

LAPORAN TUGAS KECIL 3
IF2211 STRATEGI ALGORITMA

Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding



Disusun oleh:

Jethro Jens Norbert Simatupang – 13523081

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

2025

DAFTAR ISI

DAFTAR ISI.....	2
BAB 1 PENDAHULUAN	3
BAB 2 PENJELASAN ALGORITMA.....	4
BAB 3 ANALISIS ALGORITMA	10
BAB 4 SOURCE CODE	12
BAB 5 EKSPERIMENT	31
BAB 6 ANALISIS KOMPLEKSITAS.....	42
BAB 7 IMPLEMENTASI BONUS	43
LAMPIRAN.....	44

BAB 1

PENDAHULUAN

1.1. Latar Belakang

Rush Hour adalah sebuah permainan puzzle logika berbasis grid yang menantang pemain untuk menggeser kendaraan di dalam sebuah kotak (biasanya berukuran 6x6) agar mobil utama (biasanya berwarna merah) dapat keluar dari kemacetan melalui pintu keluar di sisi papan. Setiap kendaraan hanya bisa bergerak lurus ke depan atau ke belakang sesuai dengan orientasinya (horizontal atau vertikal), dan tidak dapat berputar. Tujuan utama dari permainan ini adalah memindahkan mobil merah ke pintu keluar dengan jumlah langkah seminimal mungkin.

Permainan *Rush Hour* merupakan representasi sempurna dari masalah pencarian jalur (*pathfinding problem*). Dalam permainan ini, setiap konfigurasi kendaraan di papan dapat dianggap sebagai sebuah *state* atau *node* dalam graf dan setiap pergerakan kendaraan menuju konfigurasi baru merupakan sebuah *edge*. Tujuannya adalah menemukan jalur dari *state awal* (konfigurasi awal papan) menuju *state tujuan* (saat mobil merah mencapai pintu keluar), dengan memperhatikan *constraint-constraint* tertentu. Untuk menyelesaikan puzzle ini secara otomatis, kita dapat menerapkan beberapa algoritma pencarian, seperti *Uniform Cost Search*, *Greedy Best First Search*, dan *A* Search*.

1.2. Tujuan

Tujuan tugas kecil ini adalah:

- Mengimplementasikan berbagai algoritma *pathfinding* untuk menemukan solusi dari permainan *Rush Hour*
- Menganalisis hasil penerapan berbagai algoritma *pathfinding* untuk menemukan solusi dari permainan *Rush Hour*

1.3. Teori Singkat

Algoritma *pathfinding* adalah salah satu pendekatan pemrograman yang digunakan untuk menemukan jalur terbaik antara dua titik dalam suatu ruang pencarian, seperti graf, grid, atau peta. Jalur terbaik ini biasanya berdasarkan pada kriteria tertentu, seperti jarak terpendek, biaya terendah, waktu tercepat, dan jumlah langkah minimum. Beberapa algoritma *pathfinding* yang umum digunakan mencakup *Uniform Cost Search*, *Greedy Best First Search*, dan *A* Search*.

Uniform Cost Search (UCS) adalah algoritma pencarian berbasis *graf* yang mengeksplorasi node dengan biaya kumulatif terkecil terlebih dahulu. UCS menjamin pencarian solusi optimal. Rumus utama algoritma ini adalah $f(n) = g(n)$ dengan $g(n)$ adalah biaya dari node awal ke n .

Greedy Best First Search (GBFS) adalah algoritma pencarian heuristik yang hanya mempertimbangkan estimasi jarak ke tujuan, tanpa memperhatikan biaya langkah sejauh ini. Rumus utama algoritma ini adalah $f(n) = h(n)$ dengan $h(n)$ adalah estimasi biaya dari node n ke node target.

A Search* adalah algoritma pencarian berbasis *graf* yang menggabungkan kelebihan UCS (biaya sejauh ini) dan GBFS (estimasi ke tujuan). *A** dirancang untuk mencari solusi secara optimal dan efisien, asalkan heuristik yang digunakan adalah *admissible* dan *consistent*. Adapun rumus utama algoritma ini adalah $f(n) = g(n) + h(n)$.

BAB 2

PENJELASAN ALGORITMA

2.1. Notasi *Pseudocode*

a) Algoritma *Uniform Cost Search*

```
{Uniform Cost Search}

class UniformCostSearch inherits SearchAlgorithm

    class Node
        board
        path
        cost

        constructor Node(board, path, cost)
            self.board ← board
            self.path ← path
            self.cost ← cost
        end constructor
    end class

    function solve(initialBoard)
        startTime ← current time in milliseconds
        nodesExplored ← 0

        queue ← PriorityQueue ordered by cost ascending
        visited ← empty set

        queue.add(Node(initialBoard, empty list, 0))

        while queue is not empty do
            current ← queue.poll()
            nodesExplored ← nodesExplored + 1

            if current.board.isSolved() then
                printStats(true)
                return current.path
            endif

            if not visited.add(current.board.getStateString()) then
                continue
            endif

            for each move in current.board.getPossibleMoves() do
                newBoard ← current.board.movePiece(move.getPiece(),
                move.getDirection(), move.getSteps())
                newPath ← copy of current.path
                newPath.add(move)

                queue.add(Node(newBoard, newPath, current.cost +
                move.getSteps()))
            endfor
        endwhile

        printStats(false)
        return null
    end function

end class
```

b) Algoritma *Greedy Best First Search*

```
{Greedy Best First Search}

class GreedyBestFirstSearch inherits SearchAlgorithm

    class Node
        board
        path
        heuristic

        constructor Node(board, path, heuristic)
            self.board ← board
            self.path ← path
            self.heuristic ← heuristic
        end constructor
    end class

    function solve(initialBoard)
        startTime ← current time in milliseconds
        nodesExplored ← 0

        queue ← PriorityQueue ordered by ascending heuristic
        visited ← empty set

        queue.add(Node(initialBoard, empty list,
heuristic.calculate(initialBoard)))

        while queue is not empty do
            current ← queue.poll()
            nodesExplored ← nodesExplored + 1

            if current.board.isSolved() then
                printStats(true)
                return current.path
            endif

            if not visited.add(current.board.getStateString()) then
                continue
            endif

            for each move in current.board.getPossibleMoves() do
                newBoard ← current.board.movePiece(move.getPiece(),
move.getDirection(), move.getSteps())
                newPath ← copy of current.path
                newPath.add(move)

                queue.add(Node(newBoard, newPath,
heuristic.calculate(newBoard)))
            endfor
        endwhile

        printStats(false)
        return null
    end function

end class
```

c) Algoritma A* Search

```

{A* Search}

class AStarSearch inherits SearchAlgorithm

    class Node
        board
        path
        cost
        heuristic
        total

        constructor Node(board, path, cost, heuristic)
            self.board ← board
            self.path ← path
            self.cost ← cost
            self.heuristic ← heuristic
            self.total ← cost + heuristic
        end constructor
    end class

    function solve(initialBoard)
        startTime ← current time in milliseconds
        nodesExplored ← 0

        queue ← PriorityQueue ordered by ascending total
        costMap ← empty map

        queue.add(Node(initialBoard, empty list, 0,
        heuristic.calculate(initialBoard)))
        costMap.put(initialBoard.getStateString(), 0)

        while queue is not empty do
            current ← queue.poll()
            nodesExplored ← nodesExplored + 1

            if current.board.isSolved() then
                printStats(true)
                return current.path
            endif

            if current.cost >
costMap.getOrDefault(current.board.getStateString(), Integer.MAX_VALUE)
then
                continue
            endif

            for each move in current.board.getPossibleMoves() do
                newBoard ← current.board.movePiece(move.getPiece(),
move.getDirection(), move.getSteps())
                newCost ← current.cost + move.getSteps()
                state ← newBoard.getStateString()

                if newCost < costMap.getOrDefault(state,
Integer.MAX_VALUE) then
                    newPath ← copy of current.path
                    newPath.add(move)

                    queue.add(Node(newBoard, newPath, newCost,
heuristic.calculate(newBoard)))
                    costMap.put(state, newCost)
                endif
            endfor
        endwhile

        printStats(false)
        return null
    end function

end class

```

2.2. Penjelasan Algoritma *Uniform Cost Search*

Algoritma *Uniform Cost Search* (UCS) pada kode ini merupakan algoritma pencarian berbasis *cost* (biaya nyata dari node awal ke node n) yang menjamin menemukan solusi optimal (biaya total terendah) selama semua langkah memiliki biaya non-negatif. Tujuan algoritma ini adalah mencari jalur dari *state* awal (*initialBoard*) menuju *goal state* (*board.isSolved()* mengembalikan true) dengan total biaya minimum. UCS termasuk dalam golongan algoritma *uninformed search*.

Langkah-langkah pada algoritma ini adalah sebagai berikut:

1. Inisialisasi *Priority Queue*

PriorityQueue<Node> digunakan untuk memilih node dengan biaya terendah (*cost*) terlebih dahulu. Inisialisasi dilakukan dengan menambahkan node awal ke antrian dengan *cost* = 0.

2. Looping Selama Antrian Tidak Kosong

Node dengan *cost* terkecil diambil dari antrian (*queue.poll()*). Jika *state* tersebut adalah *goal state*, maka jalur (*path*) menuju *state* tersebut dikembalikan. Jika sudah pernah dikunjungi (*visited.contains(state)*), proses dilanjutkan ke node berikutnya (menghindari eksplorasi ulang). Ketika mengunjungi *state* baru, *state* akan ditandai sebagai sudah dikunjungi.

3. Eksplorasi Semua Gerakan yang Mungkin dari *State* Saat Ini

Untuk setiap gerakan legal, *state* baru akan dibuat setelah melakukan langkah tersebut, langkah ke jalur (*path*) yang baru akan ditambahkan, dan node baru (dengan *cost* yang diperbarui) akan ditambahkan ke antrian.

4. Kondisi Semua Node Sudah Dieksplorasi dan Tidak Ada Solusi

Algoritma akan mengembalikan nilai null dan memberikan output bahwa solusi tidak ditemukan.

Adapun komponen penting dalam algoritma ini adalah sebagai berikut:

- Board *board* = *State* papan saat ini
- List<*Move*> *path* = Urutan langkah dari *state* awal ke *state* saat ini
- int *cost* = Total biaya dari *state* awal ke *state* saat ini (*g(n)*)
- *visited* = Variabel untuk menghindari eksplorasi ulang *state* yang sudah dikunjungi.
- *getStateString()* = Representasi unik dari *state* papan.
- *move.getSteps()*: Biaya dari sebuah langkah yang ditentukan oleh banyaknya langkah yang dilakukan.

