Penyelesaian *Puzzle* Rush Hour Menggunakan Algoritma *Pathfinding*

Laporan Tugas Kecil 3
IF2211 - Strategi Algoritma

Disusun Oleh:

12821046 - Fardhan Indrayesa

12823024 - Azadi Azhrah

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2025

DAFTAR ISI

1. Penjelasan Algoritma

a. *Uniform Cost Search (UCS)*

Penggunaan algoritma ini memanfaatkan biaya (*cost*) dari titik awal ke suatu titik yang lain. Tujuan dari algoritma ini adalah menemukan jalur yang memiliki biaya sesedikit mungkin dari titik awal hingga ke titik *goal*. Berikut merupakan penjelasan dari algoritma *uniform cost search* (UCS) dalam penyelesaian permainan *Rush Hour*.

1. UCS dimulai dari *state* awal, dengan menambahkan state tersebut ke *priority queue* yang memiliki biaya 0 karena belum ada langkah yang diambil.
2. Ambil *node state* dengan biaya terkecil, lalu hapus *state* tersebut pada *priority queue*. *Node state* ini diekspansi sehingga menghasilkan *neighbour* yang mungkin akan dikunjungi.
3. Untuk setiap *neighbor* yang diekspansi, algoritma ini menghitung total biaya (jumlah gerakan) dari *node* awal hingga *node* saat ini melewati *node* tetangga yang sudah dikunjungi.
4. Setelah mengekspansi sebuah *node state*, algoritma ini mengecek apakah sudah mencapai *node goal* (kendaraan utama sudah keluar). Jika *goal node* sudah dicapai, algoritma ini berhenti, mengembalikan nilai biaya total dalam mencapai *node* ini dan mengembalikan jalur yang diambil untuk mencapai *node* tujuan.
5. Proses ini diulang hingga *priority queue* habis atau *goal node* sudah dicapai.

b. *Greedy Best First Search (GBFS)*

Algoritma *Greedy Best First Search* (GBFS) memanfaatkan fungsi evaluasi f(n) (heuristik) dalam setiap *node* untuk mengestimasi biaya dari suatu *node* ke *node* tujuan. Tujuan dari algoritma ini adalah menemukan jalur yang memiliki estimasi biaya yang cukup menjanjikan dari titik awal ke titik *goal*. Berikut merupakan penjelasan dari algoritma *Greedy Best First Search* (GBFS) dalam penyelesaian permainan *Rush Hour*.

1. Dimulai dari *node state* awal. Tambahkan *node state* ini ke *priority queue*.
2. Evaluasi seluruh *node* tetangga dari *node* saat ini. Estimasi suatu biaya berdasarkan fungsi heuristik, biasanya direpresentasikan sebagai jarak ke titik tujuan (jumlah kotak yang dihalangi kendaraan lain).
3. Dari *priority queue*, pilih *node* yang memiliki nilai heuristik yang paling rendah (*node* yang dianggap dekat dengan *node* tujuan).
4. Jika *node* yang terpilih adalah *node goal*, hentikan pencarian.
5. Jika bukan *node goal*, ulangi langkah 2 hingga 4 untuk *node* selanjutnya sampai *node goal* dicapai atau sampai *queue* habis.

c. A*

Algoritma A* merupakan algoritma yang digunakan dalam pencarian jalur terpendek yang dimulai dari *node* awal hingga *node* tujuan menggunakan graf berbobot. Algoritma ini mengandung dua nilai *cost*, yaitu g(n) (biaya sejauh ini) dan h(n) (estimasi biaya dari *goal* ke titik saat ini). Kedua *cost* ini dijumlahkan untuk disimpan

dalam *priority queue*. Berikut merupakan penjelasan dari algoritma A* dalam penyelesaian permainan Rush Hour.

1. Dimulai dari node state awal. Tambahkan node state ini ke priority queue.
2. Evaluasi node tetangga dengan nilai $f(n) = g(n) + h(n)$.
3. Dari priority queue, ambil tetangga dengan biaya f(n) terkecil.
4. Jika *node* yang dipilih adalah *goal node*, maka pencarian dihentikan.
5. Apabila bukan *goal node*, ulangi langkah 2 sampai 4 hingga menemukan *goal node* atau *queue*-nya habis.

