Laporan Tugas Kecil 3 IF2211 Strategi Algortima

SEMESTER II 2024/2025

Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding

Oleh

Faqih Muhammad Syuhada

NIM : 13523057

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

Februari 2025

DAFTAR ISI

BAB I
DESKRIPSI TUGAS..... 3
I.1 Puzzle Rush Hour..... 3
I.2 Algoritma PathFinding UCS (Uniform Cost Search)..... 4
I.3 Algoritma Greedy Best First Search..... 4
I.4 Algoritma A*..... 5
BAB II STRATEGI ALGORITMA.....7
II.1 File Astar.cpp..... 7
II.2 File UCS.cpp.....8
II.3 File Greedy BFS.cpp.....8
II.4 File Iter_deepening.cpp..... 9
BAB III ANALISIS ALGORITMA..... 10
BAB IV IMPLEMENTASI DALAM BAHASA C++.....13
IV.1 Repository Program..... 13
IV.2 Source Code.....13
BAB V TESTING..... 47
BAB VI LAMPIRAN..... 59

BAB I

DESKRIPSI TUGAS

I.1 Puzzle Rush Hour

Puzzle Rush Hour merupakan permainan berbasis grid yang bertujuan menggeser kendaraan agar kendaraan utama dapat keluar dari papan permainan melalui pintu keluar. Kendaraan dapat digeser secara horizontal atau vertikal, tetapi tidak dapat berputar atau menembus kendaraan lain.

Komponen penting dari permainan Rush Hour terdiri dari:

1. Papan – Papan merupakan tempat permainan dimainkan. Papan terdiri atas cell, yaitu sebuah singular point dari papan. Sebuah piece akan menempati cell-cell pada papan. Ketika permainan dimulai, semua piece telah diletakkan di dalam papan dengan konfigurasi tertentu berupa lokasi piece dan orientasi, antara horizontal atau vertikal. Hanya primary piece yang dapat digerakkan keluar papan melewati pintu keluar. Piece yang bukan primary piece tidak dapat digerakkan keluar papan. Papan memiliki satu pintu keluar yang pasti berada di dinding papan dan sejajar dengan orientasi primary piece.
2. Piece – Piece adalah sebuah kendaraan di dalam papan. Setiap piece memiliki posisi, ukuran, dan orientasi. Orientasi sebuah piece hanya dapat berupa horizontal atau vertikal–tidak mungkin diagonal. Piece dapat memiliki beragam ukuran, yaitu jumlah cell yang ditempati oleh piece. Secara standar, variasi ukuran sebuah piece adalah 2-piece (menempati 2 cell) atau 3-piece (menempati 3 cell). Suatu piece tidak dapat digerakkan melewati/menembus piece yang lain.
3. Primary Piece – Primary piece adalah kendaraan utama yang harus dikeluarkan dari papan (biasanya berwarna merah). Hanya boleh terdapat satu primary piece.

4. Pintu Keluar – Pintu keluar adalah tempat primary piece dapat digerakkan keluar untuk menyelesaikan permainan
5. Gerakan — Gerakan yang dimaksudkan adalah pergeseran piece di dalam permainan. Piece hanya dapat bergerak/bergeser lurus sesuai orientasinya (atas-bawah jika vertikal dan kiri-kanan jika horizontal). Suatu piece tidak dapat digerakkan melewati/menembus piece yang lain.

Permainan dimulai dengan papan yang berantakan. Pemain dapat menggeser-geser piece (termasuk primary piece) untuk membentuk jalan lurus antara primary piece dan pintu keluar. Puzzle berikut dinyatakan telah selesai apabila primary piece dapat digeser keluar papan melalui pintu keluar. Setiap blok puzzle dapat di geser sesuai arahnya horizontal atau vertikal.

Tugas dari program ini adalah menemukan cukup satu solusi dari permainan Rush Hour dengan menggunakan Algoritma *Path Finding*, atau menampilkan bahwa solusi tidak ditemukan jika tidak ada solusi yang mungkin dari puzzle.

I.2 Algoritma *PathFinding* UCS (Uniform Cost Search)

Algoritma dimana pencarian solusi optimal didasarkan pada fungsi evaluasi f(n) untuk setiap simpul, di mana f(n) = g(n), dimana g(n) adalah cost dari akar ke simpul n. Pada program ini, cost dihitung dengan menambah cost kumulatif sebesar 1 untuk setiap langkah (cost dibuat seragam). Algoritma ini memanfaatkan priority queue yang mengurutkan nilai g(n) dari yang terkecil. UCS pada dasarnya adalah Dijkstra's algorithm yang menggunakan actual cost (jumlah moves) sebagai prioritas dalam queue, dimana implementasinya menggunakan UCSState yang menyimpan board configuration dan move history dengan bestCost map untuk tracking cost optimal ke setiap state.

I.3 Algoritma Greedy Best First Search

Algoritma yang menggunakan fungsi evaluasi f(n) untuk setiap simpul, di mana f(n) = h(n), yang merupakan perkiraan cost dari simpul n menuju tujuan. Pencarian

greedy best-first akan memperluas simpul yang tampaknya paling dekat dengan tujuan. Algoritma ini memanfaatkan priority queue yang mengurutkan nilai h(n) dari yang terkecil. Greedy Best First Search memiliki beberapa permasalahan. Pertama, metode ini tidak lengkap. Kedua, metode ini rentan terjebak dalam optimal lokal minima atau plateau. Ketiga, pendekatan ini tidak dapat dibalik atau diubah (irrevocable). GreedyBestFirst hanya menggunakan heuristic cost sebagai dasar pengambilan keputusan dalam priority queue tanpa mempertimbangkan cost perjalanan sejauh ini, dimana implementasinya menggunakan state greedy yang menyimpan current board state dan history of moves dengan visited set untuk mencegah infinite loops, menggunakan heuristic function untuk memperkirakan jarak ke goal state, dan langsung meng expand node dengan cost terendah tanpa mempedulikan berapa langkah yang telah diambil untuk mencapai node tersebut, sehingga algoritma ini lebih cepat dari A* tapi tidak menjamin solusi optimal.

I.4 Algoritma A*

Algoritma yang menghindari perluasan path yang cost-nya sudah bernilai tinggi. Fungsi evaluasi f(n) didefinisikan sebagai g(n) + h(n), di mana g(n) adalah cost yang telah dikeluarkan untuk mencapai simpul n, dan h(n) adalah perkiraan cost dari simpul n ke tujuan. Jadi, f(n) adalah perkiraan total cost path melalui simpul n ke tujuan. Algoritma ini memanfaatkan priority queue yang mengurutkan nilai f(n) dari yang terkecil. Heuristik yang digunakan pada algoritma A* admissible karena untuk setiap node n, $h(n) \leq h^*(n)$, di mana $h^*(n)$ adalah cost sebenarnya untuk mencapai keadaan tujuan dari n. Heuristik yang bersifat admissible tidak pernah melebih-lebihkan cost untuk mencapai tujuan, yaitu, heuristik ini bersifat optimis. A mengkombinasikan cost sejauh ini dengan estimated cost ke tujuan untuk membentuk cost yang digunakan sebagai prioritas dalam queue, dimana implementasinya menggunakan priority queue yang menyimpan state Astar yang berisi board configuration, moves history, dan costs, serta menggunakan unordered_map untuk tracking cost optimal ke state tersebut dan closedSet untuk

mencegah revisit ke state yang sama, dengan pemilihan heuristik sehingga algoritma ini menjamin menemukan solusi optimal dengan mempertimbangkan both actual cost dan estimated remaining cost.

I.5 Algoritma *Iterative Deepening*

Algoritma *Iterative Deepening* Iterative Deepening adalah algoritma pathfinding yang menggabungkan keunggulan DFS (Depth-First Search) dan BFS (Breadth-First Search).Cara kerjanya sendiri yaitu mulai dengan kedalaman maksimum yaitu 1. Lalu, dilakukan DFS dengan batas kedalaman tersebut. Jika solusi tidak ditemukan, tingkatkan kedalaman dan ulangi DFS. Proses berulang sampai solusi ditemukan atau mencapai batas maksimum.Algoritma ini menjamin menemukan solusi jika ada. Lalu, Menemukan solusi dengan path terpendek dan hanya membutuhkan memori O(d), d = kedalaman solusi. Serta *time complexity* O(b^d). Keunggulan dari Algoritma ini sendiri yaitu menggunakan memori yang efisien seperti DFS tetapi menemukan solusi optimal seperti BFS. Lalu Cocok untuk ruang pencarian dalam dan dapat dikombinasikan dengan heuristik.

BAB II

STRATEGI ALGORITMA

II.1 File Astar.cpp

File ini berisi program untuk melakukan pencarian path dengan menggunakan algoritma A*.

Nama Kelas	Deskripsi
AStar	Kelas ini mengimplementasikan algoritma pencarian A* untuk menyelesaikan puzzle Rush Hour. Menggunakan fungsi heuristik untuk memperkirakan jarak ke tujuan dan memprioritaskan state dengan estimasi biaya terkecil.

Nama Atribut/Method	Deskripsi
heuristic	Menentukan jenis heuristik yang akan digunakan dalam perhitungan (misalnya 'manhattan' atau 'blocking').
Solve	Menjalankan algoritma pencarian untuk menyelesaikan puzzle dari board awal hingga mencapai goal state. Mengembalikan objek Solution yang berisi urutan langkah solusi, jumlah node yang dikunjungi, dan waktu eksekusi.

II.2 File UCS.cpp

Nama Kelas	Deskripsi
UCS	Kelas ini mengimplementasikan Uniform Cost Search (UCS), yaitu pencarian berdasarkan cost aktual tanpa memperhatikan heuristik, namun dapat menggunakan heuristik jika tersedia untuk keperluan perbandingan.

Nama Atribut/Method	Deskripsi
Solve	Menjalankan algoritma pencarian untuk menyelesaikan puzzle dari board awal hingga mencapai goal state. Mengembalikan objek Solution yang berisi urutan langkah solusi, jumlah node yang dikunjungi, dan waktu eksekusi.