2.3. Penjelasan Algoritma *Greedy Best First Search*

Algoritma *Greedy Best First Search* (GBFS) pada kode ini mencari jalur dari *initialBoard* menuju *goal state* dengan mengandalkan nilai heuristik (*h(n)*), yaitu *perkiraan* seberapa dekat suatu *state* dengan goal, tanpa mempertimbangkan *cost* dari awal. Algoritma ini selalu memilih *state* yang “terlihat paling dekat” ke goal berdasarkan heuristic sehingga tidak menjamin solusi optimal, tetapi biasanya lebih cepat dibanding algoritma UCS ataupun A*.

Langkah-langkah pada algoritma ini adalah sebagai berikut:

1. Inisialisasi *Priority Queue*

PriorityQueue<Node> digunakan untuk mengurutkan node berdasarkan nilai heuristik terkecil (*h(n)*). Inisialisasi dilakukan dengan menambahkan node awal (*initialBoard*) dengan nilai heuristik awal.

2. Looping Selama Antrian Tidak Kosong

Looping dilakukan dengan mengambil node dengan heuristic terendah (queue.poll()). Jika node tersebut adalah *goal state*, jalur menuju ke *state* tersebut (*path*) akan dikembalikan. Jika *state* sudah dikunjungi sebelumnya (visited.contains(state)), lanjutkan ke iterasi berikutnya. Ketika mengunjungi *state* baru, *state* akan ditandai sebagai sudah dikunjungi.

3. Eksplorasi Semua Gerakan Legal dari *State* Saat Ini

Untuk setiap langkah, newBoard (*state* baru hasil langkah) akan dibuat, jalur (*path*) akan disalin dan langkah tersebut akan ditambahkan. Nilai heuristik baru $h(\text{newBoard})$ dihitung menggunakan fungsi heuristic dan node ditambahkan ke antrian prioritas.

4. Jika Semua Node Sudah Dieksplorasi dan Tidak Ada Solusi

Algoritma akan mengembalikan nilai null dan memberikan output bahwa solusi tidak ditemukan.

Adapun komponen penting dalam algoritma ini adalah sebagai berikut:

- Board board = *State* papan saat ini
- List<Move> path = Langkah-langkah dari *state* awal ke *state* saat ini
- int heuristic = Perkiraan biaya ke *goal* ($h(n)$)
- Heuristic heuristic = Objek eksternal yang menghitung nilai estimasi $h(n)$ untuk sebuah papan.
- visited = Set yang melacak semua *state* yang sudah dieksplorasi agar tidak diproses ulang.
- Move.getSteps() = Digunakan hanya untuk langkah, bukan untuk cost

2.4. Penjelasan Algoritma *A* Search*

Algoritma *A* Search* pada kode ini mencari jalur dari initialBoard ke *goal state* dengan mempertimbangkan kombinasi *cost* dan estimasi biaya ke tujuan (heuristik). Algoritma ini menggunakan fungsi penilaian $f(n) = g(n) + h(n)$ dan menjamin solusi optimal selama $h(n)$ bersifat admissible.

Langkah-langkah pada algoritma ini adalah sebagai berikut:

1. Inisialisasi *Priority Queue*

PriorityQueue<Node> digunakan untuk mengurutkan node berdasarkan $f(n) = g(n) + h(n)$. Inisialisasi dilakukan dengan menambahkan node awal (initialBoard) dengan $g(n) = 0$ dan $h(n)$ dari heuristic.

2. Menyimpan Biaya Terkecil Tiap *State*

Map<String, Integer> costMap digunakan untuk menyimpan $g(n)$ terkecil yang ditemukan sejauh ini untuk setiap *state*.

3. Looping Selama Antrian Tidak Kosong

Looping dilakukan dengan mengambil node dengan total cost terendah ($f(n)$) dari queue. Jika *state* adalah *goal*, *path*-nya akan dikembalikan. Jika $g(n)$ saat ini lebih besar dari $g(n)$ terbaik yang sudah tercatat, node tersebut akan diabaikan.

4. Eksplorasi Langkah Legal

Untuk setiap langkah *state* baru newBoard akan dibuat dan newCost = current.cost + move.getSteps() akan dihitung sebagai $g(n)$. Jika newCost lebih kecil dari yang tercatat untuk *state* tersebut, maka *path* baru dengan langkah tersebut akan ditambahkan, $h(n)$ dari newBoard akan dihitung, langkah akan ditambahkan ke queue, dan costMap akan diperbarui.

5. Jika Semua Node Telah Dieksplorasi dan Tidak Menemukan Goal

Algoritma akan mengembalikan nilai null dan memberikan output bahwa solusi tidak ditemukan.

Adapun komponen penting dalam algoritma ini adalah sebagai berikut:

- Board board = *State* papan saat ini
- List<Move> path = Jalur dari *state* awal ke *state* ini
- int cost = Total biaya dari *state* awal ke *state* saat ini ($g(n)$)
- int heuristic = Perkiraan biaya ke goal ($h(n)$)
- int total = $g(n) + h(n)$
- Heuristic heuristic = Digunakan untuk menghitung $h(n)$ dari suatu papan.
- move.getSteps() = Mengasumsikan bahwa setiap langkah memiliki cost ($g(n)$) berdasarkan jumlah langkah
- costMap = Mencegah penambahan node yang sudah pernah ditemukan jalur lebih optimalnya

BAB 3

ANALISIS ALGORITMA

3.1 Analisis f(n) dan g(n)

Dalam algoritma-algoritma yang digunakan pada tugas ini, terdapat dua konsep utama, yaitu $g(n)$ dan $h(n)$ yang digunakan untuk menentukan prioritas eksplorasi node. Fungsi $g(n)$ merepresentasikan biaya aktual atau cost yang sudah dikeluarkan, sedangkan $h(n)$ adalah nilai heuristik, yaitu perkiraan biaya tersisa dari node n menuju tujuan (goal). Algoritma UCS hanya menggunakan $g(n)$, GBFS hanya menggunakan $h(n)$, dan A* menggabungkan keduanya dalam fungsi $f(n) = g(n) + h(n)$. Nilai ini digunakan untuk memprioritaskan node yang akan diperiksa selanjutnya agar pencarian lebih efisien dan dapat menemukan jalur optimal.

Fungsi $g(n)$ merepresentasikan total biaya yang sudah dikeluarkan untuk mencapai suatu node atau keadaan papan (*state*) tertentu dari titik awal pencarian. Biaya ini disimpan dalam variabel cost pada setiap objek Node, yang merupakan jumlah kumulatif dari langkah-langkah (steps) yang telah diambil sejak posisi awal sampai ke node tersebut. Setiap kali bergerak dengan sebuah Move, nilai cost diperbarui dengan menambahkan jumlah langkah dari gerakan baru ke total biaya sebelumnya sehingga cost mencerminkan berapa “mahal” atau berapa besar biaya yang sudah dikeluarkan untuk mencapai node tersebut. Dengan demikian, $g(n)$ adalah ukuran akurat dari total biaya jalur yang sudah ditempuh dari node awal ke node saat ini dan menjadi dasar utama UCS dalam menemukan jalur solusi dengan biaya minimum.

Fungsi $h(n)$ yang dihitung berdasarkan heuristik, dirancang untuk memperkirakan seberapa “jauh” keadaan saat ini dari solusi. Pada tugas ini, terdapat dua heuristik yang digunakan secara terpisah, yaitu BlockingHeuristic dan MobilityHeuristic. BlockingHeuristic menghitung jumlah blocker yang menghalangi jalur *primary piece* menuju goal, sedangkan MobilityHeuristic menghitung jumlah langkah legal atau kebebasan bergerak yang tersisa. Dengan menggunakan heuristik ini, algoritma GBFS dan A* mampu menghindari eksplorasi jalur yang jelas-jelas tidak efisien sehingga mempercepat proses pencarian dan tetap menjamin solusi yang ditemukan adalah optimal.

3.2 Analisis Heuristik

Heuristik dikatakan *admissible* jika untuk setiap node n , nilai heuristik $h(n)$ selalu lebih kecil dari biaya sebenarnya $h^*(n)$. Dengan kata lain, heuristik harus selalu *optimistik* atau *underestimate* biaya yang diperlukan sehingga tidak memberikan estimasi yang berlebihan yang dapat menyesatkan proses pencarian.

BlockingHeuristic menghitung jumlah blocker yang menghalangi jalur *primary piece* menuju goal. Heuristik ini memenuhi sifat *admissible* karena estimasi biaya $h(n)$ yang dihitung tidak pernah melebihi biaya sebenarnya $h^*(n)$ untuk mencapai goal dari node n . Dalam konteks ini, $h(n)$ hanya menghitung jumlah *piece* yang harus dipindahkan tanpa menambahkan penalti berlebih sehingga selalu memberikan nilai yang kurang dari atau sama dengan biaya minimum yang dibutuhkan untuk mengatasi hambatan tersebut dan mencapai tujuan. Dengan kata lain, heuristik ini *optimistic* karena tidak melebih-lebihkan biaya jalan keluar dari node saat ini menuju goal sehingga memenuhi sifat admissible, yaitu $h(n) \leq h^*(n)$ untuk setiap node n .

Sementara itu, MobilityHeuristic mengukur jumlah *piece* yang tidak dapat bergerak sama sekali. Heuristik ini memberikan gambaran kasar tentang tingkat kesulitan kondisi papan saat ini. Meskipun heuristik ini intuitif dalam menilai mobilitas, tetapi belum tentu bersifat *admissible* karena menghitung kondisi mobilitas semua *piece* tanpa mengaitkan langsung dengan jarak sebenarnya ke goal.

3.3 Analisis Kesamaan dan Perbedaan *Uniform Cost Search* dan *Breadth-First Search*

Pada implementasi UCS di tugas ini, *priority queue* digunakan untuk memilih node dengan total cost terkecil ($g(n)$) pada setiap iterasi. Jika setiap langkah atau gerakan memiliki biaya yang sama dan konstan, maka UCS akan mengeksplorasi node berdasarkan kedalaman (depth) dari pohon pencarian, sehingga urutan node yang dibangkitkan akan sama dengan BFS. Dalam kondisi ini, jalur yang ditemukan UCS dan BFS juga sama, yaitu jalur dengan jumlah langkah minimal.

