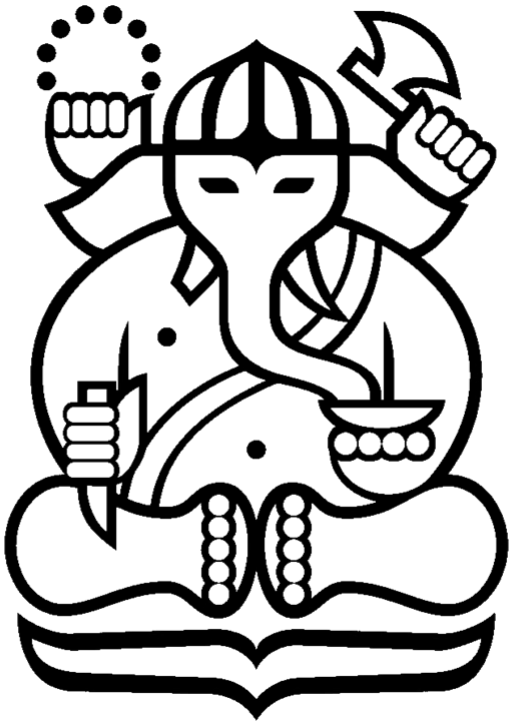
Tugas Kecil 3 IF2211 Strategi Algoritma
Penyelesaian Permainan Rush Hour Menggunakan Algoritma UCS, Greedy Best First Search, dan A*



Disusun oleh :

Hanif Kalyana Aditya (13523041)

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2025

DAFTAR ISI

BAB I
DESKRIPSI MASALAH

1.1 Algoritma UCS (*Uniform Cost Search)*

Algoritma dimana pencarian solusi optimal didasarkan pada fungsi evaluasi f(n) untuk setiap simpul, di mana f(n) = g(n), dimana g(n) adalah *cost* dari akar ke simpul n. Pada program ini, *cost* dihitung dengan menambah *cost* kumulatif sebesar 1 untuk setiap langkah (*cost* dibuat seragam). Algoritma ini memanfaatkan *priority queue* yang mengurutkan nilai g(n) dari yang terkecil.

1.2 Algoritma *Greedy Best First Search*

Algoritma yang menggunakan fungsi evaluasi f(n) untuk setiap simpul, di mana f(n) = h(n), yang merupakan perkiraan *cost* dari simpul n menuju tujuan. Pencarian *greedy best-first* akan memperluas simpul yang tampaknya paling dekat dengan tujuan. Algoritma ini memanfaatkan *priority queue* yang mengurutkan nilai h(n) dari yang terkecil. *Greedy Best First Search* memiliki beberapa permasalahan. Pertama, metode ini tidak lengkap. Kedua, metode ini rentan terjebak dalam optimal lokal minima atau plateau. Ketiga, pendekatan ini tidak dapat dibalik atau diubah (*irrevocable*).

1.3 Algoritma A*

Algoritma yang menghindari perluasan path yang *cost*-nya sudah bernilai tinggi. Fungsi evaluasi f(n) didefinisikan sebagai g(n) + h(n), di mana g(n) adalah *cost* yang telah dikeluarkan untuk mencapai simpul n, dan h(n) adalah perkiraan *cost* dari simpul n ke tujuan. Jadi, f(n) adalah perkiraan total cost path melalui simpul n ke tujuan. Algoritma ini memanfaatkan *priority queue* yang mengurutkan nilai f(n) dari yang terkecil. Heuristik yang digunakan pada algoritma A* admissible karena untuk setiap node n, $h(n) \leq h^*(n)$, di mana $h^*(n)$ adalah cost sebenarnya untuk mencapai keadaan tujuan dari n. Heuristik yang bersifat *admissible* tidak pernah melebih-lebihkan *cost* untuk mencapai tujuan, yaitu, heuristik ini bersifat optimis.

1.4 Rush Hour Puzzle

Rush Hour adalah sebuah permainan puzzle logika berbasis grid yang menantang pemain untuk menggeser kendaraan di dalam sebuah kotak (biasanya berukuran 6x6) agar mobil utama (biasanya berwarna merah) dapat keluar dari kemacetan melalui pintu keluar di sisi papan. Setiap

kendaraan hanya bisa bergerak lurus ke depan atau ke belakang sesuai dengan orientasinya (horizontal atau vertikal), dan tidak dapat berputar. Tujuan utama dari permainan ini adalah memindahkan mobil merah ke pintu keluar dengan jumlah langkah seminimal mungkin. Komponen penting dari permainan Rush Hour terdiri dari:

1. Papan

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

2. Piece

Piece adalah sebuah kendaraan di dalam papan. Setiap piece memiliki posisi, ukuran, dan orientasi. Orientasi sebuah piece hanya dapat berupa horizontal atau vertikal–tidak mungkin diagonal. Piece dapat memiliki beragam ukuran, yaitu jumlah cell yang ditempati oleh piece. Secara standar, variasi ukuran sebuah piece adalah 2-piece (menempati 2 cell) atau 3-piece (menempati 3 cell). Suatu piece tidak dapat digerakkan melewati/menembus piece yang lain.

3. Primary Piece

Primary piece adalah kendaraan utama yang harus dikeluarkan dari papan (biasanya berwarna merah). Hanya boleh terdapat satu primary piece.

4. Pintu Keluar

Pintu keluar adalah tempat primary piece dapat digerakkan keluar untuk menyelesaikan permainan

5. Gerakan

Gerakan yang dimaksudkan adalah pergeseran piece di dalam permainan. Piece hanya dapat

bergerak/bergeser lurus sesuai orientasinya (atas-bawah jika vertikal dan kiri-kanan jika horizontal). Suatu piece tidak dapat digerakkan melewati/menembus piece yang lain

BAB II
ANALISIS ALGORITMA

Definisi f(n) dan g(n)

Nilai f(n) adalah fungsi evaluasi keseluruhan yang digunakan untuk menentukan prioritas node dalam *priority queue*. Nilai f(n) berbeda-beda tergantung menggunakan algoritma apa. Nilai g(n) adalah *cost* aktual yang sudah dikeluarkan untuk mencapai node n dari node awal. Dalam Rush Hour, g(n) dihitung sebagai jumlah langkah atau transisi antara kata-kata. Nilai g(n) terus diperbarui saat node baru diproses. Sementara itu ada juga nilai h(n) yang merupakan estimasi *cost* terendah untuk mencapai node tujuan dari node n saat ini. Fungsi heuristik ini harus memperkirakan dengan akurat *cost* tersisa. Dalam Rush Hour, nilai h(n) dihitung sebagai dengan dua heuristik berbeda yaitu Manhattan Distance dan Blocking Piece. Manhattan Distance pada dasarnya menghitung jarak dari posisi mobil saat ini hingga ke pintu keluar. Sementara itu, blocking piece memperkirakan ada berapa mobil yang menghalangi jalan mobil utama ke pintu keluar. Pada algoritma UCS, nilai f(n) = g(n). Pada algoritma *Greedy Best First Search,* f(n) = h(n). Sedangkan pada algoritma A*, nilai f(n) = g(n) + h(n).

Apakah heuristik yang digunakan pada algoritma A* admissible?

Suatu heuristik dikatakan admissible jika tidak pernah melebihi biaya sebenarnya dari simpul n ke tujuan. Artinya, estimasi harus optimis atau akurasi bawah. Dalam konteks Rush Hour, Manhattan Distance untuk posisi mobil utama ke pintu keluar dianggap admissible, karena ia mengestimasi jumlah minimum langkah (tanpa memperhitungkan rintangan). Sementara itu, Blocking Heuristic (jumlah mobil yang menghalangi jalan keluar) juga admissible, selama hanya menghitung jumlah kendaraan (bukan jumlah langkah pasti untuk memindahkannya). Jadi, heuristik yang digunakan pada A* di program ini adalah admissible.

Pada penyelesaian Rush Hour, apakah algoritma UCS sama dengan BFS? (dalam artian urutan node yang dibangkitkan dan path yang dihasilkan sama)

Kedua algoritma yaitu UCS dan BFS pada dasarnya memiliki perberbedaan dalam prinsip. Algoritma

BFS memperlakukan semua langkah seolah biaya per langkah = 1 dan mengeksplorasi berdasarkan level kedalaman. Sementara itu, algoritma UCS hanya mempertimbangkan biaya aktual (g(n)) dan cocok jika ada biaya berbeda antar langkah. Meski demikian pada permainan ini, setiap langkah dianggap bernilai sama (1 langkah = 1 cost), maka UCS akan bersikap sama dengan BFS, menghasilkan urutan node dan path yang sama.

Secara teoritis, apakah algoritma A* lebih efisien dibandingkan dengan algoritma UCS untuk penyelesaian Rush Hour?

Secara teoritis iya. Penyebabnya adalah pada algoritma A* menggunakan f(n) = g(n) + h(n) yang memungkinkan pencarian lebih cepat dengan bantuan heuristik h(n). Sementara itu, algoritma UCS hanya mempertimbangkan f(n) = g(n) tanpa menjadikan langkah ke depan sebagai parameter. Akibatnya, lebih banyak node yang harus dieksplorasi. Dengan heuristik yang admissible, A* akan lebih efisien dari UCS karena mengeksplorasi lebih sedikit node namun tetap optimal.

Secara teoritis, apakah algoritma Greedy Best First Search menjamin solusi optimal untuk penyelesaian Rush Hour?

