57ms (*Test Case* 4).**

Lampiran

Tautan *repository* GitHub : Pranala GitHub

No	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	Implementasi algoritma pathfinding alternatif	✓	
6	Implementasi 2 atau lebih heuristik alternatif	✓	
7	Program memiliki GUI	✓	
8	Program dan laporan dibuat (kelompok) sendiri	✓	

Daftar Pustaka

1. Institut Teknologi Bandung, "Mata Kuliah IF3170 - Inteligensi Buatan," *Website Kuliah ITB*, [Online]. Tersedia: http://kuliah.itb.ac.id → STEI → Teknik Informatika → IF3170.
2. S. J. Russell dan P. Norvig, *Artificial Intelligence: A Modern Approach*, Edisi ke-3, Prentice-Hall International, Inc, 2010. [Online]. Tersedia: http://aima.cs.berkeley.edu/ (Edisi ke-2).
3. Massachusetts Institute of Technology, *Free Online Course Materials*, MIT OpenCourseWare. [Online]. Tersedia: http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/
4. R. Dechter, "Lecture Notes in Informed Heuristic Search," ICS 271, University of California, Irvine, Fall 2008. [Online]. Tersedia: http://www.ics.uci.edu/~dechter/courses/ics-271/fall-08/lecturenotes/4.InformedHeuristicSearch.ppt.