

LAPORAN TUGAS KECIL III
IF2211 STRATEGI ALGORITMA

RUSH HOUR SOLVER: PATHFINDING ALGORITHM



Disusun oleh:

William Andrian Dharma T - 13523006

Nathan Jovial Hartono - 13523032

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

2024

TABLE OF CONTENTS

TABLE OF CONTENTS.....	2
BAB I	
DESKRIPSI MASALAH.....	3
BAB II	
ALGORITMA PATHFINDING.....	4
BAB III	
ANALISIS TEORI ALGORITMA.....	7
BAB IV	
SOURCE CODE PROGRAM.....	9
Model.....	9
I. Board.java.....	9
II. HNode.java.....	17
III. Node.java.....	19
IV. Piece.java.....	26
V. PrimaryPiece.java.....	27
Controller.....	28
I. MainController.java.....	28
II. ParserController.java.....	31
Algorithm.....	34
I. AbstractSearch.java.....	34
II. AStar.java.....	35
III. GBFS.java.....	38
IV. TwoGreedy.java.....	40
App.....	43
I. Dockerfile.....	43
II. App.tsx.....	44
III. BoardGrid.tsx.....	47
IV. FileUploader.tsx.....	50
BAB V	
HASIL PERCOBAAN.....	55
Test Case 1:.....	55
Test Case 2:.....	59
Test Case 3:.....	63
Test Case 4:.....	65
Test Case 5:.....	69
Test Case 6:.....	73
Test Case 7:.....	77
BAB VI	
ANALISIS PERCOBAAN.....	81
BAB VII	
IMPLEMENTASI BONUS.....	83

BAB VII	
LAMPIRAN.....	84

BAB I

DESKRIPSI MASALAH

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

Komponen penting dari permainan Rush Hour terdiri dari:

1. **Papan** – *Papan* merupakan tempat permainan dimainkan.

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

2. **Piece** – *Piece* adalah sebuah kendaraan di dalam papan. Setiap *piece* memiliki *posisi*, *ukuran*, dan *orientasi*. *Orientasi* sebuah *piece* hanya dapat berupa horizontal atau vertikal—tidak mungkin diagonal. *Piece* dapat memiliki beragam *ukuran*, yaitu jumlah *cell* yang ditempati oleh *piece*. Secara standar, variasi *ukuran* sebuah *piece* adalah *2-piece* (menempati 2 *cell*) atau *3-piece* (menempati 3 *cell*). Suatu *piece* tidak dapat digerakkan melewati/menembus *piece* yang lain.
3. **Primary Piece** – *Primary piece* adalah kendaraan utama yang harus dikeluarkan dari *papan* (biasanya berwarna merah). Hanya boleh terdapat satu *primary piece*.
4. **Pintu Keluar** – *Pintu keluar* adalah tempat *primary piece* dapat digerakkan keluar untuk menyelesaikan permainan
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.

BAB II

ALGORITMA PATHFINDING

Penyelesaian permasalahan *Rush Hour* memanfaatkan algoritma *pathfinding* untuk mencari solusi dari sebuah konfigurasi papan, meskipun tidak selalu ada jaminan solusi tersedia. Algoritma *pathfinding* digunakan karena setiap iterasinya menghasilkan suatu *state* baru dari ruang kemungkinan yang ada, kemudian melakukan ekspansi terhadap *state* tersebut untuk menemukan *path* menuju *state* akhir tujuan. Dengan demikian, algoritma *pathfinding* tidak hanya berguna untuk mencari jarak terpendek antara dua titik, tetapi juga sangat bermanfaat dalam permasalahan yang melibatkan eksplorasi dan evaluasi berbagai *state*, serta mempertimbangkan informasi dari setiap *state* dalam proses pencarian solusi.

Terdapat tiga algoritma *pathfinding* umum yang dapat digunakan untuk memecahkan permasalahan *Rush Hour*: *Greedy Best First Search (GBFS)*, *Uniform Cost Search (UCS)*, dan *A**. Algoritma *pathfinding* menentukan *state* berikutnya dengan menggunakan sebuah fungsi $f(n)$ yang menyimpan sebuah nilai atau biaya dari setiap *state*, sehingga setiap *state* dianggap sebagai sebuah node dalam tree, dimana fungsi yang umum digunakan sebagai berikut:

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

- $g(n)$ menyatakan biaya antara node *state* inisial permasalahan hingga node *state* sekarang
- $h(n)$ menyatakan fungsi heuristic yang menentukan nilai heuristic yang memperkirakan biaya antara node *state* sekarang ke node tujuan akhir, dengan catatan bahwa fungsi heuristic sangat bervariasi sehingga tidak ada fungsi heuristic yang absolut atau pasti benar.

Tetapi setiap algoritma *pathfinding* memanfaatkan dan menyesuaikan fungsi biayanya masing - masing sesuai. *Greedy Best First Search* hanya melibatkan fungsi $h(n)$ dan mengabaikan fungsi $g(n)$, atau $g(n) = 0$. *Uniform Cost Search* hanya melibatkan fungsi $g(n)$ dan mengabaikan fungsi $h(n)$, atau $h(n) = 0$. *A** melibatkan kedua fungsi $h(n)$ dan $g(n)$ sehingga *pathfinding* bersifat lebih efisien dan cepat dalam beberapa kasus tertentu, dibandingkan dengan *GBFS* dan *UCS*.

Algoritma *pathfinding* secara umum bekerja sebagai berikut:

1. Menentukan node *state* inisial dari sebuah permasalahan
2. Ekspansi node *state* sekarang, menghasilkan sejumlah node hasil berdasarkan *state* sebelumnya, contohnya seperti node yang menggerakkan sebuah piece pada arah kanan 1 langkah, arah kiri 1 langkah, arah kanan 2 langkah, dst.

3. Menghitung biaya setiap node hasil ekspansi dengan menggunakan fungsi $f(n) = g(n) + h(n)$, $h(n)$ ditentukan pada node state awal permasalahan dan tidak dapat diubah
4. Memilih node state dengan biaya terkecil dan yang belum pernah digunakan pada algoritma pathfinding untuk pencarian berikutnya
5. Mengulangi langkah 2 hingga mencapai node state tujuan akhir

dengan catatan bahwa algoritma *GBFS*, *UCS*, dan *A** menggunakan langkah - langkah algoritma pathfinding diatas dengan fungsi biayanya tersendiri sehingga pemilihan node pada langkah 4 dapat berbeda pada ketiga algoritma tersebut. Berikut merupakan pseudocode gambaran umum pathfinding

```
// Declare and Initialize the starting Node of the Problem
// We assume Node points to it's parent Node
// Parent Node is defined from expansion
startNode = problem.initNode()

// Store Nodes obtained from expansion, the Nodes are used for expansion
again
Prioqueue Node Open

// Store Nodes that has been expanded
List Node close

// Store the path from goal condition
List Node path

// Add startNode to Prioqueue Open
Open.enqueue(StartNode)

while(open is not empty):

    // Pick the node that has the lowest weight/cost
    NodeToExpand = Open.dequeue()

    // Expand the Node and Check viability for Open
    for (expanded in NodeToExpand.expand()):

        // check goal condition from expanded node
        if expanded is goal:
```

```

        path = expanded.constructPath()

    // add to close if doesn't exist in close
    if(expanded not in close):
        close.add(expanded)

    // check if exists Node in Open with cheaper cost
    if(expanded in Open):
        while(node in Open):
            if(expanded.cost < node):
                // replace the Node in Open with the expanded one
                node = expanded
                break loop

    // add expanded if doesn't exist in Open
    if (expanded not in Open):
        Open.enqueue(expanded)

```

Terdapat algoritma tambahan yang kami buat yang dinamakan TwoGreedy. Algoritma ini merupakan modifikasi dari Greedy Best First Search, namun berbeda dari biasanya yang hanya mengambil satu node dengan heuristik yang paling rendah, TwoGreedy akan mengambil dua node dengan heuristik paling rendah sehingga lebih agresif. Modifikasi ini menambah search space pada tiap iterasi sehingga dapat mengurangi waktu pencarian dan jumlah node yang dikunjungi. Namun algoritma ini lebih rentan terhadap *overfitting* ke *local optima* karena dengan memilih dua node terbaik, algoritma lebih berkomitmen terhadap pilihannya dan lebih susah untuk *backtracking* seperti di GBFS biasa.

BAB III

ANALISIS TEORI ALGORITMA

$f(n)$ merupakan estimasi biaya jarak sebuah node n hingga mencapai node tujuan akhir dengan perumusan berikut:

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

- $g(n)$ menyatakan biaya antara node inisial permasalahan hingga node n
- $h(n)$ menyatakan fungsi heuristic yang menentukan nilai heuristic yang memperkirakan biaya, atau dengan kata lain estimasi biaya node n hingga node tujuan akhir

Pada permasalahan rush hour, $g(n)$ direpresentasikan sebagai depth expansion dari sebuah node dan $h(n)$ sebagai fungsi heuristic seperti fungsi yang menghitung jarak primary piece dengan goal pada state tersebut dan fungsi yang menghitung berapa banyak piece yang menghalangi primary piece dengan goal, dimana kedua fungsi dapat digunakan secara bersamaan. $f(n)$ merupakan fungsi biaya disimpan pada suatu Node dan mengikuti perumusan umum seperti diatas.

Sebuah heuristic dapat dinyatakan admissible apabila untuk setiap node, nilai estimasi biaya dari $h(n)$ tidak melebihi nilai biaya minimum sebenarnya yang dibutuhkan untuk mencapai tujuan dari node tersebut, yaitu $h^*(n)$. Dengan kata lain, heuristic harus memenuhi syarat $h(n) \leq h^*(n)$ untuk semua node dalam *search tree*. Hal ini memastikan bahwa nilai biaya estimasi dari heuristic selalu optimis dan tidak pernah *overestimate*, sehingga apabila syarat ini terpenuhi, algoritma A^* selalu optimal. Akan tetapi, apabila heuristic diabaikan pada A^* , dimana $h(n) = 0$, A^* akan memberikan hasil optimal selayaknya seperti UCS tetapi pencarian menjadi kurang efisien tanpa informasi arah menuju tujuan. Pada permasalahan rush hour, heuristic A^* tidak selalu admissible karena konfigurasi dan interaksi antar kendaraan sehingga sangat sulit mendapatkan heuristic yang konsisten dan selalu admissible dengan banyak variabel dan faktor pada permasalahan.

Dalam konteks permasalahan rush hour, penyelesaian persoalan dengan menggunakan algoritma UCS sama dengan menggunakan BFS untuk mencari path pada node tujuan. Setiap node pada penyelesaian dengan UCS menyimpan nilai biaya $f(n) = g(n)$, dimana $g(n)$ merupakan depth node pada tree dari root inisial node permasalahan, sehingga menggunakan priority queue pada permasalahan UCS sama dengan memanfaatkan queue dengan BFS karena pada kedua kasus node yang dipilih selalu memiliki nilai biaya terkecil sehingga path yang terbentuk akan selalu sama diantara kedua.