Tidak, karena GBFS hanya menggunakan f(n) = h(n) tanpa memperhatikan biaya yang telah dikeluarkan (g(n)) sehingga fokusnya hanya pada estimasi jarak ke tujuan. Akibatnya, GBFS bisa saja melewati solusi optimal karena hanya mengejar node yang terlihat “dekat” ke tujuan berdasarkan heuristik.

BAB III
SOURCE CODE PROGRAM

3.1 Repository Program

Berikut adalah pranala ke repository program:

https://github.com/hnfadtya/Tucil3_13523041

3.2 Source Code

3.2.1 AStar.java

```
package solver;

import heuristic.*;
import java.util.*;
import model.*;

public class AStar extends Solver {
   private HeuristicFunction heuristic;

   public AStar(HeuristicFunction heuristic) {
      this.heuristic = heuristic;
   }

   @Override
   public Result solve(Board initialBoard) {
      long startTime = System.nanoTime();
      PriorityQueue<Node> queue = new PriorityQueue<>(
         Comparator.comparingInt(n -> n.getCost() + heuristic.evaluate(n.getBoard()))
      );
      Set<String> visited = new HashSet<>();

      queue.add(new Node(initialBoard, new ArrayList<>(), 0));
      int visitedNodes = 0;
      while (!queue.isEmpty()) {
         Node current = queue.poll();
         visitedNodes++;

         Board board = current.getBoard();
         String key = generateBoardKey(board);
         if (visited.contains(key)) continue;
         visited.add(key);

         if (isGoal(board)) {
            long endTime = System.nanoTime();
```

```
            return new Result(current.getPath(), visitedNodes, (endTime - startTime) / 1_000_000);
        }

        queue.addAll(generateSuccessors(current));
      }

      return new Result(new ArrayList<>(), visitedNodes, 0);
   }
```

3.2.2 GreedyBFS.java

```
package solver;

import heuristic.*;
import java.util.*;
import model.*;

public class GreedyBFS extends Solver {
   private HeuristicFunction heuristic;

   public GreedyBFS(HeuristicFunction heuristic) {
      this.heuristic = heuristic;
      return new Result(new ArrayList<>(), visitedNodes, 0);
   }
}
   @Override
   public Result solve(Board initialBoard) {
      long startTime = System.nanoTime();
      PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(n ->
heuristic.evaluate(n.getBoard()))); // h(n)
      Set<String> visited = new HashSet<>();

      queue.add(new Node(initialBoard, new ArrayList<>(), 0));
      int visitedNodes = 0;

      while (!queue.isEmpty()) {
         Node current = queue.poll();
         visitedNodes++;

         Board board = current.getBoard();
```

```
            String key = generateBoardKey(board);
            if (visited.contains(key)) continue;
            visited.add(key);

            if (isGoal(board)) {

               long endTime = System.nanoTime();
               return new Result(current.getPath(), visitedNodes, (endTime - startTime) / 1_000_000);
            }
            queue.addAll(generateSuccessors(current));
          }
        return new Result(new ArrayList<>(), visitedNodes, 0);
      }
}
```

3.2.3 UCS.java

```
package solver;

import java.util.*;
import model.*;

public class UCS extends Solver {

   @Override
   public Result solve(Board initialBoard) {
      long startTime = System.nanoTime();
      long endTime;

      PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(n -> n.getCost()));
      Set<String> visited = new HashSet<>();

      Node startNode = new Node(initialBoard, new ArrayList<>(), 0);
      queue.add(startNode);

      int visitedNodes = 0;

      while (!queue.isEmpty()) {
         Node current = queue.poll(); // Returns: : the head of this queue, or null if this queue is
empty
         visitedNodes++;

         Board currentBoard = current.getBoard();

         String stateKey = generateBoardKey(currentBoard); // contoh:
A(0,2)(0,3)B(1,2)(2,2)C(4,3)(5,3)...
         if (visited.contains(stateKey)) continue;
         visited.add(stateKey);

         if (isGoal(currentBoard)) {
            endTime = System.nanoTime();
            return new Result(current.getPath(), visitedNodes, (endTime - startTime) / 1_000_000);
         }

         List<Node> successors = generateSuccessors(current);
         queue.addAll(successors);
      }

      endTime = System.nanoTime();
      return new Result(new ArrayList<>(), visitedNodes, (endTime - startTime) / 1_000_000); // no
solution found brader
   }

}
```

3.2.4 Solver.java

```
package solver;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.Map;
import model.*;

public abstract class Solver {
   public abstract  Result solve(Board initialBoard);

   protected boolean isGoal(Board board) {
      Piece primary = board.getPieces().get('P');
      Position exit = board.getExitPosition();

      for (Position pos : primary.getPositions()) {
         if (primary.isHorizontal()) {
            if (pos.row == exit.row &&
               (pos.col + 1 == exit.col || pos.col - 1 == exit.col)) {
               return true;
            }
         } else {
            if (pos.col == exit.col &&
               (pos.row + 1 == exit.row || pos.row - 1 == exit.row)) {
               return true;
            }
         }
      }

      return false;
   }

   protected List<Node> generateSuccessors(Node node) {
      List<Node> successors = new ArrayList<>();
      Board currentBoard = node.getBoard(); // ngambil board dulu
      Map<Character, Piece> pieces = currentBoard.getPieces(); // ngambil Map<Character, Piece>

      for (Piece piece : pieces.values()) { // ngecek tiap piece
         List<Move> possibleMoves = getValidMoves(currentBoard, piece); // list gerakan yg
memungkinkan (tidak berada di pinggir papan)
         for (Move move : possibleMoves) {
            Board newBoard = currentBoard.simulateMove(piece, move);
            if (newBoard != null) {
               List<Move> newPath = new ArrayList<>(node.getPath());
               newPath.add(move);
```

```
            successors.add(new Node(newBoard, newPath, node.getCost() + 1));
          }
        }
     }

     return successors;
  }

  protected List<Move> getValidMoves(Board board, Piece piece) {
     List<Move> moves = new ArrayList<>();
     char[][] grid = board.getGrid();
     int rows = board.getRows();
     int cols = board.getCols();
     List<Position> positions = piece.getPositions(); // cari tau posisi setiap kotaknya dimana aja

     positions.sort(Comparator.comparingInt(pos -> piece.isHorizontal() ? pos.col : pos.row)); //
menyesuaikan isi positions agar index pertama isinya kotak paling kiri atau paling atas

     if (piece.isHorizontal()) {
        Position leftMost = positions.get(0);
        int r = leftMost.row;
        int c = leftMost.col - 1;
        int steps = 0;

        while (c >= 0 && grid[r][c] == '.') { // akan dilewati jika leftMost.col = 0 (di pinggir kiri
papan)
           steps++;
           moves.add(new Move(piece.getId(), Move.Direction.LEFT, steps));
           c--;
        }

        Position rightMost = positions.get(positions.size() - 1);
        r = rightMost.row;
        c = rightMost.col + 1;
        steps = 0;

        while (c < cols && grid[r][c] == '.') { // akan dilewati jika rightMost.col = jumlah kolom
papan (di pinggir kanan papan)
           steps++;
           moves.add(new Move(piece.getId(), Move.Direction.RIGHT, steps));
           c++;
        }
     } else {
        Position topMost = positions.get(0);
        int r = topMost.row - 1;
        int c = topMost.col;
        int steps = 0;
```

```
        while (r >= 0 && grid[r][c] == '.') { // akan dilewati jika topMost.row = 0 (di pinggir atas papan)
           steps++;
           moves.add(new Move(piece.getId(), Move.Direction.UP, steps));
           r--;
        }

        Position bottomMost = positions.get(positions.size() - 1

        r = bottomMost.row + 1;
        c = bottomMost.col;
        steps = 0;

        while (r < rows && grid[r][c] == '.') { // akan dilewati jika bottomMost.row = jumlah baris papan (di pinggir bawah papan)
           steps++;
           moves.add(new Move(piece.getId(), Move.Direction.DOWN, steps));
           r++;
        }
     }

     return moves;
  }

  protected String generateBoardKey(Board board) {
     StringBuilder sb = new StringBuilder(); // Constructs a string builder with no characters in it and an initial capacity of 16 characters
     List<Character> ids = new ArrayList<>(board.getPieces().keySet()); // ingat bro getPieces returns Map<Character, Piece>
     Collections.sort(ids);

     for (char id : ids) {
        Piece piece = board.getPieces().get(id); // mengambil piece berdasarkan id yang terurut
        sb.append(id);
        for (Position pos : piece.getPositions()) {
           sb.append("(").append(pos.row).append(",").append(pos.col).append(")");
        }
     }

     return sb.toString();
  }
}
```

3.2.5 Node.java

```
package solver;

import java.util.List;
import model.Board;
import model.Move;

public class Node implements Comparable<Node> {
   private Board board;
   private List<Move> path;
   private int cost; // g(n)

   public Node(Board board, List<Move> path, int cost) {
      this.board = board;
      this.path = path;
      this.cost = cost;
   }

   public Board getBoard() {
      return board;
   }

   public List<Move> getPath() {
      return path;
   }

   public int getCost() {
      return cost;
   }

   @Override
   public int compareTo(Node other) {
      return Integer.compare(this.cost, other.cost);
   }
}
```

3.2.6 Result.java

```
package solver;

import java.util.List;
import model.Move;

public class Result {
   private List<Move> moves;
   private int visitedNodes;
   private long executionTime;

   public Result(List<Move> moves, int visitedNodes, long executionTime) {
      this.moves = moves;
      this.visitedNodes = visitedNodes;
      this.executionTime = executionTime;
   }

   public List<Move> getMoves() {
      return moves;
   }

   public int getVisitedNodes() {
      return visitedNodes;
   }

   public long getExecutionTime() {
      return executionTime;
   }
}
```

