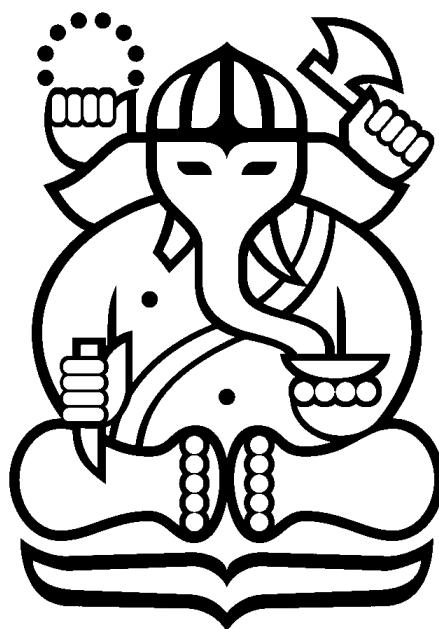


LAPORAN TUGAS KECIL 3
IF2211 STRATEGI ALGORITMA 2024/2025

**PUZZLE RUSH HOUR SOLVER MENGGUNAKAN ALGORITMA
PATHFINDING**

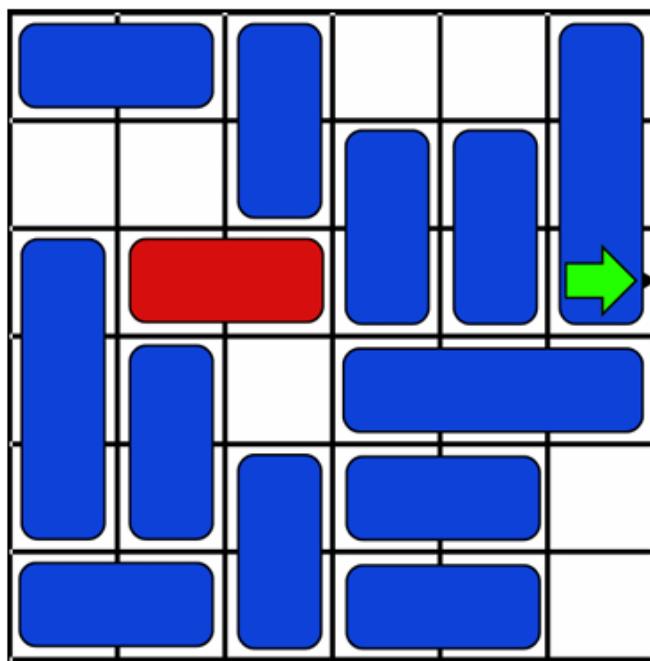


Oleh :
Muhammad Edo Raduputu Aprima
13523096

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

I. DESKRIPSI MASALAH

Rush Hour adalah sebuah permainan puzzle logika berbasis grid yang menantang pemain untuk menggeser kendaraan di dalam sebuah kotak (biasanya berukuran 6x6) agar mobil utama (biasanya berwarna merah) dapat keluar dari kemacetan melalui pintu keluar di sisi papan. Setiap kendaraan hanya bisa bergerak lurus ke depan atau ke belakang sesuai dengan IF2211 Strategi Algoritma – Tugas Kecil 3 1 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.

II. IMPLEMENTASI ALGORITMA

Terdapat beberapa algoritma pencarian yang digunakan dan diimplementasikan pada program ini. Setiap algoritma pencarian digunakan untuk menemukan urutan gerakan dari konfigurasi awal *board* menuju konfigurasi dimana *primary piece* dapat keluar dari *board*. Setiap konfigurasi unik dianggap sebagai sebuah node atau state dalam ruang pencarian. Setiap pergerakan mobil yang valid dari satu posisi ke posisi lainnya dianggap sebagai *edge* yang menghubungkan setiap node, dengan *cost* yang terkait dengan pergerakan tersebut.

• Algoritma Uniform Cost Search (UCS)

UCS adalah algoritma pencarian yang bertujuan untuk menemukan path dari node awal ke node tujuan dengan total cost kumulatif yang terendah. UCS mengeksplorasi node dengan memperluas node *n* yang belum diekspansi yang memiliki nilai *g(n)* terkecil, di mana *g(n)* adalah biaya path dari node awal ke *n*. UCS menggunakan priority queue untuk menyimpan node-node yang akan dieksplorasi atau disebut frontier, dengan prioritas ditentukan oleh *g(n)*. UCS menjamin penemuan path dengan *cost* terendah ke node tujuan, asalkan *cost* setiap langkah tidak negatif. Dalam program ini, setiap gerakan mobil memiliki *cost* = 1. Oleh karena itu, *g(n)* untuk sebuah state *n* adalah jumlah gerakan yang telah dilakukan untuk mencapai state tersebut. Dengan demikian, UCS akan selalu dijamin menemukan solusi dengan jumlah gerakan seminimal mungkin.

PSEUDOCODE:

```
FUNCTION UCS(initialBoard):
    INPUT: initialBoard (konfigurasi papan awal)
```

```

OUTPUT: path solusi (urutan gerakan) atau kegagalan

node = CREATE_NODE(state=initialBoard, gCost=0)
frontier = PRIORITY_QUEUE(ORDER_BY=gCost) // Priority queue
diurutkan berdasarkan gCost terendah
ADD node TO frontier
explored = EMPTY_SET() // Set untuk menyimpan state yang sudah
dieksplorasi

WHILE frontier IS NOT EMPTY:
    currentNode = REMOVE_LOWEST_GCOST_NODE_FROM(frontier)

    IF IS_GOAL_STATE(currentNode.state) THEN
        RETURN RECONSTRUCT_PATH(currentNode)
    ENDIF

    ADD currentNode.state TO explored

    FOR EACH validMove FROM GET_ALL_POSSIBLE_MOVES(currentNode.state):
        childState = GENERATE_NEW_BOARD_STATE(currentNode.state,
validMove)
        childNode = CREATE_NODE(state=childState, parent=currentNode,
move=validMove,
                           gCost=currentNode.gCost +
COST(validMove)) // COST(validMove) = 1

        IF childNode.state NOT IN explored AND childNode NOT IN
frontier_states:
            ADD childNode TO frontier
        ELSE IF childNode IN frontier_states AND childNode.gCost <
EXISTING_NODE_IN_FRONTIER(childNode.state).gCost:
            REPLACE_NODE_IN_FRONTIER(childNode) // Update dengan path yang
lebih baik
        ENDIF
    ENDFOR
ENDWHILE

RETURN FAILURE // Tidak ada solusi ditemukan
ENDFUNCTION

```

- **Algoritma Greedy Best First Search (GBFS)**

Greedy BFS adalah algoritma *informed search* atau pencarian heuristik. Greedy BFS menggunakan fungsi evaluasi $f(n)=h(n)$, di mana $h(n)$ adalah fungsi heuristik yang mengestimasi biaya dari node n ke node tujuan. Algoritma ini secara "rakus" memilih untuk mengekspansi node yang tampak paling dekat dengan tujuan berdasarkan nilai heuristik $h(n)$ terendah, tanpa mempertimbangkan biaya $g(n)$ yang telah dikeluarkan untuk mencapai node saat ini. Meskipun kadang dapat menemukan solusi dengan cepat, Greedy BFS tidak menjamin solusi optimal. Hal ini karena keputusan yang diambil hanya berdasarkan estimasi ke tujuan dapat

menjebaknya pada minimum lokal atau mengarahkannya melalui jalur yang panjang secara keseluruhan.

PSEUDOCODE:

```
FUNCTION GreedyBFS(initialBoard, heuristicFunction):
    INPUT: initialBoard, heuristicFunction (fungsi untuk menghitung h(n))
    OUTPUT: path solusi atau kegagalan

    node = CREATE_NODE(state=initialBoard, gCost=0) // gCost bisa diabaikan atau untuk info
    node.hCost = heuristicFunction(initialBoard)
    frontier = PRIORITY_QUEUE(ORDER_BY=hCost) // Diurutkan berdasarkan hCost terendah
    ADD node TO frontier
    explored = EMPTY_SET()

    WHILE frontier IS NOT EMPTY:
        currentNode = REMOVE_LOWEST_HCOST_NODE_FROM(frontier)

        IF IS_GOAL_STATE(currentNode.state) THEN
            RETURN RECONSTRUCT_PATH(currentNode)
        ENDIF

        ADD currentNode.state TO explored

        FOR EACH validMove FROM GET_ALL_POSSIBLE_MOVES(currentNode.state):
            childState = GENERATE_NEW_BOARD_STATE(currentNode.state,
            validMove)
            // gCost pada childNode bisa dihitung untuk informasi, tapi tidak digunakan untuk prioritas
            childNode = CREATE_NODE(state=childState, parent=currentNode,
            move=validMove,
                        gCost=currentNode.gCost + 1)
            childNode.hCost = heuristicFunction(childState)

            IF childNode.state NOT IN explored AND childNode NOT IN
            frontier_states:
                ADD childNode TO frontier
                // Greedy BFS biasanya tidak melakukan re-opening node dari explored atau updating di frontier,
                // karena hanya peduli dengan hCost. Namun, bisa ditambahkan jika diinginkan.
            ENDFOR
        ENDWHILE

        RETURN FAILURE
ENDFUNCTION
```

- **Algoritma A***

A* adalah algoritma pencarian terinformasi yang menggabungkan aspek dari UCS dan Greedy BFS. Sebagaimana dijelaskan dalam salindia kuliah (Bagian 2, hal. 6), A* mengevaluasi node berdasarkan fungsi evaluasi $f(n)=g(n)+h(n)$.

- $g(n)$ adalah biaya aktual dari node awal ke node n.
- $h(n)$ adalah estimasi biaya (nilai heuristik) dari node n ke node tujuan.
- $f(n)$ dengan demikian adalah estimasi total biaya path dari node awal ke node tujuan melalui node n. A* menggunakan *priority queue* untuk mengelola node-node yang akan dieksplorasi, dengan prioritas diberikan kepada node yang memiliki nilai $f(n)$ terkecil.

A* bertujuan menemukan solusi optimal dalam hal jumlah gerakan. Optimalitas ini dijamin jika fungsi heuristik $h(n)$ yang digunakan bersifat admissible. Dengan heuristik yang baik, A* cenderung lebih efisien (mengeksplorasi lebih sedikit node) daripada UCS karena panduan dari heuristik membantu memfokuskan pencarian.