Secara teoritis, A* akan menghasilkan hasil yang lebih optimal dibandingkan dengan algoritma UCS, dimana A* memiliki heuristic yang memberikan “arah” kepada node tujuan akhir sehingga proses pencarian pada algoritma A* menjadi lebih optimal dari pencarian dengan algoritma UCS. Sama halnya dengan permainan rush hour, A* menghasilkan hasil lebih optimal karena diarahkan dengan heuristic seperti “blocked cars” atau “distance to goal” yang memberikan arahan dalam proses pencarian sehingga lebih efisien dibandingkan dengan UCS. Akan tetapi, mungkin saja terdapat kasus dimana heuristic di A* akan overestimate atau tidak konsisten sehingga menghasilkan solusi yang tidak efisien dibandingkan UCS. Kesimpulannya, A* akan menjadi lebih efisien apabila menggunakan heuristic yang admissible dan konsisten, tetapi A* belum tentu lebih efisien apabila heuristic tidak dibuat dengan baik.

Secara teoritis, algoritma *Greedy Best First Search (GBFS)*, belum tentu menjamin solusi optimal, bahkan belum tentu menemukan sebuah solusi karena sifat GBFS yang menyimpan biaya sebuah Node dengan $f(n) = h(n)$ dimana $h(n)$ merupakan nilai heuristic. Hal ini menyebabkan GBFS terjebak pada local optimum, dimana pencarian memilih node yang lebih dekat berdasarkan heuristic, tetapi dapat menyebabkan jalan buntu ataupun solusi yang lebih buruk. Pada permasalahan rush hour, GBFS tidak selalu menjamin solusi, bahkan solusi optimal, karena faktor heuristic yang dibuat, seperti heuristic mengukur jarak *primary piece* dengan *goal* tanpa memperhatikan faktor seperti banyaknya piece yang menghalangi *goal*, sehingga GBFS akan memilih node yang menurut heuristic lebih dekat dan belum tentu menjamin efisiensi ataupun solusi.

BAB IV

SOURCE CODE PROGRAM

Model

I. Board.java

```
package kirisame.rush_solver.model;

import java.util.HashMap;
import java.util.HashSet;

public class Board {
    private int height;
    private int width;
    private char[][] board;
    private int normalPieceCount;
    private int[] endGoal = new int[2];
    private HashMap<Character, Piece> pieces = new HashMap<>();
    private boolean solved = false;
    public static final char WALL = '©'; // fuck you @kirisame
    public static final char EMPTY = ' ';
    public static final char SPACE = '.';

    public Board() {
        this.height = 0;
        this.width = 0;
        this.board = new char[0][0];
        this.normalPieceCount = 0;
        this.endGoal[0] = 0;
        this.endGoal[1] = 0;
        this.pieces = new HashMap<>();
        this.solved = false;
    }

    public int getHeight() {
        return height;
    }

    public int getWidth() {
```

```

        return width;
    }

    /**
     *
     * @return a clone of the board
     */
    public char[][] getBoard() {
        char[][] copy = new char[this.board.length][];
        for (int i = 0; i < this.board.length; i++) {
            copy[i] = this.board[i].clone();
        }
        return copy;
    }

    public void setBoard(char[][] board) {
        this.board = board.clone();
    }

    /**
     * Care to account for the border.
     */
    public void setBoardAt(int row, int col, char value) {
        if (row < 0 || col < 0 || row >= height || col >= width) {
            throw new IndexOutOfBoundsException("Coordinates are out
of bounds.");
        }
        this.board[row][col] = value;
    }

    public int getPieceCount() {
        return normalPieceCount;
    }

    public int[] getEndGoal() {
        return endGoal;
    }

    public HashMap<Character, Piece> getPieces() {
        return pieces;
    }

```

```

    }

    public HashSet<Character> getPieceIds() {
        HashSet<Character> pieceIds = new HashSet<>();
        for (Piece p : pieces.values()) {
            pieceIds.add(p.id);
        }
        return pieceIds;
    }

    public boolean isSolved() {
        return solved;
    }

    /**
     * Sets the size of the board.
     * Size is padded by 1 to account for the border.
     */
    public void setSize(int height, int width) {
        if (height <= 0 || width <= 0) {
            throw new IllegalArgumentException("Height and width must
be positive integers.");
        }
        this.width = width + 2;
        this.height = height + 2;
        this.board = new char[height + 2][width + 2];
        // Initialize the board with walls
        for (int i = 0; i < height + 2; i++) {
            for (int j = 0; j < width + 2; j++) {
                this.board[i][j] = WALL;
            }
        }
    }

    public void setPieceCount(int normalPieceCount) {
        if (normalPieceCount <= 0) {
            throw new IllegalArgumentException("Piece count must be a
positive integer.");
        }
        this.normalPieceCount = normalPieceCount;
    }

```

```

    }

    public void setEndGoal(int row, int col) {
        if (row < 0 || col < 0) {
            throw new IllegalArgumentException("X and Y coordinates
must be non-negative integers.");
        }
        this.endGoal[0] = row;
        this.endGoal[1] = col;
    }

    public String boardToString() {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                sb.append(board[i][j]);
            }
            sb.append("\n");
        }
        return sb.toString();
    }

    public void printPieces() {
        for (Piece p : pieces.values()) {
            p.printInfo();
        }
    }

    public void parsePieces() {
        pieces.clear();
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                char value = board[i][j];
                if (value != WALL && value != EMPTY && value != '.')
{
                    if (value == 'K') {
                        this.endGoal[0] = i;
                        this.endGoal[1] = j;
                        continue;
                    }
                }
            }
        }
    }

```

```

        if (!pieces.containsKey(value)) {
            int length = 0;
            int axis;
            // Check horizontal
            for (int k = j; k < width && board[i][k] ==
value; k++) {

                length++;
            }
            if (length > 1) {
                axis = 0;
                if (value == 'P') {
                    pieces.put(value, new
PrimaryPiece(length, axis, i, j));
                } else {
                    pieces.put(value, new Piece(value,
length, axis, i, j));
                }
            }
            // Check vertical
            length = 0;
            for (int k = i; k < height && board[k][j] ==
value; k++) {

                length++;
            }
            if (length > 1) {
                axis = 1;
                if (value == 'P') {
                    pieces.put(value, new
PrimaryPiece(length, axis, i, j));
                } else {
                    pieces.put(value, new Piece(value,
length, axis, i, j));
                }
            }
        }
    }
}

```

```

public Board deepCopy() {
    Board copy = new Board();

    // Copy dimensions
    copy.setSize(this.getHeight() - 2, this.getWidth() - 2); //
Account for padding

    // Copy board (deep)
    char[][] boardCopy = this.getBoard(); // Now does deep copy
after update
    copy.setBoard(boardCopy);

    // Copy normal piece count
    copy.setPieceCount(this.getPieceCount());

    // Copy end goal
    int[] goal = this.getEndGoal();
    copy.setEndGoal(goal[0], goal[1]);

    // Copy pieces (deep)
    for (Piece p : this.getPieces().values()) {
        Piece copyPiece;
        if (p instanceof PrimaryPiece) {
            copyPiece = new PrimaryPiece(p.getLength(),
p.getAxis(), p.getRow(), p.getCol());
        } else {
            copyPiece = new Piece(p.getId(), p.getLength(),
p.getAxis(), p.getRow(), p.getCol());
        }
        copy.getPieces().put(copyPiece.id, copyPiece);
    }

    return copy;
}

public boolean equals(Board other) {

    if (other.getHeight() == 0 || other.getWidth() == 0) {
        return false;
    }

```

```

        // Compare board dimensions
        if (this.height != other.getHeight() || this.width !=
other.getWidth()) {
            return false;
        }

        // Compare board contents

        char[][] otherBoard = other.getBoard();

        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                if (this.board[i][j] != otherBoard[i][j]) {
                    return false;
                }
            }
        }

        // compare normal piece count
        if (this.normalPieceCount != other.getPieceCount()) {
            return false;
        }

        // Compare end goal
        int[] otherEndGoal = other.getEndGoal();
        return (!(this.endGoal[0] != otherEndGoal[0] ||
this.endGoal[1] != otherEndGoal[1]));
    }

    /**
     * Moves the piece in the specified direction.
     *
     * @param p          the piece moved
     * @param distance positive goes Right and Up, negative goes Left
and Down
     */
    public void move(Piece p, int distance) {
        boolean positive = distance > 0;
        distance = Math.abs(distance);
    }

```



```

char[][] tempBoard = getBoard(); // deep copy from getBoard()

// Clear the piece from the board
for (int i = 0; i < p.length; i++) {
    int row = p.axis == 0 ? p.row : p.row + i;
    int col = p.axis == 0 ? p.col + i : p.col;
    tempBoard[row][col] = '.'; // Mark as empty
}

// Check collision and update piece position
for (int i = 1; i <= distance; i++) {
    int checkRow = p.axis == 0 ? p.row : (positive ? p.row +
p.length - 1 + i : p.row - i);
    int checkCol = p.axis == 0 ? (positive ? p.col + p.length
- 1 + i : p.col - i) : p.col;

    if (checkRow < 0 || checkCol < 0 || checkRow >= height ||
checkCol >= width)
        throw new IllegalArgumentException("Out of bounds");

    char cell = tempBoard[checkRow][checkCol];
    if (cell != '.' && !(cell == 'K' && p instanceof
PrimaryPiece)) {
        throw new IllegalArgumentException("Collision at (" +
checkRow + "," + checkCol + ")");
    }
}

// Update piece location
if (p.axis == 0) {
    p.col += positive ? distance : -distance;
} else {
    p.row += positive ? distance : -distance;
}

// Place the piece at the new location
for (int i = 0; i < p.length; i++) {
    int row = p.axis == 0 ? p.row : p.row + i;
    int col = p.axis == 0 ? p.col + i : p.col;
    tempBoard[row][col] = p.id;
}

```

```

    }

    // Update the board
    setBoard(tempBoard);
}

public boolean isGoal() {
    char[][] grid = this.getBoard();
    int[] goal = this.getEndGoal();
    if (goal[0] == 0) {
        return grid[goal[0] + 1][goal[1]] == 'P';
    } else if (goal[0] == this.getHeight() - 1) {
        return grid[goal[0] - 1][goal[1]] == 'P';
    } else if (goal[1] == 0) {
        return grid[goal[0]][goal[1] + 1] == 'P';
    } else if (goal[1] == this.getWidth() - 1) {
        return grid[goal[0]][goal[1] - 1] == 'P';
    } else
        return false;
}
}

```

II. HNode.java

```

package kirisame.rush_solver.model;

public class HNode extends Node {

    private int f;

    public HNode(Node node) {

        //  $f(n) = g(n) + h(n)$ 
        //  $g(n) = \text{depth}$ 
        super(node.getBoard(), node.getParent(), node.getDepth(),
node.heuristic, node.getMovedPiece(),
        node.getMoveDistance());
        this.f = node.getDepth() + this.getHeuristicValue();
    }
}

```

```

    public int getF() {
        return f;
    }

    /**
     * Compares this node with another node based on the f value.
     * This is used for sorting the priority queue.
     *
     * @param other
     * @return
     */
    public int compareTo(HNode other) {
        if (this.f == other.getF()) {
            return 0;
        }
        if (this.f > other.getF()) {
            return 1;
        }
        return -1;
    }

    /**
     * Compares this Hnode with another Hnode based on the f value
    and Board object.
     *
     * @param other
     * @return
     */
    public boolean equals(HNode other) {
        // compare f value
        if (this.f != other.getF()) {
            return false;
        }

        // compare board
        return (this.getBoard().equals(other.getBoard()));
    }
}

