TUGAS KECIL 3

Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding

IF-2211

STRATEGI ALGORITMA

13523086 Bob Kunanda
13523088 Aryo Bama Wiratama

Dosen Pengampu:

Dr. Nur Ulfa Maulidevi, S.T, M.Sc.
Dr. Ir. Rinaldi Munir, M.T.
Monterico Adrian, S.T, M.T.

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
JL. GANESA 10, BANDUNG 40132
2025

DAFTAR ISI

BAB 1 PENJELASAN ALGORITMA

Uniform Cost Search (UCS)

Pseudocode

```
procedure UCS(start_board)
  visited ← empty set
  PQ ← priority queue ordered by cost g
  root ← BoardNode(start_board, g = 0, parent = null)
  PQ.push(root)

  while PQ is not empty do
    current ← PQ.pop()
    if isGoal(current.board) then
      return reconstruct_path(current)

    for each car in current.board.cars do
      for each valid_step in all_valid_steps(car, current.board) do
        new_board ← copy of current.board
        new_board.move(car.id, valid_step)
        next_node ← BoardNode(new_board, g = current.g + 1, parent =
current)
        state ← BoardState(new_board, valid_step, car.id)

        if state not in visited then
          PQ.push(next_node)
          visited.add(state)

  return "No solution found"
```

Penjelasan Algoritma

Algoritma ini menggunakan node yang menyimpan board state dan nilai cost nya. Untuk semua pergerakan akan dimasukkan ke priority queue dan kepala dari priority queue atau node dengan nilai cost terkecil akan dikeluarkan dari priority queue, diproses, dan dicarikan anaknya untuk dimasukkan ke priority queue lagi. Nilai cost ditentukan dengan step yang sudah diambil dari initial board state sampai current board state. Proses ini berjalan terus sampai priority queue kosong. Untuk menghindari terjadinya

infinite loop, algoritma juga menggunakan visited node untuk menandakan board state yang sudah dikunjungi.

Greedy Best First Search (GBFS)

Pseudocode

```
procedure GBFS(start_board, heuristic)
  visited ← empty set
  PQ ← priority queue ordered by heuristic value
  root ← BoardNode(start_board, h = heuristic(start_board), g = 0, parent
= null)
  PQ.push(root)

  while PQ is not empty do
    current ← PQ.pop()
    if isGoal(current.board) then
      return reconstruct_path(current)

    for each car in current.board.cars do
      for each valid_step in all_valid_steps(car, current.board) do
        new_board ← copy of current.board
        new_board.move(car.id, valid_step)
        next_node ← BoardNode(new_board, h = heuristic(new_board), g =
current.g + 1, parent = current)
        state ← BoardState(new_board, valid_step, car.id)

        if state not in visited then
          PQ.push(next_node)
          visited.add(state)

  return "No solution found"
```

Penjelasan Algoritma

Algoritma ini menggunakan node yang menyimpan board state dan nilai heuristik nya. Untuk semua pergerakan akan dimasukkan ke priority queue dan kepala dari priority queue atau node dengan nilai heuristik terkecil akan dikeluarkan dari priority queue, diproses, dan dicarikan anaknya untuk dimasukkan ke priority queue lagi. Nilai heuristik yang digunakan ada dua macam yaitu blocking car dan blocking chain. Blocking car hanya memperhitungkan banyak mobil yang menghalangi jalur keluar sedangkan blocking chain memperhitungkan banyak mobil yang menghalangi jalur dan mobi-mobil yang

menghalangi mobil tersebut. Proses ini berjalan terus sampai priority queue kosong. Untuk menghindari terjadinya infinite loop, algoritma juga menggunakan visited node untuk menandakan board state yang sudah dikunjungi.

A* Search

Pseudocode

```
procedure AStar(start_board, heuristic)
  visited ← empty set
  PQ ← priority queue ordered by f = g + h
  root ← BoardNode(start_board, g = 0, h = heuristic(start_board), f = g + h, parent = null)
  PQ.push(root)

  while PQ is not empty do
    current ← PQ.pop()
    if isGoal(current.board) then
      return reconstruct_path(current)

    for each car in current.board.cars do
      for each valid_step in all_valid_steps(car, current.board) do
        new_board ← copy of current.board
        new_board.move(car.id, valid_step)
        g ← current.g + 1
        h ← heuristic(new_board)
        next_node ← BoardNode(new_board, g = g, h = h, parent = current)
        state ← BoardState(new_board, valid_step, car.id)

        if state not in visited then
          PQ.push(next_node)
          visited.add(state)

  return "No solution found"
```

