TUGAS KECIL 2

SOLVER PUZZLE RUSH HOUR MENGGUNAKAN ALGORITMA PATHFINDING

IF-2211

STRATEGI ALGORITMA

13523103 Steven Owen Liauw

13523109 Haegen Quinston

Dosen Pengampu:

Dr. Nur Ulfa Maulidevi, S.T, M.Sc.

Dr. Ir. Rinaldi Munir, M.T.

Monterico Adrian, S.T, M.T.

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

JL. GANESA 10, BANDUNG 40132

2025

DAFTAR ISI

DAFTAR ISI 2
BAB 1 PENJELASAN ALGORITMA 3
1.1 Prinsip/Kesamaan Umum 3
1.2 Uniform Cost Search 6
1.3 Greedy Best First Search 7
1.4 A* Search 7
1.5 Beam Search 7
BAB 2 ANALISIS ALGORITMA 8
2.1 Definisi f(n) dan g(n) 8
2.2 Algoritma A* 8
2.3 Algoritma UCS 8
2.4 Efektivitas Algoritma 9
BAB 3 SOURCE CODE PROGRAM 9
3.1 Arsitektur Program 9
3.2 Tabel Screenshot Source Code Program 9
BAB 4 PERCOBAAN 23
4.1 Tabel Percobaan Menggunakan UCS 23
4.2 Tabel Percobaan Menggunakan Greedy BFS 25
4.3 Tabel Percobaan Menggunakan A* Search 27
4.4 Tabel Percobaan Menggunakan Beam Search 29
4.5 Tabel Percobaan Tambahan 31
BAB 5 ANALISIS HASIL PERCOBAAN 33
5.1 Analisis Percobaan 33
5.2 Kompleksitas Algoritma 34
Uniform Cost Search (UCS) 34
Greedy Best-First Search (GBFS) 34
A★ Search 35
Beam Search (algoritma bonus) 35
BAB 6 IMPLEMENTASI BONUS 35
6.1 Algoritma Tambahan 35
6.2 Heuristik Tambahan 36
Count Direct Blockers (Heuristik Dasar) 36
Count Recursive Blockers (Heuristik Bonus #1) 37
Count Min Steps (Heuristik Bonus #2) 37
6.3 GUI 38
LAMPIRAN 38
Repository GitHub 38
Tabel Checklist 38
Referensi Tambahan 39

BAB 1 PENJELASAN ALGORITMA

1.1 Prinsip/Kesamaan Umum

Sebelum membahas varian, semua algoritma pathfinding yang digunakan mengikuti pola dasar:

Inisialisasi

- Bangun *start node* dengan $g = 0$, hitung h via fungsi heuristic, lalu $f = g + h$.
- Siapkan *open list* (PriorityQueue atau beam array) dan *closed set* (optional) untuk mendeteksi duplikat.

Loop Utama

- Ambil node terbaik dari open list (berdasarkan kriteria masing-masing algoritma).
- Cek apakah node ini *goal*; jika ya, hentikan dan rekonstruksi path via pointer parent.
- Tandai node sebagai ***visited*** dengan menambahkan serialization-nya ke closed set atau peta $state \rightarrow best\_g$.
- *Generate* semua tetangga (getNeighbors()): clone papan, geser tiap mobil satu langkah, bangun Node baru dengan $parent = current,\ g = current.g + cost$, hitung h dan f.
- Untuk setiap tetangga, jika belum pernah atau ditemukan g lebih kecil, enqueue ke open list; kalau menggunakan closed set, abaikan yang duplikat/kombinasi dengan g lebih besar.

Rekonstruksi Path

Rekonstruksi dimulai dari node goal, ikuti pointer parent mundur ke start, kumpulkan move, lalu balik urutan.

Pseudocode Umum

```
BEGIN GENERIC_SEARCH
  INPUT startBoard, heuristic, searchType, beamWidth (optional)
  OUTPUT resultNode, expansions
  FUNCTION CompareByG(a, b):
    RETURN a.g - b.g
  FUNCTION CompareByH(a, b):
    IF a.h < b.h THEN RETURN -1
```

```
    ELSE IF a.h > b.h THEN RETURN 1
    ELSE RETURN 0
  END FUNCTION
  FUNCTION CompareByF(a, b):
    IF a.f < b.f THEN RETURN -1
    ELSE IF a.f > b.f THEN RETURN 1
    ELSE RETURN 0
  END FUNCTION
  SET expansions ← 0
  SET resultNode ← null
   SET startNode ← CREATE Node(startBoard, parent = null, move = null,
heuristic)
  IF searchType = 1 THEN
    SET openQueue ← PRIORITY_QUEUE with comparator CompareByG
    CALL openQueue.enqueue(startNode)
  ELSE IF searchType = 2 THEN
    SET openQueue ← PRIORITY_QUEUE with comparator CompareByH
    CALL openQueue.enqueue(startNode)
  ELSE IF searchType = 3 THEN
    SET openQueue ← PRIORITY_QUEUE with comparator CompareByF
    CALL openQueue.enqueue(startNode)
  ELSE IF searchType = 4 THEN
    SET beam ← [startNode]
  END IF
  SET explored ← EMPTY SET
  WHILE (searchType ≠ 4 AND openQueue is not empty) OR (searchType = 4 AND
beam is not empty) DO:
    IF searchType ≠ 4 THEN
```

```
    SET current ← CALL openQueue.dequeue()
    INCREMENT expansions
    IF current.isGoal() THEN
      SET resultNode ← current
      RETURN
    END IF
    SET key ← current.serialize()
    CALL explored.add(key)
    SET neighbors ← current.getNeighbors()
    FOR EACH node IN neighbors DO:
      SET key ← node.serialize()
      IF NOT explored.contains(key) THEN
        CALL explored.add(key)
        CALL openQueue.enqueue(node)
      END IF
    END FOR
  ELSE
    SET successors ← EMPTY LIST
    FOR EACH node IN beam DO:
      INCREMENT expansions
      IF node.isGoal() THEN
        SET resultNode ← node
        RETURN
      END IF
      CALL explored.add(node.serialize())
      SET neighbors ← node.getNeighbors()
      FOR EACH neighbor IN neighbors DO:
```

```
          SET key ← neighbor.serialize()
          IF NOT explored.contains(key) THEN
            CALL successors.add(neighbor)
          END IF
        END FOR
      END FOR
      IF successors is empty THEN
        BREAK
      END IF
      CALL successors.sortBy(CompareByF)
      IF LENGTH(successors) > beamWidth THEN
        SET beam ← FIRST beamWidth elements of successors
      ELSE
        SET beam ← successors
      END IF
    END IF
  END WHILE
  SET resultNode ← null
END GENERIC_SEARCH
```

1.2 Uniform Cost Search

Uniform Cost Search (UCS) adalah algoritma *pathfinding* secara *uninformed* yang menelusuri ruang keadaan dengan selalu memilih node berbiaya terkecil $g(n)$. Karena selalu mengutamakan cost-so-far (cost keadaan secara real time), UCS akan menemukan solusi dengan total cost minimal asalkan semua edge cost ≥ 0.

Ciri khas dari implementasi UCS adalah:

- Comparator PQ: Mengurutkan berdasarkan g (dan mengabaikan h).

- Best-g map: Menyimpan map {stateString → g_terbaik} untuk memproses jalur yang lebih murah terlebih dahulu.

1.3 Greedy Best First Search

Greedy Best First Search (Greedy BFS) adalah algoritma *pathfinding* yang melakukan pemilihan berbasis heuristik. Greedy BFS selalu memilih node dengan nilai $h(n)$ terkecil, berharap "mendekati" goal lebih cepat. Menggunakan algoritma ini tidak menjamin optimalitas atau kelengkapan, tetapi seringkali sangat cepat bila heuristic penuh dengan insight yang informatif.

Ciri khas dari implementasi Greedy BFS adalah:

- Comparator PQ: Mengurutkan berdasarkan h.
- Visited set awal: Menandai state sesaat sebelum enqueue untuk mencegah loop tak terhingga.
- Tanpa update g: g hanya dipakai untuk rekonstruksi path, bukan ordering.

1.4 A* Search

A* search memadukan cost-so-far $g(n)$ dan heuristic $h(n)$ dengan memilih node ber-$f(n) = g(n) + h(n)$ terkecil. Jika h admissible (dimana selalu berlaku $h(n) \leq g(n)$) & konsisten, A* lengkap dan menemukan solusi minimal serta optimal.

Ciri khas dari implementasi A* search adalah:

- Comparator PQ: Mengurutkan berdasarkan f.
- Re-opening: Apabila ditemukan rute baru ke state yang sama dengan g lebih rendah, re-enqueue node tersebut.
- Consistency check: Memastikan $h(n) \leq c(n, m) + h(m)$ untuk semua edge.

1.5 Beam Search

Beam Search adalah variasi dari A* Search dimana memory pada proses pencarian dibatasi dengan hanya menyimpan k node terbaik (beam) tiap level. Pada tiap iterasi, generate semua tetangga, sort berdasarkan $f(n) = g(n) + h(n)$ terkecil, lalu potong menjadi beamWidth terbesar. Beam Search tidak lengkap dan memiliki kemungkinan untuk tidak optimal sesuai konstanta pembatasan k, namun keunggulannya adalah efisiensi memori.

Ciri khas dari implementasi Beam Search adalah:

- Beam array, bukan PQ: pada tiap iterasi kumpulkan semua tetangga, sort berdasarkan f, lalu beam = $successors.slice(0,\ k)$ (melakukan slicing dengan sisa sebanyak k).
- Parameter beamWidth ditentukan di main.js/CLI—nilai kecil mempercepat tapi mudah terjebak, nilai besar mendekati A*.

- Kondisi berhenti: beam kosong → no solution; goal ditemukan di tengah iterasi → langsung reconstruct path.

BAB 2 ANALISIS ALGORITMA

2.1 Definisi f(n) dan g(n)

Definisi dari g(n) adalah biaya kumulatif dari simpul akar (root) hingga simpul ke-n. Jika simpul awal diberi biaya 0, maka setiap kali kita melakukan satu langkah (misalnya menggeser sebuah mobil di Rush Hour sejauh satu sel), kita menambah biaya sebesar 1. Dengan demikian,

$$g(n) = \sum_{i=1}^{k} cost(langkah\ i).$$

Dalam implementasi Uniform Cost Search (UCS), nilai $g(n)$ inilah satu-satunya kriteria untuk memprioritaskan simpul di frontier: simpul dengan g terkecil diambil lebih dulu.