```

III. Node.java

```
package kirisame.rush_solver.model;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;

public class Node {

    private Board board;
    private Node parent;
    private int depth;
    String heuristic;
    private int heuristicValue;
    private char movedPiece;
    private int moveDistance;

    public Node(Board board, Node parent, int depth, String
    heuristic, char movedPiece, int moveDistance) {
        this.board = board.deepCopy();
        this.parent = parent;
        this.depth = depth;
        this.heuristic = heuristic;
        calculateCost();
        this.movedPiece = movedPiece;
        this.moveDistance = moveDistance;
    }

    public Node(Board board, Node parent, int depth, String
    heuristic) {
        this.board = board.deepCopy();
        this.parent = parent;
        this.depth = depth;
        this.heuristic = heuristic;
        calculateCost();
        this.movedPiece = '-';
        this.moveDistance = 0;
    }
}
```

```

public Board getBoard() {
    return board;
}

public Node getParent() {
    return parent;
}

public int getDepth() {
    return depth;
}

public int getHeuristicValue() {
    return heuristicValue;
}

public String getHeuristic() {
    return heuristic;
}

public void setHeuristic(String heuristic) {
    this.heuristic = heuristic;
}

public char getMovedPiece() {
    return movedPiece;
}

public int getMoveDistance() {
    return moveDistance;
}

/**
 * Expands the current node by checking all possible moves for
each piece
 *
 * @return a list of nodes that are the result of moving the
pieces
 */
public Node[] expand() {
    ArrayList<Node> nodes = new ArrayList<>();

```

```

        HashMap<Character, Piece> pieces =
this.board.deepCopy().getPieces();

        for (Piece piece : pieces.values()) {

            // Try moving in positive direction
            for (int dist = 1;; dist++) {
                try {
                    Board newBoard = this.board.deepCopy();
                    Piece copiedPiece =
newBoard.getPieces().get(piece.getId());
                    newBoard.move(copiedPiece, dist);
                    nodes.add(new Node(newBoard, this, this.depth +
1, this.heuristic, piece.getId(), dist));
                } catch (Exception e) {
                    break; // Stop trying further in this direction
                }
            }

            // Try moving in negative direction
            for (int dist = -1;; dist--) {
                try {
                    Board newBoard = this.board.deepCopy();
                    Piece copiedPiece =
newBoard.getPieces().get(piece.getId());
                    newBoard.move(copiedPiece, dist);
                    nodes.add(new Node(newBoard, this, this.depth +
1, this.heuristic, piece.getId(), dist));
                } catch (Exception e) {
                    break;
                }
            }

        }

        return nodes.toArray(Node[]::new);
    }

    // Heuristics
    private void calculateCost() {
        switch (this.heuristic.toLowerCase()) {

```

```

        case "blocking" -> {
            this.heuristicValue = this.blockingPieces();
        }
        default -> throw new IllegalArgumentException("Invalid
heuristic: " + heuristic);
        case "none" -> {
            this.heuristicValue = 0;
        }
        case "ucs" -> {
            this.heuristicValue = 0;
        }
        case "distance" -> {
            this.heuristicValue = this.distanceToGoal();
        }
        case "blocking_distance" -> {
            this.heuristicValue = this.distanceToGoal() +
this.blockingPieces();
        }
    }
}

/**
 * Counts the number of pieces that are blocking the primary
piece
 *
 * @param node the node to check
 * @return the number of blocking pieces
 */
public int blockingPieces() {
    int count = 0;
    int axis = this.board.getPieces().get('P').getAxis();
    int row = this.board.getPieces().get('P').getRow();
    int col = this.board.getPieces().get('P').getCol();
                                int pieceLength =
this.board.getPieces().get('P').getLength();
    HashSet<Character> visitedPieces = new HashSet<>();

    // For horizontal primary piece
    if (axis == 0) {
        // Get rightmost position of the piece

```

```

        int startCol = col + pieceLength - 1;

        if (this.board.getEndGoal()[1] < startCol) {
            // Need to move left
            for (int i = startCol; i >=
this.board.getEndGoal()[1]; i--) {
                char value = this.board.getBoard()[row][i];
                if (value == 'K') {
                    break;
                }
                if (value >= 'A' && value <= 'Z' && value != 'P')
{
                    if (!visitedPieces.contains(value)) {
                        count++;
                        visitedPieces.add(value);
                    }
                }
            }
        } else {
            // Need to move right
            for (int i = startCol; i <=
this.board.getEndGoal()[1]; i++) {
                char value = this.board.getBoard()[row][i];
                if (value == 'K') {
                    break;
                }
                if (value >= 'A' && value <= 'Z' && value != 'P')
{
                    if (!visitedPieces.contains(value)) {
                        count++;
                        visitedPieces.add(value);
                    }
                }
            }
        }
    } else {
        // For vertical primary piece
        int startRow = row + pieceLength - 1;

        if (this.board.getEndGoal()[0] < startRow) {

```



```

        // Need to move up
        for (int i = startRow; i >=
this.board.getEndGoal()[0]; i--) {
            char value = this.board.getBoard()[i][col];
            if (value == 'K') {
                break;
            }
            if (value >= 'A' && value <= 'Z' && value != 'P')
{
                if (!visitedPieces.contains(value)) {
                    count++;
                    visitedPieces.add(value);
                }
            }
        }
    } else {
        // Need to move down
        for (int i = startRow; i <=
this.board.getEndGoal()[0]; i++) {
            char value = this.board.getBoard()[i][col];
            if (value == 'K') {
                break;
            }
            if (value >= 'A' && value <= 'Z' && value != 'P')
{
                if (!visitedPieces.contains(value)) {
                    count++;
                    visitedPieces.add(value);
                }
            }
        }
    }
}

return count;
}

private int distanceToGoal() {
    Piece primary = this.getBoard().getPieces().get('P');
    int[] goal = this.getBoard().getEndGoal();

```

```

        if (primary == null) {
            System.out.println("Primary piece not found in the
board.");
            return 0;
        }
        if (primary.getAxis() == 0) {
            return Math.abs(goal[1] - (primary.getCol() +
primary.getLength() - 1));
        } else {
            return Math.abs(goal[0] - (primary.getRow() +
primary.getLength() - 1));
        }
    }

    public String getDirection() {
        Piece moved = this.board.getPieces().get(this.movedPiece);
        if (moved == null) {
            System.out.println("Moved piece not found in the
board.");
            return "";
        }
        int axis = moved.getAxis();
        if (axis == 0) {
            return (this.moveDistance < 0 ? "LEFT" : "RIGHT");
        } else {
            return (this.moveDistance < 0 ? "UP" : "DOWN");
        }
    }

    public void pieceMovementInfo() {
        if (this.movedPiece == '-') {
            System.out.println("No piece moved, this is the root
node.");
            return;
        }
        System.out.print("Piece: " + this.movedPiece);
        System.out.print(", MoveDistance: " +
Math.abs(this.moveDistance));
        System.out.println(", Direction: " + this.getDirection());
    }

```

```
}
```

IV. Piece.java

```
package kirisame.rush_solver.model;

public class Piece {
    protected char id;
    protected int length;
    protected int axis;
    protected int row;
    protected int col;

    /**
     * @param length length of the piece
     * @param axis    0 for horizontal, 1 for vertical
     * @param row     row position of the piece, Top-Left as reference
     * @param col     column position of the piece, Top-Left as reference
     */
    public Piece(char id, int length, int axis, int row, int col) {
        if (length <= 0) {
            throw new IllegalArgumentException("Length must be a positive integer.");
        }
        if (row < 0 || col < 0) {
            throw new IllegalArgumentException("X and Y coordinates must be non-negative integers.");
        }
        this.id = id;
        this.length = length;
        this.axis = axis;
        this.row = row;
        this.col = col;
    }

    public char getId() {
        return id;
    }

    public int getLength() {
```

```

        return length;
    }

    public int getAxis() {
        return axis;
    }

    public int getRow() {
        return row;
    }

    public int getCol() {
        return col;
    }

    public void printInfo() {
        System.out.println("Id: " + id);
        System.out.println("    Length: " + length);
        System.out.println("    Axis: " + axis);
        System.out.println("    Location: (" + row + ", " + col +
" )");
    }
}

```

V. PrimaryPiece.java

```

package kirisame.rush_solver.model;

public class PrimaryPiece extends Piece{

    /**
     * Constructs a PrimaryPiece object with the specified length and
    position.
     * The piece is initialized with an ID of P.
     */
    public PrimaryPiece(int length, int axis, int row, int col) {
        super('P', length, axis, row, col);
    }
}

```

Controller

I. MainController.java

```
package kirisame.rush_solver.controller;

import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.multipart.MultipartFile;

import kirisame.rush_solver.algorithm.Astar;
import kirisame.rush_solver.algorithm.GBFS;
import kirisame.rush_solver.model.Board;
import kirisame.rush_solver.model.Node;

@RestController
@RequestMapping("/api")
public class MainController {

    @GetMapping("/hello")
    public ResponseEntity<Map<String, String>> hello() {
        Map<String, String> response = new HashMap<>();
        response.put("message", "Hello, From 春ブーツ!");
        return ResponseEntity.ok(response);
    }

    @PostMapping("/solve")
    public ResponseEntity<Map<String, Object>> solve(
        @RequestParam MultipartFile file,
        @RequestParam(value = "algo", defaultValue = "astar")
        String algorithm,
```

```

        @RequestParam(value = "heur", defaultValue = "blocking")
String heuristic) {
    try {
        String content = new String(file.getBytes(),
java.nio.charset.StandardCharsets.UTF_8);

        Board rootBoard = ParserController.readFile(content);
        Node rootNode = new Node(rootBoard, null, 0, heuristic);

        ArrayList<Node> path;
        long executionTime;
        int visitedNodes;

        switch (algorithm.toLowerCase()) {
            case "gbfs" -> {
                GBFS gbfs = new GBFS(rootNode);
                path = gbfs.solve();
                executionTime = gbfs.getExecutionTimeInMillis();
                visitedNodes = gbfs.getNodeCount();
            }
            case "astar" -> {
                Astar astar = new Astar(rootNode);
                path = astar.solve();
                executionTime = astar.getExecutionTimeInMillis();
                visitedNodes = astar.getNodeCount();
            }
            case "ucs" -> {
                rootNode.setHeuristic("ucs");
                Astar astar = new Astar(rootNode);
                path = astar.solve();
                executionTime = astar.getExecutionTimeInMillis();
                visitedNodes = astar.getNodeCount();
            }
            default -> {
                Map<String, Object> errorResponse = new
HashMap<>();

                errorResponse.put("error", "Invalid algorithm.
Supported values: 'astar', 'gbfs', 'ucs'");

                return
ResponseEntity.badRequest().body(errorResponse);
            }
        }
    }
}