d. Pseudocode

Berikut merupakan pseudocode dari algoritma UCS, GBFS, dan A*.

```
PROGRAM RushHourSolver

BEGIN
    READ filename
    READ algorithm

    CALL ReadPuzzleFile(filename) → puzzle
    CALL BuildBoard(puzzle.layout) → board
    CALL FindExit(board) → exit

    CALL GetVehicle('P', puzzle.layout) → pVehicle

    IF IsBlocked(board, pVehicle, puzzle, exit) THEN
        PRINT "Tidak ada solusi: P terhalang kendaraan
sejajar"
        EXIT
    ENDIF

    SET solver TO RushHourSolver(board, puzzle.layout,
exit)

    CASE algorithm OF
        "ucs":   CALL solver.UCS() → solution
        "gbfs":  CALL solver.GBFS() → solution
        "astar": CALL solver.AStar() → solution
        OTHERS:  PRINT "Algoritma tidak valid", EXIT
    ENDCASE

    CALL PrintSolution(solution)
END


FUNCTION UCS()

    INITIALIZE priority_queue pq
    INITIALIZE map visited
```

```
    INITIALIZE map parent
    INITIALIZE map gScore

    SET key TO string of initial board
    SET gScore[key] TO 0
    INSERT Node(initial board, cost 0, priority 0) TO pq

    WHILE pq is not empty DO
        REMOVE top node FROM pq → current
        SET key TO string of current.board

        IF IsGoal(current.board) THEN
            RETURN BuildPath(key, parent)
        ENDIF

        MARK key AS visited

        CALL GetSuccessors(current.board) → successors

        FOR each successor IN successors DO
            SET skey TO string of successor.board
            SET tentative_cost TO gScore[key] + 1

            IF skey not in gScore OR tentative_cost <
gScore[skey] THEN
                SET gScore[skey] TO tentative_cost
                SET parent[skey] TO key
                INSERT Node(successor, cost =
tentative_cost, priority = tentative_cost) TO pq
            ENDIF
        ENDFOR
    ENDWHILE

    RETURN empty
ENDFUNCTION


FUNCTION GBFS()

    INITIALIZE priority_queue pq
    INITIALIZE map visited
    INITIALIZE map parent

    SET key TO string of initial board
    INSERT Node(initial board, cost 0, priority =
Heuristic(initial board)) TO pq

    WHILE pq is not empty DO
        REMOVE top node FROM pq → current
        SET key TO string of current.board
```

```
        IF IsGoal(current.board) THEN
            RETURN BuildPath(key, parent)
        ENDIF

        IF key IN visited THEN
            CONTINUE
        ENDIF

        MARK key AS visited

        CALL GetSuccessors(current.board) → successors

        FOR each successor IN successors DO
            SET skey TO string of successor.board
            IF skey not in visited THEN
                SET h TO Heuristic(successor.board)
                SET parent[skey] TO key
                INSERT Node(successor, cost = h,
priority = h) TO pq
            ENDIF
        ENDFOR
    ENDWHILE

    RETURN empty
ENDFUNCTION


FUNCTION AStar()

    INITIALIZE priority_queue pq
    INITIALIZE map visited
    INITIALIZE map parent
    INITIALIZE map gScore

    SET key TO string of initial board
    SET gScore[key] TO 0
    SET h TO Heuristic(initial board)
    INSERT Node(initial board, cost 0, priority = h) TO
pq

    WHILE pq is not empty DO
        REMOVE top node FROM pq → current
        SET key TO string of current.board

        IF IsGoal(current.board) THEN
            RETURN BuildPath(key, parent)
        ENDIF

        IF key IN visited THEN
```

```
            CONTINUE
        ENDIF

        MARK key AS visited

        CALL GetSuccessors(current.board) → successors

        FOR each successor IN successors DO
            SET skey TO string of successor.board
            SET tentative_g TO gScore[key] + 1

            IF skey not in gScore OR tentative_g <
gScore[skey] THEN
                SET gScore[skey] TO tentative_g
                SET parent[skey] TO key
                SET f TO tentative_g +
Heuristic(successor.board)
                INSERT Node(successor, cost =
tentative_g, priority = f) TO pq
            ENDIF
        ENDFOR
    ENDWHILE

    RETURN empty
ENDFUNCTION


FUNCTION IsGoal(board)
    FOR EACH cell IN board DO
        IF cell = 'P' THEN
            RETURN FALSE
        ENDIF
    ENDFOR
    RETURN TRUE
ENDFUNCTION


FUNCTION IsBlocked(board, pVehicle, puzzle, exit)

    COMPUTE dirx ← SIGN(exit.x - pVehicle.x)
    COMPUTE diry ← SIGN(exit.y - pVehicle.y)

    SET cx ← pVehicle.x + dirx
    SET cy ← pVehicle.y + diry

    WHILE cx and cy in bounds DO
        SET c ← board[cx][cy]
        IF c ≠ '.' AND c ≠ 'P' THEN
            CALL GetVehicle(c, puzzle.layout) → v
```

```
            IF v.horizontal = pVehicle.horizontal THEN
                RETURN TRUE
            ENDIF
        ENDIF
        SET cx ← cx + dirx
        SET cy ← cy + diry
    ENDWHILE

    RETURN FALSE
ENDFUNCTION


FUNCTION PrintSolution(solution)
    FOR i FROM 0 TO LENGTH(solution) - 1 DO
        PRINT "Langkah ke-", i, ": ", solution[i].id,
"-", solution[i].dir
        CALL PrintBoard(solution[i].board)
    ENDFOR
ENDFUNCTION
```

2. Implementasi

a. Source Code

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <sstream>
#include <map>
#include <queue>
#include <unordered_map>
#include <unordered_set>
#include <algorithm>
#include <chrono>

using namespace std;
using namespace chrono;

struct Vehicle {
    char id;
    int x, y; // baris, kolom
    bool horizontal;
    int length;
};

struct Puzzle {
    vector<Vehicle> layout;
};

struct State {
    map<char, int> positions;
```

```
};

struct KOut {
    int x, y;
};

struct PVhc {
    int x, y;
    bool horizontal;
    int length;
};

PVhc pVhc;
int w, h;
KOut kOut;
int node = 0;

vector<string> split(const string& str, char delim) {
    vector<string> splitted;
    istringstream ss(str);
    string token;
    while (getline(ss, token, delim)) {
        splitted.push_back(token);
    }
    return splitted;
}

void GetInit(vector<string>& vec, int& h, int& w) {
    bool kFound = false;
    if (vec.size() > h) {
        for (int i = 0; i < vec.size(); ++i) {
            for (int j = 0; j < vec[0].size(); ++j) {
                if (vec[i][j] == 'K') {
                    vec.erase(vec.begin() + i);  // hapus baris yang
mengandung K
                    kFound = true;
                    break;
                }
            }
            if (kFound) break;
        }
    } else {
        for (int i = 0; i < vec.size(); ++i) {
            if (vec[i].size() > w) {
                bool k = false;
                bool ws = false;
                char toRemove;
                for (int j = 0; j < vec[0].size(); ++j) {
                    if (vec[i][j] == ' ') {
                        ws = true;
                        toRemove = ' ';
                    } else if (vec[i][j] == 'K') {
                        k = true;
                        toRemove = 'K';
                    }
                }
                if (ws || k) {
                    vec[i].erase(remove(vec[i].begin(),
vec[i].end(), toRemove), vec[i].end());
                }
```

```
            }
        }
    }
}

void ReadFile(ifstream& file, int& w, int& h, int& nPieces,
vector<string>& initial) {
    string line;
    int i = 0;

    while (getline(file, line)) {
        if (!line.empty() && line.back() == '\r') {
            line.pop_back();
        }
        vector<string> readLine = split(line, ' ');

        if (i == 0 && readLine.size() >= 2) {  // ukuran board
            h = stoi(readLine[0]); // height
            w = stoi(readLine[1]); // width
        } else if (i == 1 && readLine.size() >= 1) {
            nPieces = stoi(readLine[0]);  // banyak kendaraan
        } else {
            for (int j = 0; j < line.size(); ++j) {
                if (line[j] == 'K') {
                    kOut.x = i - 2;
                    kOut.y = j;
                }
            }
            initial.push_back(line);
        }
        ++i;
    }
    GetInit(initial, h, w);
    file.close();
}

void GetInitLayout(vector<string>& initial, int& h, int& w, Puzzle&
puzzle) {
    map<char, bool> vIn;
    for (int i = 0; i < h; ++i) {
        for (int j = 0; j < w; ++j) {
            if (initial[i][j] == '.') {
                continue;
            }

            if (!vIn[initial[i][j]]) {
                bool isHorizontal = false;