II.3 File Greedy BFS.cpp

Nama Kelas	Deskripsi
GreedyBestFirst	Kelas ini mengimplementasikan algoritma Greedy Best-First Search untuk menyelesaikan puzzle Rush Hour. Menggunakan nilai heuristik saja tanpa mempertimbangkan cost sejauh ini.

Nama Atribut/Method	Deskripsi
heuristic	Menentukan jenis heuristik yang akan digunakan dalam perhitungan (misalnya 'manhattan' atau 'blocking').
Solve	Menjalankan algoritma pencarian untuk menyelesaikan puzzle dari board awal hingga mencapai goal state. Mengembalikan objek Solution yang berisi urutan langkah solusi, jumlah node yang dikunjungi, dan waktu eksekusi.

II.4 File Iter_deepening.cpp

Nama Kelas	Deskripsi
IterativeDeepening	Kelas ini mengimplementasikan algoritma pencarian Iterative Deepening Depth-First Search (IDDFS). Algoritma ini menggabungkan kelebihan DFS dan BFS dengan melakukan pencarian sampai kedalaman tertentu lalu meningkatkannya secara iteratif.

Nama Atribut/Method	Deskripsi
Solve	Menjalankan algoritma pencarian untuk menyelesaikan puzzle dari board awal hingga mencapai goal state. Mengembalikan objek Solution yang berisi urutan langkah solusi, jumlah node yang dikunjungi, dan waktu eksekusi.

BAB III

ANALISIS ALGORITMA

III.1 Perbandingan dan Karakteristik Fungsi Biaya

Setiap algoritma pencarian pada puzzle Rush Hour memiliki definisi fungsi evaluasi yang membedakan strategi pencariannya. Uniform Cost Search (UCS) hanya mengandalkan path cost dari state awal ke state saat ini, sehingga g(n) mencerminkan jumlah langkah yang telah dilakukan, dan f(n) = g(n). Karena UCS tidak menggunakan heuristik, algoritma ini menjelajahi semua kemungkinan secara sistematis berdasarkan urutan biaya terkecil.

Sebaliknya, Greedy Best-First Search mengabaikan biaya yang telah ditempuh (g(n)), dan hanya mempertimbangkan estimasi jarak ke goal (h(n)), sehingga f(n) = h(n). Pendekatan ini sangat bergantung pada kualitas heuristik yang digunakan, karena lebih mengutamakan kecepatan mencapai goal daripada optimalitas jalur.

Iterative Deepening Depth-First Search (IDDFS) berbeda karena tidak menggunakan f(n) ataupun g(n) secara eksplisit. Algoritma ini menggabungkan pendekatan depth-first dengan peningkatan batas kedalaman iteratif, sehingga tetap hemat memori seperti DFS, tetapi menjamin pencarian solusi secara menyeluruh seperti BFS. Meskipun tidak seefisien A*, IDDFS menjamin solusi ditemukan jika ada, dengan kedalaman minimum.

Sementara itu, A* menggunakan kombinasi dari dua komponen penting: g(n) sebagai langkah aktual dari state awal ke node saat ini, dan h(n) sebagai estimasi jarak ke tujuan. Maka f(n) = g(n) + h(n). Kombinasi ini memungkinkan A* untuk melakukan pencarian yang efisien sekaligus menjamin solusi optimal, asalkan heuristik yang digunakan adalah admissible (tidak melebihi biaya sebenarnya ke goal).

III.2 Admissibility dari Heuristik pada A*

Dalam implementasi algoritma A*, dua jenis heuristik yang digunakan telah terbukti bersifat *admissible*. Pertama adalah Manhattan Distance, yaitu menghitung jarak minimal dalam satuan langkah dari ujung *primary car* ke posisi pintu keluar, tanpa memperhitungkan mobil lain yang menghalangi. Karena tidak pernah melebih-lebihkan biaya sebenarnya, heuristik ini aman digunakan dan tetap menjamin optimalitas.

Kedua adalah Blocking Cars, yang menghitung jumlah mobil yang secara langsung menghalangi jalur *primary car* menuju pintu keluar. Meskipun lebih spesifik dibanding Manhattan, heuristik ini tetap admissible karena hanya menghitung jumlah mobil penghalang tanpa mempertimbangkan gerakan ekstra yang diperlukan untuk memindahkannya. Oleh karena itu, nilainya juga selalu sama atau kurang dari biaya aktual menuju solusi.

III.3 UCS dan Perbedaannya dengan BFS

Walaupun UCS dan BFS sering dianggap serupa dalam konteks edge cost yang seragam (yakni setiap langkah dianggap memiliki bobot sama), secara teknis UCS tidak identik dengan BFS. UCS menggunakan priority queue berdasarkan g(n) untuk memilih node dengan path cost terkecil, sedangkan BFS menggunakan FIFO queue tanpa mempertimbangkan biaya. Dalam kasus Rush Hour, keduanya dapat menghasilkan solusi optimal karena semua gerakan bernilai satu, namun perbedaan cara seleksi node membuat urutan eksplorasi state bisa berbeda.

III.4 Efisiensi A* Dibandingkan UCS

Dari sudut pandang efisiensi, A* memiliki keunggulan signifikan dibanding UCS. Karena A* memanfaatkan heuristik, pencariannya lebih terarah dan tidak perlu menjelajahi semua kemungkinan seperti UCS. UCS, yang bekerja tanpa informasi estimatif, sering kali membuang waktu pada banyak state yang

kurang relevan dengan solusi. Sementara itu, A* dapat memfokuskan pencarian hanya pada jalur-jalur yang terlihat menjanjikan, yang secara praktis sangat mengurangi jumlah node yang harus dikunjungi. Selama heuristik yang digunakan bersifat admissible dan konsisten, A* tetap menjamin solusi yang ditemukan adalah yang paling optimal.

III.5 Keterbatasan Greedy Best-First Search

Berbeda dengan A*, Greedy Best-First Search (GBFS) tidak menjamin optimalitas solusi. Hal ini disebabkan oleh sifat GBFS yang hanya mempertimbangkan h(n), yaitu estimasi jarak ke goal, tanpa melihat berapa banyak langkah yang sudah dilakukan (g(n)). Akibatnya, algoritma ini bisa saja memilih jalur yang terlihat menjanjikan di awal (berdasarkan heuristik), tetapi sebenarnya memerlukan lebih banyak langkah untuk mencapai goal. Ini membuat GBFS sangat rentan terhadap local minima, yaitu state yang tampak dekat dengan goal tapi sebenarnya bukan bagian dari jalur optimal. Meskipun demikian, GBFS seringkali lebih cepat mencapai goal dibanding A* atau UCS, sehingga berguna saat waktu lebih diutamakan dibanding kualitas solusi.

III.6 Kesimpulan Analisa Algoritma

Secara keseluruhan, pemilihan algoritma untuk Rush Hour Puzzle bergantung pada trade-off antara waktu eksekusi, jumlah memori, dan kebutuhan optimalitas solusi. A* memberikan keseimbangan terbaik antara efisiensi dan kualitas solusi, sedangkan UCS cocok jika heuristik tidak tersedia. GBFS lebih cepat tetapi tidak optimal, dan IDS cocok untuk pencarian memori rendah dengan jaminan pencapaian solusi minimum.

BAB IV

IMPLEMENTASI DALAM BAHASA C++

IV.1 Repository Program

Berikut adalah pranala ke repository program :

https://github.com/FaqihMSY/Tucil3_13523057

IV.2 Source Code

IV.2.1 car.cpp

```
#include "car.hpp"

Car::Car()
    : id('.'), start({0, 0}), length(0),
isHorizontal(true), isPrimary(false) {}

Car::Car(char id, Position start, int length, bool
isHorizontal, bool isPrimary)
    : id(id), start(start), length(length),
isHorizontal(isHorizontal), isPrimary(isPrimary) {}

Position Car::getPosition() const {
    return start;
}

int Car::getLength() const {
    return length;
}

bool Car::getIsHorizontal() const {
    return isHorizontal;
}

bool Car::getIsPrimary() const {
    return isPrimary;
```

```
}

char Car::getId() const {
    return id;
}
```