```

```

    }

    if (path == null) {
        Map<String, Object> errorResponse = new HashMap<>();
        errorResponse.put("error", "No solution found");
        return
ResponseEntity.status(500).body(errorResponse);
    }

    Map<String, Object> response = new HashMap<>();
    response.put("algorithm", algorithm);
    response.put("heuristic", heuristic);
    response.put("executionTime", executionTime);
    response.put("visitedNodes", visitedNodes);

    // Convert path to list of board states and movements
    List<Map<String, Object>> pathStates = new ArrayList<>();
    for (Node node : path) {
        Map<String, Object> state = new HashMap<>();
        state.put("board", node.getBoard().boardToString());

        // Create movement info object
        if (node.getMovedPiece() != '-') {
            Map<String, Object> movement = new HashMap<>();
            movement.put("piece",
String.valueOf(node.getMovedPiece()));
            movement.put("distance",
Math.abs(node.getMoveDistance()));
            movement.put("direction", node.getDirection());
            state.put("movement", movement);
        }
        pathStates.add(state);
    }
    response.put("path", pathStates);

    return ResponseEntity.ok(response);
} catch (IOException e) {
    Map<String, Object> errorResponse = new HashMap<>();
    errorResponse.put("error", "Failed to parse file: " +
e.getMessage());

```

```

        return ResponseEntity.status(500).body(errorResponse);
    }
}
}

```

II. ParserController.java

```

package kirisame.rush_solver.controller;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.StringReader;
import java.util.ArrayList;
import java.util.HashSet;

import kirisame.rush_solver.model.Board;

public class ParserController {
    /**
     * Parses the given file content representing a board
     configuration and constructs a Board object.
     * The file content should follow a specific format:
     * <ul>
     * <li>The first line contains the board's height and width,
     separated by a space.</li>
     * <li>The second line contains the number of normal
     pieces.</li>
     * <li>The subsequent lines represent the board layout, with
     each character indicating a piece, empty space, or special
     marker.</li>
     * </ul>
     * The method processes the file content line by line, sets up
     the board's size, pieces, and layout,
     * and handles special cases such as the end goal ('K') and empty
     spaces.
     *
     * @param fileContent the string content of the file to parse
     * @return a {@link Board} object representing the parsed board
     configuration, or {@code null} if an error occurs
     */
    public static Board readFile(String fileContent) {

```



```

        try(BufferedReader br = new BufferedReader(new
StringReader(fileContent))) {
            Board rootBoard = new Board();
            int lineCount = 0;
            int normalPieceCount = 0;
            HashSet<Character> pieceIds = new HashSet<>();
            String line;
            ArrayList<String> tempBoard = new ArrayList<>();
            while ((line = br.readLine()) != null) {
                lineCount++;
                switch (lineCount) {
                    case 1 -> {
                        String[] dimensions = line.split(" ");
                        int height = Integer.parseInt(dimensions[0]);
                        int width = Integer.parseInt(dimensions[1]);
                        rootBoard.setSize(height, width);
                    }
                    case 2 -> {
                        normalPieceCount = Integer.parseInt(line);
                        rootBoard.setPieceCount(normalPieceCount);
                    }
                    default -> {
                        tempBoard.add(line);
                    }
                }
            }
            int exitRow = -1, exitCol = -1;
            for (int i = 0; i < tempBoard.size(); i++) {
                line = tempBoard.get(i);
                for (int j = 0; j < line.length(); j++) {
                    if (line.charAt(j) == 'K') {
                        exitRow = i;
                        exitCol = j;
                        break;
                    }
                }
            }
            if (exitRow != -1)
                break;
        }
        if(exitRow==0){

```

```

        // Exit on top row, need padding
        int offset = 1;
        rootBoard.setBoardAt(0, exitCol+offset, 'K');
        for (int i = 1; i < tempBoard.size(); i++) {
            line = tempBoard.get(i);
            for (int j = 0; j < line.length(); j++) {
                rootBoard.setBoardAt(i, j+offset,
line.charAt(j));
            }
        }
    }else if(exitRow==tempBoard.size()-1){
        // Exit on bottom row, need padding
        int offset = 1;
        rootBoard.setBoardAt(rootBoard.getHeight()-1,
exitCol+offset, 'K');
        for (int i = 0; i < tempBoard.size()-1; i++) {
            line = tempBoard.get(i);
            for (int j = 0; j < line.length(); j++) {
                rootBoard.setBoardAt(i+1, j+offset,
line.charAt(j));
            }
        }
    }else if(exitCol==0){
        // Exit on left column, no need padding
        int offset = 0;
        rootBoard.setBoardAt(exitRow+1, 0, 'K');
        for (int i = 0; i < tempBoard.size(); i++) {
            line = tempBoard.get(i);
            for (int j = 1; j < line.length(); j++) {
                rootBoard.setBoardAt(i+1, j+offset,
line.charAt(j));
            }
        }
    }else if(exitCol==rootBoard.getWidth()-2){
        // Exit on right column, no need padding
        int offset = 1;
        rootBoard.setBoardAt(exitRow+1,
rootBoard.getWidth()-2, 'K');
        for (int i = 0; i < tempBoard.size(); i++) {
            line = tempBoard.get(i);

```

```

        for (int j = 0; j < line.length(); j++) {
            rootBoard.setBoardAt(i+1, j+offset,
line.charAt(j));
        }
    }

    rootBoard.parsePieces();
    System.out.println("Piece Info:");
    rootBoard.printPieces();
    System.out.println(rootBoard.boardToString());
    return rootBoard;
} catch (IOException e) {
}
return null;
}
}

```

Algorithm

I. AbstractSearch.java

```

package kirisame.rush_solver.algorithm;

import java.util.ArrayList;
import java.util.Collection;

import kirisame.rush_solver.model.Node;
public abstract class AbstractSearch {
    // Abstract class for search algorithms
    protected ArrayList<Node> path ;
    public abstract ArrayList<Node> solve();

    protected void buildPath(Node goalNode) {
        path = new ArrayList<>();
        while (goalNode != null) {
            path.add(0, goalNode);
            goalNode = goalNode.getParent();
        }
    }
}

```

```

        protected static boolean containsBoard(Collection<? extends Node>
collection, Node compNode) {
            for (Node n : collection) {
                if (n.getBoard().equals(compNode.getBoard())) {
                    return true;
                }
            }
            return false;
        }
    }
}

```

II. AStar.java

```

package kirisame.rush_solver.algorithm;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.PriorityQueue;

import kirisame.rush_solver.model.HNode;
import kirisame.rush_solver.model.Node;

public class Astar extends AbstractSearch {

    // Own Implementation of Prioqueue
    private PriorityQueue<HNode> open;
    private ArrayList<HNode> closed = new ArrayList<>();
    private long executionTime;

    /**
     * Constructs and Solves the problem with class A*
     *
     * @param startBoard
     * @param heuristic
     * @return
     */
    public Astar(Node startNode) {

        Comparator<HNode> comparator =
Comparator.comparingInt(HNode::getF);
    }
}

```

```

        open = new PriorityQueue<>(comparator);

        // Node startNode = new Node(initialBoard.deepCopy(), null,
0, heuristic);
        HNode root = new HNode(startNode);
        open.add(root);

    }

    @Override
    public ArrayList<Node> solve() {

        long startTime = System.nanoTime();

        while (!open.isEmpty()) {
            HNode current = open.poll();

            if (current.getBoard().isGoal()) {
                buildPath(current);
                executionTime = System.nanoTime() - startTime; //
Stop timing
                return path;
            }

            closed.add(current);

            for (Node successor : current.expand()) {

                HNode hSuccessor = new HNode(successor);

                if (containsBoard(open, hSuccessor)) {
                    keepBetterNodeInOpen(hSuccessor);
                } else if (!containsBoard(closed, hSuccessor)) {
                    open.add(hSuccessor);
                }
            }
        }

        System.out.println("No solution found");
        path = null; // No solution found
    }

```

```

        executionTime = System.nanoTime() - startTime; // Stop timing
        return path;
    }

    private void keepBetterNodeInOpen(HNode newNode) {
        // Inefficient PriorityQueue workaround: rebuild without the
worse node
        List<HNode> tempList = new ArrayList<>();
        boolean replaced = false;

        while (!open.isEmpty()) {
            HNode node = open.poll();

            if (!replaced &&
node.getBoard().equals(newNode.getBoard())) {
                if (node.getF() > newNode.getF()) {
                    tempList.add(newNode); // replace with better
node
                } else {
                    tempList.add(node); // keep old one
                }
                replaced = true;
            } else {
                tempList.add(node);
            }
        }

        open.addAll(tempList); // rebuild the queue
    }

    /**
     * get visited nodes count
     *
     * @return the number of visited nodes
     */
    public int getNodeCount() {
        return open.size() + closed.size();
    }

    public long getExecutionTime() {
        return executionTime;
    }

```

```

    }

    public long getExecutionTimeInMillis() {
        return executionTime / 1_000_000; // Convert to milliseconds
    }

    // public ArrayList<Node> getPath() {
    // return path;
    // }
}

```

III. GBFS.java

```

package kirisame.rush_solver.algorithm;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.PriorityQueue;

import kirisame.rush_solver.model.Node;

public class GBFS extends AbstractSearch {
    private PriorityQueue<Node> open;
    private ArrayList<Node> closed;
    private long executionTime;

    public GBFS(Node startNode) {
        Comparator<Node> comparator =
        Comparator.comparingInt(Node::getHeuristicValue);
        open = new PriorityQueue<>(comparator);
        closed = new ArrayList<>();
        open.add(startNode);
    }

    @Override
    public ArrayList<Node> solve() {

        long startTime = System.nanoTime();

        while (!open.isEmpty()) {

```

```

        Node currentNode = open.poll();
        if (currentNode.getDepth() < 3) {

System.out.println(currentNode.getBoard().boardToString());
        }
        closed.add(currentNode);
        if (currentNode.getBoard().isGoal()) {
            System.out.println("Found solution");
            buildPath(currentNode);
            executionTime = System.nanoTime() - startTime; //
Stop timing
            return this.path;
        }
        Node[] children = currentNode.expand();
        for (Node child : children) {
            if (!containsBoard(open, child) &&
!containsBoard(closed, child)) {
                open.add(child);
            }
        }
    }
    System.out.println("No solution found");
    this.path = null; // No solution found
    executionTime = System.nanoTime() - startTime; // Stop timing
    return this.path;
}

/**
 * get visited nodes count
 *
 * @return the number of visited nodes
 */
public int getNodeCount() {
    return open.size() + closed.size();
}

public long getExecutionTime() {
    return executionTime;
}

```



```

    public long getExecutionTimeInMillis() {
        return executionTime / 1_000_000; // Convert to milliseconds
    }
}

```

IV. TwoGreedy.java

```

package kirisame.rush_solver.algorithm;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.PriorityQueue;

import kirisame.rush_solver.model.Node;

public class TwoGreedy extends AbstractSearch {
    // GBFS with 2 polls at the same time
    private PriorityQueue<Node> open;
    private ArrayList<Node> closed;
    private long executionTime;

    public TwoGreedy(Node startNode) {
        Comparator<Node> comparator =
        Comparator.comparingInt(Node::getHeuristicValue);
        open = new PriorityQueue<>(comparator);
        closed = new ArrayList<>();
        open.add(startNode);
    }

    @Override
    public ArrayList<Node> solve() {