PSEUDOCODE:

```

FUNCTION A_Star(initialBoard, heuristicFunction):
    INPUT: initialBoard, heuristicFunction
    OUTPUT: path solusi atau kegagalan

    node = CREATE_NODE(state=initialBoard, gCost=0)
    node.hCost = heuristicFunction(initialBoard)
    node.fCost = node.gCost + node.hCost
    frontier = PRIORITY_QUEUE(ORDER_BY=fCost) // Diurutkan berdasarkan
fCost terendah
    ADD node TO frontier
    explored = EMPTY_SET()

    WHILE frontier IS NOT EMPTY:
        currentNode = REMOVE_LOWEST_FCOST_NODE_FROM(frontier)

        IF IS_GOAL_STATE(currentNode.state) THEN
            RETURN RECONSTRUCT_PATH(currentNode)
        ENDIF

        ADD currentNode.state TO explored

        FOR EACH validMove FROM GET_ALL_POSSIBLE_MOVES(currentNode.state):
            childState = GENERATE_NEW_BOARD_STATE(currentNode.state,
validMove)
            childNode = CREATE_NODE(state=childState, parent=currentNode,
move=validMove,
                                gCost=currentNode.gCost + 1) //
COST(validMove) = 1
            childNode.hCost = heuristicFunction(childState)
            childNode.fCost = childNode.gCost + childNode.hCost

```

```

    IF childNode.state NOT IN explored AND childNode NOT IN
    frontier_states:
        ADD childNode TO frontier
    ELSE IF childNode IN frontier_states AND childNode.fCost <
    EXISTING_NODE_IN_FRONTIER(childNode.state).fCost:
        REPLACE_NODE_IN_FRONTIER(childNode) // Update dengan path yang
        lebih baik
    ENDIF
ENDFOR
ENDWHILE

RETURN FAILURE
ENDFUNCTION

```

- **Heuristik Blocking Piece**

Heuristik ini menghitung jumlah kendaraan (selain mobil utama) yang secara fisik berada pada jalur lurus antara posisi mobil utama saat ini dan pintu keluar yang dituju. Jika tidak ada kendaraan penghalang tetapi mobil utama belum mencapai pintu keluar, heuristik ini memberikan nilai 1 (mengindikasikan bahwa setidaknya satu gerakan lagi oleh mobil utama diperlukan untuk keluar). Logikanya adalah setiap kendaraan penghalang setidaknya memerlukan satu gerakan untuk disingkirkan atau dihindari.

PSEUDOCODE:

```

FUNCTION calculateBlockingPiecesHeuristic(board):
    INPUT: board (konfigurasi papan saat ini)
    OUTPUT: estimasi biaya heuristik (integer)

    primaryPiece = GET_PRIMARY_PIECE(board)
    IF primaryPiece IS NULL THEN
        RETURN MAX_INTEGER // Atau nilai error/sangat besar
    ENDIF

    IF IS_GOAL_STATE(board, primaryPiece) THEN
        RETURN 0
    ENDIF

    blockingCount = 0
    exitY = GET_EXIT_Y(board)
    exitX = GET_EXIT_X(board)

    IF primaryPiece.orientation IS HORIZONTAL THEN
        // Cek ke arah pintu keluar di baris yang sama
        IF exitX > primaryPiece.x THEN // Pintu keluar di kanan
            FOR col FROM (primaryPiece.x + primaryPiece.length) TO
            (GET_BOARD_COLS(board) - 1)
                IF board.cell[primaryPiece.y][col] IS OCCUPIED_BY_OTHER_PIECE
            THEN

```

```

        blockingCount = blockingCount + 1
    ENDIF
    // Berhenti jika sudah melewati koordinat pintu keluar (untuk
    kasus pintu keluar tidak di tepi)
    // Dalam Rush Hour, pintu keluar biasanya di tepi.
ENDFOR
ELSE // Pintu keluar di kiri (exitX < primaryPiece.x)
    FOR col FROM (primaryPiece.x - 1) DOWNTO 0
        IF board.cell[primaryPiece.y][col] IS OCCUPIED_BY_OTHER PIECE
THEN
        blockingCount = blockingCount + 1
ENDIF
ENDFOR
ENDIF
ELSE // primaryPiece.orientation IS VERTICAL
    // Cek ke arah pintu keluar di kolom yang sama
    IF exitY > primaryPiece.y THEN // Pintu keluar di bawah
        FOR row FROM (primaryPiece.y + primaryPiece.length) TO
(GET_BOARD_ROWS(board) - 1)
            IF board.cell[row][primaryPiece.x] IS OCCUPIED_BY_OTHER PIECE
THEN
            blockingCount = blockingCount + 1
ENDIF
ENDFOR
ELSE // Pintu keluar di atas (exitY < primaryPiece.y)
    FOR row FROM (primaryPiece.y - 1) DOWNTO 0
        IF board.cell[row][primaryPiece.x] IS OCCUPIED_BY_OTHER PIECE
THEN
            blockingCount = blockingCount + 1
ENDIF
ENDFOR
ENDIF
ENDIF

IF blockingCount IS 0 AND NOT IS_GOAL_STATE(board, primaryPiece)
THEN
    RETURN 1 // Perlu minimal 1 gerakan lagi oleh mobil utama
ELSE
    RETURN blockingCount
ENDIF
ENDFUNCTION

```

III. SOURCE PROGRAM

- Algoritma UCS

```

● ● ●

1  public Solution solveWithUCS(Board initialBoard) {
2    Long startTime = System.currentTimeMillis();
3    int nodesVisitedCount = 0;
4
5    PriorityQueue<SolverNode> openSet = new PriorityQueue<>(new SolverNode.UcsComparator());
6    Set<Board> closedSet = new HashSet<>();
7
8    int startHCost = initialBoard.calculateBlockingPiecesHeuristic();
9    SolverNode startNode = new SolverNode(initialBoard, startHCost);
10   openSet.add(startNode);
11
12  while (!openSet.isEmpty()) {
13    SolverNode currentNode = openSet.poll();
14    nodesVisitedCount++;
15
16    if (closedSet.contains(currentNode.getBoardState())) {
17      continue;
18    }
19    closedSet.add(currentNode.getBoardState());
20
21    if (currentNode.getBoardState().isGoalState()) {
22      Long endTime = System.currentTimeMillis();
23      List<Move> movesToSolution = currentNode.getMovesToSolution();
24      List<Board> pathToSolution = currentNode.getPathToSolution();
25      return new Solution(movesToSolution, pathToSolution, nodesVisitedCount, endTime - startTime, true);
26    }
27
28    List<Move> possibleMoves = currentNode.getBoardState().getAllPossibleMoves();
29    for (Move move : possibleMoves) {
30      Board nextBoardState = currentNode.getBoardState().generateNewBoardState(move);
31      if (!closedSet.contains(nextBoardState)) {
32        int newGCost = currentNode.getGCost() + 1;
33        int nextHCost = nextBoardState.calculateBlockingPiecesHeuristic();
34        SolverNode successorNode = new SolverNode(nextBoardState, currentNode, move, newGCost, nextHCost);
35        openSet.add(successorNode);
36      }
37    }
38  }
39  Long endTime = System.currentTimeMillis();
40  return new Solution(Collections.emptyList(), Collections.emptyList(), nodesVisitedCount, endTime - startTime, false);
41}

```

- **Algoritma GBFS**

```

1  public Solution solveWithGreedyBFS(Board initialBoard, String heuristicType) {
2      Long startTime = System.currentTimeMillis();
3      int nodesVisitedCount = 0;
4
5      PriorityQueue<SolverNode> openSet = new PriorityQueue<>(new SolverNode.GreedyBfsComparator());
6      Set<Board> closedSet = new HashSet<>();
7
8      // int startHCost = initialBoard.calculateBlockingPiecesHeuristic();
9      int startHCost;
10     if (heuristicType.equals("Blocking Pieces")) {
11         startHCost = initialBoard.calculateBlockingPiecesHeuristic();
12     } else {
13         startHCost = initialBoard.calculateManhattanDistanceHeuristic();
14     }
15
16     SolverNode startNode = new SolverNode(initialBoard, startHCost);
17     openSet.add(startNode);
18
19     while (!openSet.isEmpty()) {
20         SolverNode currentNode = openSet.poll();
21         nodesVisitedCount++;
22         // currentNode.getBoardState().printBoard();
23
24         if (closedSet.contains(currentNode.getBoardState())) {
25             continue;
26         }
27         closedSet.add(currentNode.getBoardState());
28
29         if (currentNode.getBoardState().isGoalState()) {
30             Long endTime = System.currentTimeMillis();
31             List<Move> movesToSolution = currentNode.getMovesToSolution();
32             List<Board> pathToSolution = currentNode.getPathToSolution();
33             System.out.println("GBFS menemukan solusi dengan gCost (langkah aktual): " + currentNode.getGCost());
34             return new Solution(movesToSolution, pathToSolution, nodesVisitedCount, endTime - startTime, true);
35         }
36
37         List<Move> possibleMoves = currentNode.getBoardState().getAllPossibleMoves();
38         for (Move move : possibleMoves) {
39             Board nextBoardState = currentNode.getBoardState().generateNewBoardState(move);
40
41             if (!closedSet.contains(nextBoardState)) {
42                 int newGCost = currentNode.getGCost() + 1;
43                 int nextHCost;
44                 if (heuristicType.equals("Blocking Pieces")) {
45                     nextHCost = nextBoardState.calculateBlockingPiecesHeuristic();
46                 } else {
47                     nextHCost = nextBoardState.calculateManhattanDistanceHeuristic();
48                 }
49                 SolverNode successorNode = new SolverNode(nextBoardState, currentNode, move, newGCost, nextHCost);
50                 openSet.add(successorNode);
51             }
52         }
53     }
54
55     Long endTime = System.currentTimeMillis();
56     return new Solution(Collections.emptyList(), Collections.emptyList(), nodesVisitedCount, endTime - startTime, false);
57 }

```

- **Algoritma A***

```

● ● ●

1  public Solution solveWithAStar(Board initialBoard, String heuristicType) {
2      Long startTime = System.currentTimeMillis();
3      int nodesVisitedCount = 0;
4
5      PriorityQueue<SolverNode> openSet = new PriorityQueue<>(new SolverNode.AstarComparator());
6      Set<Board> closedSet = new HashSet<>();
7
8      int startHCost;
9      if (heuristicType.equals("Blocking Pieces")) {
10          startHCost = initialBoard.calculateBlockingPiecesHeuristic();
11      } else {
12          startHCost = initialBoard.calculateManhattanDistanceHeuristic();
13      }
14      SolverNode startNode = new SolverNode(initialBoard, startHCost);
15      openSet.add(startNode);
16      // if (usingOpenSetMap) openSetMap.put(initialBoard, startNode);
17
18      while (!openSet.isEmpty()) {
19          SolverNode currentNode = openSet.poll();
20          nodesVisitedCount++;
21
22          if (closedSet.contains(currentNode.getBoardState())) {
23              continue;
24          }
25          closedSet.add(currentNode.getBoardState());
26
27          if (currentNode.getBoardState().isGoalState()) {
28              Long endTime = System.currentTimeMillis();
29              return new Solution(currentNode.getMovesToSolution(), currentNode.getPathToSolution(), nodesVisitedCount, endTime - startTime, true);
30          }
31
32          List<Move> possibleMoves = currentNode.getBoardState().getAllPossibleMoves();
33          for (Move move : possibleMoves) {
34              Board nextBoardState = currentNode.getBoardState().generateNewBoardState(move);
35
36              if (closedSet.contains(nextBoardState)) {
37                  continue;
38              }
39
40              int newGCost = currentNode.getGCost() + 1;
41              int nextHCost;
42              if (heuristicType.equals("Blocking Pieces")) {
43                  nextHCost = nextBoardState.calculateBlockingPiecesHeuristic();
44              } else {
45                  nextHCost = nextBoardState.calculateManhattanDistanceHeuristic();
46              }
47              SolverNode successorNode = new SolverNode(nextBoardState, currentNode, move, newGCost, nextHCost);
48
49              openSet.add(successorNode);
50          }
51      }
52      Long endTime = System.currentTimeMillis();
53      return new Solution(Collections.emptyList(), Collections.emptyList(), nodesVisitedCount, endTime - startTime, false);
54  }