                // cek horizontal/vertikal
                if ((j < w - 1) && (initial[i][j+1] ==
initial[i][j])) {
                    isHorizontal = true;
                } else if ((i < h - 1) && (initial[i+1][j] ==
initial[i][j])) {
                    isHorizontal = false;
                }

                puzzle.layout.emplace_back(Vehicle{initial[i][j], i,
j, isHorizontal, 1});
                vIn[initial[i][j]] = true;
```

```
                if (initial[i][j] == 'P') {
                    pVhc = PVhc{i, j, isHorizontal, 1};
                }
            } else {
                for (auto& layout: puzzle.layout) {
                    if (layout.id == initial[i][j]) {
                        if ((layout.horizontal && layout.x == i) ||
(!layout.horizontal && layout.y == j)) {
                            layout.length = layout.length + 1;
                            if (initial[i][j] == 'P') {
                                pVhc.length = pVhc.length + 1;
                            }
                        }
                    }
                }
            }
        }
    }
}

vector<vector<char>> BuildBoard(const Puzzle& puzzle, const State&
state, int h, int w) {
    vector<vector<char>> board(h, vector<char>(w, '.'));

    for (const auto& vhc: puzzle.layout) {
        int pos = state.positions.at(vhc.id);
        int x = vhc.horizontal ? vhc.x : pos;
        int y = vhc.horizontal ? pos : vhc.y;

        for (int i = 0; i < vhc.length; ++i) {
            if (vhc.horizontal) {
                board[x][y+i] = vhc.id;
            } else {
                board[x+i][y] = vhc.id;
            }
        }
    }
    return board;
}

struct Successor {
    vector<vector<char>> board;
    char id;
    string dir;
    int cost;
};

// class get successors
class BoardManipulator {
public:
    vector<vector<char>> board;
    vector<Vehicle> vehicles;
    int height, width;

    BoardManipulator(const vector<vector<char>>& initialBoard, const
vector<Vehicle>& initialVehicles)
        : board(initialBoard), vehicles(initialVehicles) {
            height = board.size();
            width = board[0].size();
    }
```

```
    void UpdateVehiclesFromBoard() {
        for (auto& v : vehicles) {
            bool found = false;
            for (int i = 0; i < height && !found; ++i) {
                for (int j = 0; j < width && !found; ++j) {
                    if (board[i][j] == v.id) {
                        v.x = i;
                        v.y = j;
                        found = true;
                    }
                }
            }
        }
    }

    vector<vector<char>> MoveRight(const Vehicle& vhc) {
        auto newBoard = board;
        int x = vhc.x;
        int y = vhc.y;
        int rightEdge = y + vhc.length;

        while (rightEdge < width && newBoard[x][rightEdge] == '.') {
            for (int i = 0; i < vhc.length; ++i) {
                newBoard[x][y + i] = '.';
            }
            y++;
            for (int i = 0; i < vhc.length; ++i) {
                newBoard[x][y + i] = vhc.id;
            }
            rightEdge++;
        }

        if (vhc.id == 'P' && vhc.horizontal && kOut.y >= width && x
== kOut.x) {
            bool clear = true;
            for (int cy = y + vhc.length; cy < width; ++cy) {
                if (newBoard[x][cy] != '.') {
                    clear = false;
                    break;
                }
            }
            if (clear) {
                for (int i = 0; i < vhc.length; ++i) {
                    if (y + i < width) {
                        newBoard[x][y + i] = '.';
                    }
                }
            }
        }
        return newBoard;
    }

    vector<vector<char>> MoveLeft(const Vehicle& vhc) {
        auto newBoard = board;
        int x = vhc.x;
        int y = vhc.y;

        while (y > 0 && newBoard[x][y - 1] == '.') {
            for (int i = 0; i < vhc.length; ++i) {
```

```
                newBoard[x][y + i] = '.';
            }
            y--;
            for (int i = 0; i < vhc.length; ++i) {
                newBoard[x][y + i] = vhc.id;
            }
        }

        if (vhc.id == 'P' && vhc.horizontal && kOut.y <= 0 && x ==
kOut.x) {
            bool clear = true;
            for (int cy = y - 1; cy >= 0; --cy) {
                if (newBoard[x][cy] != '.') {
                    clear = false;
                    break;
                }
            }
            if (clear) {
                for (int i = 0; i < vhc.length; ++i) {
                    if (y + i < width) {
                        newBoard[x][y + i] = '.';
                    }
                }
            }
        }

        return newBoard;
    }

    vector<vector<char>> MoveDown(const Vehicle& vhc) {
        auto newBoard = board;
        int x = vhc.x;
        int y = vhc.y;
        int bottomEdge = x + vhc.length;

        while (bottomEdge < height && newBoard[bottomEdge][y] ==
'.') {
            for (int i = 0; i < vhc.length; ++i) {
                newBoard[x + i][y] = '.';
            }
            x++;
            for (int i = 0; i < vhc.length; ++i) {
                newBoard[x + i][y] = vhc.id;
            }
            bottomEdge++;
        }

        if (vhc.id == 'P' && !vhc.horizontal && kOut.x >= height &&
y == kOut.y) {
            bool clear = true;
            for (int cx = x + vhc.length; cx < height; ++cx) {
                if (newBoard[cx][y] != '.') {
                    clear = false;
                    break;
                }
            }
            if (clear) {
                for (int i = 0; i < vhc.length; ++i) {
                    if (x + i < height) {
                        newBoard[x + i][y] = '.';
```