3.2.7 BoardReader.java

```
package parser;

import java.io.*;
import java.util.*;
import model.*;

public class BoardReader {

   public static Board readFromFile(String filename) throws IOException {
      BufferedReader br = new BufferedReader(new FileReader(filename));

      // 1. Baca dimensi papan
      String[] dims = br.readLine().trim().split(" ");
      int rows = Integer.parseInt(dims[0]);
      int cols = Integer.parseInt(dims[1]);

      // 2. Baca jumlah piece (tidak digunakan langsung)
      int pieceCount = Integer.parseInt(br.readLine().trim());

      // 3. Baca baris-baris konfigurasi
      List<String> rawLines = new ArrayList<>();
      String line;
      while ((line = br.readLine()) != null) {
         if (!line.trim().isEmpty()) {
            rawLines.add(line);
         }
      }
      br.close();

      // 4. Deteksi posisi pintu keluar dan bentuk grid murni
      Position exitPosition = null;
      List<String> boardLines = new ArrayList<>();

      if (rawLines.size() == rows + 1) { // K di atas atau bawah
         if (rawLines.get(0).contains("K")) { // K di atas
            int kCol = rawLines.get(0).indexOf('K');
            exitPosition = new Position(-1, kCol); // kRow = -1 (di atas)
            boardLines = rawLines.subList(1, rawLines.size()); // membuang baris paling atas
         } else { // K di bawah
            int kCol = rawLines.get(rawLines.size() - 1).indexOf('K');
            exitPosition = new Position(rows, kCol); // kRow = rows (di bawah)
            boardLines = rawLines.subList(0, rows); // membuang baris paling bawah
         }
      } else { // K ada di kiri atau kanan (di dalam baris grid)
         if ((rawLines.get(0)).charAt(0) == ' ') { // K di kiri
            for (int r = 0; r < rows; r++) {
               String row = rawLines.get(r);
               if (row.charAt(0) == 'K') {
                  exitPosition = new Position(r, -1); // kCol = -1 (di kiri)
```

```
            }
            boardLines.add(row.substring(1));  // membuang baris paling kiri
         }
      } else {
         for (int r = 0; r < rows; r++) {
            String row = rawLines.get(r);
            if (row.charAt(cols) == 'K') {
               exitPosition = new Position(r, cols); // kCol = cols (di kanan)
            }
            boardLines.add(row.substring(0, row.length() - 1)); // buang 'K'
         }
      }
   }

   // 5. Bangun grid dan mapping piece
   char[][] grid = new char[rows][cols];
   Map<Character, List<Position>> piecePositions = new HashMap<>();

   for (int r = 0; r < rows; r++) {
      String row = boardLines.get(r);
      for (int c = 0; c < cols; c++) {
         char ch = row.charAt(c);
         grid[r][c] = ch;
         // System.out.print(ch);

         if (Character.isUpperCase(ch)) {
            piecePositions.putIfAbsent(ch, new ArrayList<>());
            piecePositions.get(ch).add(new Position(r, c));
         }
      }
      // System.out.println();
   }

   // 6. Buat objek Piece
   Map<Character, Piece> pieces = new HashMap<>();
   for (Map.Entry<Character, List<Position>> entry : piecePositions.entrySet()) {
      char id = entry.getKey();
      pieces.put(id, new Piece(id, entry.getValue()));
   }

   // 7. Return board
   return new Board(rows, cols, grid, pieces, exitPosition);
  }
}
```

3.2.8 Board.java

```
package model;

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

public class Board {
   private int rows;
   private int cols;
   private char[][] grid;
   private Map<Character, Piece> pieces;
   private Position exitPosition;

   public Board(int rows, int cols, char[][] grid, Map<Character, Piece> pieces, Position
exitPosition) {
      this.rows = rows;
      this.cols = cols;
      this.grid = grid;
      this.pieces = pieces;
      this.exitPosition = exitPosition;
   }

   public int getRows() {
      return rows;
   }

   public int getCols() {
      return cols;
   }

   public char[][] getGrid() {
      return grid;
   }

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

   public Position getExitPosition() {
      return exitPosition;
   }

   public char getCell(int row, int col) {
      return grid[row][col];
   }

            return null;
       }
```

```
    for (int i = 0; i < getRows(); i++) {
        column[i] = grid[i][col];
    }
    return column;
}

public char[] getAllRow(int row) {
    return grid[row];
}

public void printBoard() {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            System.out.print(grid[i][j]);
        }
        System.out.println();
    }
}

public Board simulateMove(Piece piece, Move move) {
    int steps = move.getSteps();
    Move.Direction dir = move.getDirection();

    // Salin board ini
    Board copiedBoard = this.copyBoard();
    Map<Character, Piece> newPieces = copiedBoard.getPieces();
    Piece targetPiece = newPieces.get(piece.getId());
    List<Position> oldPositions = targetPiece.getPositions();
    char[][] newGrid = copiedBoard.getGrid();

    for (Position pos : oldPositions) {
        newGrid[pos.row][pos.col] = '.';
    }

    List<Position> newPositions = new ArrayList<>();
    for (Position pos : oldPositions) {
        int newRow = pos.row;
        int newCol = pos.col;

        switch (dir) {
            case UP -> newRow -= steps;
            case DOWN -> newRow += steps;
            case LEFT -> newCol -= steps;
            case RIGHT -> newCol += steps;
        }

        if (newRow < 0 || newRow >= rows || newCol < 0 || newCol >= cols) {
            return null;
        }
```

```
            if (newGrid[newRow][newCol] != '.') {
                return null;
            }

            newPositions.add(new Position(newRow, newCol));
        }

        for (Position pos : newPositions) {
            newGrid[pos.row][pos.col] = piece.getId();
        }

        targetPiece.setPositions(newPositions);
        return copiedBoard;
    }

    public Board copyBoard() {
        // mengcopy grid lama (deep copy)
        char[][] newGrid = new char[rows][cols];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                newGrid[i][j] = grid[i][j];
            }
        }

        // mengcopy pieces lama
        Map<Character, Piece> newPieces = new HashMap<>();
        for (Map.Entry<Character, Piece> entry : pieces.entrySet()) {
            List<Position> newPositions = new ArrayList<>();
            for (Position pos : entry.getValue().getPositions()) {
                newPositions.add(new Position(pos.row, pos.col));
            }
            newPieces.put(entry.getKey(), new Piece(entry.getKey(), newPositions));
        }

        return new Board(rows, cols, newGrid, newPieces, new Position(exitPosition.row,
exitPosition.col));
    }
}
```

3.2.9 Move.java

```
package model;


public class Move {
   private char pieceId;
   private Direction direction; // direction: orientasi piece
   private int steps; // steps: kotak keberapa dari sebuah piece. kotak paling kiri atau atas bernilai satu dan seterusnya


   public enum Direction {
      UP, DOWN, LEFT, RIGHT
   }


   public Move(char pieceId, Direction direction, int steps) {
      this.pieceId = pieceId;
      this.direction = direction;
      this.steps = steps;
   }


   public char getPieceId() {
      return pieceId;
   }


   public Direction getDirection() {
      return direction;
   }


   public int getSteps() {
      return steps;
   }


   public String toString() {
      return pieceId + " " + direction + " " + steps;
   }
}
```

3.2.10 Piece.java

```
package model;

import java.util.List;

public class Piece {
   private char id;
   private List<Position> positions;
   private boolean isHorizontal;

   public Piece(char id, List<Position> positions) {
      this.id = id;
      this.positions = positions;
      this.isHorizontal = determineOrientation();
   }

   private boolean determineOrientation() {
      if (positions.size() < 2) return true; // default horizontal kalo cm satu kotak
      Position first = positions.get(0);
      Position second = positions.get(1);
      return first.row == second.row;
   }

   public char getId() {
      return id;
   }

   public List<Position> getPositions() {
      return positions;
   }

   public boolean isHorizontal() {
      return isHorizontal;
   }
```

3.2.11 Position.java

```
package model;

public class Position {
   public int row;
   public int col;

   public Position(int row, int col) {
      this.row = row;
      this.col = col;
   }

   public boolean equals(Object obj) {
      if (this == obj) return true;
      if (!(obj instanceof Position)) return false;
      Position other = (Position) obj;
      return this.row == other.row && this.col == other.col;
   }

   public int hashCode() {
      return 31 * row + col;
   }

   public String toString() {
      return "(" + row + "," + col + ")";
   }
}
```

3.2.12 HeuristicFunction.java

```
package heuristic;

import model.Board;

public interface HeuristicFunction {
   int evaluate(Board board);
}
```

3.2.13 ManhattanDistanceHeuristic.java

```java
package heuristic;

import java.util.Comparator;
import java.util.List;
import model.*;

public class ManhattanDistanceHeuristic implements HeuristicFunction {

   @Override
   public int evaluate(Board board) {
      Piece primary = board.getPieces().get('P');
      List<Position> positions = primary.getPositions();
      positions.sort((a, b) -> primary.isHorizontal() ? a.col - b.col : a.row - b.row);

      Position farthest = positions.get(positions.size() - 1);
      Position exit = board.getExitPosition();

      if (primary.isHorizontal()) {
         positions.sort(Comparator.comparingInt(p -> p.col));
         int distance = exit.col - farthest.col;
         return Math.max(0, distance - 1);  // -1 karena harus berdiri 1 sebelum keluar
      } else {
         positions.sort(Comparator.comparingInt(p -> p.row));
         int distance = exit.row - farthest.row;
         return Math.max(0, distance - 1);
      }
   }
}
```