Namun, apabila langkah-langkah memiliki biaya yang bervariasi, UCS akan memprioritaskan node dengan total biaya terendah, bukan berdasarkan urutan kedalaman. Ini menyebabkan UCS bisa mengunjungi node dengan depth lebih dalam lebih awal daripada node di depth lebih dangkal. Oleh karena itu, urutan node yang dibangkitkan oleh UCS dan BFS akan berbeda dalam kasus biaya langkah tidak seragam dan path yang dihasilkan oleh UCS akan menjadi jalur biaya minimum yang mungkin berbeda dengan path BFS yang hanya meminimalkan jumlah langkah.

3.4 Analisis Efisiensi *A* Search* Dibanding *Uniform Cost Search*

Algoritma *Uniform Cost Search* melakukan eksplorasi node berdasarkan biaya dari node awal ke node tersebut ($g(n)$) tanpa mempertimbangkan seberapa dekat node tersebut dengan tujuan akhir. Akibatnya, UCS akan mengunjungi semua jalur dengan cost rendah terlebih dahulu dan bisa menyebabkan eksplorasi ruang pencarian yang besar dan tidak fokus pada solusi. Dalam konteks Rush Hour, di mana ruang pencarian bisa sangat besar dan terdapat banyak konfigurasi papan, UCS cenderung lebih lambat.

Sebaliknya, algoritma A^* menggunakan fungsi evaluasi $f(n) = g(n) + h(n)$ dengan $h(n)$ adalah heuristik yang memperkirakan biaya tersisa menuju goal. Dengan digunakannya heuristik, A^* mampu melakukan pencarian langsung ke arah solusi dengan lebih efisien dan mengurangi jumlah node yang perlu dieksplorasi dibanding UCS. Pada permainan Rush Hour, heuristik seperti BlockingHeuristic atau MobilityHeuristic memberikan estimasi yang membantu A^* fokus mengeksplorasi jalur yang menjanjikan sehingga waktu pencarian dan jumlah node yang dikunjungi lebih sedikit daripada UCS. Oleh karena itu, A^* umumnya lebih efisien dibandingkan UCS walaupun biasanya solusinya lebih tidak optimal.

3.5 Analisis Keoptimalan Algoritma *Greedy Best First Search*

Greedy Best First Search (GBFS) adalah algoritma pencarian yang memilih node berikutnya berdasarkan nilai heuristik $h(n)$. GBFS tidak mempertimbangkan biaya aktual jalur yang telah dilalui ($g(n)$) sehingga hanya fokus pada node yang secara heuristik tampak paling menjanjikan. Karena itu, algoritma ini rentan memilih jalur yang secara heuristik terlihat bagus, tetapi sebenarnya lebih panjang atau lebih mahal jika dilanjutkan ke goal. Maka, dapat disimpulkan bahwa GBFS tidak menjamin solusi optimal dan lebih cocok digunakan ketika efisiensi waktu pencarian lebih diutamakan.

BAB 4

SOURCE CODE

4.1 Main.java

```

1      // Pilihan heuristic
2      if (choice == 2 || choice == 3) {
3          System.out.println("\nSelect heuristic:");
4          System.out.println("1. Mobility");
5          System.out.println("2. Blocking Pieces");
6          System.out.print("Choice: ");
7          int hChoice = scanner.nextInt();
8
9          heuristic = hChoice == 1 ? new MobilityHeuristic() : new BlockingHeuristic();
10     }
11
12    switch (choice) {
13        case 1:
14            algorithm = new UniformCostSearch();
15            break;
16        case 2:
17            algorithm = new GreedyBestFirstSearch(heuristic);
18            break;
19        case 3:
20            algorithm = new AStarSearch(heuristic);
21            break;
22        default:
23            System.out.println("Invalid choice");
24            return;
25    }
26
27    System.out.println("\nSolving puzzle...");
28    // Menjalankan algoritma
29    List<Move> solution = algorithm.solve(board);
30
31    if (solution != null) {
32        System.out.println("\nSolution found in " + solution.size() + " moves:");
33        printSolution(board, solution);
34
35        // Menyimpan hasil solusi ke file
36        scanner.nextLine();
37        String outputFileName;
38        while (true) {
39            System.out.print("\nEnter output file name (must end with .txt): ");
40            outputFileName = scanner.nextLine().trim();
41            if (outputFileName.toLowerCase().endsWith(".txt")) {
42                break;
43            } else {
44                System.out.println("Invalid file name. The output file must have a '.txt' extension.");
45            }
46        }
47        String outputDir = "test/output";
48        File outputDirectory = new File(outputDir);
49        if (!outputDirectory.exists()) {
50            outputDirectory.mkdirs();
51        }
52        String outputPath = outputDir + File.separator + outputFileName;
53        saveSolutionToFile(outputPath, board, solution);
54        System.out.println("\nSolution saved to: " + outputPath);
55
56    } else {
57        System.out.println("\nNo solution found");
58        scanner.nextLine(); // consume newline left from previous nextInt()
59        String outputFileName;
60        while (true) {
61            System.out.print("\nEnter output file name to save the result (must end with .txt): ");
62            outputFileName = scanner.nextLine().trim();
63            if (outputFileName.toLowerCase().endsWith(".txt")) {
64                break;
65            } else {
66                System.out.println("Invalid file name. The output file must have a '.txt' extension.");
67            }
68        }
69
70        String outputDir = "test/output";
71        File outputDirectory = new File(outputDir);
72        if (!outputDirectory.exists()) {
73            outputDirectory.mkdirs();
74        }
75
76        String outputPath = outputDir + File.separator + outputFileName;
77        saveNoSolutionToFile(outputPath, board);
78        System.out.println("\nResult saved to: " + outputPath);
79    }
80 }
81

```

```

1 // Menyimpan solusi ke file .txt
2 private static void saveSolutionToFile(String filePath, Board initialBoard, List<Move> solution) {
3     try (PrintWriter writer = new PrintWriter(new FileWriter(filePath))) {
4         Board currentBoard = initialBoard;
5         writer.println("Initial board:");
6         writeBoardToFile(writer, currentBoard, null);
7
8         for (int i = 0; i < solution.size(); i++) {
9             Move move = solution.get(i);
10            writer.printf("\nMove %d: %s\n", i + 1, move);
11
12            currentBoard = currentBoard.movePiece(move.getPiece(), move.getDirection(), move.getSteps());
13            writeBoardToFile(writer, currentBoard, move.getPiece());
14        }
15
16        // Move terakhir
17        if (currentBoard.isSolved()) {
18            Piece p = currentBoard.getPrimaryPiece();
19            String exitDirection = p.getOrientation().equals("horizontal") ? "right" : "down";
20            Move finalMove = new Move(p, exitDirection, 1);
21
22            writer.printf("\nMove %d: %s\n", solution.size() + 1, finalMove);
23
24            char[][] grid = new char[currentBoard.getRows()][currentBoard.getCols()];
25            for (int i = 0; i < currentBoard.getRows(); i++) {
26                System.arraycopy(currentBoard.getGrid()[i], 0, grid[i], 0, currentBoard.getCols());
27            }
28
29            for (int k = 0; k < p.getLength(); k++) {
30                if (p.getOrientation().equals("horizontal")) {
31                    grid[p.getRow()][p.getCol() + k] = '.';
32                } else {
33                    grid[p.getRow() + k][p.getCol()] = '.';
34                }
35            }
36
37            Board exitedBoard = new Board(
38                currentBoard.getRows(),
39                currentBoard.getCols(),
40                grid,
41                currentBoard.getPieces(),
42                currentBoard.getExitRow(),
43                currentBoard.getExitCol()
44            );
45
46            writeBoardToFile(writer, exitedBoard, null);
47        }
48    } catch (IOException e) {
49        System.err.println("Error saving solution to file: " + e.getMessage());
50    }
51 }
52
53 private static void saveNoSolutionToFile(String filePath, Board initialBoard) {
54     try (PrintWriter writer = new PrintWriter(new FileWriter(filePath))) {
55         writer.println("Initial board:");
56         writeBoardToFile(writer, initialBoard, null);
57         writer.println("\nNo solution found for this puzzle.");
58     } catch (IOException e) {
59         System.err.println("Error saving result to file: " + e.getMessage());
60     }
61 }
62
63 // Menulis kondisi board ke file
64 private static void writeBoardToFile(PrintWriter writer, Board board, Piece movingPiece) {
65     char[][] grid = board.getGrid();
66     int rows = board.getRows();
67     int cols = board.getCols();
68
69     if (board.getExitCol() == cols) {
70         for (int i = 0; i < rows; i++) {
71             String line = new String(grid[i]);
72             if (i == board.getExitRow()) {
73                 writer.println(line + "K");
74             } else {
75                 writer.println(line + " ");
76             }
77         }
78     } else {
79         for (int i = 0; i < rows; i++) {
80             writer.println(new String(grid[i]));
81         }
82     }
83 }
84
85 // Menampilkan solusi ke Layar
86 private static void printSolution(Board initialBoard, List<Move> solution) {
87     Board currentBoard = initialBoard;
88     System.out.println("Initial board:");
89     printBoard(currentBoard, null);
90
91     for (int i = 0; i < solution.size(); i++) {
92         Move move = solution.get(i);
93         System.out.printf("\nMove %d: %s\n", i + 1, move);
94
95         // Mencari piece di board saat ini
96         Piece pieceToMove = null;
97         for (Piece p : currentBoard.getPieces()) {
98             if (p.getId() == move.getPiece().getId()) {
99                 pieceToMove = p;
100                break;
101            }
102        }
103    }