```
                    }
                }
            }
        }
        return newBoard;
    }

    vector<vector<char>> MoveUp(const Vehicle& vhc) {
        auto newBoard = board;
        int x = vhc.x;
        int y = vhc.y;

        while (x > 0 && newBoard[x - 1][y] == '.') {
            for (int i = 0; i < vhc.length; ++i) {
                newBoard[x + i][y] = '.';
            }
            x--;
            for (int i = 0; i < vhc.length; ++i) {
                newBoard[x + i][y] = vhc.id;
            }
        }

        if (vhc.id == 'P' && !vhc.horizontal && kOut.x <= 0 && y ==
kOut.y) {
            bool clear = true;
            for (int cx = x - 1; cx >= 0; --cx) {
                if (newBoard[cx][y] != '.') {
                    clear = false;
                    break;
                }
            }
            if (clear) {
                for (int i = 0; i < vhc.length; ++i) {
                    if (x + i < height) {
                        newBoard[x + i][y] = '.';
                    }
                }
            }
        }
        return newBoard;
    }

    vector<Successor> GetSuccessors() {
        vector<Successor> successors;

        for (const auto& vhc: vehicles) {
            if (vhc.horizontal) {
                auto rightBoard = MoveRight(vhc);
                if (rightBoard != board) {
                    successors.push_back({rightBoard, vhc.id,
"kanan"});
                }

                auto leftBoard = MoveLeft(vhc);
                if (leftBoard != board) {
                    successors.push_back({leftBoard, vhc.id,
"kiri"});
                }
            } else {
                auto downBoard = MoveDown(vhc);
```

```
                if (downBoard != board) {
                    successors.push_back({downBoard, vhc.id,
"bawah"});
                }

                auto upBoard = MoveUp(vhc);
                if (upBoard != board) {
                    successors.push_back({upBoard, vhc.id, "atas"});
                }
            }
        }
        return successors;
    }
};

void PrintBoard(Successor& s, ostream& out, bool useColor = true) {
    vector<vector<char>> boardState = s.board;

    auto colorWrap = [&](const string& code, char c) -> string {
        if (!useColor) return string(1, c);
        return code + c + "\033[0m";
    };

    auto printChar = [&](int j, int k) {
        char current = boardState[j][k];
        if (current == 'P') {
            out << colorWrap("\033[31m", 'P');
        } else if (current == s.id) {
            out << colorWrap("\033[34m", current);
        } else {
            out << current;
        }
    };

    if (pVhc.horizontal) {
        if (kOut.y == 0) {  // exit di kiri
            for (int j = 0; j < boardState.size(); ++j) {
                out << (j == kOut.x ? colorWrap("\033[32m", 'K') : "
");
                for (int k = 0; k < boardState[0].size(); ++k) {
                    printChar(j, k);
                }
                out << endl;
            }
        } else {  // exit di kanan
            for (int j = 0; j < boardState.size(); ++j) {
                for (int k = 0; k < boardState[0].size(); ++k) {
                    printChar(j, k);
                }
                out << (j == kOut.x ? colorWrap("\033[32m", 'K') :
"");
                out << endl;
            }
        }
    } else {
        if (kOut.x == 0) { // exit di atas
            for (int j = 0; j < boardState.size()+1; ++j) {
                for (int k = 0; k < boardState[0].size(); ++k) {
                    if (j == 0) {
                        out << (k == kOut.y ? colorWrap("\033[32m",
```

```
'K') : " ");
                    } else {
                        printChar(j - 1, k);
                    }
                }
                out << endl;
            }
        } else {  // exit di bawah
            for (int j = 0; j < boardState.size()+1; ++j) {
                for (int k = 0; k < boardState[0].size(); ++k) {
                    if (j == h) {
                        out << (k == kOut.y ? colorWrap("\033[32m",
'K') : " ");
                    } else {
                        printChar(j, k);
                    }
                }
                out << endl;
            }
        }
    }
}

class RushHourSolver {
    private:
        vector<vector<char>> initialBoard;
        vector<Vehicle> initialVehicles;

    public:
        RushHourSolver(const vector<vector<char>>& board, const
vector<Vehicle>& vehicles)
            : initialBoard(board), initialVehicles(vehicles) {}

        string boardToString(const vector<vector<char>>& board) {
            string s;
            for (const auto& row: board) {
                for (char ch: row) s += ch;
            }
            return s;
        }

        // Cek goal
        bool IsGoal(const vector<vector<char>>& board) {
            bool PFound = false;
            int px = -1, py = -1;

            for (int i = 0; i < board.size(); ++i) {
                for (int j = 0; j < board[0].size(); ++j) {
                    if (board[i][j] == 'P') {
                        PFound = true;
                        px = i;  // 2
                        py = j;  // 1
                        break;
                    }
                }
                if (PFound) break;
            }

            if (!PFound) return true;
            else return false;
```

```
        }

        // g(n): banyak gerakan
        // h(n): banyak blok yang menghalangi P untuk keluar
        int heuristic(const vector<vector<char>>& board) {
            int px = -1, py = -1;

            for (int i = 0; i < h; ++i) {
                for (int j = 0; j < w; ++j) {
                    if (board[i][j] == 'P') {
                        px = i;
                        py = j;
                        break;
                    }
                }
                if (px != -1) break;
            }

            int dx = kOut.x - px;
            int dy = kOut.y - py;
            int dirx = (dx != 0) ? dx / abs(dx) : 0;
            int diry = (dy != 0) ? dy / abs(dy) : 0;

            int cx = px + dirx, cy = py + diry;
            int count = 0;

            while (cx >= 0 && cx < board.size() && cy >= 0 && cy <
board[0].size()) {
                if (board[cx][cy] != '.' && board[cx][cy] != 'P')
count++;
                cx += dirx;
                cy += diry;
            }

            return count;
        }

        // buat path untuk node dari root ke goal
        vector<Successor> BuildPath(const string& goalKey,
            unordered_map<string, string>& parent,
unordered_map<string,
            Successor>& boardStorage) {

            vector<Successor> path;
            string k = goalKey;
            while (boardStorage.count(k)) {
                path.push_back(boardStorage[k]);
                if (!parent.count(k)) break;
                k = parent[k];
            }
            reverse(path.begin(), path.end());
            return path;
        }

        struct Node {
            Successor board;
            int cost;

            bool operator>(const Node& other) const {
                if (cost != other.cost) {
```

```
                    return cost > other.cost;
                }
                // jika cost sama, dahulukan state yang menggerakkan
P
                if (board.id == 'P' && other.board.id != 'P') return
false;
                if (board.id != 'P' && other.board.id == 'P') return
true;

                return false;
            }
        };

        // Algoritma UCS
        vector<Successor> UCS() {
            priority_queue<Node, vector<Node>, greater<Node>> pq;
            unordered_set<string> visited;
            unordered_map<string, string> parent;
            unordered_map<string, Successor> boardStorage;