Penjelasan Algoritma

Algoritma ini menggunakan node yang menyimpan board state, nilai heuristik nya, dan nilai cost nya. Untuk semua pergerakan akan dimasukkan ke priority queue dan kepala dari priority queue atau node dengan nilai cost ditambah nilai heuristik terkecil akan dikeluarkan dari priority queue, diproses, dan dicarikan anaknya untuk dimasukkan ke priority queue lagi. Nilai cost ditentukan dengan step yang sudah

diambil dari initial board state sampai current board state. Nilai heuristik yang digunakan ada dua macam yaitu blocking car dan blocking chain. Blocking car hanya memperhitungkan banyak mobil yang menghalangi jalur keluar sedangkan blocking chain memperhitungkan banyak mobil yang menghalangi jalur dan mobi-mobil yang menghalangi mobil tersebut. Proses ini berjalan terus sampai priority queue kosong. Untuk menghindari terjadinya infinite loop, algoritma juga menggunakan visited node untuk menandakan board state yang sudah dikunjungi.

Beam Search

Pseudocode

```
procedure BeamSearch(start_board, heuristic, beamWidth)
  visited ← empty set
  PQ ← priority queue ordered by heuristic
  PQ.push(BoardNode(start_board, g=0, parent=null))
  found ← false
  currentNode ← null

  while PQ not empty do
    currentLevel ← all nodes in PQ
    clear PQ    children ← empty priority queue ordered by heuristic

    for node in currentLevel do
      currentNode ← node

      if isGoal(node.board) then
        found ← true
        break

      for car in node.board.cars do
        for step in validSteps(car, node.board) do
          newBoard ← copy and move car by step
          nextNode ← BoardNode(newBoard, g=node.g+1, parent=node,
h=heuristic(newBoard))
          state ← BoardState(newBoard, step, car.id)
          if state not in visited then
            children.push(nextNode)
            visited.add(state)

    if found then break

    for i in 1 to beamWidth do
```

```
    if children empty then break
    PQ.push(children.pop())

if not found then
  print "No solution found"
  return size(visited)

reconstruct path from currentNode
return size(visited)
```

Penjelasan Algoritma

Algoritma ini menggunakan node yang menyimpan board state dan nilai heuristik nya. Untuk semua pergerakan akan dimasukkan ke priority queue dan kepala dari priority queue atau node dengan nilai heuristik terkecil akan dikeluarkan dari priority queue, diproses, dan dicarikan anaknya untuk dimasukkan ke priority queue lagi. Nilai heuristik yang digunakan ada dua macam yaitu blocking car dan blocking chain. Blocking car hanya memperhitungkan banyak mobil yang menghalangi jalur keluar sedangkan blocking chain memperhitungkan banyak mobil yang menghalangi jalur dan mobi-mobil yang menghalangi mobil tersebut. Proses ini berjalan terus sampai priority queue kosong. Berbeda dengan GBFS, algoritma ini setelah mengambil kepala dengan nilai heuristik terkecil, algoritma akan menghapus saudaranya sehingga lebih memungkinkan untuk tidak mendapatkan solusi. Untuk menghindari terjadinya infinite loop, algoritma juga menggunakan visited node untuk menandakan board state yang sudah dikunjungi.

BAB 2 ANALISIS ALGORITMA

1. Definisi f(n) dan g(n)

Secara luas f(n) merupakan nilai yang digunakan sebagai pembanding untuk prioqueue dalam berbagai algoritma dan fungsi dari f(n) dapat berbeda setiap algoritmanya. g(n) merupakan nilai dari awal ke kondisi node tersebut sekarang.

Pada program yang kami gunakan berbeda algoritma memiliki nilai f(n) yang berbeda-beda seperti:

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

Dengan nilai g(n) merupakan jumlah step untuk mencapai kondisi board .

2. Apakah heuristik yang digunakan pada algoritma A* admissible?

Ya, semua heuristik yang digunakan pada algoritma A* admissible. Heuristik yang admissible artinya heuristic yang tidak melebih-lebihkan nilai sesungguhnya yang dalam hal ini adalah jumlah gerakan yang dibutuhkan pada game rush hour. Penting dalam algoritma A* untuk menggunakan heuristik yang admissible. Apabila heuristik yang digunakan tidak admissible, algoritma A* akan berpotensi menemukan solusi yang tidak optimal.