IV.2.2 board.cpp

```
#include "board.hpp"
#include <cctype>
#include <iostream>
#include <set>
#include <sstream>
#include <utility>


Board::Board(int rows, int cols, int numPieces):
rows(rows), cols(cols), numPieces(numPieces) {
    grid.assign(rows, std::vector<char>(cols, '.'));
}

bool Board::isValidPosition(int r, int c) const {
    return r >= 0 && r < rows && c >= 0 && c < cols;
}

bool Board::isEdge(int r, int c) const {
    return r == 0 || r == rows - 1 || c == 0 ||  c == cols
- 1;
}

bool Board::isValidConfiguration(const
std::vector<std::string> &cfg) const {
    if (static_cast<int>(cfg.size()) != rows) {
        std::cerr << "Invalid row count. Expected: " <<
```

```
rows << ", Got: " << cfg.size() << "\n";
        return false;
    }
    for (size_t i = 0; i < cfg.size(); ++i) {
        if (static_cast<int>(cfg[i].size()) != cols) {
            std::cerr << "Invalid column count at row " <<
i << ". Expected: " << cols
                      << ", Got: " << cfg[i].size() <<
"\n";
            return false;
        }
    }
    return true;
}

bool Board::validateAlignment() const {
    auto it = cars.find('P');
    if (it == cars.end()) return false;

    const Car& primary = it->second;
    Position p = primary.getPosition();

    if (primary.getIsHorizontal()){

        // std::cout<<"hor\n";
        return p.row == exitPos.row;
    }
    else{

        // std::cout<<"nahhor\n";
        return p.col == exitPos.col;
    }
}

bool Board::loadFromString(const std::vector<std::string>
&cfg) {
```

```
    if (!isValidConfiguration(cfg)) {
        std::cerr << "Invalid configuration dimensions\n"
<< toString() << "\n";
        return false;
    }

    cars.clear();
    grid.clear();
    for (const auto &line : cfg) {
        grid.push_back(std::vector<char>(line.begin(),
line.end()));
    }


    Position pStart{-1,-1};
    bool horiz = true;
    int leng = 0;

    for (int r = 0; r < rows && pStart.row == -1; ++r) {
        for (int c = 0; c < cols; ++c) {
            if (grid[r][c] == 'P') {
                pStart = {r,c};
                break;
            }
        }
    }

    if (pStart.row == -1) {
        std::cerr << "Primary piece (P) not found\n" <<
toString() << "\n";
        return false;
    }
    if (pStart.col + 1 < cols &&
grid[pStart.row][pStart.col + 1] == 'P') {
        horiz = true;
        int c = pStart.col;
```

```
        while (c < cols && grid[pStart.row][c] == 'P') {
            ++leng;
            ++c;
        }
    } else {
        horiz = false;
        int r = pStart.row;
        while (r < rows && grid[r][pStart.col] == 'P') {
            ++leng;
            ++r;
        }
    }

    if (leng < 2 || leng > 3) {
        std::cerr << "Primary piece has invalid length: "
<< leng << "\n" << toString() << "\n";
        return false;
    }

    cars.emplace('P', Car('P', pStart, leng, horiz,
true));

    for (int r = 0; r < rows; ++r) {
        for (int c = 0; c < cols; ++c) {
            char id = grid[r][c];
            if (id == '.' || id == 'P' || cars.find(id) !=
cars.end()) continue;

            bool h = true;
            int l = 1;
            if (c + 1 < cols && grid[r][c + 1] == id) {
                int cc = c + 1;
                while (cc < cols && grid[r][cc] == id) {
                    ++l;
                    ++cc;
                }
```

```
            } else {
                h = false;
                int rr = r + 1;
                while (rr < rows && grid[rr][c] == id) {
                    ++l;
                    ++rr;
                }
            }

            if (l < 2 || l > 3) {
                std::cerr << "Car " << id << " has invalid
length: " << l << "\n" << toString() << "\n";
                return false;
            }
            cars.emplace(id, Car(id, Position{r,c}, l, h,
false));
        }
    }

    if (!validateAlignment()) {
        std::cerr << "Primary piece not aligned with
exit\n"
                  << "Primary position: (" <<
cars['P'].getPosition().row << ","
                  << cars['P'].getPosition().col << ")\n"
                  << "Exit position: (" << exitPos.row <<
"," << exitPos.col << ")\n"
                  << toString() << "\n";
        return false;
    }

    return true;
}

char Board::getCell(int r,int c) const { return
isValidPosition(r,c)?grid[r][c]:'.'; }
```

```
bool Board::isValidMove(const Car& car, const std::string&
dir) const {
    Position p = car.getPosition();
    int l = car.getLength();
    if (car.getIsHorizontal()) {
        if (dir == "kiri") {
            if (!(p.col > 0 && grid[p.row][p.col-1] ==
'.')) {
                // std::cerr << "Invalid left move for car
at (" << p.row << "," << p.col << ")\n";
                return false;
            }
            return true;
        }
        if (dir == "kanan") {
            if (!(p.col + l < cols && grid[p.row][p.col+l]
== '.')) {
                // std::cerr << "Invalid right move for
car at (" << p.row << "," << p.col << ")\n";
                return false;
            }
            return true;
        }
    } else {
        if (dir == "atas") {
            if (!(p.row > 0 && grid[p.row-1][p.col] ==
'.')) {
                // std::cerr << "Invalid up move for car
at (" << p.row << "," << p.col << ")\n";
                return false;
            }
            return true;
        }
        if (dir == "bawah") {
            if (!(p.row + l < rows && grid[p.row+l][p.col]
```

```
== '.')) {
                // std::cerr << "Invalid down move for car
at (" << p.row << "," << p.col << ")\n";
                return false;
            }
            return true;
        }
    }
    std::cerr << "Invalid direction: " << dir << "\n";
    return false;
}

bool Board::makeMove(char id, const std::string& dir) {
    auto it = cars.find(id);
    if (it == cars.end() || !isValidMove(it->second, dir))
{
        // std::cerr << "Invalid move: car=" << id << ",
direction=" << dir << "\n";
        return false;
    }
    Car &car=it->second; Position old=car.getPosition();
Position neu=old;
    if(dir=="kiri")--neu.col; else
if(dir=="kanan")++neu.col; else if(dir=="atas")--neu.row;
else ++neu.row;
    for(int i=0;i<car.getLength();++i)
grid[car.getIsHorizontal()?old.row:old.row+i][car.getIsHor
izontal()?old.col+i:old.col] = '.';
    car = Car(id, neu, car.getLength(),
car.getIsHorizontal(), id=='P');
    for(int i=0;i<car.getLength();++i)
grid[car.getIsHorizontal()?neu.row:neu.row+i][car.getIsHor
izontal()?neu.col+i:neu.col] = id;
    return true;
}
```

```
std::vector<std::pair<char,std::string>>
Board::getPossibleMoves() const {
    std::vector<std::pair<char,std::string>> mv;
    for (auto &kv:cars) {
        const Car &c=kv.second;
        if(c.getIsHorizontal()){
            if(isValidMove(c,"kiri"))
mv.push_back({kv.first,"kiri"});
            if(isValidMove(c,"kanan"))
mv.push_back({kv.first,"kanan"});
        }
        else {
            if(isValidMove(c,"atas"))
mv.push_back({kv.first,"atas"});
            if(isValidMove(c,"bawah"))
mv.push_back({kv.first,"bawah"});
        }
    }
    return mv;
}

bool Board::isGoalState() const {
    auto it = cars.find('P');
    if (it == cars.end()) {
        std::cerr << "Primary piece not found in goal
check\n";
        return false;
    }
    const Car&p=it->second; Position pos=p.getPosition();
    if(p.getIsHorizontal()) return pos.row==exitPos.row &&
((pos.col+p.getLength()==exitPos.col)||(exitPos.col+1==pos
.col));
    return pos.col==exitPos.col &&
((pos.row+p.getLength()==exitPos.row)||(exitPos.row+1==pos
.row));
}
```

```
std::string Board::toString() const {
    std::stringstream ss;
    for(int r=0;r<rows;++r){
        for(int c=0;c<cols;++c)
        ss<<grid[r][c];
        if(r<rows-1) ss<<'\n';
    }
    return ss.str();
}

Car Board::getPrimaryCar() const {
    auto it=cars.find('P');
    return it!=cars.end()?it->second:Car();
}
Position Board::getExitPosition() const { return exitPos;
}
```

IV.2.3 astar.cpp

```
#include "astar.hpp"
#include "../heuristics/manhattan.hpp"
#include "../heuristics/blocking_cars.hpp"
#include <queue>
#include <unordered_map>
#include <unordered_set>

AStar::AStar(std::string h) : heuristic(h) {}

struct AStarState {
    Board board;
    std::vector<std::pair<char, std::string>> moves;
    int g_cost;
```

```
    int h_cost;

    int f_cost() const { return g_cost + h_cost; }

    bool operator>(const AStarState& other) const {
        if (f_cost() == other.f_cost()) {
            return h_cost > other.h_cost;
        }
        return f_cost() > other.f_cost();
    }
};

Solution AStar::solve(const Board& initialBoard) {
    auto startTime =
std::chrono::high_resolution_clock::now();

    std::priority_queue<AStarState,
std::vector<AStarState>, std::greater<>> openSet;
    std::unordered_map<std::string, int> gScore;
    std::unordered_set<std::string> closedSet;
    int nodesVisited = 0;

    Position exitPos = initialBoard.getExitPosition();
    bool exitOnTopOrLeft = exitPos.col == -1 ||
exitPos.row == -1;
    int initial_h = 0;

    if (heuristic != "none") {
        if (exitOnTopOrLeft) {
            initial_h = (heuristic == "manhattan") ?

ManhattanHeuristic::calculateReversed(initialBoard) :

BlockingCarsHeuristic::calculateReversed(initialBoard);
        } else {
            initial_h = (heuristic == "manhattan") ?
```

```
ManhattanHeuristic::calculate(initialBoard) :

BlockingCarsHeuristic::calculate(initialBoard);
        }

        if (initial_h == INT_MAX) {
            return {{}, nodesVisited, 0};
        }
    }

    std::string initialStr = initialBoard.toString();
    gScore[initialStr] = 0;
    openSet.push({initialBoard, {}, 0, initial_h});

    while (!openSet.empty()) {
        AStarState current = openSet.top();
        openSet.pop();
        nodesVisited++;

        if (current.board.isGoalState()) {
            auto endTime =
std::chrono::high_resolution_clock::now();
            auto duration =
std::chrono::duration_cast<std::chrono::milliseconds>(endT
ime - startTime);
            return {current.moves, nodesVisited,
duration.count()};
        }

        std::string currentStr = current.board.toString();
        if (closedSet.count(currentStr)) continue;
        closedSet.insert(currentStr);
        for (const auto& [piece, direction] :
current.board.getPossibleMoves()) {
            Board nextBoard = current.board;
```

```
            if (!nextBoard.makeMove(piece, direction))
continue;

            std::string nextStr = nextBoard.toString();
            if (closedSet.count(nextStr)) continue;

            int tentative_g = current.g_cost + 1;
            if (gScore.count(nextStr) && tentative_g >=
gScore[nextStr]) continue;

            int next_h = 0;
            if (heuristic != "none") {
                if (exitOnTopOrLeft) {
                    next_h = (heuristic == "manhattan") ?

ManhattanHeuristic::calculateReversed(nextBoard) :

BlockingCarsHeuristic::calculateReversed(nextBoard);
                } else {
                    next_h = (heuristic == "manhattan") ?

ManhattanHeuristic::calculate(nextBoard) :

BlockingCarsHeuristic::calculate(nextBoard);
                }

                if (next_h == INT_MAX) {
                    continue;
                }
            }

            auto nextMoves = current.moves;
            nextMoves.push_back({piece, direction});
            gScore[nextStr] = tentative_g;
            openSet.push({nextBoard, nextMoves,
tentative_g, next_h});
```

```
        }
    }

    return {{}, nodesVisited, 0};
}
```