            Successor initBoard = Successor{initialBoard, '-', ""};
            string startKey = boardToString(initialBoard);

            // tambahkan initial ke queue
            pq.push({initBoard, 0});
            boardStorage[startKey] = initBoard;

            // pencarian loop
            while (!pq.empty()) {
                Node current = pq.top(); pq.pop();  // ambil node
prioritas, lalu hapus
                string key = boardToString(current.board.board);
                if (visited.count(key)) continue; // lewati apabila
sudah pernah dikunjungi
                visited.insert(key);

                if (IsGoal(current.board.board)) {
                    boardStorage[key] = current.board;
                    return BuildPath(key, parent, boardStorage);
                }

                BoardManipulator bm(current.board.board,
initialVehicles);
                bm.UpdateVehiclesFromBoard();
                auto successors = bm.GetSuccessors();  // bangkitkan
node tetangga

                for (auto& succ: successors) {
                    string sKey = boardToString(succ.board);
                    if (!visited.count(sKey)) {
                        int cost = current.cost + 1; // hitung cost
                        succ.cost = cost;
                        pq.push({succ, cost});
                        parent[sKey] = key;
                        boardStorage[sKey] = succ;
                    }
                }
                ++node;
            }
            return {};
```

```
        }

        // Algoritma GBFS
        vector<Successor> GBFS() {
            priority_queue<Node, vector<Node>, greater<Node>> pq;
            unordered_set<string> visited;
            unordered_map<string, string> parent;
            unordered_map<string, Successor> boardStorage;

            Successor initBoard = Successor{initialBoard, '-', ""};
            string startKey = boardToString(initialBoard);

            pq.push({initBoard, heuristic(initialBoard)});
            boardStorage[startKey] = initBoard;

            while (!pq.empty()) {
                Node current = pq.top(); pq.pop();
                string key = boardToString(current.board.board);
                if (visited.count(key)) continue;
                visited.insert(key);

                if (IsGoal(current.board.board)) {
                    boardStorage[key] = current.board;
                    return BuildPath(key, parent, boardStorage);
                }

                BoardManipulator bm(current.board.board,
initialVehicles);
                bm.UpdateVehiclesFromBoard();
                auto successors = bm.GetSuccessors();

                for (auto& succ: successors) {
                    string sKey = boardToString(succ.board);
                    if (!visited.count(sKey)) {
                        int cost = heuristic(succ.board);
                        succ.cost = cost;
                        pq.push({succ, cost});
                        parent[sKey] = key;
                        boardStorage[sKey] = succ;
                    }
                }
                ++node;
            }
            return {};
        }

        // Algoritma A*
        vector<Successor> AStar() {
            priority_queue<Node, vector<Node>, greater<Node>> pq;
            unordered_map<string, int> gScore;
            unordered_map<string, string> parent;
            unordered_map<string, Successor> boardStorage;

            Successor initBoard = Successor{initialBoard, '-', ""};
            string startKey = boardToString(initialBoard);

            pq.push({initBoard, 0 + heuristic(initBoard.board)});
            gScore[startKey] = 0;
            boardStorage[startKey] = initBoard;
```

```
            while (!pq.empty()) {
                Node current = pq.top(); pq.pop();
                string key = boardToString(current.board.board);

                if (IsGoal(current.board.board)) {
                    boardStorage[key] = current.board;
                    return BuildPath(key, parent, boardStorage);
                }

                BoardManipulator bm(current.board.board,
initialVehicles);
                bm.UpdateVehiclesFromBoard();
                auto successors = bm.GetSuccessors();

                for (auto& succ: successors) {
                    string sKey = boardToString(succ.board);
                    int dummyg = current.cost + 1;

                    if (!gScore.count(sKey) || dummyg <
gScore[sKey]) {
                        int f = dummyg + heuristic(succ.board);
                        succ.cost = f;
                        gScore[sKey] = dummyg;
                        parent[sKey] = key;
                        boardStorage[sKey] = succ;
                        pq.push({succ, f});
                    }
                }
                ++node;
            }
            return {};
        }
};

// fungsi untuk memeriksa, apakah kendaraan utama, P, terhalang oleh
kendaraan yang sejajar juga menuju K
bool IsBlocked(vector<vector<char>>& board, Puzzle& puzzle) {
    int dx = kOut.x - pVhc.x;
    int dy = kOut.y - pVhc.y;
    int dirx = (dx != 0) ? dx / abs(dx) : 0;
    int diry = (dy != 0) ? dy / abs(dy) : 0;

    int cx = pVhc.x + dirx;
    int cy = pVhc.y + diry;

    while (cx >= 0 && cx < h && cy >= 0 && cy < w) {
        char c = board[cx][cy];
        if (c != '.' && c != 'P') {
            Vehicle vhc;

            for (const auto& p: puzzle.layout) {
                if (p.id == c) {
                    vhc = p;
                    break;
                }
            }

            bool isSameOrientation = false;
            if (pVhc.horizontal) {
                if (vhc.horizontal) {
```

```
                    isSameOrientation = true;
                }
            } else {
                if (!vhc.horizontal) {
                    isSameOrientation = true;
                }
            }

            if (isSameOrientation) {
                return true;
            }
        }
        cx += dirx;
        cy += diry;
    }

    return false;
}

int main(int argc, char* argval[]) {
    if (argc < 3) {
        cout << "Masukkan perintah: " << argval[0] << " <nama_file>
<algoritma>\n";
        return 1;
    }

    string fname = argval[1];  // nama file
    string algo = argval[2];  // algoritma
    ifstream file(fname);

    if (!file.is_open()) {
        cerr << "Eror saat membuka file: " << fname << "\n";
        return 1;
    }

    int nPieces;
    Puzzle puzzle;
    vector<string> initial;

    ReadFile(file, w, h, nPieces, initial);
    GetInitLayout(initial, h, w, puzzle);

    State state;
    // getstate
    for (auto& vhc: puzzle.layout) {
        if (vhc.horizontal) {
            state.positions[vhc.id] = vhc.y;
        } else {
            state.positions[vhc.id] = vhc.x;
        }
    }

    ofstream out("result_" + fname);

    if (out.is_open()) {
        auto board = BuildBoard(puzzle, state, h, w);
        if (IsBlocked(board, puzzle)) {
            cout << "Tidak ditemukan solusi.\n";
            out << "Tidak ditemukan solusi.\n";
            return 1;
```