```

```

1         if (pieceToMove == null) {
2             throw new IllegalStateException("Piece " + move.getPiece().getId() + " not found in board");
3         }
4
5         currentBoard = currentBoard.movePiece(pieceToMove, move.getDirection(), move.getSteps());
6         printBoard(currentBoard, pieceToMove);
7     }
8
9     // Menampilkan langkah keluar terakhir jika puzzle sudah terselesaikan
10    if (currentBoard.isSolved()) {
11        Piece p = currentBoard.getPrimaryPiece();
12        String exitDirection = p.getOrientation().equals("horizontal") ? "right" : "down";
13        Move finalMove = new Move(p, exitDirection, p.getLength());
14
15        System.out.printf("\nMove %d: %s\n", solution.size() + 1, finalMove);
16
17        char[][] grid = new char[currentBoard.getRows()][currentBoard.getCols()];
18        for (int i = 0; i < currentBoard.getRows(); i++) {
19            System.arraycopy(currentBoard.getGrid()[i], 0, grid[i], 0, currentBoard.getCols());
20        }
21
22        for (int k = 0; k < p.getLength(); k++) {
23            if (p.getOrientation().equals("horizontal")) {
24                grid[p.getRow()][p.getCol() + k] = '.';
25            } else {
26                grid[p.getRow() + k][p.getCol()] = '.';
27            }
28        }
29
30        Board exitedBoard = new Board(
31            currentBoard.getRows(),
32            currentBoard.getCols(),
33            grid,
34            currentBoard.getPieces(),
35            currentBoard.getExitRow(),
36            currentBoard.getExitCol()
37        );
38
39        printBoard(exitedBoard, null);
40    }
41 }
42
43 // Menampilkan kondisi board
44 private static void printBoard(Board board, Piece movingPiece) {
45     final String RESET = "\u001B[0m";
46     final String RED = "\u001B[31m";
47     final String YELLOW = "\u001B[33m";
48     final String BLUE = "\u001B[34m";
49
50     char[][] grid = board.getGrid();
51     int rows = board.getRows();
52     int cols = board.getCols();
53     Piece primaryPiece = board.getPrimaryPiece();
54
55     for (int i = 0; i < rows; i++) {
56         for (int j = 0; j < cols; j++) {
57             char cell = grid[i][j];
58
59             if (cell == primaryPiece.getId()) {
60                 // Primary piece dalam warna biru
61                 System.out.print(BLUE + cell + RESET);
62             } else if (movingPiece != null && cell == movingPiece.getId()) {
63                 // Moving piece dalam warna kuning
64                 System.out.print(YELLOW + cell + RESET);
65             } else if (cell == '.') {
66                 System.out.print(cell);
67             } else if (cell != '.') {
68                 System.out.print(cell);
69             } else {
70                 System.out.print(cell);
71             }
72         }
73
74         // Pintu keluar dalam warna merah
75         if (board.getExitCol() == cols && i == board.getExitRow()) {
76             System.out.print(RED + 'K' + RESET);
77         } else if (board.getExitCol() == cols) {
78             System.out.print(' ');
79         }
80
81         System.out.println();
82     }
83 }
84 }
```

4.2 Board.java



```
1 package model;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class Board {
7     private int rows, cols;
8     private char[][] grid;
9     private List<Piece> pieces;
10    private Piece primaryPiece;
11    private int exitRow, exitCol;
12
13    public Board(int rows, int cols, char[][] grid, List<Piece> pieces, int exitRow, int exitCol) {
14        this.rows = rows;
15        this.cols = cols;
16        this.grid = grid;
17        this.pieces = pieces;
18        this.exitRow = exitRow;
19        this.exitCol = exitCol;
20        for (Piece p : pieces) {
21            if (p.isPrimary()) {
22                this.primaryPiece = p;
23                break;
24            }
25        }
26        if (this.primaryPiece == null) {
27            throw new IllegalStateException("No primary piece (P) found in the board");
28        }
29    }
30
31    public Board(Board other) {
32        this.rows = other.rows;
33        this.cols = other.cols;
34        this.grid = new char[rows][cols];
35        for (int i = 0; i < rows; i++) {
36            System.arraycopy(other.grid[i], 0, this.grid[i], 0, cols);
37        }
38        this.pieces = new ArrayList<>();
39        for (Piece p : other.pieces) {
40            Piece newPiece = new Piece(p.getId(), p.getRow(), p.getCol(), p.getLength(), p.getOrientation(),
41                p.isPrimary());
42            this.pieces.add(newPiece);
43            if (p.isPrimary()) {
44                this.primaryPiece = newPiece;
45            }
46        }
47        this.exitRow = other.exitRow;
48        this.exitCol = other.exitCol;
49    }
50
51    // Memeriksa apakah move valid
52    public boolean isValidMove(Piece piece, String direction, int steps) {
53        int newRow = piece.getRow();
54        int newCol = piece.getCol();
55
56        // Memeriksa posisi setelah move
57        if (piece.getOrientation().equals("horizontal")) {
58            if (direction.equals("left")) {
59                newCol -= steps;
60            } else if (direction.equals("right")) {
61                newCol += steps;
62            } else {
63                return false;
64            }
65        } else {
66            if (direction.equals("up")) {
67                newRow -= steps;
68            } else if (direction.equals("down")) {
69                newRow += steps;
70            } else {
71                return false;
72            }
73        }
74    }
}
```

```

1     if (newRow < 0 || newCol < 0) {
2         return false;
3     }
4
5     if (piece.getOrientation().equals("horizontal")) {
6         if (newCol + piece.getLength() > cols) {
7             return false;
8         }
9     } else {
10        if (newRow + piece.getLength() > rows) {
11            return false;
12        }
13    }
14
15    char[][] tempGrid = new char[rows][cols];
16    for (int i = 0; i < rows; i++) {
17        for (int j = 0; j < cols; j++) {
18            tempGrid[i][j] = grid[i][j] == 'K' ? 'K' : '.';
19        }
20    }
21
22    for (Piece p : pieces) {
23        if (p != piece) {
24            for (int[] cell : p.getOccupiedCells()) {
25                tempGrid[cell[0]][cell[1]] = p.getId();
26            }
27        }
28    }
29
30    for (int i = 0; i < piece.getLength(); i++) {
31        int r = piece.getOrientation().equals("horizontal") ? newRow : newRow + i;
32        int c = piece.getOrientation().equals("horizontal") ? newCol + i : newCol;
33        if (tempGrid[r][c] != '.' && tempGrid[r][c] != 'K') {
34            return false;
35        }
36    }
37
38    return true;
39}
40
41 // Menggerakkan piece
42 public Board movePiece(Piece piece, String direction, int steps) {
43     Board newBoard = new Board(this);
44
45     Piece movedPiece = null;
46     for (Piece p : newBoard.pieces) {
47         if (p.equals(piece)) {
48             movedPiece = p;
49             break;
50         }
51     }
52
53     if (movedPiece == null) {
54         throw new IllegalArgumentException("Piece not found in the board");
55     }
56
57     int newRow = movedPiece.getRow();
58     int newCol = movedPiece.getCol();
59
60     if (direction.equals("left")) {
61         newCol -= steps;
62     } else if (direction.equals("right")) {
63         newCol += steps;
64     } else if (direction.equals("up")) {
65         newRow -= steps;
66     } else if (direction.equals("down")) {
67         newRow += steps;
68     }
69

```

```

1      // Mengupdate grid
2      for (int i = 0; i < rows; i++) {
3          for (int j = 0; j < cols; j++) {
4              newBoard.grid[i][j] = newBoard.grid[i][j] == 'K' ? 'K' : '.';
5          }
6      }
7
8      // Membuat piece baru dengan posisi yang telah diupdate
9      Piece updatedPiece = new Piece(
10          movedPiece.getId(),
11          newRow,
12          newCol,
13          movedPiece.getLength(),
14          movedPiece.getOrientation(),
15          movedPiece.isPrimary()
16      );
17
18      int pieceIndex = newBoard.pieces.indexOf(movedPiece);
19      newBoard.pieces.set(pieceIndex, updatedPiece);
20
21      if (updatedPiece.isPrimary()) {
22          newBoard.primaryPiece = updatedPiece;
23      }
24
25      for (Piece p : newBoard.pieces) {
26          for (int[] cell : p.getOccupiedCells()) {
27              newBoard.grid[cell[0]][cell[1]] = p.getId();
28          }
29      }
30
31      return newBoard;
32  }
33
34  // Memeriksa apakah target state sudah tercapai
35  public boolean isSolved() {
36      if (primaryPiece.getOrientation().equals("horizontal")) {
37          int rightEnd = primaryPiece.getCol() + primaryPiece.getLength() - 1;
38          return primaryPiece.getRow() == exitRow && rightEnd == cols - 1;
39      }
40      else {
41          int bottomEnd = primaryPiece.getRow() + primaryPiece.getLength() - 1;
42          return primaryPiece.getCol() == exitCol && bottomEnd == rows - 1;
43      }
44  }
45
46  public String getState() {
47      StringBuilder sb = new StringBuilder();
48      for (Piece p : pieces) {
49          sb.append(p.getId()).append(p.getRow()).append(p.getCol());
50      }
51      return sb.toString();
52  }
53
54  public List<Move> getPossibleMoves() {
55      List<Move> moves = new ArrayList<>();
56      for (Object[] moveObj : getPossibleMovesArray()) {
57          Piece piece = (Piece) moveObj[0];
58          String direction = (String) moveObj[1];
59          int steps = (Integer) moveObj[2];
60          moves.add(new Move(piece, direction, steps));
61      }
62      return moves;
63  }
64

```

```

1  private List<Object[]> getPossibleMovesArray() {
2      List<Object[]> moves = new ArrayList<>();
3      for (Piece piece : pieces) {
4          String[] directions = piece.getOrientation().equals("horizontal")
5              ? new String[]{"left", "right"}
6              : new String[]{"up", "down"};
7          for (String dir : directions) {
8              int maxSteps = piece.getOrientation().equals("horizontal")
9                  ? (dir.equals("left") ? piece.getCol() : cols - piece.getCol() - piece.getLength())
10                 : (dir.equals("up") ? piece.getRow() : rows - piece.getRow() - piece.getLength());
11              for (int steps = 1; steps <= maxSteps; steps++) {
12                  if (isValidMove(piece, dir, steps)) {
13                      moves.add(new Object[]{piece, dir, steps});
14                  } else {
15                      break;
16                  }
17              }
18          }
19      }
20      return moves;
21  }
22
23  // Getters
24  public int getRows() {
25      return rows;
26  }
27
28  public int getCols() {
29      return cols;
30  }
31
32  public char[][] getGrid() {
33      return grid;
34  }
35
36  public List<Piece> getPieces() {
37      return pieces;
38  }
39
40  public int getExitRow() {
41      return exitRow;
42  }
43
44  public int getExitCol() {
45      return exitCol;
46  }
47
48  public Piece getPrimaryPiece() {
49      return primaryPiece;
50  }
51
52  public String getStateString() {
53      return getState();
54  }
55
56 }