Terdapat dua heuristik yang kami gunakan pada program kami, yaitu Blocking Car dan Blocking Chain. Blocking Car menghitung semua mobil yang menghalangi pintu exit. Heuristik ini admissible karena untuk menyelesaikan puzzle, pemain setidaknya membutuhkan sejumlah gerakan untuk memindahkan mobil-mobil tersebut. Kedua kami menggunakan Blocking Chain, mobil yang dihitung pada heuristik ini **bukan hanya** mobil yang menghalangi exit (mobil penghalang), melainkan juga mobil yang menghalangi mobil penghalang. Heuristik ini admissible karena untuk menyelesaikan permainan, pemain juga harus menggerakkan mobil yang menghalangi mobil penghalang exit tersebut.

3. Apakah algoritma UCS sama dengan BFS?

Berbeda, UCS memprioritaskan untuk pemrosesan node dengan g(n) terkecil dahulu sedangkan BFS hanya mengambil yang pada queue terdepan. Perbedaan lain adalah UCS menggunakan prioqueue sedangkan BFS queue biasa.

4. Apakah algoritma A* lebih efisien dibandingkan dengan algoritma UCS pada penyelesaian Rush Hour?

Iya, karena A* mempertimbangkan heuristik h(n) ditambah g(n) sehingga prioritas node yang diambil lebih akurat dibandingkan UCS yang hanya membandingkan g(n) untuk priority queue nya ditambah lagi g(n) yang kami gunakan adalah berdasarkan steps yang diambil dari awal sampai mencapai board state tersebut sehingga dapat dipersepsikan kalau algoritma UCS memiliki percabangan dan pemrosesan yang sama dengan BFS.

5. Apakah algoritma Greedy Best First Search menjamin solusi optimal untuk penyelesaian Rush Hour?

Tidak, karena GBFS menggunakan strategi greedy di mana strategi ini tidak cocok untuk pencarian solusi optimal karena menggunakan pendekatan menggunakan estimasi heuristik h(n). Estimasi seperti heuristik hanya memberi jawaban yang estimasi juga bukan optimal.

BAB 3 SOURCE PROGRAM

Uniform Cost Search (UCS)

```
package com.project.backend.algorithms.UCS;
import java.util.Comparator;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;
import java.util.Set;

import com.project.backend.models.Board;
import com.project.backend.models.BoardNode;
import com.project.backend.models.BoardState;
import com.project.backend.models.Car;

public class UCS {

    public static int solveUCS(Board board, List<BoardState>
result){

        boolean found = false;

        Comparator<BoardNode> minHeap = (a, b) ->
Integer.compare(a.getG(), b.getG());

        PriorityQueue<BoardNode> pq = new PriorityQueue<>(minHeap);

        BoardNode root = new BoardNode(board, 0, null, 0, null);

        Set<BoardState> visited = new HashSet<>();

        BoardNode currentNode = null;

        pq.add(root);

        while (!pq.isEmpty()){

            currentNode = pq.poll();

            Board currentBoard = currentNode.getBoard();

            if (currentBoard.isSolve()){
                found = true;
                Board newBoard = new Board(currentBoard);
```

```
                newBoard.finalMove();

                currentNode = new
BoardNode(newBoard,0,0,currentNode,0,'P');
                break;
            }
            Map<Character, Car> cars = currentBoard.getCars();

            for (Car car : cars.values()){
                int startRow = car.getStartRow();
                int startCol = car.getStartCol();
                int step = 0;

                if (car.getOrientation().equals("horizontal")){

                    // Gerak ke kanan
                    while(startCol + car.getLength() + step <
currentBoard.getWidth()  && currentBoard.isSpaceEmpty(startRow,
startCol + car.getLength() + step)){
                        step++;
                    }

                    if (step != 0){
                        Board newBoard = new Board(currentBoard);
                        newBoard.move(car.getId(), step);

                        BoardNode nextNode = new
BoardNode(newBoard,currentNode.getG() + 1, currentNode, step,
car.getId());
                        BoardState nextState = new
BoardState(newBoard, step, car.getId());

                        if(visited.contains(nextState)){
                            continue;
                        }
                        pq.add(nextNode);
                        visited.add(nextState);

                    }

                    step = 0;
                    // Gerak ke kiri
                    while(startCol - 1 + step >= 0 &&
currentBoard.isSpaceEmpty(startRow, startCol + step - 1)){
                        step--;
                    }

                    if (step != 0){
                        Board newBoard = new Board(currentBoard);
                        newBoard.move(car.getId(), step);
```

```
                        BoardNode nextNode = new
BoardNode(newBoard,currentNode.getG() + 1, currentNode, step,
car.getId());
                        BoardState nextState = new
BoardState(newBoard, step, car.getId());

                        if(visited.contains(nextState)){
                            continue;
                        }
                        pq.add(nextNode);
                        visited.add(nextState);
                    }
                }else{
                    // Gerak ke bawah
                    while(startRow + car.getLength() + step <
currentBoard.getHeight() && currentBoard.isSpaceEmpty(startRow +
car.getLength() + step, startCol)){
                        step++;
                    }

                    if (step != 0){
                        Board newBoard = new Board(currentBoard);
                        newBoard.move(car.getId(), step);

                        BoardNode nextNode = new
BoardNode(newBoard,currentNode.getG() + 1, currentNode, step,
car.getId());
                        BoardState nextState = new
BoardState(newBoard, step, car.getId());

                        if(visited.contains(nextState)){
                            continue;
                        }
                        pq.add(nextNode);
                        visited.add(nextState);
                    }

                    step = 0;
                    // Gerak ke atas
                    while( startRow + step - 1>= 0 &&
currentBoard.isSpaceEmpty(startRow + step - 1, startCol)){
                        step--;
                    }

                    if (step != 0){
                        Board newBoard = new Board(currentBoard);
                        newBoard.move(car.getId(), step);

                        BoardNode nextNode = new
BoardNode(newBoard,currentNode.getG() + 1, currentNode, step,
car.getId());
```

```
                        BoardState nextState = new
BoardState(newBoard, step, car.getId());

                        if(visited.contains(nextState)){
                            continue;
                        }
                        pq.add(nextNode);
                        visited.add(nextState);
                    }
                }
            }
        }

        if(!found || currentNode == null){
            System.out.println("Tidak ada solusi yang ditemukan");
            return visited.size();
        }

        while(currentNode.getParent() != null){
            result.addFirst(currentNode.getState());
            currentNode =  currentNode.getParent();
        }
        return visited.size();
    }
}
```