```
        }

        RushHourSolver solver(board, puzzle.layout);

        vector<Successor> solution;
        auto start = high_resolution_clock::now();
        if (algo == "ucs") {
            solution = solver.UCS();
        } else if (algo == "gbfs") {
            solution = solver.GBFS();
        } else if (algo == "astar") {
            solution = solver.AStar();
        } else {
            cout << "Masukkan pilihan algoritma: ucs, gbfs, astar\n";
            return 1;
        }
        auto end = high_resolution_clock::now();
        auto duration = duration_cast<milliseconds>(end - start);

        if (!solution.empty()) {
            cout << "Solusi ditemukan:\n";
                for (int step = 0; step < solution.size(); ++step) {
                    auto& s = solution[step];
                    if (step == 0) {
                        cout << "Papan Awal\n";
                        out << "Papan Awal\n";
                    } else {
                        cout << "Gerakan " << step << ": " << s.id
<< "-" << s.dir << endl;
                        out << "Gerakan " << step << ": " << s.id <<
"-" << s.dir << endl;
                    }
                    PrintBoard(s, cout, true);
                    PrintBoard(s, out, false);
                    cout << endl;
                    out << endl;
                }
                cout << "Jumlah node: " << node << endl;
                cout << "Waktu eksekusi: " << duration.count() << "
ms" << endl;

                out << "Jumlah node: " << node << endl;
                out << "Waktu eksekusi: " << duration.count() << "
ms" << endl;
        } else {
            cout << "Tidak ditemukan solusi.\n";
            out << "Tidak ditemukan solusi.\n";
        }
    }