3.2.14 BlockingHeuristic.java

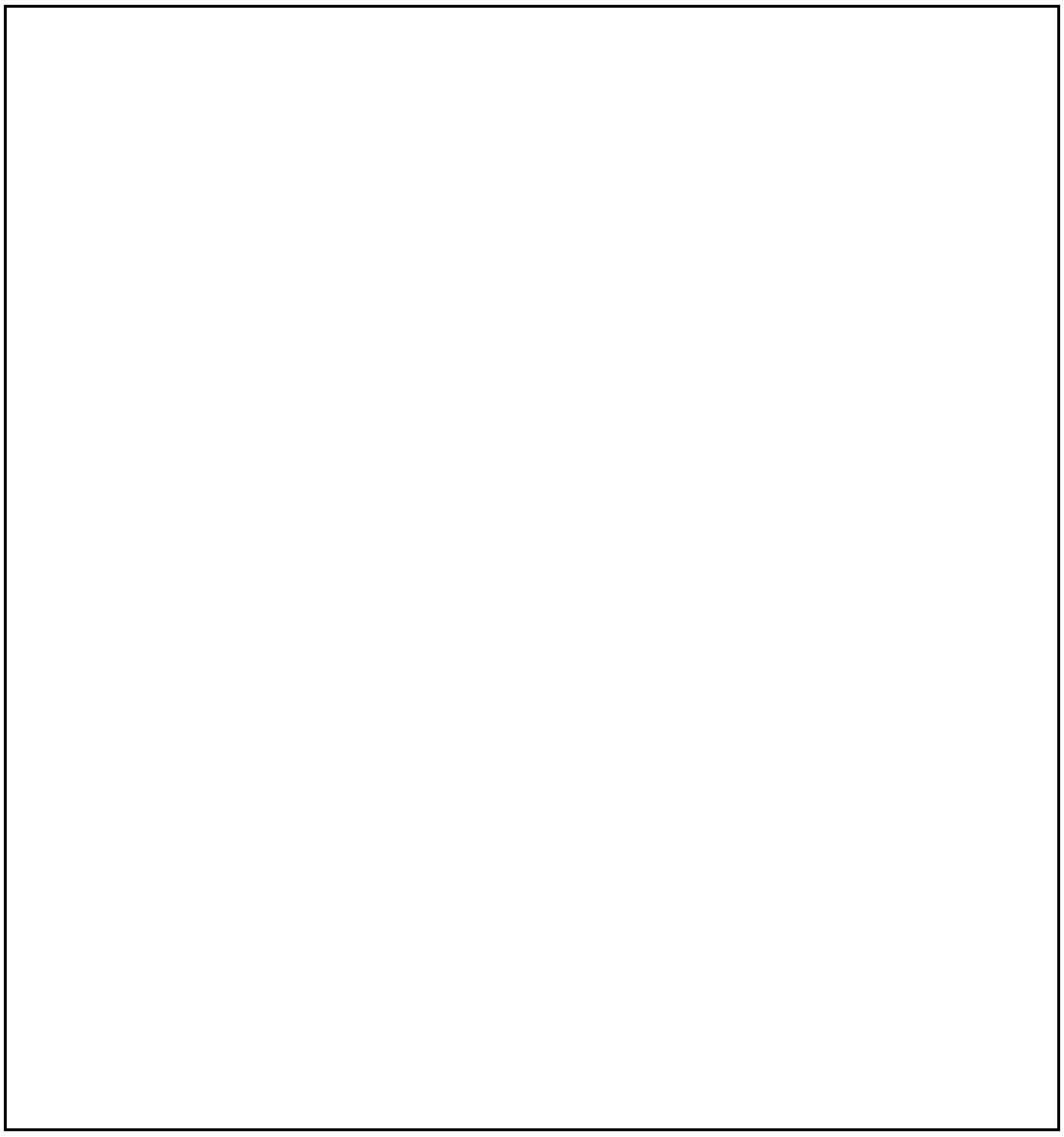
```
package heuristic;

import java.util.*;
import model.*;

public class BlockingHeuristic implements HeuristicFunction {

   @Override
   public int evaluate(Board board) {
      Piece primary = board.getPieces().get('P');
      List<Position> positions = primary.getPositions();
      positions.sort((a, b) -> primary.isHorizontal() ? a.col - b.col : a.row - b.row);

      Position farthest = positions.get(positions.size() - 1);
      Position exit = board.getExitPosition();

      if (primary.isHorizontal() && exit.col == board.getCols()) { // jika di exit di sebelah kanan
         int start = farthest.col + 1;
         char[] array = board.getAllRow(farthest.row); // baris di index row

         return countBlock(array, start, exit.col);
      } else if (!primary.isHorizontal() && exit.row == board.getRows()) { // jika di exit di sebelah
bawah
         int start = farthest.row + 1;
         char[] array = board.getAllColumn(farthest.col); // kolom di index col

         return countBlock(array, start, exit.row);
      } else if (primary.isHorizontal() && exit.col == -1) { // jika di exit di sebelah kiri
         int end = farthest.col - positions.size() + 1;
         char[] array = board.getAllRow(farthest.row); // baris di index row

         return countBlock(array, 0, end);
      } else { // jika di exit di sebelah atas
         int end = farthest.row  - positions.size() + 1;
         char[] array = board.getAllColumn(farthest.col); // kolom di index col

         return countBlock(array, 0, end);
      }
   }

   private static int countBlock(char[] array, int start, int end) {
      Set<Character> blockingPiece = new HashSet<>();
      for (int i = start; i < end; i++) {
         char id = array[i];
         if (id != '.') {
            if (!blockingPiece.contains(id)) blockingPiece.add(id);
         }
      }
      return blockingPiece.size();
```

BAB IV
MASUKAN DAN LUARAN PROGRAM

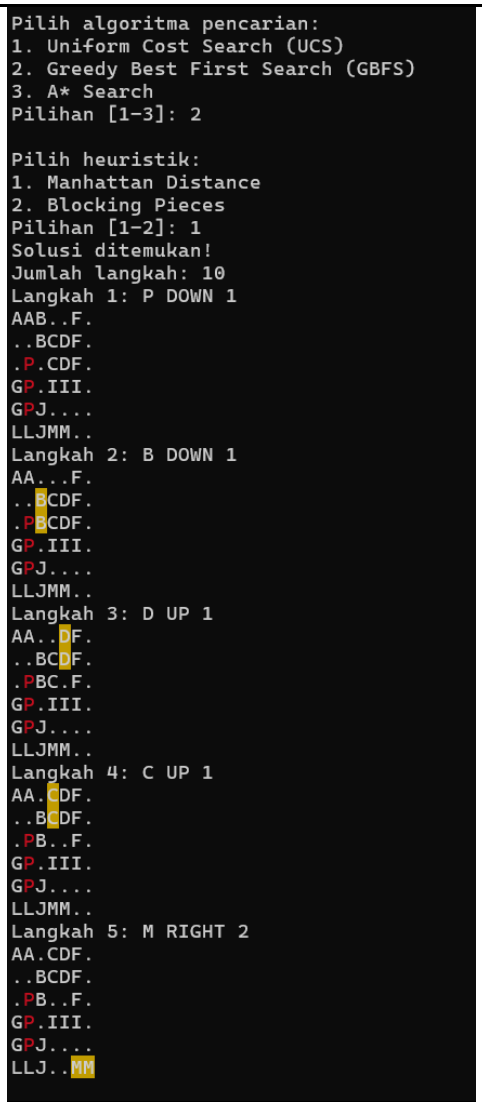
5.1 Test Case 1

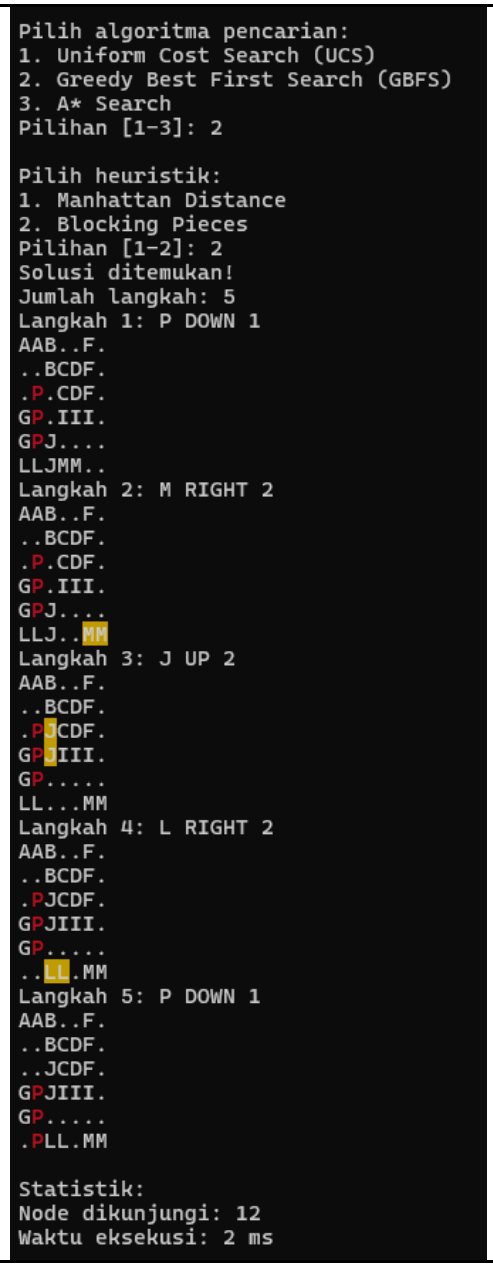
Uniform Cost Search (UCS)

```
Konfigurasi papan berhasil dibaca!

AAB..F.
.PBCDF.
.P.CDF.
GP.III.
G.J....
LLJMM..

Pilih algoritma pencarian:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Pilihan [1-3]: 1
Solusi ditemukan!
Jumlah langkah: 4
Langkah 1: M RIGHT 2
AAB..F.
.PBCDF.
.P.CDF.
GP.III.
G.J....
LLJ..MM
Langkah 2: J UP 1
AAB..F.
.PBCDF.
.P.CDF.
GPJIII.
G.J....
LL...MM
Langkah 3: L RIGHT 2
AAB..F.
.PBCDF.
.P.CDF.
GPJIII.
G.J....
..LL.MM
Langkah 4: P DOWN 2
AAB..F.
..BCDF.
...CDF.
GPJIII.
GPJ....
.PLL.MM

Statistik:
Node dikunjungi: 1683
Waktu eksekusi: 81 ms
```