```

```

        long startTime = System.nanoTime();

        while (!open.isEmpty()) {
            Node currentNode = open.poll();
            Node secondNode = open.poll();

            if (currentNode != null) {
                closed.add(currentNode);
                if (currentNode.getBoard().isGoal()) {
                    System.out.println("Found solution");
                    buildPath(currentNode);
                    executionTime = System.nanoTime() - startTime; //
Stop timing

                    return this.path;
                }
                Node[] children = currentNode.expand();
                for (Node child : children) {
                    if (!containsBoard(open, child) &&
!containsBoard(closed, child)) {
                        open.add(child);
                    }
                }
            }

            if (secondNode != null) {
                closed.add(secondNode);
                if (secondNode.getBoard().isGoal()) {
                    System.out.println("Found solution");
                    buildPath(secondNode);

```

```

        executionTime = System.nanoTime() - startTime; //
Stop timing

        return this.path;
    }

    Node[] secondChildren = secondNode.expand();
    for (Node child : secondChildren) {
        if (!containsBoard(open, child) &&
!containsBoard(closed, child)) {
            open.add(child);
        }
    }
}

}

System.out.println("No solution found");
this.path = null; // No solution found
executionTime = System.nanoTime() - startTime; // Stop timing
return this.path;
}

/**
 * get visited nodes count
 *
 * @return the number of visited nodes
 */
public int getNodeCount() {
    return open.size() + closed.size();
}

public long getExecutionTime() {

```

```

        return executionTime;
    }

    public long getExecutionTimeInMillis() {
        return executionTime / 1_000_000; // Convert to milliseconds
    }
}

```

App

I. Dockerfile

```

# Build frontend

FROM node:20-alpine AS frontend-build
WORKDIR /app/frontend
COPY frontend/package*.json ./
RUN npm install
COPY frontend/ ./
RUN npm run build

# Build backend

FROM maven:3.9-eclipse-temurin-24-alpine AS backend-build
WORKDIR /app
COPY rush_solver/pom.xml ./
COPY rush_solver/src ./src
RUN mvn package -DskipTests

# Final stage

FROM eclipse-temurin:24-jre-alpine

```

```
WORKDIR /app

# Copy built artifacts

COPY --from=backend-build /app/target/*.jar app.jar
COPY --from=frontend-build /app/frontend/dist ./static

# Expose port

EXPOSE 10005

# Set environment variables

ENV SPRING_PROFILES_ACTIVE=prod

# Run the application

CMD ["java", "-jar", "app.jar"]
```

II. App.tsx

```
import './App.css';
import { useState } from 'react';
import FileUploader from './components/FileUploader';
import PathCarousel from './components/carousel/PathCarousel';

function App() {
  const [solutionData, setSolutionData] = useState<{
    algorithm: string;
    heuristic: string;
    executionTime: number;
    visitedNodes: number;
    path: Array<{
      board: string;
      movement?: {
```

```

        piece: string;
        distance: number;
        direction: string;
    };
    }>;
} | null>(null);
const [error, setError] = useState<string | null>(null);

const handleSolutionFound = (
    data: typeof solutionData,
    err: string | null,
) => {
    setSolutionData(data);
    setError(err);
};

return (
    <div className="flex flex-col bg-black min-h-screen
items-center">
        <div className="my-5 text-amber-400 items-center
text-center">
            <h1 className="text-3xl font-bold">Rush Hour Puzzle
Solver</h1>
            <h1 className="text-amber-400">Gurt: Yo</h1>
        </div>
        <div className="flex min-w-full justify-center
items-center">
            <FileUploader onSolutionFound={handleSolutionFound} />
        </div>
        {error && (

```

```

        <div className="flex text-red-500 items-center
text-center mt-4">
            <p>{error}</p>
        </div>
    )}
    {solutionData && (
        <div className="text-white flex flex-col items-center
">
            <div>
                <p>
                    Found solution with
                    {solutionData.path.length} steps
                </p>
                <p>Time: {solutionData.executionTime}ms</p>
                <p>Nodes visited:
                    {solutionData.visitedNodes}</p>
            </div>
            <PathCarousel
slides={solutionData.path}></PathCarousel>
        </div>
    )}
</div>
);
}

export default App;

```

III. BoardGrid.tsx

```
import React from "react";

interface BoardGridProps {
  board: string;
}

const BoardGrid: React.FC<BoardGridProps> = ({ board }) => {
  // Convert board string into 2D array
  const rows = board.trim().split("\n");
  const numRows = rows.length;
  const numCols = rows[0].length;

  // Map characters to colors
  const getBackgroundColor = (cell: string) => {
    if (
      cell === "P" ||
      cell === "K" ||
      cell === "." ||
      cell === " " ||
      cell === "0"
    )
      return undefined;

    var charCode = cell.charCodeAt(0) - 65;
    var red = ((charCode + 1) * 50) % 256;
    var green = ((charCode + 1) * 230) % 256;
    var blue = ((charCode + 1) * 120) % 256;
    return { backgroundColor: `rgb(${red},${green},${blue})` };
  };
};
```



```

return (
  <div
    className="flex justify-center items-center w-full
max-w-[400px]"
    style={{
      aspectRatio: `${numCols} / ${numRows}`,
    }}
  >
    <div
      className="grid gap-0.5 p-2 bg-gray-900 rounded-lg
w-full h-full"
      style={{
        gridTemplateColumns: `repeat(${numCols}, 1fr)`,
        gridTemplateRows: `repeat(${numRows}, 1fr)`,
      }}
    >
      {rows.map((row, i) =>
        row.split("").map((cell, j) => {
          const isSpecial =
            cell === "P" ||
            cell === "K" ||
            cell === "." ||
            cell === " " ||
            cell === "@";
          let className =
            "aspect-square flex items-center
justify-center rounded transition-colors duration-200 ";
          if (cell === "P") className += "bg-red-500";
          else if (cell === "K") className +=
"bg-yellow-500";

```

```

        else if (cell === "." || cell === " ")
            className += "bg-gray-800";
            else if (cell === "0") className +=
"bg-gray-900";

    return (
        <div
            key={`-${i}-${j}`}
            className={className}
            style={
                !isSpecial
                    ? getBackgroundColor(cell)
                    : undefined
            }
            title={
                cell !== "." &&
                cell !== " " &&
                cell !== "0"
                    ? `Piece ${cell}`
                    : "Empty"
            }
        >
            {cell !== "." &&
                cell !== " " &&
                cell !== "0" && (
                    <span className="flex
items-center justify-center h-full text-white font-bold text-sm">
                        {cell}
                    </span>
                )}
        </div>
    )

```

```

        );
    })),
    )}
    </div>
  </div>
);
};

export default BoardGrid;

```

IV. FileUploader.tsx

```

import { Input } from "@components/ui/input";
import { Label } from "@components/ui/label";
import { useRef } from "react";
import Combobox from "../ComboBox";
import { useAppContext } from "@hooks/AppProvider";

interface FileUploaderProps {
  onSolutionFound?: (
    data: {
      algorithm: string;
      heuristic: string;
      executionTime: number;
      visitedNodes: number;
      path: Array<{
        board: string;
        movement?: {
          piece: string;
          distance: number;

```

```

        direction: string;
    };
    }>;
    } | null,
    err: string | null,
) => void;
}

export default function FileUploader({ onSolutionFound }:
FileUploaderProps) {
    const fileRef = useRef<HTMLInputElement>(null);
    const appContext = useAppContext();

    const handleUpload = async () => {
        const file = fileRef.current?.files?.[0];
        if (!file) {
            onSolutionFound?.(null, "Please select a file.");
            return;
        }
        if (!file.name.toLowerCase().endsWith(".txt")) {
            onSolutionFound?.(null, "Please select a .txt file.");
            return;
        }
        const formData = new FormData();
        formData.append("file", file);
        let url = "/api/solve";
        if (appContext.state.algorithm) {
            url += `?algo=${appContext.state.algorithm}`;
        }
        if (appContext.state.heuristic) {

```

```

        url += `${url.includes("?") ? "&" : ""}heur=${appContext.state.heuristic}`;
    }

    try {
        const res = await fetch(url, {
            method: "POST",
            body: formData,
        });
        const data = await res.json();
        if (res.ok) {
            onSolutionFound?.(data, null);
        } else {
            onSolutionFound?.(null, data.error);
        }
    } catch (err) {
        onSolutionFound?.(
            null,
            "An error occurred while uploading the file.",
        );
    }
};

return (
    <div className="grid w-full max-w-sm items-center gap-1.5">
        <Label className="text-amber-400" htmlFor="picture">
            Puzzle txt file
        </Label>
        <Input id="picture" type="file" ref={fileRef} />
        <div className="flex flex-row justify-center gap-x-2">

```

```

        <Combobox
          param="algorithm"
          options={
            new Map([
              ["A*", "astar"],
              ["GBFS", "gbfs"],
              ["UCS", "ucs"],
            ])
          }
        ></Combobox>

        <div
          className={` transition duration-750
            ${appContext.state.algorithm === "ucs" ?
"pointer-events-none opacity-50" : ""}`}
        >

          <Combobox
            param="heuristic"
            options={
              new Map([
                ["Blocking Pieces", "blocking"],
                ["Distance to Goal", "distance"],
                ["Combined Heuristic",
"blocking_distance"],
              ])
            }
          ></Combobox>

        </div>
      </div>

      <button

```

```
                className="bg-black text-amber-400 text-lg mt-2
transition hover:bg-amber-400 hover:text-black p-2 rounded-md
duration-200"
                type="button"
                onClick={handleUpload}
            >
                Upload & Solve
            </button>
        </div>
    );
}
```



BAB V

HASIL PERCOBAAN

Test Case 1:

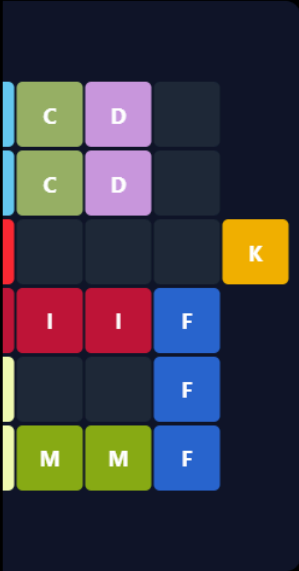
Puzzle txt file

Choose File test1.txt

A*  Blocking Pieces 


Upload & Solve

Found solution with 6 steps
Time: 35ms
Nodes visited: 203



F 3 steps down

ep 5



Move piece P 3 steps right

Step 6

Puzzle txt file

Browse... test1.txt

GBFS



Blocking Pieces

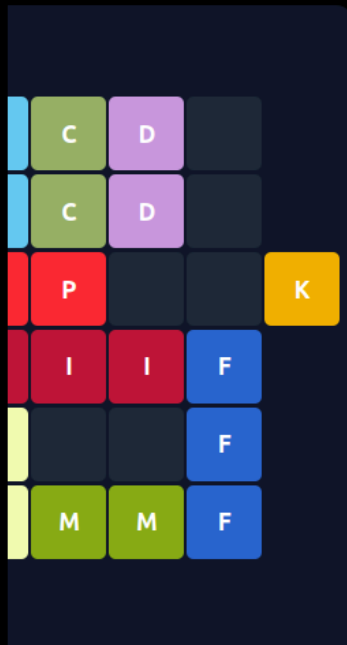


Upload & Solve

Found solution with 7 steps

Time: 40ms

Nodes visited: 355



F 3 steps down

tep 6



Move piece P 2 steps right

Step 7



Start

Puzzle txt file

Browse... test1.txt

UCS



Blocking Pieces

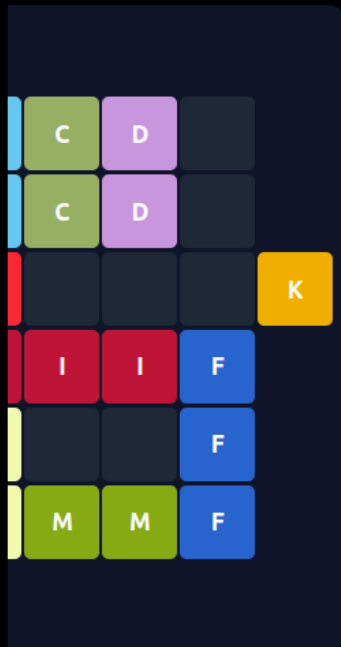


Upload & Solve

Found solution with 6 steps

Time: 112ms

Nodes visited: 477



Move piece D 1 steps up

Step 5



Move piece P 3 steps right

Step 6



Start

Puzzle txt file

Browse... test1.txt

Two Greedy



Blocking Pieces

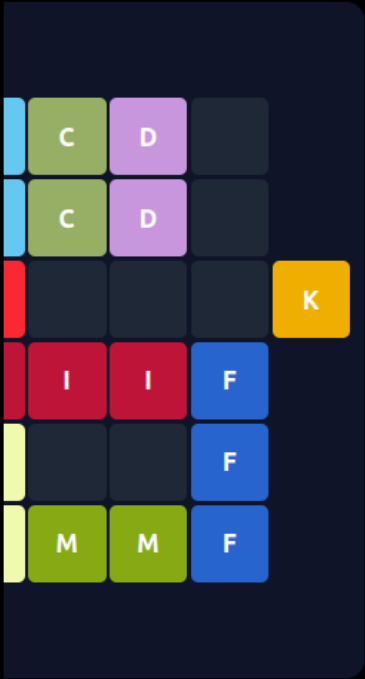


Upload & Solve

Found solution with 6 steps

Time: 7ms

Nodes visited: 58



F 3 steps down

Step 5



Move piece P 3 steps right

Step 6



Start

Test Case 2:

rushsolver.kirisame.jp.net

Utility Coding 日本語 AI tools 筋トレ DS/ML Kuliah WebDev File Storage Akade... So You Want to Tra...

Puzzle txt file

Choose File test2.txt

A* Distance to Goal

Upload & Solve

Found solution with 8 steps
Time: 102ms
Nodes visited: 310

	J	M
F	J	M
F	J	M
	J	M
	J	M
	J	M

A 2 steps left

Step 7

		P			
A	A	P		J	M
B	B		F	J	M
C	C		F	J	M
L	L			J	M
D	E			J	M
D	E			J	M

Move piece P 3 steps up

Step 8

Puzzle txt file

Browse... test2.txt

GBFS



Distance to Goal



Upload & Solve

Found solution with 13 steps

Time: 6ms

Nodes visited: 153



: A 1 steps left

Step 12



Move piece P 2 steps up

Step 13



Start

Puzzle txt file

Browse... test2.txt

UCS



Distance to Goal



Upload & Solve

Found solution with 7 steps

Time: 322ms

Nodes visited: 858



B 1 steps left

Step 6



Move piece P 4 steps up

Step 7



Start

Puzzle txt file

Browse... test2.txt

Two Greedy



Distance to Goal

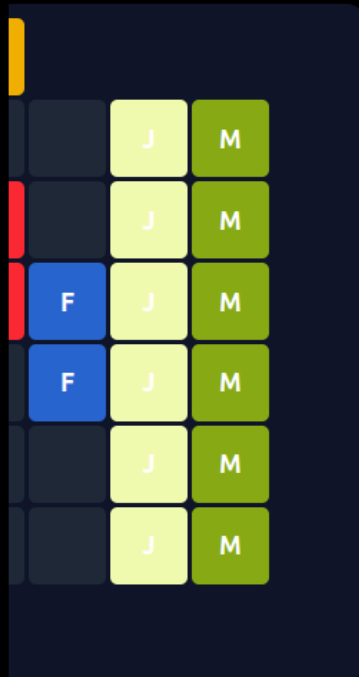


Upload & Solve

Found solution with 37 steps

Time: 71ms

Nodes visited: 522



Move piece A 1 steps left

Step 36



Move piece P 2 steps up

Step 37



Start

Test Case 3:

The screenshot shows a web browser at the URL `rushsolver.kirisame.jp.net`. The browser's address bar and tabs are visible at the top. The page has a dark background with yellow text and buttons. The main heading is "Rush Hour Puzzle Solver" in a large yellow font, with the subtitle "Gurt: Yo" below it. Under the heading, there is a section titled "Puzzle txt file" in yellow. This section contains a file selection input field with the text "Choose File test3.txt", a dropdown menu currently showing "A*", and another dropdown menu labeled "Distance to Goal". Below these inputs is a yellow button labeled "Upload & Solve". At the bottom of the form area, the text "No solution found" is displayed in red.

Utility Coding 日本語 AI tools 筋トレ DS/ML Kuliah WebDev File Storage Akade... So You Want to T

Rush Hour Puzzle Solver

Gurt: Yo

Puzzle txt file

Choose File test3.txt

A*

Distance to Goal

Upload & Solve

No solution found

This is a close-up view of the puzzle solver interface. It features a dark background with yellow text and buttons. The section is titled "Puzzle txt file" in yellow. Below the title is a file selection input field containing the text "Browse... test3.txt". To the left of this field is a dropdown menu showing "UCS", and to the right is a greyed-out dropdown menu labeled "Distance to Goal". Below these elements is a large yellow button with the text "Upload & Solve". At the bottom of the section, the text "No solution found" is written in red.

Puzzle txt file

Browse... test3.txt

UCS

Distance to Goal

Upload & Solve

No solution found

Puzzle txt file

Browse... test3.txt

GBFS



Distance to Goal



Upload & Solve

No solution found

Puzzle txt file

Browse... test3.txt

Two Greedy



Distance to Goal



Upload & Solve

No solution found

Test Case 4:

rushsolver.kirisame.jp.net

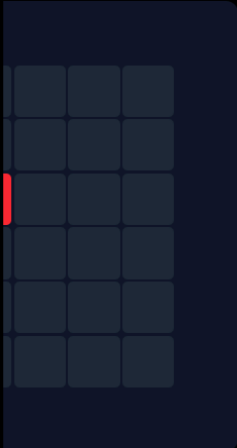
Puzzle txt file

Choose File test4.txt

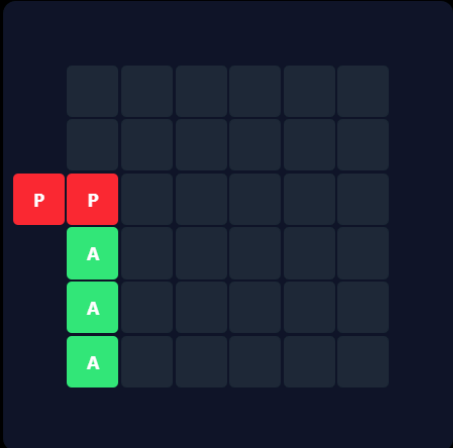
GBFS Combined Heuristic

Upload & Solve

Found solution with 3 steps
Time: 0ms
Nodes visited: 12



A 2 steps down



Move piece P 2 steps left

ep 2 Step 3

The image shows a web browser window with the URL 'rushsolver.kirisame.jp.net'. The browser's address bar and tabs are visible at the top. The main content area has a dark background. Under the heading 'Puzzle txt file', there is a file selection button labeled 'Choose File test4.txt'. Below this are two dropdown menus: 'GBFS' and 'Combined Heuristic'. The section 'Upload & Solve' displays the results: 'Found solution with 3 steps', 'Time: 0ms', and 'Nodes visited: 12'. Two 6x6 grid visualizations show the puzzle state at different steps. The first grid, labeled 'ep 2', shows a red piece 'P' on the left edge and three green pieces 'A' in the second column. The second grid, labeled 'Step 3', shows the red piece 'P' moved two steps left to the first column, with the three green pieces 'A' remaining in the second column. The text 'A 2 steps down' is below the first grid, and 'Move piece P 2 steps left' is below the second grid. At the bottom of each grid, the step number 'ep 2' and 'Step 3' are displayed in yellow.