    out.close();
    return 0;
}
```

b. Implementasi

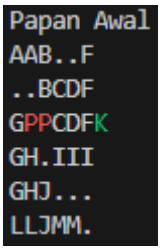
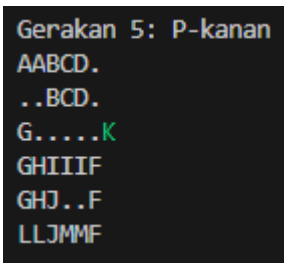
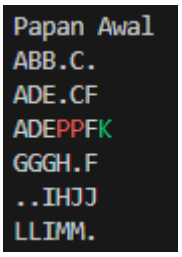
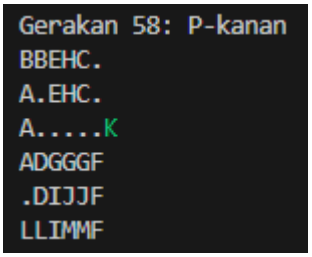
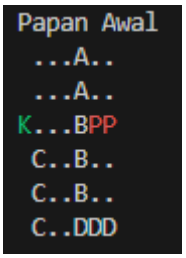
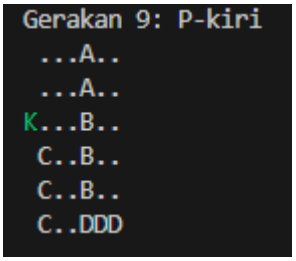
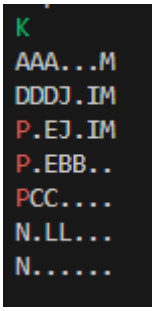
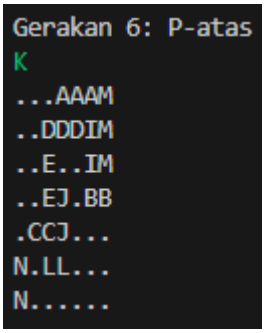
Berdasarkan *source code* di atas, berikut merupakan penjelasan dari fungsi dan prosedur yang digunakan.

Class	
`RushHourSolver`	Kelas RushHourSolver merupakan inti dari pemecah permainan Rush Hour, yang bertujuan untuk mengeluarkan mobil utama 'P' dari papan permainan menggunakan algoritma pencarian.
Atribut	
`initialBoard`	Menyimpan representasi awal papan permainan dalam bentuk matriks karakter 2D
`initial Vehicles`	Menyimpan daftar kendaraan beserta posisinya pada awal permainan
Fungsi dan Prosedur	
`boardToString`	Mengubah papan 2D menjadi string satu dimensi agar dapat digunakan sebagai kunci dalam struktur data seperti unordered_map atau unordered_set
`IsGoal`	Menentukan apakah permainan telah selesai, yang dalam implementasinya berarti mobil 'P' tidak lagi ditemukan di papan.
`heuristic`	Menghitung banyaknya kendaraan yang menghalangi jalur mobil 'P' menuju pintu keluar, dan digunakan dalam algoritma berbasis heuristik seperti Greedy Best-First Search dan A*.
`BuildPath`	Untuk membangun kembali jalur solusi dari kondisi awal hingga kondisi goal, digunakan fungsi BuildPath, yang menelusuri peta parent dari node tujuan ke node awal.
`UCS`	Bekerja seperti BFS dengan mempertimbangkan jumlah langkah dari awal dan tidak menggunakan heuristik.
`GBFS`	Menggunakan hanya nilai heuristik untuk memperkirakan jarak ke goal, sehingga lebih cepat namun tidak menjamin solusi optimal.

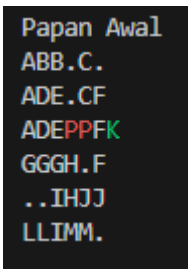
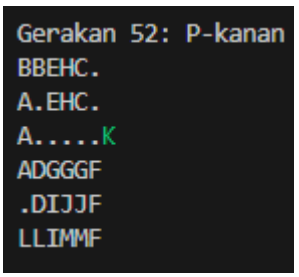
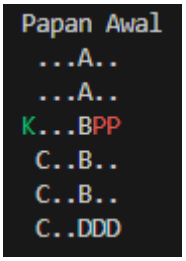
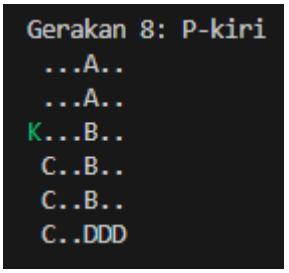
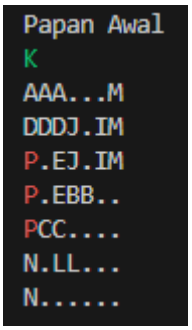
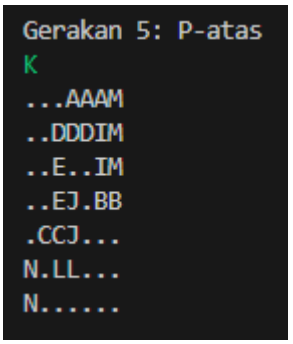
`AStar`	Menggunakan hanya nilai heuristik untuk memperkirakan jarak ke goal, sehingga lebih cepat namun tidak menjamin solusi optimal.

3. Hasil dan Analisis

a. Hasil

Konfigurasi	State Awal	State Akhir
UCS		
- Ukuran: 6, 6 - Banyak kendaraan: 11 - Jumlah gerakan: 5 - Jumlah node: 134 - Waktu eksekusi: 10 ms		
- Ukuran: 6, 6 - Banyak kendaraan: 12 - Jumlah gerakan: 58 - Jumlah node: 2220 - Waktu eksekusi: 195		
- Ukuran: 6, 6 - Banyak kendaraan: 4 - Jumlah gerakan: 9 - Jumlah node: 20 - Waktu eksekusi: 0 ms		
- Ukuran: 7, 7 - Banyak kendaraan: 9 - Jumlah gerakan: 6 - Jumlah node: 512 - Waktu eksekusi: 51 ms		

<table>
<tr><th colspan="3">GBFS</th></tr>
<tr>
<td>- Ukuran: 6, 6
- Banyak kendaraan: 11
- Jumlah gerakan: 11
- Jumlah node: 618
- Waktu eksekusi: 62 ms</td>
<td><pre>Solusi ditemukan:
Papan Awal
AAB..F
..BCDF
GPPCDFK
GH.III
GHJ...
LLJMM.</pre></td>
<td><pre>Gerakan 11: P-kanan
AABCD.
..BCD.
G.....K
GHIIIF
GHJ..F
LLJMMF</pre></td>
</tr>
<tr>
<td>- Ukuran: 6, 6
- Banyak kendaraan: 12
- Jumlah gerakan: 118
- Jumlah node: 2862
- Waktu eksekusi: 233 ms</td>
<td><pre>Papan Awal
ABB.C.
ADE.CF
ADEPPFK
GGGH.F
..IHJJ
LLIMM.</pre></td>
<td><pre>Gerakan 118: P-kanan
BBEHC.
A.EHC.
AD....K
ADGGGF
JJI..F
LLIMMF</pre></td>
</tr>
<tr>
<td>- Ukuran: 6, 6
- Banyak kendaraan: 4
- Jumlah gerakan: 10
- Jumlah node: 15
- Waktu eksekusi: 0 ms</td>
<td><pre>Papan Awal
 ...A..
 ...A..
K...BPP
 C..B..
 C..B..
 C..DDD</pre></td>
<td><pre>Gerakan 10: P-kiri
 ...A..
 ...A..
K...B..
 C..B..
 C..B..
 CDDD..</pre></td>
</tr>
<tr>
<td>- Ukuran: 7, 7
- Banyak kendaraan: 9
- Jumlah gerakan: 21
- Jumlah node: 91
- Waktu eksekusi: 10 ms</td>
<td><pre>Papan Awal
K
AAA...M
DDDJ.IM
P.EJ.IM
P.EBB..
PCC....
N.LL...
N.....</pre></td>
<td><pre>Gerakan 21: P-atas
K
...AAAM
..DDDIM
.....IM
..E.BB.
..E..CC
NLLJ...
N..J...</pre></td>
</tr>
<tr><th colspan="3">A*</th></tr>
<tr>
<td>- Ukuran: 6, 6
- Banyak kendaraan: 11
- Jumlah gerakan: 5
- Jumlah node: 117
- Waktu eksekusi: 14</td>
<td><pre>Papan Awal
AAB..F
..BCDF
GPPCDFK
GH.III
GHJ...
LLJMM.</pre></td>
<td><pre>Gerakan 5: P-kanan
AABCD.
..BCD.
G.....K
GHIIIF
GHJ..F
LLJMMF</pre></td>
</tr>
</table>

- Ukuran: 6, 6 - Banyak kendaraan: 12 - Jumlah gerakan: 52 - Jumlah node: 2134 - Waktu eksekusi: 144 ms		
- Ukuran: 6, 6 - Banyak kendaraan: 4 - Jumlah gerakan: 8 - Jumlah node: 21 - Waktu eksekusi: 1 ms		
- Ukuran: 7, 7 - Banyak kendaraan: 9 - Jumlah gerakan: 5 - Jumlah node: 294 - Waktu eksekusi: 27 ms		

b. Analisis

Fungsi *cost* ($g(n)$) yang digunakan pada implementasi algoritma UCS, GBFS, dan A* yang digunakan, yaitu jumlah langkah yang telah diambil dari *state* awal hingga ke *state* saat ini. Sedangkan fungsi heuristik ($h(n)$) yang digunakan adalah banyaknya blok yang terhalang dari kendaraan utama sampai ke jalan keluar. Jadi, fungsi $f(n)$ yang digunakan adalah penjumlahan dari jumlah langkah yang sudah dilalui dari awal state dan jumlah blok yang terhalang oleh kendaraan untuk jalur keluar.

Heuristik yang digunakan pada algoritma A* bersifat *admissible* karena banyaknya blok yang terhalang dari jalan keluar hingga kendaraan P tidak mungkin lebih dari jarak blok terhalang yang sebenarnya. Estimasi ini hanya menghitung jumlah blok yang terhalang kendaraan untuk jalan keluar, sehingga nilainya selalu lebih kecil atau sama dengan langkah minimum yang dibutuhkan. Dengan demikian, heuristik ini menjamin bahwa A* dapat menemukan solusi yang optimal.

Pada permainan Rush Hour, algoritma UCS dapat sama dengan BFS apabila mempunyai *cost* yang sama pada setiap pembangkitan *node* tetangga baru dan sistem antrian algoritmanya memanfaatkan sistem antrian FIFO. Namun, pada implementasi