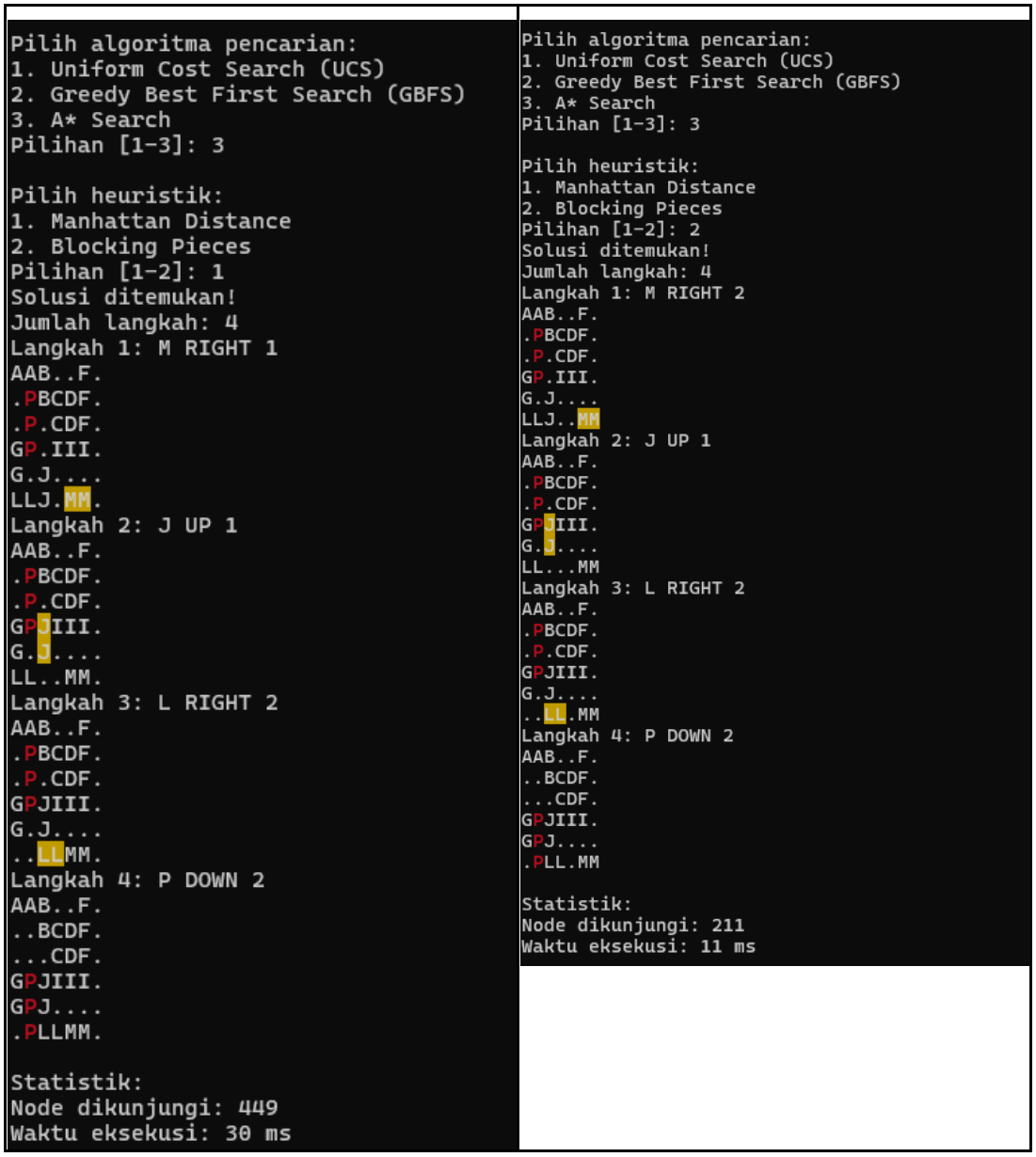
Greedy Best First Search (GBFS)	
Manhattan Distance	Blocking


Pilih algoritma pencarian:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Pilihan [1-3]: 2
Pilih heuristik:
1. Manhattan Distance
2. Blocking Pieces
Pilihan [1-2]: 1
Solusi ditemukan!
Jumlah langkah: 10
Langkah 1: P DOWN 1
AAB..F.
..BCDF.
.P.CDF.
GP.III.
GPJ....
LLJMM..
Langkah 2: B DOWN 1
AA...F.
..BCDF.
.PBCDF.
GP.III.
GPJ....
LLJMM..
Langkah 3: D UP 1
AA..DF.
..BCDF.
.PBC.F.
GP.III.
GPJ....
LLJMM..
Langkah 4: C UP 1
AA.CDF.
..BCDF.
.PB..F.
GP.III.
GPJ....
LLJMM..
Langkah 5: M RIGHT 2
AA.CDF.
..BCDF.
.PB..F.
GP.III.
GPJ....
LLJ..MM


Pilih algoritma pencarian:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Pilihan [1-3]: 2
Pilih heuristik:
1. Manhattan Distance
2. Blocking Pieces
Pilihan [1-2]: 2
Solusi ditemukan!
Jumlah langkah: 5
Langkah 1: P DOWN 1
AAB..F.
..BCDF.
.P.CDF.
GP.III.
GPJ....
LLJMM..
Langkah 2: M RIGHT 2
AAB..F.
..BCDF.
.P.CDF.
GP.III.
GPJ....
LLJ..MM
Langkah 3: J UP 2
AAB..F.
..BCDF.
.PJCDF.
GPJIII.
GP.....
LL...MM
Langkah 4: L RIGHT 2
AAB..F.
..BCDF.
.PJCDF.
GPJIII.
GP.....
..LL.MM
Langkah 5: P DOWN 1
AAB..F.
..BCDF.
..JCDF.
GPJIII.
GP.....
.PLL.MM
Statistik:
Node dikunjungi: 12
Waktu eksekusi: 2 ms

```
Langkah 6: J UP 1
AA.CDF.
..BCDF.
.PB..F.
GPJIII.
GPJ....
LL...MM
Langkah 7: M LEFT 2
AA.CDF.
..BCDF.
.PB..F.
GPJIII.
GPJ....
LL.MM..
Langkah 8: M RIGHT 1
AA.CDF.
..BCDF.
.PB..F.
GPJIII.
GPJ....
LL..MM.
Langkah 9: L RIGHT 2
AA.CDF.
..BCDF.
.PB..F.
GPJIII.
GPJ....
..LLMM.
Langkah 10: P DOWN 1
AA.CDF.
..BCDF.
..B..F.
GPJIII.
GPJ....
.PLLMM.

Statistik:
Node dikunjungi: 58
Waktu eksekusi: 19 ms
```