Puzzle txt file

Browse... test4.txt

A*



Combined Heuristic

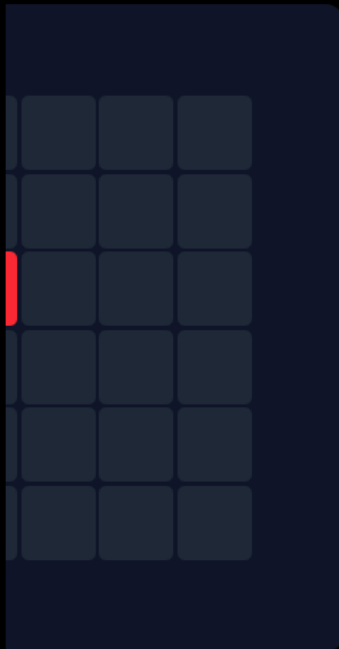


Upload & Solve

Found solution with 3 steps

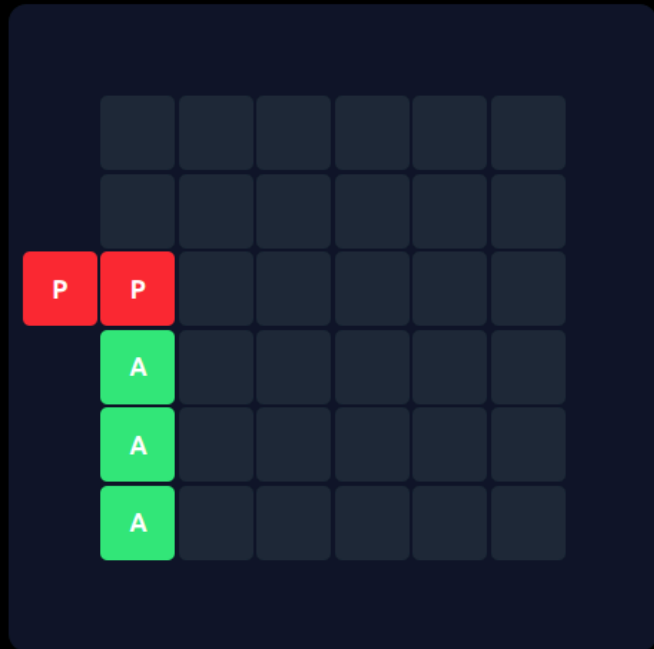
Time: 0ms

Nodes visited: 11



A 2 steps down

ep 2



Move piece P 2 steps left

Step 3



Start

Puzzle txt file

Browse... test4.txt

UCS

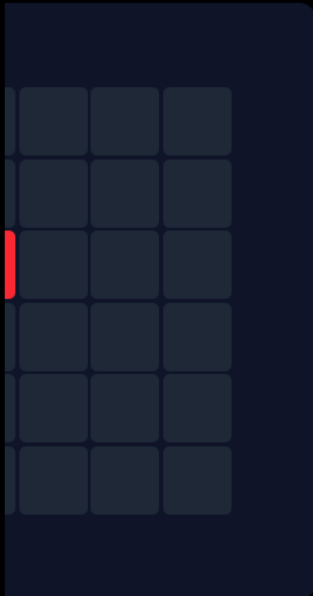
Combined Heuristic

Upload & Solve

Found solution with 3 steps

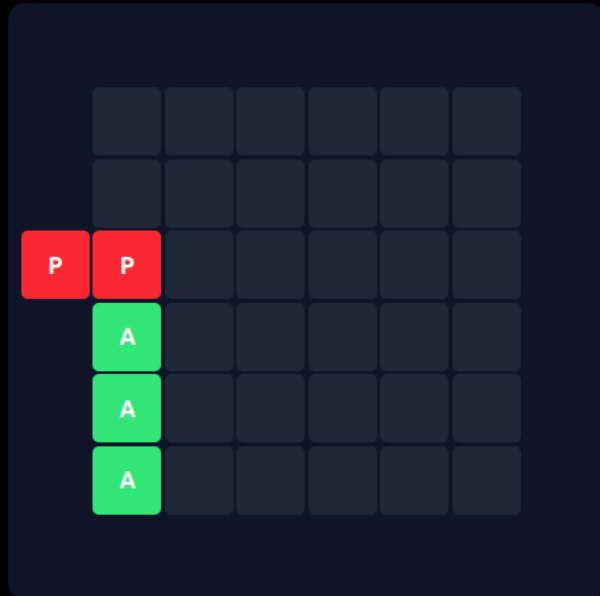
Time: 0ms

Nodes visited: 17



A 2 steps down

ep 2



Move piece P 2 steps left

Step 3



Start

Puzzle txt file

Browse... test4.txt

Two Greedy



Combined Heuristic

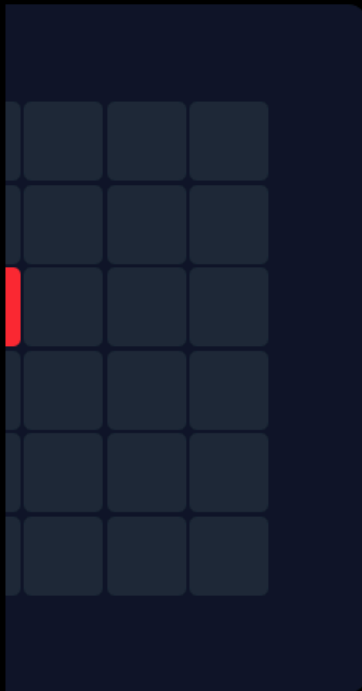


Upload & Solve

Found solution with 3 steps

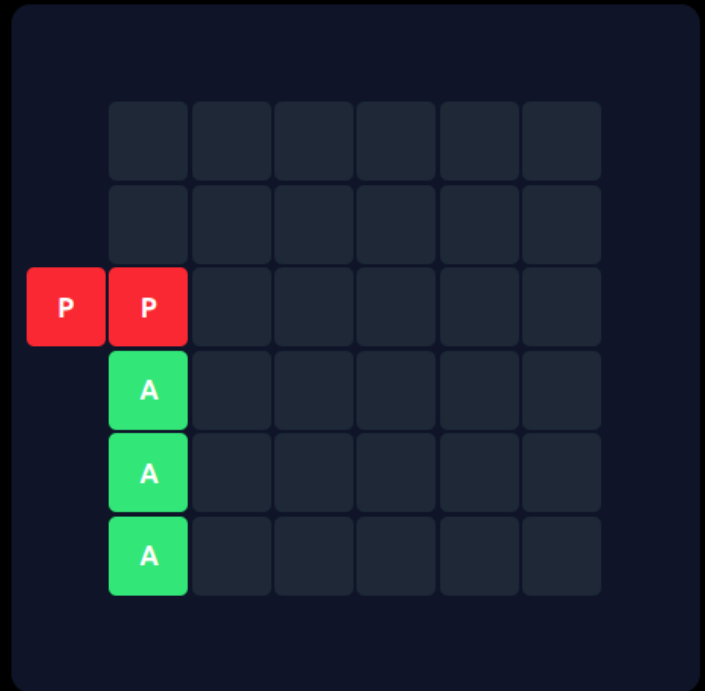
Time: 0ms

Nodes visited: 15



A 2 steps down

Step 2



Move piece P 2 steps left

Step 3



Start

Test Case 5:


rushsolver.kirisame.jp.net

Choose File test5.txt

UCS Select heuristic

Upload & Solve

Found solution with 9 steps
Time: 2ms
Nodes visited: 13



Step 8: B 1 steps left

Step 9: Move piece P 1 steps down

Puzzle txt file

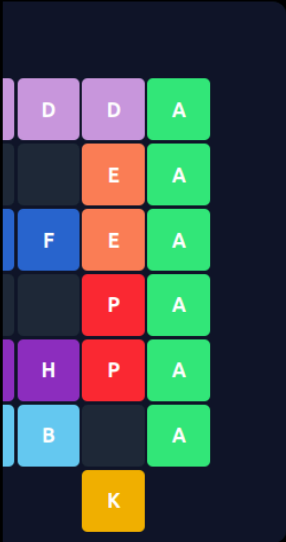
Browse... test5.txt

A*

Combined Heuristic

Upload & Solve

Found solution with 9 steps
Time: 2ms
Nodes visited: 13



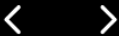
B 1 steps left

Step 8



Move piece P 2 steps down

Step 9



Start

Puzzle txt file

Browse... test5.txt

GBFS



Combined Heuristic

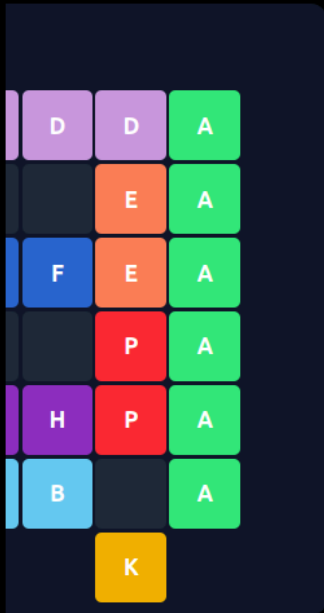


Upload & Solve

Found solution with 9 steps

Time: 1ms

Nodes visited: 13



B 1 steps left

Step 8



Move piece P 2 steps down

Step 9



Start

Puzzle txt file

Browse... test5.txt

Two Greedy



Combined Heuristic

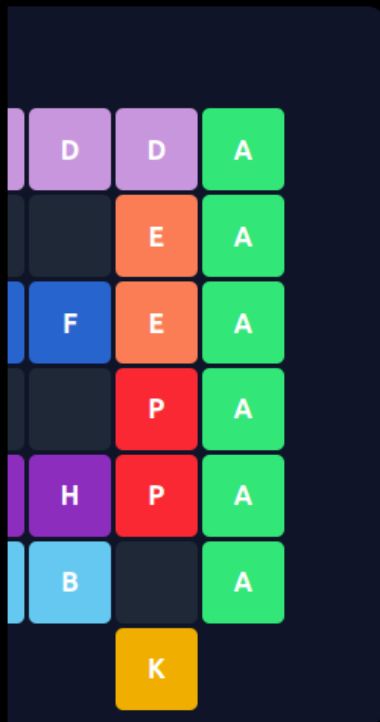


Upload & Solve

Found solution with 9 steps

Time: 2ms

Nodes visited: 17



Move piece B 1 steps left

Step 8



Move piece P 2 steps down

Step 9



Start

Test Case 6:

rushsolver.kirisame.jp.net

Choose File test6.txt

A* Combined Heuristic

Upload & Solve

Found solution with 7 steps
Time: 3ms
Nodes visited: 60

	C	D	F	
	C	D	F	
	P	P		K
	I	I	I	

ep 6

Step 7

Move piece P 2 steps right

A	H	H	C	D	F	
A	B	B	C	D	F	
G					P	P
G			I	I	I	

< > Start

Puzzle txt file

Browse... test6.txt

GBFS

Combined Heuristic

Upload & Solve

Found solution with 8 steps

Time: 5ms

Nodes visited: 53



Move piece F 1 steps up

Step 7



Move piece P 2 steps right

Step 8



Start

Puzzle txt file

Browse... test6.txt

UCS

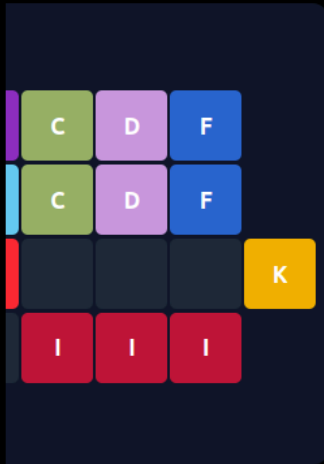
Combined Heuristic

Upload & Solve

Found solution with 6 steps

Time: 22ms

Nodes visited: 107



Move piece D 1 steps up

Step 5



Move piece P 4 steps right

Step 6



Start

Puzzle txt file

Browse... test6.txt

Two Greedy



Combined Heuristic

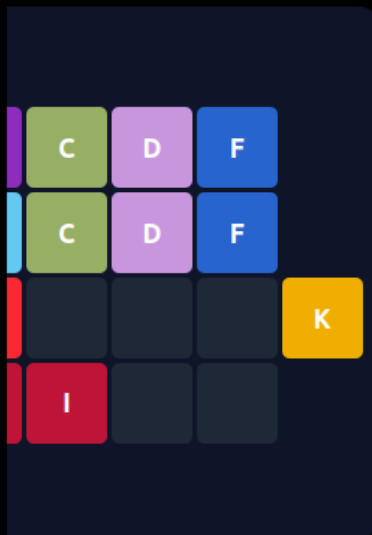


Upload & Solve

Found solution with 7 steps

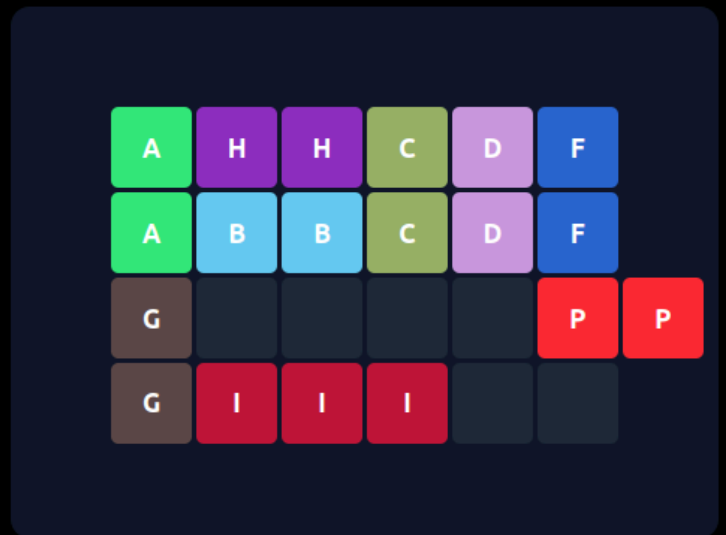
Time: 6ms

Nodes visited: 69



Move piece F 1 steps up

Step 6



Move piece P 4 steps right

Step 7



Start

Test Case 7:

Choose File test7.txt

GBFS

Blocking Pieces

Upload & Solve

Found solution with 4 steps
Time: 2ms
Nodes visited: 80

D	F	
D	F	
I	I	
E	E	
H	H	

I 1 steps right

A	A			
B		D	F	
B		D	F	
		I	I	
	P	E	E	
	P	H	H	
	K			

Move piece P 3 steps down

Puzzle txt file

Browse... test7.txt

A*

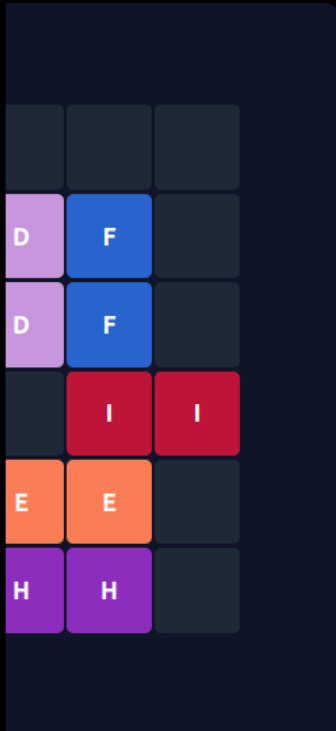
Blocking Pieces

Upload & Solve

Found solution with 4 steps

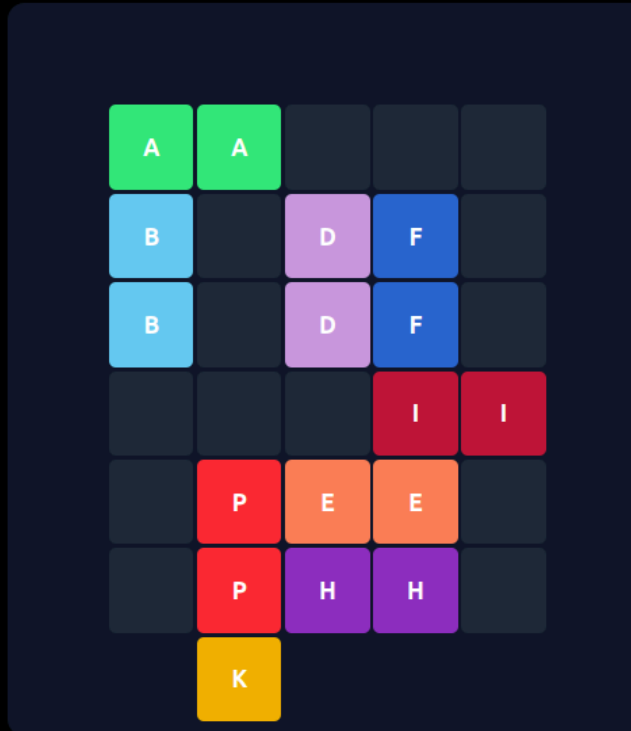
Time: 15ms

Nodes visited: 226



I 2 steps right

Step 3



Move piece P 3 steps down

Step 4



Start

Puzzle txt file

Browse... test7.txt

UCS

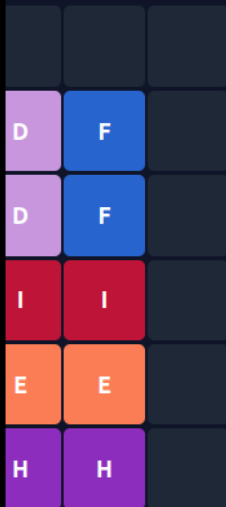
Blocking Pieces

Upload & Solve

Found solution with 4 steps

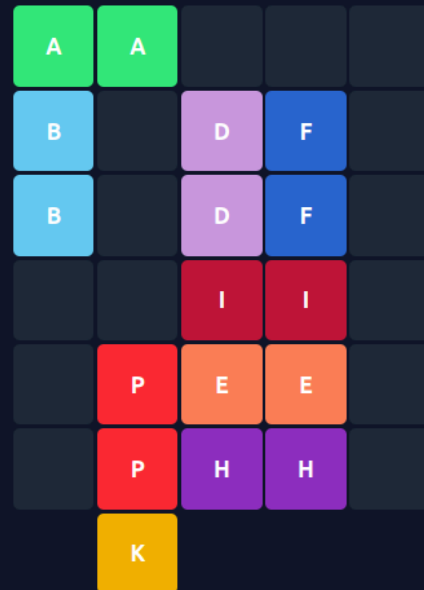
Time: 270ms

Nodes visited: 776



I 1 steps right

Step 3



Move piece P 3 steps down

Step 4



Start

Puzzle txt file

Browse... test7.txt

Two Greedy



Blocking Pieces

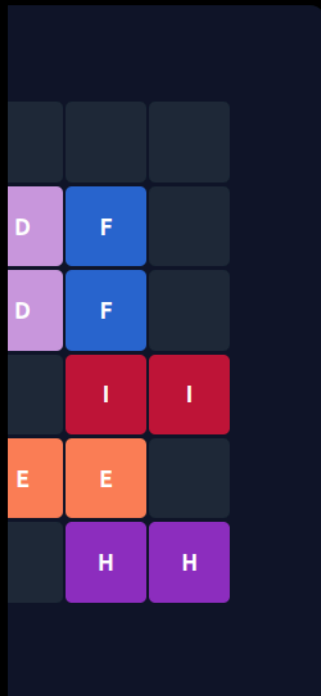


Upload & Solve

Found solution with 4 steps

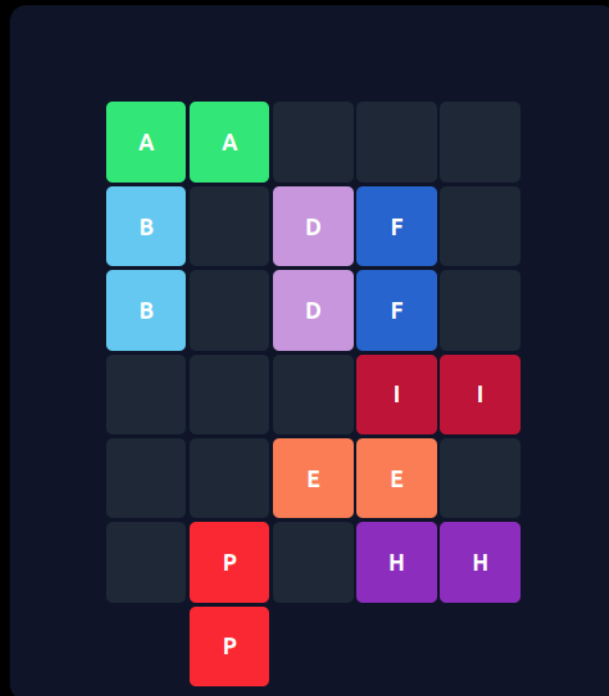
Time: 2ms

Nodes visited: 111



H 3 steps right

Step 3



Move piece P 4 steps down

Step 4



Start

BAB VI

ANALISIS PERCOBAAN

Berdasarkan hasil percobaan didapatkan bahwa algoritma pathfinding UCS merupakan algoritma paling tidak efisien dari ketiga algoritma lainnya dengan node count paling banyak dari semua percobaan. Hal tersebut terjadi karena UCS memilih node dengan biaya terkecil untuk di expand dan biaya node pada algoritma UCS merupakan depth node relatif terhadap root node tree solusi, sehingga penyelesaian UCS sama dengan menggunakan BFS. Algoritma A* dan GBFS tidak selalu konsisten efisiensinya. Terdapat beberapa kasus dimana satu diantara kedua algoritma lebih efisien. Hal ini terjadi karena heuristik yang telah dibuat tidak admissible sehingga A* dapat overestimate pencariannya sehingga solusi menjadi kurang efisien dan GBFS menjadi lebih efisien dalam beberapa kasus tertentu.

Berikut merupakan analisa time complexity setiap algoritma pathfinding pada permasalahan rush hour:

- *Greedy Best First Search:*
 - Worst case: $O(b^d)$, b merupakan branching factor atau average node count yang di-expand oleh successor, d merupakan depth tree solusi
 - Heuristik yang tidak konsisten akan menghasilkan jumlah depth yang lebih besar dari jumlah steps yang diperlukan untuk solusi terpendek
- *Uniform Cost Search:*
 - Worst & Average case: $O(b^C)$, b merupakan branching factor atau average node count yang di-expand oleh successor, C merupakan jumlah steps yang diperlukan untuk mendapatkan solusi terpendek
- A*:
 - Worst case: $O(b^C)$, b merupakan branching factor atau average node count yang di-expand oleh successor, C merupakan jumlah steps yang diperlukan untuk mendapatkan solusi terpendek
 - Namun, algoritma A* dapat menjadi lebih efisien secara waktu dibandingkan UCS apabila heuristic admissible
- TwoGreedy (Custom):
 - Worst case: $O(b^d)$, b merupakan branching factor atau average node count yang di-expand oleh successor, d merupakan depth tree solusi.
 - Space Complexity: sama seperti GBFS, yaitu $O(b^d)$

- Sama seperti GBFS karena walaupun mengambil dua node, secara asimtotik tidak akan berpengaruh.

Perlu ditambahkan bahwa $\text{space complexity} \approx \text{time complexity}$ untuk ketiga algoritma utama pada permasalahan ini.

BAB VII

IMPLEMENTASI BONUS

Terdapat tiga heuristik yang diimplementasikan pada aplikasi. pertama heuristik “blocking” yang menghitung berapa banyak mobil yang menghalang atau membatasi *primary piece* untuk mencapai *goal*. Kedua heuristik “distance” yang menghitung jarak *primary piece* posisi sekarang hingga mencapai *goal*. Terakhir heuristik “blocking_distance” yang merupakan kombinasi antara kedua heuristik “blocking” dan heuristik “distance”.

Juga terdapat satu algoritma pencarian tambahan yaitu TwoGreedy yang merupakan varian GBFS yang lebih agresif sehingga dapat mengurangi waktu pencarian dan jumlah node yang dikunjungi.

GUI dibuat dengan arsitektur web React + Vite + Tailwind untuk *frontend* yang di-*serve* oleh *backend* Spring Boot. Animasi untuk alur solusi berbentuk *slideshow* dari tiap langkah yang diambil. Visualisasi papan dilakukan dengan Grid yang ditampilkan menggunakan *library* Embla Carousel dan menyediakan tombol maju-mundur dan juga tombol *autoplay*. Setiap piece diberi warna yang berbeda agar lebih mudah terdiferensiasi.

Website ini juga dideploy di <https://rushsolver.kirisame.jp.net/>

BAB VII

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 kelompok	✓	

LINK GITHUB: https://github.com/kirisame-ame/Tucil3_13523006_13523032