Greedy Best First Search (GBFS)

```
package com.project.backend.algorithms.GBFS;
import java.util.Comparator;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;
import java.util.Set;

import com.project.backend.heuristic.CountHeuristic;
import com.project.backend.models.Board;
import com.project.backend.models.BoardNode;
import com.project.backend.models.BoardState;
import com.project.backend.models.Car;

public class GBFS {

    public static int solveGBFS(Board board, CountHeuristic
heuristic, List<BoardState> result){
```

```
        boolean found = false;

        Comparator<BoardNode> minHeap = (a, b) ->
Integer.compare(a.getH(), b.getH());

        PriorityQueue<BoardNode> pq = new PriorityQueue<>(minHeap);

        BoardNode root = new BoardNode(board, 0, null, 0, null);

        Set<BoardState> visited = new HashSet<>();

        BoardNode currentNode = null;

        pq.add(root);

        while (!pq.isEmpty()){

            currentNode = pq.poll();
            Board currentBoard = currentNode.getBoard();

            if (currentBoard.isSolve()){
                found = true;
                Board newBoard = new Board(currentBoard);
                newBoard.finalMove();

                currentNode = new
BoardNode(newBoard,0,0,currentNode,0,'P');
                break;
            }
            Map<Character, Car> cars = currentBoard.getCars();

            for (Car car : cars.values()){
                int startRow = car.getStartRow();
                int startCol = car.getStartCol();
                int step = 0;

                if (car.getOrientation().equals("horizontal")){

                    // Gerak ke kanan
                    while(startCol + car.getLength() + step <
currentBoard.getWidth()  && currentBoard.isSpaceEmpty(startRow,
startCol + car.getLength() + step)){
                        step++;
                    }

                    if (step != 0){
                        Board newBoard = new Board(currentBoard);
                        newBoard.move(car.getId(), step);

                        BoardNode nextNode = new
```

```
BoardNode(newBoard, heuristic.getValue(newBoard),
currentNode.getG() + 1,currentNode, step, car.getId());
                        BoardState nextState = new
BoardState(newBoard, step, car.getId());

                        if(visited.contains(nextState)){
                            continue;
                        }
                        pq.add(nextNode);
                        visited.add(nextState);
                    }

                    step = 0;
                    // Gerak ke kiri
                    while(startCol - 1 + step >= 0 &&
currentBoard.isSpaceEmpty(startRow, startCol + step - 1)){
                        step--;
                    }

                    if (step != 0){
                        Board newBoard = new Board(currentBoard);
                        newBoard.move(car.getId(), step);

                        BoardNode nextNode = new
BoardNode(newBoard, heuristic.getValue(newBoard),
currentNode.getG() + 1,currentNode, step, car.getId());
                        BoardState nextState = new
BoardState(newBoard, step, car.getId());

                        if(visited.contains(nextState)){
                            continue;
                        }
                        pq.add(nextNode);
                        visited.add(nextState);
                    }
                }else{
                    // Gerak ke bawah
                    while(startRow + car.getLength() + step <
currentBoard.getHeight() && currentBoard.isSpaceEmpty(startRow +
car.getLength() + step, startCol)){
                        step++;
                    }

                    if (step != 0){
                        Board newBoard = new Board(currentBoard);
                        newBoard.move(car.getId(), step);

                        BoardNode nextNode = new
BoardNode(newBoard, heuristic.getValue(newBoard),
currentNode.getG() + 1,currentNode, step, car.getId());
                        BoardState nextState = new
```

```
BoardState(newBoard, step, car.getId());

                        if(visited.contains(nextState)){
                            continue;
                        }
                        pq.add(nextNode);
                        visited.add(nextState);
                    }

                    step = 0;
                    // Gerak ke atas
                    while( startRow + step - 1>= 0 &&
currentBoard.isSpaceEmpty(startRow + step - 1, startCol)){
                        step--;
                    }

                    if (step != 0){
                        Board newBoard = new Board(currentBoard);
                        newBoard.move(car.getId(), step);

                        BoardNode nextNode = new
BoardNode(newBoard, heuristic.getValue(newBoard),
currentNode.getG() + 1,currentNode, step, car.getId());
                        BoardState nextState = new
BoardState(newBoard, step, car.getId());

                        if(visited.contains(nextState)){
                            continue;
                        }
                        pq.add(nextNode);
                        visited.add(nextState);
                    }
                }
            }
        }

        if(!found || currentNode == null){
            System.out.println("Tidak ada solusi yang ditemukan");
            return visited.size();
        }

        while(currentNode.getParent() != null){
            result.addFirst(currentNode.getState());
            currentNode =  currentNode.getParent();
        }
        return visited.size();
    }
}
```

A* Search

```
package com.project.backend.algorithms.AStar;

import java.util.Comparator;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;
import java.util.Set;

import com.project.backend.heuristic.CountHeuristic;
import com.project.backend.models.Board;
import com.project.backend.models.BoardNode;
import com.project.backend.models.BoardState;
import com.project.backend.models.Car;

public class AStar {

    public static int solveAStar(Board board, CountHeuristic
heuristic, List<BoardState> result){

        boolean found = false;

        Comparator<BoardNode> minHeap = (a, b) ->
Integer.compare(a.getG() + a.getH(), b.getG() + b.getH());

        PriorityQueue<BoardNode> pq = new PriorityQueue<>(minHeap);

        BoardNode root = new BoardNode(board, 0, null, 0, null);

        Set<BoardState> visited = new HashSet<>();

        BoardNode currentNode = null;

        pq.add(root);

        while (!pq.isEmpty()){

            currentNode = pq.poll();
            Board currentBoard = currentNode.getBoard();

            if (currentBoard.isSolve()){
                found = true;
                Board newBoard = new Board(currentBoard);
                newBoard.finalMove();

                currentNode = new
BoardNode(newBoard,0,0,currentNode,0,'P');
```

```
                break;
            }
            Map<Character, Car> cars = currentBoard.getCars();

            for (Car car : cars.values()){
                int startRow = car.getStartRow();
                int startCol = car.getStartCol();
                int step = 0;

                if (car.getOrientation().equals("horizontal")){

                    // Gerak ke kanan
                    while(startCol + car.getLength() + step <
currentBoard.getWidth()  && currentBoard.isSpaceEmpty(startRow,
startCol + car.getLength() + step)){ // Pake while biar mastiin
gerak ampe mentok
                        step++;
                    }

                    if (step != 0){
                        Board newBoard = new Board(currentBoard);
                        newBoard.move(car.getId(), step);

                        BoardNode nextNode = new
BoardNode(newBoard, heuristic.getValue(newBoard),
currentNode.getG() + 1,currentNode, step, car.getId());
                        BoardState nextState = new
BoardState(newBoard, step, car.getId());

                        if(visited.contains(nextState)){
                            continue;
                        }
                        pq.add(nextNode);
                        visited.add(nextState);
                    }

                    step = 0;
                    // Gerak ke kiri
                    while(startCol - 1 + step >= 0 &&
currentBoard.isSpaceEmpty(startRow, startCol + step - 1)){
                        step--;
                    }

                    if (step != 0){
                        Board newBoard = new Board(currentBoard);
                        newBoard.move(car.getId(), step);

                        BoardNode nextNode = new
BoardNode(newBoard, heuristic.getValue(newBoard),
currentNode.getG() + 1, currentNode, step, car.getId());
                        BoardState nextState = new
```

```
BoardState(newBoard, step, car.getId());

                        if(visited.contains(nextState)){
                            continue;
                        }
                        pq.add(nextNode);
                        visited.add(nextState);
                    }
                }else{
                    // Gerak ke bawah
                    while(startRow + car.getLength() + step <
currentBoard.getHeight() && currentBoard.isSpaceEmpty(startRow +
car.getLength() + step, startCol)){                          step++;
                    }

                    if (step != 0){
                        Board newBoard = new Board(currentBoard);
                        newBoard.move(car.getId(), step);

                        BoardNode nextNode = new
BoardNode(newBoard, heuristic.getValue(newBoard),
currentNode.getG() + 1, currentNode, step, car.getId());
                        BoardState nextState = new
BoardState(newBoard, step, car.getId());

                        if(visited.contains(nextState)){
                            continue;
                        }
                        pq.add(nextNode);
                        visited.add(nextState);
                    }

                    step = 0;
                    // Gerak ke atas
                    while( startRow + step - 1>= 0 &&
currentBoard.isSpaceEmpty(startRow + step - 1, startCol)){
step--;
                    }

                    if (step != 0){
                        Board newBoard = new Board(currentBoard);
                        newBoard.move(car.getId(), step);

                        BoardNode nextNode = new
BoardNode(newBoard, heuristic.getValue(newBoard),
currentNode.getG() + 1, currentNode, step, car.getId());
                        BoardState nextState = new
BoardState(newBoard, step, car.getId());

                        if(visited.contains(nextState)){
                            continue;
```

```
                        }
                        pq.add(nextNode);
                        visited.add(nextState);
                    }
                }
            }
        }

        if(!found || currentNode == null){
            System.out.println("Tidak ada solusi yang ditemukan");
        }


        while(currentNode.getParent() != null){
            result.addFirst(currentNode.getState());
            currentNode =  currentNode.getParent();
        }

        return visited.size();
    }
}
```