```

4.3 Piece.java

```
● ○ ●
1 package model;
2
3 import java.util.Objects;
4
5 public class Piece {
6     private final char id;
7     private final int length;
8     private int row, col;
9     private final String orientation;
10    private final boolean isPrimary;
11
12    public Piece(char id, int row, int col, int length, String orientation, boolean isPrimary) {
13        this.id = id;
14        this.row = row;
15        this.col = col;
16        this.length = length;
17        this.orientation = orientation;
18        this.isPrimary = isPrimary;
19    }
20
21    // Getters
22    public char getId() { return id; }
23    public int getRow() { return row; }
24    public int getCol() { return col; }
25    public int getLength() { return length; }
26    public String getOrientation() { return orientation; }
27    public boolean isPrimary() { return isPrimary; }
28
29    // Memindahkan posisi Piece berdasarkan delta baris dan kolom
30    public void move(int dr, int dc) {
31        row += dr;
32        col += dc;
33    }
34
35    // Mengkloning Piece
36    public Piece clone() {
37        return new Piece(id, row, col, length, orientation, isPrimary);
38    }
39
40    // Setters
41    public void setRow(int row) {
42        this.row = row;
43    }
44    public void setCol(int col) {
45        this.col = col;
46    }
47
48    @Override
49    // Memeriksa apakah dua piece sama
50    public boolean equals(Object o) {
51        if (this == o) return true;
52        if (o == null || getClass() != o.getClass()) return false;
53        Piece piece = (Piece) o;
54        return id == piece.id &&
55            row == piece.row &&
56            col == piece.col &&
57            length == piece.length &&
58            isPrimary == piece.isPrimary &&
59            orientation.equals(piece.orientation);
60    }
61
62    @Override
63    // Menghasilkan nilai hash berdasarkan semua atribut yang relevan
64    public int hashCode() {
65        return Objects.hash(id, row, col, length, orientation, isPrimary);
66    }
67
68
69    // Mengembalikan array koordinat cell yang ditempati Piece
70    public int[][] getOccupiedCells() {
71        int[][] cells = new int[length][2];
72        for (int i = 0; i < length; i++) {
73            cells[i][0] = orientation.equals("horizontal") ? row : row + i;
74            cells[i][1] = orientation.equals("horizontal") ? col + i : col;
75        }
76        return cells;
77    }
78
79    public boolean isHorizontal() {
80        return orientation.equals("horizontal");
81    }
82
83    public boolean isVertical() {
84        return orientation.equals("vertical");
85    }
86
87    @Override
88    public String toString() {
89        return String.format("%c(%d,%d,%d,%s)", id, row, col, length, orientation);
90    }
91 }
```

4.4 Move.java

```
● ● ●
1 package model;
2
3 public class Move {
4     private Piece piece;
5     private String direction;
6     private int steps;
7
8     public Move(Piece piece, String direction, int steps) {
9         this.piece = piece;
10        this.direction = direction;
11        this.steps = steps;
12    }
13
14    // Getters
15    public Piece getPiece() { return piece; }
16    public String getDirection() { return direction; }
17    public int getSteps() { return steps; }
18
19    @Override
20    public String toString() {
21        return piece.getId() + " " + direction + " " + steps;
22    }
23 }
```

4.5 SearchAlgorithm.java

```
● ● ●
1 package algorithms;
2
3 import java.util.List;
4 import model.Board;
5 import model.Move;
6
7 public abstract class SearchAlgorithm {
8     protected int nodesExplored;
9     protected long startTime;
10
11     public abstract List<Move> solve(Board initialBoard);
12
13     public int getNodesExplored() {
14         return nodesExplored;
15     }
16
17     public long getSearchTime() {
18         return System.currentTimeMillis() - startTime;
19     }
20
21     protected void printStats(boolean solutionFound) {
22         System.out.println("\n==== Search Statistics ===");
23         System.out.println("Solution found: " + solutionFound);
24         System.out.println("Nodes explored: " + nodesExplored);
25         System.out.printf("Time taken: %.3f seconds\n", getSearchTime() / 1000.0);
26     }
27 }
```

4.6 UniformCostSearch.java

```
1 package algorithms;
2
3 import java.util.*;
4
5 import model.Board;
6 import model.Move;
7
8 public class UniformCostSearch extends SearchAlgorithm {
9     // Node untuk menyimpan state papan saat ini, jalur gerakan menuju state tersebut, dan total cost dari root
10    private static class Node {
11        Board board;
12        List<Move> path;
13        int cost;
14
15        Node(Board board, List<Move> path, int cost) {
16            this.board = board;
17            this.path = path;
18            this.cost = cost; // Biaya dari root ke node ini ( $g(n)$ )
19        }
20    }
21
22    @Override
23    public List<Move> solve(Board initialBoard) {
24        startTime = System.currentTimeMillis(); // Inisialisasi waktu pencarian
25        nodesExplored = 0; // Inisialisasi jumlah node yang dieksplorasi
26
27        // PriorityQueue berdasarkan cost terkecil (min-heap), sesuai prinsip UCS
28        PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(n -> n.cost));
29        // Set untuk melacak state yang sudah dikunjungi
30        Set<String> visited = new HashSet<>();
31
32        // Menambahkan node awal ke dalam antrian
33        queue.add(new Node(initialBoard, new ArrayList<>(), 0));
34
35        while (!queue.isEmpty()) {
36            Node current = queue.poll(); // Mengambil node dengan cost terkecil
37            nodesExplored++;
38
39            // Kondisi goal state tercapai
40            if (current.board.isSolved()) {
41                printStats(true);
42                return current.path;
43            }
44
45            // Skip state yang pernah dikunjungi
46            if (!visited.add(current.board.getStateString())) {
47                continue;
48            }
49
50            // Iterasi semua gerakan Legal dari state saat ini
51            for (Move move : current.board.getPossibleMoves()) {
52                Board newBoard = current.board.movePiece(move.getPiece(), move.getDirection(), move.getSteps());
53                // Salin jalur lama dan tambahkan langkah baru ke jalur
54                List<Move> newPath = new ArrayList<>(current.path);
55                newPath.add(move);
56
57                // Menambahkan node baru ke antrian
58                queue.add(new Node(newBoard, newPath, current.cost + move.getSteps()));
59            }
60        }
61
62        // Kasus tidak ada solusi
63        printStats(false);
64        return null;
65    }
66 }
```

4.7 GreedyBestFirstSearch.java

```
● ● ●
1 package algorithms;
2
3 import java.util.*;
4
5 import model.Board;
6 import model.Move;
7 import heuristics.Heuristic;
8
9 public class GreedyBestFirstSearch extends SearchAlgorithm {
10     private final Heuristic heuristic;
11
12     public GreedyBestFirstSearch(Heuristic heuristic) {
13         this.heuristic = heuristic;
14     }
15
16     // Node untuk menyimpan state papan, jalur Langkah, dan nilai heurstik
17     private static class Node {
18         Board board;
19         List<Move> path;
20         int heuristic;
21
22         Node(Board board, List<Move> path, int heuristic) {
23             this.board = board;
24             this.path = path;
25             this.heuristic = heuristic; // Perkiraan biaya ke goal (h(n))
26         }
27     }
28
29     @Override
30     public List<Move> solve(Board initialBoard) {
31         startTime = System.currentTimeMillis(); // Inisialisasi waktu pencarian
32         nodesExplored = 0; // Inisialisasi jumlah node yang dieksplorasi
33
34         // PriorityQueue berdasarkan nilai heurstik terkecil
35         PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(n -> n.heuristic));
36         // Set untuk melacak state yang sudah dikunjungi
37         Set<String> visited = new HashSet<>();
38
39         // Memasukkan node awal dengan nilai heurstik awal
40         queue.add(new Node(initialBoard, new ArrayList<>(), heuristic.calculate(initialBoard)));
41
42         while (!queue.isEmpty()) {
43             Node current = queue.poll(); // Mengambil node dengan heurstik terendah
44             nodesExplored++;
45
46             // Kasus goal state tercapai
47             if (current.board.isSolved()) {
48                 printStats(true);
49                 return current.path;
50             }
51
52             // Skip state yang pernah dikunjungi
53             if (!visited.add(current.board.getStateString())) {
54                 continue;
55             }
56
57             // Iterasi semua gerakan Legal dari state saat ini
58             for (Move move : current.board.getPossibleMoves()) {
59                 Board newBoard = current.board.movePiece(move.getPiece(), move.getDirection(), move.getSteps());
60                 // Salin jalur lama dan tambahkan langkah baru ke jalur
61                 List<Move> newPath = new ArrayList<>(current.path);
62                 newPath.add(move);
63
64                 // Menambahkan node baru ke antrian
65                 queue.add(new Node(newBoard, newPath, heuristic.calculate(newBoard)));
66             }
67         }
68
69         // Kasus tidak ada solusi
70         printStats(false);
71         return null;
72     }
73 }
```

4.8 AstarSearch.java

```
● ● ●
1 package algorithms;
2
3 import java.util.*;
4
5 import model.Board;
6 import model.Move;
7 import heuristics.Heuristic;
8
9 public class AstarSearch extends SearchAlgorithm {
10     private final Heuristic heuristic;
11
12     public AstarSearch(Heuristic heuristic) {
13         this.heuristic = heuristic;
14     }
15
16     // Node yang merepresentasikan satu state dalam pencarian
17     private static class Node {
18         Board board;
19         List<Move> path;
20         int cost; // Biaya dari root ke node ini ( $g(n)$ )
21         int heuristic; // Perkiraan biaya ke goal ( $h(n)$ )
22         int total;
23
24         Node(Board board, List<Move> path, int cost, int heuristic) {
25             this.board = board;
26             this.path = path;
27             this.cost = cost;
28             this.heuristic = heuristic;
29             this.total = cost + heuristic;
30         }
31     }
32
33     @Override
34     public List<Move> solve(Board initialBoard) {
35         startTime = System.currentTimeMillis(); // Inisialisasi waktu pencarian
36         nodesExplored = 0; // Inisialisasi jumlah node yang dieksplorasi
37
38         // PriorityQueue berdasarkan total cost  $f(n) = g(n) + h(n)$ 
39         PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(n -> n.total));
40         // Menyimpan cost minimum untuk setiap state
41         Map<String, Integer> costMap = new HashMap<>();
42
43         // Menambahkan node awal
44         queue.add(new Node(initialBoard, new ArrayList<>(), 0, heuristic.calculate(initialBoard)));
45         costMap.put(initialBoard.getStateString(), 0);
46
47         while (!queue.isEmpty()) {
48             Node current = queue.poll(); // Mengambil node dengan  $f(n)$  terendah
49             nodesExplored++;
50
51             // Kasus goal state tercapai
52             if (current.board.isSolved()) {
53                 printStats(true);
54                 return current.path;
55             }
56
57             // Skip jika kita sudah menemukan jalur yang lebih murah ke state ini
58             if (current.cost > costMap.getOrDefault(current.board.getStateString(), Integer.MAX_VALUE)) {
59                 continue;
60             }
61
62             // Eksplorasi semua langkah legal dari state saat ini
63             for (Move move : current.board.getPossibleMoves()) {
64                 Board newBoard = current.board.movePiece(move.getPiece(), move.getDirection(), move.getSteps());
65                 int newCost = current.cost + move.getSteps();
66                 String state = newBoard.getStateString();
67
68                 // Jika ditemukan jalur lebih murah ke state ini, simpan dan tambahkan ke queue
69                 if (newCost < costMap.getOrDefault(state, Integer.MAX_VALUE)) {
70                     List<Move> newPath = new ArrayList<>(current.path);
71                     newPath.add(move);
72
73                     queue.add(new Node(newBoard, newPath, newCost, heuristic.calculate(newBoard)));
74                     costMap.put(state, newCost); // Memperbarui cost minimum ke state ini
75                 }
76             }
77         }
78
79         // Kasus tidak ada solusi
80         printStats(false);
81         return null;
82     }
83 }
```