Sementara, $f(n)$ adalah kombinasi antara biaya sejauh ini $g(n)$ dan perkiraan biaya tersisa $h(n)$. Dengan kata lain,

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

Dalam A*, yang disimpan di priority queue bukan hanya $g(n)$, melainkan $f(n)$. Artinya, simpul dengan nilai f terkecil (yakni yang diperkirakan paling murah totalnya ke tujuan) akan diekspansi lebih dulu.

2.2 Algoritma A*

Berdasarkan materi yang telah dipelajari pada kuliah, heuristik yang admissible adalah yang memenuhi $h(n) \leq h^*(n)$, di mana $h^*(n)$ adalah biaya sebenarnya terendah dari *n* ke tujuan. Pada implementasi A* di program ini, heuristik default menggabungkan dua komponen:

- Jarak (distance): jumlah langkah minimum mobil utama harus bergerak menuju pintu keluar, tanpa mempertimbangkan mobil lain.
- Penghalang (blockers): jumlah mobil yang memblokir jalur langsung antara mobil utama dan pintu keluar, dengan asumsi setiap mobil setidaknya perlu digeser satu kali.

Karena kombinasi kedua komponen ini tidak pernah lebih dari biaya minimum sejati, maka heuristik yang digunakan memenuhi syarat *admissible*.

2.3 Algoritma UCS

Algoritma UCS mengekspansi simpul berdasarkan g(n), yaitu total biaya kumulatif dari root hingga n. Di puzzle Rush Hour, setiap pergeseran satu sel berbiaya 1. Oleh karena itu UCS akan mengeksplorasi layer demi layer (cost = 0, 1, 2, …), persis sama urutannya dengan Breadth-First Search (BFS) ketika semua langkah memiliki biaya yang sama.

2.4 Efektivitas Algoritma

A* menggunakan nilai $f(n) = g(n) + h(n)$ untuk memprioritaskan simpul yang diperkirakan paling murah menuju tujuan. Dengan heuristik admissible, A* menghindari eksplorasi simpul-simpul yang jelas tidak mengarah ke solusi optimal. Secara teoritis, A* lebih efisien daripada UCS (atau Dijkstra) dalam hal jumlah ekspansi simpul pada puzzle Rush Hour.

Sementara, algoritma Greedy BFS hanya memprioritaskan berdasarkan $h(n)$ dan mengabaikan $g(n)$. Pendekatan ini sering kali lebih cepat menemukan sebuah solusi, tetapi karena tidak memperhitungkan biaya yang sudah dikeluarkan, tidak menjamin solusi tersebut adalah yang paling optimal.

BAB 3 SOURCE CODE PROGRAM

3.1 Arsitektur Program

Program kami dikembangkan dengan menggunakan JavaScript untuk bagian backend, serta TypeScript, HTML, dan CSS untuk frontend, yang dibangun dengan framework React dan Tailwind CSS. Seluruh kode frontend dan backend disimpan di dalam folder src. Di dalam src terdapat dua subfolder utama frontend dan backend, yang masing-masing berisi kode untuk laman web lokal yang dikembangkan. Di luar kedua folder tersebut, terdapat lima file penting, yaitu:

- board.js: Berisi definisi kelas Board serta berbagai metode yang mengelola status dan operasi pada papan permainan.
- priorityqueue.js: Mengimplementasikan kelas PriorityQueue, yaitu struktur data antrian prioritas yang digunakan dalam algoritma pencarian.
- node.js: Mendefinisikan kelas Node yang merepresentasikan suatu state dalam pencarian, termasuk referensi ke node induk dan instance Board yang terkait.
- search.js: Mengimplementasikan fungsi-fungsi pencarian solusi dengan memanfaatkan kelas Node dan PriorityQueue sebagai dasar mekanisme pencarian berbasis graf.
- main.js: Merupakan entry point program yang mengatur input-output, memanggil fungsi pencarian dari search.js, serta mengelola proses rekonstruksi dan penampilan solusi.

Sebagai rangkuman, dependency utama program ini adalah sebagai berikut: kelas Board dan PriorityQueue merupakan komponen fundamental, dimana Board merepresentasikan state permainan yang sedang dicari solusinya, dan PriorityQueue berfungsi sebagai struktur data antrian yang terurut berdasarkan prioritas. Kelas Node memanfaatkan metode dari Board untuk mendapatkan node anak dalam ruang pencarian. Selanjutnya, file search.js menggabungkan implementasi Node dan PriorityQueue untuk membangun metode pencarian solusi yang efisien. Semua modul ini kemudian diekspor dan digabungkan dalam main.js, dimana program utama melakukan handling I/O, menjalankan algoritma pencarian, serta membentuk rantai langkah solusi untuk ditampilkan kepada pengguna.

3.2 Tabel Screenshot Source Code Program

Nama File	Screenshot Penting Source Code

board.js	 Konstruktor beserta metode addCar(), placeCars() pada board yang fundamental

```
export class Car {

    constructor(id, row, col, length, orientation, isTarget = false) {
        this.id = id;
        this.row = row;
        this.col = col;
        this.length = length;
        this.orientation = orientation;
        this.isTarget = isTarget;
    }
}

export class Board {

    constructor(height, width, cars = [], exitPos, exitOrientation) {
        this.height = height;
        this.width = width;
        this.cars = cars;
        this.exit = exitPos;
        this.exitOrientation = exitOrientation;
        this.grid = this._createGrid();
        this._placeCars();
    }

    _createGrid() {
        return Array.from({ length: this.height }, () =>
        Array.from({ length: this.width }, () => null)
        );
    }

    _placeCars() {
        this.grid = this._createGrid();
        this.cars.forEach(({ id, row, col, length, orientation }) => {
            for (let offset = 0; offset < length; offset++) {
                const r = orientation === 'V' ? row + offset : row;
                const c = orientation === 'H' ? col + offset : col;
                this.grid[r][c] = id;
            }
        });
    }

    addCar(car) {
        this.cars.push(car);
        this._placeCars();
```

```
export class Board {
    countDirectBlockers() {
                if (this.grid[row][c]) blockers++;
            }
        }
        else {
            exitIdx  = this.exit.row;
            frontIdx = exitIdx < 0
                ? target.row
                : target.row + target.length - 1;

            distance = Math.abs(exitIdx - frontIdx);

            const col = target.col;
            const minR = Math.min(frontIdx, exitIdx);
            const maxR = Math.max(frontIdx, exitIdx);
            for (let r = minR + 1; r < maxR; r++) {
                if (this.grid[r][col]) blockers++;
            }
        }
        return distance + (blockers * 3);
    }

    countRecursiveBlockers() {
        const base = this.countDirectBlockers();

        const target = this.cars.find(c => c.isTarget);
        const burdenCache = new Map();
        const burden = this.carBlockers(target.id, burdenCache);

        return base + burden;
    }

    countMinSteps() {
        const target = this.cars.find(c => c.isTarget);

        let frontIdx, exitIdx, distance = 0;
        let blockers = [];

        if (this.exitOrientation === 'H') {
            exitIdx  = this.exit.col;
            frontIdx = exitIdx < 0
                ? target.col
                : target.col + target.length - 1;
```

```
export class Board {
    countMinSteps() {

            distance = Math.abs(exitIdx - frontIdx);

            const row = target.row;
            const minC = Math.min(frontIdx, exitIdx);
            const maxC = Math.max(frontIdx, exitIdx);

            for (let c = minC + 1; c < maxC; c++) {
                if (this.grid[row][c]) {
                    const car = this.cars.find(c => c.id === this.grid[row][c]);
                    if (car) {
                        blockers.push(car.id);
                    }
                }
            }
        }
        else {
            exitIdx  = this.exit.row;
            frontIdx = exitIdx < 0
                ? target.row
                : target.row + target.length - 1;

            distance = Math.abs(exitIdx - frontIdx);

            const col = target.col;
            const minR = Math.min(frontIdx, exitIdx);
            const maxR = Math.max(frontIdx, exitIdx);
            for (let r = minR + 1; r < maxR; r++) {
                if (this.grid[r][col]){
                    const car = this.cars.find(c => c.id === this.grid[r][col]);
                    if (car) {
                        blockers.push(car.id);
                    }
                }
            }
        }
        let movesNeeded = 0;
        for (const id of blockers) {
            const car = this.cars.find(c => c.id === id);
            movesNeeded += this.minStepsToFree(car);
        }
        return distance + movesNeeded;
```

Fungsi-fungsi di atas merupakan 3 implementasi heuristik yang berbeda.

```
serialize() {
    return this.cars
    .map(c => `${c.id}:${c.row},${c.col}`)
    .sort()
    .join('/');
}

isGoal() {
    const target = this.cars.find(c => c.isTarget);
    if (!target) return false;

    if (this.exitOrientation === 'H') {
        const exitCol = this.exit.col;
        const targetEndCol = target.col + target.length - 1;
        if (target.row !== this.exit.row) return false;

        if (exitCol < 0) {
            return target.col === 0;
        } else if (exitCol >= this.width) {
            return targetEndCol === this.width - 1;
        }
    } else {
        const exitRow = this.exit.row;
        const targetEndRow = target.row + target.length - 1;
        if (target.col !== this.exit.col) return false;

        if (exitRow < 0) {
            return target.row === 0;
        } else if (exitRow >= this.height) {
            return targetEndRow === this.height - 1;
        }
    }
    return false;
}
```

Fungsi serialize() dan isGoal() dibuat dalam implementasi perbandingan dengan duplicate state dan goal state.