Beam Search

```
package com.project.backend.algorithms.Beam;
import java.util.Comparator;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;
import java.util.Set;

import com.project.backend.heuristic.CountHeuristic;
import com.project.backend.models.Board;
import com.project.backend.models.BoardNode;
import com.project.backend.models.BoardState;
import com.project.backend.models.Car;

public class Beam {

    public static int solveBeamSearch(Board board, CountHeuristic
heuristic, List<BoardState> result, int beamWidth){

        boolean found = false;

        Comparator<BoardNode> minHeap =
Comparator.comparingInt(BoardNode::getH);
        PriorityQueue<BoardNode> pq = new PriorityQueue<>(minHeap);
```

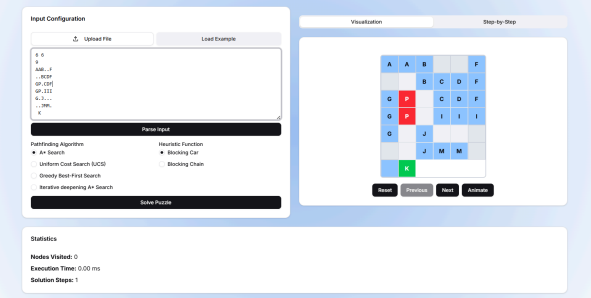
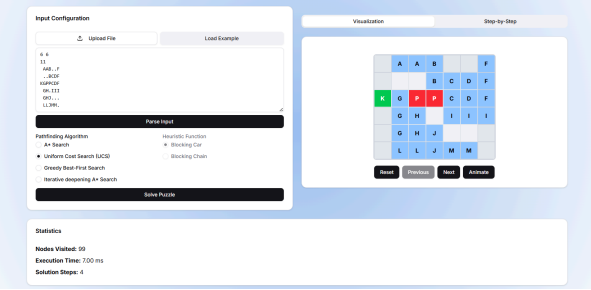
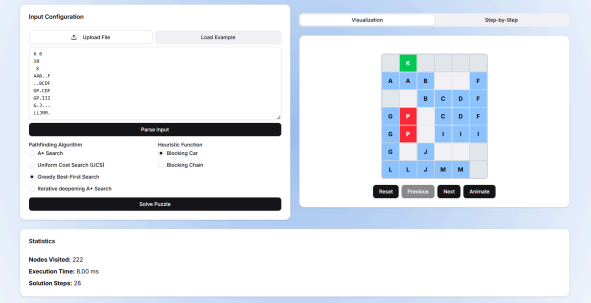
```
        BoardNode root = new BoardNode(board, 0, null, 0, null);
        Set<BoardState> visited = new HashSet<>();
        BoardNode currentNode = null;

        pq.add(root);

        while (!pq.isEmpty()) {
            List<BoardNode> currentLevel = new
java.util.ArrayList<>(pq);
            pq.clear();

            PriorityQueue<BoardNode> children = new
PriorityQueue<>(minHeap);

            for (BoardNode parentNode : currentLevel) {
                currentNode = parentNode;
                Board currentBoard = currentNode.getBoard();

                if (currentBoard.isSolve()) {
                    found = true;
                    Board newBoard = new Board(currentBoard);
                    newBoard.finalMove();
                    currentNode = new BoardNode(newBoard, 0, 0,
currentNode, 0, 'P');
                    break;
                }

                Map<Character, Car> cars = currentBoard.getCars();

                for (Car car : cars.values()) {
                    int startRow = car.getStartRow();
                    int startCol = car.getStartCol();
                    int step;

                    if (car.getOrientation().equals("horizontal"))
{
                        // Gerak ke kanan
                        step = 1;
                        while (startCol + car.getLength() + step -
1 < currentBoard.getWidth() &&
                                currentBoard.isSpaceEmpty(startRow,
startCol + car.getLength() + step - 1)) {

                            Board newBoard = new
Board(currentBoard);
                            newBoard.move(car.getId(), step);

                            BoardNode nextNode = new
BoardNode(newBoard, heuristic.getValue(newBoard),
                                    currentNode.getG() + 1,
```

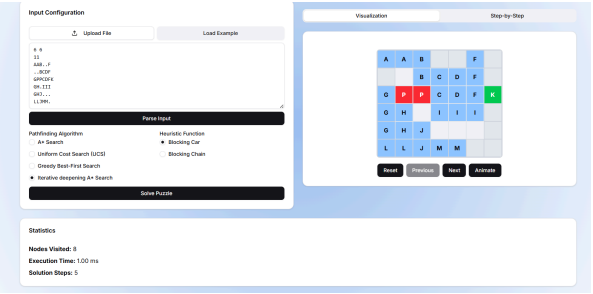
```
currentNode, step, car.getId());

                            BoardState nextState = new
BoardState(newBoard, step, car.getId());
                            if (!visited.contains(nextState)) {
                                children.add(nextNode);
                                visited.add(nextState);
                            }
                            step++;
                        }

                        // Gerak ke kiri
                        step = -1;
                        while (startCol + step >= 0 &&
                               currentBoard.isSpaceEmpty(startRow,
startCol + step)) {