```

- **Algoritma IDA***

```

● ● ●
1  public Solution solveWithIDAStar(Board initialBoard, String heuristicType) {
2      Long starttime = System.currentTimeMillis();
3      idaNodesVisitedTotal = 0;
4
5      // Hitung h-cost awal
6      int initialHCost;
7      if ("Manhattan".equals(heuristicType)) {
8          initialHCost = initialBoard.calculateManhattanDistanceHeuristic();
9      } else {
10         initialHCost = initialBoard.calculateBlockingPiecesHeuristic();
11     }
12
13     int fLimit = initialHCost;
14     SolverNode startNode = new SolverNode(initialBoard, null, null, 0, initialHCost);
15
16     final int MAX_IDA_ITERATIONS = 100;
17     final int MAX_NODES_VISITED_IDA = 7000000;
18
19     for (int iteration = 0; iteration < MAX_IDA_ITERATIONS; iteration++) {
20         System.out.println("IDA*: Iterasi " + (iteration + 1) + ", fLimit = " + fLimit + ", Total Nodes Sejauh Ini: " + idaNodesVisitedTotal);
21         Set<Board> pathCurrentlyExploring = new HashSet<>();
22         Object[] searchResult = idaSearch(startNode, fLimit, heuristicType, pathCurrentlyExploring);
23         SolverNode solutionNode = (SolverNode) searchResult[0];
24         int nextFLimitCandidate = (Integer) searchResult[1];
25
26         if (solutionNode == null) {
27             long endTime = System.currentTimeMillis();
28             List<Move> movesToSolution = solutionNode.getMovesToSolution();
29             List<Board> pathToSolution = solutionNode.getPathToSolution();
30             System.out.println("IDA* menemukan solusi dengan fCost = " + solutionNode.getfCost() + " (g=" + solutionNode.getgCost() + ", h=" + solutionNode.gethCost() + "), Total Nodes: " + idaNodesVisitedTotal);
31             return new Solution(movesToSolution, pathToSolution, idaNodesVisitedTotal, endTime, true);
32         }
33
34         if (nextFLimitCandidate == Integer.MAX_VALUE) {
35             System.out.println("IDA*: Tidak ada node lagi yang bisa dieksplorasi (nextFLimit adalah MAX_VALUE). Tidak ada solusi.");
36             break;
37         }
38         fLimit = nextFLimitCandidate;
39         if (idaNodesVisitedTotal >= MAX_NODES_VISITED_IDA) {
40             System.out.println("IDA*: Batas total node (" + MAX_NODES_VISITED_IDA + ") tercapai.");
41             break;
42         }
43     }
44
45     long endTime = System.currentTimeMillis();
46     System.out.println("IDA*: Gagal menemukan solusi setelah semua iterasi atau batas tercapai. Total Nodes: " + idaNodesVisitedTotal);
47     return new Solution(Collections.emptyList(), Collections.emptyList(), idaNodesVisitedTotal, endTime, false);
48 }

```

```

1  private Object[] ida_search(SolverNode currentNode, int fLimit, String heuristicType, Set<Board> pathCurrentlyExploring) {
2      idaNodesVisitedTotal++;
3      Board currentBoardState = currentNode.getBoardState();
4      int currentFCost = currentNode.getFCost();
5
6      if (currentFCost > fLimit) {
7          return new Object[]{null, currentFCost};
8      }
9
10     if (currentBoardState.isGoalState()) {
11         return new Object[]{currentNode, currentFCost};
12     }
13
14     if (pathCurrentlyExploring.contains(currentBoardState)) {
15         return new Object[]{null, Integer.MAX_VALUE};
16     }
17     pathCurrentlyExploring.add(currentBoardState);
18
19     int minNextFLimit = Integer.MAX_VALUE;
20     List<Move> possibleMoves = currentBoardState.getAllPossibleMoves();
21
22     for (Move move : possibleMoves) {
23         Board nextBoardState = currentBoardState.generateNewBoardState(move);
24         int gCostSuccessor = currentNode.getGCost() + 1;
25         int hCostSuccessor;
26
27         if ("Manhattan Distance".equals(heuristicType)) {
28             hCostSuccessor = nextBoardState.calculateManhattanDistanceHeuristic();
29         } else {
30             hCostSuccessor = nextBoardState.calculateBlockingPiecesHeuristic();
31         }
32
33         SolverNode successorNode = new SolverNode(nextBoardState, currentNode, move, gCostSuccessor, hCostSuccessor);
34
35         Object[] searchResult = ida_search(successorNode, fLimit, heuristicType, pathCurrentlyExploring);
36         SolverNode foundSolution = (SolverNode) searchResult[0];
37         int potentialNextFLimit = (Integer) searchResult[1];
38
39         if (foundSolution != null) {
40             pathCurrentlyExploring.remove(currentBoardState);
41             return new Object[]{foundSolution, 0};
42         }
43
44         if (potentialNextFLimit < minNextFLimit) {
45             minNextFLimit = potentialNextFLimit;
46         }
47     }
48
49     pathCurrentlyExploring.remove(currentBoardState);
50     return new Object[]{null, minNextFLimit};
51 }

```

- Algoritma IDS

```

● ● ●

1  public Solution solveWithIDS(Board initialBoard) {
2      Long startTime = System.currentTimeMillis();
3      idsNodesVisitedTotal = 0;
4      SolverNode startNode = new SolverNode(initialBoard, 0);
5      for (int depthLimit = 0; ; depthLimit++) {
6          System.out.println("IDS: Mencoba dengan depthLimit = " + depthLimit);
7          Set<Board> visitedInCurrentDls = new HashSet<>();
8
9          SolverNode solutionNode = dls(startNode, depthLimit, 0, visitedInCurrentDls);
10
11         if (solutionNode != null) {
12             Long endTime = System.currentTimeMillis();
13             List<Move> movesToSolution = solutionNode.getMovesToSolution();
14             List<Board> pathToSolution = solutionNode.getPathToSolution();
15             System.out.println("IDS menemukan solusi pada kedalaman: " + solutionNode.getGCost());
16             return new Solution(movesToSolution, pathToSolution, idsNodesVisitedTotal, endTime - startTime, true);
17         }
18
19         if (idsNodesVisitedTotal > 2000000 && depthLimit > 30) { // Batas pengaman sementara
20             System.out.println("IDS: Melebihi batas iterasi/node, kemungkinan tidak ada solusi atau solusi sangat dalam.");
21             break;
22         }
23     }
24     Long endTime = System.currentTimeMillis();
25     return new Solution(Collections.emptyList(), Collections.emptyList(), idsNodesVisitedTotal, endTime - startTime, false);
26 }
27
28 private SolverNode dls(SolverNode currentNode, int depthLimit, int currentDepth, Set<Board> visitedInCurrentDls) {
29     idsNodesVisitedTotal++;
30
31     Board currentBoardState = currentNode.getBoardState();
32
33     if (currentBoardState.isGoalState()) {
34         return currentNode;
35     }
36
37     if (currentDepth >= depthLimit) {
38         return null;
39     }
40
41     if (visitedInCurrentDls.contains(currentBoardState)) {
42         return null;
43     }
44     visitedInCurrentDls.add(currentBoardState);
45
46     List<Move> possibleMoves = currentBoardState.getAllPossibleMoves();
47
48     for (Move move : possibleMoves) {
49         Board nextBoardState = currentBoardState.generateNewBoardState(move);
50         SolverNode successorNode = new SolverNode(nextBoardState, currentNode, move, currentDepth + 1, 0);
51
52         SolverNode resultNode = dls(successorNode, depthLimit, currentDepth + 1, visitedInCurrentDls);
53
54         if (resultNode != null) {
55             visitedInCurrentDls.remove(currentBoardState);
56             return resultNode;
57         }
58     }
59
60     visitedInCurrentDls.remove(currentBoardState);
61     return null;
62 }

```

- Heuristik Blocking Piece

```

1  public int calculateBlockingPiecesHeuristic() {
2      Piece primaryPiece = this.getPieceById('P');
3      if (primaryPiece == null) {
4          System.err.println("Error di heuristik: Primary piece 'P' tidak ditemukan!");
5          return Integer.MAX_VALUE;
6      }
7
8      if (this.isGoalState()) {
9          return 0;
10     }
11
12     Set<Character> blockingPieces = new HashSet<>();
13     char[][] grid = this.gridRepresentation;
14
15     if (primaryPiece.getOrientation() == Orientation.HORIZONTAL) {
16         if (this.exitX == 0) { // Pintu keluar di kiri
17             for (int c = primaryPiece.getX() - 1; c >= 0; c--) {
18                 if (grid[primaryPiece.getY()][c] != '.' && grid[primaryPiece.getY()][c] != primaryPiece.getId()) {
19                     blockingPieces.add(grid[primaryPiece.getY()][c]);
20                 }
21             }
22         } else { // Pintu keluar di kanan (this.exitX == this.cols - 1)
23             for (int c = primaryPiece.getX() + primaryPiece.getLength(); c < this.cols; c++) {
24                 if (grid[primaryPiece.getY()][c] != '.' && grid[primaryPiece.getY()][c] != primaryPiece.getId()) {
25                     blockingPieces.add(grid[primaryPiece.getY()][c]);
26                 }
27             }
28         }
29     } else { // Primary Piece VERTIKAL
30         if (this.exitY == 0) { // Pintu keluar di atas
31             for (int r = primaryPiece.getY() - 1; r >= 0; r--) {
32                 if (grid[r][primaryPiece.getX()] != '.' && grid[r][primaryPiece.getX()] != primaryPiece.getId()) {
33                     blockingPieces.add(grid[r][primaryPiece.getX()]);
34                 }
35             }
36         } else { // Pintu keluar di bawah (this.exitY == this.rows - 1)
37             for (int r = primaryPiece.getY() + primaryPiece.getLength(); r < this.rows; r++) {
38                 if (grid[r][primaryPiece.getX()] != '.' && grid[r][primaryPiece.getX()] != primaryPiece.getId()) {
39                     blockingPieces.add(grid[r][primaryPiece.getX()]);
40                 }
41             }
42         }
43     }
44     int hValue = blockingPieces.size();
45     if (hValue == 0 && !this.isGoalState()) {
46         hValue = 1;
47     }
48     return hValue;
49 }