IV.2.4 greedy.cpp

```
#include "greedy.hpp"
#include "../heuristics/manhattan.hpp"
#include "../heuristics/blocking_cars.hpp"
#include <queue>
#include <unordered_set>

GreedyBestFirst::GreedyBestFirst(std::string h) :
heuristic(h) {}

struct GreedyState {
    Board board;
    std::vector<std::pair<char, std::string>> moves;
    int h_cost;

    bool operator>(const GreedyState& other) const {
        return h_cost > other.h_cost;
    }
};

Solution GreedyBestFirst::solve(const Board& initialBoard)
{
    auto startTime =
std::chrono::high_resolution_clock::now();

    std::priority_queue<GreedyState,
std::vector<GreedyState>, std::greater<>> openSet;
```

```
    std::unordered_set<std::string> visited;
    int nodesVisited = 0;

    int initial_h;
    if (heuristic == "manhattan") {
        initial_h =
ManhattanHeuristic::calculate(initialBoard);
    } else {
        initial_h =
BlockingCarsHeuristic::calculate(initialBoard);
    }

    openSet.push({initialBoard, {}, initial_h});
    visited.reserve(10000);

    while (!openSet.empty()) {
        GreedyState current = openSet.top();
        openSet.pop();
        nodesVisited++;

        if (current.board.isGoalState()) {
            auto endTime =
std::chrono::high_resolution_clock::now();
            auto duration =
std::chrono::duration_cast<std::chrono::milliseconds>(endT
ime - startTime);
            return {current.moves, nodesVisited,
duration.count()};
        }

        std::string currentStr = current.board.toString();
        if (visited.find(currentStr) != visited.end()) {
            continue;
        }
        visited.insert(currentStr);
```

```
        auto possibleMoves =
current.board.getPossibleMoves();
        for (const auto& move : possibleMoves) {
            Board nextBoard = current.board;
            if (!nextBoard.makeMove(move.first,
move.second)) {
                continue;
            }

            std::string nextStr = nextBoard.toString();
            if (visited.find(nextStr) != visited.end()) {
                continue;
            }

            int next_h;
            if (heuristic == "manhattan") {
                next_h =
ManhattanHeuristic::calculate(nextBoard);
            } else {
                next_h =
BlockingCarsHeuristic::calculate(nextBoard);
            }

            auto nextMoves = current.moves;
            nextMoves.push_back(move);
            openSet.push({nextBoard, nextMoves, next_h});
        }
    }

    return {{}, nodesVisited, 0};
}
```

IV.2.5 iter_deepening.cpp

```cpp
#include "iter_deepening.hpp"
#include <chrono>
#include <unordered_set>
#include <algorithm>

IterativeDeepening::IterativeDeepening() {}

struct IDState {
    Board board;
    std::vector<std::pair<char, std::string>> moves;
    int depth;

    bool operator==(const IDState& other) const {
        return board.toString() == other.board.toString();
    }
};

Solution IterativeDeepening::solve(const Board&
initialBoard) {
    auto startTime =
std::chrono::high_resolution_clock::now();
    int maxDepth = 1;
    int nodesVisited = 0;

    Position exitPos = initialBoard.getExitPosition();
    Car primaryCar = initialBoard.getPrimaryCar();
    bool isHorizontal = primaryCar.getIsHorizontal();

    while (maxDepth <= MAX_DEPTH) {
        std::unordered_set<std::string> visited;
        visited.reserve(10000);

        IDState initial{initialBoard, {}, 0};
        auto result = dfs(initial, 0, maxDepth, visited,
nodesVisited, exitPos, isHorizontal);
```

```
        if (!result.moves.empty()) {
            auto endTime =
std::chrono::high_resolution_clock::now();
            auto duration =
std::chrono::duration_cast<std::chrono::milliseconds>(endT
ime - startTime);
            return {result.moves, nodesVisited,
duration.count()};
        }

        if (nodesVisited > MAX_NODES) {
            break;
        }

        maxDepth++;
    }

    return {{}, nodesVisited, 0};
}


Solution IterativeDeepening::dfs(
    const IDState& state,
    int depth,
    int maxDepth,
    std::unordered_set<std::string>& visited,
    int& nodesVisited,
    const Position& exitPos,
    bool isHorizontal
) {
    if (depth > maxDepth) return {{}, nodesVisited, 0};

    nodesVisited++;
    std::string stateStr = state.board.toString();

    if (state.board.isGoalState()) {
```

```
        return {state.moves, nodesVisited, 0};
    }

    if (visited.count(stateStr)) {
        return {{}, nodesVisited, 0};
    }
    visited.insert(stateStr);

    auto moves = state.board.getPossibleMoves();
    Car primary = state.board.getPrimaryCar();
    Position primPos = primary.getPosition();

    std::sort(moves.begin(), moves.end(),
    [&](const auto& a, const auto& b) {
        if ((a.first == 'P') != (b.first == 'P')) {
            return a.first == 'P';
        }

        if (a.first == 'P') {
            if (isHorizontal) {
                return (exitPos.col > primPos.col) ?
                       (a.second == "kanan") : (a.second
== "kiri");
            } else {
                return (exitPos.row > primPos.row) ?
                       (a.second == "bawah") : (a.second
== "atas");
            }
        }

        Position piecePos;
        for (int row = 0; row < state.board.getRows();
row++) {
            for (int col = 0; col < state.board.getCols();
col++) {
                if (state.board.getCell(row, col) ==
```

```
a.first) {
                    piecePos = {row, col};
                    break;
                }
            }
        }

        if (isPieceBlocking(state.board, a.first,
exitPos)) {
            if (isHorizontal) {
                return a.second == (piecePos.row >
primPos.row ? "bawah" : "atas");
            } else {
                return a.second == (piecePos.col >
primPos.col ? "kanan" : "kiri");
            }
        }

        return false;
    });


    for (const auto& [piece, direction] : moves) {
        if (depth > maxDepth/2 && piece != 'P' &&
            !isPieceBlocking(state.board, piece, exitPos))
{
            continue;
        }

        Board nextBoard = state.board;
        if (!nextBoard.makeMove(piece, direction))
continue;

        auto nextMoves = state.moves;
        nextMoves.push_back({piece, direction});
```

```
        IDState nextState{nextBoard, nextMoves, depth +
1};
        auto result = dfs(nextState, depth + 1, maxDepth,
visited, nodesVisited, exitPos, isHorizontal);

        if (!result.moves.empty()) return result;

        if (nodesVisited > MAX_NODES) break;
    }

    return {{}, nodesVisited, 0};
}


bool IterativeDeepening::isPieceBlocking(const Board&
board, char piece, const Position& exitPos) {
    Car primary = board.getPrimaryCar();
    Position primPos = primary.getPosition();
    int length = primary.getLength();

    if (primary.getIsHorizontal()) {
        if (primPos.row != exitPos.row) return false;
        int startCol = primPos.col + length;
        int endCol = exitPos.col;

        for (int col = startCol; col < endCol; col++) {
            if (board.getCell(primPos.row, col) == piece)
{
                return true;
            }
        }
    } else {
        if (primPos.col != exitPos.col) return false;
        int startRow = primPos.row + length;
        int endRow = exitPos.row;
```

```
        for (int row = startRow; row < endRow; row++) {
            if (board.getCell(row, primPos.col) == piece)
{
                return true;
            }
        }
    }

    return false;
}
```

IV.2.6 ucs.cpp

```
#include "ucs.hpp"
#include <queue>
#include <unordered_set>
#include <unordered_map>

struct UCSState {
    Board board;
    std::vector<std::pair<char, std::string>> moves;
    int cost;

    bool operator>(const UCSState& other) const {
        return cost > other.cost;
    }
};

Solution UCS::solve(const Board& initialBoard) {
    auto startTime =
std::chrono::high_resolution_clock::now();

    std::priority_queue<UCSState, std::vector<UCSState>,
std::greater<>> pq;
```

```
    std::unordered_map<std::string, int> bestCost;
    int nodesVisited = 0;

    std::string initialStr = initialBoard.toString();
    pq.push({initialBoard, {}, 0});
    bestCost[initialStr] = 0;

    while (!pq.empty()) {
        UCSState current = pq.top();
        pq.pop();
        nodesVisited++;

        std::string currentStr = current.board.toString();

        if (current.cost > bestCost[currentStr]) {
            continue;
        }

        if (current.board.isGoalState()) {
            auto endTime =
std::chrono::high_resolution_clock::now();
            auto duration =
std::chrono::duration_cast<std::chrono::milliseconds>(endT
ime - startTime);
            return {current.moves, nodesVisited,
duration.count()};
        }

        for (const auto& [piece, direction] :
current.board.getPossibleMoves()) {
            Board nextBoard = current.board;
            if (!nextBoard.makeMove(piece, direction)) {
                continue;
            }

            std::string nextStr = nextBoard.toString();
```

```
            int nextCost = current.cost + 1;

            if (bestCost.find(nextStr) != bestCost.end()
&&
                nextCost >= bestCost[nextStr]) {
                continue;
            }

            bestCost[nextStr] = nextCost;

            auto nextMoves = current.moves;
            nextMoves.push_back({piece, direction});
            pq.push({nextBoard, nextMoves, nextCost});
        }
    }

    return {{}, nodesVisited, 0};
}
```

IV.2.7 blocking_cars.cpp

```
#include "blocking_cars.hpp"
#include <set>
#include <queue>
#include <algorithm>

int BlockingCarsHeuristic::calculate(const Board& board) {
    const Car& primary = board.getPrimaryCar();
    Position pos = primary.getPosition();
    Position exitPos = board.getExitPosition();
    int blocking = 0;

    if (primary.getIsHorizontal()) {
        if (pos.row != exitPos.row) {
```

```
            return INT_MAX;
        }
        int startCol = pos.col + primary.getLength();
        int endCol = exitPos.col;
        for (int col = startCol; col < endCol; col++) {
            char piece = board.getCell(pos.row, col);
            if (piece != '.' && piece != 'P') blocking++;
        }
    } else {
        if (pos.col != exitPos.col) {
            return INT_MAX;
        }
        int startRow = pos.row + primary.getLength();
        int endRow = exitPos.row;
        for (int row = startRow; row < endRow; row++) {
            char piece = board.getCell(row, pos.col);
            if (piece != '.' && piece != 'P') blocking++;
        }
    }

    return blocking;
}

int BlockingCarsHeuristic::calculateReversed(const Board&
board) {
    const Car& primary = board.getPrimaryCar();
    Position pos = primary.getPosition();
    Position exitPos = board.getExitPosition();
    int blocking = 0;

    if (exitPos.row == -1) {
        if (primary.getIsHorizontal()) {
            return INT_MAX;
        }
        for (int row = 0; row < pos.row; row++) {
            char piece = board.getCell(row, pos.col);
```

```
            if (piece != '.' && piece != 'P') blocking++;
        }
        return blocking;
    }

    if (exitPos.col == -1) {
        if (!primary.getIsHorizontal()) {
            return INT_MAX;
        }
        for (int col = 0; col < pos.col; col++) {
            char piece = board.getCell(pos.row, col);
            if (piece != '.' && piece != 'P') blocking++;
        }
        return blocking;
    }

    if (primary.getIsHorizontal()) {
        for (int col = pos.col + primary.getLength(); col
< board.getCols(); col++) {
            char piece = board.getCell(pos.row, col);
            if (piece != '.' && piece != 'P') blocking++;
        }
    } else {
        for (int row = pos.row + primary.getLength(); row
< board.getRows(); row++) {
            char piece = board.getCell(row, pos.col);
            if (piece != '.' && piece != 'P') blocking++;
        }
    }

    return blocking;
}
```

IV.2.8 manhattan.cpp

```cpp
#include "manhattan.hpp"
#include <cstdlib>
#include <algorithm>

int ManhattanHeuristic::calculate(const Board& board) {
    Car primaryCar = board.getPrimaryCar();
    if (primaryCar.getLength() == 0) {
        return INT_MAX;
    }

    Position exitPos = board.getExitPosition();
    Position carPos = primaryCar.getPosition();

    if (primaryCar.getIsHorizontal()) {
        if (carPos.row != exitPos.row || carPos.col > exitPos.col) {
            return INT_MAX;
        }

        int endPos = carPos.col + primaryCar.getLength() - 1;
        return std::max(0, exitPos.col - endPos);
    }
    else {
        if (carPos.col != exitPos.col || carPos.row > exitPos.row) {
            return INT_MAX;
        }

        int endPos = carPos.row + primaryCar.getLength() - 1;
        return std::max(0, exitPos.row - endPos);
    }
}

int ManhattanHeuristic::manhattanDistance(const Position&
```

```
p1, const Position& p2) {
    return std::abs(p1.row - p2.row) + std::abs(p1.col -
p2.col);
}

int ManhattanHeuristic::calculateReversed(const Board&
board) {
    const Car& primary = board.getPrimaryCar();
    Position pos = primary.getPosition();
    Position exitPos = board.getExitPosition();

    if (primary.getIsHorizontal()) {
        return pos.col;
    } else {
        return std::abs(pos.row - exitPos.row);
    }
}
```

IV.2.9 main.cpp

```
#include <iostream>
#include <string>
#include <chrono>
#include <limits>
#include <filesystem>
#include <memory>
#include <iomanip>
#include <cctype>
#include <fstream>
#include "board/board.hpp"
#include "algorithms/solver.hpp"
#include "algorithms/ucs.hpp"
#include "algorithms/astar.hpp"
#include "algorithms/greedy.hpp"
```

```cpp
#include "algorithms/iter_deepening.hpp"
#include "heuristics/manhattan.hpp"
#include "heuristics/blocking_cars.hpp"
#include "utils/file_reader.hpp"
#include "utils/output_writer.hpp"

#ifdef _WIN32
    #include <windows.h>
    void enableConsoleFeatures() {
        HANDLE hOut = GetStdHandle(STD_OUTPUT_HANDLE);
        DWORD dwMode = 0;
        if (GetConsoleMode(hOut, &dwMode)) {
            dwMode |= ENABLE_VIRTUAL_TERMINAL_PROCESSING;
            SetConsoleMode(hOut, dwMode);
        }
        SetConsoleOutputCP(CP_UTF8);
        SetConsoleCP(CP_UTF8);
    }
#endif

namespace ansi {
    constexpr const char* reset  = "\033[0m";
    constexpr const char* bold   = "\033[1m";
    constexpr const char* cyan   = "\033[96m";
    constexpr const char* yellow = "\033[93m";
    constexpr const char* green  = "\033[92m";
    constexpr const char* red    = "\033[91m";
}

void printDivider(char fill = '=') {
    std::cout << std::string(60, fill) << "\n";
}

void printHeader() {
    using namespace ansi;
    printDivider();
```


std::cout << cyan << bold
<< "
\n"
<< "
\n"
<< "
\n"
<< "
\n"
<< "
\n"
<< "
\n"
<< "
\n"
<< "
\n"
<< "
\n"
<< "
\n"
<< "
\n"
<< "
\n"
<< "
\n"
<< "
\n"
<< "
S O L V E
R O O P
"
<< reset << "\n";
printDivider();
}
int promptChoice(const std::string& label, int min, int
max) {
std::cout << ansi::yellow << label << ansi::reset;
int choice;
while (!(std::cin >> choice) || choice < min || choice
> max) {
std::cin.clear();
std::cin.ignore(std::numeric_limits<std::streamsize>::max(
), '\n');
std::cout << ansi::red << "Input tidak valid.
Pilih "
<< min << "-" << max << ": " <<

```
ansi::reset;
    }
    return choice;
}

void solveOnce() {
    std::string filename;
    do {
        std::cout << "Masukkan path file konfigurasi
puzzle (.txt): ";
        std::cin  >> filename;
        if (!std::filesystem::exists(filename))
            std::cout << ansi::red << "File tidak
ditemukan. Coba lagi.\n" << ansi::reset;
    } while (!std::filesystem::exists(filename));

    std::cout << ansi::green << u8"\u2714  File terbaca
dengan sukses!\n" << ansi::reset;


std::cin.ignore(std::numeric_limits<std::streamsize>::max(
), '\n');
    std::string reportBase;
    std::cout << "Simpan laporan ke test/<nama>.txt
(kosong = tidak menyimpan): ";
    std::getline(std::cin, reportBase);
    std::unique_ptr<std::ofstream> report;
    if (!reportBase.empty()) {
        std::filesystem::create_directory("../test");
        std::string path = "../test/" + reportBase +
".txt";
        report = std::make_unique<std::ofstream>(path);
        if (!*report) {
            std::cerr << ansi::red << "Gagal membuat file
laporan: " << path << ansi::reset << "\n";
            report.reset();
```

```
        }
    }

    Board board = FileReader::readBoard(filename);

    std::cout << "\nPilih algoritma:\n";
    std::cout << " 1. UCS (Uniform Cost Search)\n";
    std::cout << " 2. Greedy Best-First Search\n";
    std::cout << " 3. A*\n";
    std::cout << " 4. Iterative Deepening\n";
    int algoChoice = promptChoice("Pilihan: ", 1, 4);

    std::string heuristic;
    if (algoChoice == 2 || algoChoice == 3) {
        std::cout << "\nPilih heuristic:\n";
        std::cout << " 1. Manhattan Distance\n";
        std::cout << " 2. Blocking Cars\n";
        int hChoice = promptChoice("Pilihan: ", 1, 2);
        heuristic = (hChoice == 1) ? "manhattan" :
"blocking";
        std::cout << ansi::green << u8"\u2714  Menggunakan
"
                  << (heuristic == "manhattan" ?
"Manhattan Distance" : "Blocking Cars")
                  << " heuristic\n" << ansi::reset;
    }

    std::unique_ptr<Solver> solver;
    switch (algoChoice) {
        case 1: solver = std::make_unique<UCS>(); break;
        case 2: solver =
std::make_unique<GreedyBestFirst>(heuristic); break;
        case 3: solver =
std::make_unique<AStar>(heuristic); break;
        case 4: solver =
std::make_unique<IterativeDeepening>(); break;
```

```
    }

    std::cout << "\n===  Mencari solusi...\n";
    if (report) *report << "=== Rush Hour Solver Report
===\n\n";

    auto t0 = std::chrono::high_resolution_clock::now();
    auto result = solver->solve(board);
    auto t1 = std::chrono::high_resolution_clock::now();
    long long ms  =
std::chrono::duration_cast<std::chrono::milliseconds>(t1 -
t0).count();

    std::cout << "\n";
    if (result.moves.empty()) {
        std::cout << ansi::red << u8"\u2718  Tidak
ditemukan solusi!\n" << ansi::reset;
        if (report) *report << "Tidak ditemukan
solusi!\n";
    } else {
        OutputWriter::printSolution(board, result);
        if (report) {
            OutputWriter::printSolution(board, result,
*report, false);
            *report << "Waktu eksekusi: " << ms << "
ms\n";
        }
        std::cout << ansi::cyan << "Waktu eksekusi: " <<
ms << " ms\n" << ansi::reset;
    }

    if (report) std::cout << ansi::green << u8"\u2714
Laporan tersimpan di " << report->tellp() << " byte.\n" <<
ansi::reset;
}
```