                            Board newBoard = new
Board(currentBoard);
                            newBoard.move(car.getId(), step);

                            BoardNode nextNode = new
BoardNode(newBoard, heuristic.getValue(newBoard),
                                      currentNode.getG() + 1,
currentNode, step, car.getId());

                            BoardState nextState = new
BoardState(newBoard, step, car.getId());
                            if (!visited.contains(nextState)) {
                                children.add(nextNode);
                                visited.add(nextState);
                            }
                            step--;
                        }
                    } else {
                        // Gerak ke bawah
                        step = 1;
                        while (startRow + car.getLength() + step -
1 < currentBoard.getHeight() &&
                               currentBoard.isSpaceEmpty(startRow +
car.getLength() + step - 1, startCol)) {

                            Board newBoard = new
Board(currentBoard);
                            newBoard.move(car.getId(), step);

                            BoardNode nextNode = new
BoardNode(newBoard, heuristic.getValue(newBoard),
                                      currentNode.getG() + 1,
currentNode, step, car.getId());
```

```
                            BoardState nextState = new
BoardState(newBoard, step, car.getId());
                            if (!visited.contains(nextState)) {
                                children.add(nextNode);
                                visited.add(nextState);
                            }
                            step++;
                        }

                        // Gerak ke atas
                        step = -1;
                        while (startRow + step >= 0 &&
                               currentBoard.isSpaceEmpty(startRow +
step, startCol)) {

                            Board newBoard = new
Board(currentBoard);
                            newBoard.move(car.getId(), step);

                            BoardNode nextNode = new
BoardNode(newBoard, heuristic.getValue(newBoard),
                                      currentNode.getG() + 1,
currentNode, step, car.getId());

                            BoardState nextState = new
BoardState(newBoard, step, car.getId());
                            if (!visited.contains(nextState)) {
                                children.add(nextNode);
                                visited.add(nextState);
                            }
                            step--;
                        }
                    }
                }
            }

            if (found) break;

            int added = 0;
            while (!children.isEmpty() && added < beamWidth) {
                pq.add(children.poll());
                added++;
            }
        }
        if (!found || currentNode == null) {
            System.out.println("Tidak ada solusi yang ditemukan");
            return visited.size();
        }
        while (currentNode.getParent() != null) {
            result.addFirst(currentNode.getState());
            currentNode = currentNode.getParent();
```

```
            }

            return visited.size();
        }
}
```