```

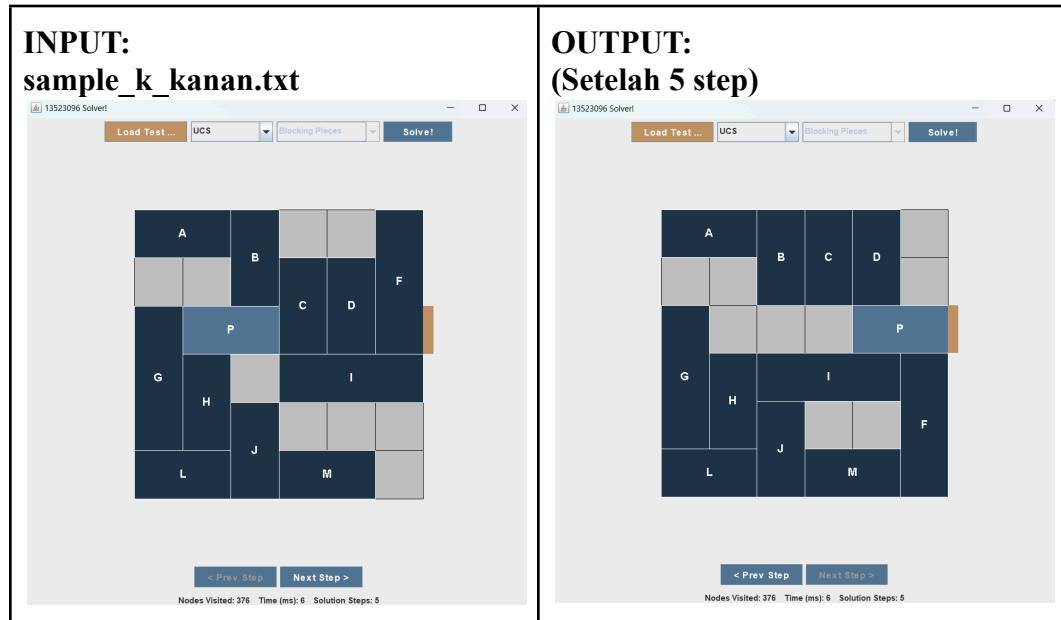
- Heuristik Manhattan Distance

```
1  public int calculateManhattanDistanceHeuristic() {
2      Piece primaryPiece = this.getPieceById('P');
3
4      if (primaryPiece == null) {
5          System.err.println("Error di Manhattan Distance heuristic: Primary piece 'P' tidak ditemukan!");
6          return Integer.MAX_VALUE;
7      }
8
9      if (this.isGoalState()) {
10
11         int distance = 0;
12
13         if (primaryPiece.getOrientation() == Orientation.HORIZONTAL) {
14             if (this.exitX > (primaryPiece.getX() + primaryPiece.getLength() - 1)) {
15                 distance = this.exitX - (primaryPiece.getX() + primaryPiece.getLength() - 1);
16             }
17             else if (this.exitX < primaryPiece.getX()) {
18                 distance = primaryPiece.getX() - this.exitX;
19             }
20             else if (this.exitX == 0) { // Pintu keluar di paling kiri
21                 distance = primaryPiece.getX();
22             } else if (this.exitX == this.cols - 1) { // Pintu keluar di paling kanan
23                 distance = this.exitX - (primaryPiece.getX() + primaryPiece.getLength() - 1);
24             } else {
25                 if(this.exitX >= primaryPiece.getX() + primaryPiece.getLength() - 1){
26                     distance = this.exitX - (primaryPiece.getX() + primaryPiece.getLength() - 1);
27                 }
28                 else if (this.exitX <= primaryPiece.getX()){
29                     distance = primaryPiece.getX() - this.exitX;
30                 } else {
31                     distance = 0;
32                 }
33             }
34         }
35
36     } else { // Primary piece is VERTICAL
37         if (this.exitY > (primaryPiece.getY() + primaryPiece.getLength() - 1)) {
38             distance = this.exitY - (primaryPiece.getY() + primaryPiece.getLength() - 1);
39         }
40         else if (this.exitY < primaryPiece.getY()) {
41             distance = primaryPiece.getY() - this.exitY;
42         }
43         else if (this.exitY == 0) { // Pintu keluar di paling atas
44             distance = primaryPiece.getY();
45         } else if (this.exitY == this.rows - 1) { // Pintu keluar di paling bawah
46             distance = this.exitY - (primaryPiece.getY() + primaryPiece.getLength() - 1);
47         } else {
48             if(this.exitY >= primaryPiece.getY() + primaryPiece.getLength() - 1){
49                 distance = this.exitY - (primaryPiece.getY() + primaryPiece.getLength() - 1);
50             }
51             else if (this.exitY <= primaryPiece.getY()){
52                 distance = primaryPiece.getY() - this.exitY;
53             } else {
54                 distance = 0;
55             }
56         }
57     }
58     return Math.max(0, distance);
59 }
```

IV. UJI TEST CASE

Untuk output dari setiap test case, karena mungkin bisa menghasilkan banyak step, hanya dilampirkan bagian akhirnya saja, untuk step lengkapnya dapat dilakukan dengan klik “Next Step” pada GUI program.

- **Algoritma UCS**
TEST CASE 1

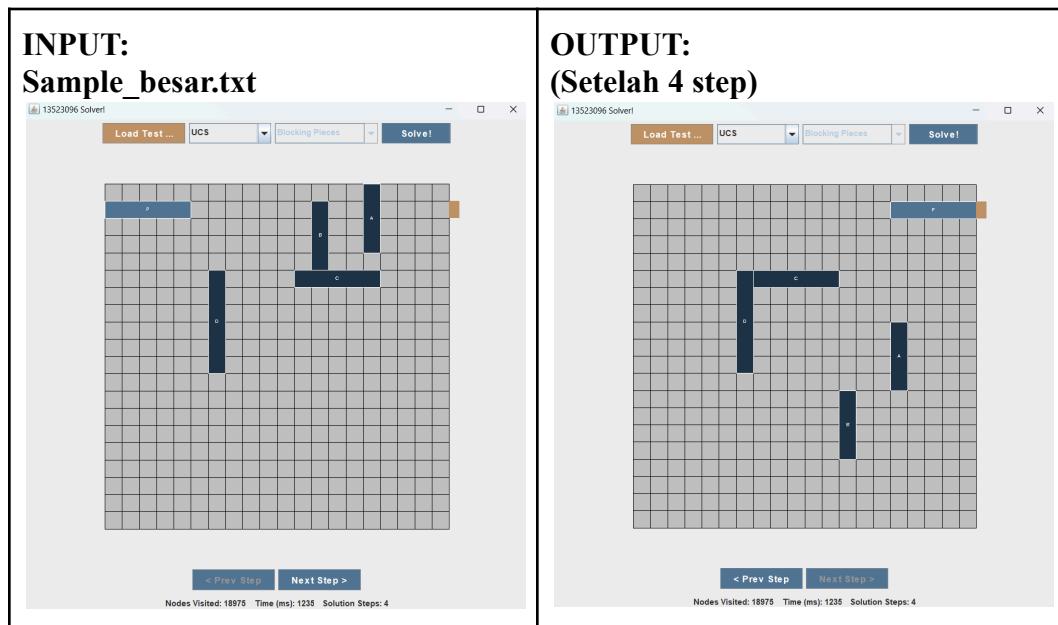


TEST CASE 2

INPUT: sample_k_bawah.txt	OUTPUT: (Setelah 4 step)
------------------------------	-----------------------------

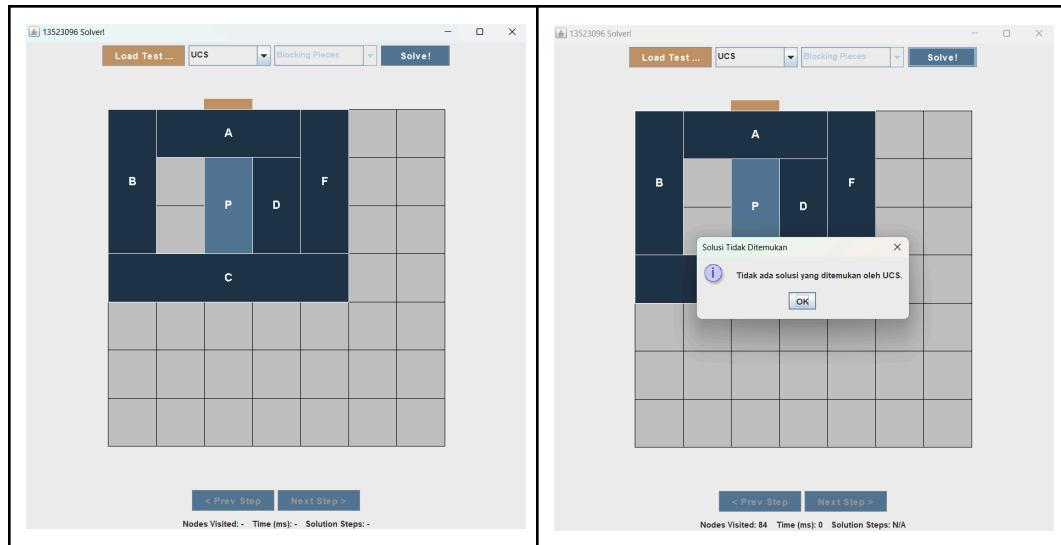


TEST CASE 3



TEST CASE 4

<p>INPUT: Sample_no_solution.txt</p>	<p>OUTPUT: (Tidak ada solusi)</p>
---	--



- **Algoritma GBFS**
TEST CASE 1

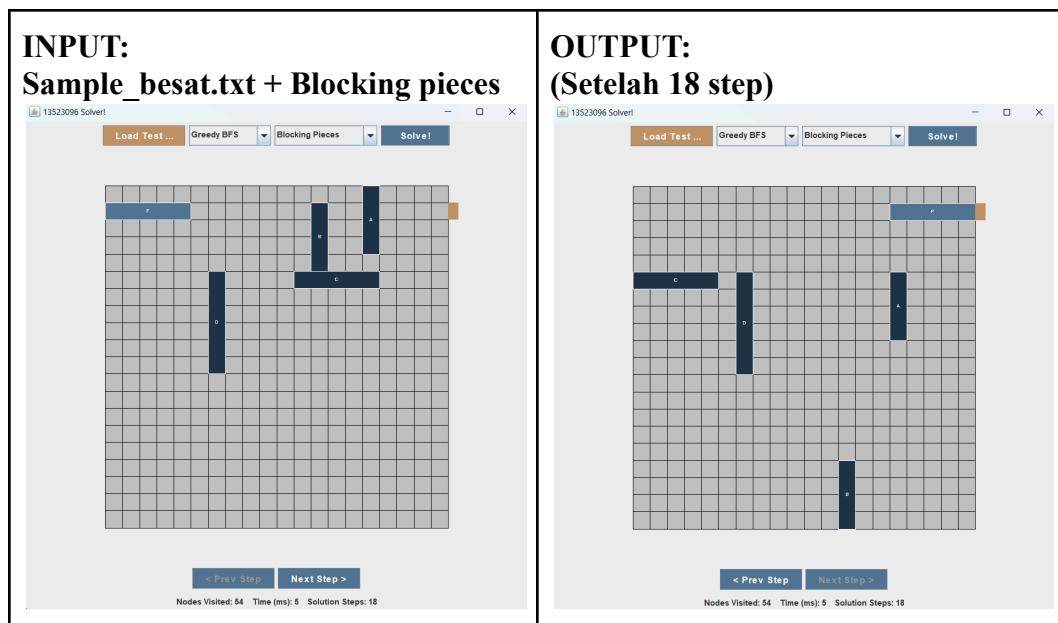
<p>INPUT: Sample_k_kanan.txt + Manhattan Distance</p> <p>< Prev Step Next Step ></p> <p>Nodes Visited: - Time (ms): - Solution Steps: -</p>	<p>OUTPUT: (Setelah 56 step)</p> <p>< Prev Step Next Step ></p> <p>Nodes Visited: 392 Time (ms): 6 Solution Steps: 56</p>
--	--

TEST CASE 2

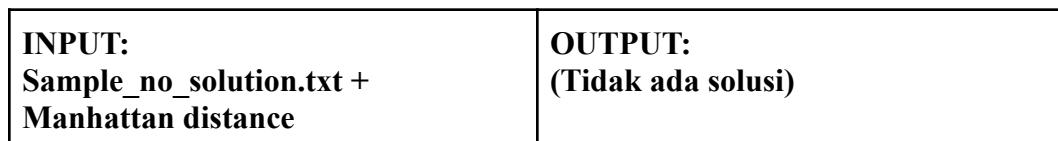
<p>INPUT: Sample_k_bawah.txt + Blocking Pieces</p>	<p>OUTPUT: (Setelah 4 step)</p>
---	--

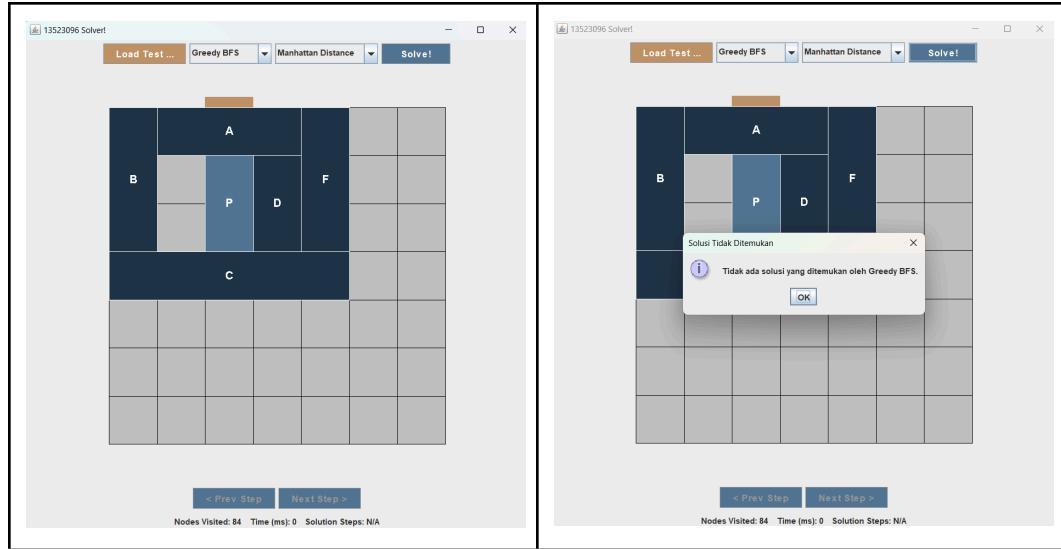


TEST CASE 3



TEST CASE 4





- **Algoritma A***
TEST CASE 1

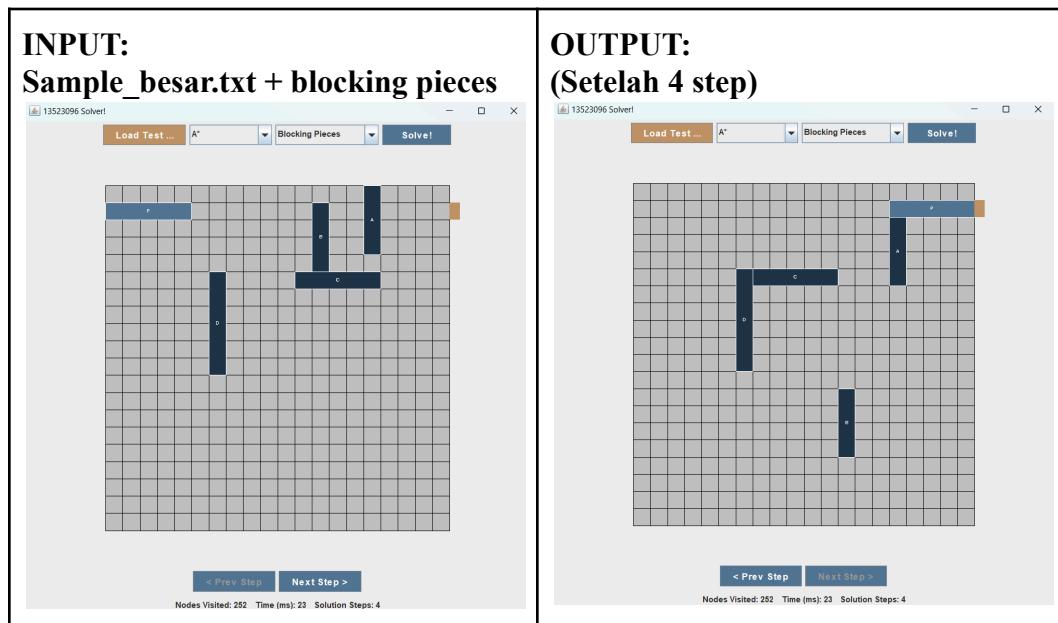
<p>INPUT: Sample_k_kanan.txt + blocking pieces</p> <p>< Prev Step Next Step ></p> <p>Nodes Visited: 34 Time (ms): 2 Solution Steps: 5</p>	<p>OUTPUT: (Setelah 5 step)</p> <p>< Prev Step Next Step ></p> <p>Nodes Visited: 34 Time (ms): 2 Solution Steps: 5</p>
--	---

TEST CASE 2

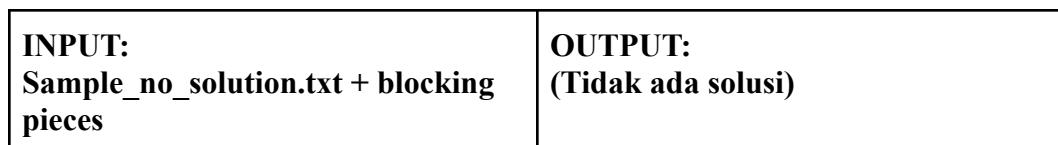
<p>INPUT: Sample_k_bawah.txt + Manhattan distance</p>	<p>OUTPUT: (Setelah 5 step)</p>
--	--

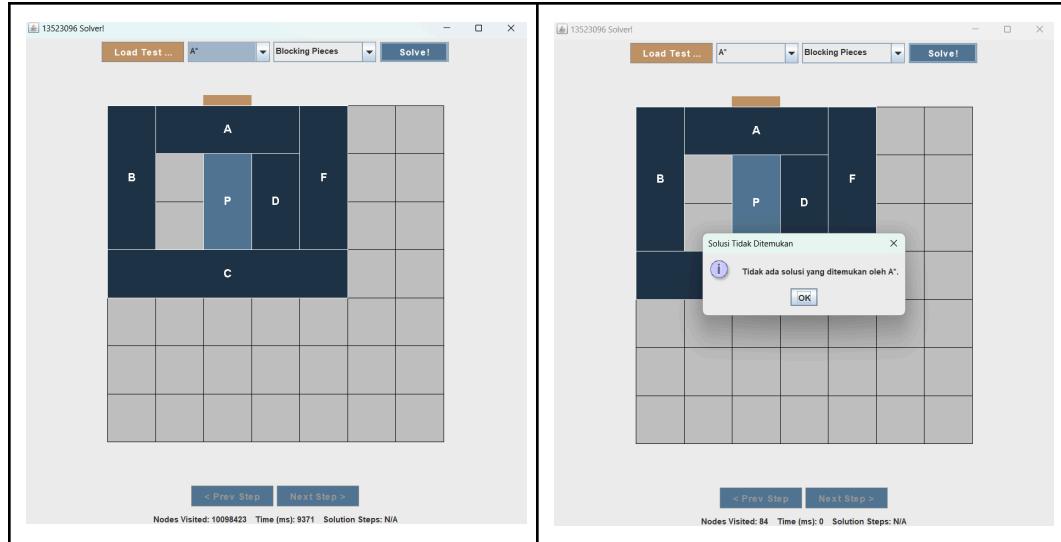


TEST CASE 3



TEST CASE 4





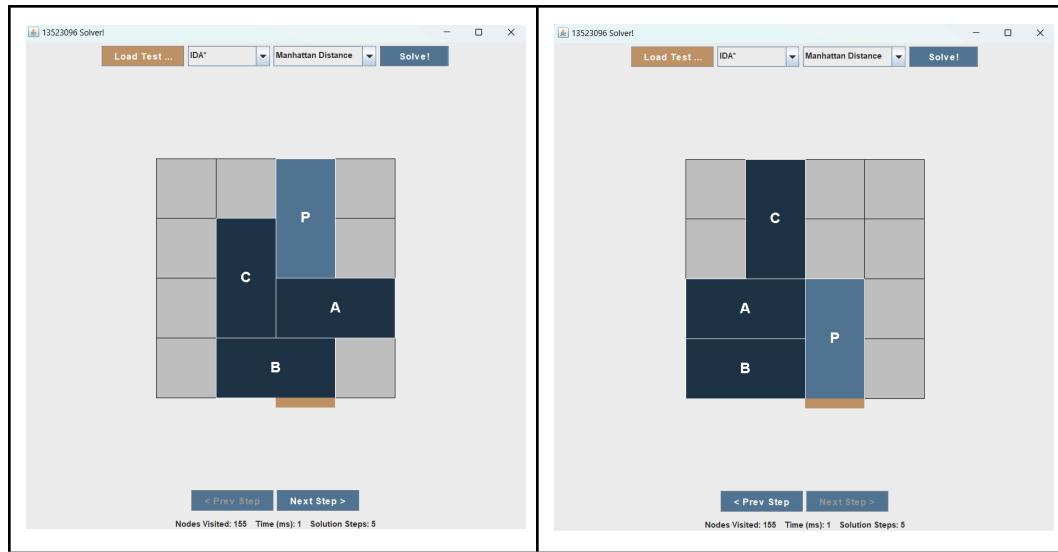
- **Algoritma IDA***

TEST CASE 1

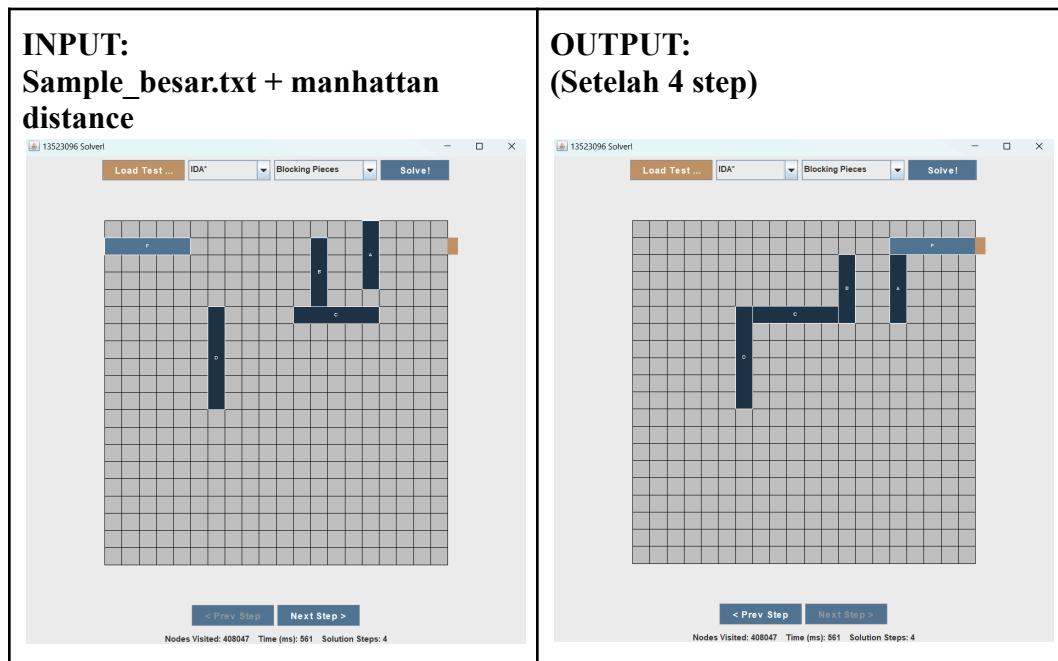
<p>INPUT: Sample_k_kanan.txt + blocking pieces</p> <p>< Prev Step Next Step ></p> <p>Nodes Visited: 10098423 Time (ms): 9371 Solution Steps: N/A</p>	<p>OUTPUT: (Setelah 5 step)</p> <p>< Prev Step Next Step ></p> <p>Nodes Visited: 1040 Time (ms): 4 Solution Steps: 5</p>
---	---