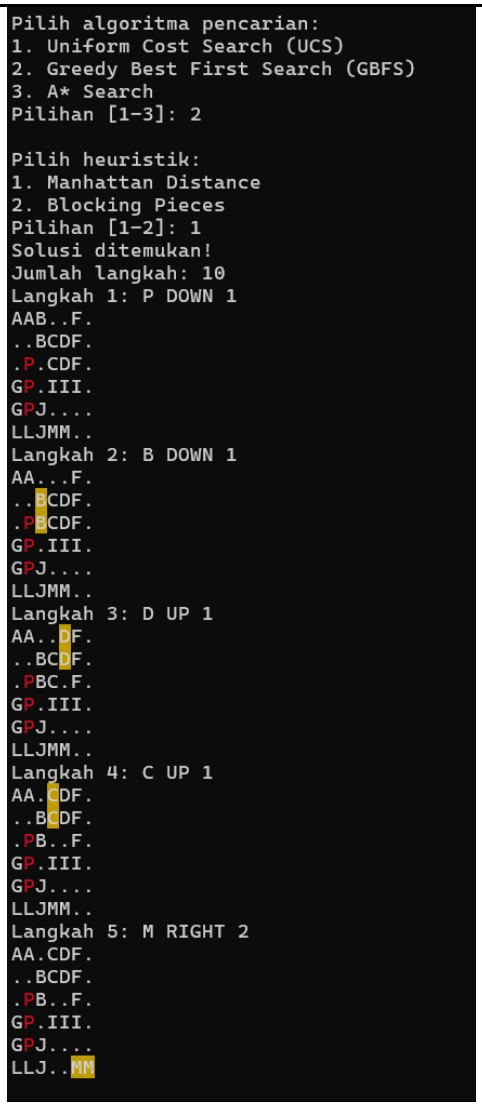
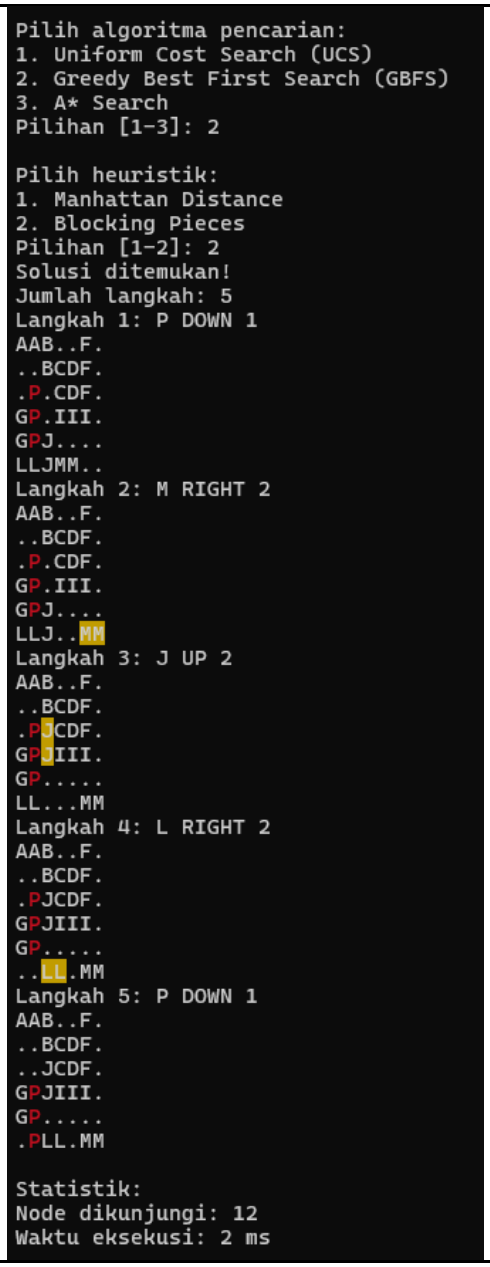
A*	
Manhattan Distance	Blocking


Pilih algoritma pencarian:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Pilihan [1-3]: 3
Pilih heuristik:
1. Manhattan Distance
2. Blocking Pieces
Pilihan [1-2]: 1
Solusi ditemukan!
Jumlah langkah: 4
Langkah 1: M RIGHT 1
AAB..F.
.PBCDF.
.P.CDF.
GP.III.
G.J....
LLJ.MM.
Langkah 2: J UP 1
AAB..F.
.PBCDF.
.P.CDF.
GPJIII.
G.J....
LL..MM.
Langkah 3: L RIGHT 2
AAB..F.
.PBCDF.
.P.CDF.
GPJIII.
G.J....
..LLMM.
Langkah 4: P DOWN 2
AAB..F.
..BCDF.
...CDF.
GPJIII.
GPJ....
.PLLMM.
Statistik:
Node dikunjungi: 449
Waktu eksekusi: 30 ms
Pilih algoritma pencarian:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Pilihan [1-3]: 3
Pilih heuristik:
1. Manhattan Distance
2. Blocking Pieces
Pilihan [1-2]: 2
Solusi ditemukan!
Jumlah langkah: 4
Langkah 1: M RIGHT 2
AAB..F.
.PBCDF.
.P.CDF.
GP.III.
G.J....
LLJ..MM
Langkah 2: J UP 1
AAB..F.
.PBCDF.
.P.CDF.
GPJIII.
G.J....
LL...MM
Langkah 3: L RIGHT 2
AAB..F.
.PBCDF.
.P.CDF.
GPJIII.
G.J....
..LL.MM
Langkah 4: P DOWN 2
AAB..F.
..BCDF.
...CDF.
GPJIII.
GPJ....
.PLL.MM
Statistik:
Node dikunjungi: 211
Waktu eksekusi: 11 ms

5.2 Test Case 2

Uniform Cost Search (UCS)

```
Konfigurasi papan berhasil dibaca!

AAB..F.
.PBCDF.
.P.CDF.
GP.III.
G.J....
LLJMM..

Pilih algoritma pencarian:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Pilihan [1-3]: 1
Solusi ditemukan!
Jumlah langkah: 4
Langkah 1: M RIGHT 2
AAB..F.
.PBCDF.
.P.CDF.
GP.III.
G.J....
LLJ..MM
Langkah 2: J UP 1
AAB..F.
.PBCDF.
.P.CDF.
GPJIII.
G.J....
LL...MM
Langkah 3: L RIGHT 2
AAB..F.
.PBCDF.
.P.CDF.
GPJIII.
G.J....
..LL.MM
Langkah 4: P DOWN 2
AAB..F.
..BCDF.
...CDF.
GPJIII.
GPJ....
.PLL.MM

Statistik:
Node dikunjungi: 1683
Waktu eksekusi: 81 ms
```

Greedy Best First Search (GBFS)	
Manhattan Distance	Blocking
	

```
Langkah 6: J UP 1
AA.CDF.
..BCDF.
.PB..F.
GPJIII.
GPJ....
LL...MM
Langkah 7: M LEFT 2
AA.CDF.
..BCDF.
.PB..F.
GPJIII.
GPJ....
LL.MM..
Langkah 8: M RIGHT 1
AA.CDF.
..BCDF.
.PB..F.
GPJIII.
GPJ....
LL..MM.
Langkah 9: L RIGHT 2
AA.CDF.
..BCDF.
.PB..F.
GPJIII.
GPJ....
..LLMM.
Langkah 10: P DOWN 1
AA.CDF.
..BCDF.
..B..F.
GPJIII.
GPJ....
.PLLMM.

Statistik:
Node dikunjungi: 58
Waktu eksekusi: 19 ms
```

A*	
Manhattan Distance	Blocking

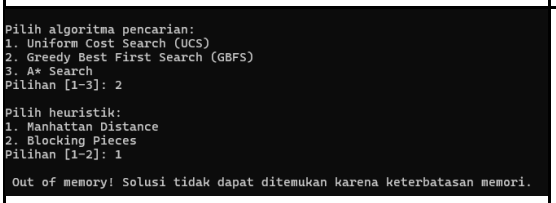
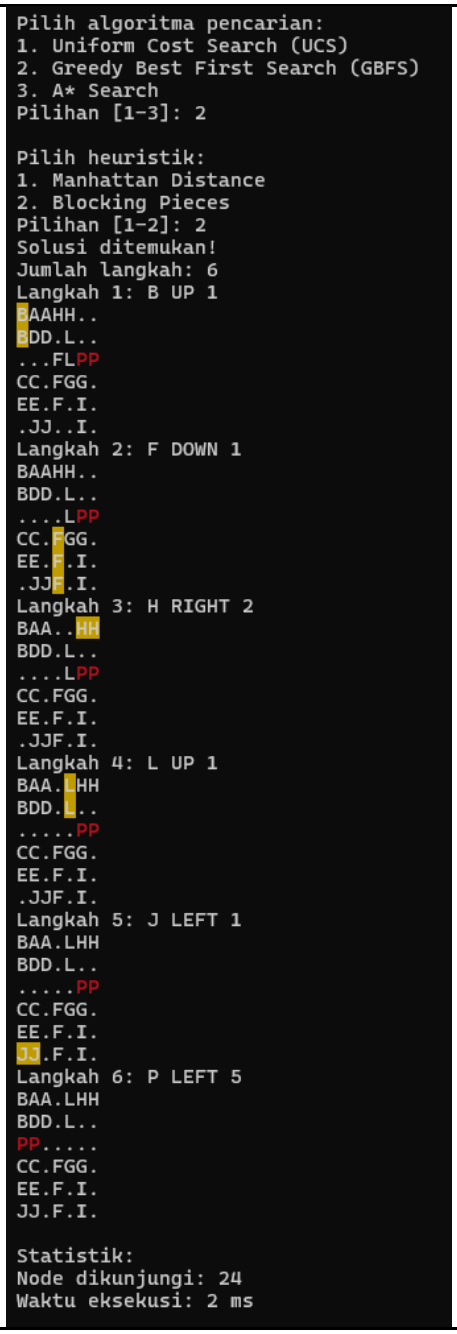
```
Pilih algoritma pencarian:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Pilihan [1-3]: 3

Pilih heuristik:
1. Manhattan Distance
2. Blocking Pieces
Pilihan [1-2]: 1
Solusi ditemukan!
Jumlah langkah: 4
Langkah 1: M RIGHT 1
AAB..F.
.PBCDF.
.P.CDF.
GP.III.
G.J....
LLJ.MM.
Langkah 2: J UP 1
AAB..F.
.PBCDF.
.P.CDF.
GPJIII.
G.J....
LL..MM.
Langkah 3: L RIGHT 2
AAB..F.
.PBCDF.
.P.CDF.
GPJIII.
G.J....
..LLMM.
Langkah 4: P DOWN 2
AAB..F.
..BCDF.
...CDF.
GPJIII.
GPJ....
.PLLMM.

Statistik:
Node dikunjungi: 449
Waktu eksekusi: 30 ms
```

```
Pilih algoritma pencarian:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Pilihan [1-3]: 3

Pilih heuristik:
1. Manhattan Distance
2. Blocking Pieces
Pilihan [1-2]: 2
Solusi ditemukan!
Jumlah langkah: 4
Langkah 1: M RIGHT 2
AAB..F.
.PBCDF.
.P.CDF.
GP.III.
G.J....
LLJ..MM
Langkah 2: J UP 1
AAB..F.
.PBCDF.
.P.CDF.
GPJIII.
G.J....
LL...MM
Langkah 3: L RIGHT 2
AAB..F.
.PBCDF.
.P.CDF.
GPJIII.
G.J....
..LL.MM
Langkah 4: P DOWN 2
AAB..F.
..BCDF.
...CDF.
GPJIII.
GPJ....
.PLL.MM

Statistik:
Node dikunjungi: 211
Waktu eksekusi: 11 ms
```