	`` getSuccessorMoves() { const result = []; this.cars.forEach(car => { for (let step = -1; ; step--) { if (!this.canMove(car, step)) break; const next = this.clone(); next.moveCar(car.id, step); result.push({ board: next, move: { id: car.id, delta: step } }); } for (let step = 1; ; step++) { if (!this.canMove(car, step)) break; const next = this.clone(); next.moveCar(car.id, step); result.push({ board: next, move: { id: car.id, delta: step } }); } }); return result; } ``` Fungsi getSuccessorMoves() dibuat untuk mencari seluruh kemungkinan suksesor dari board/node.

node.js	 Class Node beserta konstruktor dan seluruh metodenya.

priorityqueue.js

```
export class PriorityQueue {
    constructor(comparator = (a, b) => a - b) {
        this._heap = [];
        this._comparator = comparator;
    }

    enqueue(value) {
        this._heap.push(value);
        this._heapifyUp(this._heap.length - 1);
    }

    dequeue() {
        if (this.isEmpty()) {
        throw new Error('PriorityQueue is empty');
        }
        this._swap(0, this._heap.length - 1);
        const value = this._heap.pop();
        this._heapifyDown(0);
        return value;
    }

    peek() {
        if (this.isEmpty()) {
        return null;
        }
        return this._heap[0];
    }

    isEmpty() {
        return this._heap.length === 0;
    }

    size() {
        return this._heap.length;
    }

    _heapifyUp(index) {
        let parent = Math.floor((index - 1) / 2);
        while (
        index > 0 &&
        this._comparator(this._heap[index], this._heap[parent]) < 0
        ) {
        this._swap(index, parent);
        index = parent;
```

```
export class PriorityQueue {
    _heapifyUp(index) {
        parent = Math.floor((index - 1) / 2);
        }
    }

    _heapifyDown(index) {
        const last = this._heap.length - 1;
        while (true) {
        let left = index * 2 + 1;
        let right = index * 2 + 2;
        let smallest = index;
        if (
            left <= last &&
            this._comparator(this._heap[left], this._heap[smallest]) < 0
        ) {
            smallest = left;
        }
        if (
            right <= last &&
            this._comparator(this._heap[right], this._heap[smallest]) < 0
        ) {
            smallest = right;
        }
        if (smallest === index) break;
        this._swap(index, smallest);
        index = smallest;
        }
    }

    _swap(i, j) {
        [this._heap[i], this._heap[j]] = [this._heap[j], this._heap[i]];
    }
}
```

Kelas Priority Queue

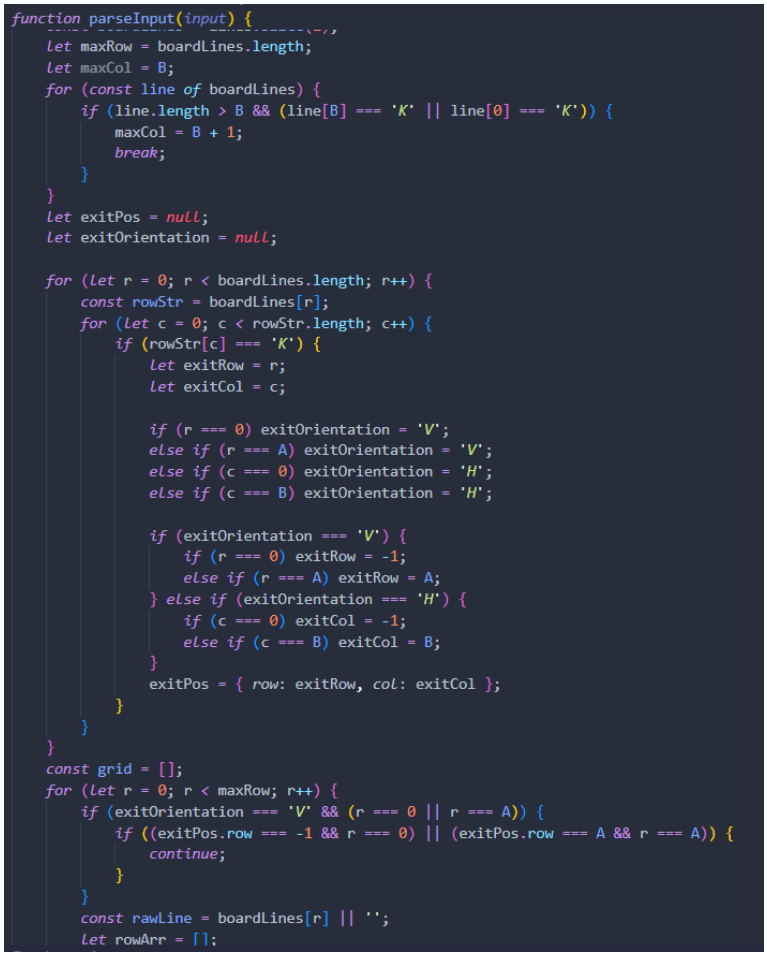
search.js	``` export function aStarSearch(startBoard, heuristic = 1) { const start = new Node(startBoard, null, null, heuristic); const pq = new PriorityQueue((a, b) => a.f - b.f); pq.enqueue(start); const explored = new Set(); let expansions = 0; while (!pq.isEmpty()) { const current = pq.dequeue(); expansions++; const key = current.serialize(); if (explored.has(key)) continue; if (current.isGoal()) { return { node: current, expansions }; } explored.add(key); for (const nbr of current.getNeighbors()) { const nbrKey = nbr.serialize(); if (!explored.has(nbrKey)) { pq.enqueue(nbr); } } } return { node: null, expansions }; } ``` Struktur umum fungsi searching pada search.js (ditampilkan A* sebagai contoh). Queue secara umum bekerja dengan prosedur yang sama, hanya saja heuristik yang digunakan berbeda-beda.

main.js	

```
import fs from 'fs';
import { Board, Car } from './board.js';
import { uniformCostSearch, greedyBestFirstSearch, aStarSearch, beamSearch} from './search.js';
import { error } from 'console';
import { parse } from 'path';

const inputFile = process.argv[2];
const algoArg = process.argv[3];
const heurArg = process.argv[4];

if (!inputFile) {
  console.error('Usage: node main.js <inputfile.txt>');
  process.exit(1);
}

const fileContent = fs.readFileSync(inputFile, 'utf-8');

function parseInput(input) {
    const lines = input.trim().split('\n').map(line => line.replace(/\r$/, ''));
    const headerRe = /^\s*(\d+)\s+(\d+)\s*$/;
    const m = headerRe.exec(lines[0]);
    if (!m) {
        throw new Error(`Baris 1 harus "A B", tapi anda menginput: "${lines[0]}"`);
    }
    let [A, B] = lines[0].split(' ').map(Number);

    const singleIntRe = /^(\d+)$/;
    if (!singleIntRe.test(lines[1].trim())) {
        throw new Error(`Baris 2 harus integer tunggal, tapi anda menginput: "${lines[1]}"`);
    }
    const N = Number(lines[1]);

    var rawBoard = lines.slice(2);
    const boardLinesValidate = rawBoard.map(row => row.trim());
    let exitOrientationValidate = null;
    for (let r = 0; r < rawBoard.length; r++) {
        const row = rawBoard[r];
        for (let c = 0; c < row.length; c++) {
            if (row[c] === 'K') {
                if (r === 0 || r === A) exitOrientationValidate = 'V';
                else if (c === 0 || c === B) exitOrientationValidate = 'H';
            }
        }
    }
```

```
function parseInput(input) {

    const expectedRows = exitOrientationValidate === 'V' ? A + 1 : A;
    if (boardLinesValidate.length !== expectedRows) {
        throw new Error(`Diperlukan tepat ${expectedRows} baris konfigurasi papan, tapi ada ${boardLinesValidate.length}.`);
    }

    const validCharRe = /^[\.A-PR-ZK]+$/;
    for (let i = 0; i < boardLinesValidate.length; i++) {
        const row = boardLinesValidate[i];
        if (exitOrientationValidate === 'V' && (i === 0 || i === boardLinesValidate.length - 1)) {
            if (row !== 'K' && i == boardLinesValidate.length - 1) {
                throw new Error(`Baris ${i + 3} untuk exit vertikal harus hanya 'K', tapi: "${row}"`);
            }
            if(row == 'K') break;
        }
        const hasExit = row.includes('K');
        const expectedLen = hasExit ? B + 1 : B;
        if (row.length !== expectedLen) {
            throw new Error(
                `Baris konfigurasi ke-${i + 3} panjangnya harus ${expectedLen}` + `${hasExit ? ` (termasuk 'K')` : ''}, tapi ${row.length}.`
            );
        }
        if (!validCharRe.test(row)) {
            throw new Error(`Karakter tidak valid di baris ${i + 3}: "${row}".` + ` Hanya diperbolehkan '.', 'P', 'K', dan huruf lainnya.`);
        }
    }

    var cKvertical;
    if(exitOrientationValidate === 'V') {
        const firstRow  = rawBoard[0];
        const lastrRow = rawBoard[rawBoard.length - 1];
        if(firstRow.includes('K')){
            cKvertical = firstRow.indexOf('K');
        }
        else if(lastrRow.includes('K')){
            cKvertical = lastrRow.indexOf('K');
        }
    }

    const pCoords = [];
    const exitCoords = [];
    for (let r = 0; r < boardLinesValidate.length; r++) {
        const row = boardLinesValidate[r];
```

Bagian dari fungsi parseInput() yang merupakan fungsi pengecekan/security input pengguna.



```
function parseInput(input) {
    let maxRow = boardLines.length;
    let maxCol = B;
    for (const line of boardLines) {
        if (line.length > B && (line[B] === 'K' || line[0] === 'K')) {
            maxCol = B + 1;
            break;
        }
    }
    let exitPos = null;
    let exitOrientation = null;

    for (let r = 0; r < boardLines.length; r++) {
        const rowStr = boardLines[r];
        for (let c = 0; c < rowStr.length; c++) {
            if (rowStr[c] === 'K') {
                let exitRow = r;
                let exitCol = c;

                if (r === 0) exitOrientation = 'V';
                else if (r === A) exitOrientation = 'V';
                else if (c === 0) exitOrientation = 'H';
                else if (c === B) exitOrientation = 'H';

                if (exitOrientation === 'V') {
                    if (r === 0) exitRow = -1;
                    else if (r === A) exitRow = A;
                } else if (exitOrientation === 'H') {
                    if (c === 0) exitCol = -1;
                    else if (c === B) exitCol = B;
                }
                exitPos = { row: exitRow, col: exitCol };
            }
        }
    }
    const grid = [];
    for (let r = 0; r < maxRow; r++) {
        if (exitOrientation === 'V' && (r === 0 || r === A)) {
            if ((exitPos.row === -1 && r === 0) || (exitPos.row === A && r === A)) {
                continue;
            }
        }
        const rawLine = boardLines[r] || '';
        let rowArr = [];
```