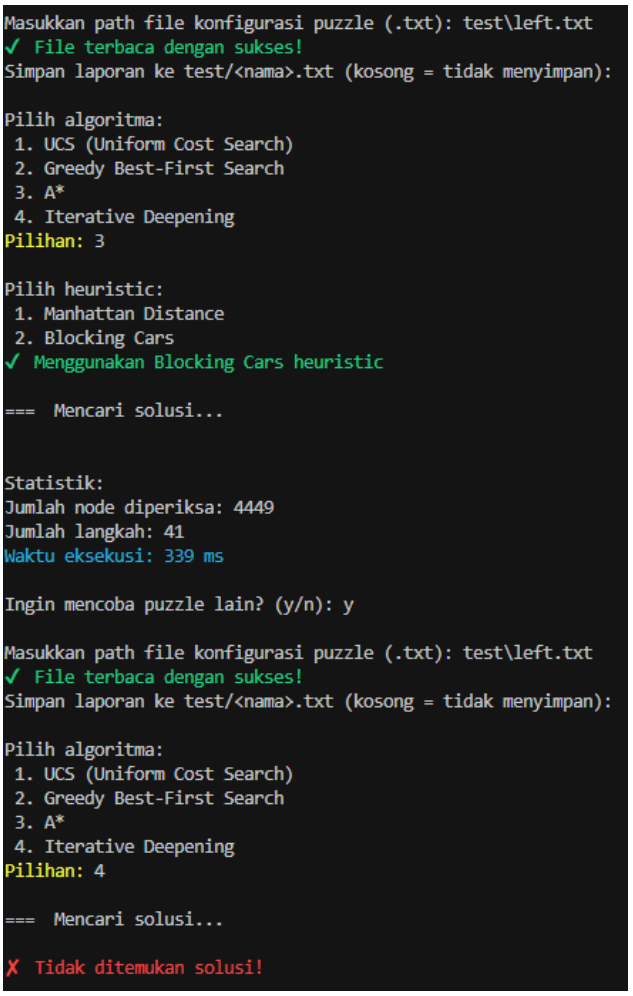
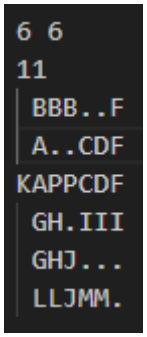
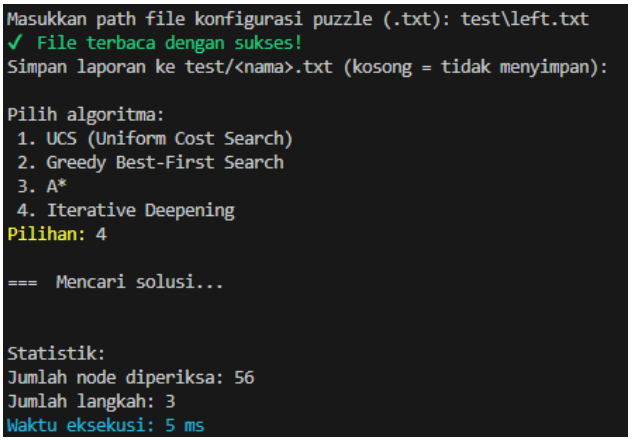
```
int main() {
#ifdef _WIN32
    enableConsoleFeatures();
#endif
    try {
        printHeader();
        char again;
        do {
            solveOnce();
            std::cout << "\nIngin mencoba puzzle lain?
(y/n): ";
            std::cin >> again;
            again =
static_cast<char>(std::tolower(again));
            std::cout << "\n";
        } while (again == 'y');

        std::cout << ansi::yellow << "Terima kasih telah
menggunakan\n" << ansi::reset;
        printDivider();

    } catch (const std::exception& e) {
        std::cerr << ansi::red << "Error: " << e.what() <<
ansi::reset << std::endl;
        return 1;
    }
    return 0;
}
```

BAB V

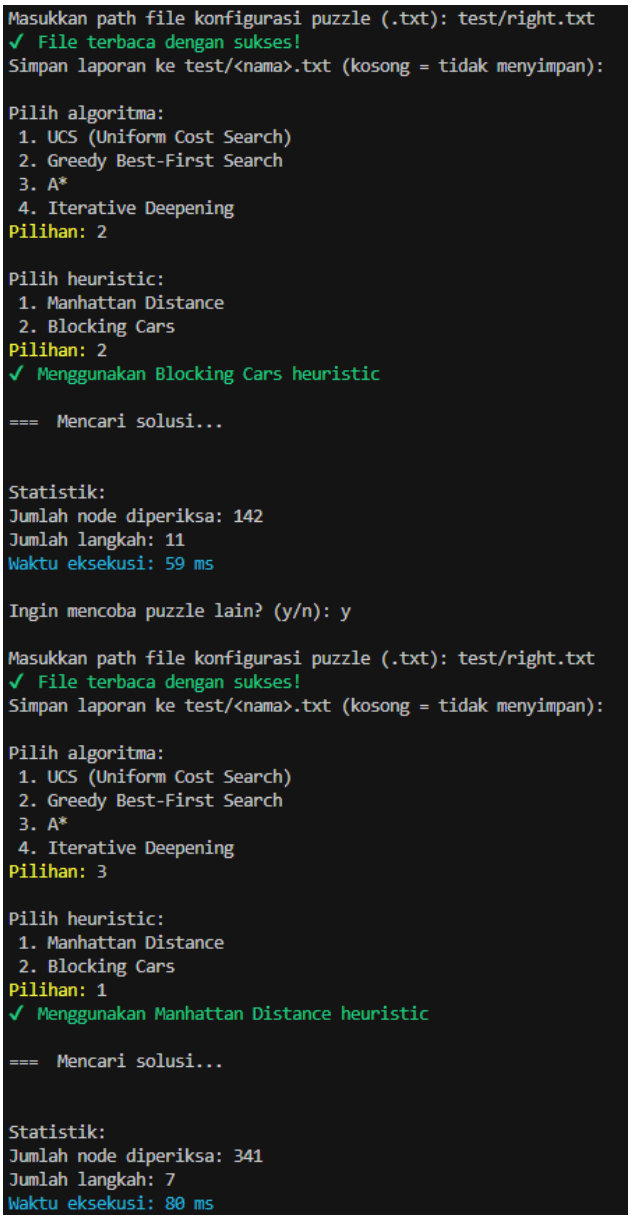
TESTING

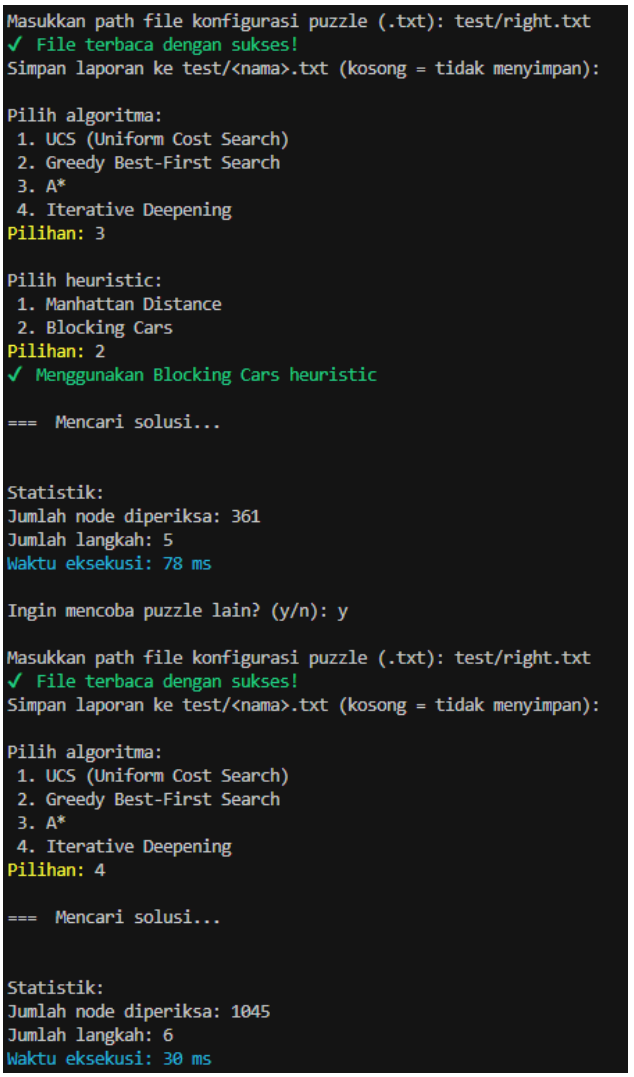
NO	Test Case		Penjelasan
	Input	Output	
1.	6 6 12 .AABB. CCDE.. .FDEGG KHFPPIJ HFL.IJ H.LMMJ	Masukkan path file konfigurasi puzzle (.txt): test\left.txt ✓ File terbaca dengan sukses! Simpan laporan ke test/<nama>.txt (kosong = tidak menyimpan): Pilih algoritma: 1. UCS (Uniform Cost Search) 2. Greedy Best-First Search 3. A* 4. Iterative Deepening Pilihan: 1 === Mencari solusi... Statistik: Jumlah node diperiksa: 5204 Jumlah langkah: 44 Waktu eksekusi: 420 ms Ingin mencoba puzzle lain? (y/n): y Masukkan path file konfigurasi puzzle (.txt): test\left.txt ✓ File terbaca dengan sukses! Simpan laporan ke test/<nama>.txt (kosong = tidak menyimpan): Pilih algoritma: 1. UCS (Uniform Cost Search) 2. Greedy Best-First Search 3. A* 4. Iterative Deepening Pilihan: 2 Pilih heuristic: 1. Manhattan Distance 2. Blocking Cars Pilihan: 1 ✓ Menggunakan Manhattan Distance heuristic === Mencari solusi... Statistik: Jumlah node diperiksa: 5123 Jumlah langkah: 329 Waktu eksekusi: 823 ms	Kasus Exit di Kiri untuk semua algoritma selain IDS berhasil, IDS gagal karena kompleks puzzle

