Laporan Tugas Kecil 3
IF2211 Strategi Algoritma

Penyelesaian Permainan Word Ladder Menggunakan Algoritma UCS, Greedy Best First Search, dan A*

Disusun oleh:

Andi Farhan Hidayat 13523128

Muhammad Aulia Azka 13523137

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2024

Daftar Isi

BAB 1

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.

Pada tugas ini penulis harus membuat program sederhana yang mengimplementasikan algoritma *pathfinding Greedy Best First Search, UCS (Uniform Cost Search), dan A** dalam menyelesaikan permainan Rush Hour.

BAB 2
Landasan Teori

2.1 Algoritma *Uniform Cost Search*

Algoritma *Uniform Cost Search* (UCS) adalah algoritma pencarian jalur yang menggunakan strategi eksplorasi berdasarkan biaya kumulatif terendah dari titik awalke node tertentu. UCS menggunakan struktur data *priority queue* untuk memilih node dengan biaya paling rendah yang belum dikunjungi. UCS termasuk dalam kategori blind search karena tidak memanfaatkan informasi tambahan seperti heuristik. Algoritma ini memastikan pencarian solusi yang optimal jika semua bobot sisi bernilai non-negatif.

Uniform Cost Search (UCS) bekerja mirip seperti *Breadth First Search* (BFS), namun bukan berdasarkan kedalaman level, melainkan berdasarkan total cost. Karena selalu memilih node dengan cost terkecil terlebih dahulu, UCS menjamin bahwa saat goal ditemukan, jalur tersebut adalah jalur dengan cost yang minimum.

Langkah umum algoritma UCS:

1. Inisialisasi priority queue dengan node awal dan cost 0;
2. Selama priority queue tidak kosong:
   - Ambil node dengan cost terendah
   - Jika node tersebut adalah goal, kembalikan path sebagai solusi
   - Ekspansi node tersebut dan tambahkan node - node hasil ekspansi ke priority queue dengan cost kumulatif

2.2 Algoritma *Greedy Best First Search*

Algoritma *Greedy Best First Search* adalah algoritma pencarian yang berfokus pada perluasan node dengan nilai heuristic terendah tanpa mempertimbangkan total cost dari node awal. *Greedy Best*

First Search termasuk dalam kategori *informed search* karena menggunakan fungsi heuristic (h(n)) sebagai estimasi jarak dari suatu node ke goal. Tujuannya adalah untuk mencapai goal secepat mungkin, bukan untuk mendapatkan jalur optimal. Namun, algoritma ini tidak menjamin solusi optimal dan bahkan tidak selalu selesai. Dalam beberapa kasus, GBFS bisa terjebak dalam loop atau jalan buntu jika heuristik mengarahkan ke arah yang salah

Langkah umum algoritma GBFS:

1. Inisialisasi priority queue dengan node awal dan nilai heuristic
2. Selama priority queue tidak kosong:
   - Ambil node dengan heuristic terendah
   - Jika node tersebut adalah goal, kembalikan path sebagai solusi
   - Ekspansi node tersebut dan tambahkan hasil ekspansi ke queue berdasarkan nilai heuristic

2.3 Algoritma *A**

Algoritma *A** adalah algoritma pencarian yang menggabungkan keunggulan UCS dan GBFS dengan memperhitungkan cost kumulatif dari awal (g(n)) dan estimasi jarak ke goal (h(n)) melalui fungsi evaluasi f(n) = g(n) + h(n). Dengan pendekatan ini, A* tidak hanya berusaha mencapai goal dengan cepat, tetapi juga hasil solusi yang optimal. A* termasuk algoritma informed search dan sangat powerful karena menjamin *complete* dan optimal, asalkan fungsi heuristic yang digunakan bersifat admissible.

Langkah umum algoritma A*:

1. Inisialisasi priority queue dengan node awal dan f(n) = g(n) + h(n)
2. Selama *priority queue* tidak kosong:
   - Ambil node dengan nilai f(n) terendah
   - Jika node tersebut adalah goal, kembalikan path sebagai solusi
   - Ekspansi node tersebut dan tambahkan node hasil ekspansi ke queue dengan nilai f(n) masing - masing

2.4 Algoritma *Beam Search*

Beam Search adalah algoritma pencarian heuristik yang merupakan modifikasi dari *Breadth-First Search* (BFS). Perbedaan algoritma ini dengan BFS terletak pada pembatasan jumlah *node* yang dipertahankan pada setiap level pencarian. *Beam Search* hanya menyimpan sejumlah node terbaik pada setiap langkah, sebanyak nilai beamwidth yang ditentukan. Beam Search menggabungkan prinsip

heuristic search dan *controlled memory*. Tujuannya adalah menyeimbangkan efisiensi waktu dan ruang dengan kualitas solusi. *Beam Search* memiliki beberapa kelebihan yaitu lebih cepat dan hemat memori. Namun algoritma ini memiliki kelemahan yaitu tidak optimal dan terdapat kemungkinan gagal menemukan solusi walaupun sebenarnya terdapat solusi.

Langkah umum algoritma *Beam Search*:

1. Inisialisasi priority queue:
    - Tambahkan node awal
    - Set nilai *beam width*
2. Selama queue tidak kosong:
    - Ambil semua node di queue dan ekspansi semua kemungkinan
    - Hitung nilai heuristic untuk setiap node hasil ekspansi
    - Urutkan semua successor berdasarkan nilai h(n)
    - Simpan hanya *beam width* node teratas ke dalam queue untuk level berikutnya
3. Cek solusi:
    - Jika ada node yang merupakan goal, maka hentikan proses dan kembalikan *path* solusi
    - Jika tidak ada node yang tersisa, artinya pencarian gagal.

BAB 3
Analisis dan Implementasi Masalah

3.1 Definisi f(n) dan g(n)

G(n) adalah fungsi yang menyatakan biaya kumulatif dari titik awal (*initial node*) ke suatu node n. Dalam konteks ini (Permainan Rush Hour), g(n) sering dianggap sebagai jumlah langkah yang dilakukan dari awal hingga mencapai node n.

F(n) adalah fungsi evaluasi yang digunakan oleh algoritma pencarian untuk menentukan prioritas suatu node. Definisi dari f(n) sendiri berbeda untuk tiap algoritma. Dalam konteks tugas ini:

1. UCS: f(n) = g(n)
2. GBFS: f(n) = h(n)
3. A*: f(n) = g(n) + h(n)

3.2 Penjelasan heuristic yang digunakan A*

Heuristic yang digunakan pada algoritma A* dalam implementasi tugas ini adalah *admissible.* Heuristic yang dikatakan *admissible* adalah heuristik yang tidak pernah melebihi biaya minimum sebenarnya dari node saat ini ke goal. Dalam implementasi tugas ini terdapat 3 heuristic:

1. Manhattan Distance:
   Heuristic ini hanya menghitung jarak dari posisi ujung primary piece ke pintu keluar dalam garis lurus (tanpa memperhitungkan rintangan). Heuristic ini tidak akan melebihi cost oleh karena itu, heuristic ini *admissible*.
2. Blocking Pieces:
   Heuristik ini menghitung jumlah kendaraan yang menghalangi jalur primary piece ke goal. Heuristic ini *admissible* karena hanya memperkirakan rintangan yang perlu disingkirkan.
3. Distance to Exit:
   Heuristic ini merupakan kombinasi konservatif dari Blocking Pieces dan Manhattan Distance. Dengan penggabungan tersebut, heuristic ini tetap *admissible* selama tidak melebih - lebihkan cost menuju goal.

3.3 Apakah UCS sama dengan BFS pada penyelesaian Rush Hour?

Secara urutan node yang dibangkitkan dan path yang dihasilkan, UCS bisa dibilang mirip tapi tidak identik dengan BFS. Cara kerja UCS adalah memperluas node berdasarkan total cost terendah, pada tugas ini semua langkah memiliki cost tetap, maka urutan eksplorasi UCS akan identik dengan BFS apabila semua node diekspansi dalam urutan level yang sama.

3.4 Apakah A* lebih efisien daripada UCS pada Rush Hour?

Secara teoritis A* lebih efisien dibandingkan UCS. Hal ini disebabkan karena A* menggunakan *heuristic* yang mengarahkan pencarian ke goal. UCS mencari semua kemungkinan path secara eksploratif berdasarkan cost, sedangkan A* menggabungkan cost aktual dan estimasi ke goal, sehingga pencarian bisa lebih fokus ke jalur yang memiliki peluang tinggi ke goal. Dengan *heuristic* yang *admissible* seperti *Manhattan Distance*, *Blocking Pieces*, dan *Distance to Exit*, A* tetap optimal dan lebih cepat daripada UCS karena mampu memangkas jalur eksplorasi terhadap jalur yang tidak perlu.

3.5 Apakah GBFS menjamin solusi optimal pada Rush Hour?

Greedy Best First Search (GBFS) tidak menjamin solusi optimal pada permasalahan ini. Hal ini disebabkan karena *Greedy Best First Search* (GBFS) hanya menggunakan nilai *heuristic* untuk menentukan prioritas node yang akan diekspansi, tanpa mempertimbangkan cost yang telah dikeluarkan. Oleh sebab itu, algoritma *Greedy Best First Search* (GBFS) tidak menjamin solusi yang optimal meskipun bisa lebih cepat mencapai goal.

Greedy Best First Search (GBFS) bisa terjebak dalam jalur yang tampak menjanjikan berdasarkan heuristic, namun ternyata dengan memilih jalur tersebut, cost keseluruhan lebih tinggi. Hal ini menyebabkan solusi yang ditemukan tidak selalu merupakan solusi minimum langkah. Dengan heuristic seperti *Manhattan Distance*, *Blocking Pieces*, maupun *Distance to Exit*, *Greedy Best First Search* (GBFS) tetap tidak dapat menjamin optimalitas karena algoritma tersebut mengabaikan total cost yang sudah dikeluarkan.

BAB 4

Source Code

4.1 UCS.ts

Kode ini merupakan implementasi algoritma Uniform Cost Search (UCS). UCS bekerja dengan menjelajahi node berdasarkan path cost terendah terlebih dahulu menggunakan struktur priority queue. Algoritma dimulai dari keadaan awal (startNode) dan secara bertahap mengeksplorasi kemungkinan gerakan yang dihasilkan oleh MovementManager. Setiap gerakan menghasilkan board baru yang dikemas dalam objek SearchNode dengan biaya (*cost*) yang terus bertambah tiap langkah. Untuk mencegah siklus, state board disimpan dalam Set bernama visited. Ketika ditemukan board yang memenuhi kondisi tujuan (*isGoal()*), algoritma berhenti dan mengembalikan jalur solusi, waktu eksekusi, serta jumlah node yang telah dikunjungi.

```ts
"use client";

import { Board } from "../models/Board";
import { MovementManager } from "../utils/Move";
import { SearchNode } from "../models/SearchNode";
import PriorityQueue from "ts-priority-queue";
import { SearchResult } from "../models/SearchResult";

// UCS Algorithm
export class UCSAlgorithm {
  static search(initialBoard: Board): SearchResult {
    const startTime = performance.now();

    // Priority queue ordered by path cost
    const priorityQueue = new PriorityQueue<SearchNode>({
      comparator: (a, b) => a.cost - b.cost
    });

    // Initialize starting node
    const startNode = new SearchNode(initialBoard);
    priorityQueue.queue(startNode);

    const visited = new Set<string>();
    visited.add(startNode.getBoardString());

    let nodesVisited = 0;
    while (priorityQueue.length > 0) {
      // Node with lowest cost
      const current = priorityQueue.dequeue();
      nodesVisited++;

      if (current.board.isGoal()) {
        const endTime = performance.now();
        return {
          success: true,
          path: current.getPath(),
          nodesVisited,
          executionTime: endTime - startTime
        };
      }

      // All possible moves
      const possibleMoves = MovementManager.getPossibleMoves(current.board);

      for (const move of possibleMoves) {
        // Apply move
        const newBoard = MovementManager.applyMove(current.board, move);
        const newBoardString = newBoard.getGrid().map(row =>
row.join('')).join('');
        if (visited.has(newBoardString)) {
          continue;
        }

        // Create new node
        const newNode = new SearchNode(
          newBoard,
          current,
          move,
          current.cost + 1 // increase cost each move
        );

        // Add the new node
        priorityQueue.queue(newNode);
        visited.add(newBoardString);
      }
    }

    const endTime = performance.now();
    return {
      success: false,
      path: [],
      nodesVisited,
      executionTime: endTime - startTime
    };
  }
}
```

4.2 a-star.ts

Kode ini merupakan implementasi algoritma A*. Algoritma ini memanfaatkan *priority queue* yang mengurutkan node berdasarkan fungsi nilai f(n) = g(n) + h(n), dengan g(n) adalah biaya aktual dari awal ke node saat ini, dan h(n) adalah estimasi heuristik dari node ke tujuan yang dihitung oleh kelas Heuristics dengan tipe heuristik yang bisa dipilih. Dari node awal, algoritma menjelajah semua gerakan yang memungkinkan, menghasilkan state baru, dan memasukkannya ke *priority queue* jika belum pernah dikunjungi. Proses ini berulang hingga ditemukan kondisi goal, lalu mengembalikan jalur solusi, jumlah node yang dikunjungi, dan waktu eksekusi. Jika tidak ditemukan solusi, algoritma mengembalikan hasil gagal.

```
"use client";

import { Board } from "../models/Board";
import { MovementManager } from "../utils/Move";
import { SearchNode } from "../models/SearchNode";
import PriorityQueue from "ts-priority-queue";
import { HeuristicType } from "@/store/GameStore";
import { SearchResult } from "../models/SearchResult";
import { Heuristics } from "./heuristics";

// A* Algorithm
export class AStarAlgorithm {
  static search(initialBoard: Board, heuristicType: HeuristicType = HeuristicType.MANHATTAN): SearchResult
{   const startTime = performance.now();

    // Priority queue ordered by f(n) = g(n) + h(n)
    // g(n) = cost so far; h(n) = heuristic cost
    const priorityQueue = new PriorityQueue<SearchNode>({
      comparator: (a, b) => {
        const aHeuristic = Heuristics.calculateHeuristic(a.board, heuristicType);
        const bHeuristic = Heuristics.calculateHeuristic(b.board, heuristicType);
        const aTotal = a.cost + aHeuristic;
        const bTotal = b.cost + bHeuristic;

        return aTotal - bTotal;
      },
    });

    // Initialize starting node
    const startNode = new SearchNode(initialBoard);
    priorityQueue.queue(startNode);

    const visited = new Set<string>();
    visited.add(startNode.getBoardString());

    let nodesVisited = 0;

    while (priorityQueue.length > 0) {
      // Lowest f(n) node
      const current = priorityQueue.dequeue();
      nodesVisited++;

      if (current.board.isGoal()) {
        const endTime = performance.now();
        return {
          success: true,
          path: current.getPath(),
          nodesVisited,
          executionTime: endTime - startTime,
        };
      }

      // All possible moves
      const possibleMoves = MovementManager.getPossibleMoves(current.board);

      for (const move of possibleMoves) {
        // Apply move
        const newBoard = MovementManager.applyMove(current.board, move);
        const newBoardString = newBoard
          .getGrid()
          .map((row) => row.join(""))
          .join("");

        if (visited.has(newBoardString)) {
          continue;
        }

        // Create new node
        const newNode = new SearchNode(
          newBoard,
          current,
          move,
          current.cost + 1 // g(n) = parent's g(n) + 1
        );

        priorityQueue.queue(newNode);
        visited.add(newBoardString);
      }
    }

    // No solution found
    const endTime = performance.now();
    return {
      success: false,
      path: [],
      nodesVisited,
      executionTime: endTime - startTime,
    };
  }
}
```

4.3 gbfs.ts

Kode ini merupakan implementasi algoritma Greedy Best-First Search (GBFS) dengan memanfaatkan *priority queue* yang mengurutkan node berdasarkan nilai heuristik h(n) saja, tanpa memperhitungkan biaya jalur yang sudah ditempuh. Dari node awal, algoritma memilih node dengan estimasi heuristik terendah ke tujuan, kemudian menjelajah semua kemungkinan gerakan dan memasukkan state baru yang belum dikunjungi ke *priority queue*. Proses ini berlanjut hingga mencapai kondisi goal atau tidak ada lagi node untuk dieksplorasi. Hasilnya berupa jalur solusi jika ditemukan, jumlah node yang dikunjungi, dan waktu eksekusi, atau kegagalan jika solusi tidak ditemukan. Algoritma ini cenderung lebih cepat tapi tidak selalu optimal dibanding A*.	

```
"use client";

import { Board } from "../models/Board";
import { MovementManager } from "../utils/Move";
import { SearchNode } from "../models/SearchNode";
import PriorityQueue from "ts-priority-queue";
import { HeuristicType } from "@/store/GameStore";
import { SearchResult } from "../models/SearchResult";
import { Heuristics } from "./heuristics";

// Greedy Best-First Search Algorithm
export class GBFSAlgorithm {
  static search(initialBoard: Board, heuristicType: HeuristicType = HeuristicType.MANHATTAN): SearchResult
{   const startTime = performance.now();

    // Priority queue ordered only by heuristic value h(n)
    const priorityQueue = new PriorityQueue<SearchNode>({
      comparator: (a, b) => {
        const aHeuristic = Heuristics.calculateHeuristic(a.board, heuristicType);
        const bHeuristic = Heuristics.calculateHeuristic(b.board, heuristicType);

        return aHeuristic - bHeuristic;
      },
    });

    // Initialize starting node
    const startNode = new SearchNode(initialBoard);
    priorityQueue.queue(startNode);

    const visited = new Set<string>();
    visited.add(startNode.getBoardString());

    let nodesVisited = 0;

    while (priorityQueue.length > 0) {
      // Lowest heuristic value node
      const current = priorityQueue.dequeue();
      nodesVisited++;

      if (current.board.isGoal()) {
        const endTime = performance.now();
        return {
          success: true,
          path: current.getPath(),
          nodesVisited,
          executionTime: endTime - startTime,
        };
      }

      // All possible moves
      const possibleMoves = MovementManager.getPossibleMoves(current.board);

      for (const move of possibleMoves) {
        // Apply move
        const newBoard = MovementManager.applyMove(current.board, move);
        const newBoardString = newBoard
          .getGrid()
          .map((row) => row.join(""))
          .join("");

        if (visited.has(newBoardString)) {
          continue;
        }

        // Create new node with updated cost
        const newNode = new SearchNode(newBoard, current, move, current.cost + 1);

        priorityQueue.queue(newNode);
        visited.add(newBoardString);
      }
    }

    // No solution found
    const endTime = performance.now();
    return {
      success: false,
      path: [],
      nodesVisited,
      executionTime: endTime - startTime,
    };
  }
}
```

4.4 heuristic.ts

Kode ini mengimplementasikan perhitungan heuristik berdasarkan tipe heuristik yang dipilih. Terdapat 3 heuristik, yaitu *Manhattan Distance, Blocking Pieces, dan Distance to Exit.* Heuristik Manhattan Distance menghitung jarak lurus dari ujung primary piece ke pintu keluar (exit) baik secara horizontal maupun vertikal tergantung orientasi dan tanpa memperhatikan rintangan. Hal tersebut memberikan estimasi jarak minimum yang harus ditempuh. Kemudian, heuristik Blocking Pieces menghitung jumlah kendaraan yang menghalangi jalur primary piece ke exit. Setiap kendaraan penghalang dihitung satu kali (unik), lalu dikalikan dua untuk memberi penalti tambahan terhadap rintangan. Terakhir, heuristik Distance to Exit merupakan gabungan dari keduanya dengan formula manhattan + blocking.

```
"use client";

import { Board } from "../models/Board";
import { HeuristicType } from "@/store/GameStore";

export class Heuristics {
  static calculateHeuristic(board: Board, heuristicType: HeuristicType): number {
    switch (heuristicType) {
      case HeuristicType.MANHATTAN:
        return this.manhattanDistanceHeuristic(board);
      case HeuristicType.BLOCKING:
        return this.blockingPiecesHeuristic(board);
      case HeuristicType.DISTANCE:
        return this.distanceToExitHeuristic(board);
      default:
        return this.manhattanDistanceHeuristic(board);
    }
  }

  // Manhattan distance heuristic
  static manhattanDistanceHeuristic(board: Board): number {
    let primaryPiece = null;
    for (const piece of board.getPieces().values()) {
      if (piece.isPrimary) {
        primaryPiece = piece;
        break;
      }
    }

    if (!primaryPiece) return 0;

    const exitRow = board.getExitRow();
    const exitCol = board.getExitCol();

    let distance = 0;

    // Horizontal
    if (primaryPiece.isHorizontal) {
      // Right exit
      if (exitCol === board.getCols()) {
        const rightEdgeOfPiece = primaryPiece.col + primaryPiece.length - 1;
        const rightEdgeOfBoard = board.getCols() - 1;
        distance = rightEdgeOfBoard - rightEdgeOfPiece;
      }
      // Left exit
      else if (exitCol === -1) {
        distance = primaryPiece.col;
      }
    }
    // Vertical
    else {
      // Bottom exit
      if (exitRow === board.getRows()) {
        const bottomEdgeOfPiece = primaryPiece.row + primaryPiece.length - 1;
        const bottomEdgeOfBoard = board.getRows() - 1;
        distance = bottomEdgeOfBoard - bottomEdgeOfPiece;
      }
      // Top exit
      else if (exitRow === -1) {
        distance = primaryPiece.row;
      }
    }

    return distance;
  }

  // Blocking pieces heuristic
  static blockingPiecesHeuristic(board: Board): number {
    let primaryPiece = null;
    for (const piece of board.getPieces().values()) {
      if (piece.isPrimary) {
        primaryPiece = piece;
        break;
      }
    }

    if (!primaryPiece) return 0;

    const grid = board.getGrid();
    const exitRow = board.getExitRow();
    const exitCol = board.getExitCol();
```

```
    let blockingPieces = 0;

    // Horizontal
    if (primaryPiece.isHorizontal) {
      // Right exit
      if (exitCol === board.getCols() && primaryPiece.row === exitRow)
{       const rightEdge = primaryPiece.col + primaryPiece.length;
        const blockedCells = new Set<string>();

        // Each cell from piece to exit
        for (let c = rightEdge; c < board.getCols(); c++) {
          const cellValue = grid[primaryPiece.row][c];
          if (cellValue !== "." && cellValue !== "P") {
            blockedCells.add(cellValue);
          }
        }

        blockingPieces = blockedCells.size;
      }
      // Left exit
      else if (exitCol === -1 && primaryPiece.row === exitRow) {
        const blockedCells = new Set<string>();

        // Each cell from piece to exit
        for (let c = primaryPiece.col - 1; c >= 0; c--) {
          const cellValue = grid[primaryPiece.row][c];
          if (cellValue !== "." && cellValue !== "P") {
            blockedCells.add(cellValue);
          }
        }

        blockingPieces = blockedCells.size;
      }
    }
    // Vertical
    else {
      // Bottom exit
      if (exitRow === board.getRows() && primaryPiece.col === exitCol)
{       const bottomEdge = primaryPiece.row + primaryPiece.length;
        const blockedCells = new Set<string>();

        // Each cell from piece to exit
        for (let r = bottomEdge; r < board.getRows(); r++) {
          const cellValue = grid[r][primaryPiece.col];
          if (cellValue !== "." && cellValue !== "P") {
            blockedCells.add(cellValue);
          }
        }

        blockingPieces = blockedCells.size;
      }
      // Top exit
      else if (exitRow === -1 && primaryPiece.col === exitCol) {
        const blockedCells = new Set<string>();

        // Each cell from piece to exit
        for (let r = primaryPiece.row - 1; r >= 0; r--) {
          const cellValue = grid[r][primaryPiece.col];
          if (cellValue !== "." && cellValue !== "P") {
            blockedCells.add(cellValue);
          }
        }

        blockingPieces = blockedCells.size;
      }
    }

    return blockingPieces * 2; // Each blocking piece counts double
  }

  // Distance to exit heuristic - combines manhattan and blocking
  static distanceToExitHeuristic(board: Board): number {
    const distance = this.manhattanDistanceHeuristic(board);
    const blocking = this.blockingPiecesHeuristic(board);

    return distance + blocking * 0.5; // More conservative for A*
  }

  // Distance to exit heuristic for GBFS
  static gbfsDistanceToExitHeuristic(board: Board): number {
    const distance = this.manhattanDistanceHeuristic(board);
    const blocking = this.blockingPiecesHeuristic(board);

    return distance + blocking;
  }
}
```

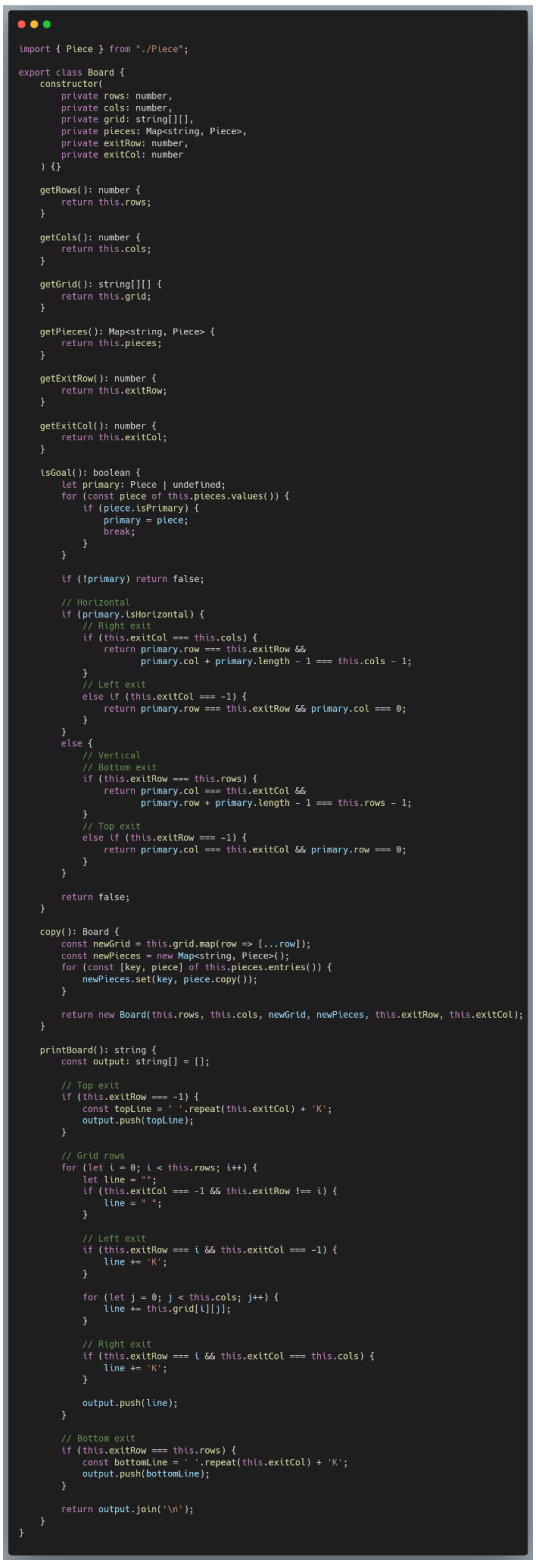
4.5 beamSearch.ts

Kode ini mengimplementasikan algoritma Beam Search dengan membatasi jumlah node yang diperluas di setiap tingkat pencarian sesuai lebar beam (*beamWidth*). Algoritma memulai dari node awal, lalu secara iteratif mengeksplorasi semua penerus dari node-node dalam beam saat ini, mengurutkannya berdasarkan nilai heuristik, dan hanya memilih sejumlah terbatas node terbaik (sesuai *beamWidth*) untuk melanjutkan ke tingkat berikutnya. Setiap node baru yang sudah dikunjungi disimpan agar tidak diproses ulang. Proses berlanjut hingga ditemukan solusi atau tidak ada penerus yang tersisa. Kode juga mencatat jumlah node yang dikunjungi, waktu eksekusi, serta statistik ukuran beam untuk analisis performa. Dengan begitu, Beam Search efisien dalam memori dan waktu tetapi dengan risiko tidak selalu menemukan solusi yang optimal.

```
"use client";

import { Board } from "../models/Board";
import { MovementManager } from "../utils/Move";
import { SearchNode } from "../models/SearchNode";
import { HeuristicType } from "@/store/GameStore";
import { SearchResult } from "../models/SearchResult";
import { Heuristics } from "./heuristics";
import PriorityQueue from "ts-priority-queue";

// Beam Search Algorithm
export class BeamSearchAlgorithm {
  static search(
    initialBoard: Board,
    heuristicType: HeuristicType = HeuristicType.MANHATTAN,
    beamWidth: number = 5 // default
  ): SearchResult {
    const startTime = performance.now();

    // Initialize
    const startNode = new SearchNode(initialBoard);
    let beam: SearchNode[] = [startNode];

    const visited = new Set<string>();
    visited.add(startNode.getBoardString());

    let nodesVisited = 0;
    let maxBeamSize = 1;

    const beamSizeHistory: number[] = [1];
    let depthReached = 0;

    // Search loop
    while (beam.length > 0) {
      depthReached++;

      // Priority queue for successors
      const successorQueue = new PriorityQueue<SearchNode>({
        comparator: (a, b) => {
          const aHeuristic = Heuristics.calculateHeuristic(a.board, heuristicType);
          const bHeuristic = Heuristics.calculateHeuristic(b.board, heuristicType);
          return aHeuristic - bHeuristic;
        }
      });

      for (const node of beam) {
        nodesVisited++;

        if (node.board.isGoal()) {
          const endTime = performance.now();
          const executionTime = endTime - startTime;

          // Calculate average beam size
          const avgBeamSize = beamSizeHistory.reduce((a, b) => a + b, 0) / beamSizeHistory.length;

          console.log(`Beam Search found solution at depth ${depthReached}`);
          console.log(`Visited ${nodesVisited} nodes with max beam size ${maxBeamSize}`);
          console.log(`Average beam size: ${avgBeamSize.toFixed(2)}`);

          return {
            success: true,
            path: node.getPath(),
            nodesVisited,
            executionTime,
          };
        }

        // All possible moves
        const possibleMoves = MovementManager.getPossibleMoves(node.board);

        // Create successor nodes
        for (const move of possibleMoves) {
          const newBoard = MovementManager.applyMove(node.board, move);
          const boardString = newBoard
            .getGrid()
            .map((row) => row.join(""))
            .join("");

          if (visited.has(boardString)) {
            continue;
          }
          visited.add(boardString);

          // Create new node
          const newNode = new SearchNode(
            newBoard,
            node,
            move,
            node.cost + 1 // g(n) = parent's g(n) + 1
          );

          // Add to priority queue
          successorQueue.queue(newNode);
        }
      }

      // If no successors, failed
      if (successorQueue.length === 0) {
        const endTime = performance.now();

        return {
          success: false,
          path: [],
          nodesVisited,
          executionTime: endTime - startTime,
        };
      }

      // Top k nodes from priority queue
      beam = [];
      const count = Math.min(beamWidth, successorQueue.length);
      for (let i = 0; i < count; i++) {
        beam.push(successorQueue.dequeue());
      }

      maxBeamSize = Math.max(maxBeamSize, beam.length);
      beamSizeHistory.push(beam.length);
    }

    // Beam empty and no solution found
    const endTime = performance.now();
    return {
      success: false,
      path: [],
      nodesVisited,
      executionTime: endTime - startTime,
    };
  }
}
```

4.6 Board.ts

Board.ts berisi class Board yang merupakan representasi utama papan permainan Rush Hour. Class ini berisi informasi tentang Ukuran papan, posisi pieces (kendaraan), letak pintu keluar, logika untuk mendeteksi apakah sudah *goal* atau tidak, dan *getter-setter*.



```
import { Piece } from "./Piece";

export class Board {
    constructor(
        private rows: number,
        private cols: number,
        private grid: string[][],
        private pieces: Map<string, Piece>,
        private exitRow: number,
        private exitCol: number
    ) {}

    getRows(): number {
        return this.rows;
    }

    getCols(): number {
        return this.cols;
    }

    getGrid(): string[][] {
        return this.grid;
    }

    getPieces(): Map<string, Piece> {
        return this.pieces;
    }

    getExitRow(): number {
        return this.exitRow;
    }

    getExitCol(): number {
        return this.exitCol;
    }

    isGoal(): boolean {
        let primary: Piece | undefined;
        for (const piece of this.pieces.values()) {
            if (piece.isPrimary) {
                primary = piece;
                break;
            }
        }

        if (!primary) return false;

        // Horizontal
        if (primary.isHorizontal) {
            // Right exit
            if (this.exitCol === this.cols) {
                return primary.row === this.exitRow &&
                       primary.col + primary.length - 1 === this.cols - 1;
            }
            // Left exit
            else if (this.exitCol === -1) {
                return primary.row === this.exitRow && primary.col === 0;
            }
        }
        else {
            // Vertical
            // Bottom exit
            if (this.exitRow === this.rows) {
                return primary.col === this.exitCol &&
                       primary.row + primary.length - 1 === this.rows - 1;
            }
            // Top exit
            else if (this.exitRow === -1) {
                return primary.col === this.exitCol && primary.row === 0;
            }
        }

        return false;
    }

    copy(): Board {
        const newGrid = this.grid.map(row => [...row]);
        const newPieces = new Map<string, Piece>();
        for (const [key, piece] of this.pieces.entries()) {
            newPieces.set(key, piece.copy());
        }

        return new Board(this.rows, this.cols, newGrid, newPieces, this.exitRow, this.exitCol);
    }

    printBoard(): string {
        const output: string[] = [];

        // Top exit
        if (this.exitRow === -1) {
            const topLine = ' '.repeat(this.exitCol) + 'K';
            output.push(topLine);
        }

        // Grid rows
        for (let i = 0; i < this.rows; i++) {
            let line = "";
            if (this.exitCol === -1 && this.exitRow !== i) {
                line = " ";
            }

            // Left exit
            if (this.exitRow === i && this.exitCol === -1) {
                line += 'K';
            }

            for (let j = 0; j < this.cols; j++) {
                line += this.grid[i][j];
            }

            // Right exit
            if (this.exitRow === i && this.exitCol === this.cols) {
                line += 'K';
            }

            output.push(line);
        }

        // Bottom exit
        if (this.exitRow === this.rows) {
            const bottomLine = ' '.repeat(this.exitCol) + 'K';
            output.push(bottomLine);
        }

        return output.join('\n');
    }
}
```

4.7 Piece.ts

Piece.ts berisi class Piece yang merupakan representasi dari kendaraan pada permainan Rush Hour. Class ini berisi informasi seperti id, *length* (panjang *piece*), dan apakah *piece* adalah *primary*.	

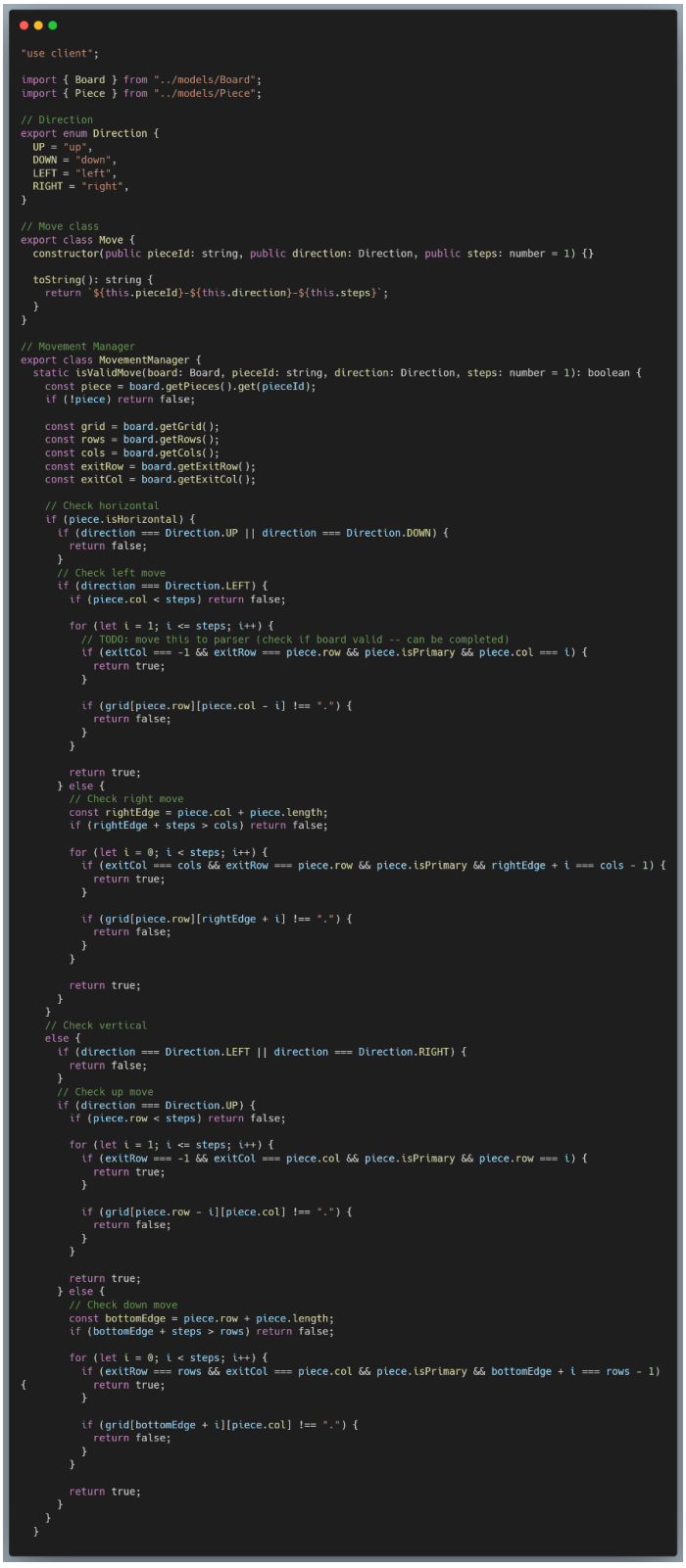
```ts
export class Piece {
    constructor(
        public id: string,
        public col: number,
        public row: number,
        public length: number,
        public isHorizontal: boolean,
        public isPrimary: boolean
    ) {}

    copy(): Piece {
        return new Piece(
            this.id,
            this.col,
            this.row,
            this.length,
            this.isHorizontal,
            this.isPrimary
        );
    }
}
```

4.8 SearchNode.ts

SearchNode.ts berisi class SearchNode. SearchNode digunakan untuk merepresentasikan sebuah state atau node dalam pohon pencarian.	

```ts
import { Board } from "./Board";
import { Move } from "../utils/Move";

export class SearchNode {
  public board: Board;
  public parent: SearchNode | null;
  public move: Move | null;
  public cost: number;

  constructor(board: Board, parent: SearchNode | null = null, move: Move | null = null, cost: numb
    this.board = board;
    this.parent = parent;
    this.move = move;
    this.cost = cost;
  }

  // Get path from the root to this node
  getPath(): Move[] {
    const path: Move[] = [];

    this.buildPath(path);

    return path;
  }

  // Helper method to build the path recursively
  private buildPath(path: Move[]): void {
    if (this.parent === null || this.move === null) {
      return; // Base case: reached root node
    }

    this.parent.buildPath(path);

    path.push(this.move);
  }

  // Get string representation of the board for checking visited states
  getBoardString(): string {
    return this.board.getGrid().map(row => row.join('')).join('');
  }
}
```

4.9 Move.ts

Move.ts berisi class Move yang merepresentasikan satu langkah yang dilakukan oleh kendaraan dan class MovementManager yang bertugas untuk mengevaluasi dan menerapkan langkah - langkah, termasuk mencari semua langkah yang valid dari state sekarang.	

```
"use client";

import { Board } from "../models/Board";
import { Piece } from "../models/Piece";

// Direction
export enum Direction {
  UP = "up",
  DOWN = "down",
  LEFT = "left",
  RIGHT = "right",
}

// Move class
export class Move {
  constructor(public pieceId: string, public direction: Direction, public steps: number = 1) {}

  toString(): string {
    return `${this.pieceId}-${this.direction}-${this.steps}`;
  }
}

// Movement Manager
export class MovementManager {
  static isValidMove(board: Board, pieceId: string, direction: Direction, steps: number = 1): boolean {
    const piece = board.getPieces().get(pieceId);
    if (!piece) return false;

    const grid = board.getGrid();
    const rows = board.getRows();
    const cols = board.getCols();
    const exitRow = board.getExitRow();
    const exitCol = board.getExitCol();

    // Check horizontal
    if (piece.isHorizontal) {
      if (direction === Direction.UP || direction === Direction.DOWN) {
        return false;
      }
      // Check left move
      if (direction === Direction.LEFT) {
        if (piece.col < steps) return false;

        for (let i = 1; i <= steps; i++) {
          // TODO: move this to parser (check if board valid -- can be completed)
          if (exitCol === -1 && exitRow === piece.row && piece.isPrimary && piece.col === i) {
            return true;
          }

          if (grid[piece.row][piece.col - i] !== ".") {
            return false;
          }
        }

        return true;
      } else {
        // Check right move
        const rightEdge = piece.col + piece.length;
        if (rightEdge + steps > cols) return false;

        for (let i = 0; i < steps; i++) {
          if (exitCol === cols && exitRow === piece.row && piece.isPrimary && rightEdge + i === cols - 1) {
            return true;
          }

          if (grid[piece.row][rightEdge + i] !== ".") {
            return false;
          }
        }

        return true;
      }
    }
    // Check vertical
    else {
      if (direction === Direction.LEFT || direction === Direction.RIGHT) {
        return false;
      }
      // Check up move
      if (direction === Direction.UP) {
        if (piece.row < steps) return false;

        for (let i = 1; i <= steps; i++) {
          if (exitRow === -1 && exitCol === piece.col && piece.isPrimary && piece.row === i) {
            return true;
          }

          if (grid[piece.row - i][piece.col] !== ".") {
            return false;
          }
        }

        return true;
      } else {
        // Check down move
        const bottomEdge = piece.row + piece.length;
        if (bottomEdge + steps > rows) return false;

        for (let i = 0; i < steps; i++) {
          if (exitRow === rows && exitCol === piece.col && piece.isPrimary && bottomEdge + i === rows - 1)
{
            return true;
          }

          if (grid[bottomEdge + i][piece.col] !== ".") {
            return false;
          }
        }

        return true;
      }
    }
  }
```

```
// Apply move
static applyMove(board: Board, move: Move): Board {
  const newBoard = board.copy();
  const pieces = newBoard.getPieces();
  const piece = pieces.get(move.pieceId);

  // Return original board if not valid
  if (!piece || !this.isValidMove(board, move.pieceId, move.direction, move.steps)) {
    return board;
  }

  const grid = newBoard.getGrid();

  // Remove piece
  this.clearPieceFromGrid(grid, piece);

  // Update piece position
  switch (move.direction) {
    case Direction.UP:
      piece.row -= move.steps;
      break;
    case Direction.DOWN:
      piece.row += move.steps;
      break;
    case Direction.LEFT:
      piece.col -= move.steps;
      break;
    case Direction.RIGHT:
      piece.col += move.steps;
      break;
  }

  // Place piece to new position
  this.placePieceOnGrid(grid, piece);

  return newBoard;
}

// All possible moves, including multiple steps (separated)
static getPossibleMoves(board: Board): Move[] {
  const moves: Move[] = [];
  const pieces = board.getPieces();

  // Loop each piece
  for (const [id] of pieces) {
    // Loop each direction
    for (const direction of Object.values(Direction)) {
      const dir = direction as Direction;
      const maxSteps = this.getMaxSteps(board, id, dir);

      // Loop each possible steps
      for (let steps = 1; steps <= maxSteps; steps++) {
        if (this.isValidMove(board, id, dir, steps)) {
          moves.push(new Move(id, dir, steps));
        }
      }
    }
  }

  return moves;
}

// Clear piece from grid
private static clearPieceFromGrid(grid: string[][], piece: Piece): void {
  if (piece.isHorizontal) {
    for (let c = piece.col; c < piece.col + piece.length; c++) {
      grid[piece.row][c] = ".";
    }
  } else {
    for (let r = piece.row; r < piece.row + piece.length; r++) {
      grid[r][piece.col] = ".";
    }
  }
}

// Place piece on grid
private static placePieceOnGrid(grid: string[][], piece: Piece): void {
  if (piece.isHorizontal) {
    for (let c = piece.col; c < piece.col + piece.length; c++) {
      grid[piece.row][c] = piece.id;
    }
  } else {
    for (let r = piece.row; r < piece.row + piece.length; r++) {
      grid[r][piece.col] = piece.id;
    }
  }
}

// Max possibple steps
static getMaxSteps(board: Board, pieceId: string, direction: Direction): number {
  const piece = board.getPieces().get(pieceId);
  if (!piece) return 0;

  const grid = board.getGrid();
  const rows = board.getRows();
  const cols = board.getCols();
  const exitRow = board.getExitRow();
  const exitCol = board.getExitCol();
  let steps = 0;
```

```
    // Horizontal piece
    if (piece.isHorizontal) {
      if (direction === Direction.LEFT) {
        for (let c = piece.col - 1; c >= 0; c--) {
          if (grid[piece.row][c] === ".") {
            steps++;
          } else {
            break;
          }
        }

        // Left exit
        if (piece.isPrimary && exitCol === -1 && exitRow === piece.row && steps >= piece.col) {
          return piece.col;
        }
      } else if (direction === Direction.RIGHT) {
        for (let c = piece.col + piece.length; c < cols; c++) {
          if (grid[piece.row][c] === ".") {
            steps++;
          } else {
            break;
          }
        }

        // Right exit
        if (piece.isPrimary && exitCol === cols && exitRow === piece.row && piece.col + piece.length + steps >=
cols) {
          return cols - (piece.col + piece.length); // Can move all the way to the exit
        }
      }
    } else {
      // Vertical piece
      if (direction === Direction.UP) {
        for (let r = piece.row - 1; r >= 0; r--) {
          if (grid[r][piece.col] === ".") {
            steps++;
          } else {
            break;
          }
        }

        // Top exit
        if (piece.isPrimary && exitRow === -1 && exitCol === piece.col && steps >= piece.row) {
          return piece.row;
        }
      } else if (direction === Direction.DOWN) {
        for (let r = piece.row + piece.length; r < rows; r++) {
          if (grid[r][piece.col] === ".") {
            steps++;
          } else {
            break;
          }
        }

        // Bottom exit
        if (piece.isPrimary && exitRow === rows && exitCol === piece.col && piece.row + piece.length + steps >=
rows) {
          return rows - (piece.row + piece.length);
        }
      }
    }

    return steps;
  }
}
```

4.10 GameStore.ts

GameStore.ts berfungsi untuk menyimpan seluruh state dan logic interaktif yang berkaitan dengan program.	

```
"use client";

import { create } from "zustand";
import { Board } from "@/lib/models/Board";
import { Move, MovementManager } from "@/lib/utils/Move";
import { UCSAlgorithm } from "@/lib/algorithms/ucs";
import { AStarAlgorithm } from "@/lib/algorithms/a-star";
import { GBFSAlgorithm } from "@/lib/algorithms/gbfs";
import { SearchResult } from "@/lib/models/SearchResult";
import { BeamSearchAlgorithm } from "@/lib/algorithms/beamSearch";

// Algorithm Type
export enum AlgorithmType {
  UCS = "ucs",
  GBFS = "gbfs",
  ASTAR = "a-star",
  BEAM = "beam-search",
}

// Heuristic Type
export enum HeuristicType {
  MANHATTAN = "manhattan",
  BLOCKING = "blocking",
  DISTANCE = "distance",
}

// Game State
interface GameState {
  board: Board | null;
  selectedAlgorithm: AlgorithmType;
  selectedHeuristic: HeuristicType;
  solution: SearchResult | null;
  solutionPath: Move[];
  currentMoveIndex: number;
  isAnimating: boolean;
  animationSpeed: number;
  beamWidth: number;

  setBoard: (board: Board) => void;
  setAlgorithm: (algorithm: AlgorithmType) => void;
  setHeuristic: (heuristic: HeuristicType) => void;
  solvePuzzle: () => void;
  resetSolution: () => void;
  nextMove: () => void;
  prevMove: () => void;
  jumpToMove: (index: number) => void;
  startAnimation: () => void;
  stopAnimation: () => void;
  setAnimationSpeed: (speed: number) => void;
  getCurrentBoard: () => Board | null;
  setBeamWidth: (beam: number) => void;
}

// Create store
const useGameStore = create<GameState>((set, get) => ({
  board: null,
  selectedAlgorithm: AlgorithmType.UCS,
  selectedHeuristic: HeuristicType.MANHATTAN,
  solution: null,
  solutionPath: [],
  currentMoveIndex: -1,
  isAnimating: false,
  animationSpeed: 500, // milliseconds per move
  beamWidth: 5, // default

  setBoard: (board) =>
    set({
      board,
      solution: null,
      solutionPath: [],
      currentMoveIndex: -1,
      isAnimating: false,
    }),

  setAlgorithm: (algorithm) => set({ selectedAlgorithm: algorithm }),

  setHeuristic: (heuristic) => set({ selectedHeuristic: heuristic }),

  solvePuzzle: () => {
    const { board, selectedAlgorithm, selectedHeuristic, beamWidth } = get();
    if (!board) return;

    let solution: SearchResult | null = null;

    switch (selectedAlgorithm) {
      case AlgorithmType.UCS:
        solution = UCSAlgorithm.search(board);
        break;
      case AlgorithmType.ASTAR:
        solution = AStarAlgorithm.search(board, selectedHeuristic);
        break;
      case AlgorithmType.GBFS:
        solution = GBFSAlgorithm.search(board, selectedHeuristic);
        break;
      case AlgorithmType.BEAM:
        solution = BeamSearchAlgorithm.search(board, selectedHeuristic, beamWidth);
        break;
      default:
        solution = UCSAlgorithm.search(board);
    }
```

```
    set({
      solution,
      solutionPath: solution.path,
      currentMoveIndex: -1,
    });
  },

  resetSolution: () =>
    set({
      solution: null,
      solutionPath: [],
      currentMoveIndex: -1,
      isAnimating: false,
    }),

  nextMove: () => {
    const { board, solutionPath, currentMoveIndex } = get();
    if (!board || currentMoveIndex >= solutionPath.length - 1) return;

    const nextIndex = currentMoveIndex + 1;
    set({ currentMoveIndex: nextIndex });
  },

  prevMove: () => {
    const { currentMoveIndex } = get();
    if (currentMoveIndex <= -1) return;

    const prevIndex = currentMoveIndex - 1;
    set({ currentMoveIndex: prevIndex });
  },

  jumpToMove: (index) => {
    const { solutionPath } = get();
    if (index < -1 || index >= solutionPath.length) return;

    set({ currentMoveIndex: index });
  },

  startAnimation: () => {
    const { isAnimating, solutionPath, currentMoveIndex } = get();
    if (isAnimating || currentMoveIndex >= solutionPath.length - 1) return;

    set({ isAnimating: true });

    const animate = () => {
      const { isAnimating, solutionPath, currentMoveIndex, animationSpeed } =
get();
      if (!isAnimating || currentMoveIndex >= solutionPath.length - 1) {
        set({ isAnimating: false });
        return;
      }

      // Move to next step
      get().nextMove();

      // Schedule next animation frame
      setTimeout(animate, animationSpeed);
    };

    // Start animation
    setTimeout(animate, get().animationSpeed);
  },

  stopAnimation: () => set({ isAnimating: false }),

  setAnimationSpeed: (speed) => set({ animationSpeed: speed }),

  getCurrentBoard: () => {
    const { board, solutionPath, currentMoveIndex } = get();
    if (!board || currentMoveIndex < 0) return board;

    // Apply all moves
    let currentBoard = board.copy();
    for (let i = 0; i <= currentMoveIndex; i++) {
      const move = solutionPath[i];
      currentBoard = MovementManager.applyMove(currentBoard, move);
    }

    return currentBoard;
  },
  setBeamWidth: (beam: number) => set({ beamWidth: beam }),
}));

export default useGameStore;
```

4.11 Parser.ts

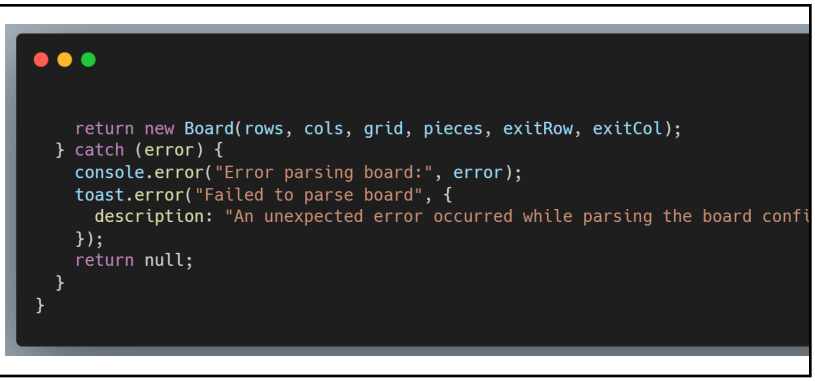
Kode ini berfungsi untuk membaca dan memvalidasi konfigurasi papan permainan dari string input yang memuat dimensi papan, jumlah potongan, serta tata letak grid, termasuk posisi keluaran (*exit*) yang ditandai dengan huruf 'K'. Fungsi ini memastikan format input benar, seperti jumlah baris dan kolom yang valid, jumlah potongan yang sesuai, serta memastikan potongan-potongan (*pieces*) terbentuk secara terhubung dan hanya berbentuk garis horizontal atau vertikal tanpa celah. Selain itu, fungsi memeriksa keberadaan *primary piece* ('P') yang harus tersambung dan memastikan posisi *exit* sejajar dengan *primary piece* sesuai orientasinya. Jika seluruh validasi terpenuhi, fungsi mengembalikan objek *Board* lengkap dengan data grid, potongan, dan posisi keluaran. Jika terdapat kesalahan, fungsi menampilkan pesan error menggunakan *toast sonner* dan mengembalikan null.

```
import { toast } from "sonner";
import { Board } from "../models/Board";
import { Piece } from "../models/Piece";

class ConcretePiece implements Piece {
  constructor(
    public id: string,
    public row: number,
    public col: number,
    public length: number,
    public isHorizontal: boolean,
    public isPrimary: boolean
  ) {}

  copy(): Piece {
    return new ConcretePiece(this.id, this.row, this.col, this.length, this.isHorizontal, this.isPrimary);
  }
}

export function parseBoardFromString(content: string): Board | null {
  try {
    const lines = content.split(/\r?\n/).filter((line) => line.trim() !== "");

    // At least 3 lines (dimensions, pieceCount, and at least one grid line)
    if (lines.length < 3) {
      toast.error("Invalid file format", { description: "File does not contain enough lines." });
      return null;
    }

    // Check dimension number
    const [A, B] = lines[0].split(" ").map(Number);
    if (isNaN(A) || isNaN(B) || A <= 0 || B <= 0) {
      toast.error("Invalid dimensions", { description: "Board dimensions must be positive numbers." });
      return null;
    }

    const rows = A;
    const cols = B;

    // Check number of pieces
    const N = parseInt(lines[1], 10);
    if (isNaN(N) || N < 0) {
      toast.error("Invalid piece count", { description: "Number of pieces must be a non-negative integer." });
      return null;
    }

    // Check if file has enough lines to contain the complete board configuration
    // Minimum expected: 2 (dimensions, pieceCount) + rows (grid lines)
    // Maximum expected: 2 + rows + 2 (possible K on top and bottom)
    if (lines.length < 2 + rows) {
      toast.error("Invalid file format", {
        description: `The file does not contain enough lines for the board configuration. Expected at least ${2 +
rows} lines.`,
      });
      return null;
    }

    if (lines.length > 2 + rows + 2) {
      toast.error("Invalid file format", {
        description: `The file contains too many lines. Expected at most ${2 + rows + 2} lines.`,
      });
      return null;
    }

    // Identify possible K positions in the file
    let hasTopExit = false;
    let hasBottomExit = false;

    // Top Exit
    if (lines[2].includes("K") && !lines[2].includes(".") && !lines[2].match(/[A-Z]/g)?.filter((char) => char !==
"K").length) {
      hasTopExit = true;
    }

    // Bottom Exit
    const lastLine = lines[lines.length - 1];
    if (lastLine.includes("K") && !lastLine.includes(".") && !lastLine.match(/[A-Z]/g)?.filter((char) => char !==
"K").length) {
        hasBottomExit = true;
    }

    // Grid line start based on exit position
    let gridStartIndex = 2;
    if (hasTopExit) {
      gridStartIndex = 3;
    }

    // Extract grid lines
    const possibleGridEndIndex = gridStartIndex + rows;
    if (possibleGridEndIndex > lines.length) {
      toast.error("Invalid file format", {
        description: "The file does not contain enough lines for the grid.",
      });
      return null;
    }

    const gridLines = lines.slice(gridStartIndex, possibleGridEndIndex);

    // Check grid lines length (cols)
    for (let i = 0; i < gridLines.length; i++) {
      const line = gridLines[i];
      const nonWhitespaceChars = line.replace(/\s/g, "").length;
      if (nonWhitespaceChars < cols) {
        toast.error("Invalid grid line", {
          description: `Line ${i + 1} has insufficient content. Expected at least ${cols} grid cells.`,
        });
        return null;
      }
    }
```

```
let exitRow = -1;
let exitCol = -1;

// Top exit
if (hasTopExit) {
  exitRow = -1;
  exitCol = lines[2].indexOf("K");
  if (exitCol < 0 || exitCol >= cols) {
    toast.error("Invalid exit position", {
      description: "Top exit (K) must be within the column range of the grid.",
    });
    return null;
  }
}

// Bottom exit
if (hasBottomExit) {
  exitRow = rows;
  exitCol = lines[lines.length - 1].indexOf("K");
  if (exitCol < 0 || exitCol >= cols) {
    toast.error("Invalid exit position", {
      description: "Bottom exit (K) must be within the column range of the grid.",
    });
    return null;
  }
}

// Left or right exit
if (exitRow === -1 && exitCol === -1) {
  for (let i = 0; i < gridLines.length; i++) {
    const line = gridLines[i];

    // Left exit
    if (line[0] === "K") {
      exitRow = i;
      exitCol = -1;
      break;
    }

    // Right exit
    const nonWhitespaceChars = line.replace(/\s/g, "");
    if (nonWhitespaceChars.length > cols && nonWhitespaceChars[cols] === "K") {
      exitRow = i;
      exitCol = cols;
      break;
    }
  }
}

// Validate exit
if (exitRow === -1 && exitCol === -1) {
  toast.error("No exit found", {
    description: "The board must have an exit marked with 'K' at one of the edges.",
  });
  return null;
}

const grid: string[][] = Array(rows)
  .fill(0)
  .map(() => Array(cols).fill("."));

for (let i = 0; i < rows; i++) {
  const line = gridLines[i];
  const nonWhitespaceLine = line.replace(/\s/g, "");

  const startIndex = exitCol === -1 && exitRow === i ? 1 : 0;

  let gridColIndex = 0;
  for (let j = startIndex; j < nonWhitespaceLine.length && gridColIndex < cols; j++) {
    if (exitCol === cols && exitRow === i && j === cols) {
      continue;
    }

    const char = nonWhitespaceLine[j];

    if (char === "." || char.match(/[A-Z0-9]/i)) {
      grid[i][gridColIndex] = char;
      gridColIndex++;
    }
  }
}

// Check K not inside grid
for (let i = 0; i < rows; i++) {
  for (let j = 0; j < cols; j++) {
    if (grid[i][j] === "K") {
      toast.error("Invalid exit position", {
        description: "The exit 'K' cannot be inside the grid. It must be placed on the border.",
      });
      return null;
    }
  }
}
```

```
    // Find connected components
    const pieces = new Map<string, Piece>();
    const symbolComponents = new Map<string, Array<Array<{ row: number; col: number }>>>();

    // Collect all unique symbols
    for (let i = 0; i < rows; i++) {
      for (let j = 0; j < cols; j++) {
        const symbol = grid[i][j];
        if (symbol === ".") continue;

        if (!symbolComponents.has(symbol)) {
          symbolComponents.set(symbol, []);
        }
      }
    }

    // Helper to find connected components
    const visited = new Set<string>();

    function findConnected(row: number, col: number, symbol: string, component: Array<{ row: number; col: number }>)
{
      const key = `${row},${col}`;
      if (row < 0 || row >= rows || col < 0 || col >= cols || visited.has(key) || grid[row][col] !== symbol) {
        return;
      }

      visited.add(key);
      component.push({ row, col });

      findConnected(row + 1, col, symbol, component);
      findConnected(row - 1, col, symbol, component);
      findConnected(row, col + 1, symbol, component);
      findConnected(row, col - 1, symbol, component);
    }

    // Find all connected components
    for (let i = 0; i < rows; i++) {
      for (let j = 0; j < cols; j++) {
        const symbol = grid[i][j];
        if (symbol === "." || visited.has(`${i},${j}`)) continue;

        const component: Array<{ row: number; col: number }> = [];
        findConnected(i, j, symbol, component);

        if (component.length > 0) {
          symbolComponents.get(symbol)?.push(component);
        }
      }
    }

    // Check primary piece
    if (!symbolComponents.has("P")) {
      toast.error("No primary piece found", {
        description: "The board must contain exactly one primary piece marked with 'P'.",
      });
      return null;
    }

    const primaryPieceComponents = symbolComponents.get("P") || [];
    if (primaryPieceComponents.length === 0) {
      toast.error("No primary piece found", {
        description: "The board must contain exactly one primary piece marked with 'P'.",
      });
      return null;
    }

    if (primaryPieceComponents.length > 1) {
      toast.error("Multiple primary pieces", {
        description: `Found ${primaryPieceComponents.length} separate primary pieces. The board must contain exactly
one connected primary piece.`,
      });
      return null;
    }

    // Check if any symbol has multiple disconnected components
    for (const [symbol, components] of symbolComponents.entries()) {
      if (components.length > 1) {
        toast.error("Disconnected piece", {
          description: `Piece '${symbol}' appears in ${components.length} separate locations. Each piece must be
connected.`,
        });
        return null;
      }
    }

    // Check number of non-primary pieces
    const nonPrimaryPieceCount = symbolComponents.size - (symbolComponents.has("P") ? 1 : 0);
    if (N !== nonPrimaryPieceCount) {
      toast.error("Piece count mismatch", {
        description: `The file specifies ${N} non-primary pieces but ${nonPrimaryPieceCount} were found in the
grid.`,
      });
      return null;
    }

    // Create piece objects and check orientation
    for (const [symbol, components] of symbolComponents.entries()) {
      if (components.length === 0) continue;

      const cells = components[0];

      const rowIndices = new Set<number>();
      const colIndices = new Set<number>();

      for (const { row, col } of cells) {
        rowIndices.add(row);
        colIndices.add(col);
      }
```

```
      // Find the top-left corner
      cells.sort((a, b) => {
        if (a.row !== b.row) return a.row - b.row;
        return a.col - b.col;
      });

      const topLeft = cells[0];

      // Check orientation
      const isHorizontal = rowIndices.size === 1;
      const isVertical = colIndices.size === 1;

      // (horizontal xor vertical)
      if (!isHorizontal && !isVertical) {
        toast.error("Invalid piece shape", {
          description: `Piece '${symbol}' is not a straight line. Pieces must be either horizontal or vertical.`,
        });
        return null;
      }

      // Check continuity
      if (isHorizontal) {
        const sortedCols = Array.from(colIndices).sort((a, b) => a - b);
        for (let i = 1; i < sortedCols.length; i++) {
          if (sortedCols[i] !== sortedCols[i - 1] + 1) {
            toast.error("Discontinuous piece", {
              description: `Horizontal piece '${symbol}' has gaps between its cells.`,
            });
            return null;
          }
        }
      } else {
        const sortedRows = Array.from(rowIndices).sort((a, b) => a - b);
        for (let i = 1; i < sortedRows.length; i++) {
          if (sortedRows[i] !== sortedRows[i - 1] + 1) {
            toast.error("Discontinuous piece", {
              description: `Vertical piece '${symbol}' has gaps between its cells.`,
            });
            return null;
          }
        }
      }

      // Create piece
      pieces.set(symbol, new ConcretePiece(symbol, topLeft.row, topLeft.col, cells.length, isHorizontal, symbol ===
"P"));
    }

    const primaryPiece = pieces.get("P");
    if (!primaryPiece) {
      toast.error("No primary piece found", {
        description: "The board must contain exactly one primary piece marked with 'P'.",
      });
      return null;
    }

    // Check alignment
    // Horizontal primary piece
    if (primaryPiece.isHorizontal) {
      //  Left/right exit
      if (exitCol !== -1 && exitCol !== cols) {
        toast.error("Exit not aligned with primary piece", {
          description: "For a horizontal primary piece, the exit must be on the left or right side of the board.",
        });
        return null;
      }

      if ((exitCol === -1 || exitCol === cols) && exitRow !== primaryPiece.row) {
        toast.error("Exit not aligned with primary piece", {
          description: "For a horizontal primary piece, the exit must be on the same row.",
        });
        return null;
      }
    } else {
      // Vertical primary piece
      // Top/bottom exit
      if (exitRow !== -1 && exitRow !== rows) {
        toast.error("Exit not aligned with primary piece", {
          description: "For a vertical primary piece, the exit must be on the top or bottom side of the board.",
        });
        return null;
      }

      if ((exitRow === -1 || exitRow === rows) && exitCol !== primaryPiece.col) {
        toast.error("Exit not aligned with primary piece", {
          description: "For a vertical primary piece, the exit must be on the same column.",
        });
        return null;
      }
    }

    // All passed
    toast.success("Board loaded successfully", {
      description: `${rows}x${cols} board with ${pieces.size} pieces and valid exit configuration.`,
    });
```

```
    return new Board(rows, cols, grid, pieces, exitRow, exitCol);
  } catch (error) {
    console.error("Error parsing board:", error);
    toast.error("Failed to parse board", {
      description: "An unexpected error occurred while parsing the board confi
    });
    return null;
  }
}
```

4.12 Page.tsx

Kode ini merupakan kode utama yang sebagai *home page* dari aplikasi ini. *Home page* memuat komponen Navbar, InputSearch, dan Footer.	

```
import Footer from "@/components/Footer";
import InputSearch from "@/components/InputSearch";
import Navbar from "@/components/Navbar";

export default function Home() {
  return (
    <>
      <Navbar />
      <div className="min-h-screen w-full bg-rush-primary p-10 flex justify-center items-center mt-14">
        <InputSearch />
      </div>
      <Footer />
    </>
  );
}
```

4.13 layout.tsx

Kode ini merupakan *wrapper* atau sebagai layout pengaturan dari semua kode pada aplikasi ini. Terdapat pula metadata dari aplikasi dan *wrapper* untuk *toaster*.	

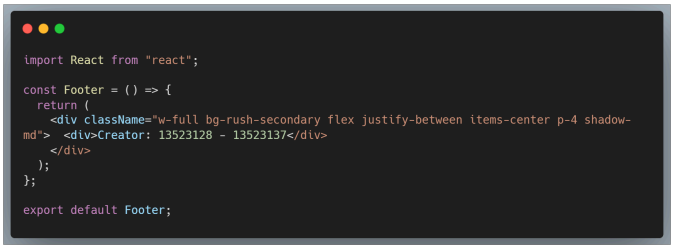
```
import type { Metadata } from "next";
import { Geist, Geist_Mono } from "next/font/google";
import "./globals.css";
import ToastProvider from "@/components/ToastProvider";

const geistSans = Geist({
  variable: "--font-geist-sans",
  subsets: ["latin"],
});

const geistMono = Geist_Mono({
  variable: "--font-geist-mono",
  subsets: ["latin"],
});

export const metadata: Metadata = {
  title: "RushPush",
  description: "Rush Hour Solver with Pathfinding",
};

export default function RootLayout({
  children,
}: Readonly<{
  children: React.ReactNode;
}>) {
  return (
    <html lang="en">
      <body className={`${geistSans.variable} ${geistMono.variable} antialiased`}>
        <ToastProvider>{children}</ToastProvider>
      </body>
    </html>
  );
}
```

4.14 AlgoInput.tsx

Kode ini merupakan komponen yang mengurus terkait input tipe algoritma pencarian rute yang akan digunakan. Khusus pada algoritma *Beam Search*, terdapat pula input untuk *beamWidth*	

```tsx
"use client";

import useGameStore from "@/store/GameStore";
import { AlgorithmType } from "@/store/GameStore";
import React, { ChangeEvent } from "react";

interface Algorithm {
  value: string;
  label: string;
}

const AlgoInput = () => {
  const { selectedAlgorithm, setAlgorithm, setBeamWidth, beamWidth } = useGameStore();

  // List of algorithms
  const algorithms: Algorithm[] = [
    { value: AlgorithmType.UCS, label: "Uniform Cost Search" },
    { value: AlgorithmType.GBFS, label: "Greedy Best-First Search" },
    { value: AlgorithmType.ASTAR, label: "A* Search" },
    { value: AlgorithmType.BEAM, label: "Beam Search" },
  ];

  const handleChange = (e: ChangeEvent<HTMLSelectElement>): void => {
    e.preventDefault();
    const selectedValue = e.target.value as AlgorithmType;
    setAlgorithm(selectedValue);
  };

  const handleBeamWidthChange = (e: React.ChangeEvent<HTMLInputElement>): void => {
    e.preventDefault();
    const value = parseInt(e.target.value, 10);
    if (!isNaN(value) && value > 0) {
      setBeamWidth(value);
    }
  };

  return (
    <div className="bg-rush-primary p-5 rounded-2xl w-full text-rush-accent-1">
      <label className="block text-sm font-medium mb-2">Select Algorithm</label>
      <div className="">
        <select
          id="algorithm-select"
          value={selectedAlgorithm}
          onChange={handleChange}
          className="w-full p-2 border border-gray-300 rounded-md shadow-sm focus:border-blue-500 focus:outline-
none">
          {algorithms.map((algo) => (
            <option key={algo.value} value={algo.value}>
              {algo.label}
            </option>
          ))}
        </select>
      </div>
      {selectedAlgorithm && (
        <p className="mt-2 text-sm text-gray-600">
          Selected: {algorithms.find((algo) => algo.value === selectedAlgorithm)?.label}
        </p>
      )}
      {selectedAlgorithm === AlgorithmType.BEAM && (
        <div className="mt-5">
          <label htmlFor="beam-width" className="block text-sm font-medium mb-2">
            Beam Width
          </label>
          <div className="flex items-center gap-2">
            <input
              id="beam-width"
              type="number"
              min="1"
              max="1000"
              value={beamWidth}
              onChange={handleBeamWidthChange}
              className="w-full p-2 border border-gray-300 rounded-md shadow-sm focus:border-blue-500 focus:outline-
none"
            />
          </div>
          <p className="mt-2 text-sm text-gray-600">
            Beam width controls how many nodes to keep at each search level.
          </p>
          <div className="mt-2">
            <input
              type="range"
              min="5"
              max="500"
              step="5"
              value={beamWidth}
              onChange={handleBeamWidthChange}
              className="w-full h-2 bg-gray-200 rounded-lg appearance-none cursor-pointer"
            />
            <div className="flex justify-between text-xs text-gray-500 mt-1">
              <span>Faster (5)</span>
              <span>More thorough (500)</span>
            </div>
          </div>
        </div>
      )}
    </div>
  );
};

export default AlgoInput;
```

4.15 Footer.tsx

Kode ini merupakan komponen footer yang berisi data NIM pembuat aplikasi ini.	

```tsx
import React from "react";

const Footer = () => {
  return (
    <div className="w-full bg-rush-secondary flex justify-between items-center p-4 shadow-
md">  <div>Creator: 13523128 - 13523137</div>
    </div>
  );
};

export default Footer;
```

4.16 HeuristicInput.tsx

Kode ini merupakan komponen untuk melakukan input tipe heuristik yang akan digunakan.	

```tsx
"use client";

import useGameStore from "@/store/GameStore";
import { HeuristicType } from "@/store/GameStore";
import React, { ChangeEvent } from "react";

interface Heuristic {
  value: string;
  label: string;
}

const HeuristicInput = () => {
  const setHeuristic = useGameStore((state) => state.setHeuristic);
  const selectedHeuristic = useGameStore((state) => state.selectedHeuristic);

  // List of heuristics
  const heuristics: Heuristic[] = [
    { value: HeuristicType.MANHATTAN, label: "Manhattan Distance" },
    { value: HeuristicType.BLOCKING, label: "Blocking Pieces" },
    { value: HeuristicType.DISTANCE, label: "Distance to Exit" },
  ];

  const handleChange = (e: ChangeEvent<HTMLSelectElement>): void => {
    e.preventDefault();
    const selectedValue = e.target.value as HeuristicType;
    setHeuristic(selectedValue);
  };

  return (
    <div className="bg-rush-primary p-5 rounded-2xl w-full text-rush-accent-1">
      <label className="block text-sm font-medium mb-2">Select Heuristic</label>
      <div className="">
        <select
          id="heuristic-select"
          value={selectedHeuristic}
          onChange={handleChange}
          className="w-full p-2 border border-gray-300 rounded-md shadow-sm focus:border-blue-500 focus:outline-
none">
          {heuristics.map((algo) => (
            <option key={algo.value} value={algo.value}>
              {algo.label}
            </option>
          ))}
        </select>
      </div>
      {selectedHeuristic && (
        <p className="mt-2 text-sm text-gray-600">
          Selected: {heuristics.find((algo) => algo.value === selectedHeuristic)?.label}
        </p>
      )}
    </div>
  );
};

export default HeuristicInput;
```

4.17 InputSearch.tsx

Kode ini merupakan komponen *wrapper* untuk input algoritma, file testcase, dan heuristik. Kode ini mengecualikan input heuristik dari algoritma *UCS* dan *Beam Search*. Kode ini juga *wrapper* untuk komponen solusi jika solusi telah ditemukan.	

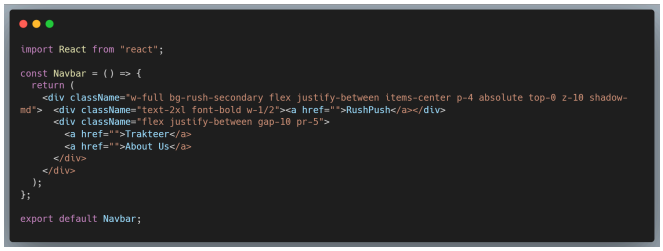
```tsx
"use client";

import React from "react";
import Loader from "./Loader";
import AlgoInput from "./AlgoInput";
import HeuristicInput from "./HeuristicInput";
import RenderBoard from "./RenderBoard";
import useGameStore, { AlgorithmType } from "@/store/GameStore";
import SolutionDisplay from "./Solution";

const InputSearch = () => {
  const { solvePuzzle, solution, resetSolution, selectedAlgorithm } = useGameStore();

  return (
    <div className="w-2/3 bg-rush-secondary rounded-2xl p-5 flex flex-col justify-center items-center">
      <div className="text-3xl font-bold">RushPush Solver</div>
      <div className="w-full p-5 flex justify-center items-center gap-10">
        <div className="flex flex-col gap-5 w-3/5">
          <Loader />
          {!solution && <RenderBoard activeMovePiece={null} />}
        </div>
        <div className="flex flex-col justify-center items-center gap-5 w-2/5">
          <AlgoInput />
          {selectedAlgorithm !== AlgorithmType.UCS && selectedAlgorithm !== AlgorithmType.BEAM && <HeuristicInput
/>}
          {!solution && (
            <button
              className="py-2 px-4 w-1/2 bg-rush-accent-2 rounded-3xl text-rush-secondary font-bold hover:cursor-
pointer hover:bg-rush-primary/80 hover:text-rush-accent-1"
              onClick={solvePuzzle}>
              Start RushPush!
            </button>
          )}
        </div>
      </div>

      {solution && (
        <div className="w-full flex flex-col justify-center items-center gap-5">
          <SolutionDisplay />
          <button
            onClick={resetSolution}
            className="py-2 px-4 w-1/3 bg-rush-accent-2 rounded-3xl text-rush-secondary font-bold hover:cursor-
pointer hover:bg-rush-primary/80 hover:text-rush-accent-1">
            Reset
          </button>
        </div>
      )}
    </div>
  );
};

export default InputSearch;
```

4.18 Loader.tsx

Kode ini merupakan komponen yang akan menerima file input *test case* berupa .txt, memprosesnya dengan fungsi dari *parser*, dan menyimpan *state*-nya pada *GameStore*.	

```tsx
"use client";

import { parseBoardFromString } from "@/lib/utils/Parser";
import useGameStore from "@/store/GameStore";
import React from "react";
import { toast } from "sonner";

const Loader = () => {
  const setBoard = useGameStore((state) => state.setBoard);

  const handleFileChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    const file = e.target.files?.[0];
    if (file && file.type === "text/plain") {
      const reader = new FileReader();
      reader.onload = (event) => {
        const text = event.target?.result as string;
        const board = parseBoardFromString(text);
        if (board) {
          setBoard(board);
        }
      };
      reader.readAsText(file);
    } else {
      toast.warning("Please upload a .txt file");
    }
  };

  return (
    <div className="p-6 h-full w-full rounded-2xl shadow-md bg-rush-primary text-rush-secondary">
      <h2 className="text-2xl font-semibold mb-4">Insert Game Configuration {`(.txt)`}</h2>

      <label className="block mb-4">
        <input
          type="file"
          accept=".txt"
          onChange={handleFileChange}
          className="block w-full text-sm text-rush-secondary
                     file:mr-4 file:py-2 file:px-4
                     file:rounded-full file:border-0
                     file:text-sm file:font-semibold
                     file:bg-rush-accent-2 file:text-rush-secondary
                     hover:file:bg-rush-accent-2/90 file:hover:cursor-pointer"
        />
      </label>
    </div>
  );
};

export default Loader;
```

4.19 Navbar.tsx

Kode ini merupakan komponen Navbar yang berisi nama aplikasi dan link-link terkait.	

```tsx
import React from "react";

const Navbar = () => {
  return (
    <div className="w-full bg-rush-secondary flex justify-between items-center p-4 absolute top-0 z-10 shadow-
md">  <div className="text-2xl font-bold w-1/2"><a href="">RushPush</a></div>
      <div className="flex justify-between gap-10 pr-5">
        <a href="">Trakteer</a>
        <a href="">About Us</a>
      </div>
    </div>
  );
};

export default Navbar;
```

4.20 RenderBoard.tsx

Kode ini merupakan komponen yang akan me-*render* board dari input, output, bahkan menampilkan urutan langkah solusi.	

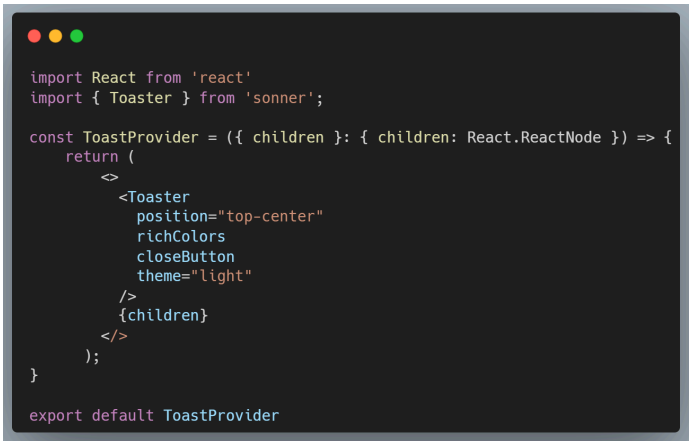
```tsx
import useGameStore from "@/store/GameStore";
import React from "react";

interface RenderBoardProps {
  activeMovePiece: string | null;
}

const RenderBoard: React.FC<RenderBoardProps> = ({ activeMovePiece}) => {
  const { getCurrentBoard } = useGameStore();
  const currentBoard = getCurrentBoard();
  if (!currentBoard) return null;

  const grid = currentBoard.getGrid();
  const exitRow = currentBoard.getExitRow();
  const exitCol = currentBoard.getExitCol();
  const rows = currentBoard.getRows();
  const cols = currentBoard.getCols();

  const getCellColor = (cellValue: string) => {
    if (cellValue === "P") {
      return "bg-red-500";
    }
    // Highlight move piece
    else if (cellValue === activeMovePiece) {
      return "bg-yellow-500";
    }
    // Other pieces
    // TODO: add unique color for each alphabet
    else if (cellValue !== ".") {
      return "bg-blue-500";
    }
    // Empty cell
    return "bg-rush-accent-2/50";
  };

  return (
    <div className="p-5 bg-rush-primary rounded-2xl flex flex-col items-center justify-center">
      {/* Top exit */}
      {exitRow === -1 && (
        <div className="flex">
          {Array.from({ length: cols }).map((_, j) => (
            <div
              key={`top-${j}`}
              className={`w-12 h-12 flex items-center justify-center  m-1 rounded-xl border-rush-accent-2 border ${
                j === exitCol ? "bg-green-500" : "opacity-0"
              }`}>
              {j === exitCol && "K"}
            </div>
          ))}
        </div>
      )}

      {/* Main grid with left and right exits */}
      <div className="flex">
        {/* Left exit */}
        {exitCol === -1 && (
          <div className="flex flex-col">
            {Array.from({ length: rows }).map((_, i) => (
              <div
                key={`left-${i}`}
                className={`w-12 h-12 flex items-center justify-center m-1 rounded-xl border-rush-accent-2 border ${
                  i === exitRow ? "bg-green-500" : "opacity-0"
                }`}>
                {i === exitRow && "K"}
              </div>
            ))}
          </div>
        )}

        {/* Main grid */}
        <div className="flex flex-col">
          {Array.from({ length: rows }).map((_, i) => (
            <div key={`row-${i}`} className="flex">
              {Array.from({ length: cols }).map((_, j) => (
                <div
                  key={`cell-${i}-${j}`}
                  className={`w-12 h-12 flex items-center justify-center rounded-xl m-1 border-rush-accent-2 border
${getCellColor(grid[i][j])}`}>
                  {grid[i][j] !== "." ? grid[i][j] : ""}
                </div>
              ))}
            </div>
          ))}
        </div>

        {/* Right exit */}
        {exitCol === cols && (
          <div className="flex flex-col">
            {Array.from({ length: rows }).map((_, i) => (
              <div
                key={`right-${i}`}
                className={`w-12 h-12 flex items-center justify-center m-1 rounded-xl border-rush-accent-2 border ${
                  i === exitRow ? "bg-green-500" : "opacity-0"
                }`}>
                {i === exitRow && "K"}
              </div>
            ))}
          </div>
        )}
      </div>

      {/* Bottom exit */}
      {exitRow === rows && (
        <div className="flex">
          {Array.from({ length: cols }).map((_, j) => (
            <div
              key={`bottom-${j}`}
              className={`w-12 h-12 flex items-center justify-center  m-1 rounded-xl border-rush-accent-2 border ${
                j === exitCol ? "bg-green-500" : "opacity-0"
              }`}>
              {j === exitCol && "K"}
            </div>
          ))}
        </div>
      )}
    </div>
  );
};

export default RenderBoard;
```

4.21 Solution.tsx

Kode ini merupakan komponen *controller* yang menampilkan navigasi untuk memutar animasi urutan langkah solusi beserta pengaturan kecepatannya, menampilkan data statistik hasil pencarian (waktu, jumlah langkah, dan node yang dikunjungi), serta menampilkan daftar langkah beserta informasi *ID piece*, arah, dan jumlah langkahnya.	

```
"use client";

import React, { useEffect, useState } from "react";
import useGameStore from "@/store/GameStore";
import RenderBoard from "./RenderBoard";

const SolutionDisplay = () => {
  const {
    board,
    solution,
    solutionPath,
    currentMoveIndex,
    isAnimating,
    animationSpeed,
    nextMove,
    prevMove,
    jumpToMove,
    startAnimation,
    stopAnimation,
    setAnimationSpeed,
  } = useGameStore();

  const [activeMovePiece, setActiveMovePiece] = useState<string | null>(null);

  // Reset activeMovePiece when currentMoveIndex changes
  useEffect(() => {
    if (currentMoveIndex >= 0 && currentMoveIndex < solutionPath.length) {
      setActiveMovePiece(solutionPath[currentMoveIndex].pieceId);
    } else {
      setActiveMovePiece(null);
    }
  }, [currentMoveIndex, solutionPath]);

  if (!board) {
    return (
      <div className="p-6 rounded-2xl shadow-md bg-rush-primary text-rush-secondary">
        <p>Please load a board configuration first.</p>
      </div>
    );
  }

  return (
    <div className="p-6 rounded-2xl border-rush-primary border-3 bg-rush-secondary text-rush-primary w-full">
      <div className="flex justify-center items-center gap-10 w-full">
        <RenderBoard activeMovePiece={activeMovePiece} />

        {solution && (
          <div className="w-1/2">
            <h3 className="text-2xl font-semibold">Solution</h3>
            <div className="mb-4">
              <p>Success: {solution.success ? "yes" : "no"}</p>
              <p>Nodes Visited: {solution.nodesVisited}</p>
              <p>Execution Time: {solution.executionTime.toFixed(2)} ms</p>
              <p>Solution Length: {solutionPath.length} moves</p>
            </div>

            <div className="mb-4">
              <p className="mb-2">
                Current Move: {currentMoveIndex + 1} / {solutionPath.length}
              </p>
              <div className="flex gap-2">
                <button
                  onClick={prevMove}
                  disabled={currentMoveIndex < 0}
                  className="bg-rush-accent-1 text-rush-primary px-3 py-1 rounded-lg disabled:opacity-50">
                  Prev
                </button>
                <button
                  onClick={nextMove}
                  disabled={currentMoveIndex >= solutionPath.length - 1}
                  className="bg-rush-accent-1 text-rush-primary px-3 py-1 rounded-lg disabled:opacity-50">
                  Next
                </button>
                {!isAnimating ? (
                  <button
                    onClick={startAnimation}
                    disabled={currentMoveIndex >= solutionPath.length - 1}
                    className="bg-rush-accent-1 text-rush-accent-2 px-3 py-1 rounded-lg disabled:opacity-50">
                    Play
                  </button>
                ) : (
                  <button onClick={stopAnimation} className="bg-rush-accent-2 text-rush-secondary px-3 py-1 rounded-
lg">
                    Pause
                  </button>
                )}
                <button
                  onClick={() => jumpToMove(-1)}
                  disabled={currentMoveIndex < 0}
                  className="bg-rush-accent-1 text-rush-primary px-3 py-1 rounded-lg disabled:opacity-50">
                  Reset
                </button>
              </div>
            </div>

            <div className="mb-4">
              <label className="block text-sm font-medium mb-2">Animation Speed: {animationSpeed}ms</label>
              <input
                type="range"
                min="10"
                max="2000"
                step="10"
                value={animationSpeed}
                onChange={(e) => setAnimationSpeed(Number(e.target.value))}
                className="w-full"
              />
            </div>

            <h4 className="text-lg font-medium mb-2">Move Sequence:</h4>
            <div className="h-64 overflow-y-auto border border-rush-accent-2 p-4 rounded-lg">
              <ul>
                {solutionPath.map((move, index) => (
                  <li
                    key={index}
                    className={`mb-1 p-2 rounded cursor-pointer ${
                      index === currentMoveIndex ? "bg-rush-primary text-rush-accent-2" : ""
                    }`}
                    onClick={() => jumpToMove(index)}>
                    {index + 1}. {move.pieceId}-{move.direction} ({move.steps} step{move.steps > 1 ? "s" : ""})
                  </li>
                ))}
              </ul>
            </div>
          </div>
        )}
      </div>
    </div>
  );
};

export default SolutionDisplay;
```

4.22 ToastProvider.tsx

Kode ini merupakan komponen *wrapper* pada layout sehingga tiap komponen aplikasi memiliki pengaturan *toaster* yang sama.	

```tsx
import React from 'react'
import { Toaster } from 'sonner';

const ToastProvider = ({ children }: { children: React.ReactNode }) => {
    return (
        <>
          <Toaster
            position="top-center"
            richColors
            closeButton
            theme="light"
          />
          {children}
        </>
      );
}

export default ToastProvider
```

BAB 5

Percobaan

5.1 Test Case 1

Algoritma: Uniform Cost Search

Input 00_test.txt: 6 6 11

AAB..F

..BCDF

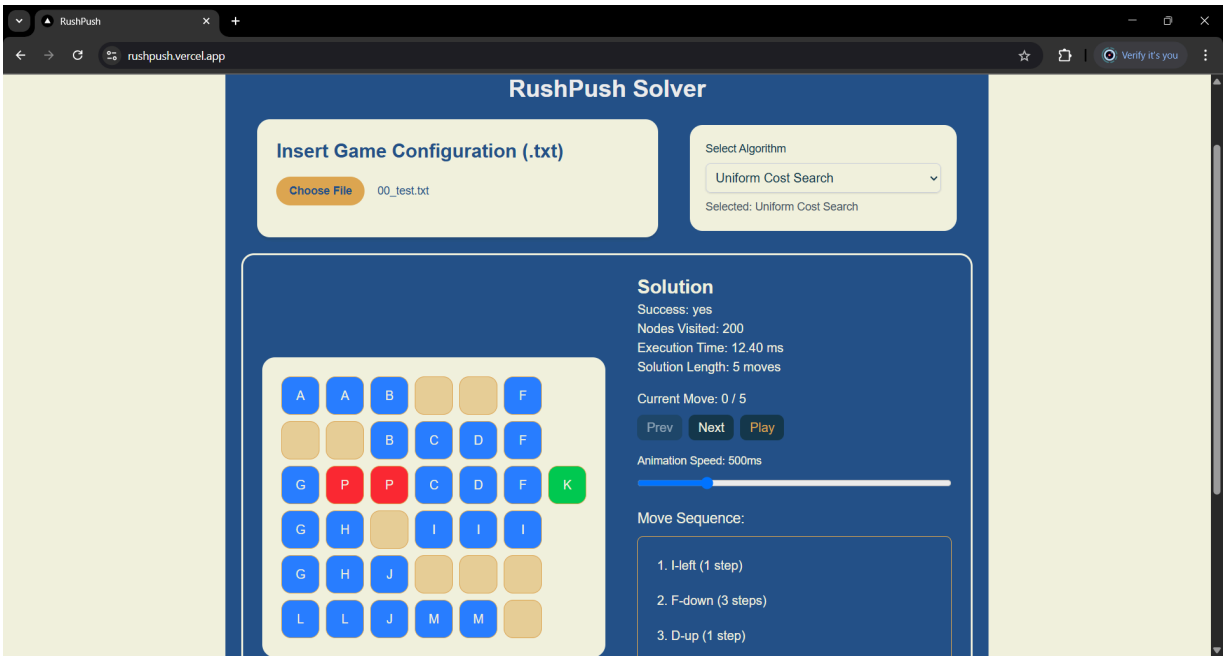
GPPCDFK

GH.III

GHJ...