BAB 4 TANGKAPAN LAYAR

No.	Keterangan	Tangkapan Layar
1.	Exit berada di bawah dan menggunakan algoritma A*	
2.	Exit berada di kiri dan menggunakan algoritma UCS	
3.	Exit berada di atas dan menggunakan algoritma GBFS	

4.	Exit berada di kanan dan menggunakan algoritma IDA	

BAB 5 HASIL ANALISIS PERCOBAAN ALGORITMA PATHFINDING

Kompleksitas Algoritma

Dalam kasus terburuk A*, GBFS, dan UCS akan memiliki kompleksitas O(b^m) dengan b adalah branching factor dan m adalah kedalamannya. Pada kasus ini algoritma UCS sebenarnya akan bekerja seperti caranya BFS karena ketika memperhatikan steps, maka node dengan depth terendah akan diprioritaskan. Namun untuk A* dan GBFS karena menggunakan heuristik, maka keduanya untuk rata-rata akan lebih cepat dari UCS dan BFS. Dalam kasus terburuk dan rata-rata Beam search akan memiliki kompleksitas waktu O(bm) karena algoritma ini tidak memedulikan saudaranya dari node yang diproses sehingga proses dari algoritma ini akan menjadi sangat cepat namun mengorbankan ketepatan.

BAB 6 PENJELASAN IMPLEMENTASI BONUS

Algoritma Alternatif

Algoritma alternatif yang digunakan adalah Beam search. Beam search memiliki kesamaan dengan GBFS yaitu menggunakan heuristik, namun perbedaanya adalah GBFS masih menyimpan saudara dari node yang diproses sedangkan Beam search tidak. Hal ini lah yang membuatnya menjadi algoritma tercepat dibandingkan dengan yang lain dengan mengorbankan ketepatan sehingga memungkinkan terjadinya false positive pada hasil yang seharusnya memiliki jawaban.

Heuristic Alternatif

Kami menggunakan dua heuristic yaitu blocking car dan blocking chain. Blocking car menghitung jumlah jalur antara primary dengan exit yang terhalang. Blocking chain sebenarnya juga menghitung jumlah jalur yang terhalang dengan tambahan jumlah mobil yang menghalang halangan tersebut.

Graphic User Interface

Pada tugas ini, kami menggunakan bahasa Java untuk backend, Typescript untuk frontend, dan Tailwind untuk styling. Untuk penyambungan backend dengan frontend dibantu dengan Spinboot. Terdapat 3 cara input yang dapat dilakukan:

1. Input secara graphical
   Input ini dapat dilakukan pada puzzle editor dengan menaruh piece-piece yang diinginkan.
2. Input melalui penulisan txt
   Pada puzzle solver terdapat input configuration untuk memasukkan format txt melalui ketikan
3. Input melalui upload txt
   Pada puzzle solver dapat juga dilakukan upload txt melalui tombol upload file.

Lalu setelah input, dapat dilakukan parsing yang akan mengubah data txt menjadi data yang sesuai. Algoritma dan heuristic yang diinginkan dapat dipilih pada tombol di bawah parse input. Penyelesaian puzzle lalu bisa dilakukan dengan tombol solve puzzle. Pada tahap visualisasi terdapat pilihan animasi untuk melihat step by step dari penyelesaian puzzle atau bisa melalui laman step by step jika tidak ingin melihat animasi. Pada bawah halaman terdapat statistik yang menampilkan node yang dikunjungi, waktu eksekusi, dan steps untuk mencapai solusi.

LAMPIRAN

Repository GitHub

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	✅	