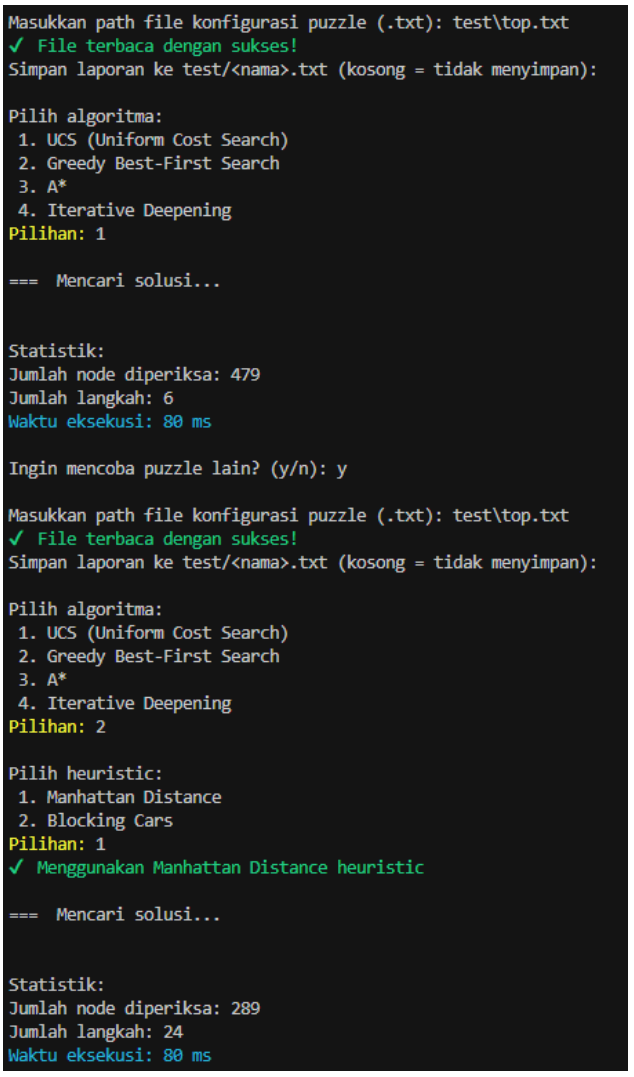
		Masukkan path file konfigurasi puzzle (.txt): test\left.txt ✓ File terbaca dengan sukses! Simpan laporan ke test/<nama>.txt (kosong = tidak menyimpan): Pilih algoritma: 1. UCS (Uniform Cost Search) 2. Greedy Best-First Search 3. A* 4. Iterative Deepening Pilihan: 2 Pilih heuristic: 1. Manhattan Distance 2. Blocking Cars Pilihan: 2 ✓ Menggunakan Blocking Cars heuristic === Mencari solusi... Statistik: Jumlah node diperiksa: 5123 Jumlah langkah: 329 Waktu eksekusi: 835 ms Ingin mencoba puzzle lain? (y/n): y Masukkan path file konfigurasi puzzle (.txt): test\left.txt ✓ File terbaca dengan sukses! Simpan laporan ke test/<nama>.txt (kosong = tidak menyimpan): Pilih algoritma: 1. UCS (Uniform Cost Search) 2. Greedy Best-First Search 3. A* 4. Iterative Deepening Pilihan: 3 Pilih heuristic: 1. Manhattan Distance 2. Blocking Cars Pilihan: 1 ✓ Menggunakan Manhattan Distance heuristic === Mencari solusi... Statistik: Jumlah node diperiksa: 4323 Jumlah langkah: 42 Waktu eksekusi: 346 ms	

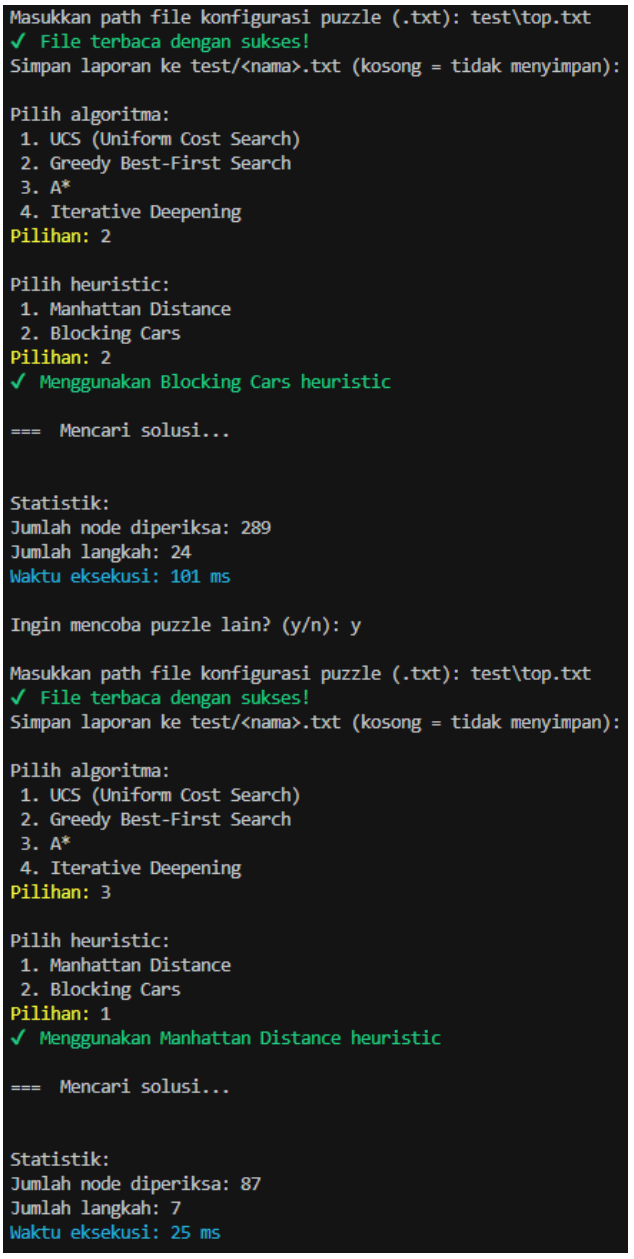
			
2.			Kasus Exit di kiri tidak kompleks sehingga IDS berhasil

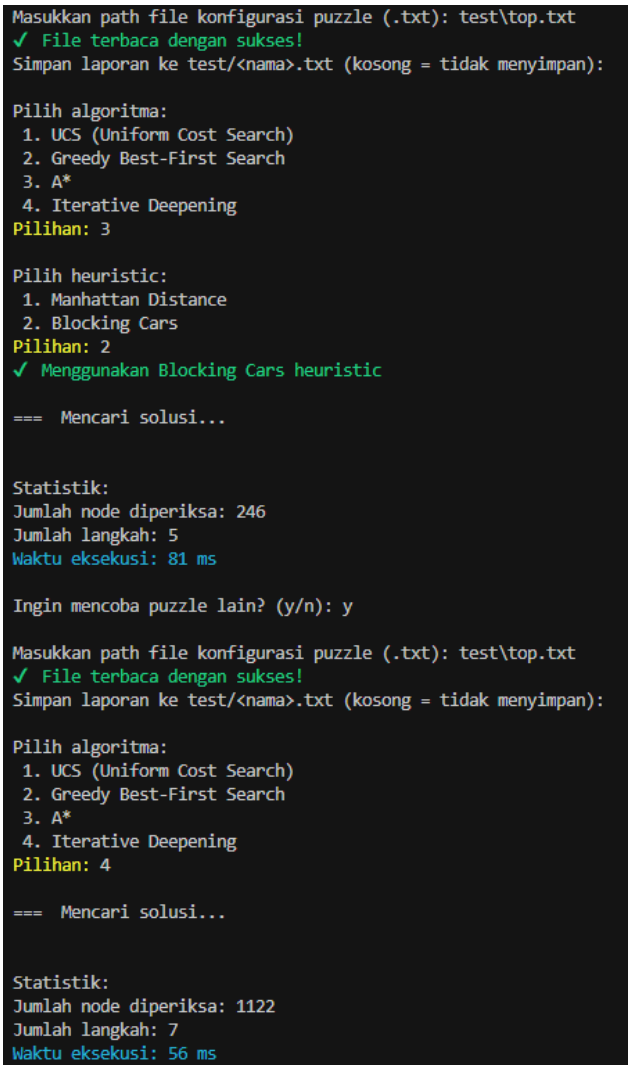
3.	6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	Masukkan path file konfigurasi puzzle (.txt): test/right.txt ✓ File terbaca dengan sukses! Simpan laporan ke test/<nama>.txt (kosong = tidak menyimpan): Pilih algoritma: 1. UCS (Uniform Cost Search) 2. Greedy Best-First Search 3. A* 4. Iterative Deepening Pilihan: 1 === Mencari solusi... Statistik: Jumlah node diperiksa: 586 Jumlah langkah: 7 Waktu eksekusi: 100 ms Ingin mencoba puzzle lain? (y/n): y Masukkan path file konfigurasi puzzle (.txt): test/right.txt ✓ File terbaca dengan sukses! Simpan laporan ke test/<nama>.txt (kosong = tidak menyimpan): 2 Pilih algoritma: 1. UCS (Uniform Cost Search) 2. Greedy Best-First Search 3. A* 4. Iterative Deepening Pilihan: 2 Pilih heuristic: 1. Manhattan Distance 2. Blocking Cars Pilihan: 1 ✓ Menggunakan Manhattan Distance heuristic === Mencari solusi... Statistik: Jumlah node diperiksa: 429 Jumlah langkah: 41 Waktu eksekusi: 41 ms ✓ Laporan tersimpan di 113 byte.	Kasus Exit di kanan