Bagian parsing input dari parseInput()

```
const algorithm = parseInt(algoArg, 10);
const heuristic = parseInt(heurArg, 10);
const board = parseInput(fileContent);

const startTime = Date.now();
let solutionNode = null;
let expansions = 0;

switch (algorithm) {
    case 2: {
        const result = greedyBestFirstSearch(board, heuristic);
        solutionNode = result.node;
        expansions   = result.expansions;
        break;
    }
    case 3: {
        const result = aStarSearch(board, heuristic);
        solutionNode = result.node;
        expansions   = result.expansions;
        break;
    }
    case 4: {
        const result = beamSearch(board, 1000, heuristic);
        solutionNode = result.node;
        expansions   = result.expansions;
        break;
    }
    case 1:
    default: {
        const result = uniformCostSearch(board, heuristic);
        solutionNode = result.node;
        expansions   = result.expansions;
        break;
    }
}

const endTime = Date.now();
const elapsedTime = endTime - startTime;
```

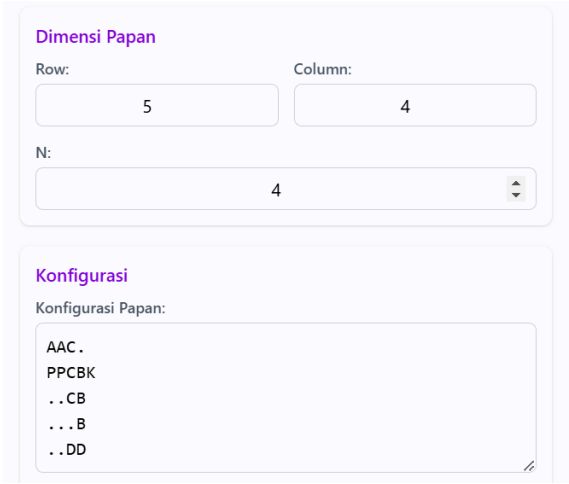
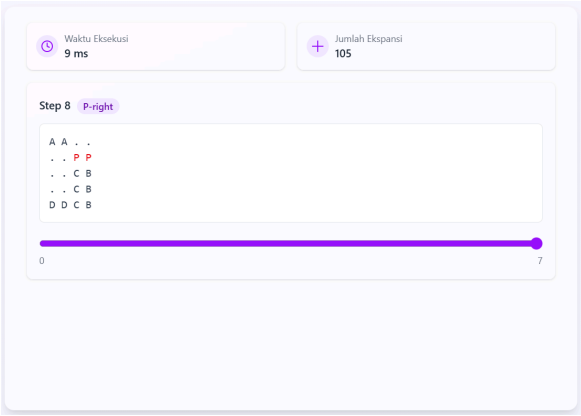
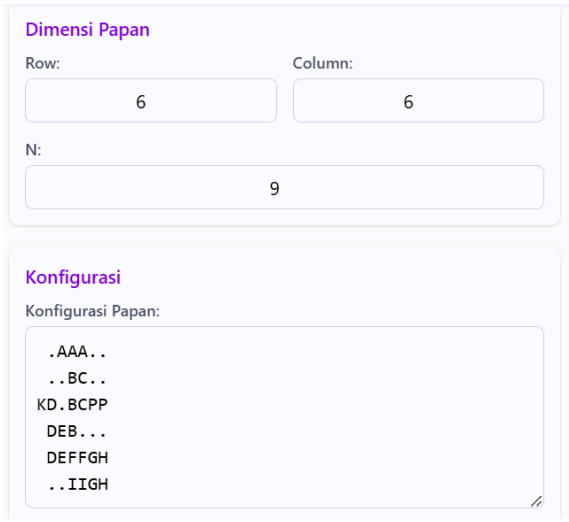
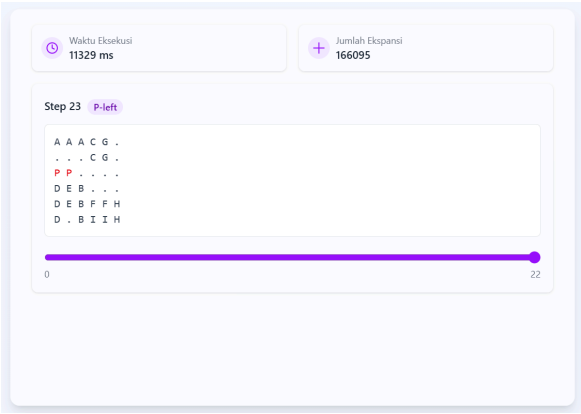
Bagian dari main.js yang mengimplementasikan fungsi searching berdasarkan input pengguna.

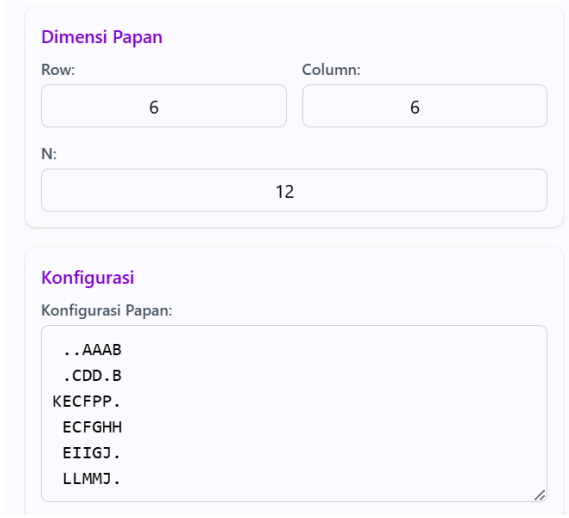
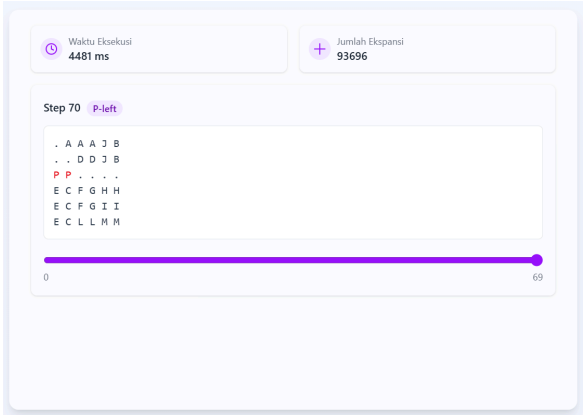
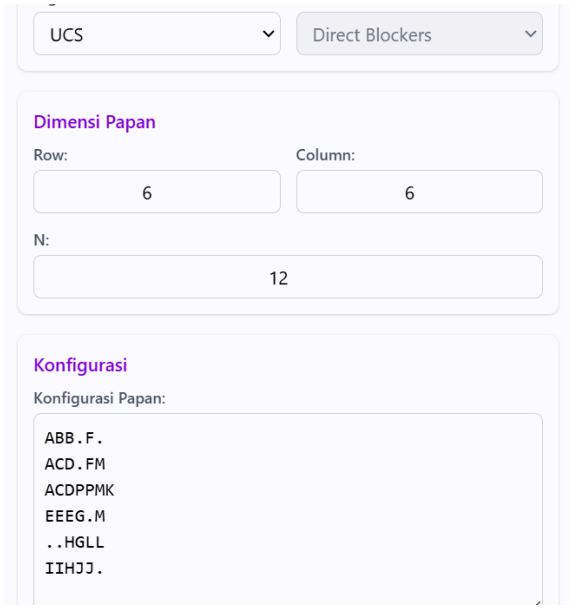
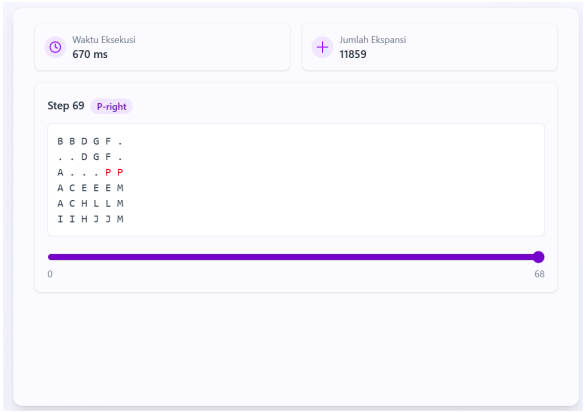
	 Bagian dari program yang mengembalikan time, expansions, solution ke pemanggil API.

BAB 4 PERCOBAAN

4.1 Tabel Percobaan Menggunakan UCS

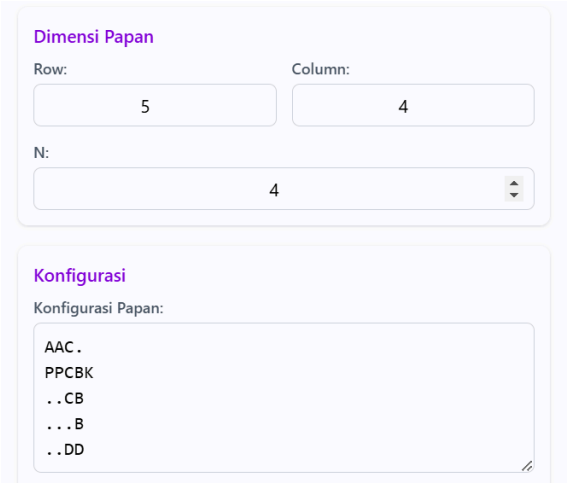
No.	Input	Output

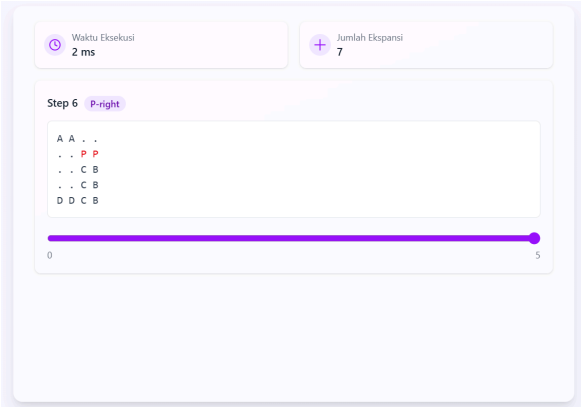
1	 Input *easy* level	
2	 Input *medium* level	