TEST CASE 2

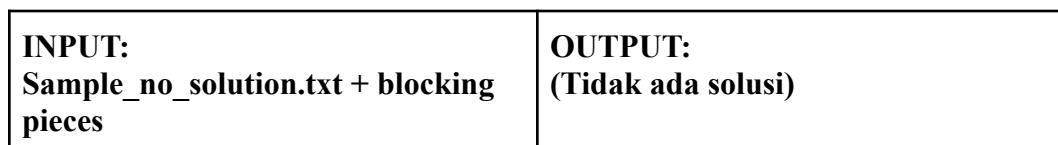
<p>INPUT: Sample_k_bawah.txt + Manhattan distance</p>	<p>OUTPUT: (Setelah 5 step)</p>
--	--

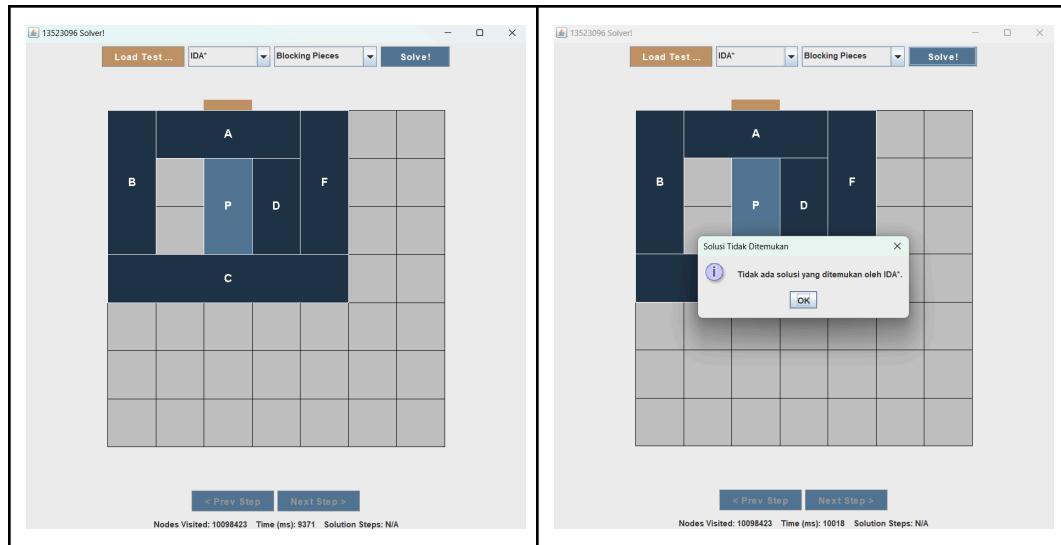


TEST CASE 3

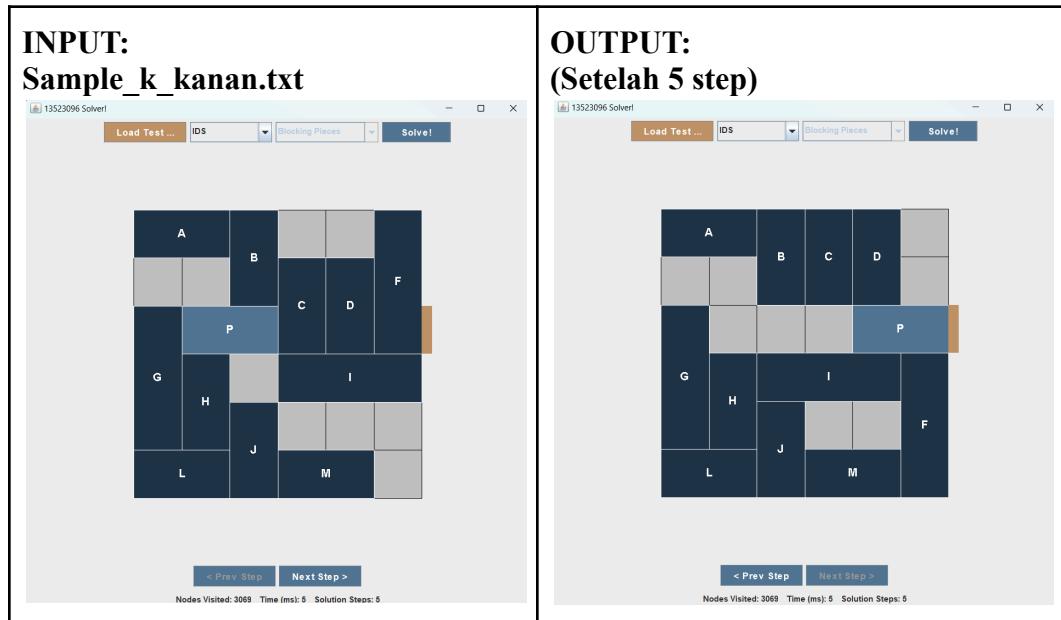


TEST CASE 4





- **Algoritma IDS**
TEST CASE 1

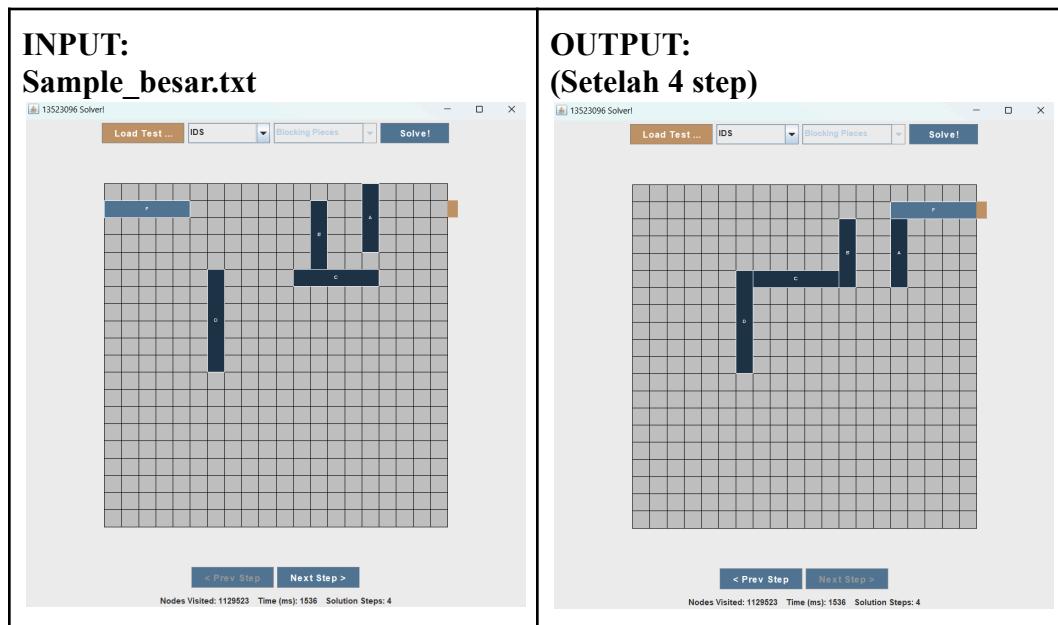


TEST CASE 2

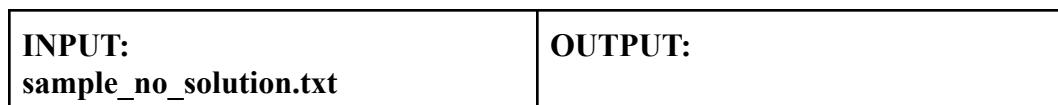
INPUT: sample_k_bawah.txt	OUTPUT: (Setelah 4 step)
--	---



TEST CASE 3



TEST CASE 4





V. ANALISIS

- **Definisi dari $f(n)$ dan $g(n)$.**
 - $g(n)$: Merupakan biaya aktual dari path yang telah ditemukan dari node awal ke node n. Dalam konteks Rush Hour, di mana setiap pergeseran mobil dihitung sebagai 1 unit biaya, $g(n)$ adalah jumlah langkah atau gerakan mobil yang telah dilakukan untuk mencapai konfigurasi papan (node) n dari konfigurasi awal.
 - $f(n)$: Ini adalah fungsi evaluasi yang digunakan oleh algoritma A*, didefinisikan sebagai $f(n)=g(n)+h(n)$. Fungsi ini merepresentasikan estimasi total biaya dari path solusi yang melalui node n.
- **Apakah heuristik yang digunakan pada algoritma A* admissible? Jelaskan sesuai definisi admissible dari salindia kuliah.**

Sebuah fungsi heuristik $h(n)$ dikatakan admissible jika untuk setiap node n, nilai $h(n)$ tidak pernah melebih-lebihkan biaya aktual ($h^*(n)$) untuk mencapai node tujuan dari node n ($h(n) \leq h^*(n)$). Heuristik yang *admissible* bersifat optimistik dan merupakan syarat penting agar A* dapat menjamin penemuan solusi optimal.
- **Heuristik Jumlah Mobil Penghalang (Blocking Pieces Heuristic):** Heuristik ini admissible.

Heuristik ini admissible. Setiap mobil yang secara langsung menghalangi jalur mobil utama pasti memerlukan setidaknya satu gerakan untuk disingkirkan. Oleh karena itu, jumlah mobil penghalang tidak akan pernah lebih besar dari jumlah gerakan minimum aktual yang diperlukan untuk membersihkan jalur tersebut. Jika tidak ada penghalang dan mobil utama belum di tujuan (saat $h(n)=1$), ini juga

tidak melebih-lebihkan karena minimal satu gerakan lagi oleh mobil utama pasti diperlukan.

- **Heuristik Jarak Manhattan (*Manhattan Distance Heuristic*):** Heuristik ini **admissible**.