4.9 Heuristic.java

```
● ● ●
1 package heuristics;
2
3 import model.Board;
4
5 public interface Heuristic {
6     int calculate(Board board);
7     String getName();
8 }
```

4.10 BlockingHeuristic.java

```
● ● ●
1 package heuristics;
2
3 import model.Board;
4 import model.Piece;
5
6 import java.util.HashSet;
7 import java.util.Set;
8
9 // Heuristik berdasarkan jumlah dan mobilitas piece yang menghalangi primary piece
10 public class BlockingHeuristic implements Heuristic {
11     @Override
12     public int calculate(Board board) {
13         Piece main = board.getPrimaryPiece();
14         Set<Character> blockers = new HashSet<>();
15
16         if (main.isHorizontal()) {
17             int row = main.getRow();
18             int col = main.getCol() + main.getLength();
19
20             // Menelusuri ke kanan sampai exit
21             while (col < board.getCols()) {
22                 char cell = board.getGrid()[row][col];
23                 if (cell != '.' && cell != 'K' && cell != main.getId()) {
24                     blockers.add(cell);
25                 }
26                 col++;
27             }
28         } else {
29             int col = main.getCol();
30             int row = main.getRow() + main.getLength();
31
32             // Menelusuri ke bawah sampai exit
33             while (row < board.getRows()) {
34                 char cell = board.getGrid()[row][col];
35                 if (cell != '.' && cell != 'K' && cell != main.getId()) {
36                     blockers.add(cell);
37                 }
38                 row++;
39             }
40         }
41
42         // Menghitung jumlah blocker tanpa penalti tambahan
43         return blockers.size();
44     }
45
46     @Override
47     public String getName() {
48         return "Blocking Heuristic";
49     }
50 }
```

4.11 MobilityHeuristic.java

```
● ● ●
1 package heuristics;
2
3 import model.Board;
4 import model.Piece;
5
6 // Heuristik yang mengukur jumlah piece yang tidak bisa bergerak
7 public class MobilityHeuristic implements Heuristic {
8     @Override
9     public int calculate(Board board) {
10         int immobileCount = 0;
11
12         // Menghitung jumlah piece yang tidak bisa bergerak sama sekali
13         for (Piece piece : board.getPieces()) {
14             if (!canMove(board, piece)) {
15                 immobileCount++;
16             }
17         }
18
19         return immobileCount; // Semakin banyak yang tidak bisa bergerak, semakin buruk
20     }
21
22     // Mengecek apakah sebuah piece bisa bergerak ke arah mana pun
23     private boolean canMove(Board board, Piece piece) {
24         char[][] grid = board.getGrid();
25         int row = piece.getRow();
26         int col = piece.getCol();
27         int len = piece.getLength();
28
29         if (piece.isHorizontal()) {
30             // Cek kiri
31             if (col > 0 && grid[row][col - 1] == '.') return true;
32             // Cek kanan
33             if (col + len < board.getCols() && grid[row][col + len] == '.') return true;
34         } else {
35             // Cek atas
36             if (row > 0 && grid[row - 1][col] == '.') return true;
37             // Cek bawah
38             if (row + len < board.getRows() && grid[row + len][col] == '.') return true;
39         }
40
41         return false; // Tidak ada ruang untuk bergerak
42     }
43
44     @Override
45     public String getName() {
46         return "Mobility Heuristic";
47     }
48 }
```

4.12 InputHandler.java

```
 1 package handler;
 2
 3 import java.io.BufferedReader;
 4 import java.io.FileReader;
 5 import java.util.ArrayList;
 6 import java.util.HashMap;
 7 import java.util.HashSet;
 8 import java.util.LinkedList;
 9 import java.util.List;
10 import java.util.Map;
11 import java.util.Queue;
12 import java.util.Set;
13
14 import model.Board;
15 import model.Piece;
16
17 public class InputHandler {
18     // Constants for validation
19     private static final int MIN_PIECE_SIZE = 2;
20     private static final char PRIMARY_PIECE = 'P';
21     private static final char EMPTY_CELL = '.';
22
23     // Kelas exception untuk kesalahan validasi puzzle
24     public static class PuzzleValidationException extends Exception {
25         public PuzzleValidationException(String message) {
26             super(message);
27         }
28     }
29
30     // Method utama untuk membaca input file
31     public Board readInput(String filePath) throws Exception {
32         BufferedReader reader = new BufferedReader(new FileReader(filePath));
33
34         // Membaca dimensi papan (baris dan kolom)
35         String[] dimensions = reader.readLine().trim().split(" ");
36         if (dimensions.length != 2) {
37             throw new PuzzleValidationException("First line must contain exactly two numbers (rows and columns)");
38         }
39
40         int rows, cols;
41         try {
42             rows = Integer.parseInt(dimensions[0]);
43             cols = Integer.parseInt(dimensions[1]);
44             if (rows <= 0 || cols <= 0) {
45                 throw new PuzzleValidationException("Rows and columns must be positive numbers");
46             }
47         } catch (NumberFormatException e) {
48             throw new PuzzleValidationException("Rows and columns must be valid numbers");
49         }
50
51         // Membaca jumlah piece
52         String pieceCountLine = reader.readLine();
53         if (pieceCountLine == null) {
54             throw new PuzzleValidationException("Missing piece count line");
55         }
56
57         int numPieces;
58         try {
59             numPieces = Integer.parseInt(pieceCountLine.trim());
60             if (numPieces < 0) {
61                 throw new PuzzleValidationException("Piece count must be a non-negative number");
62             }
63         } catch (NumberFormatException e) {
64             throw new PuzzleValidationException("Piece count must be a valid number");
65         }
66     }
}
```

```

1 // Membaca grid
2 char[][] grid = new char[rows][cols];
3 List<Piece> pieces = new ArrayList<>();
4 int exitRow = -1, exitCol = -1;
5
6 for (int i = 0; i < rows; i++) {
7     String line = reader.readLine();
8     if (line == null) {
9         throw new PuzzleValidationException("Not enough rows in the input file");
10    }
11    line = line.trim();
12
13    // Pemeriksaan untuk 'K' di kanan
14    if (line.length() > cols && line.charAt(cols) == 'K') {
15        exitRow = i;
16        exitCol = cols;
17        line = line.substring(0, cols);
18    }
19
20    // Validasi panjang baris
21    if (line.length() != cols) {
22        throw new PuzzleValidationException(
23            String.format("Row %d has %d characters, expected %d", i+1, line.length(), cols));
24    }
25
26    for (int j = 0; j < cols; j++) {
27        grid[i][j] = line.charAt(j);
28        if (grid[i][j] == 'K') {
29            exitRow = i;
30            exitCol = j;
31        }
32    }
33 }
34 reader.close();
35
36 // Tidak ditemukan pintu keluar
37 if (exitRow == -1 || exitCol == -1) {
38     exitRow = rows / 2;
39     exitCol = cols;
40 }
41
42 validatePuzzle(grid, numPieces);
43
44 // Identifikasi piece
45 boolean[][] visited = new boolean[rows][cols];
46 for (int i = 0; i < rows; i++) {
47     for (int j = 0; j < cols; j++) {
48         if (grid[i][j] != EMPTY_CELL && grid[i][j] != 'K' && !visited[i][j]) {
49             char id = grid[i][j];
50             boolean isPrimary = id == PRIMARY_PIECE;
51             int size = 1;
52             String orientation;
53
54             if (j + 1 < cols && grid[i][j + 1] == id) {
55                 orientation = "horizontal";
56                 while (j + size < cols && grid[i][j + size] == id) {
57                     size++;
58                 }
59             } else {
60                 orientation = "vertical";
61                 while (i + size < rows && grid[i + size][j] == id) {
62                     size++;
63                 }
64             }
65         }
66     }
67 }

```

```

1         pieces.add(new Piece(id, i, j, size, orientation, isPrimary));
2
3     for (int k = 0; k < size; k++) {
4         if (orientation.equals("horizontal")) {
5             visited[i][j + k] = true;
6         } else {
7             visited[i + k][j] = true;
8         }
9     }
10    }
11 }
12
13 return new Board(rows, cols, grid, pieces, exitRow, exitCol);
14 }
15 }
16
17 // Validasi konfigurasi puzzle
18 private void validatePuzzle(char[][] grid, int expectedPieceCount) throws PuzzleValidationException {
19     if (grid == null || grid.Length == 0 || grid[0].Length == 0) {
20         throw new PuzzleValidationException("Invalid grid: Grid cannot be null or empty");
21     }
22
23     Map<Character, List<int[]>> piecePositions = new HashMap<>();
24     int rows = grid.Length;
25     int cols = grid[0].Length;
26
27     for (int i = 0; i < rows; i++) {
28         for (int j = 0; j < cols; j++) {
29             char cell = grid[i][j];
30             if (cell != EMPTY_CELL && cell != 'K') {
31                 piecePositions.computeIfAbsent(cell, k -> new ArrayList<>())
32                     .add(new int[]{i, j});
33             }
34         }
35     }
36
37     // Validasi keberadaan potongan utama
38     if (!piecePositions.containsKey(PRIMARY_PIECE)) {
39         throw new PuzzleValidationException("Invalid configuration: Primary piece 'P' is missing");
40     }
41
42     // Validasi ukuran minimum setiap potongan
43     for (Map.Entry<Character, List<int[]>> entry : piecePositions.entrySet()) {
44         if (entry.getValue().size() < MIN_PIECE_SIZE) {
45             throw new PuzzleValidationException(
46                 String.format("Invalid piece '%c': Each piece must be at least %d cells",
47                             entry.getKey(), MIN_PIECE_SIZE));
48         }
49     }
50
51     // Validasi jumlah potongan
52     int actualPieceCount = piecePositions.size() - 1;
53     if (actualPieceCount != expectedPieceCount) {
54         throw new PuzzleValidationException(
55             String.format("Invalid piece count: Expected %d pieces (excluding primary piece), found %d",
56                         expectedPieceCount, actualPieceCount));
57     }
58
59     // Validasi apakah setiap potongan saling terhubung
60     for (Map.Entry<Character, List<int[]>> entry : piecePositions.entrySet()) {
61         if (!isPieceConnected(entry.getValue())) {
62             throw new PuzzleValidationException(
63                 String.format("Invalid piece '%c': All cells must be connected", entry.getKey()));
64         }
65     }
66
67     // Validasi

```

```

1      // Validasi apakah ada potongan yang saling tumpang tindih
2      if (hasOverlappingPieces(piecePositions)) {
3          throw new PuzzleValidationException("Invalid configuration: Pieces cannot overlap");
4      }
5  }
6
7  // Mengecek apakah semua sel dalam satu potongan saling terhubung
8  private boolean isPieceConnected(List<int[]> positions) {
9      if (positions.size() < 2) return true;
10
11     Set<String> visited = new HashSet<>();
12     Queue<int[]> queue = new LinkedList<>();
13     queue.add(positions.get(0));
14     visited.add(positions.get(0)[0] + "," + positions.get(0)[1]);
15
16     int[][] directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
17
18     while (!queue.isEmpty()) {
19         int[] current = queue.poll();
20
21         for (int[] dir : directions) {
22             int newRow = current[0] + dir[0];
23             int newCol = current[1] + dir[1];
24             String key = newRow + "," + newCol;
25
26             if (!visited.contains(key)) {
27                 for (int[] pos : positions) {
28                     if (pos[0] == newRow && pos[1] == newCol) {
29                         queue.add(pos);
30                         visited.add(key);
31                         break;
32                     }
33                 }
34             }
35         }
36     }
37
38     return visited.size() == positions.size();
39 }
40
41 // Mengecek apakah ada dua potongan yang menempati sel yang sama
42 private boolean hasOverlappingPieces(Map<Character, List<int[]>> piecePositions) {
43     Set<String> allPositions = new HashSet<>();
44     for (List<int[]> positions : piecePositions.values()) {
45         for (int[] pos : positions) {
46             String key = pos[0] + "," + pos[1];
47             if (!allPositions.add(key)) {
48                 return true;
49             }
50         }
51     }
52     return false;
53 }
54 }
```

BAB 5

EKSPERIMEN

5.1 Test Case 1

Input:

```
3 3  
2  
PPAK  
BBA  
...
```

Output:

```
Enter puzzle file name: input1.txt  
  
Select algorithm:  
1. Uniform Cost Search  
2. Greedy Best First Search  
3. A* Search  
Choice: 1  
  
Solving puzzle...  
  
==> Search Statistics ==>  
Solution found: true  
Nodes explored: 3  
Time taken: 0,012 seconds  
  
Solution found in 2 moves:  
Initial board:  
PPAK  
BBA  
...  
  
Move 1: A down 1  
PP.K  
BBA  
..A  
  
Move 2: P right 1  
.PPK  
BBA  
..A  
  
Move 3: P right 2  
...K  
BBA  
..A
```

5.2 Test Case 2

Input:

```
6 6  
9  
.ADD.E  
.ABC.E  
PPBC.FK  
IIGH.F  
..GH..  
.....
```

Output:

```
Enter puzzle file name: input2.txt

Select algorithm:
1. Uniform Cost Search
2. Greedy Best First Search
3. A* Search
Choice: 2

Select heuristic:
1. Mobility
2. Blocking Pieces
Choice: 1

Solving puzzle...

==> Search Statistics <==
Solution found: true
Nodes explored: 2237
Time taken: 0,103 seconds

Solution found in 28 moves:
Initial board:
.A.DD.E
.A.BC.E
PPBC.FK
II.II.G.H.F
..G.H..
....G.

Move 1: G down 1
.A.DD.E
.A.BC.E
PPBC.FK
II.II.H.F
..G.H..
...G.H.

Move 2: F down 1
.A.DD.E
.A.BC.E
PPBC..K
II.II.H.F
..G.H.F
...G.H.

Move 3: H down 1
.A.DD.E
.A.BC.E
PPBC..K
II.II...F
..G.H.F
...G.H.

Move 4: E down 1
.A.DD..
.A.BC.E
PPBC.EK
II.II...F
..G.H.F
...G.H.

Move 5: D right 1
.A.DD.
.A.BC.E
PPBC.EK
II.II...F
..G.H.F
...G.H.

Move 6: B up 1
.A.BDD.
.A.BC.E
PP.C.EK
II.II...F
..G.H.F
...G.H.
```

Move 7: F down 1

.ABDD.
.ABC.E
PP.C.EK
II....
..GH.F
..GH.F

Move 8: C down 1

.ABDD.
.AB..E
PP.C.EK
II.C..
..GH.F
..GH.F

Move 9: D right 1

.AB.DD
.AB..E
PP.C.EK
II.C..
..GH.F
..GH.F

Move 10: C up 2

.ABCD
.ABC.E
PP...EK
II....
..GH.F
..GH.F

Move 11: I right 3

.ABCDD
.ABC.E
PP...EK
...II.
..GH.F
..GH.F

Move 12: P right 3

.ABCDD
.ABC.E
...PPEK
...II.
..GH.F
..GH.F

Move 13: A down 3

..BCDD
..BC.E
...PPEK
.A.II.
.AGH.F
..GH.F

Move 14: P left 3

..BCDD
..BC.E
PP...EK
.A.II.
.AGH.F
..GH.F

Move 15: C down 1

..B.DD
..BC.E
PP.C.EK
.A.II.
.AGH.F
..GH.F

Move 16: A down 1

..B.DD
..BC.E
PP.C.EK
...II.
.AGH.F
.AGH.F

```
Move 17: I left 3
..B.DD
..BC.E
PP.C.EK
II.....
.AGH.F
.AGH.F

Move 18: D left 1
..BDD.
..BC.E
PP.C.EK
II.....
.AGH.F
.AGH.F

Move 19: I right 4
..BDD.
..BC.E
PP.C.EK
....II
.AGH.F
.AGH.F

Move 20: B down 1
...DD.
...BC.E
PPBC.EK
....II
.AGH.F
.AGH.F

Move 21: D left 3
DD.....
..BC.E
PPBC.EK
....II
.AGH.F
.AGH.F

Move 22: A up 1
DD.....
..BC.E
PPBC.EK
.A...II
.AGH.F
..GH.F

Move 23: B up 1
DDB... .
..BC.E
PP.C.EK
.A..II
.AGH.F
..GH.F

Move 24: C up 1
DDBC.. .
..BC.E
PP...EK
.A..II
.AGH.F
..GH.F

Move 25: I left 1
DDBC.. .
..BC.E
PP...EK
.A.II..
.AGH.F
..GH.F

Move 26: F up 1
DDBC.. .
..BC.E
PP...EK
.A.IIF
.AGH.F
..GH..
```

```
Move 27: E up 1
```

```
DDBC.E
```

```
..BC.E
```

```
PP....K
```

```
.A.IIF
```

```
.AGH.F
```

```
..GH..
```

```
Move 28: P right 4
```

```
DDBC.E
```

```
..BC.E
```

```
....PPK
```

```
.A.IIF
```

```
.AGH.F
```

```
..GH..
```

```
Move 29: P right 2
```

```
DDBC.E
```

```
..BC.E
```

```
.....K
```

```
.A.IIF
```

```
.AGH.F
```

```
..GH..
```

5.3 Test Case 3

Input:

```
6 6
7
..BB..
..AACC
PPDD..K
EE.F..
...FGG
.....
```

Output:

```
Enter puzzle file name: input3.txt
```

```
Select algorithm:
```

1. Uniform Cost Search
2. Greedy Best First Search
3. A* Search

```
Choice: 1
```

```
Solving puzzle...
```

```
==> Search Statistics ==>
```

```
Solution found: false
```

```
Nodes explored: 92393
```

```
Time taken: 0,380 seconds
```

```
No solution found
```

5.4 Test Case 4

Input:

```
6 6
11
..ABBB
E.ADCC
EPPD..K
.F.DGG
JFHH.I
JLLL.I
```

Output:

```
Move 167: L left 3
```

```
BBBD.I
```

```
ECCD.I
```

```
EPPD..K
```

```
JFAGG.
```

```
JFA.HH
```

```
LLL...
```

```
Move 168: G right 1
```

```
BBBD.I
```

```
ECCD.I
```

```
EPPD..K
```

```
JFA.GG
```

```
JFA.HH
```

```
LLL...
```

```
Move 169: D down 3
```

```
BBB..I
```

```
ECC..I
```

```
EPP...K
```

```
JFADGG
```

```
JFADHH
```

```
LLLD..
```

```
Move 170: P right 3
```

```
BBB..I
```

```
ECC..I
```

```
E...PPK
```

```
JFADGG
```

```
JFADHH
```

```
LLLD..
```

```
Move 171: P right 2
```

```
BBB..I
```

```
ECC..I
```

```
E.....K
```

```
JFADGG
```

```
JFADHH
```

```
LLLD..
```

5.5 Test Case 5

Input:

```
6 6
9
ABCCE.
ABDDE.
PPFH..K
..FHI.
..GHI.
...G...
```

Output:

```
Enter puzzle file name: input5.txt
```

```
Select algorithm:
```

- 1. Uniform Cost Search
- 2. Greedy Best First Search
- 3. A* Search

```
Choice: 3
```

```
Select heuristic:
```

- 1. Mobility
- 2. Blocking Pieces

```
Choice: 1
```

```
Solving puzzle...
```

```
==> Search Statistics ==>
```

```
Solution found: true
```

```
Nodes explored: 4402
```

```
Time taken: 0,298 seconds
```

```
Solution found in 20 moves:
```

```
Initial board:
```

```
ABCCE.  
ABDDE.  
PPFH..K  
.FH.  
.GHI.  
.G...
```

```
Move 1: E down 1
```

```
ABCC..  
ABDDE.  
PPFHE.K  
.FH.  
.GHI.  
.G...
```

```
Move 2: I down 1
```

```
ABCC..  
ABDDE.  
PPFHE.K  
.FH.  
.GHI.  
.G.I.
```

```
Move 3: C right 1
```

```
AB.CC.  
ABDDE.  
PPFHE.K  
.FH.  
.GHI.  
.G.I.
```

```
Move 4: E down 1
```

```
AB.CC.  
ABDD..  
PPFHE.K  
.FHE.  
.GHI.  
.G.I.
```

```
Move 5: D right 1
```

```
AB.CC.  
AB.DD.  
PPFHE.K  
.FHE.  
.GHI.  
.G.I.
```

```
Move 6: F up 2
```

```
AB.FCC.  
AB.FDD.  
PP.HE.K  
.HE.  
.GHI.  
.G.I.
```

Move 7: H down 1

ABFCC.
ABFDD.
PP..E.K
...HE.
..GHI.
.GHI.

Move 8: P right 2

ABFCC.
ABFDD.
..PPE.K
...HE.
..GHI.
.GHI.

Move 9: A down 3

.BFCC.
.BFDD.
..PPE.K
A..HE
A.GHI
.GHI.

Move 10: B down 3

..FCC.
..FDD.
..PPE.K
AB.HE
ABGHI
.GHI.

Move 11: P left 2

..FCC.
..FDD.
PP..E.K
AB.HE.
ABGHI.
.GHI.

Move 12: F down 1

...CC.
...FDD.
PPF.E.K
AB.HE.
ABGHI.
.GHI.

Move 13: C left 1

...CC..
...FDD.
PPF.E.K
AB.HE.
ABGHI.
.GHI.

Move 14: F down 1

...CC..
...DD.
PPF.E.K
ABFHE.
ABGHI.
.GHI.

Move 15: D left 1

...CC..
...**DD**..
PPF.E.K
ABFHE.
ABGHI.
.GHI.

Move 16: E up 2

...CCE.
...DDE.
PPF...K
ABFH..
ABGHI.
.GHI.

```

Move 17: D left 2
..CCE.
DD..E.
PPF...K
ABFH..
ABGHI.
..GHI.

Move 18: C left 2
CC..E.
DD..E.
PPF...K
ABFH..
ABGHI.
..GHI.

Move 19: F up 2
CCF.E.
DDF.E.
PP....K
AB.H..
ABGHI.
..GHI.

Move 20: P right 4
CCF.E.
DDF.E.
....PPK
AB.H..
ABGHI.
..GHI.

Move 21: P right 2
CCF.E.
DDF.E.
.....K
AB.H..
ABGHI.
..GHI.

```

5.6 Test Case 6

Input:

```

6 6
9
ABCCE.
ABDDE.
PPFH..K
..FHI.
..GHI.
..G...

```

Output:

```

Enter puzzle file name: input6.txt

Select algorithm:
1. Uniform Cost Search
2. Greedy Best First search
3. A* Search
Choice: 3

Select heuristic:
1. Mobility
2. Blocking Pieces
Choice: 2

Solving puzzle...

*** Search Statistics ***
Solution found: true
Nodes explored: 3953
Time taken: 0,269 seconds

Solution found in 19 moves:
Initial board:
ABCCE.
ABDDE.
PPFH..K
..FHI.
..GHI.
..G...

Move 1: H down 1
ABCCE.
ABDDE.
PPF...K
..FHI.
..GHI.
..GH...

```

Move 2: I down 1

ABCCE.
ABDDE.
PPF...K
..FH..
..GHI.
..GHI.

Move 3: E down 2

ABCC..
ABDD..
PPF.E.K
..FHE.
..GHI.
..GHI.

Move 4: C right 1

AB.CC.
ABDD..
PPF.E.K
..FHE.
..GHI.
..GHI.

Move 5: D right 1

AB.CC.
AB.DD.
PPF.E.K
..FHE.
..GHI.
..GHI.

Move 6: F up 2

ABFCC.
ABFDD.
PP..E.K
...HE.
..GHI.
..GHI.

Move 7: P right 2

ABFCC.
ABFDD.
..PPE.K
...HE.
..GHI.
..GHI.

Move 8: A down 3

.BFCC.
.BFDD.
..PPE.K
A..HE.
A.GHI.
..GHI.

Move 9: B down 3

..FCC.
..FDD.
..PPE.K
AB.HE.
ABGHI.
..GHI.

Move 10: P left 2

..FCC.
..FDD.
PP..E.K
AB.HE.
ABGHI.
..GHI.

Move 11: F down 1

...CC.
...FDD.
PPF.E.K
AB.HE.
ABGHI.
..GHI.

Move 12: C left 1

..CC..
..FDD.
PPF.E.K
AB.HE.
ABGHI.
..GHI.

Move 13: F down 1

..CC..
...DD.
PPF.E.K
ABFHE.
ABGHI.
..GHI.

Move 14: D left 1

..CC..
..DD..
PPF.E.K
ABFHE.
ABGHI.
..GHI.

Move 15: E up 2

..CCE.
..DDE.
PPF...K
ABFH..
ABGHI.
..GHI.

Move 16: C left 2

CC..E.
..DDE.
PPF...K
ABFH..
ABGHI.
..GHI.

Move 17: D left 2

CC..E.
DD..E.
PPF...K
ABFH..
ABGHI.
..GHI.

Move 18: F up 2

CCF.E.
DDF.E.
PP....K
AB.H..
ABGHI.
..GHI.

Move 19: P right 4

CCF.E.
DDF.E.
....PPK
AB.H..
ABGHI.
..GHI.

Move 20: P right 2

CCF.E.
DDF.E.
.....K
AB.H..
ABGHI.
..GHI.

BAB 6

ANALISIS KOMPLEKSITAS

6.1 Analisis Kompleksitas Algoritma *Uniform Cost Search*

- b = *branching factor* (rata-rata banyaknya langkah/aksi legal per *state*)
- C^* = biaya solusi optimal (jumlah langkah minimal dari *state* awal ke goal)
- d = kedalaman solusi optimal (jumlah langkah dalam path ke goal)
- n = jumlah total node yang dieksplorasi hingga mencapai solusi

UCS mengeksplorasi semua node dengan total cost $g(n)$ lebih kecil atau sama dengan C^* sebelum mencapai solusi optimal. Dalam kasus terburuk, UCS akan memeriksa semua kemungkinan kombinasi langkah hingga total cost C^* . Maka, kompleksitas waktunya adalah $O(b^{C^*})$ (karena bisa mengeksplorasi seluruh *state* dengan $cost \leq C^*$).

6.2 Analisis Kompleksitas Algoritma *Greedy Best First Search*

- b = *branching factor* (jumlah rata-rata langkah legal dari suatu *state*)
 - d = kedalaman solusi (jumlah langkah dari initial *state* ke goal)
 - $h(n)$ = nilai heuristik (estimasi biaya dari node ke goal)
 - n = jumlah total node yang dieksplorasi (bervariasi tergantung akurasi heuristik)
- GBFS memprioritaskan eksplorasi node yang tampak paling dekat ke goal, berdasarkan nilai heuristik $h(n)$, bukan total cost. Dalam kasus terburuk, heuristik bisa menyesatkan pencarian ke jalur yang salah, menyebabkan eksplorasi banyak node yang sebenarnya tidak menuju solusi. Maka, kompleksitas waktunya adalah $O(b^m)$, dengan m adalah kedalaman maksimum pencarian sebelum goal ditemukan.

6.3 Analisis Kompleksitas Algoritma *A* Search*

- b = *branching factor* (jumlah rata-rata langkah legal dari suatu *state*)
- d = kedalaman solusi optimal (jumlah langkah optimal ke goal)
- n = jumlah node yang di-expand
- $f(n) = g(n) + h(n)$ = total cost dari node awal ke node n , dengan $g(n)$ adalah biaya aktual dari start ke n dan $h(n)$ adalah perkiraan biaya dari n ke goal (heuristik)

Dalam kasus terburuk, A* bisa mengunjungi semua node yang memiliki $f(n) \leq f^*$ dengan f^* adalah cost dari solusi optimal. Maka, kompleksitas waktunya adalah $O(b^d)$.

BAB 7

IMPLEMENTASI BONUS

7.1 Heuristik Blocking

BlockingHeuristic adalah pendekatan heuristik yang mengevaluasi *state* papan permainan dengan menghitung jumlah *blocker* atau *piece* penghalang yang berada di jalur utama (*primary piece*) menuju titik tujuan (*goal*). Heuristik ini bekerja dengan mengidentifikasi posisi *primary piece* dan memeriksa sel-sel di jalur lurus ke arah pintu keluar untuk mencari *piece* lain yang menghalangi pergerakan langsung ke tujuan. Setiap *piece* yang ditemukan di jalur tersebut, selama bukan bagian dari ruang kosong, titik tujuan, atau *primary piece* itu sendiri, dihitung sebagai *blocker* yang unik menggunakan struktur data *HashSet*. Hasil dari heuristik ini adalah jumlah *blocker* yang ditemukan, yang mencerminkan tingkat kesulitan atau hambatan yang harus diatasi untuk mencapai *goal*.

7.2 Heuristik Mobility

MobilityHeuristic adalah pendekatan heuristik yang mengevaluasi keadaan papan permainan berdasarkan jumlah *piece* yang tidak memiliki ruang gerak sama sekali. Heuristik ini menghitung banyaknya *piece* yang sepenuhnya terblokir dan tidak dapat bergerak ke arah mana pun dengan memeriksa apakah terdapat sel kosong di salah satu sisi *piece* sesuai orientasinya. Untuk setiap *piece* pada papan, fungsi ini menggunakan logika pengecekan sederhana, yaitu jika tidak ada ruang kosong di kiri atau kanan untuk *piece* horizontal, atau di atas atau bawah untuk *piece* vertikal, maka *piece* tersebut dianggap tidak bisa bergerak. Semakin banyak *piece* yang tidak dapat bergerak, semakin buruk nilai keadaan papan yang ditunjukkan oleh heuristik ini, karena rendahnya mobilitas menunjukkan tingkat kemacetan yang tinggi pada konfigurasi permainan.

LAMPIRAN

1. Repository GitHub

Berikut adalah pranala ke repository GitHub yang berisi kode sumber dan dokumentasi proyek ini:

https://github.com/JethroJNS/Tucil3_13523081.git

2. Tabel Evaluasi

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	✓	

Catatan: Program belum berhasil meng-*handle* kasus pintu keluar di atas, kiri, dan bawah file