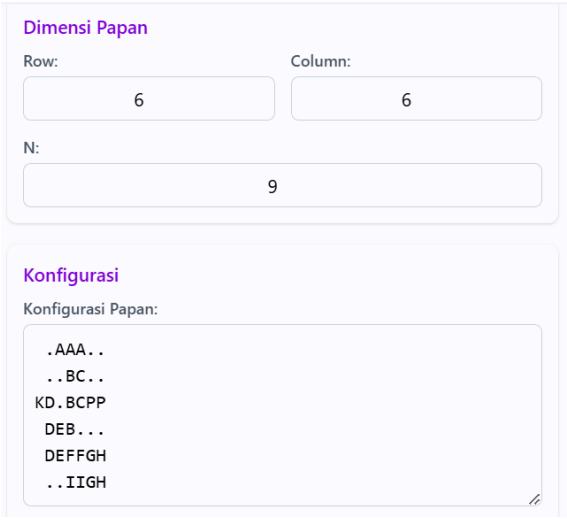
3	 Input *hard* level	
4	 Input *hard level*	

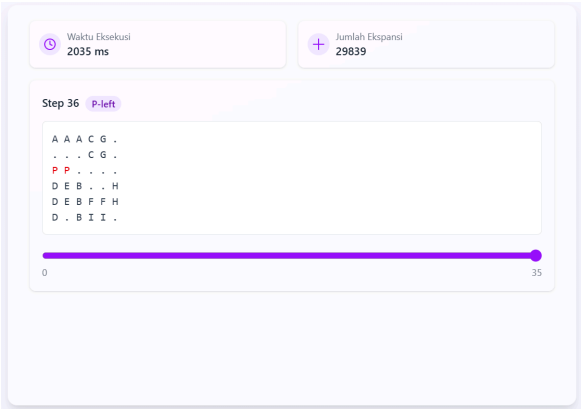
4.2 Tabel Percobaan Menggunakan Greedy BFS

No.	Input	Output

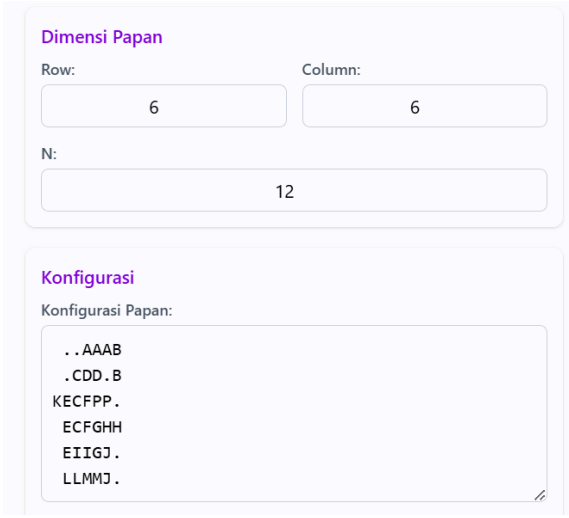
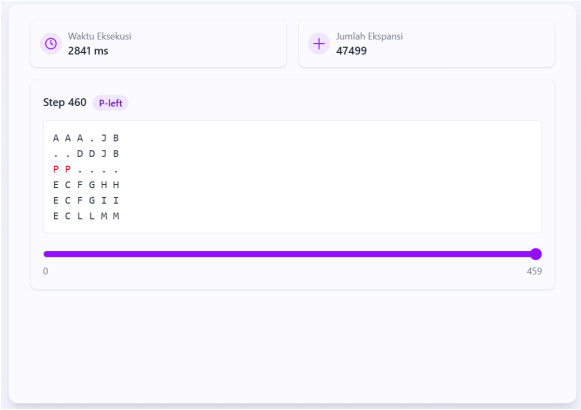
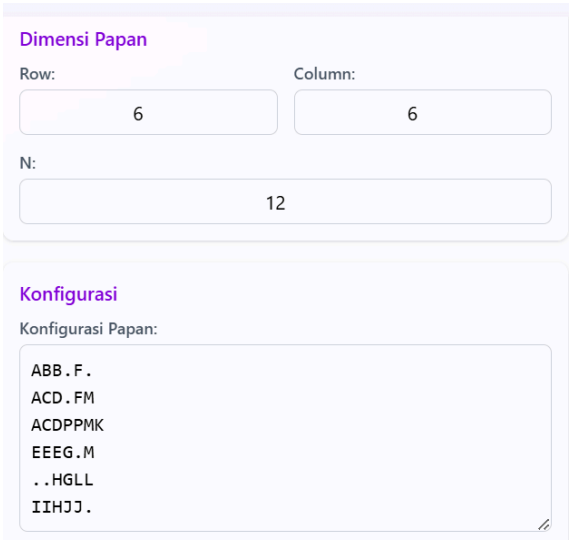
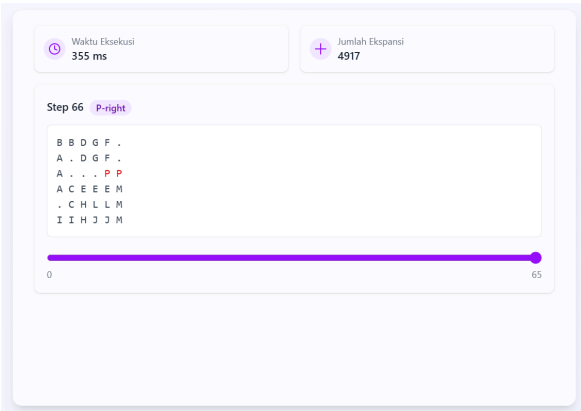
<table>
<tr>
<td>1</td>
<td>


Input easy level, menggunakan heuristik dasar.
</td>
<td>


</td>
</tr>
<tr>
<td>2</td>
<td>


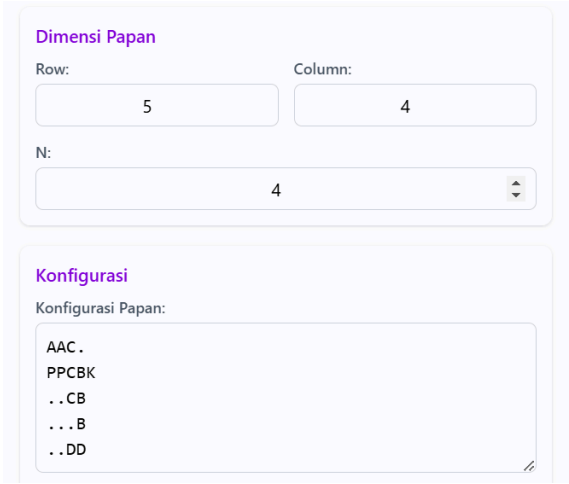
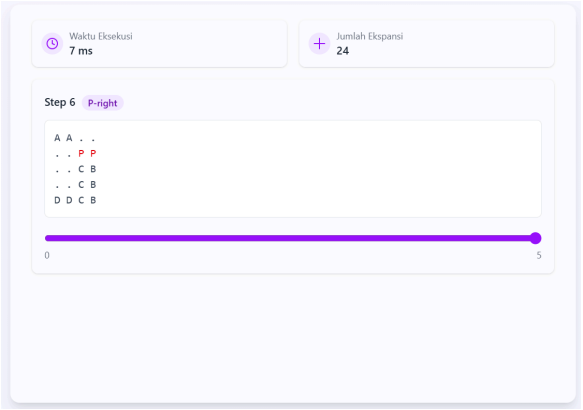
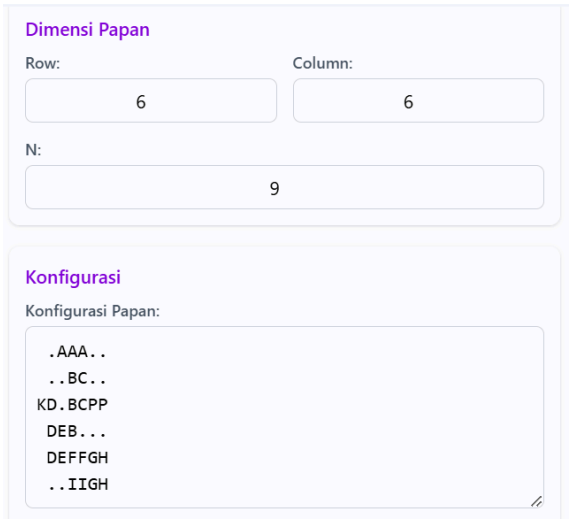
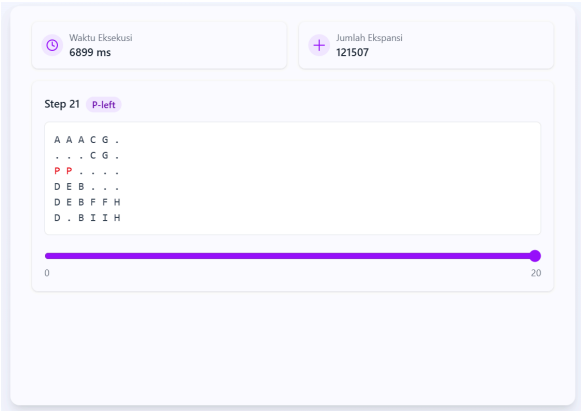
Input medium level menggunakan heuristik Recursive Blockers
</td>
<td>


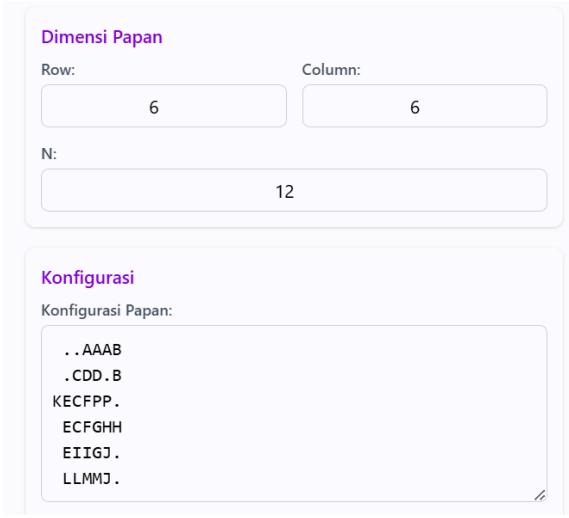
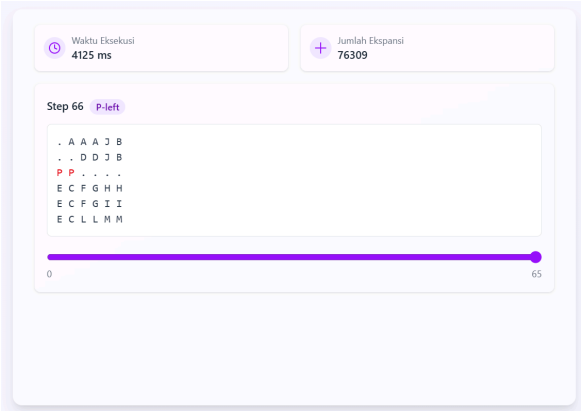
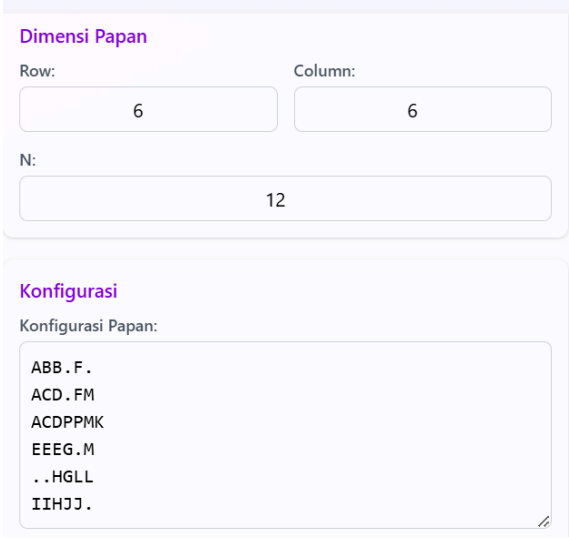
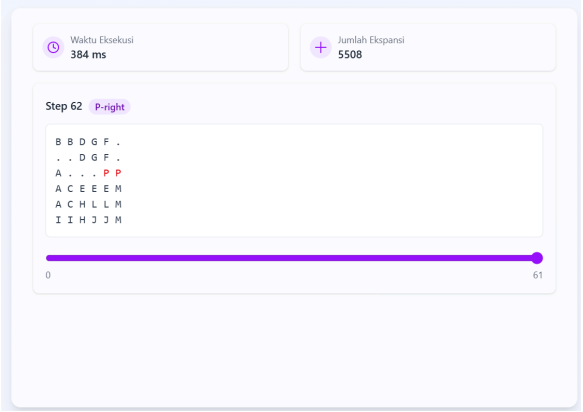
</td>
</tr>
</table>

3	 Input hard level menggunakan heuristik MinSteps	
4	 Input *hard level* dengan heuristik Recursive Blockers.	

4.3 Tabel Percobaan Menggunakan A* Search

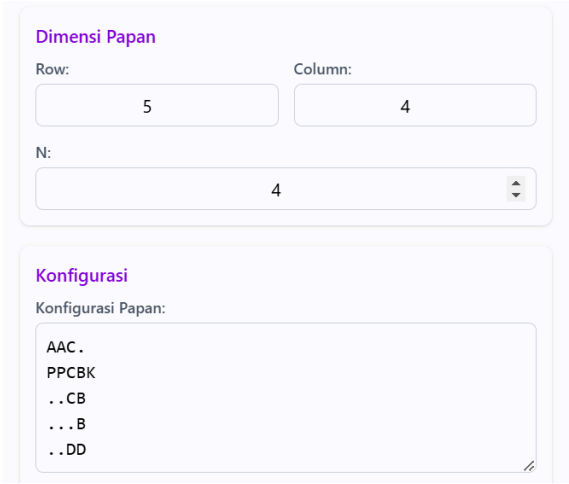
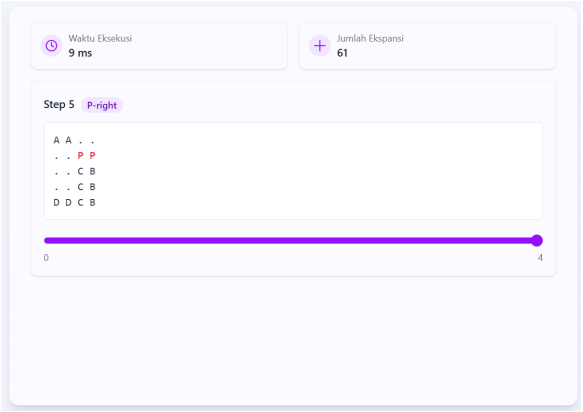
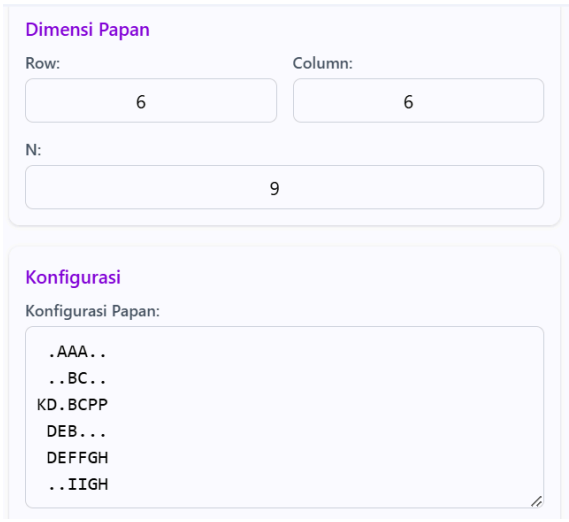
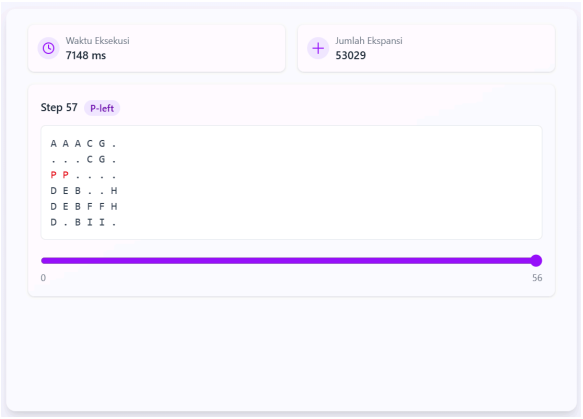
No.	Input	Output

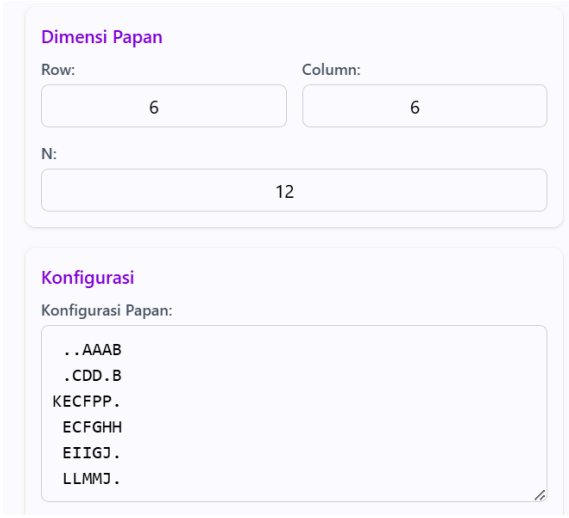
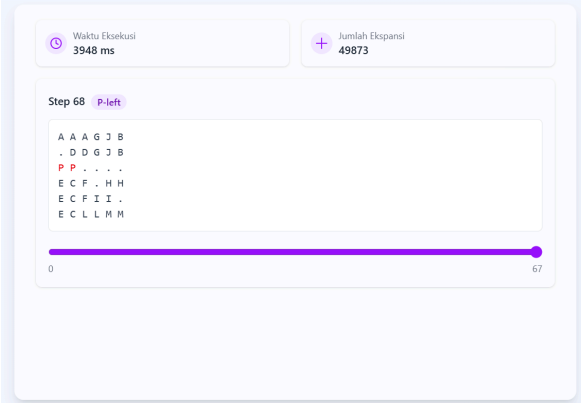
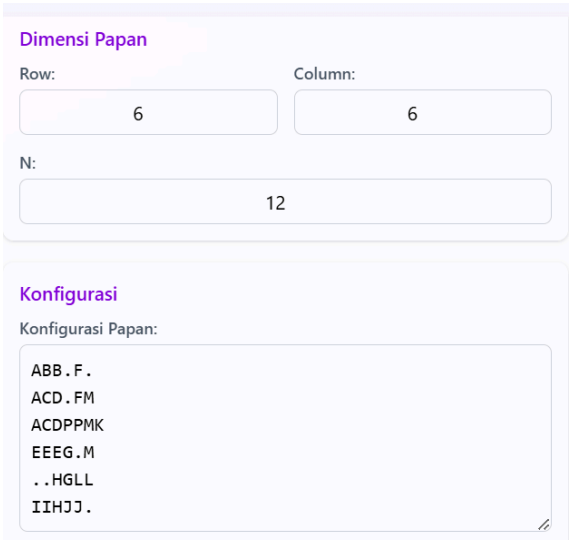
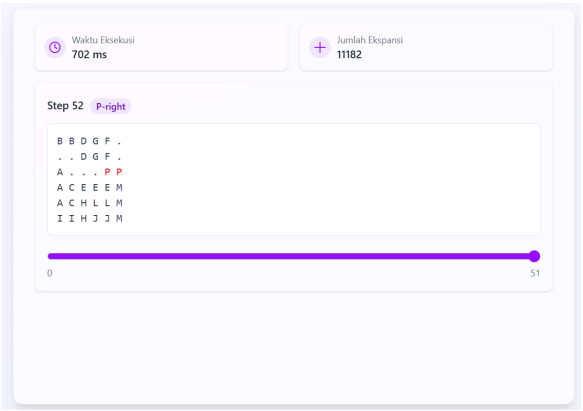
1	 Input easy level, menggunakan heuristik dasar.	
2	 Input medium level menggunakan heuristik Recursive Blockers	

3	 Input hard level menggunakan heuristik MinSteps	
4	 Input *hard level* dengan heuristik Recursive Blockers.	

4.4 Tabel Percobaan Menggunakan Beam Search

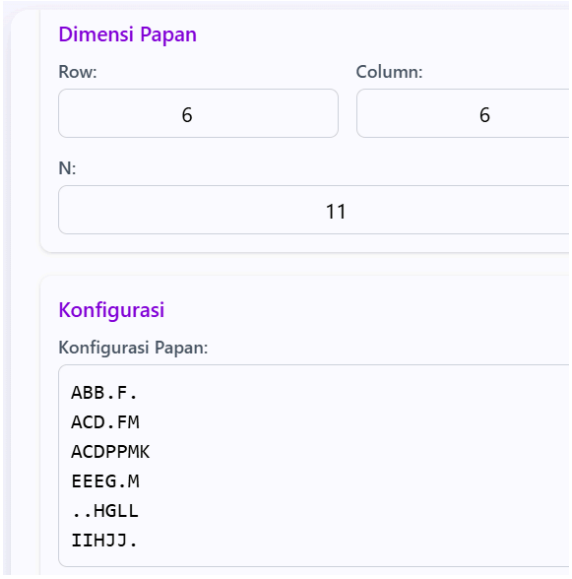
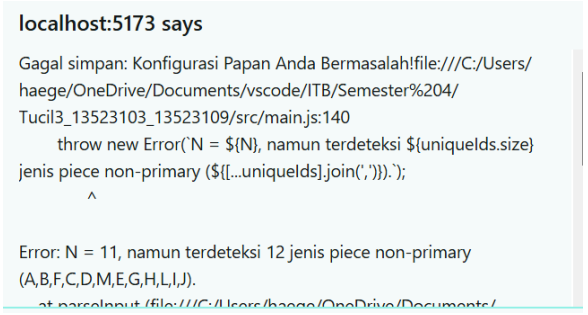
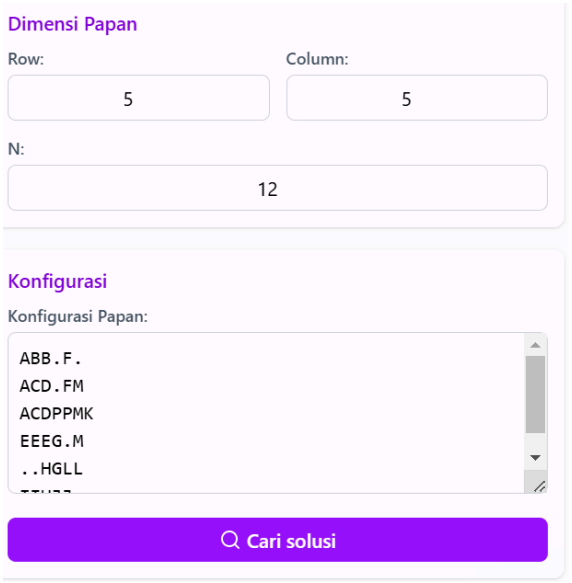
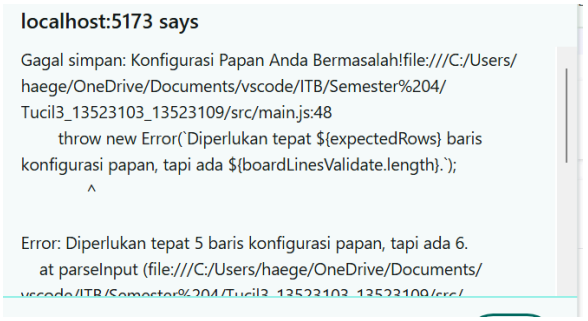
No.	Input	Output

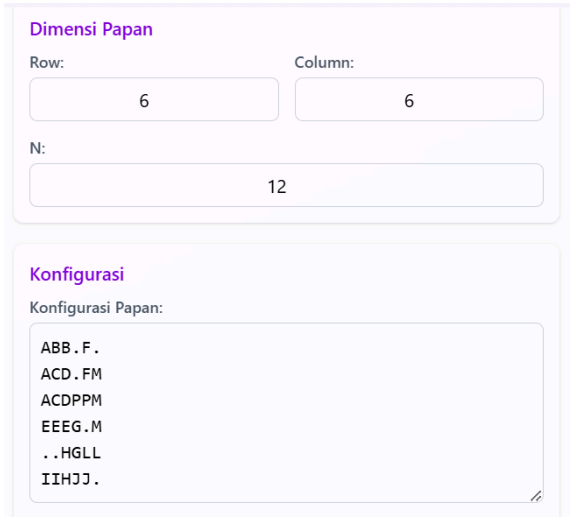
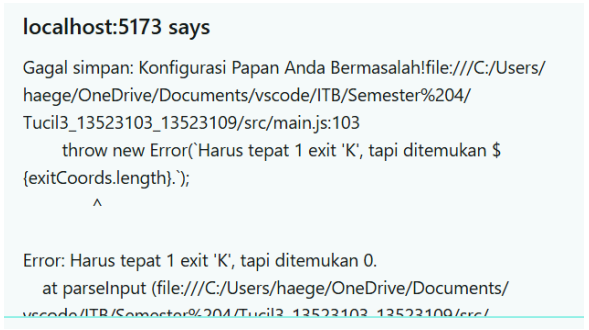
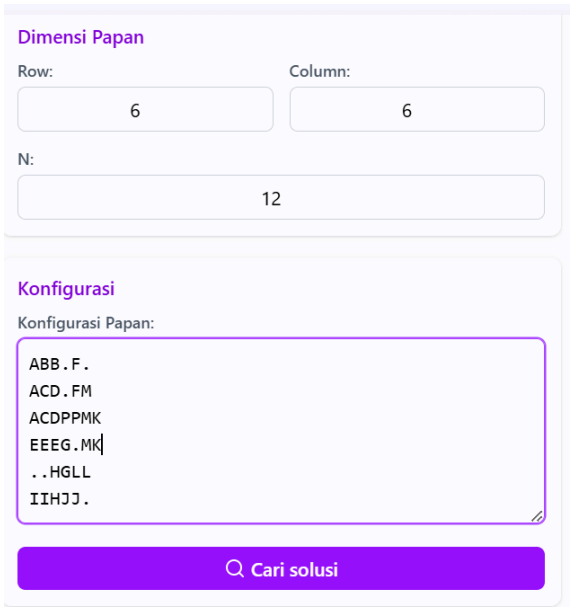
1	 Input easy level, menggunakan heuristik dasar.	
2	 Input medium level menggunakan heuristik Recursive Blockers	

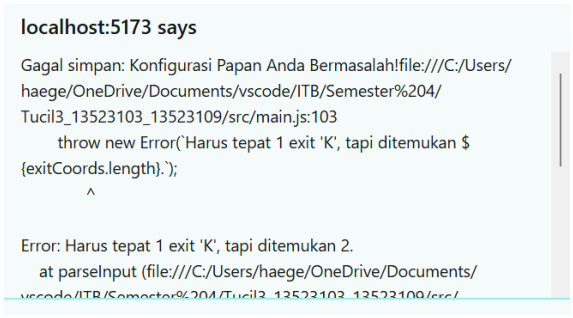
3	 Input hard level menggunakan heuristik MinSteps	
4	 Input *hard level* dengan heuristik Recursive Blockers.	

4.5 Tabel Percobaan Tambahan

No.	Input	Output

1	 Input jumlah N yang tidak sesuai	
2	 Input kolom dan row tidak sesuai dimensi papan	

<table>
<tr>
<td>3</td>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>4</td>
<td> 

Lebih dari 2 jalur K pada input</td>
<td> 
</td>
</tr>
</table>

BAB 5 ANALISIS HASIL PERCOBAAN

5.1 Analisis Percobaan

Berdasarkan hasil percobaan dengan keempat algoritma menggunakan 4 test case, diperoleh beberapa insight sebagai berikut:

- *Uniform Cost Search* (UCS) tetap menjadi algoritma terbaik dalam menentukan jalur terpendek (dilihat dari jumlah node yang dikunjungi). Hal ini karena UCS melakukan pencarian berdasarkan cost kumulatif sehingga menjamin solusi optimal. Namun, algoritma ini cenderung melakukan eksplorasi yang luas (blind iteration) terhadap banyak node karena setiap langkah memiliki cost yang sama, sehingga memerlukan waktu dan sumber daya yang lebih besar.

- *Greedy Best First Search* (GBFS) mampu mengurangi jumlah node yang dikunjungi secara signifikan dibandingkan UCS dengan memprioritaskan node yang tampak paling menjanjikan berdasarkan heuristic. Namun, algoritma ini tidak menjamin solusi optimal, dan pada kasus tertentu (misalnya test case 3) solusi yang dihasilkan memiliki langkah ekstra yang tidak efisien.

- *A Search** menggabungkan keunggulan UCS dan GBFS dengan mempertimbangkan cost kumulatif serta heuristic yang admissible. Algoritma ini mampu menghasilkan jalur terpendek dengan jumlah node yang dikunjungi relatif efisien. Secara umum, A* memberikan keseimbangan optimal antara reliabilitas (keakuratan solusi) dan efisiensi pencarian.

- *Beam Search* merupakan variasi dari A* yang membatasi jumlah node yang dieksplorasi pada setiap tingkat (beam width). Algoritma ini mengurangi jumlah node yang dikunjungi secara drastis sehingga lebih cepat, tetapi dengan risiko kehilangan solusi karena pruning yang terlalu agresif dan tidak terkendali. Oleh karena itu, beam search cenderung kurang reliabel dalam menemukan solusi optimal.

5.2 Kompleksitas Algoritma

Uniform Cost Search (UCS)

- **Kompleksitas Waktu**

  Pada ruang status dengan faktor cabang rata-rata b dan kedalaman solusi d, UCS pada kasus terburuk akan mengekspansi hingga semua simpul sampai kedalaman d, sehingga $O(b\^d)$. Karena setiap enqueue/dequeue di priority queue memerlukan waktu $O(log\ M)$(dengan M = ukuran frontier), totalnya menjadi $O(b\^d \times log\ b\^d)$.