Heuristik ini admissible. Jarak Manhattan yang dihitung adalah jumlah gerakan minimum absolut yang harus dilakukan oleh mobil utama sendiri jika tidak ada penghalang. Karena heuristik ini tidak memperhitungkan gerakan mobil lain yang mungkin diperlukan untuk membersihkan jalur atau gerakan tambahan mobil utama untuk bermanuver, ia tidak akan pernah melebih-lebihkan total gerakan sebenarnya yang dibutuhkan. Akan tetapi dalam konteks permainan ini, karena pada kondisi lain satu pergerakan piece bisa langsung dalam beberapa cell, maka untuk heuristik ini yang menyebabkan primary piece maju setiap ada kosong di depannya membuat dia melakukan lebih banyak pergerakan dibandingkan sekali pergerakan diakhir.

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

Ya, dalam penyelesaian Rush Hour, algoritma UCS berperilaku identik dengan Breadth-First Search (BFS). BFS dan IDS menemukan path dengan jumlah langkah paling sedikit. UCS juga bertujuan menemukan path dengan biaya terendah. Karena setiap gerakan dalam Rush Hour memiliki biaya seragam (yaitu 1), maka biaya kumulatif $g(n)$ dalam UCS setara dengan kedalaman (jumlah langkah) node n dari node awal. Mengingat BFS mengeksplorasi node level demi level berdasarkan kedalaman dan UCS mengeksplorasi node berdasarkan $g(n)$ terkecil (yang dalam kasus ini sama dengan kedalaman), kedua algoritma akan mengeksplorasi node dalam urutan yang sama dan menghasilkan path solusi dengan jumlah gerakan minimal yang sama.

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

Secara teoritis, ya, algoritma A* berpotensi lebih efisien dibandingkan UCS dalam penyelesaian Rush Hour, asalkan menggunakan heuristik yang *admissible* dan informatif. Efisiensi diukur dari jumlah node yang dieksplorasi. UCS akan mengeksplorasi semua node pada suatu kedalaman sebelum melanjutkan ke kedalaman berikutnya. A*, dengan fungsi $f(n)=g(n)+h(n)$, menggunakan informasi heuristik $h(n)$ untuk memprioritaskan node yang lebih menjanjikan, sehingga dapat lebih terarah dan menghindari eksplorasi cabang yang kurang relevan. Jika $h(n)=0$, A* menjadi UCS. Namun, dengan heuristik yang baik, A*

cenderung mengeksplorasi lebih sedikit node. Ide dasar A* adalah untuk menghindari perluasan path yang sudah mahal.

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

Secara teoritis, algoritma Greedy Best-First Search (Greedy BFS) tidak menjamin solusi optimal untuk penyelesaian Rush Hour. Greedy BFS dapat menemukan solusi yang bukan optimal. Penyebabnya adalah Greedy BFS hanya fokus pada nilai heuristik $h(n)$ (estimasi ke tujuan) dan mengabaikan biaya $g(n)$ (biaya dari awal). Keputusan "rakus" berdasarkan $h(n)$ bisa mengarah ke jalur yang panjang secara keseluruhan meskipun setiap langkah terlihat mendekati tujuan secara lokal. Greedy BFS juga memiliki masalah lain seperti ketidaklengkapan dan kemungkinan terjebak.

Selain analisis teoritis, dilakukan pula pengamatan terhadap kinerja implementasi algoritma-algoritma pathfinding pada beberapa kasus uji. Pengamatan ini mencakup aspek waktu eksekusi dan jumlah node yang dikunjungi, yang merupakan indikator penting dari efisiensi algoritma. Karena keterbatasan waktu untuk pengujian ekstensif pada semua kombinasi, analisis berikut lebih bersifat kualitatif berdasarkan sifat algoritma dan beberapa observasi umum yang diharapkan.

Secara umum, Uniform Cost Search (UCS), yang setara dengan BFS pada kasus ini, cenderung mengeksplorasi sejumlah besar node karena sifatnya yang *exhaustive* level demi level. Hal ini terutama terlihat pada kasus uji dengan ruang pencarian yang besar atau solusi yang cukup dalam, yang dapat mengakibatkan waktu eksekusi yang relatif lebih lama dibandingkan algoritma terinformasi. Namun, UCS selalu menjamin solusi optimal (jumlah langkah tersingkat).

Greedy Best-First Search (Greedy BFS), dengan kedua heuristik (*Blocking Pieces* dan *Manhattan Distance*), seringkali menemukan solusi (jika ada) dengan sangat cepat dan mengunjungi jumlah node yang jauh lebih sedikit dibandingkan UCS. Hal ini karena sifat "rakus"-nya yang langsung menuju ke state yang tampak paling menjanjikan. Akan tetapi, solusi yang dihasilkan oleh Greedy BFS tidak dijamin optimal. Pada beberapa kasus uji, terutama yang kompleks atau memiliki banyak "jebakan" heuristik, Greedy BFS bisa menghasilkan solusi dengan jumlah langkah yang lebih banyak atau bahkan gagal menemukan solusi jika terjebak pada minimum lokal (meskipun dalam implementasi ini, dengan *closed list*, ia seharusnya tidak terjebak dalam loop tak terbatas pada graf finite).

Algoritma A*, ketika digunakan dengan heuristik yang *admissible* (baik *Blocking Pieces* maupun *Manhattan Distance*), diharapkan menunjukkan kinerja yang baik. A* bertujuan

untuk menemukan solusi optimal seperti UCS, tetapi dengan panduan heuristik, ia cenderung lebih efisien dengan mengeksplorasi lebih sedikit node. Kualitas heuristik sangat mempengaruhi kinerjanya; heuristik yang lebih akurat (lebih mendekati biaya sebenarnya tanpa melebih-lebihkan) akan membuat A* lebih cepat. Diperkirakan A* dengan heuristik *Manhattan Distance* bisa sedikit lebih unggul dalam memangkas ruang pencarian pada kasus tertentu dibandingkan *Blocking Pieces* jika ia memberikan estimasi yang lebih bervariasi dan mendekati h^* .

Untuk algoritma alternatif, Iterative Deepening Search (IDS) juga menjamin solusi optimal. Secara teoretis, IDS lebih unggul dari BFS/UCS dalam hal penggunaan memori ($O(bd)$ vs $O(b^d)$). Namun, karena IDS melakukan iterasi ulang dari awal pada setiap peningkatan batas kedalaman, total waktu eksekusinya bisa lebih lama, terutama jika solusi berada pada kedalaman yang cukup besar, karena node-node di level atas akan sering dikunjungi ulang. Jumlah node yang dikunjungi oleh IDS bisa jadi lebih besar dari total node unik yang dieksplorasi oleh BFS/UCS karena perhitungan ulang ini.

Iterative Deepening A* (IDA*) diharapkan memberikan keseimbangan antara optimalitas A*, efisiensi memori IDS, dan panduan heuristik. Seperti IDS, ia mengunjungi ulang state, yang dapat mempengaruhi waktu eksekusi. Namun, dengan batas *f-cost* yang terus meningkat secara cerdas (berdasarkan nilai f minimum dari node yang terpotong), IDA* berusaha memfokuskan pencarinya mirip dengan A*. Dibandingkan A* standar, IDA* akan jauh lebih hemat memori pada masalah dengan ruang keadaan yang sangat besar. Performanya relatif terhadap A* akan bergantung pada kualitas heuristik dan struktur spesifik dari masalah. Dengan heuristik yang baik, IDA* bisa menjadi sangat efektif.

Dalam hal kompleksitas waktu teoretis untuk kasus terburuk, sebagian besar algoritma ini (UCS, A*, IDS, IDA*) dapat memiliki kompleksitas eksponensial ($O(bd)$ atau $O(bm)$) tergantung pada sifat graf dan heuristik. Namun, dalam praktiknya pada puzzle Rush Hour dengan heuristik yang layak, A* dan IDA* diharapkan dapat menyelesaikan banyak masalah dengan lebih efisien daripada pencarian lain pada masalah yang sama. Greedy BFS bisa sangat cepat tetapi tidak ada jaminan optimalitas.

VI. IMPLEMENTASI BONUS

Selain memenuhi spesifikasi wajib, program ini juga mengimplementasikan beberapa fitur bonus. Fitur-fitur bonus yang berhasil diimplementasikan adalah sebagai berikut:

- **Implementasi Graphical User Interface (GUI)**

Sebuah antarmuka pengguna grafis (GUI) telah dikembangkan menggunakan Java Swing untuk mempermudah interaksi pengguna dengan program. GUI ini memungkinkan pengguna untuk memuat file konfigurasi papan Rush Hour dalam format .txt melalui dialog pemilihan file, menampilkan papan permainan secara

visual, termasuk posisi semua piece (mobil) dan pintu keluar, memilih algoritma pathfinding yang akan digunakan (UCS, Greedy BFS, A*, IDS, IDA*) melalui sebuah dropdown menu, memilih fungsi heuristik yang akan digunakan (untuk algoritma GBFS, A*, dan IDA*) melalui dropdown menu kedua, memulai proses pencarian solusi dengan menekan tombol "Solve!", menampilkan statistik hasil pencarian, yaitu jumlah node yang dikunjungi, waktu eksekusi program dalam milidetik, dan jumlah langkah dalam solusi yang ditemukan.

Setelah solusi ditemukan, GUI menampilkan konfigurasi papan awal. Pengguna kemudian dapat menavigasi langkah-langkah solusi satu per satu (maju atau mundur) menggunakan tombol "Next Step >" dan "< Prev Step", di mana setiap langkah akan memperbarui tampilan visual papan. Ini memberikan visualisasi pergerakan piece dari awal hingga solusi tercapai. Program juga secara otomatis mencoba memuat file konfigurasi default (test/init_board.txt) saat pertama kali GUI. Implementasi GUI dilakukan dalam kelas GUI.java yang mengatur tata letak komponen, interaksi pengguna, dan pemanggilan solver. Tampilan visual papan dirender oleh kelas BoardPanel.java.

- **Implementasi Heuristik Alternatif**

Program ini mengimplementasikan dua fungsi heuristik yang berbeda, yang dapat dipilih oleh pengguna saat menggunakan algoritma pencarian terinformasi (Greedy BFS, A*, IDA*):

- **Heuristik Manhattan Distance**

Heuristik ini mengestimasi jumlah gerakan minimal yang dibutuhkan oleh mobil utama *saja* untuk mencapai posisi pintu keluar, dengan asumsi tidak ada kendaraan lain di papan. Jika mobil utama berorientasi horizontal, heuristik menghitung jarak (jumlah sel) antara tepi mobil utama yang relevan (yang akan menuju pintu keluar) dan kolom pintu keluar. Jika berorientasi vertikal, dihitung jarak antara tepi mobil utama yang relevan dan baris pintu keluar.