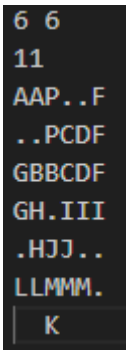
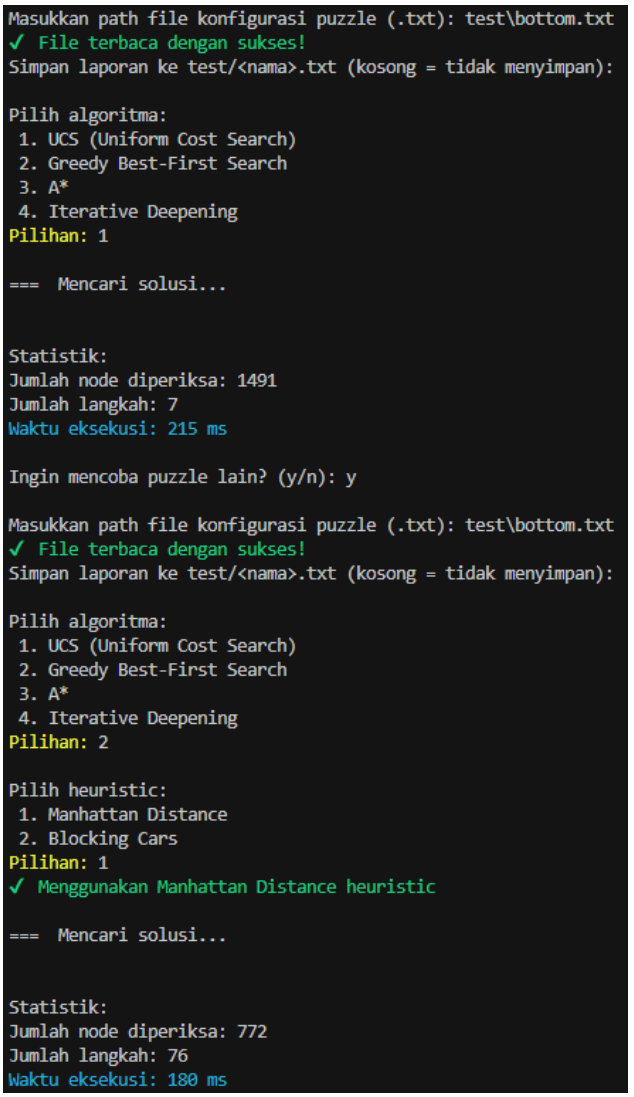
		 Masukkan path file konfigurasi puzzle (.txt): test/right.txt ✓ File terbaca dengan sukses! Simpan laporan ke test/<nama>.txt (kosong = tidak menyimpan): Pilih algoritma: 1. UCS (Uniform Cost Search) 2. Greedy Best-First Search 3. A* 4. Iterative Deepening Pilihan: 2 Pilih heuristic: 1. Manhattan Distance 2. Blocking Cars Pilihan: 2 ✓ Menggunakan Blocking Cars heuristic === Mencari solusi... Statistik: Jumlah node diperiksa: 142 Jumlah langkah: 11 Waktu eksekusi: 59 ms Ingin mencoba puzzle lain? (y/n): y Masukkan path file konfigurasi puzzle (.txt): test/right.txt ✓ File terbaca dengan sukses! Simpan laporan ke test/<nama>.txt (kosong = tidak menyimpan): Pilih algoritma: 1. UCS (Uniform Cost Search) 2. Greedy Best-First Search 3. A* 4. Iterative Deepening Pilihan: 3 Pilih heuristic: 1. Manhattan Distance 2. Blocking Cars Pilihan: 1 ✓ Menggunakan Manhattan Distance heuristic === Mencari solusi... Statistik: Jumlah node diperiksa: 341 Jumlah langkah: 7 Waktu eksekusi: 80 ms	

		 Masukkan path file konfigurasi puzzle (.txt): test/right.txt ✓ File terbaca dengan sukses! Simpan laporan ke test/<nama>.txt (kosong = tidak menyimpan): Pilih algoritma: 1. UCS (Uniform Cost Search) 2. Greedy Best-First Search 3. A* 4. Iterative Deepening Pilihan: 3 Pilih heuristic: 1. Manhattan Distance 2. Blocking Cars Pilihan: 2 ✓ Menggunakan Blocking Cars heuristic === Mencari solusi... Statistik: Jumlah node diperiksa: 361 Jumlah langkah: 5 Waktu eksekusi: 78 ms Ingin mencoba puzzle lain? (y/n): y Masukkan path file konfigurasi puzzle (.txt): test/right.txt ✓ File terbaca dengan sukses! Simpan laporan ke test/<nama>.txt (kosong = tidak menyimpan): Pilih algoritma: 1. UCS (Uniform Cost Search) 2. Greedy Best-First Search 3. A* 4. Iterative Deepening Pilihan: 4 === Mencari solusi... Statistik: Jumlah node diperiksa: 1045 Jumlah langkah: 6 Waktu eksekusi: 30 ms	

4.			Kasus Exit di atas

		 Masukkan path file konfigurasi puzzle (.txt): test\top.txt ✓ File terbaca dengan sukses! Simpan laporan ke test/<nama>.txt (kosong = tidak menyimpan): Pilih algoritma: 1. UCS (Uniform Cost Search) 2. Greedy Best-First Search 3. A* 4. Iterative Deepening Pilihan: 2 Pilih heuristic: 1. Manhattan Distance 2. Blocking Cars Pilihan: 2 ✓ Menggunakan Blocking Cars heuristic === Mencari solusi... Statistik: Jumlah node diperiksa: 289 Jumlah langkah: 24 Waktu eksekusi: 101 ms Ingin mencoba puzzle lain? (y/n): y Masukkan path file konfigurasi puzzle (.txt): test\top.txt ✓ File terbaca dengan sukses! Simpan laporan ke test/<nama>.txt (kosong = tidak menyimpan): Pilih algoritma: 1. UCS (Uniform Cost Search) 2. Greedy Best-First Search 3. A* 4. Iterative Deepening Pilihan: 3 Pilih heuristic: 1. Manhattan Distance 2. Blocking Cars Pilihan: 1 ✓ Menggunakan Manhattan Distance heuristic === Mencari solusi... Statistik: Jumlah node diperiksa: 87 Jumlah langkah: 7 Waktu eksekusi: 25 ms	

		 Masukkan path file konfigurasi puzzle (.txt): test\top.txt ✓ File terbaca dengan sukses! Simpan laporan ke test/<nama>.txt (kosong = tidak menyimpan): Pilih algoritma: 1. UCS (Uniform Cost Search) 2. Greedy Best-First Search 3. A* 4. Iterative Deepening Pilihan: 3 Pilih heuristic: 1. Manhattan Distance 2. Blocking Cars Pilihan: 2 ✓ Menggunakan Blocking Cars heuristic === Mencari solusi... Statistik: Jumlah node diperiksa: 246 Jumlah langkah: 5 Waktu eksekusi: 81 ms Ingin mencoba puzzle lain? (y/n): y Masukkan path file konfigurasi puzzle (.txt): test\top.txt ✓ File terbaca dengan sukses! Simpan laporan ke test/<nama>.txt (kosong = tidak menyimpan): Pilih algoritma: 1. UCS (Uniform Cost Search) 2. Greedy Best-First Search 3. A* 4. Iterative Deepening Pilihan: 4 === Mencari solusi... Statistik: Jumlah node diperiksa: 1122 Jumlah langkah: 7 Waktu eksekusi: 56 ms	

5.			Kasus Exit di bawah

		`` Masukkan path file konfigurasi puzzle (.txt): test\bottom.txt ✓ File terbaca dengan sukses! Simpan laporan ke test/<nama>.txt (kosong = tidak menyimpan): Pilih algoritma: 1. UCS (Uniform Cost Search) 2. Greedy Best-First Search 3. A* 4. Iterative Deepening Pilihan: 2 Pilih heuristic: 1. Manhattan Distance 2. Blocking Cars Pilihan: 2 ✓ Menggunakan Blocking Cars heuristic === Mencari solusi... Statistik: Jumlah node diperiksa: 45 Jumlah langkah: 8 Waktu eksekusi: 22 ms Ingin mencoba puzzle lain? (y/n): y Masukkan path file konfigurasi puzzle (.txt): test\bottom.txt ✓ File terbaca dengan sukses! Simpan laporan ke test/<nama>.txt (kosong = tidak menyimpan): Pilih algoritma: 1. UCS (Uniform Cost Search) 2. Greedy Best-First Search 3. A* 4. Iterative Deepening Pilihan: 3 Pilih heuristic: 1. Manhattan Distance 2. Blocking Cars Pilihan: 1 ✓ Menggunakan Manhattan Distance heuristic === Mencari solusi... Statistik: Jumlah node diperiksa: 110 Jumlah langkah: 7 Waktu eksekusi: 45 ms ```	

		Masukkan path file konfigurasi puzzle (.txt): test\bottom.txt ✓ File terbaca dengan sukses! Simpan laporan ke test/<nama>.txt (kosong = tidak menyimpan): Pilih algoritma: 1. UCS (Uniform Cost Search) 2. Greedy Best-First Search 3. A* 4. Iterative Deepening Pilihan: 3 Pilih heuristic: 1. Manhattan Distance 2. Blocking Cars Pilihan: 2 ✓ Menggunakan Blocking Cars heuristic === Mencari solusi... Statistik: Jumlah node diperiksa: 387 Jumlah langkah: 5 Waktu eksekusi: 87 ms Ingin mencoba puzzle lain? (y/n): y Masukkan path file konfigurasi puzzle (.txt): test\bottom.txt ✓ File terbaca dengan sukses! Simpan laporan ke test/<nama>.txt (kosong = tidak menyimpan): Pilih algoritma: 1. UCS (Uniform Cost Search) 2. Greedy Best-First Search 3. A* 4. Iterative Deepening Pilihan: 4 === Mencari solusi... Statistik: Jumlah node diperiksa: 7324 Jumlah langkah: 9 Waktu eksekusi: 222 ms	
6.	6 6 11 AAB... ..BCD. GPPCD.K GH.III GHJ... LLJMM.	Masukkan path file konfigurasi puzzle (.txt): test\false.txt ✓ File terbaca dengan sukses! Simpan laporan ke test/<nama>.txt (kosong = tidak menyimpan): Error: Jumlah piece tidak cocok header	Kasus jumlah piece tidak sesuai header

BAB VI

LAMPIRAN

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 sendiri	✓	

DAFTAR REFERENSI

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/stima24-25.htm

https://www.youtube.com/shorts/zgR7uI8r360

https://www.youtube.com/watch?v=zfvWi_RxoGw