ini, sistem antrian yang digunakan merupakan *priority queue*, yang tidak menjamin urutan FIFO apabila *node* tetangga memiliki *cost* yang sama.

Berdasarkan hasil yang diperoleh, jumlah gerakan pada algoritma A* selalu lebih kecil atau sama dengan jumlah gerakan pada algoritma UCS. Hal ini menandakan bahwa algoritma A* cukup efisien dalam menyelesaikan permainan Rush Hour. Algoritma A* mengandalkan fungsi cost dari awal node hingga node tujuan, yang menunjukkan bahwa algoritma ini mempertimbangkan jumlah langkah dan jumlah blok yang terhalangi bernilai minimum. Sedangkan pada algoritma UCS, fungsi *cost* yang digunakan hanya jumlah langkah yang telah dilakukan dari awal *state*, yang mengindikasikan algoritma ini hanya mempertimbangkan biaya dari awal *node* hingga *state* saat ini saja, sehingga sering kali mengeksplorasi lebih banyak node yang tidak mengarah langsung ke tujuan. Hal ini menyebabkan UCS cenderung membutuhkan lebih banyak langkah dibandingkan A*.

Pada penyelesaian permainan Rush Hour, algoritma Greedy Best First Search tidak selalu menghasilkan solusi yang optimal dalam menyelesaikan permainan. Hal ini karena pada algoritma GBFS, hanya mengandalkan fungsi heuristik yang mengestimasi jarak dari kendaraan utama sampai ke jalan keluar, tidak dari *state* awal. Selain itu, pada algoritma ini juga tidak dijamin menemukan solusi karena bisa saja terjebak di node yang tidak memiliki tetangga (tidak *complete*).

Kompleksitas waktu dan ruang algoritma UCS dalam notasi Big O adalah O(b^d) dengan d adalah kedalaman dari ruang pencarian dan b adalah *branching factor*. UCS mengeksplorasi semua node dengan *cost* rendah terlebih dahulu sebelum beralih ke *cost* yang lebih tinggi, sehingga cukup optimal.

Kompleksitas waktu dan ruang algoritma GBFS dalam notasi Big O adalah O(b^m) dengan m adalah kedalaman maksimum pencarian yang dapat lebih besar dari d. Algoritma ini hanya berfokus pada heuristik, sehingga bisa terjebak atau memiliki jalur yang panjang.

Kompleksitas waktu dan ruang algoritma A* dalam notasi Big O adalah O(b^d) dengan d adalah kedalaman ruang pencarian. Jika heuristik yang digunakan kurang optimal, kompleksitas waktunya bisa mendekati O(b^m). Algoritma ini bersifat optimal dan efisien apabila menggunakan heuristik yang *admissible* dan konsisten.

4. Kesimpulan

Pada tugas ini, penulis telah berhasil membuat program penyelesaian puzzle rush hour dengan algoritma pathfinding. Berdasarkan hasil running program, algoritma A* selalu dapat menemukan solusi yang optimal. Algoritma A* selalu lebih kecil atau sama dengan jumlah

gerakan pada algoritma UCS. Algoritma UCS seringkali mengeksplorasi lebih banyak *node* yang tidak mengarah langsung ke tujuan. Hal ini menyebabkan UCS cenderung membutuhkan lebih banyak langkah dibandingkan A*. Algoritma Greedy Best First Search tidak selalu menghasilkan solusi yang optimal. Hal ini karena pada algoritma GBFS, hanya mengandalkan fungsi heuristik yang mengestimasi jarak dari kendaraan utama sampai ke jalan keluar, tidak dari *state* awal. Selain itu, pada algoritma ini juga tidak dijamin menemukan solusi karena bisa saja terjebak di *node* yang tidak memiliki tetangga (tidak *complete*).

Pada implementasi ini, terdapat dua atribut utama yang digunakan, yaitu *initialBoard* dan *initial Vehicles. initialBoard* ini digunakan untuk representasi awal papan permainan dalam bentuk matriks karakter 2D. *Initial Vehicles* digunakan untuk menyimpan daftar kendaraan beserta posisinya pada awal permainan. Terdapat juga fungsi seperti *boardToString, IsGoal, heuristic,* dan *BuildPath*. *boardToString* digunakan untuk mengubah papan 2D menjadi string, lalu ditentukan apakah 'P' sudah keluar dari papan permainan oleh *IsGoal*. Nantinya, jalur solusi akan dibangun kembali oleh *BuildPath* dan banyaknya kendaraan akan dideteksi oleh *heuristic*.

Selain itu kompleksitas algoritma untuk algoritma UCS dalam notasi Big O adalah O(b^d) dengan d adalah kedalaman dari ruang pencarian dan b adalah *branching factor*. Kompleksitas waktu dan ruang algoritma GBFS dalam notasi Big O adalah O(b^m) dengan m adalah kedalaman maksimum pencarian yang dapat lebih besar dari d. Kompleksitas waktu dan ruang algoritma A* dalam notasi Big O adalah O(b^d) dengan d adalah kedalaman ruang pencarian. Jika heuristik yang digunakan kurang optimal, kompleksitas waktunya bisa mendekati O(b^m).

Oleh karena itu, penulis telah berhasil mengimplementasikan penyelesaian puzzle *Rush Hour* dengan memanfaatkan algoritma pathfinding UCS, GBFS, dan A*. Dari hasil pengujian dan analisis, dapat disimpulkan bahwa algoritma A* merupakan pilihan paling efektif dan efisien dalam menyelesaikan permainan ini.

REFERENSI

Greedy Best-First Search in AI. GreeksforGeeks. Retrieved May 20, 2025, from https://www.geeksforgeeks.org/greedy-best-first-search-in-ai/

Munir, R. (2025). *Algoritma Pathfinding*. Homepage Rinaldi Munir. Retrieved May 20, 2025, from https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-(2025)-Bagian1.pdf

The A Algorithm: A Complete Guide*. DataCamp. Retrieved May 20, 2025, from https://www.datacamp.com/tutorial/a-star-algorithm

Uniform Cost Search (UCS) in AI. GreeksforGeeks. Retrieved May 20, 2025, from https://www.geeksforgeeks.org/uniform-cost-search-ucs-in-ai/

LAMPIRAN

Link *repository* github: https://github.com/fardhan248/Tucil3_12821046_12823024.git

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	**[Bonus]** Implementasi algoritma pathfinding alternatif		✓
6	**[Bonus]** Implementasi 2 atau lebih heuristik alternatif		✓
7	**[Bonus]** Program memiliki GUI		✓
8	Program dan laporan dibuat sendiri	✓	