PSEUDOCODE:

```
FUNCTION calculateManhattanDistanceHeuristic(board):
    INPUT: board (konfigurasi papan saat ini)
    OUTPUT: estimasi biaya heuristik (integer)

    primaryPiece = GET_PRIMARY_PIECE(board)
    IF primaryPiece IS NULL THEN
        RETURN MAX_INTEGER
    ENDIF

    IF IS_GOAL_STATE(board, primaryPiece) THEN
        RETURN 0
    ENDIF
```

```

distance = 0
exitY = GET_EXIT_Y(board)
exitX = GET_EXIT_X(board)

IF primaryPiece.orientation IS HORIZONTAL THEN
    // Asumsi: primaryPiece.y == exitY
    IF exitX > (primaryPiece.x + primaryPiece.length - 1) THEN //
        Pintu keluar di kanan mobil
            distance = exitX - (primaryPiece.x + primaryPiece.length - 1)
        ELSE IF exitX < primaryPiece.x THEN // Pintu keluar di kiri mobil
            distance = primaryPiece.x - exitX
        ELSE // Mobil sudah sejajar atau melintasi kolom pintu keluar
            // Jika pintu keluar di tepi kiri
            IF exitX == 0 THEN distance = primaryPiece.x
            // Jika pintu keluar di tepi kanan
            ELSE IF exitX == (GET_BOARD_COLS(board) - 1) THEN
                distance = exitX - (primaryPiece.x + primaryPiece.length - 1)
            ELSE
                distance = 0 // Jika sudah di kolom exit dan bukan di tepi,
                anggap jarak 0 untuk mobil itu sendiri
            ENDIF
        ENDIF
    ELSE // primaryPiece.orientation IS VERTICAL
        // Asumsi: primaryPiece.x == exitX
        IF exitY > (primaryPiece.y + primaryPiece.length - 1) THEN //
            Pintu keluar di bawah mobil
                distance = exitY - (primaryPiece.y + primaryPiece.length - 1)
            ELSE IF exitY < primaryPiece.y THEN // Pintu keluar di atas mobil
                distance = primaryPiece.y - exitY
            ELSE
                IF exitY == 0 THEN distance = primaryPiece.y
                ELSE IF exitY == (GET_BOARD_ROWS(board) - 1) THEN
                    distance = exitY - (primaryPiece.y + primaryPiece.length - 1)
                ELSE
                    distance = 0
                ENDIF
            ENDIF
        ENDIF
    ENDIF

    RETURN MAX(0, distance) // Pastikan tidak negatif
ENDFUNCTION

```

- **Implementasi Algoritma Pathfinding Alternatif**

- **Algoritma IDA***

Iterative Deepening A* (IDA*) adalah varian dari A* yang menggunakan pendekatan *iterative deepening* untuk efisiensi memori. IDA* tidak secara eksplisit dibahas dalam salindia yang diberikan, namun konsepnya merupakan gabungan dari IDS dan A*. IDA* melakukan serangkaian pencarian depth-first. Alih-alih menggunakan kedalaman sebagai batas,

IDA* menggunakan nilai total estimasi biaya, $f(n)=g(n)+h(n)$, sebagai batas potong (*f-limit*). Pencarian dimulai dengan f-limit awal (biasanya $h(\text{node awal})$). Pada setiap iterasi, algoritma melakukan pencarian depth-first yang hanya mengeksplorasi node-node yang $f(n)$ -nya tidak melebihi f-limit saat ini. Jika solusi tidak ditemukan, f-limit berikutnya adalah nilai $f(n)$ minimum dari semua node yang terpotong (yang $f(n)$ -nya melebihi f-limit sebelumnya). Ini diulang hingga solusi ditemukan. Dengan heuristik yang *admissible*, IDA* menjamin penemuan solusi optimal (jumlah gerakan minimal) dalam Rush Hour. Keunggulan utamanya adalah penggunaan memori yang sebanding dengan DFS ($O(bd)$), yang jauh lebih baik daripada A* standar (yang menyimpan semua node di *open* dan *closed list*, bisa $O(bd)$). Namun, IDA* juga dapat mengunjungi ulang state, yang bisa membuat waktu eksekusinya lebih lama dibandingkan A* jika A* memiliki cukup memori.

PSEUDOCODE:

```

FUNCTION IDA_Star(initialBoard, heuristicFunction):
    INPUT: initialBoard, heuristicFunction
    OUTPUT: path solusi atau kegagalan

    rootNode = CREATE_NODE(state=initialBoard, gCost=0)
    rootNode.hCost = heuristicFunction(initialBoard)
    fLimit = rootNode.hCost // Batas f awal

    LOOP: // Loop iterasi IDA*
        pathCurrentlyExploring = EMPTY_SET() // Deteksi siklus per iterasi
        DFS
        // Hasil dari ida_search bisa berupa [foundNode, nextFLimit]
        searchResult = IDA_SEARCH(rootNode, fLimit, heuristicFunction,
        pathCurrentlyExploring)
        solutionNode = searchResult[0]
        nextFLimitCandidate = searchResult[1]

        IF solutionNode IS NOT NULL THEN // Solusi ditemukan
            RETURN RECONSTRUCT_PATH(solutionNode)
        ENDIF

        IF nextFLimitCandidate IS MAX_INTEGER THEN // Tidak ada node lagi
        yang bisa dieksplorasi
            RETURN FAILURE
        ENDIF

        fLimit = nextFLimitCandidate // Update fLimit untuk iterasi
        berikutnya
        // Tambahkan kondisi berhenti jika fLimit terlalu besar atau
        iterasi terlalu banyak
    ENDLOOP

```

```

FUNCTION IDA_SEARCH(currentNode, fLimit, heuristicFunction,
pathCurrentlyExploring):
    INPUT: currentNode, fLimit saat ini, heuristicFunction,
    pathCurrentlyExploring
    OUTPUT: Array [node solusi atau null, f-cost minimum berikutnya atau
MAX_INTEGER]

    currentFCost = currentNode.gCost + currentNode.hCost

    IF currentFCost > fLimit THEN
        RETURN [NULL, currentFCost] // Terpotong, kembalikan f-costnya
        untuk fLimit berikutnya
    ENDIF

    IF IS_GOAL_STATE(currentNode.state) THEN
        RETURN [currentNode, currentFCost] // Solusi ditemukan
    ENDIF

    IF currentNode.state IN pathCurrentlyExploring THEN
        RETURN [NULL, MAX_INTEGER] // Siklus
    ENDIF
    ADD currentNode.state TO pathCurrentlyExploring

    minSuccessorFLimit = MAX_INTEGER

    FOR EACH validMove FROM GET_ALL_POSSIBLE_MOVES(currentNode.state):
        childState = GENERATE_NEW_BOARD_STATE(currentNode.state,
validMove)
        childNode = CREATE_NODE(state=childState, parent=currentNode,
move=validMove,
                gCost=currentNode.gCost + 1)
        childNode.hCost = heuristicFunction(childState)
        // fCost anak akan dihitung di pemanggilan rekursif IDA_SEARCH
berikutnya

        recursiveResult = IDA_SEARCH(childNode, fLimit, heuristicFunction,
pathCurrentlyExploring)
        foundSolutionFromChild = recursiveResult[0]
        potentialNextFLimitFromChild = recursiveResult[1]

        IF foundSolutionFromChild IS NOT NULL THEN // Solusi ditemukan
        dari anak
            REMOVE currentNode.state FROM pathCurrentlyExploring // Backtrack
            RETURN [foundSolutionFromChild, 0] // 0 sebagai placeholder,
            tidak akan digunakan
        ENDIF

        minSuccessorFLimit = MIN(minSuccessorFLimit,
potentialNextFLimitFromChild)
    ENDFOR

    REMOVE currentNode.state FROM pathCurrentlyExploring // Backtrack
    RETURN [NULL, minSuccessorFLimit]

```

```
ENDFUNCTION
```

- **Algoritma IDS**

Iterative Deepening Search (IDS) adalah strategi pencarian tak terinformasi (*uninformed search*) yang disebutkan dalam salindia (Bagian 1, hal. 6, 9). IDS menggabungkan kelebihan DFS (efisiensi memori) dan BFS (kelengkapan dan optimalitas untuk biaya langkah seragam). IDS bekerja dengan melakukan serangkaian Depth-Limited Search (DLS) dengan batas kedalaman (depth-limit) yang meningkat secara bertahap (0, 1, 2, ...). Pada setiap iterasi, DLS menjelajahi semua node hingga batas kedalaman yang ditentukan. Jika solusi tidak ditemukan, depth-limit ditingkatkan, dan pencarian DLS diulang dari awal. Karena setiap gerakan dalam Rush Hour memiliki biaya 1, IDS akan menemukan solusi dengan jumlah gerakan paling sedikit (solusi optimal). Penggunaan memorinya sebanding dengan DFS ($O(bd)$, di mana b adalah faktor percabangan dan d adalah kedalaman solusi). Kekurangannya adalah node-node di level atas akan dieksplorasi berulang kali pada setiap peningkatan depth-limit.

PSEUDOCODE:

```
FUNCTION IDS(initialBoard):
    INPUT: initialBoard
    OUTPUT: path solusi atau kegagalan

    FOR depthLimit FROM 0 TO INFINITY (atau batas maksimum yang
    ditentukan):
        visitedInCurrentDLS = EMPTY_SET() // Untuk deteksi siklus dalam
        satu iterasi DLS
        startNode = CREATE_NODE(state=initialBoard, gCost=0) // gCost
        adalah kedalaman
        resultNode = DLS(startNode, depthLimit, 0, visitedInCurrentDLS)
        IF resultNode IS NOT FAILURE THEN
            RETURN RECONSTRUCT_PATH(resultNode)
        ENDIF
    ENDFOR
    RETURN FAILURE // Jika loop selesai (misalnya, karena batas
    maksimum)

FUNCTION DLS(currentNode, depthLimit, currentDepth,
visitedInCurrentDLS):
    INPUT: currentNode, depthLimit, currentDepth, visitedInCurrentDLS
    OUTPUT: node solusi atau kegagalan

    IF IS_GOAL_STATE(currentNode.state) THEN
        RETURN currentNode
    ENDIF
```

```

IF currentDepth >= depthLimit THEN
    RETURN FAILURE // Batas kedalaman tercapai
ENDIF

IF currentNode.state IN visitedInCurrentDLS THEN
    RETURN FAILURE // Siklus terdeteksi dalam path DLS saat ini
ENDIF
ADD currentNode.state TO visitedInCurrentDLS

FOR EACH validMove FROM GET_ALL_POSSIBLE_MOVES(currentNode.state):
    childState = GENERATE_NEW_BOARD_STATE(currentNode.state,
    validMove)
    // gCost untuk childNode adalah currentDepth + 1
    childNode = CREATE_NODE(state=childState, parent=currentNode,
    move=validMove,
                                gCost=currentDepth + 1)
    result = DLS(childNode, depthLimit, currentDepth + 1,
    visitedInCurrentDLS)
    IF result IS NOT FAILURE THEN
        REMOVE currentNode.state FROM visitedInCurrentDLS // Backtrack
        RETURN result
    ENDIF
ENDFOR

REMOVE currentNode.state FROM visitedInCurrentDLS // Backtrack
RETURN FAILURE
ENDFUNCTION

```

VII. LAMPIRAN

Repositori program ini dapat dilihat [disini](#).

Poin	Ya	Tidak
1. Program berhasil dikompilasi tanpa kesalahan	V	
2. Program berhasil dijalankan	V	
3. Solusi yang diberikan program benar dan mematuhi aturan permainan	V	
4. Program dapat membaca masukan berkas .txt dan menyimpan solusi berupa print board tahap per tahap dalam berkas .txt	V	
5. [Bonus] Implementasi algoritma pathfinding alternatif	V	

6. [Bonus] Implementasi 2 atau lebih heuristik alternatif	V	
7. [Bonus] Program memiliki GUI	V	
8. Program dan laporan dibuat (kelompok) sendiri	V	