5.3 Test Case 3

Uniform Cost Search (UCS)

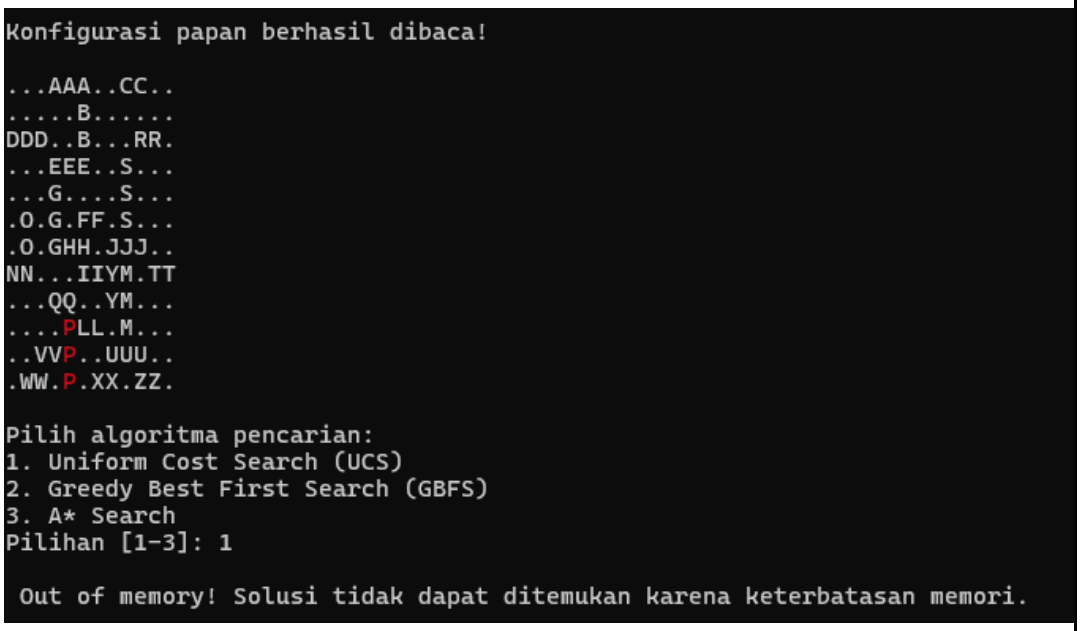
```
Pilih algoritma pencarian:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Pilihan [1-3]: 1
Solusi ditemukan!
Jumlah langkah: 5
Langkah 1: B UP 1
BAAHH..
BDD.L..
...FLPP
CC.FGG.
EE.F.I.
.JJ..I.
Langkah 2: F DOWN 1
BAAHH..
BDD.L..
....LPP
CC.FGG.
EE.F.I.
.JJF.I.
Langkah 3: H RIGHT 2
BAA..HH
BDD.L..
....LPP
CC.FGG.
EE.F.I.
.JJF.I.
Langkah 4: L UP 1
BAA.LHH
BDD.L..
.....PP
CC.FGG.
EE.F.I.
.JJF.I.
Langkah 5: P LEFT 5
BAA.LHH
BDD.L..
PP.....
CC.FGG.
EE.F.I.
.JJF.I.

Statistik:
Node dikunjungi: 7547
Waktu eksekusi: 208 ms
```

Greedy Best First Search (GBFS)	
Manhattan Distance	Blocking
	
A*	
Manhattan Distance	Blocking

```
Pilih algoritma pencarian:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Pilihan [1-3]: 3

Pilih heuristik:
1. Manhattan Distance
2. Blocking Pieces
Pilihan [1-2]: 1
Solusi ditemukan!
Jumlah langkah: 5
Langkah 1: B UP 1
BAAHH..
BDD.L..
...FLPP
CC.FGG.
EE.F.I.
.JJ..I.
Langkah 2: F DOWN 1
BAAHH..
BDD.L..
....LPP
CC.FGG.
EE.F.I.
.JJF.I.
Langkah 3: H RIGHT 2
BAA..HH
BDD.L..
....LPP
CC.FGG.
EE.F.I.
.JJF.I.
Langkah 4: L UP 1
BAA.LHH
BDD.L..
.....PP
CC.FGG.
EE.F.I.
.JJF.I.
Langkah 5: P LEFT 5
BAA.LHH
BDD.L..
PP.....
CC.FGG.
EE.F.I.
.JJF.I.

Statistik:
Node dikunjungi: 7547
Waktu eksekusi: 193 ms
```

```
Pilih algoritma pencarian:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Pilihan [1-3]: 3

Pilih heuristik:
1. Manhattan Distance
2. Blocking Pieces
Pilihan [1-2]: 2
Solusi ditemukan!
Jumlah langkah: 5
Langkah 1: B UP 1
BAAHH..
BDD.L..
...FLPP
CC.FGG.
EE.F.I.
.JJ..I.
Langkah 2: F DOWN 1
BAAHH..
BDD.L..
....LPP
CC.FGG.
EE.F.I.
.JJF.I.
Langkah 3: H RIGHT 2
BAA..HH
BDD.L..
....LPP
CC.FGG.
EE.F.I.
.JJF.I.
Langkah 4: L UP 1
BAA.LHH
BDD.L..
.....PP
CC.FGG.
EE.F.I.
.JJF.I.
Langkah 5: P LEFT 5
BAA.LHH
BDD.L..
PP.....
CC.FGG.
EE.F.I.
.JJF.I.

Statistik:
Node dikunjungi: 251
Waktu eksekusi: 18 ms
```

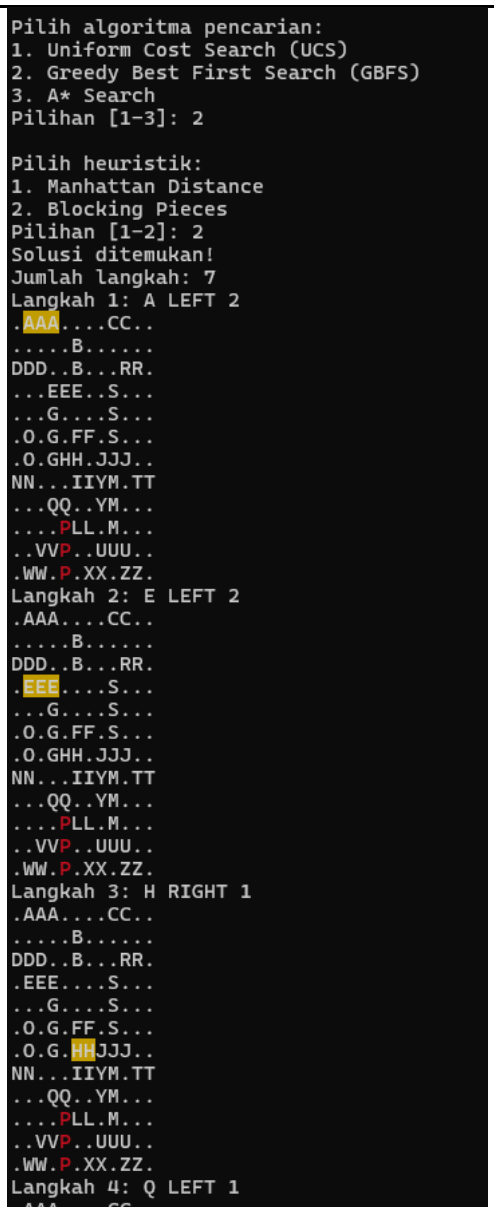
5.4 Test Case 4

Uniform Cost Search (UCS)


```
Konfigurasi papan berhasil dibaca!

...AAA..CC..
.....B......
DDD..B...RR.
...EEE..S...
...G....S...
.O.G.FF.S...
.O.GHH.JJJ..
NN...IIYM.TT
...QQ..YM...
....PLL.M...
..VVP..UUU..
.WW.P.XX.ZZ.

Pilih algoritma pencarian:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Pilihan [1-3]: 1

 Out of memory! Solusi tidak dapat ditemukan karena keterbatasan memori.
```