LLJMM.

Initial Condition:



Final Condition:

5.2 Test Case 2

Algoritma: Greedy Best-First Search

Heuristik: Manhattan Distance

Input 00_test.txt:

6 6

11

AAB..F

..BCDF

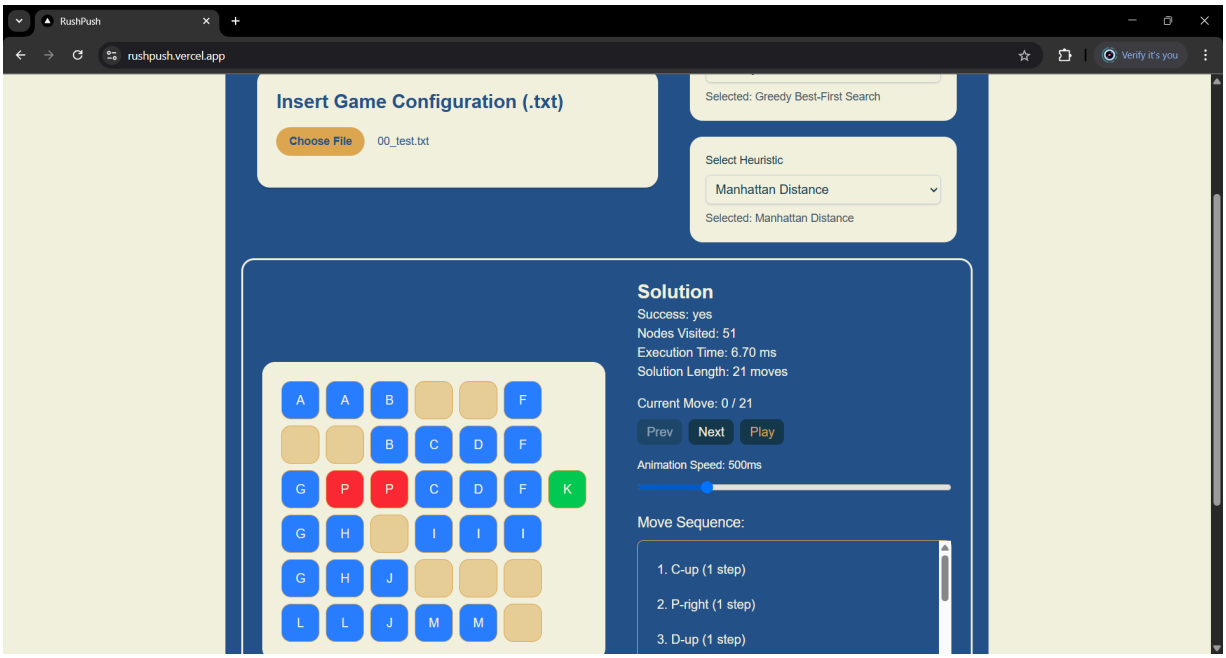
GPPCDFK

GH.III

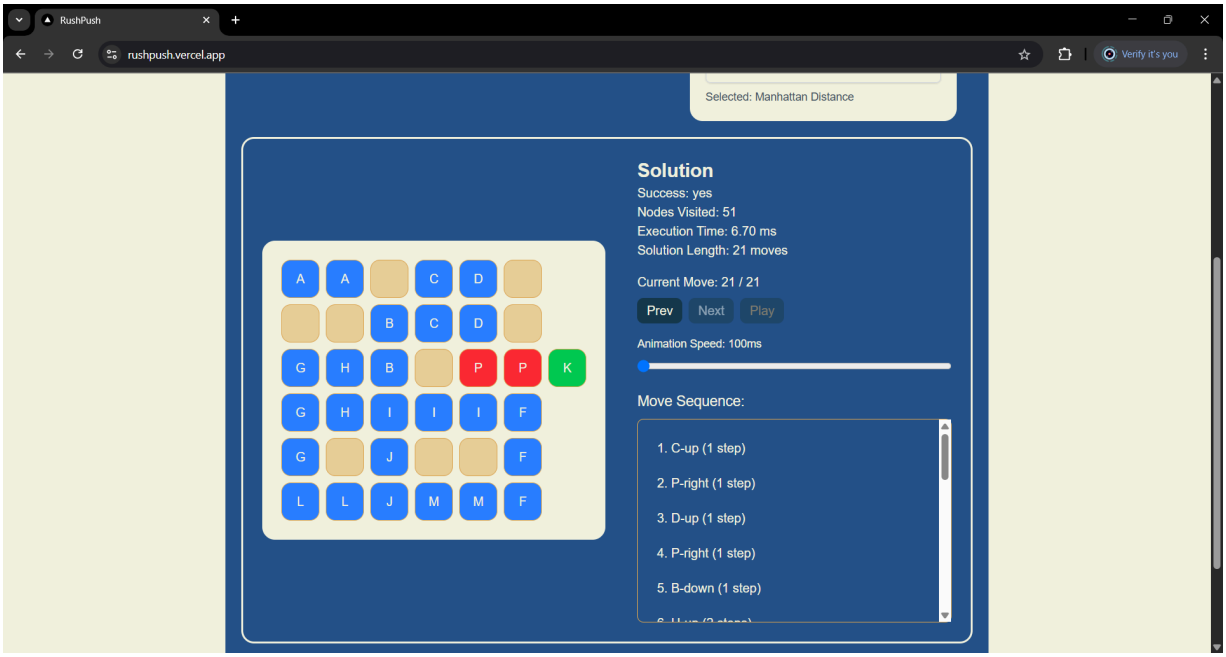
GHJ...

LLJMM.

Initial Condition:



Final Condition:



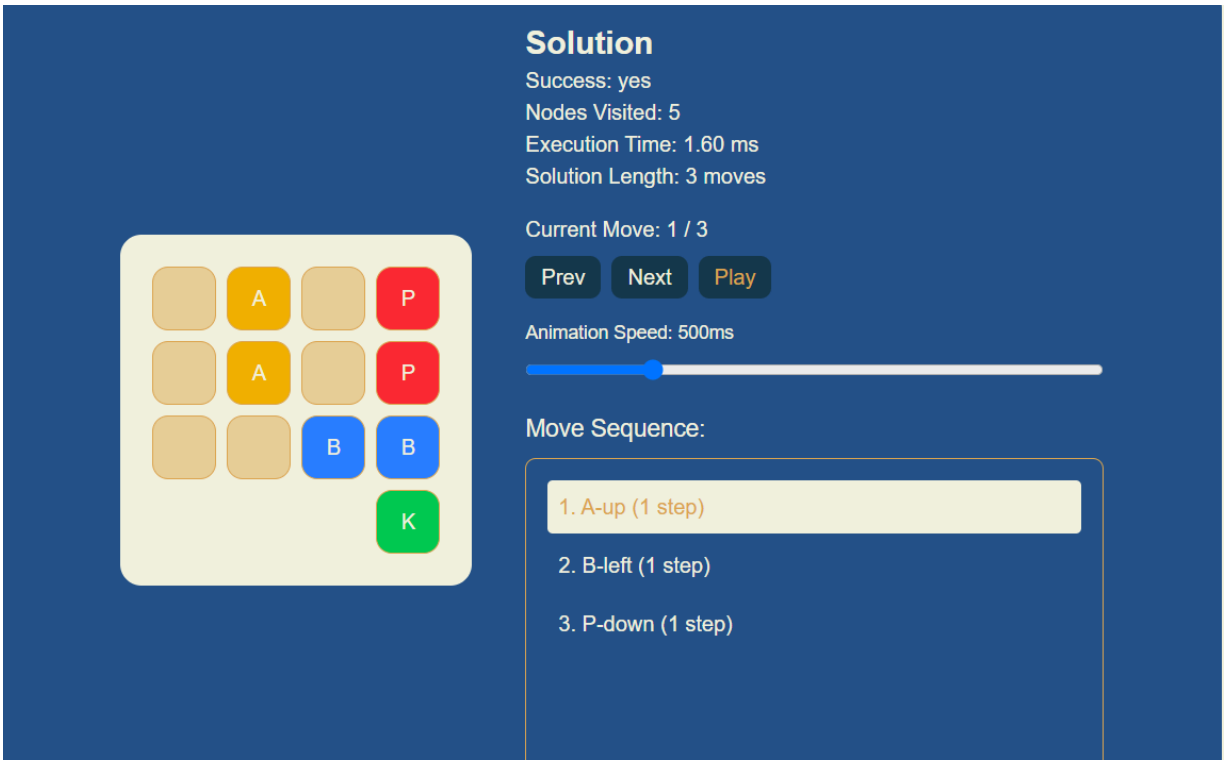
5.3 Test Case 3

Algoritma: Uniform Cost Search

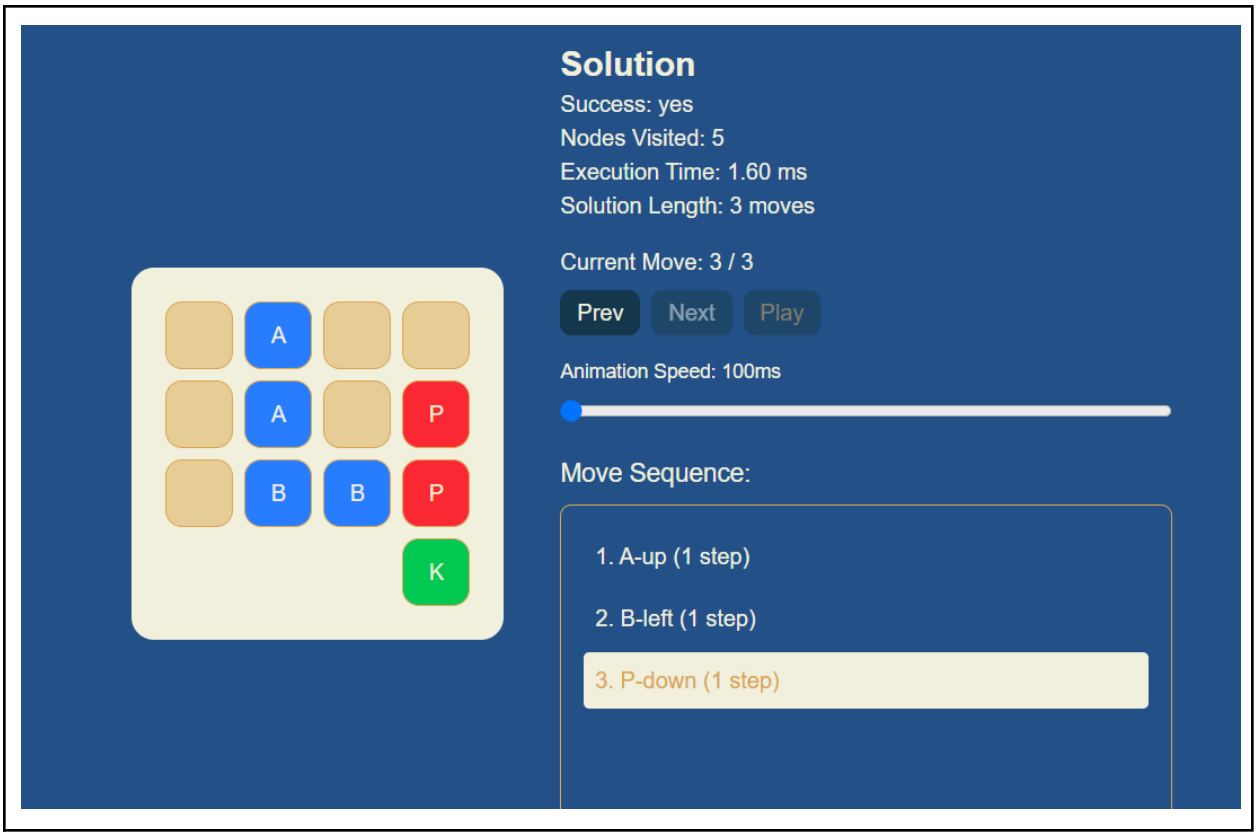
Input 01_test.txt

```
3 4
2
...P
.A.P
.ABB
  K
```

Initial condition:



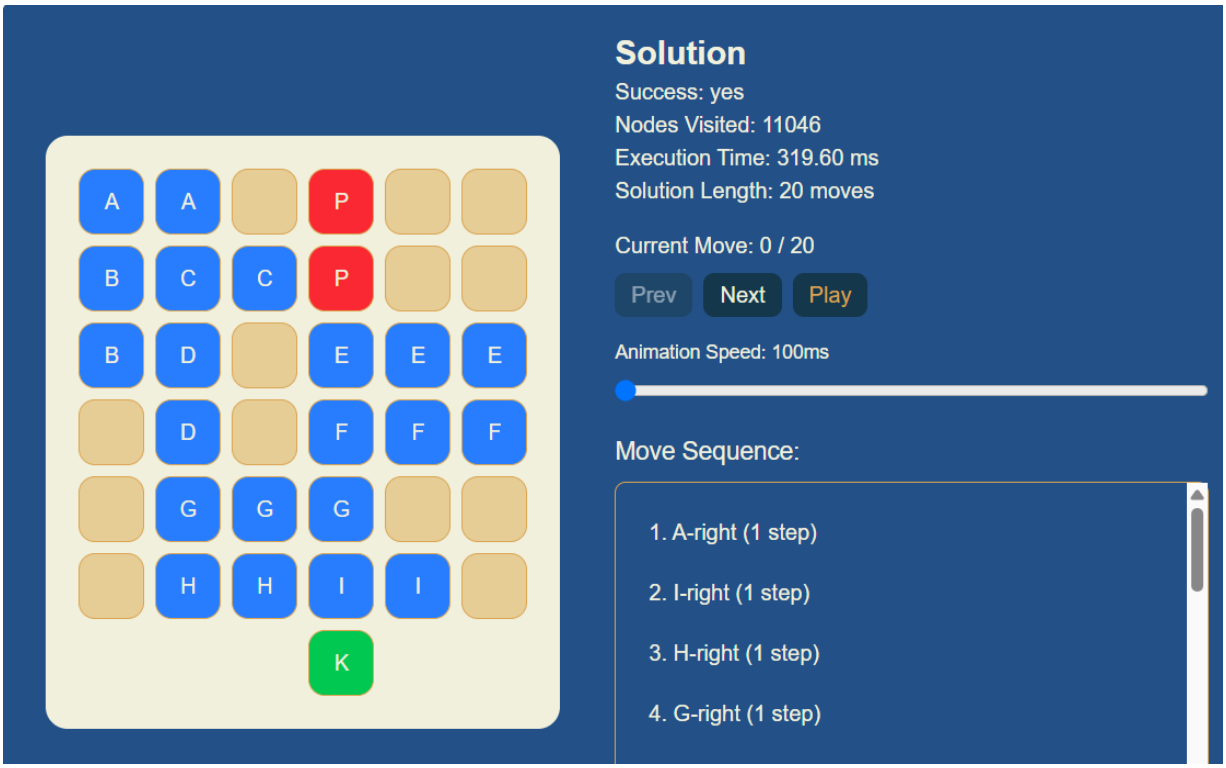
Final Condition:



5.4 Test Case 4

Algoritma: Uniform Cost Search

Input 02_test.txt: 6 6 9 AA.P.. BCCP.. BD.EEE .D.FFF .GGG.. .HHII. K
Initial condition:

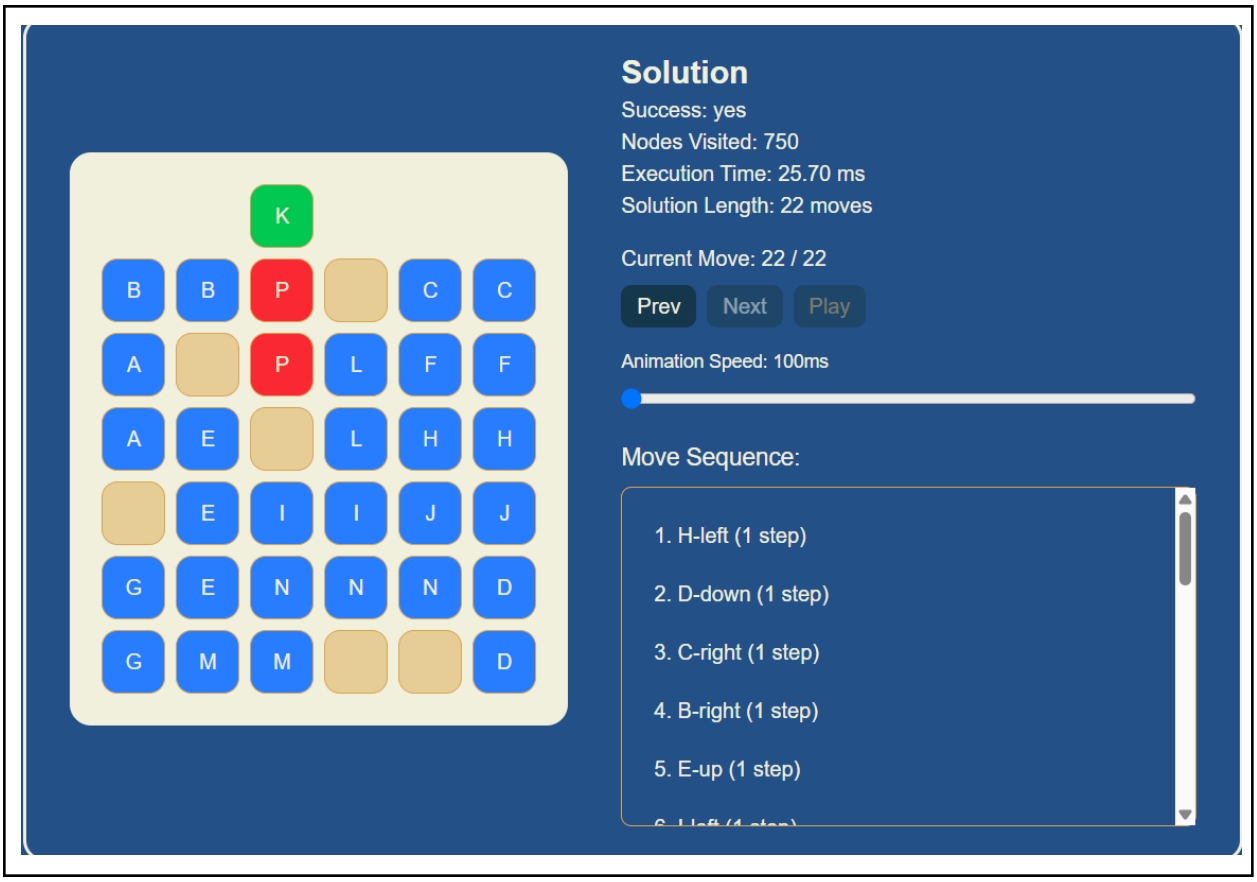


Final condition

5.5 Test Case 5

Algoritma: Uniform Cost Search

Input 03_test.txt 6 6 13   K ABBCCD AEPFFD GEP.HH GEIIJJ NNNL.. ...LMM
Initial condition: 
Final condition:

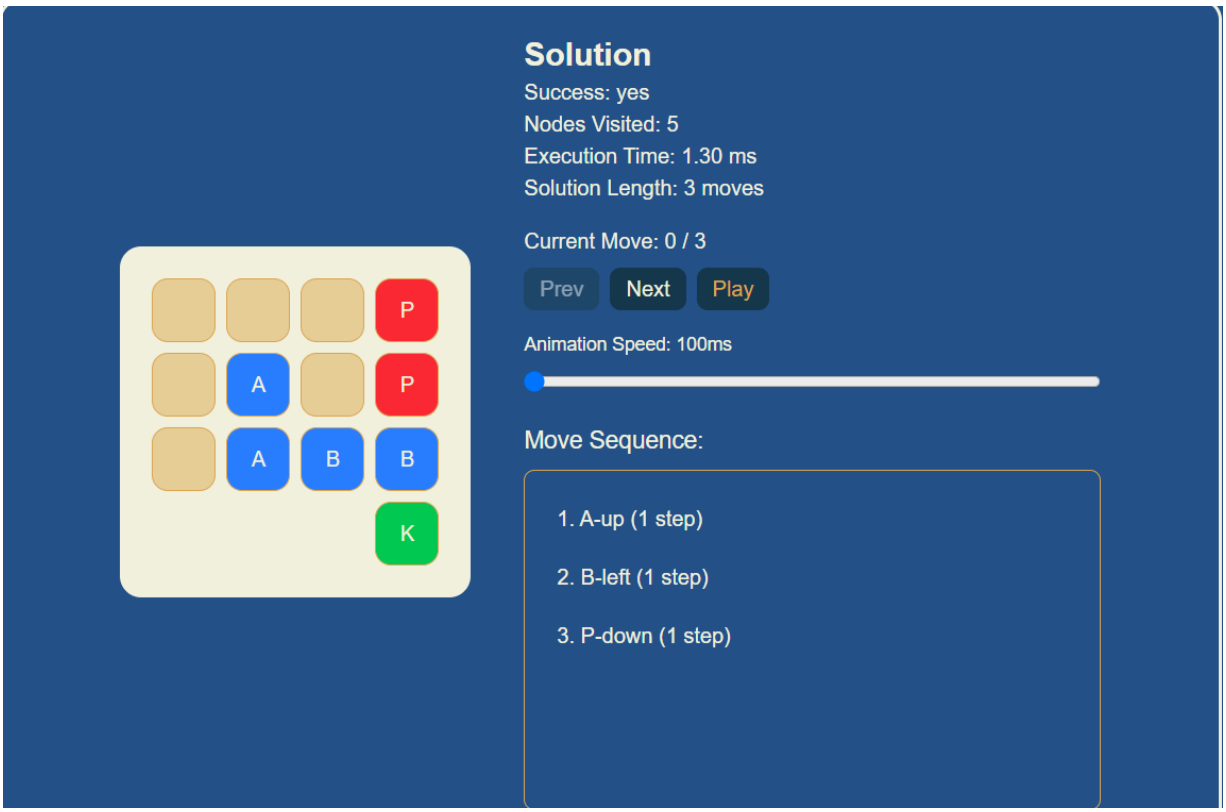


5.6 Test Case 6

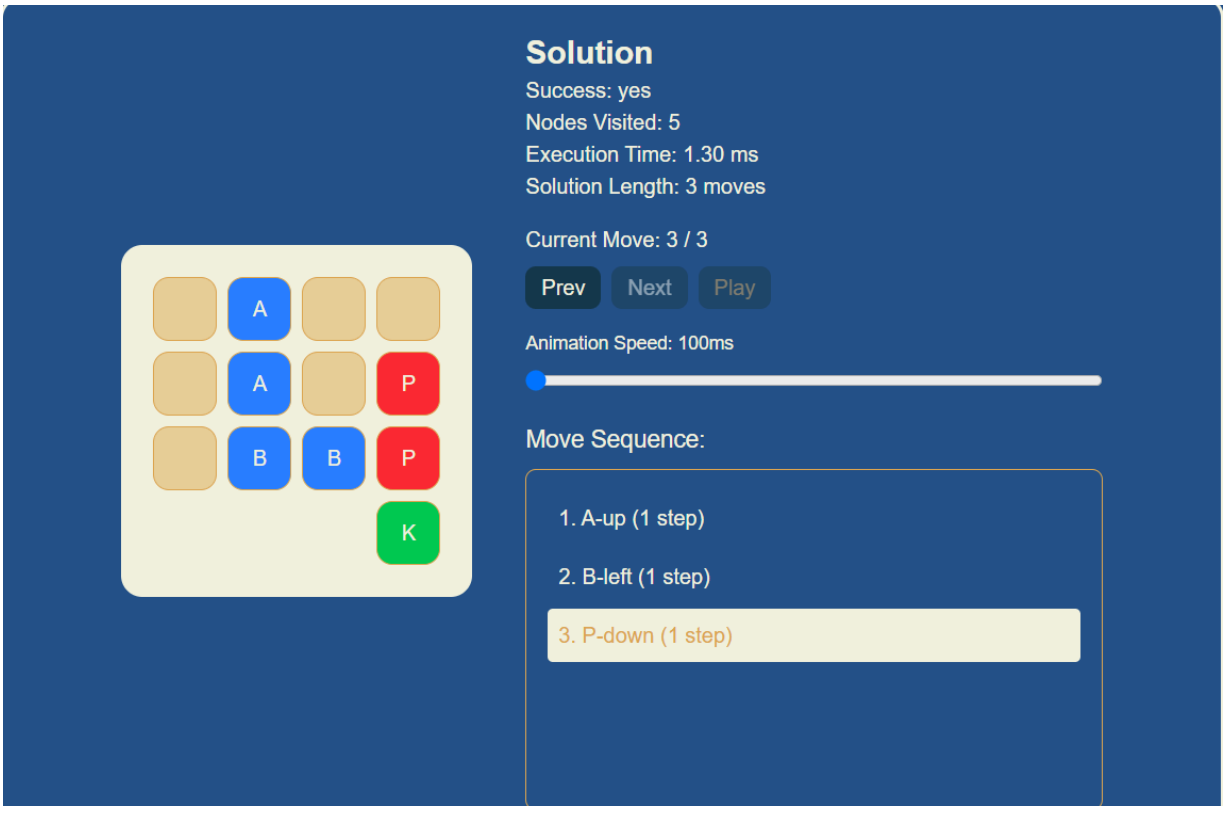
Algoritma: Greedy Best-First Search

Heuristik: Blocking Pieces

Input 01_test.txt: 3 4 2 ...P .A.P .ABB   K
Initial condition:



Final condition:



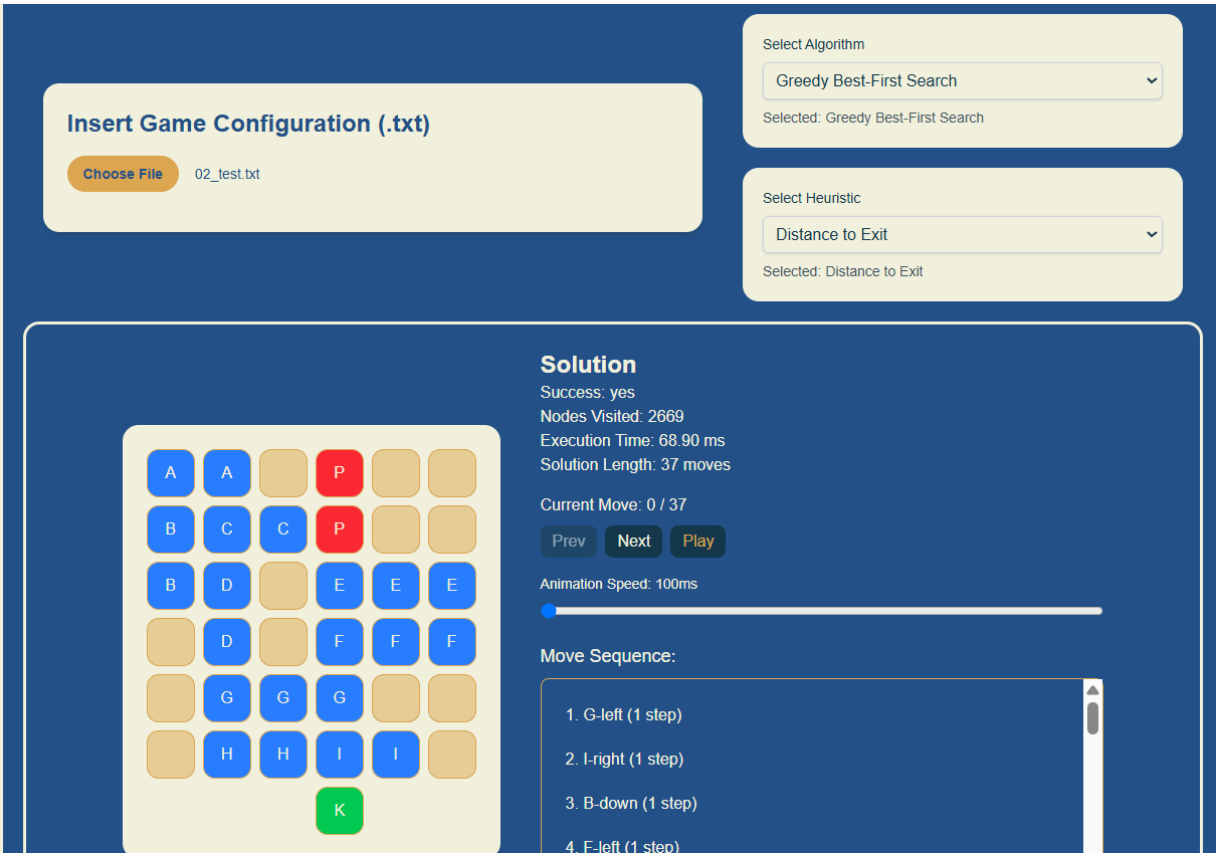
5.7 Test Case 7

Algoritma: Greedy Best-First Search

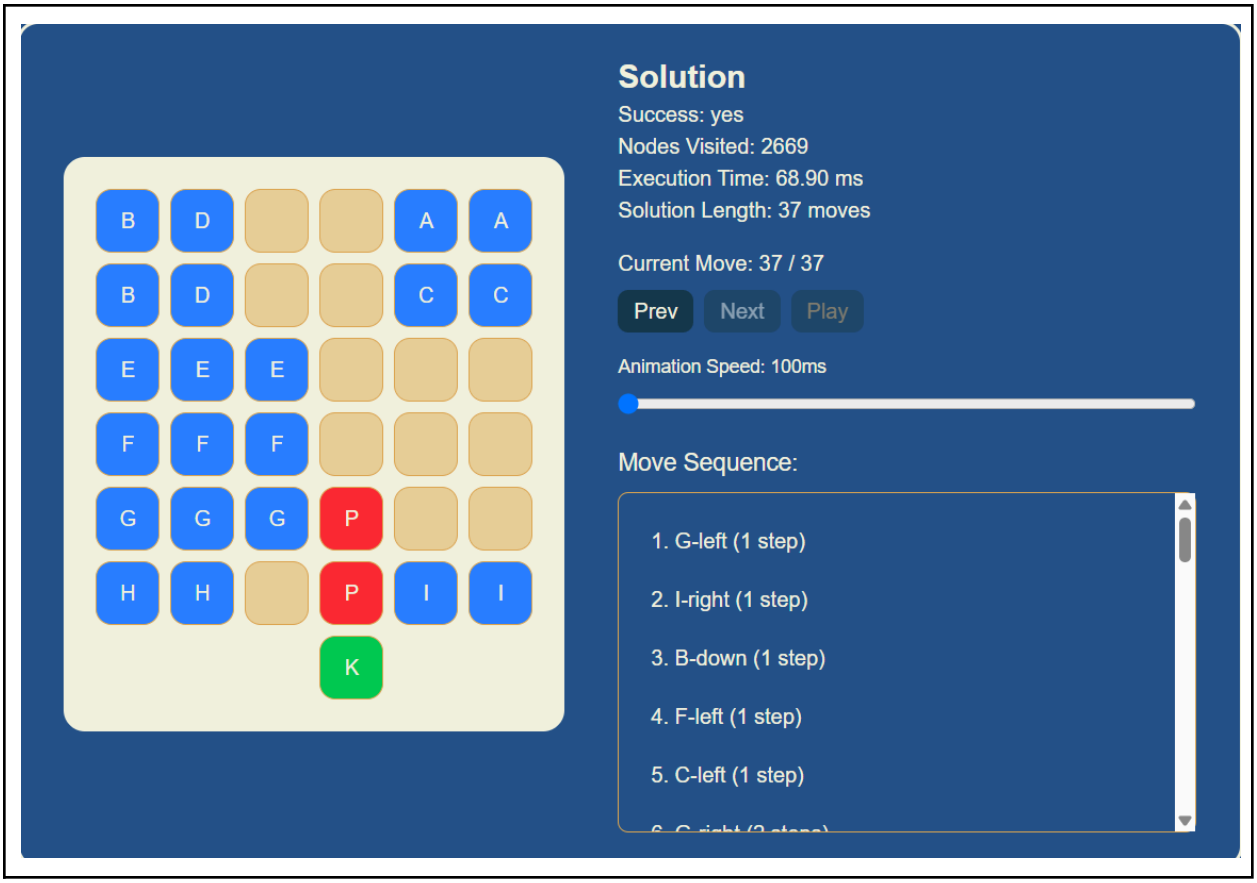
Heuristik: Distance to Exit

Input 02_test.txt
6 6
9
AA.P..
BCCP..
BD.EEE
.D.FFF
.GGG..
.HHII.
  K

Initial condition



Final condition:

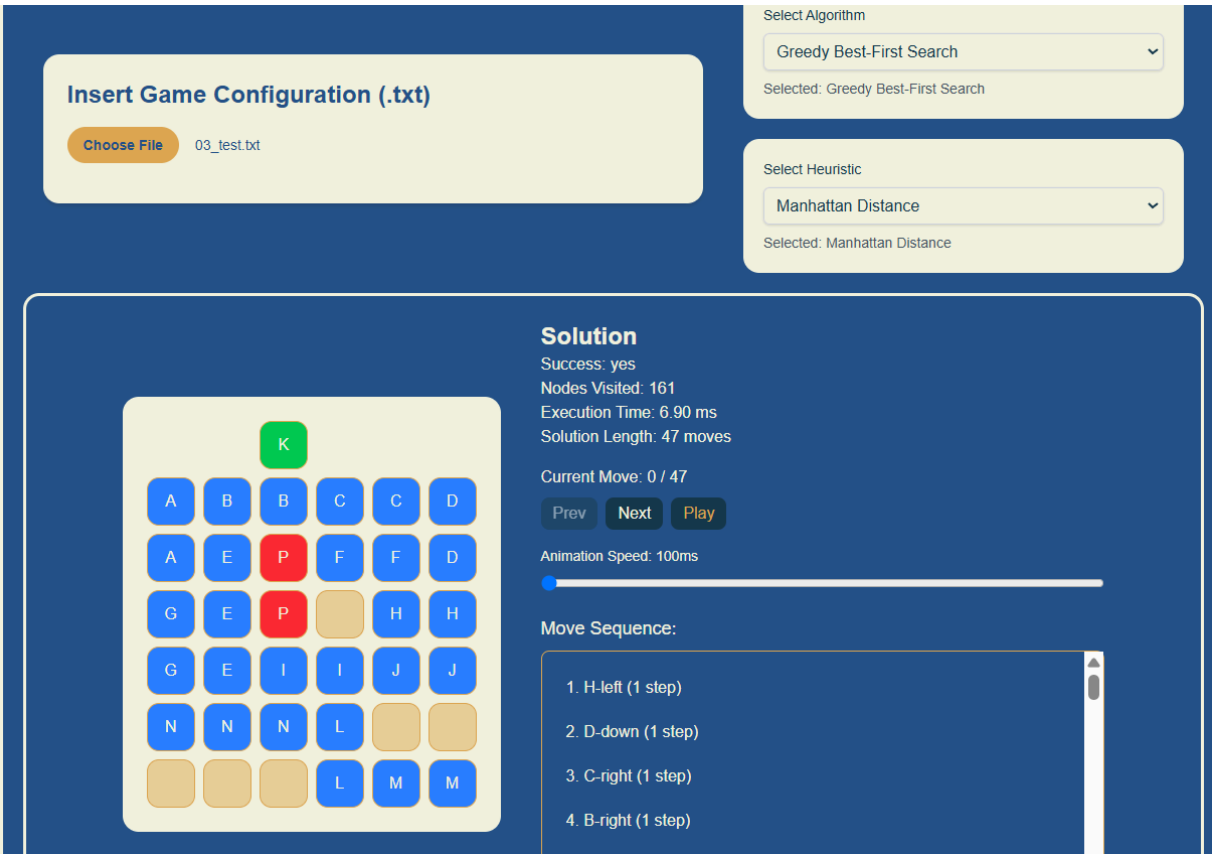


5.8 Test Case 8

Algoritma: Greedy Best-First Search

Heuristik: Manhattan Distance

Input 03_test.txt: 6 6 13   K ABBCCD AEPFFD GEP.HH GEIIJJ NNNL.. ...LMM
Initial condition:



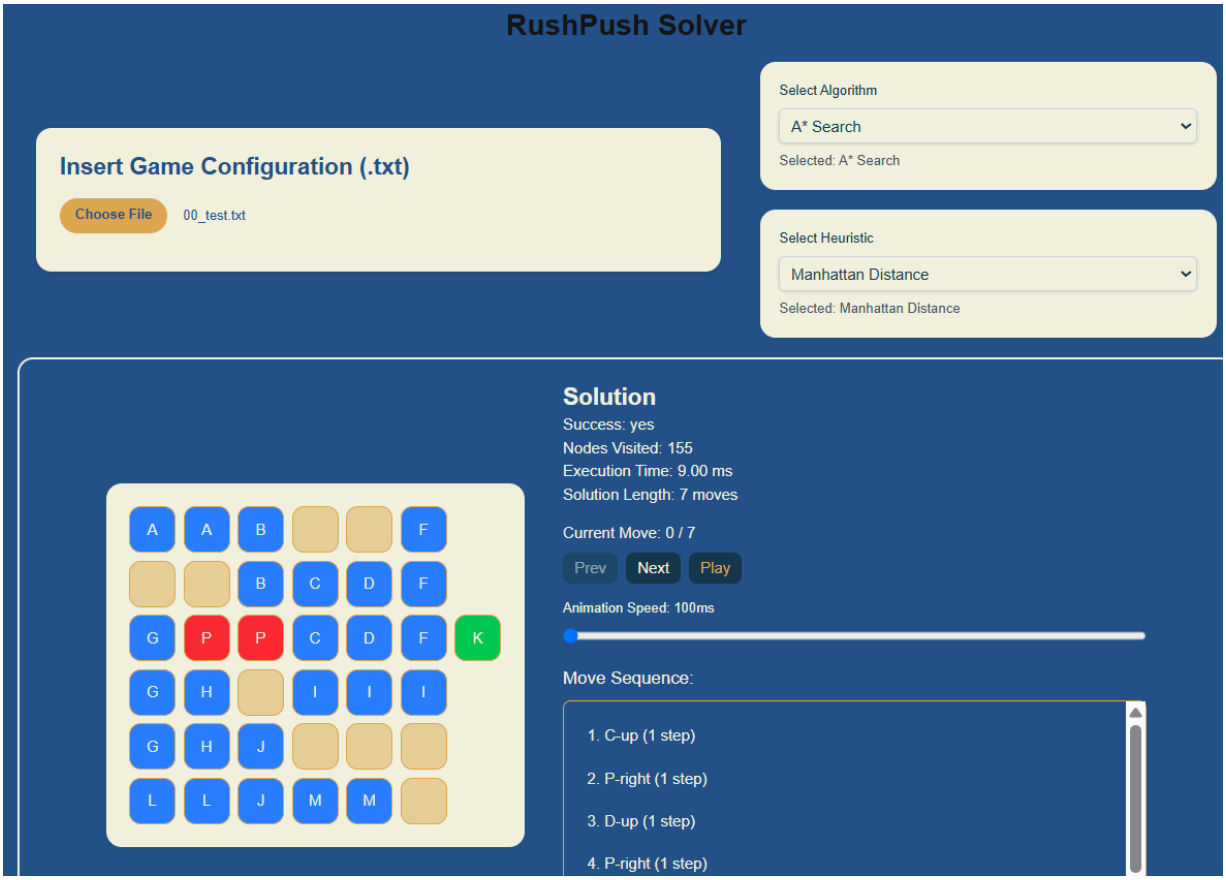
Final condition:

5.9 Test Case 9

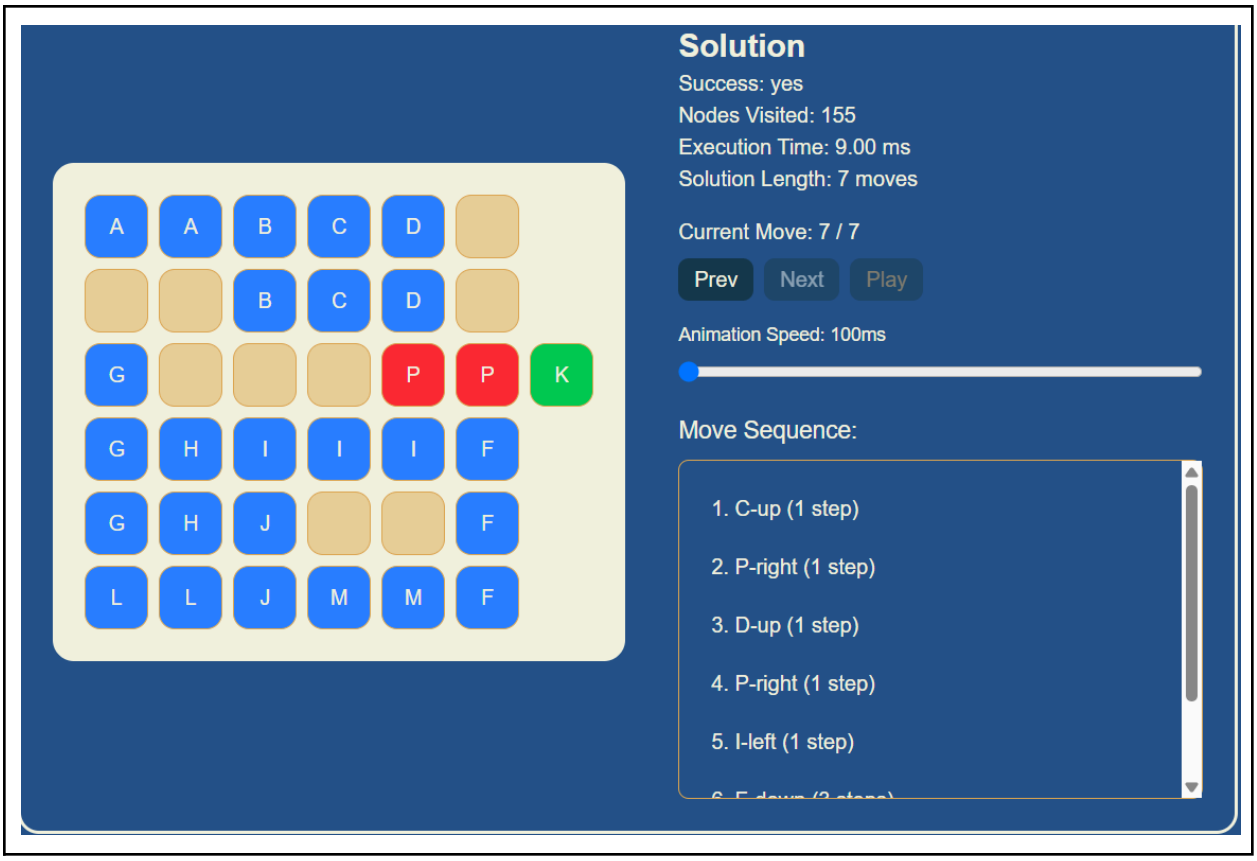
Algoritma: A* Search

Heuristik: Manhattan Distance

Input 00_test.txt: 6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.
Initial condition:



Final condition:

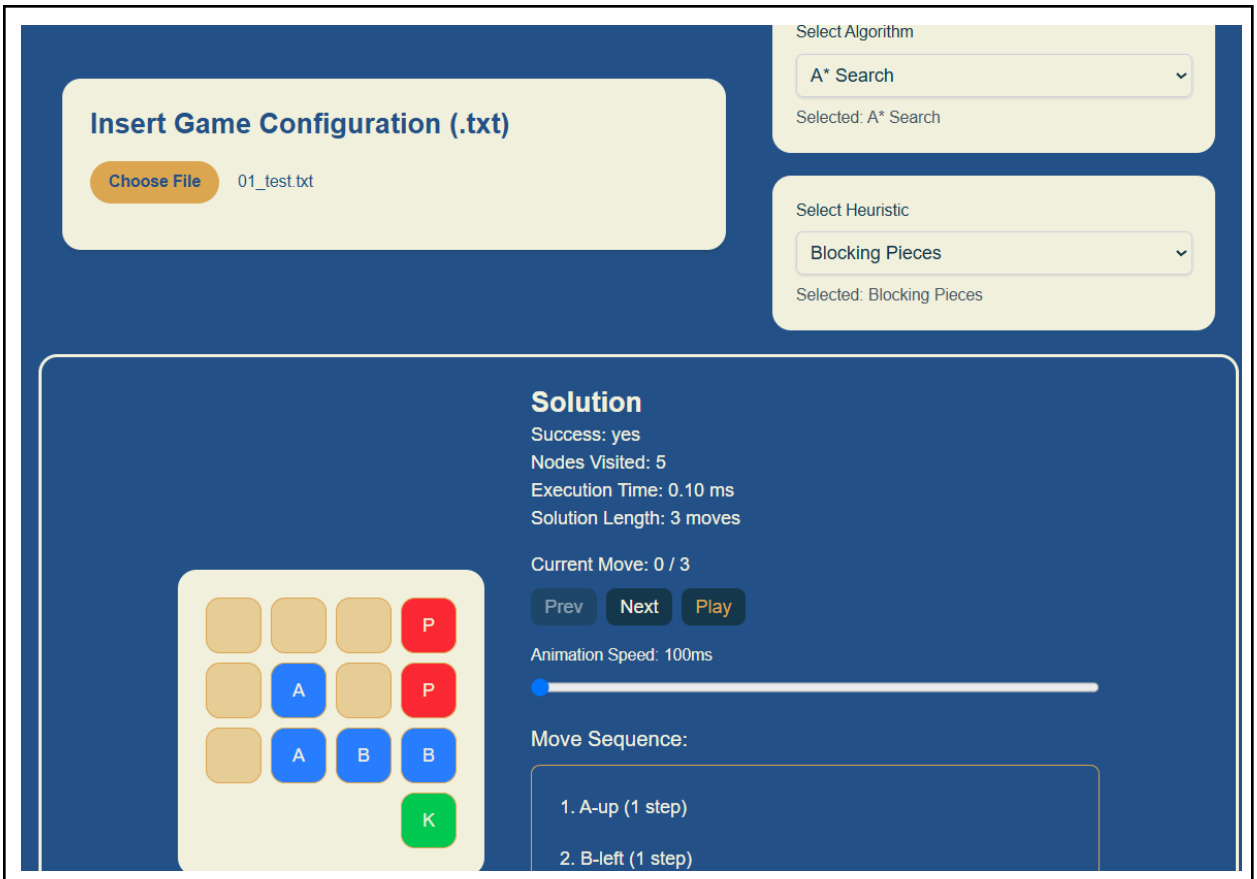


5.10 Test Case 10

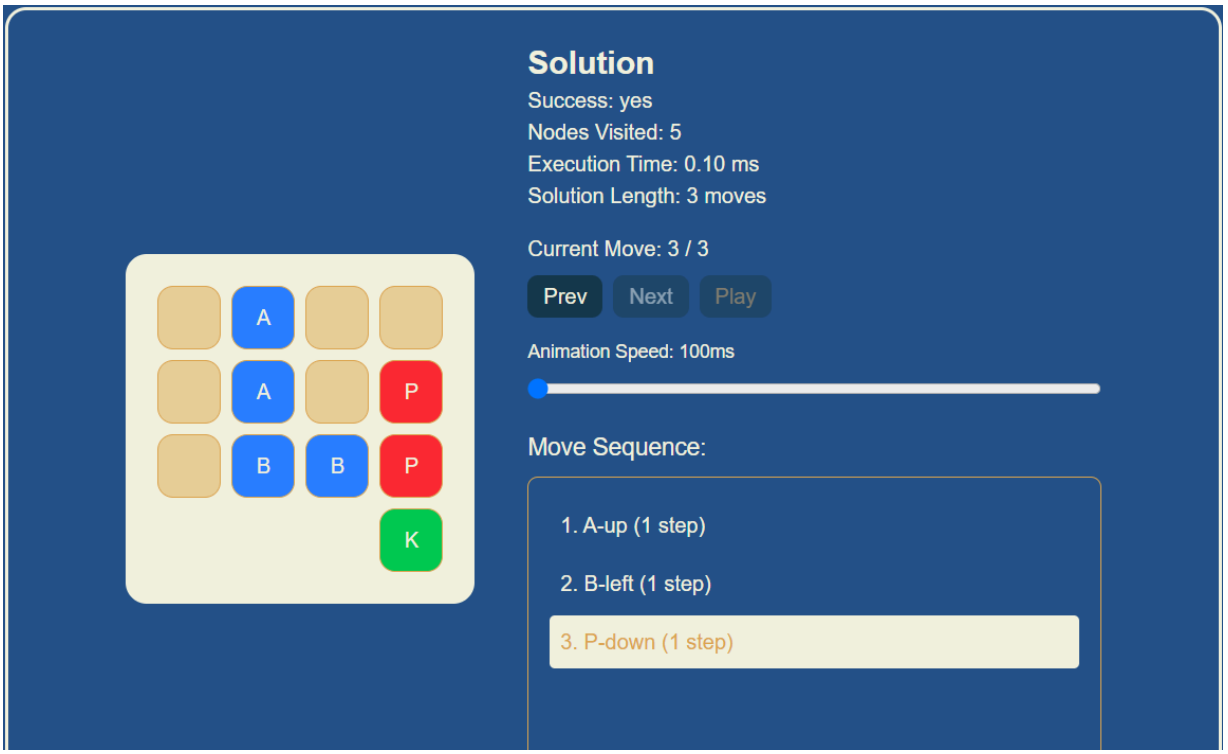
Algoritma: A* Search

Heuristik: Blocking Pieces

Input 01_test.txt: 3 4 2 ...P .A.P .ABB   K
Initial condition:



Final condition:



5.11 Test Case 11

Algoritma: A* Search

Heuristik: Manhattan Distance

Input 02_test.txt:
6 6
9
AA.P..
BCCP..
BD.EEE
.D.FFF
.GGG..
.HHII.
  K

Initial condition:

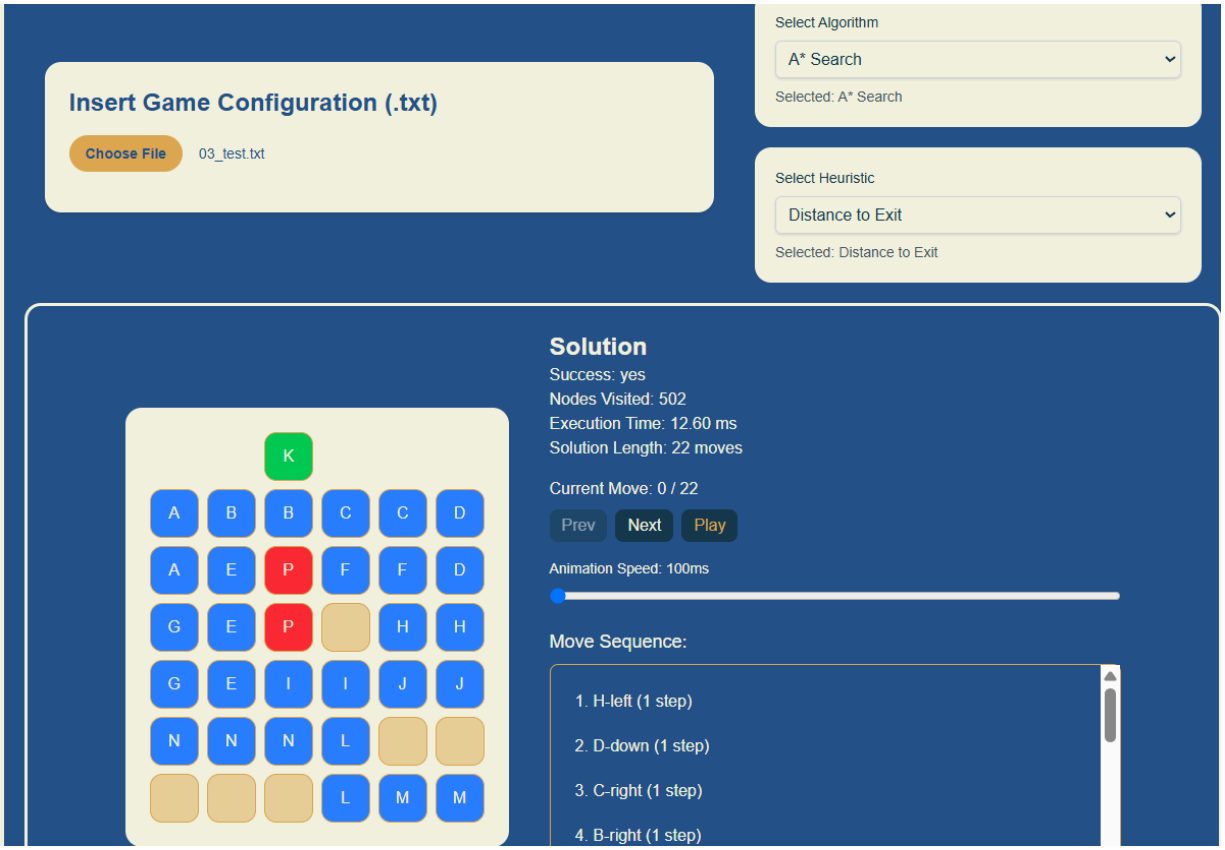
Final condition:

5.12 Test Case 12

Algoritma: A* Search

Heuristik: Distance to Exit

Input 03_test.txt: 6 6 13   K ABBCCD AEPFFD GEP.HH GEIIJJ NNNL.. ...LMM
Initial condition:



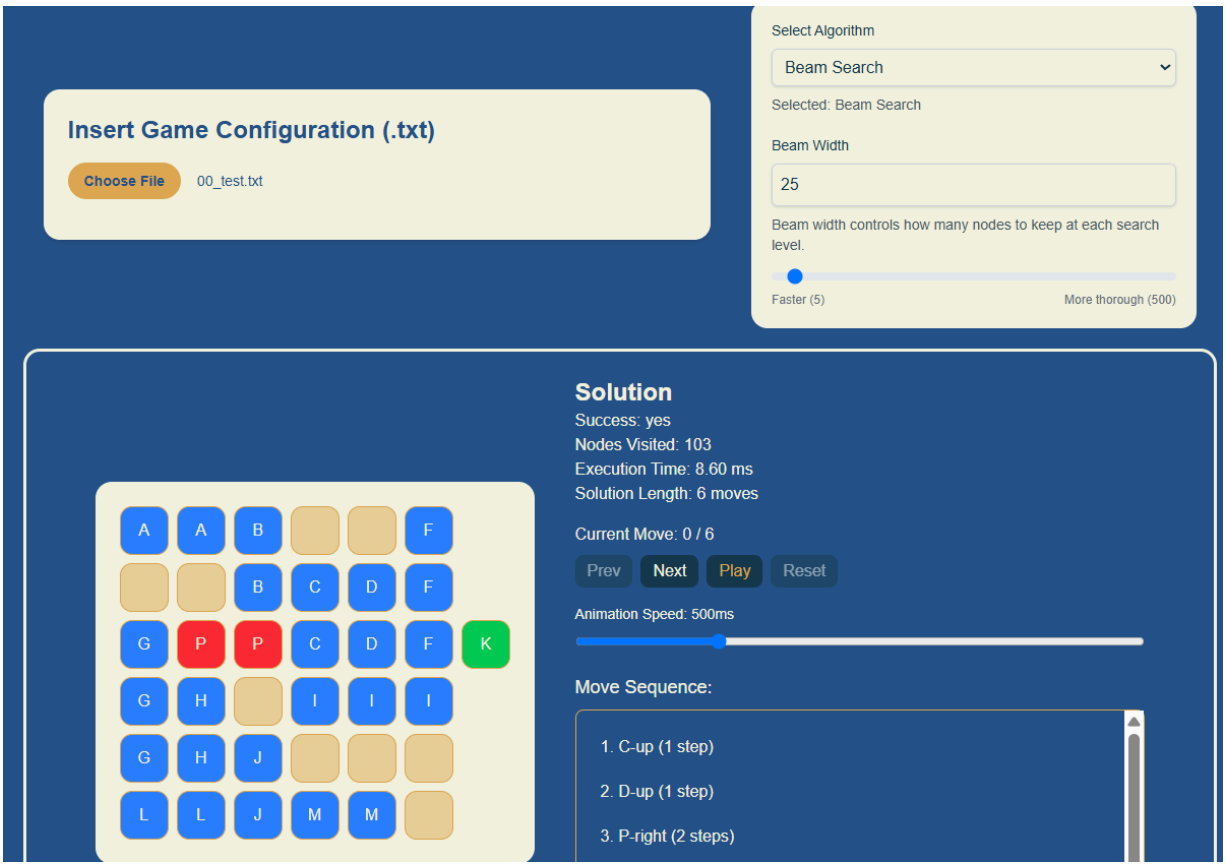
Final condition:

5.13 Test Case 13

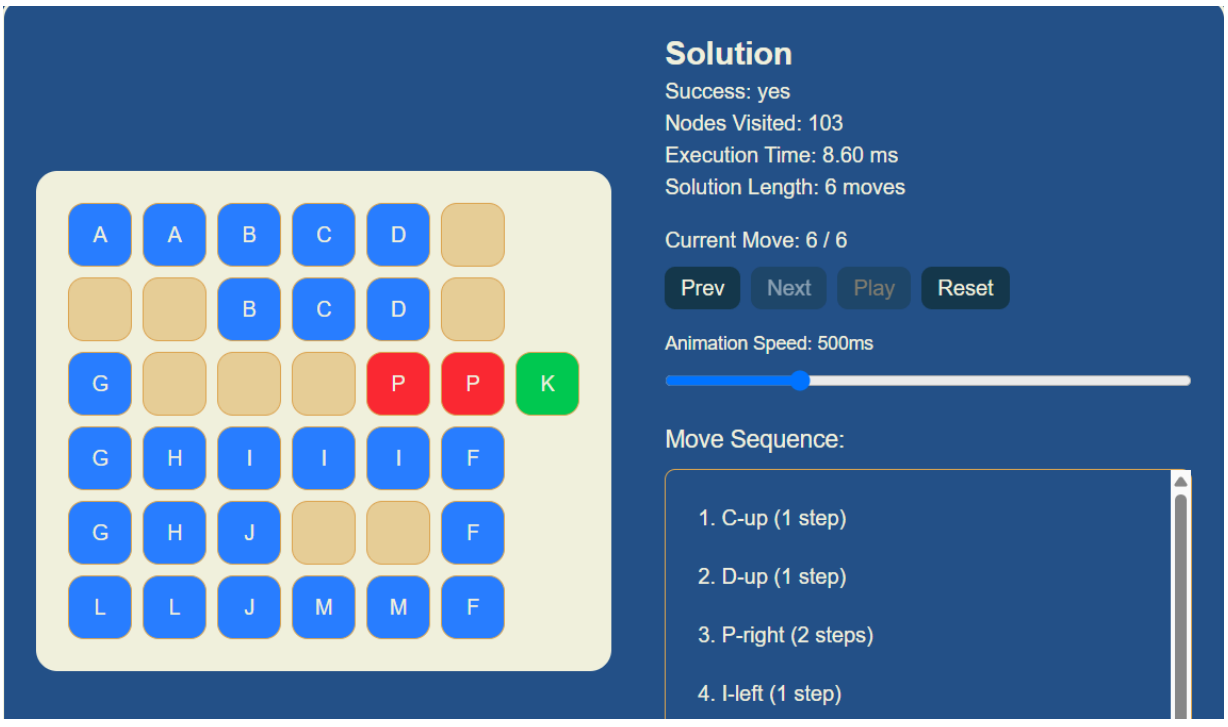
Algoritma: Beam Search

Width: 25

Input 00_test.txt: 6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.
Initial condition:



Final condition:



5.14 Test Case 14

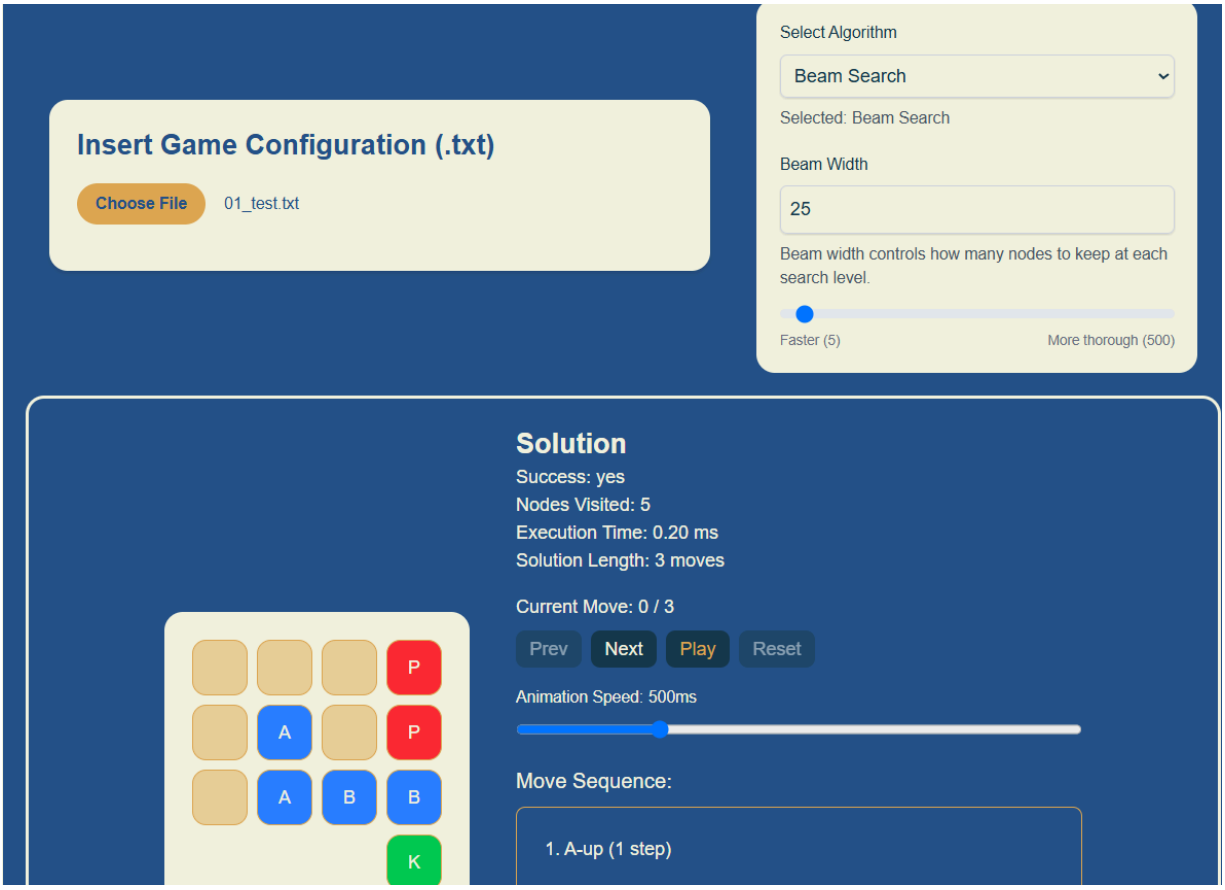
Algoritma: Beam Search

Width: 25

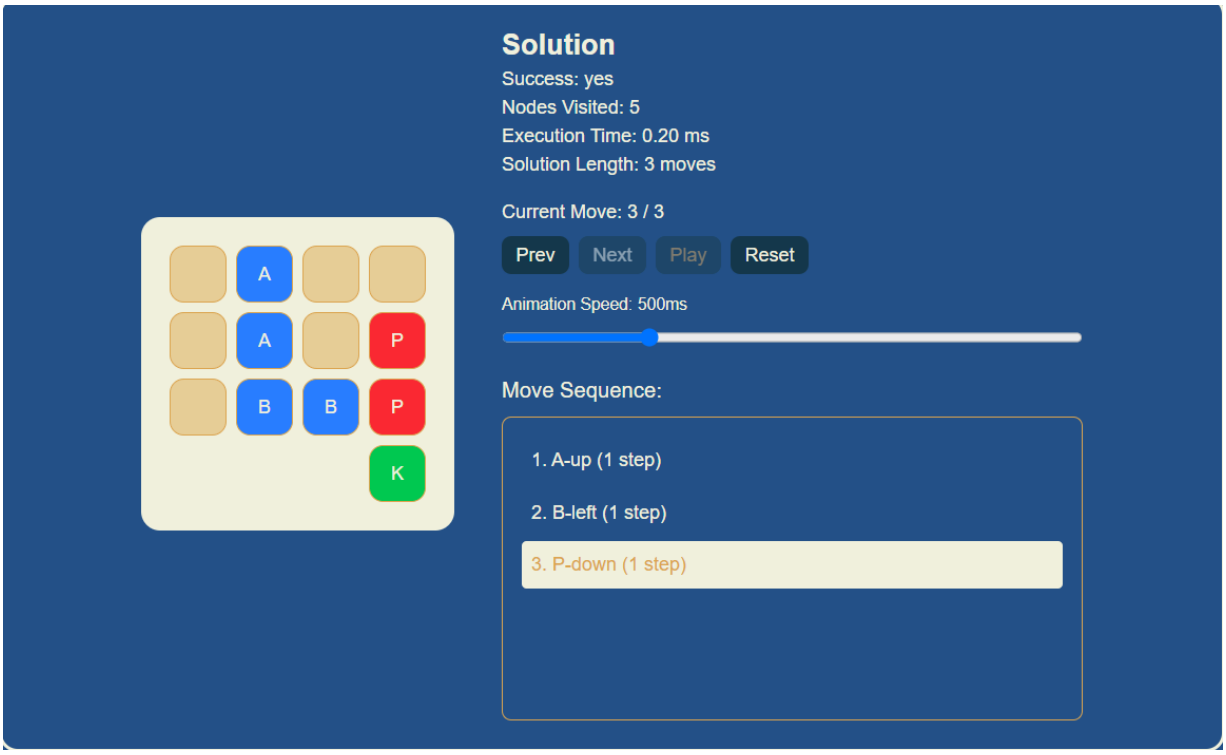
Input 01_test.txt:

```
3 4
2
...P
.A.P
.ABB
  K
```

Initial condition:



Final condition:

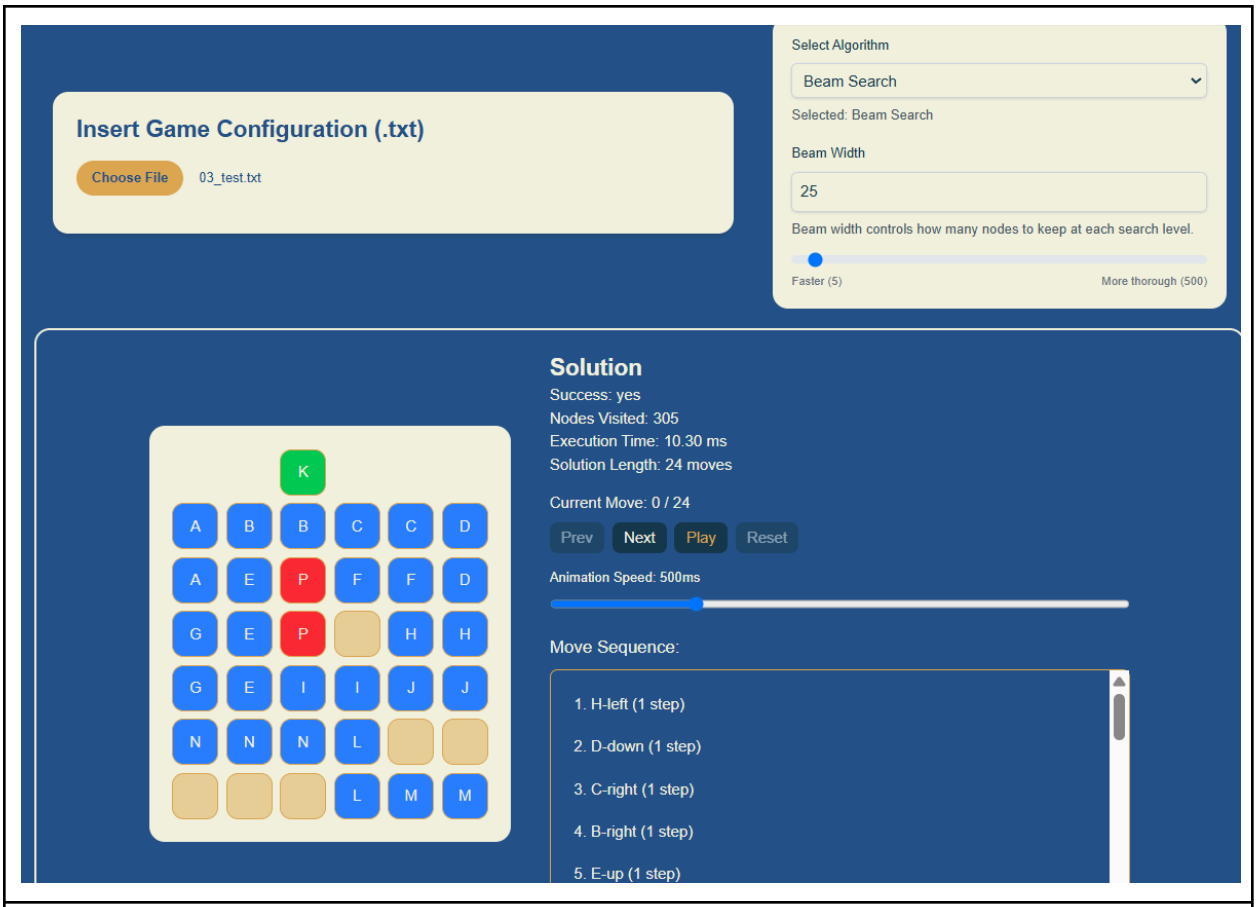


5.15 Test Case 15

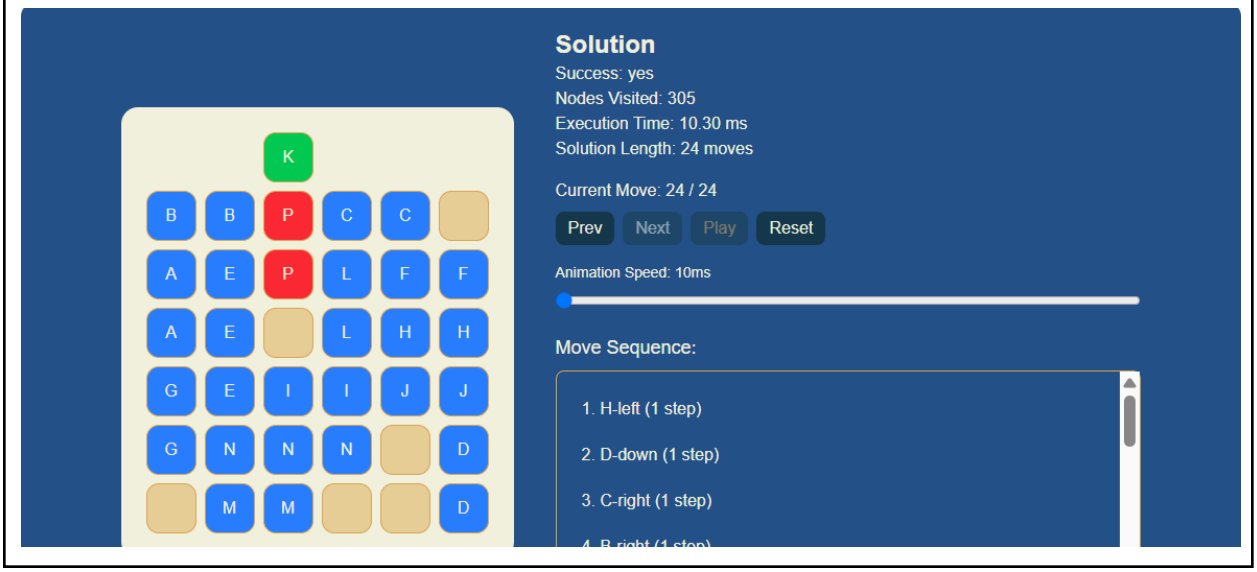
Algoritma: Beam Search

Width: 25

Input 03_test.txt: 6 6 13   K ABBCCD AEPFFD GEP.HH GEIIJJ NNNL.. ...LMM
Initial condition:



Final condition:



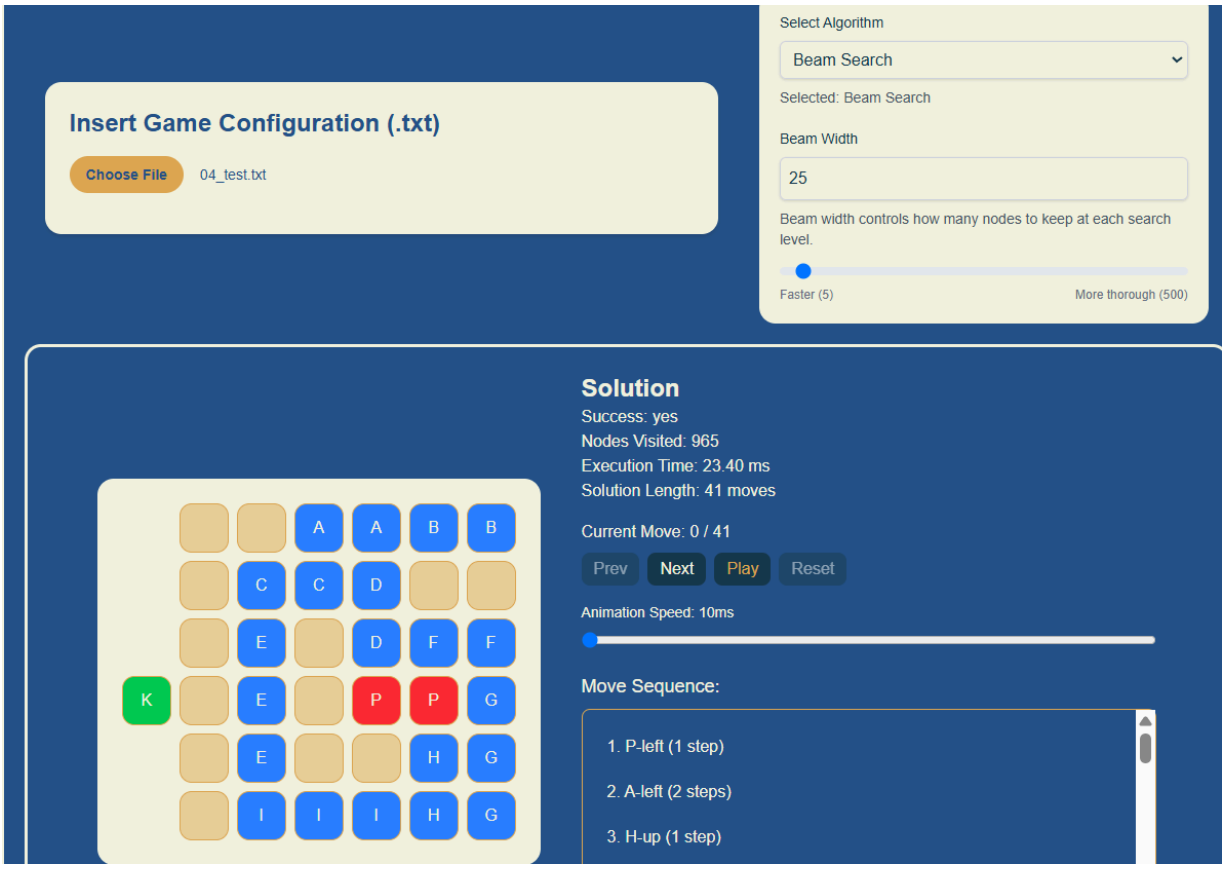
5.16 Test Case 16

Algoritma: Beam Search

Width: 25

Input 04_test.txt:
6 6
9
..AABB
.CCD..
.E.DFF
K.E.PPG
.E..HG
.IIIHG

Initial condition:



Final condition:

BAB 6

Analisis Solusi *UCS, GBFS, Beam Search,* dan *A**

6.1 Analisis UCS

1. Test case 00_test.txt

   - Node visited: 200
   - Execution time: 12.6ms
   - Solution length: 5

   Papan cukup padat dan memerlukan kombinasi gerakan dari beberapa kendaraan. UCS berhasil menemukan solusi optimal dengan jumlah langkah minimum dan waktu eksekusi efisien.

2. Test case 01_test.txt

   - Node visited: 5

- Execution time: 1.6ms
- Solution length: 3

Papan sangat sederhana dan solusi dapat ditemukan secara langsung tanpa eksplorasi besar. UCS hanya perlu menelusuri sedikit node untuk mencapai goal.

3. Test case 02_test.txt
    - Node visited:11046
    - Execution time: 319.6ms
    - Solution length: 20

    Konfigurasi papan sangat kompleks dengan banyak kendaraan yang memblokir jalur. UCS tetap berhasil menemukan solusi optimal, namun dengan waktu dan node yang jauh lebih besar karena eksplorasi state yang luas.
4. Test case 03_test.txt
    - Node visited: 750
    - Execution time: 25.7ms
    - Solution length: 22

    Pada test case ini posisi exit terdapat pada posisi yang tidak biasa yaitu di atas, namun UCS tetap mampu menyelesaikan dengan solusi optimal. Jumlah node yang dikunjungi masih terkontrol dengan baik.

UCS adalah algoritma varian dari BFS, sehingga *Time Complexity*-nya adalah O(b^d), di mana b adalah branching factor dan d adalah kedalaman solusi. Semua node perlu disimpan dalam priority queue hingga goal ditemukan. Hal ini menyebabkan *Space Complexity*-nya O(b^d). Dari keempat kasus uji, terlihat bahwa waktu eksekusi tidak linier terhadap panjang solusi, melainkan sangat dipengaruhi oleh *branching factor* dan kepadatan papan juga cukup mempengaruhi. Kasus seperti pada kasus uji 01_test.txt yang solusinya pendek dan minim kendaraan (*Piece*), memberikan hasil yang cepat. Namun pada kasus uji 02_test.txt menunjukan bahwa meskipun solusinya hanya 20 langkah, UCS harus mengeksplorasi 11046 node, hal ini menunjukan bahwa tanpa heuristic, pencarian bisa sangat tidak efisien

6.2 Analisis GBFS

1. Test case 00_test.txt
    - Node visited: 51
    - Execution time: 6.7ms

- Solution length: 21

Walaupun cepat dan hanya mengunjungi 51 node, solusi yang ditemukan sangat panjang. Ini mencerminkan kelemahan utama GBFS: tidak mempertimbangkan cost sebelumnya, sehingga hanya fokus mengejar h(n) terpendek meskipun jalur tersebut tidak efisien.

2. Test case 01_test.txt
    - Node visited: 5
    - Execution time: 1.3ms
    - Solution length: 3

    GBFS berhasil langsung mengarahkan ke goal. Di kasus sederhana, GBFS bisa sangat efisien dan akurat.
3. Test case 02_test.txt
    - Node visited: 2669
    - Execution time: 68.9ms
    - Solution length: 37

    Node yang dikunjungi lebih sedikit dibandingkan algoritma sebelumnya (UCS), tapi solusinya sangat panjang (37 langkah). Ini menunjukkan *trade-off* antara kecepatan dan optimalitas. GBFS terlalu agresif pada arah "menuju goal", tapi tidak memperhatikan langkah-langkah suboptimal yang menumpuk.
4. Test case 03_test.txt
    - Node visited: 161
    - Execution time: 6.9ms
    - Solution length: 47

    GBFS lebih cepat dibandingkan algoritma sebelumnya (UCS) tapi tidak optimal. Hal ini menunjukan bahwa heuristik h(n) saja tidak cukup untuk mengevaluasi kualitas jalur

GBFS tidak menjamin eksplorasi optimal dan dapat "tersesat" meskipun branching factor (b) dan depth (m) kecil. Sehingga *Time Complexity*-nya O(b^m). *Space Complexity* dari GBFS sendiri adalah O(b^m) karena frontier dan visited yaitu

6.3 Analisis A*

1. Test case 00_test.txt
    - Node visited: 155
    - Execution time: 9ms
    - Solution length: 7

    A* Search dengan heuristic *manhattan distance* menghasilkan solusi yang cukup optimal
2. Test case 01_test.txt
    - Node visited: 5
    - Execution time: 0.1ms
    - Solution length: 3

    Bisa dilihat A* search dapat menghasilkan solusi yang lebih baik dari pada UCS dan GBFS
3. Test case 02_test.txt
    - Node visited: 10394
    - Execution time: 190.8ms
    - Solution length: 22

    Meski solusi ditemukan, jumlah node yang diekspansi sangat besar. Menunjukkan bahwa pada konfigurasi kompleks, kualitas heuristic sangat menentukan. Di kasus uji ini, *manhattan distance* tampaknya belum cukup informatif untuk menyempitkan pencarian secara drastis.
4. Test case 03_text.txt
    - Node visited: 502
    - Execution time: 12.6ms
    - Solution length: 22

    Hasil dari kasus uji menunjukan hasil yang lebih bagus dari UCS. Hal ini menunjukkan bahwa heuristic yang digunakan cukup baik untuk mempercepat pencarian.

A* Search memiliki Time Complexity dan Space Complexity yang bergantung pada kualitas heuristic. Umumnya *Time Complexity*-nya adalah O(b^d) dan *Space Complexity*-nya O(b^d). Dengan heuristic yang baik, A* Search dapat menjadi algoritma yang efisien dan optimal. Heuristic seperti *Blocking Pieces* dan *Distance to Exit* terbukti sangat membantu mempercepat pencarian dibanding hanya menggunakan *Manhattan Distance* saja.

6.4 Analisis Beam Search

1. Test case 00_test.txt
    - Node visited: 103
    - Execution time: 8.6ms
    - Solution length: 6
2. Test case 01_test.txt
    - Node visited: 5
    - Execution time: 0.2ms
    - Solution length: 3
3. Test case 03_test.txt
    - Node visited: 305
    - Execution time: 10.3ms
    - Solution length
4. Test case 04_test.txt
    - Node visited: 965
    - Execution time: 23.4ms
    - Solution length: 41

Beam Search adalah algoritma *heuristic-based* yang melakukan pencarian secara *level-by-level* seperti Algoritma BFS, namun pada setiap levelnya hanya mempertahankan sejumlah *best* node sesuai nilai heuristic. *Beam Search* tidak menjamin optimalitas karena bisa mengabaikan solusi yang lebih baik jika tidak masuk ke dalam beam width yang dipilih. Beam Search memiliki *Time Complexity* O(b.d) (b = beam width, d = kedalaman) karena hanya b node disimpan dan dievaluasi pada setiap level. Sedangkan untuk *Space Complexity*-nya sendiri adalah O(b). *Beam Search* hanya b node aktif pada setiap level.

BAB 7
Penjelasan Implementasi Bonus

7.1 Kakas yang Digunakan

7.1.1 Next

Next.js adalah kakas React modern yang membantu pembuatan web cepat dan SEO-*friendly*. Next.js yang digunakan adalah tipe app router sehingga dapat lebih baik merestrukturisasi kode program.

7.1.2 ts-priority-queue

Ts-priority-queue adalah kakas bantuan dari *library* Typescript untuk membantu membuat dan memproses data dalam *priority queue*. Kakas ini mempermudah implementasi algoritma pencarian rute pada kasus Rush Hour menggunakan A*, GBFS, UCS, dan Beam Search.

7.1.3 Tailwind

Tailwind adalah kakas yang mempermudah penulisan kode *styling* pada tiap komponen sebagai *utility-first* CSS. Kakas ini mempercepat proses pengembangan aplikasi dari sisi user interface (UI).

7.1.4 Zustand

Zustand adalah kakas *state management library* untuk React yang simpel dan ringan dalam menyimpan data serta mengaksesnya.

7.1.5 Sonner

Sonner digunakan sebagai bantuan *library* tampilan notifikasi yang ringan untuk React.

7.1.6 Vercel

Vercel adalah platform yang digunakan untuk men-*deploy* aplikasi ini dengan mudah dan cepat.

7.2 Menghubungkan Frontend dan Backend

Aplikasi ini menggunakan struktur *serverless* dan hanya bergantung pada fungsi dan model yang telah dibuat dalam bahasa Typescript pada direktori lib.

7.3 Cara Penggunaan

Opsi pertama, anda dapat mengaksesnya langsung melalui link deploy berikut:
https://rushpush.vercel.app/

Opsi kedua, anda dapat menjalankan perintah berikut:

- git clone https://github.com/Azzkaaaa/Tucil3_13523128_13523137.git
- npm install

- npm run dev

Kemudian akses localhost:3000 pada browser anda.

7.4 Algoritma Tambahan – Beam Search

Aplikasi ini menambahkan algoritma pencarian rute *Beam Search*, yaitu algoritma pencarian heuristik yang merupakan variasi dari Best-First Search, tetapi dengan pembatasan jumlah node yang dipertimbangkan pada setiap level. Alih-alih menyimpan semua kemungkinan node seperti pada A* atau BFS, Beam Search hanya menyimpan k node terbaik (berdasarkan nilai heuristik) pada setiap langkah, dengan k disebut sebagai *beam width*. Hal tersebut membuat *Beam Search* lebih efisien dalam penggunaan memori dan waktu, meskipun berisiko mengabaikan solusi optimal. Berikut adalah *pseudocode* Beam Search:

```
FUNCTION BeamSearch(initialBoard, heuristicType, beamWidth):
    startNode ← new SearchNode(initialBoard)
    beam ← [startNode]
    visited ← new Set()
    visited.ADD(startNode.getBoardString())

    nodesVisited ← 0
    maxBeamSize ← 1
    beamSizeHistory ← [1]
    depthReached ← 0
    startTime ← current time

    WHILE beam is not empty:
        depthReached ← depthReached + 1
        successorQueue ← new PriorityQueue ordered by Heuristic(node.board)

        FOR each node IN beam:
            nodesVisited ← nodesVisited + 1

            IF node.board is goal:
                RETURN SearchResult with path from node, success = true,
                       nodesVisited, and execution time

            possibleMoves ← getPossibleMoves(node.board)

            FOR each move IN possibleMoves:
                newBoard ← applyMove(node.board, move)
                boardString ← flatten newBoard into string

                IF boardString in visited:
                    CONTINUE
                visited.ADD(boardString)

                newNode ← new SearchNode(newBoard, node, move, node.cost + 1)
                successorQueue.QUEUE(newNode)

        IF successorQueue is empty:
            RETURN failure SearchResult

        beam ← empty list
        FOR i = 1 to MIN(beamWidth, successorQueue.length):
            beam.ADD(successorQueue.DEQUEUE())

        maxBeamSize ← max(maxBeamSize, beam.length)
        beamSizeHistory.ADD(beam.length)

    RETURN failure SearchResult
```

Gambar 7.4.1. *Pseudocode* Beam Search

Pencarian dimulai dari node awal (*initialBoard*) dan menyimpannya dalam beam, yaitu daftar kandidat node terbaik saat ini. Pada setiap iterasi, semua node dalam beam diperluas untuk menghasilkan

successors. Semua *successors* kemudian dimasukkan ke dalam *priority queue* dan disortir berdasarkan nilai heuristik yang mencerminkan seberapa dekat ke tujuan.

Setelah itu, hanya sejumlah *beamWidth* node dengan heuristik terbaik yang dipilih sebagai beam baru untuk iterasi berikutnya. Proses ini diulang sampai ditemukan solusi (*node* merupakan *goal*), atau *beam* kosong (tidak ada solusi lebih lanjut). Dengan begitu, *Beam Search* mengontrol memori dan waktu pencarian secara efisien dengan mengorbankan kemungkinan melewatkan solusi optimal karena hanya mempertahankan sebagian kecil dari seluruh kemungkinan jalur.

7.5 Multi Heuristik

Aplikasi ini mengimplementasikan 3 heuristik pada masing-masing algoritma pencarian rute A* dan Greedy Best-First Search (GBFS). Ketiga heuristik tersebut adalah *Manhattan Distance, Blocking Pieces,* dan *Distance to Exit*. Heuristik *Manhattan Distance* menghitung jarak lurus dari ujung primary piece ke pintu keluar (*exit*) baik secara horizontal maupun vertikal tergantung orientasi dan tanpa memperhatikan rintangan. Hal tersebut memberikan estimasi jarak minimum yang harus ditempuh. Kemudian, heuristik *Blocking Pieces* menghitung jumlah kendaraan yang menghalangi jalur *primary piece* ke exit. Setiap kendaraan penghalang dihitung satu kali (unik), lalu dikalikan dua untuk memberi penalti tambahan terhadap rintangan. Terakhir, heuristik *Distance to Exit* merupakan gabungan dari keduanya dengan formula manhattan + blocking.

Daftar Pustaka

1. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf
2. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf

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) sendiri	☑	☐
9. Program berhasil dikompilasi tanpa kesalahan	☑	☐

Link Repository : https://github.com/Azzkaaaa/Tucil3_13523128_13523137

Link Fork Repo (untuk deploy):

https://github.com/andi-frame/Tucil3_13523128_13523137

Link Deploy: https://rushpush.vercel.app/