- **Kompleksitas Ruang**

  Menyimpan frontier dan explored set hingga ukuran $T(b\^d)$, jadi $O(b\^d)$.

Greedy Best-First Search (GBFS)

- **Kompleksitas Waktu**

  GBFS hanya memprioritaskan berdasarkan heuristik $h(n)$ tanpa mempertimbangkan biaya kumulatif $g(n)$. Dalam kasus terburuk, ia dapat mengekspansi hampir semua simpul sampai kedalaman mmm sebelum menemukan solusi, sehingga $O(b\^m)$. Sama seperti UCS, setiap operasi heap (enqueue/dequeue) butuh $O(log\ M)$, sehingga total $O(b\^m \times log\ b\^m)$.

- **Kompleksitas Ruang**

Frontier dan explored set tumbuh hingga $T(b\hat{}m)$, jadi $O(b\hat{}m)$.

A★ Search

- **Kompleksitas Waktu**

  Worst-case masih eksponensial, $O(b\hat{}d)$, di mana d adalah kedalaman solusi optimal. Dengan heuristik *admissible* dan konsisten, jumlah simpul yang diekspansi secara signifikan lebih sedikit daripada UCS, tetapi dalam analisis asymptotic tetap eksponensial. Namun, overhead heap menambah faktor log seperti pada UCS, sehingga menjadi $O(b\hat{}d \times log\ b\hat{}d)$.

- **Kompleksitas Ruang**

  Menyimpan frontier dan explored set hingga $T(b\hat{}d)$, sehingga $O(b\hat{}d)$.

Beam Search (algoritma bonus)

- **Kompleksitas Waktu**

  Dengan lebar beam β, setiap iterasi hanya mempertahankan β simpul terbaik. Setiap simpul mengembang ke rata-rata b tetangga, di-sort berdasarkan $f(n)$ dan dipotong jadi β. Jika solusi ditemukan pada kedalaman d, total waktu yang dibutuhkan sekitar $O(d \times (b\beta\ +\ \beta\ log\ \beta))$, atau disederhanakan menjadi $O(d \times b\beta\ log\ \beta)$.

- **Kompleksitas Ruang**

  Frontier dibatasi β, jadi $O(\beta)$.

Catatan Umum

Semua algoritma di atas mengekspansi simpul menggunakan metode *getNeighbors()* di Node/Board, dimana metode tersebut melakukan *copy* papan dan menghitung heuristik. Selain itu, karena ruang status Rush Hour relatif kecil (dengan ukuran papan umum 6×6 dengan belasan potongan mobil), overhead eksponensial masih dapat ditangani untuk banyak kasus uji, tetapi selalu perlu waspada terhadap kombinasi puzzle yang mendekati kompleksitas maksimum.

BAB 6 IMPLEMENTASI BONUS

6.1 Algoritma Tambahan

Beam Search adalah varian dari A* yang membatasi ukuran *frontier* (daftar simpul yang sedang dipertimbangkan) pada setiap langkah, untuk mengendalikan penggunaan memori dan waktu eksekusi. Alih-alih menyimpan semua tetangga, Beam Search hanya mempertahankan k simpul terbaik (disebut *beam width*) berdasarkan nilai $f(n) = g(n) + h(n)$.

Dalam kode program (file search.js), fungsi ini diimplementasikan sebagai:

```
export function beamSearch(startBoard, beamWidth = 1000, heuristic = 1) {
… }
```

Dimana

- **startBoard** adalah konfigurasi papan awal.
- **beamWidth** (default = 1000) menentukan jumlah simpul maksimum yang akan dipertahankan di setiap iterasi.
- **heuristic** memilih fungsi $h(n)$ yang dipakai untuk menghitung $f(n)$ search.

Langkah-langkah utama algoritma Beam Search terbagi menjadi 3:

1. **Inisialisasi**

   Membuat simpul awal Node(startBoard, null, null, heuristic) dan masukkan ke array beam. Lalu, menyiapkan explored set untuk menghindari *revisit*.

2. **Loop utama** (selama beam tidak kosong):

   Untuk tiap simpul di beam, dihitung ekspansi (increment expansions), cek apakah sudah isGoal(). Lalu, algoritma menambahkan semua tetangga (node.getNeighbors()) yang belum ada di explored ke daftar successors. Jika successors kosong, hentikan dan laporkan "no solution". Successors diurutkan menurut nilai f menaik, kemudian pilih *beamWidth* simpul paling menjanjikan menjadi beam berikutnya.

3. **Terminasi**

   Jika di tengah loop ditemukan simpul tujuan, kembalikan simpul tersebut dan jumlah ekspansi. Sementara jika loop selesai tanpa menemukan solusi, kembalikan no solution beserta jumlah ekspansi.

Keunggulan dari algoritma ini adalah mengurangi penggunaan memori dibanding A* penuh, karena beam tidak tumbuh melebihi ukuran tetap. Sementara, keterbatasannya adalah algoritma ini tidak menjamin menemukan solusi optimal—dimana jika *beam width* terlalu kecil, jalur terbaik bisa terpotong dari frontier.

6.2 Heuristik Tambahan

Count Direct Blockers (Heuristik Dasar)

Heuristik countDirectBlockers adalah fungsi heuristik yang menghitung jarak minimum dari ujung depan mobil target hingga pintu keluar (*distance*), serta jumlah mobil penghalang (*blockers*) di antara jalur lurus mobil target dan pintu keluar.

Heuristik ini berawal dari menemukan objek mobil target (c.isTarget) dari array cars. Lalu, ia menentukan indeks *frontIdx* (ujung depan mobil target) dan indeks *exitIdx* (posisi pintu keluar) berdasarkan orientasi horizontal/vertikal.

Distance dari mobil target ke posisi pintu keluar dihitung dengan formula $distance = |exitIdx - frontIdx|$. Sehabis itu, dilakukan iterasi di antara baris/kolom frontIdx dan exitIdx, meng *increment* blockers setiap kali menemukan ID mobil (this.grid[...]).

Nilai akhir dari heuristik ini adalah $h1(n) = distance + blockers$.

Count Recursive Blockers (Heuristik Bonus #1)

Heuristik ini adalah kombinasi heuristik dasar (countDirectBlockers) ditambah beban tambahan dari setiap penghalang yang terhubung secara berantai. Cara kerja implementasi dari algoritma ini adalah menghitung nilai heuristik dasar. Lalu, untuk mobil target, dipanggil fungsi rekursif *carBlockers*(target,id, burden).

Fungsi *carBlockers*(carId) awalnya memuat blockers = set() berisi semua ID mobil yang memblokir lajur mobil carId (baris penuh untuk mobil horizontal, kolom penuh untuk vertikal). Lalu, dihitung $weight = blockers.size$. Setiap bId di blockers ditambahkan 0.5 × carBlockers(bId) untuk memasukkan beban mobil-mobil penghalang dari penghalang itu sendiri, dengan cache (burdenCache) dan visiting set untuk mencegah loop tak hingga. Setelah itu, fungsi langsung mengembalikan weight.

Dengan demikian, total nilai heuristiknya adalah $h2(n) = base + burden$, dimana burden merupakan hasil dari panggilan fungsi rekursi. Dengan cara mengimplementasikan heuristik semacam ini, menggerakkan mobil yang memblokir banyak lajur (atau lajur berantai) akan menaikkan nilai heuristik secara proporsional.

Count Min Steps (Heuristik Bonus #2)

Heuristik ini menghitung jarak langsung dari ujung depan mobil target ke pintu keluar ditambah jumlah langkah minimum yang diperlukan untuk membebaskan setiap mobil penghalang. Berbeda dengan kedua heuristik sebelumnya, heuristik ini mencoba memperkirakan total pergeseran langkah yang tersisa, bukan sekadar menghitung jumlah penghalang atau beban rekursif.

Sama seperti countDirectBlockers, cara kerja dari heuristik ini adalah mencari distance dan mengumpulkan daftar blockers berupa ID mobil yang menghalangi jalur lurus mobil target.

Lalu untuk setiap id dalam blockers, fungsi memanggil method this.minStepsToFree(car) untuk menghitung berapa langkah minimal yang dibutuhkan agar mobil car dapat digeser sepenuhnya dari jalur target. Semua langkah tersebut dijumlahkan menjadi movesNeeded.

Nilai yang dikembalikan adalah $h3(n) = distance + movesNeeded$.

6.3 GUI

Frontend

Aplikasi React.jsx ini menyediakan sidebar dengan input terkontrol untuk menentukan ukuran papan, nilai "N", algoritma, heuristik, dan konfigurasi awal puzzle, kemudian mengirim data tersebut ke backend. Setelah menerima array solusi {step, move, board, cars}, komponen utama akan menampilkan setiap langkah dengan animasi play-pause yang dikendalikan oleh state dan setInterval, memperlihatkan papan dengan huruf "P" berwarna merah, badge gerakan saat ini, serta ada slider yang tersinkronisasi untuk memilih langkah secara manual.

Backend

Aplikasi ini memanfaatkan Express.js sebagai server HTTP untuk mengeksekusi skrip utama (main.js) yang bertugas menjalankan logika pemecahan puzzle, kemudian menangkap output yang dihasilkan yang meliputi detail langkah, waktu eksekusi, dan jumlah ekspansi dan menyusunnya dalam format JSON. Setelah itu, JSON tersebut dikirimkan melalui endpoint API ke frontend React untuk ditampilkan di dalam website.

LAMPIRAN

Repository GitHub

Tabel Checklist

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

Referensi Tambahan

Materi kuliah IF3170 Inteligensi Buatan Teknik Informatika ITB:

http://kuliah.itb.ac.id → STEI → Teknik Informatika → IF3170

Stuart J Russell & Peter Norvig, Artificial Intelligence: A Modern Approach, 3rd Edition, Prentice-Hall International, Inc, 2010, Textbook

http://aima.cs.berkeley.edu/ (2nd edition)

Free online course materials | MIT OpenCourseWare Website:

http://ocw.mit.edu/courses/electrical-engineering-andcomputer-science/

Lecture Notes in Informed Heuristic Search, ICS 271 Fall 2008,

http://www.ics.uci.edu/~dechter/courses/ics-271/fall-08/lecturenotes/4.InformedHeuristicSearch.ppt