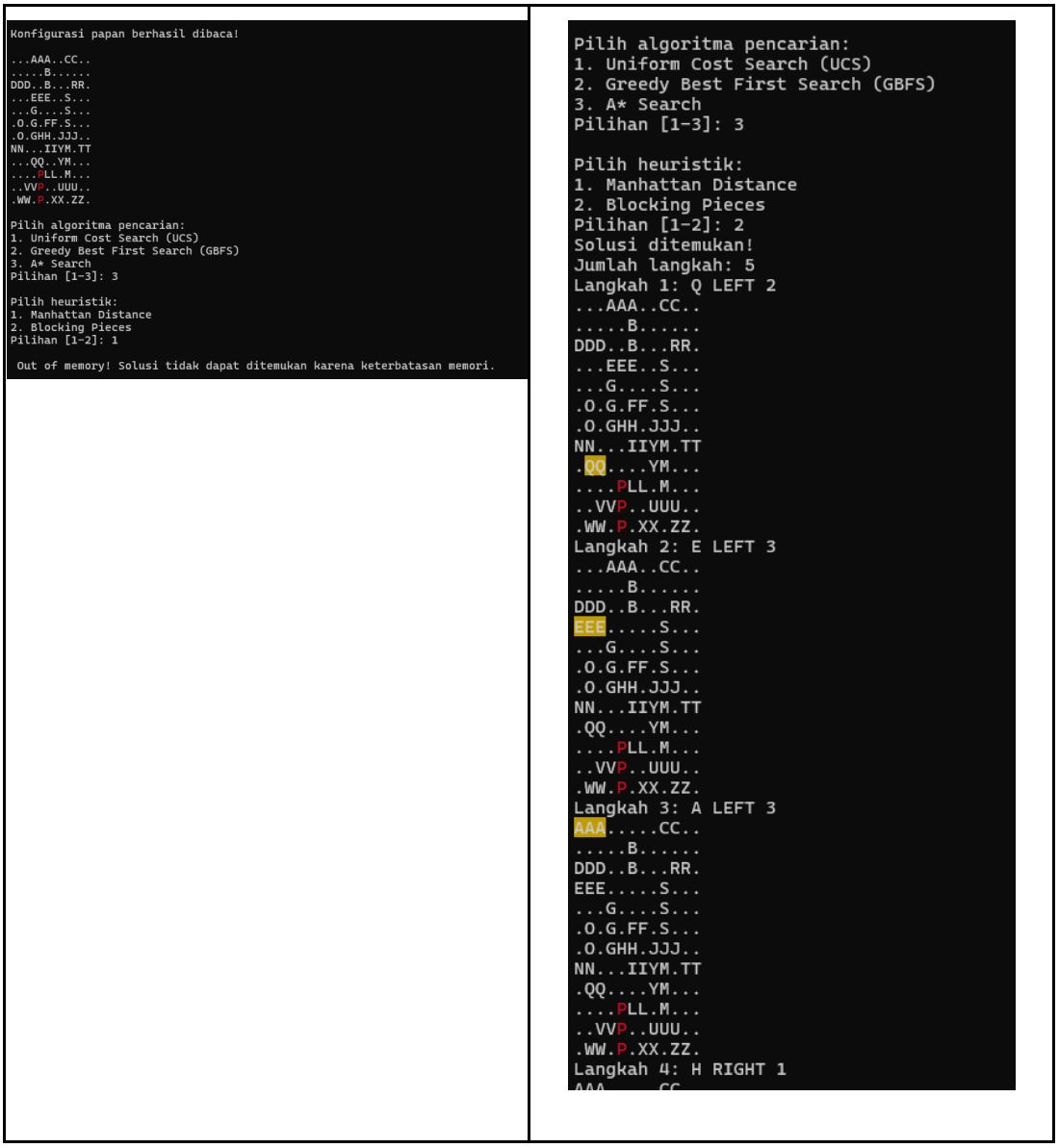
Greedy Best First Search (GBFS)	
Manhattan Distance	Blocking


...AAA..CC..
.....B.....
DDD..B...RR.
...EEE..S...
...G....S...
.O.G.FF.S...
.O.GHH.JJJ..
NN...IIYM.TT
...QQ..YM...
....PLL.M...
..VVP..UUU..
.WW.P.XX.ZZ.
Pilih algoritma pencarian:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Pilihan [1-3]: 2
Pilih heuristik:
1. Manhattan Distance
2. Blocking Pieces
Pilihan [1-2]: 1
Out of memory! Solusi tidak dapat ditemukan karena keterbatasan memori.


Pilih algoritma pencarian:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Pilihan [1-3]: 2
Pilih heuristik:
1. Manhattan Distance
2. Blocking Pieces
Pilihan [1-2]: 2
Solusi ditemukan!
Jumlah langkah: 7
Langkah 1: A LEFT 2
.AAA....CC..
.....B.....
DDD..B...RR.
...EEE..S...
...G....S...
.O.G.FF.S...
.O.GHH.JJJ..
NN...IIYM.TT
...QQ..YM...
....PLL.M...
..VVP..UUU..
.WW.P.XX.ZZ.
Langkah 2: E LEFT 2
.AAA....CC..
.....B.....
DDD..B...RR.
.EEE....S...
...G....S...
.O.G.FF.S...
.O.GHH.JJJ..
NN...IIYM.TT
...QQ..YM...
....PLL.M...
..VVP..UUU..
.WW.P.XX.ZZ.
Langkah 3: H RIGHT 1
.AAA....CC..
.....B.....
DDD..B...RR.
.EEE....S...
...G....S...
.O.G.FF.S...
.O.G.HHJJJ..
NN...IIYM.TT
...QQ..YM...
....PLL.M...
..VVP..UUU..
.WW.P.XX.ZZ.
Langkah 4: Q LEFT 1

<table>
<tr><td></td><td>
<pre>
Langkah 4: Q LEFT 1
.AAA....CC..
.....B......
DDD..B...RR.
.EEE....S...
...G....S...
.O.G.FF.S...
.O.G.HHJJJ..
NN...IIYM.TT
..QQ...YM...
....PLL.M...
..VVP..UUU..
.WW.P.XX.ZZ.
Langkah 5: G DOWN 1
.AAA....CC..
.....B......
DDD..B...RR.
.EEE....S...
........S...
.O.G.FF.S...
.O.G.HHJJJ..
NN.G.IIYM.TT
..QQ...YM...
....PLL.M...
..VVP..UUU..
.WW.P.XX.ZZ.
Langkah 6: A RIGHT 4
.....AAACC..
.....B......
DDD..B...RR.
.EEE....S...
........S...
.O.G.FF.S...
.O.G.HHJJJ..
NN.G.IIYM.TT
..QQ...YM...
....PLL.M...
..VVP..UUU..
.WW.P.XX.ZZ.
Langkah 7: P UP 9
....PAAACC..
....PB......
DDD.PB...RR.
.EEE....S...
........S...
.O.G.FF.S...
.O.G.HHJJJ..
NN.G.IIYM.TT
..QQ...YM...
.....LL.M...
..VV...UUU..
.WW...XX.ZZ.

Statistik:
Node dikunjungi: 73
Waktu eksekusi: 70 ms
</pre>
</td></tr>
<tr><td colspan="2">A*</td></tr>
<tr><td>Manhattan Distance</td><td>Blocking</td></tr>
</table>


Konfigurasi papan berhasil dibaca!
...AAA..CC..
.....B......
DDD..B...RR.
...EEE..S...
...G....S...
.O.G.FF.S...
.O.GHH.JJJ..
NN...IIYM.TT
...QQ..YM...
....PLL.M...
..VVP..UUU..
.WW.P.XX.ZZ.
Pilih algoritma pencarian:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Pilihan [1-3]: 3
Pilih heuristik:
1. Manhattan Distance
2. Blocking Pieces
Pilihan [1-2]: 1
Out of memory! Solusi tidak dapat ditemukan karena keterbatasan memori.
Pilih algoritma pencarian:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Pilihan [1-3]: 3
Pilih heuristik:
1. Manhattan Distance
2. Blocking Pieces
Pilihan [1-2]: 2
Solusi ditemukan!
Jumlah langkah: 5
Langkah 1: Q LEFT 2
...AAA..CC..
.....B......
DDD..B...RR.
...EEE..S...
...G....S...
.O.G.FF.S...
.O.GHH.JJJ..
NN...IIYM.TT
.QQ....YM...
....PLL.M...
..VVP..UUU..
.WW.P.XX.ZZ.
Langkah 2: E LEFT 3
...AAA..CC..
.....B......
DDD..B...RR.
EEE.....S...
...G....S...
.O.G.FF.S...
.O.GHH.JJJ..
NN...IIYM.TT
.QQ....YM...
....PLL.M...
..VVP..UUU..
.WW.P.XX.ZZ.
Langkah 3: A LEFT 3
AAA.....CC..
.....B......
DDD..B...RR.
EEE.....S...
...G....S...
.O.G.FF.S...
.O.GHH.JJJ..
NN...IIYM.TT
.QQ....YM...
....PLL.M...
..VVP..UUU..
.WW.P.XX.ZZ.
Langkah 4: H RIGHT 1

	 .WW.P.XX.ZZ. Langkah 4: H RIGHT 1 AAA.....CC.. .....B..... DDD..B...RR. EEE.....S... ...G....S... .O.G.FF.S... .O.G.HHJJJ.. NN...IIYM.TT .QQ....YM... ....PLL.M... ..VVP..UUU.. .WW.P.XX.ZZ. Langkah 5: P UP 9 AAA.P...CC.. ....PB..... DDD.PB...RR. EEE.....S... ...G....S... .O.G.FF.S... .O.G.HHJJJ.. NN...IIYM.TT .QQ....YM... .....LL.M... ..VV...UUU.. .WW...XX.ZZ. Statistik: Node dikunjungi: 674 Waktu eksekusi: 241 ms

LAMPIRAN

No	Poin	Ya	Tidak
1.	Program berhasil dikompilasi tanpa kesalahan	✓	
2.	Program berhasil dijalankan	✓	
3.	Solusi yang diberikan program benar dan mematuhi aturan permainan	✓	
4.	Program dapat membaca masukan berkas .txt dan menyimpan solusi berupa print board tahap per tahap dalam berkas .txt	✓	
5.	[Bonus] Implementasi algoritma pathfinding alternatif		✓
6.	[Bonus] Implementasi 2 atau lebih heuristik alternatif	✓	
7.	Bonus] Program memiliki GUI		✓
8.	Program dan laporan dibuat (kelompok) sendiri	✓	

REFERENSI

[1] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-(2025)-Bagian1.pdf#page=17.00

[2] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf