

LAPORAN TUGAS KECIL 3
Penyelesaian Puzzle Rush Hour Menggunakan Algoritma
Pathfinding

Disusun untuk Memenuhi Tugas Kecil 3 Mata Kuliah Strategi Algoritma IF2211

Dosen Pengampu:

Dr. Ir. Rinaldi Munir, M.T.



Albertus Christian Poandy 13523077

Karol Yangqian Poetrachya 13523093

Kelas K2

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2025

DAFTAR ISI

DAFTAR ISI.....	2
BAB I	
DESKRIPSI MASALAH.....	4
1.1. Deskripsi Masalah.....	4
1.2. Algoritma.....	5
1.3. Analisis Algoritma.....	8
1.4. Pseudocode.....	10
BAB II	
IMPLEMENTASI.....	13
2.1. Implementasi Piece.java.....	13
2.2. Implementasi PrimaryPiece.java.....	13
2.3. Implementasi Move.java.....	14
2.4. Implementasi State.java.....	14
2.5. Implementasi Heuristic.java.....	14
2.6. Implementasi Reader.java.....	15
2.7. Implementasi Solver.java.....	15
2.8. Implementasi MainController.java.....	16
BAB III	
SOURCE CODE.....	17
3.1. Repository Program.....	17
3.2. Source Code Program.....	17
3.2.1. Piece.java.....	17
3.2.2. PrimaryPiece.java.....	19
3.2.3. Move.java.....	20
3.2.4. State.java.....	21
3.2.5. Heuristic.java.....	23
3.2.6. Reader.java.....	25
3.2.7. Solver.java.....	31
3.2.8. MainController.java.....	33
BAB IV	
PERCOBAAN DAN ANALISIS.....	56
4.1. Percobaan.....	56
4.1.1. Test Case 1: UCS dengan Solusi di Kanan.....	56
4.1.2. Test Case 2: UCS dengan Solusi di Bawah.....	56
4.1.3. Test Case 3: UCS dengan Solusi Panjang.....	57
4.1.4. Test Case 4: UCS Tanpa Solusi.....	57
4.1.5. Test Case 1: GBFS dengan Heuristik Manhattan.....	57
4.1.6. Test Case 2: GBFS dengan Heuristik Blocking Piece Count.....	58

4.1.7. Test Case 3: GBFS dengan Solusi Panjang.....	58
4.1.8. Test Case 4: GBFS Tanpa Solusi.....	59
4.1.9. Test Case 1: A* dengan Heuristik Manhattan.....	59
4.1.10. Test Case 2: A* dengan Heuristik Blocking Piece Count.....	60
4.1.11. Test Case 3: A* dengan Solusi Panjang.....	60
4.1.12. Test Case 4: A* Tanpa Solusi.....	61
4.2. Analisis.....	61
4.2.1. Kompleksitas Algoritma.....	61
4.2.2. Analisis Percobaan.....	63
LAMPIRAN.....	65
REFERENSI.....	66

BAB I

DESKRIPSI MASALAH

1.1. Deskripsi Masalah

Rush Hour adalah permainan *puzzle* berbasis kisi atau *grid* dengan tujuan untuk mengeluarkan mobil utama dengan jumlah langkah seminimal mungkin dari papan yang merepresentasikan kemacetan mobil. Setiap mobil dapat menempati 2 atau lebih *cell* dalam satu garis yang sama dan hanya dapat bergerak maju atau mundur sesuai orientasinya. Beberapa komponen dalam permainan Rush Hour meliputi:

1. Papan

Papan adalah tempat permainan dimainkan yang terdiri atas *cell* yang merepresentasikan sebuah titik pada papan. Sebelum permainan dimulai, setiap *piece* atau mobil dikonfigurasi untuk menempati beberapa *cell* secara berurutan pada lokasi dan dengan orientasi tertentu (horizontal atau vertikal).

2. Piece

Piece merepresentasikan sebuah kendaraan yang memiliki posisi, ukuran, dan orientasi pada papan. Orientasi dari sebuah *piece* adalah salah satu dari vertikal atau horizontal. Setiap *piece* hanya dapat bergerak satu arah sesuai orientasinya. Ketika digerakkan, *piece* tidak boleh melewati atau menembus *piece* lain atau tembok. Dalam permainan sebenarnya, hanya *primary piece* yang dapat keluar dari papan melalui pintu keluar.

3. Primary Piece

Primary piece adalah kendaraan utama satu-satunya yang harus dikeluarkan dari papan sebagai tujuan dari permainan. Biasanya, *primary piece* ditandai dengan warna mencolok seperti merah.

4. Pintu keluar

Ketika *primary piece* digerakkan melalui atau mencapai pintu keluar, permainan dikatakan selesai.

5. Gerakan

Gerakan adalah sebuah aktivitas *piece* bergeser dalam papan untuk berpindah lokasi *cell*. Gerakan *piece* yang valid adalah sesuai orientasinya, yakni atas-bawah untuk *piece* vertikal dan kiri-kanan untuk *piece* horizontal. Setiap gerakan tidak boleh menyebabkan *piece* saling tumpang tindih.

1.2. Algoritma

1.2.1. *Uniform Cost Search* (UCS)

1. Inisialisasi:

- Buat sebuah struktur data *PriorityQueue* yang dinamakan *openSet*, yang akan mengurutkan *state-state* (node-node) yang ada berdasarkan nilai $g(n)$ terkecil.
- Buat sebuah struktur data *Set* yang dinamakan *closedSet* untuk menyimpan *state-state* yang sudah pernah dieksplorasi.
- Buat state awal dengan $g(\text{initialState}) = 0$, lalu masukkan ke dalam *openSet* dan *closedSet*

2. Loop Utama:

Selama *openSet* tidak kosong:

- Ambil (dan hapus) state dari *openSet* yang memiliki nilai $g(n)$ terkecil, beri nama *currentState*.
- Cek posisi *primary piece* pada *currentState*, jika telah berada pada pintu keluar, maka posisi telah ditemukan. Jalur untuk mencapai *state* ini pasti merupakan jalur optimal. Hentikan pencarian dan kembalikan jalur yang ditemukan
- Jika posisi *primary piece* tidak berada pada pintu keluar: bangkitkan semua node (state) suksesor yang valid dari *currentState*. *State* suksesor

yang valid dari *currentState* adalah seluruh kemungkinan melakukan satu langkah pergerakan pada satu buah *piece* pada *board*.

- Untuk setiap suksesor yang valid, hitung $g(successor)$, yaitu $g(successor) = g(currentState) + 1$.
- Untuk setiap suksesor yang valid, jika suksesor belum ada di *closedSet*, tambahkan suksesor ke *openSet* dan *closedSet*.

3. Tidak ada solusi:

Jika *openSet* kosong dan belum ditemukan jalur menuju tujuan, maka tidak ada solusi yang dapat diapai dari state awal.

1.2.2. Greedy Best First Search (GBFS)

1. Inisialisasi:

- Buat sebuah struktur data *PriorityQueue* yang dinamakan *openSet*, yang akan mengurutkan *state-state* (node-node) yang ada berdasarkan nilai $h(n)$ terkecil. Perhitungan nilai $h(n)$ ditentukan berdasarkan jenis heuristik yang dipilih.
- Buat sebuah struktur data *Set* yang dinamakan *closedSet* untuk menyimpan *state-state* yang sudah pernah dieksplorasi.
- Buat state awal dengan $g(initialState) = 0$, hitung nilai $h(n)$, lalu masukkan ke dalam *openSet* dan *closedSet*

2. Loop Utama:

Selama *openSet* tidak kosong:

- Ambil (dan hapus) state dari *openSet* yang memiliki nilai $h(n)$ terkecil, beri nama *currentState*.
- Cek posisi *primary piece* pada *currentState*, jika telah berada pada pintu keluar, maka posisi telah ditemukan. Jalur untuk mencapai *state* ini pasti merupakan jalur optimal. Hentikan pencarian dan kembalikan jalur yang ditemukan
- Jika posisi *primary piece* tidak berada pada pintu keluar: bangkitkan semua node (state) suksesor yang valid dari *currentState*. State suksesor yang valid dari *currentState* adalah seluruh kemungkinan melakukan satu langkah pergerakan pada satu buah *piece* pada *board*.

- Untuk setiap suksesor yang valid, hitung $g(successor)$, yaitu $g(successor) = g(currentState) + 1$. Kemudian, hitung $h(successor)$.
- Untuk setiap suksesor yang valid, jika suksesor belum ada di *closedSet*, tambahkan suksesor ke *openSet* dan *closedSet*.

3. Tidak ada solusi:

Jika *openSet* kosong dan belum ditemukan jalur menuju tujuan, maka tidak ada solusi yang dapat diapai dari state awal.

1.2.3. A*

1. Inisialisasi:

- Buat sebuah struktur data *PriorityQueue* yang dinamakan *openSet*, yang akan mengurutkan *state-state* (node-node) yang ada berdasarkan nilai $f(n)$ terkecil, di mana $f(n) = g(n) + h(n)$. Perhitungan nilai $h(n)$ ditentukan berdasarkan jenis heuristik yang dipilih.
- Buat sebuah struktur data *Set* yang dinamakan *closedSet* untuk menyimpan *state-state* yang sudah pernah dieksplorasi.
- Buat state awal dengan $g(initialState) = 0$, hitung nilai $h(n)$, hitung nilai $f(n)$, lalu masukkan ke dalam *openSet* dan *closedSet*

2. Loop Utama:

Selama *openSet* tidak kosong:

- Ambil (dan hapus) state dari *openSet* yang memiliki nilai $f(n)$ terkecil, beri nama *currentState*.
- Cek posisi *primary piece* pada *currentState*, jika telah berada pada pintu keluar, maka posisi telah ditemukan. Jalur untuk mencapai *state* ini pasti merupakan jalur optimal. Hentikan pencarian dan kembalikan jalur yang ditemukan
- Jika posisi *primary piece* tidak berada pada pintu keluar: bangkitkan semua node (state) suksesor yang valid dari *currentState*. State suksesor yang valid dari *currentState* adalah seluruh kemungkinan melakukan satu langkah pergerakan pada satu buah *piece* pada *board*.

- Untuk setiap suksesor yang valid, hitung $g(successor)$, yaitu $g(successor) = g(currentState) + 1$. Kemudian, hitung $h(successor)$, dan nilai $f(successor)$.
- Untuk setiap suksesor yang valid, jika suksesor belum ada di *closedSet*, tambahkan suksesor ke *openSet* dan *closedSet*.

3. Tidak ada solusi:

Jika *openSet* kosong dan belum ditemukan jalur menuju tujuan, maka tidak ada solusi yang dapat diapai dari state awal.

1.3. Analisis Algoritma

1.3.1. *Uniform Cost Search (UCS)*

Uniform Cost Search (UCS) adalah algoritma pencarian yang mengeksplorasi ruang keadaan dengan memprioritaskan node atau keadaan yang memiliki total biaya $g(n)$ terendah dari keadaan awal. Dalam konteks permainan Rush Hour, $g(n)$ diartikan sebagai jumlah total gerakan yang telah dilakukan untuk mencapai konfigurasi papan saat ini dari konfigurasi awal. Setiap gerakan satu *piece* dihitung sebagai satu unit biaya. UCS bekerja dengan cara selalu memilih jalur dengan total gerakan dari titik awal paling sedikit, yaitu $g(n)$ yang paling kecil, untuk diekspansi. Jika setiap eksplorasi gerakan baru memiliki biaya yang sama, maka perilaku UCS dalam mengeksplorasi node akan identik dengan Breadth-First Search (BFS). Keduanya akan menjelajahi semua keadaan yang dapat dicapai dengan k langkah gerakan sebelum menjelajahi keadaan yang memerlukan $k+1$ langkah gerakan. Oleh karena itu, untuk Rush Hour dengan biaya langkah seragam, yaitu 1 unit biaya langkah gerakan pada tiap eksplorasi konfigurasi papan yang baru, urutan *node* yang dibangkitkan akan sama seperti urutan pembangkitan pada BFS, serta UCS menjamin penemuan solusi dengan jumlah gerakan minimal, sama seperti BFS.

1.3.2. *Greedy Best First Search (GBFS)*

Greedy Best-First Search adalah algoritma pencarian terinformasi yang memilih node untuk dieksplorasi selanjutnya murni berdasarkan fungsi heuristik $h(n)$.

Fungsi $h(n)$ memberikan estimasi biaya yang diperlukan dari keadaan n saat ini menuju ke keadaan tujuan. Algoritma ini disebut *greedy* "serakah" karena selalu bergerak ke keadaan yang tampak paling dekat dengan solusi tanpa mempertimbangkan biaya $g(n)$ yang telah dikeluarkan untuk mencapai keadaan tersebut. Dalam kasus Rush Hour dengan heuristik Manhattan Distance, $h(n)$ akan mengestimasi jumlah langkah minimum yang dibutuhkan *primary piece* untuk mencapai pintu keluar, yaitu dengan mengabaikan keberadaan *piece-piece* lain yang mungkin menghalangi. Sedangkan, dengan heuristik Blocking Piece Count, $h(n)$ akan mengestimasi jumlah langkah minimum yang dibutuhkan *primary piece* untuk mencapai pintu keluar, yaitu dengan menghitung banyak *piece* yang menghalangi *primary piece* untuk mencapai pintuk keluar.

Karena sifatnya yang hanya fokus pada heuristik, Greedy BFS tidak selalu menjamin penemuan solusi optimal (jumlah langkah paling sedikit). Ia mungkin menemukan solusi dengan cepat karena pendekatannya yang langsung, tetapi solusi tersebut bisa jadi suboptimal. Ada kemungkinan algoritma ini terjebak pada jalur yang awalnya terlihat optimal (nilai $h(n)$ rendah) namun ternyata memerlukan banyak langkah atau merupakan jalan buntu. Oleh karena itu, Greedy BFS kurang dapat diandalkan untuk menemukan jumlah gerakan minimal dalam permainan Rush Hour, meskipun bisa jadi memiliki waktu yang lebih cepat dalam menemukan suatu solusi.

1.3.3. A*

Algoritma A* adalah algoritma pencarian terinformasi yang menggabungkan aspek-aspek dari UCS dan Greedy BFS. Fungsi evaluasinya, $f(n) = h(n) + g(n)$, mempertimbangkan baik biaya aktual yang telah dikeluarkan untuk mencapai keadaan n dari awal $g(n)$, maupun estimasi biaya heuristik dari keadaan n menuju tujuan $h(n)$. Maka, algoritma A* mencoba menyeimbangkan antara memilih jalur yang sejauh ini murah dan jalur yang secara heuristik terlihat optimal.

Pada algoritma A*, jika heuristik $h(n)$ yang digunakan bersifat *admissible*, artinya tidak pernah melebihi-lebihkan biaya sebenarnya untuk mencapai tujuan, maka A* dijamin akan menemukan solusi optimal (jumlah langkah paling sedikit). Heuristik Manhattan Distance, yang menghitung jumlah gerakan horizontal dan vertikal minimum mobil utama ke target tanpa hambatan, bersifat *admissible* karena keberadaan mobil lain hanya bisa menambah atau mempertahankan jumlah gerakan yang dibutuhkan, tidak pernah menguranginya di bawah estimasi Manhattan Distance. Kemudian, heuristik Blocking Piece Count, yang menghitung jumlah mobil yang menghalangi mobil utama ke target, juga bersifat *admissible* karena untuk mencapai target, dibutuhkan minimal langkah sebanyak mobil yang menghalangi mobil utama, yaitu untuk memindahkan seluruh mobil penghalang tersebut agar mobil utama dapat mencapai pintu keluar. Dibandingkan UCS, A* berpotensi lebih efisien karena panduan heuristik dapat memfokuskan pencarian ke arah yang lebih menjanjikan, sehingga mengurangi jumlah node yang perlu dieksplorasi.

1.4. Pseudocode

1.4.1. Uniform Cost Search (UCS)

```
FUNCTION UCS(initialState):
  INITIALIZE openSet : PriorityQueue (ordered by smallest g(n))
  INITIALIZE closedSet : Set
  initialState.gCost = 0
  ADD initialState TO openSet
  ADD initialState TO closedSet

  WHILE openSet IS NOT EMPTY:
    currentState = POP openSet

    IF currentState IS goal_state THEN:
      PRINT "Solution found"
      RETURN path_to(currentState)
    END IF
```

```

    FOR EACH successor OF generateSuccessors(currentState):
        { g(successor) is calculated during generateSuccessors }
        IF successor IS NOT IN closedSet THEN:
            ADD successor TO openSet
            ADD successor TO closedSet
        END IF
    END FOR
END WHILE

PRINT "No solution found"
RETURN empty_list
END FUNCTION

```

1.4.2. *Greedy Best First Search (GBFS)*

```

FUNCTION GBFS(initialState):
    INITIALIZE openSet : PriorityQueue (ordered by smallest h(n))
    INITIALIZE closedSet : Set
    initialState.gCost = 0
    ADD initialState TO openSet
    ADD initialState TO closedSet

    WHILE openSet IS NOT EMPTY:
        currentState = POP openSet

        IF currentState IS goal_state THEN:
            PRINT "Solution found"
            RETURN path_to(currentState)
        END IF

        FOR EACH successor OF generateSuccessors(currentState):
            { g(successor), h(successor) is calculated during generateSuccessors }
            IF successor IS NOT IN closedSet THEN:
                ADD successor TO openSet
                ADD successor TO closedSet
            END IF
        END FOR
    END WHILE

    PRINT "No solution found"
    RETURN empty_list
END FUNCTION

```

1.4.3. *A**

```

FUNCTION AStar(initialState):
  INITIALIZE openSet : PriorityQueue (ordered by smallest  $f(n)$ )
  INITIALIZE closedSet : Set
  initialState.gCost = 0
  ADD initialState TO openSet
  ADD initialState TO closedSet

  WHILE openSet IS NOT EMPTY:
    currentState = POP openSet

    IF currentState IS goal_state THEN:
      PRINT "Solution found"
      RETURN path_to(currentState)
    END IF

    FOR EACH successor OF generateSuccessors(currentState):
      { g(successor), h(successor) is calculated during generateSuccessors }
      IF successor IS NOT IN closedSet THEN:
        ADD successor TO openSet
        ADD successor TO closedSet
      END IF
    END FOR
  END WHILE

  PRINT "No solution found"
  RETURN empty_list
END FUNCTION

```

BAB II

IMPLEMENTASI

2.1. Implementasi Piece.java

Fungsi/Prosedur/Class	Deskripsi
place(int i, int j, Board board)	Mencoba memindahkan piece ke posisi (i, j) pada board. Jika posisi tersebut valid (tidak bertabrakan dan masih dalam board), piece akan dipindahkan dan fungsi mengembalikan true. Jika tidak valid, mengembalikan false.
moveForward(Board board)	Menggerakkan piece satu langkah ke depan (bawah jika vertikal, kanan jika horizontal) pada board. Menggunakan fungsi place untuk memvalidasi dan melakukan perpindahan.
moveBackward(Board board)	Menggerakkan piece satu langkah ke belakang (atas jika vertikal, kiri jika horizontal) pada board. Juga menggunakan fungsi place untuk validasi dan perpindahan.
getPosI() getPosJ() getWidth() getHeight()	Digunakan untuk mengambil nilai atribut-atribut piece.

2.2. Implementasi PrimaryPiece.java

Fungsi/Prosedur/Class	Deskripsi
isWinningPos(int i, int j, Board board)	Mengecek apakah posisi (i, j) pada board adalah posisi kemenangan (goal) untuk PrimaryPiece.
place(int i, int j, Board board)	Mencoba menempatkan PrimaryPiece di posisi (i, j) pada board. Jika posisi tersebut adalah posisi kemenangan, atribut win di-set true.
getWin()	Mengembalikan status apakah PrimaryPiece sudah mencapai posisi kemenangan.

2.3. Implementasi Move.java

Fungsi/Prosedur/Class	Deskripsi
Move(char pieceColor, String direction)	Konstruktor untuk membuat objek Move, menyimpan warna piece dan arah gerakan (misal: "UP", "DOWN", "LEFT", "RIGHT").
toString()	Mengembalikan representasi string dari gerakan, misal "A - RIGHT". Jika pieceColor adalah '-', mengembalikan string kosong.

2.4. Implementasi State.java

Fungsi/Prosedur/Class	Deskripsi
isGoal()	Mengecek apakah state ini merupakan state tujuan (goal), yaitu primary piece sudah mencapai posisi kemenangan.
generateSuccessors()	Menghasilkan semua kemungkinan state berikutnya (successor) dari state saat ini, dengan mencoba semua pergerakan yang valid untuk setiap piece.
equals(Object obj) & hashCode()	Membandingkan dua state berdasarkan konfigurasi board-nya, sehingga bisa digunakan untuk pencarian dan pengecekan duplikasi state.
getMoves()	Mengembalikan urutan state dari awal hingga state saat ini, berguna untuk melacak solusi atau langkah-langkah yang diambil.

2.5. Implementasi Heuristic.java

Fungsi/Prosedur/Class	Deskripsi
calculateH(State state)	Fungsi utama untuk menghitung nilai heuristik pada sebuah state. Memilih metode heuristik sesuai dengan heuristicType (misal: "MANHATTAN" atau "BLOCKING_PIECE_COUNT").
manhattanDistance(State state)	Menghitung jarak Manhattan antara primary piece dan posisi goal pada board. Semakin kecil nilainya, semakin dekat ke solusi.

blockingPieceCount(State state)	Menghitung jumlah piece yang menghalangi jalur primary piece menuju goal. Semakin sedikit penghalang, semakin dekat ke solusi.
---------------------------------	--

2.6. Implementasi Reader.java

Fungsi/Prosedur/Class	Deskripsi
getBoard() getPieces() getPrimaryPieceRef() getKPos()	Fungsi getter untuk mengambil objek Board, daftar Piece, referensi PrimaryPiece, dan posisi goal (K) yang sudah diproses dari input.
parseK(String[] lines, int width, int height)	Mencari dan memvalidasi posisi goal (K) pada input, memastikan hanya ada satu K dan posisinya valid di tepi board.
isKPosValid()	Memastikan posisi goal (K) sudah sejajar dengan PrimaryPiece (baik secara vertikal maupun horizontal).
removeKAndSpaceFromInput(), isValidBoard(String[] lines, int width, int height)	Membersihkan input dari karakter K dan spasi, serta memvalidasi karakter pada board agar hanya berisi karakter yang valid.

2.7. Implementasi Solver.java

Fungsi/Prosedur/Class	Deskripsi
solve(SearchMode searchMode)	Fungsi utama untuk mencari solusi. Menggunakan algoritma pencarian (UCS, A*, atau Greedy) sesuai parameter, dan mengembalikan urutan state solusi jika ditemukan.
SearchMode (enum)	Enum yang mendefinisikan mode pencarian: GREEDY, A_STAR, dan UCS. Digunakan untuk memilih strategi pencarian solusi.
hasFoundSolution()	Mengembalikan status apakah solusi telah ditemukan oleh solver.
getNumMoves()	Mengembalikan jumlah node/state yang telah diekspansi selama proses pencarian solusi.

2.8. Implementasi MainController.java

Fungsi/Prosedur/Class	Deskripsi
initialize() initializeBoard() initializePieces() initializeGoalDisplay()	Mengatur inisialisasi awal aplikasi, membaca konfigurasi papan dari input/file, membuat objek-objek piece dan goal pada board, serta menyiapkan tampilan awal.
onClickPlay() onClickNext() onClickPrevious() onClickToStart() onClickToEnd() nextStep() previousStep() setRectanglesToCurrentState()	Mengatur animasi dan navigasi langkah solusi pada board, baik secara otomatis (play) maupun manual (next, previous, ke awal, ke akhir).
onClickApplyConfiguration() onClickUploadFile() onClickSolve()	Menangani aksi tombol utama: menerapkan konfigurasi board, mengunggah file konfigurasi, dan menjalankan proses pencarian solusi.
makeDraggable() onRectPressed() onRectDragged() onRectReleased()	Mengatur interaksi drag-and-drop pada piece di board, termasuk logika snapping dan deteksi batas/collision.
calculateLeftmostX() calculateRightmostX() calculateTopmostY() calculateBottommostY() snapToGrid()	Fungsi-fungsi bantu untuk menghitung batas pergerakan piece dan memastikan piece tetap pada grid yang valid.

BAB III

SOURCE CODE

3.1. Repository Program

Tautan menuju repository program adalah sebagai berikut:
https://github.com/karolyangqian/Tucil3_13523077_13523093.git

3.2. Source Code Program

3.2.1. Piece.java

```
package rushhour;

public class Piece {
    protected int posI, posJ;
    protected int width, height;
    protected char color;
    protected boolean isVertical;

    public Piece(char color, int width, int height, int i, int j) {
        this.color = color;
        this.width = width;
        this.height = height;
        this.isVertical = this.width < this.height;
        this.posI = i;
        this.posJ = j;
        if (this.isVertical && this.width != 1) {
            String errorMsg = "Invalid piece: " + color + " at (" + i + ", " + j +
            "). " +
            "Expected: piece's dimensions has to be 1x" + height + ", but found: "
+ width + "x" + height;
            throw new IllegalArgumentException(errorMsg);
        }
        if (!this.isVertical && this.height != 1) {
            String errorMsg = "Invalid piece: " + color + " at (" + i + ", " + j +
            "). " +
            "Expected: piece's dimensions has to be " + width + "x1, but found: " +
width + "x" + height;
            throw new IllegalArgumentException(errorMsg);
        }
        if (this.width == this.height){
            String errorMsg = "Invalid piece: " + color + " at (" + i + ", " + j +
            "). " +
            "Expected: piece's dimensions cannot be 1x1";
        }
    }
}
```

```

        throw new IllegalArgumentException(errorMsg);
    }
}

public Piece(Piece piece) {
    this.posI = piece.posI;
    this.posJ = piece.posJ;
    this.width = piece.width;
    this.height = piece.height;
    this.color = piece.color;
    this.isVertical = piece.isVertical;
}

public boolean isPlaceable(int i, int j, Board board) {
    if (i < 0 || j < 0 || i + height > board.getHeight() || j + width >
board.getWidth()) {
        return false;
    }
    for (int k = 0; k < height; k++) {
        for (int l = 0; l < width; l++) {
            if (board.isOccupied(i + k, j + l) && board.getPieceAt(i + k, j +
l) != this) {
                return false;
            }
        }
    }
    return true;
}

public boolean place(int i, int j, Board board){
    if (!isPlaceable(i, j, board)) return false;
    board.unplacePiece(this);
    this.posI = i;
    this.posJ = j;
    board.placePiece(this);
    return true;
}

public boolean moveForward(Board board) {
    return isVertical ? place(posI + 1, posJ, board) : place(posI, posJ + 1,
board);
}

public boolean moveBackward(Board board) {
    return isVertical ? place(posI - 1, posJ, board) : place(posI, posJ - 1,
board);
}

```

```

    public int getPosI() {
        return posI;
    }
    public int getPosJ() {
        return posJ;
    }
    public int getWidth() {
        return width;
    }
    public int getHeight() {
        return height;
    }
    public char getColor() {
        return color;
    }
    public boolean isVertical() {
        return isVertical;
    }
}

```

3.2.2. PrimaryPiece.java

```

package rushhour;

public class PrimaryPiece extends Piece {

    private boolean win = false;

    public PrimaryPiece(char color, int width, int height, int i, int j) {
        super(color, width, height, i, j);
    }

    public PrimaryPiece(PrimaryPiece piece) {
        super(piece);
        this.win = false;
    }

    public boolean isWinningPos(int i, int j, Board board) {
        // System.out.println("ok");
        for (int k = 0; k < getHeight(); k++) {
            for (int l = 0; l < getWidth(); l++) {
                if (i + k == board.getWinPosI() && j + l == board.getWinPosJ()) {
                    return true;
                }
            }
        }
    }
}

```

```

        }
    }
}

return false;
}

@Override
public boolean place(int i, int j, Board board) {
    if (!isPlaceable(i, j, board)) return false;

    if (isWinningPos(i, j, board)){
        this.win = true;
    }
    board.unplacePiece(this);
    this.posI = i;
    this.posJ = j;
    board.placePiece(this);
    return true;
}

public boolean getWin() {
    return win;
}
}

```

3.2.3. Move.java

```

package rushhour;

class Move {
    char pieceColor;
    String direction; // e.g., "UP", "DOWN", "LEFT", "RIGHT"

    public Move(char pieceColor, String direction) {
        this.pieceColor = pieceColor;
        this.direction = direction;
    }

    @Override
    public String toString() {
        if (pieceColor == '-') return "";
        return pieceColor + " - " + direction;
    }
}

```

3.2.4. State.java

```
package rushhour;

import java.util.ArrayList;
import java.util.List;

public class State {
    Board boardConfiguration;
    List<Piece> piecesState;
    State parent;
    Move lastMove;
    int gCost;
    int hCost;
    PrimaryPiece primaryPieceRef;

    public State(Board boardConfig, List<Piece> pieces, State parent, Move
lastMove, int gCost, PrimaryPiece primaryPieceRef) {
        this.piecesState = new ArrayList<>();
        this.boardConfiguration = boardConfig;
        for (Piece p : pieces) {
            this.piecesState.add(p); // Pieces must be cloneable
        }
        this.parent = parent;
        this.lastMove = lastMove;
        this.gCost = gCost;
        this.primaryPieceRef = primaryPieceRef;
        this.hCost = Heuristics.calculateH(this);
    }

    public int getFCost() {
        return gCost + hCost;
    }

    public boolean isGoal() {
        return primaryPieceRef.getWin();
    }

    @Override
    public int hashCode() {
        return boardConfiguration.toString().hashCode();
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
    }
}
```

```

        State otherState = (State) obj;
        return
boardConfiguration.toString().equals(otherState.boardConfiguration.toString());
    }

    @Override
    public String toString() {
        return boardConfiguration.toString();
    }

    // Helper to generate successor states
    public List<State> generateSuccessors() {
        List<State> successors = new ArrayList<>();
        // check for all possible moves
        for (Piece piece : piecesState) {
            for (int i = 0; i < 2; i++){
                Board newBoard = new Board(boardConfiguration);
                List<Piece> newPieces = new ArrayList<>();
                Piece chosenPiece = null;
                PrimaryPiece newPrimaryPiece = null;
                for (Piece p : piecesState) {
                    Piece newPiece;
                    if (p == primaryPieceRef){
                        newPiece = new PrimaryPiece((PrimaryPiece) p);
                        newPrimaryPiece = (PrimaryPiece) newPiece;
                    }
                    else {
                        newPiece = new Piece(p);
                    }
                    newPieces.add(newPiece);
                    if (p == piece) chosenPiece = newPiece;
                }
                newBoard.buildBoard(newPieces);

                State newState;
                if (i == 0) {
                    String direction = chosenPiece.isVertical() ? "UP" : "LEFT";
                    Move move = new Move(chosenPiece.getColor(), direction);
                    while (chosenPiece.moveBackward(newBoard)){
                        newState = new State(newBoard, newPieces, this, move, gCost
+ 1, newPrimaryPiece);
                        successors.add(newState);
                    }
                }
                else {
                    String direction = chosenPiece.isVertical() ? "DOWN" : "RIGHT";
                    Move move = new Move(chosenPiece.getColor(), direction);

```

```

        while (chosenPiece.moveForward(newBoard)){
            newState = new State(newBoard, newPieces, this, move, gCost
+ 1, newPrimaryPiece);
            successors.add(newState);
        }
    }
}

return successors;
}

public List<State> getMoves(){
    State currentState = this;
    List<Move> moves = new ArrayList<>();
    List<State> states = new ArrayList<>();
    while (currentState != null) {
        if (currentState.lastMove != null) {
            moves.add(currentState.lastMove);
        }
        states.add(currentState);
        currentState = currentState.parent;
    }
    List<State> reversedStates = new ArrayList<>();
    reversedStates.add(states.get(states.size() - 1)); // Add the initial state
    for (int i = moves.size() - 1; i >= 0; i--) {
        reversedStates.add(states.get(i));
    }
    return reversedStates;
}

public List<Piece> getPieces() {
    return piecesState;
}
}

```

3.2.5. Heuristic.java

```

package rushhour;

import java.util.HashSet;
import java.util.Set;

public class Heuristics {
    // MANHATTAN

```

```

public static String heuristicType;

public static int calculateH(State state) {
    if (heuristicType.equals("MANHATTAN")) {
        return manhattanDistance(state);
    }
    else if (heuristicType.equals("BLOCKING_PIECE_COUNT")){
        return blockingPieceCount(state);
    }
    return 0; // Default case, should not happen
}

private static int manhattanDistance(State state) {
    int goalI = state.boardConfiguration.getWinPosI();
    int goalJ = state.boardConfiguration.getWinPosJ();
    int pieceITop = state.primaryPieceRef.getPosI();
    int pieceIBottom = state.primaryPieceRef.getPosI() +
state.primaryPieceRef.getHeight() - 1;
    int pieceJLeft = state.primaryPieceRef.getPosJ();
    int pieceJRight = state.primaryPieceRef.getPosJ() +
state.primaryPieceRef.getWidth() - 1;

    int dx = 0;
    if (goalI < pieceITop) {
        dx = pieceITop - goalI;
    } else if (goalI > pieceIBottom) {
        dx = goalI - pieceIBottom;
    }
    int dy = 0;
    if (goalJ < pieceJLeft) {
        dy = pieceJLeft - goalJ;
    } else if (goalJ > pieceJRight) {
        dy = goalJ - pieceJRight;
    }
    return dx + dy;
}

private static int blockingPieceCount(State state) {
    PrimaryPiece p = state.primaryPieceRef;
    Board board = state.boardConfiguration;
    int goalI = board.getWinPosI();
    int goalJ = board.getWinPosJ();
    Set<Piece> distinctPiece = new HashSet<>();

    if (p.isVertical()){
        if (p.getPosJ() != goalJ) return 0;
    }
}

```



```

        int pieceTopEdge = p.getPosI();
        int pieceBottomEdge = p.getPosI() + p.getHeight() - 1;

        if (pieceBottomEdge < goalJ) {
            for (int r = pieceBottomEdge + 1; r <= goalJ; r++) {
                Piece pp = board.getPieceAt(r, p.getPosJ());
                if (pp != null) distinctPiece.add(pp);
            }
        } else if (pieceTopEdge > goalJ) {
            for (int r = pieceTopEdge - 1; r >= goalJ; r--) {
                Piece pp = board.getPieceAt(r, p.getPosJ());
                if (pp != null) distinctPiece.add(pp);
            }
        }
    }
    else {
        if (p.getPosI() != goalI) return 0;

        int pieceLeftEdge = p.getPosJ();
        int pieceRightEdge = p.getPosJ() + p.getWidth() - 1;

        if (pieceRightEdge < goalJ) {
            for (int c = pieceRightEdge + 1; c <= goalJ; c++) {
                Piece pp = board.getPieceAt(p.getPosI(), c);
                if (pp != null) distinctPiece.add(pp);
            }
        } else if (pieceLeftEdge > goalJ) {
            for (int c = pieceLeftEdge - 1; c >= goalJ; c--) {
                Piece pp = board.getPieceAt(p.getPosI(), c);
                if (pp != null) distinctPiece.add(pp);
            }
        }
    }
    return distinctPiece.size();
}
}

```

3.2.6. Reader.java

```

package rushhour;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

```

```

public class Reader {
    private Board board;
    private List<Piece> pieces;
    private PrimaryPiece primaryPieceRef = null;
    private int[] kPos;

    public Reader(String input, int boardWidth, int boardHeight, int
numNonPrimaryPieces) {
        this.pieces = new ArrayList<>();
        String[] lines = input.split("\n");

        try {
            kPos = parseK(lines, boardWidth, boardHeight);
        } catch (IllegalArgumentException e) {
            throw new IllegalArgumentException("Invalid input: " + e.getMessage());
        }

        lines = removeKAndSpaceFromInput(lines);

        // Pemeriksaan defensif jika jumlah baris input string tidak sesuai dengan
boardHeight
        if (lines.length != boardHeight) {
            throw new IllegalArgumentException("Jumlah baris input string (" +
lines.length
                + ") tidak sesuai dengan boardHeight (" + boardHeight + ")");
        }

        char[][] grid = new char[boardHeight][boardWidth];
        boolean[][] visited = new boolean[boardHeight][boardWidth];
        for (int i = 0; i < boardHeight; i++) {
            if (lines[i].length() != boardWidth) {
                throw new IllegalArgumentException("Panjang baris input string ke-"
+ i + " (" + lines[i].length()
                    + ") tidak sesuai dengan boardWidth (" + boardWidth + ")");
            }
            for (int j = 0; j < boardWidth; j++) {
                grid[i][j] = lines[i].charAt(j);
                visited[i][j] = false;
            }
        }

        if (!isValidBoard(lines, boardWidth, boardHeight)) {
            throw new IllegalArgumentException("Invalid input: Invalid characters
in the board");
        }
    }

```

```

Set<Character> processedPieceChars = new HashSet<>();

for (int r = 0; r < boardHeight; r++) {
    for (int c = 0; c < boardWidth; c++) {
        char currentChar = grid[r][c];

        if (currentChar == '.') { // Sel kosong
            continue;
        }

        if (processedPieceChars.contains(currentChar)) { // Potongan ini
            sudah diproses
            if (!visited[r][c]){
                throw new IllegalArgumentException("Invalid Piece: \' +
currentChar + "\'");
            }
            continue;
        }

        // Potongan baru ditemukan
        char pieceColor = currentChar;
        int pieceTopI = r;
        int pieceLeftJ = c;

        // Tentukan lebar dan tinggi aktual dari potongan ini dari (r,c)
        // dalam batas-batas boardWidth dan boardHeight.
        int currentPieceActualWidth = 0;
        while (c + currentPieceActualWidth < boardWidth && grid[r][c +
currentPieceActualWidth] == pieceColor) {
            visited[r][c + currentPieceActualWidth] = true; // Tandai
            sebagai dikunjungi
            currentPieceActualWidth++;
        }

        int currentPieceActualHeight = 0;
        while (r + currentPieceActualHeight < boardHeight && grid[r +
currentPieceActualHeight][c] == pieceColor) {
            visited[r + currentPieceActualHeight][c] = true; // Tandai
            sebagai dikunjungi
            currentPieceActualHeight++;
        }

        Piece newPiece;
        try {
            if (pieceColor == 'P'){
                this.primaryPieceRef = new PrimaryPiece(pieceColor,

```

```

currentPieceActualWidth, currentPieceActualHeight, pieceTopI, pieceLeftJ);
        newPiece = this.primaryPieceRef;
    }
    else {
        newPiece = new Piece(pieceColor, currentPieceActualWidth,
currentPieceActualHeight, pieceTopI, pieceLeftJ);
    }
    } catch (IllegalArgumentException e) {
        throw new IllegalArgumentException("Invalid input: " +
e.getMessage());
    }
    this.pieces.add(newPiece);
    processedPieceChars.add(pieceColor);
}
}

if (this.primaryPieceRef == null){
    throw new IllegalArgumentException("Invalid input: No primary piece
\'P\'");
}

if (!isKPosValid()){
    throw new IllegalArgumentException("Invalid input: K position is not
aligned with primary piece \'P\'");
}

int count = 0;
for(Piece p : this.pieces){
    if(p.getColor() != 'P'){
        count++;
    }
}

if (count != numNonPrimaryPieces) {
    System.err.println("Peringatan: Jumlah potongan non-primer yang
ditemukan (" + count
        + ") tidak sesuai dengan yang diharapkan N (" +
numNonPrimaryPieces + ")");
}

this.board = new Board(boardWidth, boardHeight, kPos[0], kPos[1]);
this.board.buildBoard(this.pieces);
}

public Board getBoard() {
    return board;
}

```

```

    }

    public List<Piece> getPieces() {
        return pieces;
    }

    public PrimaryPiece getPrimaryPieceRef() {
        return primaryPieceRef;
    }

    public int[] getKPos() {
        return kPos;
    }

    private boolean isKPosValid(){
        if (this.primaryPieceRef.isVertical()){
            return this.primaryPieceRef.getPosJ() == this.kPos[1];
        }
        else {
            return this.primaryPieceRef.getPosI() == this.kPos[0];
        }
    }

    private int[] parseK(String[] lines, int width, int height) {
        int[] k = new int[]{-1, -1};
        for (int i = 0; i < lines.length; i++) {
            for (int j = 0; j < lines[i].length(); j++) {
                if (lines[i].charAt(j) == 'K') {
                    if (k[0] != -1) {
                        throw new IllegalArgumentException("Multiple K pieces
found");
                    }
                    k[0] = i; k[1] = j;
                }
            }
        }

        if (k[0] == -1) {
            throw new IllegalArgumentException("No K piece found");
        }
        int r = k[0];
        int c = k[1];
        int newR = -1;
        int newC = -1;

        if (r == height) {
            if (c >= 0 && c < width) {

```

```

        newR = height - 1;
        newC = c;
    } else {
        throw new IllegalArgumentException("Invalid K position");
    }
}
else if (c == width) {
    if (r >= 0 && r < height) {
        newC = width - 1;
        newR = r;
    } else {
        throw new IllegalArgumentException("Invalid K position");
    }
}
else if (r >= 0 && r < height && c >= 0 && c < width) {
    boolean onWall = (r == 0 || c == 0);
    if (onWall) {
        newR = r;
        newC = c;
    } else {
        throw new IllegalArgumentException("Invalid K position");
    }
}
else {
    throw new IllegalArgumentException("Invalid K position");
}

return new int[]{newR, newC};
}

private boolean isValidBoard(String[] lines, int width, int height) {
    for (String line : lines) {
        for (char c : line.toCharArray()) {
            if (c != '.' && !Character.isLetter(c) &&
!Character.isUpperCase(c)) {
                return false;
            }
        }
    }
    return true;
}

private String[] removeKAndSpaceFromInput(String[] input){
    List<String> newInput = new ArrayList<>();
    for (String line : input) {
        StringBuilder newLine = new StringBuilder();

```

```

        for (int j = 0; j < line.length(); j++) {
            if (line.charAt(j) != 'K' && line.charAt(j) != ' ') {
                newLine.append(line.charAt(j));
            }
        }
        if (newLine.length() > 0) {
            newInput.add(newLine.toString());
        }
    }
    return newInput.toArray(new String[0]);
}
}

```

3.2.7. Solver.java

```

package rushhour;

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

public class Solver {
    private Board board;
    private List<Piece> pieces;
    private PrimaryPiece primaryPiece;
    private int numMoves = 0;
    private boolean foundSolution = false;

    public Solver(Board board, List<Piece> pieces, PrimaryPiece primaryPiece) {
        this.board = board;
        this.pieces = pieces;
        this.primaryPiece = primaryPiece;
    }

    public enum SearchMode {
        GREEDY, A_STAR, UCS
    }

    public List<State> solve(SearchMode searchMode) {
        numMoves = 0;
        foundSolution = false;

        PriorityQueue<State> openSet;
    }
}

```

```

        if (searchMode == SearchMode.A_STAR) {
            openSet = new
PriorityQueue<>(Comparator.comparingInt(State::getFCost));
        }
        else if (searchMode == SearchMode.GREEDY) {
            openSet = new PriorityQueue<>(Comparator.comparingInt(s -> s.hCost));
        }
        else {
            openSet = new PriorityQueue<>(Comparator.comparingInt(s -> s.gCost));
        }

        Set<State> closedSet = new HashSet<>();

        State initialState = new State(this.board, this.pieces, null, null, 0,
this.primaryPiece);
        openSet.add(initialState);
        closedSet.add(initialState);
        while (!openSet.isEmpty()) {
            State currentState = openSet.poll();
            numMoves++;

            if (currentState.isGoal()) {
                foundSolution = true;
                System.out.println("Found solution in " + numMoves + " nodes.");
                return currentState.getMoves();
            }

            List<State> successors = currentState.generateSuccessors();
            for (State successor : successors) {
                if (!closedSet.contains(successor)) {
                    openSet.add(successor);
                    closedSet.add(successor);
                }
            }
        }

        System.out.println("No solution found.");
        System.out.println("numMoves: " + numMoves);
        return new java.util.ArrayList<>();
    }

    public int getNumMoves() {
        return numMoves;
    }

    public boolean hasFoundSolution() {
        return foundSolution;
    }

```



```
}  
  
}
```

3.2.8. MainController.java

```
package com.stima;  
  
import java.io.File;  
import java.nio.file.Files;  
import java.util.ArrayList;  
import java.util.List;  
  
import javafx.animation.KeyFrame;  
import javafx.animation.KeyValue;  
import javafx.animation.SequentialTransition;  
import javafx.animation.Timeline;  
import javafx.concurrent.Task;  
import javafx.fxml.FXML;  
import javafx.scene.control.Button;  
import javafx.scene.control.ChoiceBox;  
import javafx.scene.control.Label;  
import javafx.scene.control.TextArea;  
import javafx.scene.input.MouseEvent;  
import javafx.scene.layout.Pane;  
import javafx.scene.paint.Color;  
import javafx.scene.shape.Polygon;  
import javafx.scene.shape.Rectangle;  
import javafx.scene.text.Text;  
import javafx.scene.transform.Rotate;  
import javafx.stage.FileChooser;  
import javafx.stage.Stage;  
import javafx.util.Duration;  
import rushhour.Board;  
import rushhour.Heuristics;  
import rushhour.Piece;  
import rushhour.PrimaryPiece;  
import rushhour.Reader;  
import rushhour.Solver;  
import rushhour.Solver.SearchMode;  
import rushhour.State;  
  
public class MainController {  
  
    // Constants  
    private static final int MIN_GRID_SIZE = 12;
```

```

private static final int GRID_BORDER = 2;

private static final int BOARD_PANE_WIDTH = 400;
private static final int BOARD_PANE_HEIGHT = 350;

private static final int STEP_DURATION = 200; // milliseconds

// FXML Components
@FXML private ChoiceBox<String> algorithmChoiceBox;
@FXML private ChoiceBox<String> heuristicChoiceBox;
@FXML private Button uploadFileButton;
@FXML private Button solveButton;
@FXML private TextArea boardTextArea;
@FXML private Text alertMessageText;
@FXML private Text filenameText;
@FXML private Button nextButton;
@FXML private Button previousButton;
@FXML private Button toStartButton;
@FXML private Button toEndButton;
@FXML private Button playButton;
@FXML private Button applyConfigurationButton;
@FXML private Label stepCounterLabel;

// Game State
private int gridSize;
private Board board;
private File currentFile;
private long solvingStartTime;
private List<Piece> pieces;
private PrimaryPiece primaryPiece;
private List<State> solutionSteps;
private int currentStep = 0;
private Solver solver;
private boolean isPlaying = false;
private boolean isSolved = false;
private boolean isConfigured = false;
private SequentialTransition sequentialTransition;

// FXML Components for the board
@FXML private Pane boardPane;
@FXML private List<PieceRectangle> boardRectangles;

// Dragging variables
private double dragStartX;
private double dragStartY;
private double rectStartX;
private double rectStartY;

```

```

@FXML
public void initialize() {
    isSolved = false;
    isPlaying = false;
    isConfigured = false;

    clearAlerts();
    clearBoard();

    algorithmChoiceBox.getSelectionModel().selectedItemProperty().addListener((obs,
oldVal, newVal) -> {
        boolean heuristicNeeded = !newVal.equals("UCS");
        heuristicChoiceBox.setValue(heuristicNeeded ? "Manhattan" : "");
        heuristicChoiceBox.setDisable(!heuristicNeeded);
    });

    algorithmChoiceBox.getItems().addAll("UCS", "A*", "GBFS");
    algorithmChoiceBox.setValue("UCS");

    heuristicChoiceBox.getItems().addAll("Manhattan", "Blocking Piece Count");
    heuristicChoiceBox.setValue("");
}

private void initializeBoard() {

    String s = boardTextArea.getText();

    if(!validateBoardInput(s)) {
        throw new IllegalArgumentException("Board is empty");
    }

    int row, col, numPieces;

    // input parsing
    String[] lines = s.split("\\R");
    String[] firstLine = lines[0].split(" ");
    if (firstLine.length != 2) {
        throw new IllegalArgumentException("Must provide two integers for rows
and columns");
    }
    try {
        row = Integer.parseInt(firstLine[0]);
        col = Integer.parseInt(firstLine[1]);
    } catch (NumberFormatException e) {

```

```

        throw new IllegalArgumentException("Rows and columns must be integers");
    }

    if (row <= 0 || col <= 0) {
        throw new IllegalArgumentException("Rows and columns must be non zero
positive integers");
    }

    String[] secondLine = lines[1].split(" ");
    if (secondLine.length != 1) {
        throw new IllegalArgumentException("Must provide one integer for number
of pieces");
    }
    try {
        numPieces = Integer.parseInt(secondLine[0]);
    } catch (NumberFormatException e) {
        throw new IllegalArgumentException("Number of pieces must be an
integer");
    }
    if (numPieces <= 0) {
        throw new IllegalArgumentException("Number of pieces must be a non zero
positive integer");
    }

    StringBuilder sb = new StringBuilder();
    for (int i = 2; i < lines.length; i++) {
        sb.append(lines[i]);
        if (i != lines.length - 1) {
            sb.append("\n");
        }
    }

    s = sb.toString();

    // initialize board and pieces
    try {
        Reader r = new Reader(s, col, row, numPieces);
        pieces = r.getPieces();
        primaryPiece = r.getPrimaryPieceRef();
        board = r.getBoard();
    } catch (IllegalArgumentException e) {
        throw new IllegalArgumentException(e.getMessage());
    }

    gridSize = Math.max(calculateGridSize(row, col), MIN_GRID_SIZE);

    boardPane.setPrefSize(col * gridSize, row * gridSize);

```

```

        boardPane.setStyle("-fx-background-color: lightgray;");

        Rectangle clip = new Rectangle(boardPane.getPrefWidth(),
boardPane.getPrefHeight());
        clip.setLayoutX(boardPane.getLayoutX());
        clip.setLayoutY(boardPane.getLayoutY());
        boardPane.setClip(clip);

        initializePieces();

        solver = new Solver(board, pieces, primaryPiece);

        initializeGoalDisplay();

        boardPane.getChildren().forEach(this::makeDraggable);
    }

    private void initializeGoalDisplay() {
        double triangleBase = gridSize * 0.5;
        double triangleHeight = gridSize * 0.5;
        GoalTriangle goalTriangle = new GoalTriangle(triangleBase, triangleHeight);
        goalTriangle.setFill(Color.GREEN);
        goalTriangle.setStroke(Color.BLACK);
        goalTriangle.setStrokeWidth(1);
        goalTriangle.setTranslateX((board.getWinPosJ() + 0.5) * gridSize -
triangleBase / 2);
        goalTriangle.setTranslateY((board.getWinPosI() + 0.5) * gridSize -
triangleHeight / 2);
        goalTriangle.setMouseTransparent(true);
        if (board.getWinPosI() == 0) {
            goalTriangle.setAngle(270);
        } else if (board.getWinPosI() == board.getHeight() - 1) {
            goalTriangle.setAngle(90);
        } else if (board.getWinPosJ() == 0) {
            goalTriangle.setAngle(180);
        } else if (board.getWinPosJ() == board.getWidth() - 1) {
            goalTriangle.setAngle(0);
        }
        boardPane.getChildren().add(goalTriangle);
    }

    private void initializePieces() {

        // Initialize pieces on the board pane
        boardRectangles = new ArrayList<>();
        for (Piece piece : pieces) {

```

```

        PieceRectangle rect = new PieceRectangle(piece.getWidth() * gridSize -
2*GRID_BORDER,
                                piece.getHeight() *
gridSize - 2*GRID_BORDER);
        rect.setX(piece.getPosJ() * gridSize + GRID_BORDER);
        rect.setY(piece.getPosI() * gridSize + GRID_BORDER);
        if (piece instanceof PrimaryPiece) {
            rect.setColor(javafx.scene.paint.Color.RED);
            rect.setPrimaryPiece(true);
        } else {
            rect.setColor(javafx.scene.paint.Color.BLUE);
        }
        rect.setStroke(javafx.scene.paint.Color.BLACK);
        rect.setStrokeWidth(1);
        rect.setStrokeLineJoin(javafx.scene.shape.StrokeLineJoin.ROUND);
        rect.setStrokeType(javafx.scene.shape.StrokeType.INSIDE);
        boardPane.getChildren().add(rect);
        boardRectangles.add(rect);
    }
}

//
=====
// Animation logic
//
=====

private Timeline createPieceTimeline(PieceRectangle rect, int cellsX, int
cellsY) {

    double targetX = rect.getX() + cellsX * gridSize;
    double targetY = rect.getY() + cellsY * gridSize;

    Timeline timeline = new Timeline(
        new KeyFrame(Duration.millis(STEP_DURATION),
            new KeyValue(rect.xProperty(), targetX),
            new KeyValue(rect.yProperty(), targetY))
    );
    return timeline;
}

private void movePieceRectangle(PieceRectangle rect, int cellsX, int cellsY) {
    if (rect == null) return;

    Timeline timeline = createPieceTimeline(rect, cellsX, cellsY);
    timeline.play();
}

```

```

@FXML
private void onClickPlay() {
    if (solutionSteps == null || solutionSteps.isEmpty()) {
        return;
    }

    setRectanglesToCurrentState();
    if (isPlaying) {
        isPlaying = false;
        playButton.setText("Play");
        nextButton.setDisable(false);
        previousButton.setDisable(false);
        toStartButton.setDisable(false);
        toEndButton.setDisable(false);
        if (sequentialTransition != null) {
            sequentialTransition.stop();
        }
        return;
    }

    isPlaying = true;
    playButton.setText("Pause");
    nextButton.setDisable(true);
    previousButton.setDisable(true);
    toStartButton.setDisable(true);
    toEndButton.setDisable(true);

    // Create a sequential transition for all steps
    if (sequentialTransition != null) {
        sequentialTransition.stop();
    }
    sequentialTransition = new SequentialTransition();

    State currentState = solutionSteps.get(currentStep);
    int deltaX = 0, deltaY = 0, pieceIndex = -1, stepCount = 0;
    Timeline timeline = null;

    // track last movement of each piece
    ArrayList<int[]>[] deltas = new ArrayList[pieces.size()];
    for (int i = 0; i < deltas.length; i++) {
        deltas[i] = new ArrayList<>();
        deltas[i].add(new int[]{0, 0});
    }

    // iterate through the solution steps

```

```

    int step;
    for (step = currentStep; step < solutionSteps.size() - 1 && isPlaying;
step++) {
        State nextState = solutionSteps.get(step + 1);

        timeline = null;

        // search for a piece that moves
        for (int i = 0; i < pieces.size(); i++) {
            Piece piece = currentState.getPieces().get(i);
            Piece nextPiece = nextState.getPieces().get(i);
            if (piece.getPosI() != nextPiece.getPosI() || piece.getPosJ() !=
nextPiece.getPosJ()) {
                // piece moved
                stepCount++;
                if (pieceIndex == -1) {
                    stepCount = 0;
                    pieceIndex = i;
                    deltaX = nextPiece.getPosJ() - piece.getPosJ();
                    deltaY = nextPiece.getPosI() - piece.getPosI();
                } else if (pieceIndex == i) {
                    deltaX += nextPiece.getPosJ() - piece.getPosJ();
                    deltaY += nextPiece.getPosI() - piece.getPosI();
                } else {
                    PieceRectangle previousRect =
boardRectangles.get(pieceIndex);
                    int[] lastDelta =
deltas[pieceIndex].remove(deltas[pieceIndex].size() - 1);
                    int[] newDelta = new int[]{lastDelta[0] + deltaX,
lastDelta[1] + deltaY};
                    timeline = createPieceTimeline(previousRect, newDelta[0],
newDelta[1]);
                    deltas[pieceIndex].add(new int[]{newDelta[0],
newDelta[1]});

                    final int currentStepFinal = stepCount;
                    timeline.setOnFinished(e -> {
                        currentStep += currentStepFinal;
                        updateStepCounterLabel();
                    });

                    pieceIndex = i;
                    stepCount = 0;
                    deltaX = nextPiece.getPosJ() - piece.getPosJ();
                    deltaY = nextPiece.getPosI() - piece.getPosI();
                }
            }
            currentState = nextState;
            break;

```



```

        }
    }

    if (timeline == null) {
        continue;
    }
    sequentialTransition.getChildren().add(timeline);
}

if (pieceIndex != -1) {
    int[] lastDelta = deltas[pieceIndex].remove(deltas[pieceIndex].size() -
1);
    int[] newDelta = new int[]{lastDelta[0] + deltaX, lastDelta[1] +
deltaY};

    Piece pieceOnPlay =
solutionSteps.get(currentStep).getPieces().get(pieceIndex);

    // debug condition
    // System.out.println("is primary piece: " + (pieceOnPlay instanceof
PrimaryPiece));
    // System.out.println("is solved: " + isSolved);
    // System.out.println("is last step: " + (step + stepCount ==
solutionSteps.size() - 1));
    // System.out.println("step: " + step);
    // System.out.println("step count: " + stepCount);
    if (pieceOnPlay instanceof PrimaryPiece && isSolved && step + stepCount
>= solutionSteps.size() - 1) {
        if (currentState.getPieces().get(pieceIndex).isVertical()) {
            newDelta[1] = board.getWinPosI() - pieceOnPlay.getPosI() -
(board.getWinPosI() == 0 ? pieceOnPlay.getHeight() - 1 : 0);
        } else {
            newDelta[0] = board.getWinPosJ() - pieceOnPlay.getPosJ() -
(board.getWinPosJ() == 0 ? pieceOnPlay.getWidth() - 1 : 0);
        }
    }
    timeline = createPieceTimeline(boardRectangles.get(pieceIndex),
newDelta[0], newDelta[1]);
    final int currentStepFinal = stepCount;
    timeline.setOnFinished(e -> {
        currentStep += currentStepFinal;
        updateStepCounterLabel();
    });
    sequentialTransition.getChildren().add(timeline);
}

```

```

        sequentialTransition.setOnFinished(e -> {
            currentStep = solutionSteps.size() - 1;
            isPlaying = false;
            playButton.setText("Play");
            nextButton.setDisable(false);
            previousButton.setDisable(false);
            toStartButton.setDisable(false);
            toEndButton.setDisable(false);
        });

        currentStep++;
        updateStepCounterLabel();
        sequentialTransition.play();
    }

@FXML
private void onClickToStart() {
    if (!isPlaying) {
        setRectanglesToCurrentState();
        currentStep = 0;
        setRectanglesToCurrentState();
        updateStepCounterLabel();
    }
}

@FXML
private void onClickToEnd() {
    if (!isPlaying) {
        setRectanglesToCurrentState();
        currentStep = solutionSteps.size() - 1;
        setRectanglesToCurrentState();
        updateStepCounterLabel();
    }
}

@FXML
private void onClickNext() {
    if (!isPlaying) {
        setRectanglesToCurrentState();
        nextStep();
    }
}

private void nextStep() {
    if (currentStep < solutionSteps.size() - 1) {
        State currentState = solutionSteps.get(currentStep);

```

```

        State nextState = solutionSteps.get(++currentStep);
        for (int i = 0; i < pieces.size(); i++) {
            Piece piece = currentState.getPieces().get(i);
            Piece nextPiece = nextState.getPieces().get(i);
            if (piece.getPosI() != nextPiece.getPosI() || piece.getPosJ() !=
nextPiece.getPosJ()) {
                int nextPieceI = nextPiece.getPosI();
                int nextPieceJ = nextPiece.getPosJ();

                if (currentStep == solutionSteps.size() - 1 && piece instanceof
PrimaryPiece) {
                    nextPieceI = board.getWinPosI() == 0 ? -piece.getHeight() +
1 : board.getWinPosI();
                    nextPieceJ = board.getWinPosJ() == 0 ? -piece.getWidth() +
1 : board.getWinPosJ();
                }

                int deltaX = nextPieceJ - piece.getPosJ();
                int deltaY = nextPieceI - piece.getPosI();

                movePieceRectangle(boardRectangles.get(i), deltaX, deltaY);
            }
        }
        updateStepCounterLabel();
    }
}

@FXML
private void onClickPrevious() {
    if (!isPlaying) {
        setRectanglesToCurrentState();
        previousStep();
    }
}

private void previousStep() {
    if (currentStep > 0) {
        State currentState = solutionSteps.get(currentStep);
        State prevState = solutionSteps.get(--currentStep);
        for (int i = 0; i < pieces.size(); i++) {
            Piece piece = currentState.getPieces().get(i);
            Piece prevPiece = prevState.getPieces().get(i);
            if (piece.getPosI() != prevPiece.getPosI() || piece.getPosJ() !=
prevPiece.getPosJ()) {
                int pieceI = piece.getPosI();
                int pieceJ = piece.getPosJ();
                if (currentStep+1 == solutionSteps.size() - 1 && piece

```

```

instanceof PrimaryPiece) {
    pieceI = board.getWinPosI() == 0 ? -piece.getHeight() + 1 :
board.getWinPosI();
    pieceJ = board.getWinPosJ() == 0 ? -piece.getWidth() + 1 :
board.getWinPosJ();
}
    int deltaX = prevPiece.getPosJ() - pieceJ;
    int deltaY = prevPiece.getPosI() - pieceI;
    movePieceRectangle(boardRectangles.get(i), deltaX, deltaY);
    break;
}
}
updateStepCounterLabel();
}

private void setRectanglesToCurrentState() {
    if (currentStep < solutionSteps.size()) {
        State currentState = solutionSteps.get(currentStep);
        for (int i = 0; i < pieces.size(); i++) {
            Piece piece = currentState.getPieces().get(i);
            PieceRectangle rect = boardRectangles.get(i);
            int pieceI = piece.getPosI();
            int pieceJ = piece.getPosJ();
            if (currentStep == solutionSteps.size() - 1 && piece instanceof
PrimaryPiece) {
                pieceI = board.getWinPosI() == 0 ? -piece.getHeight() + 1 :
board.getWinPosI();
                pieceJ = board.getWinPosJ() == 0 ? -piece.getWidth() + 1 :
board.getWinPosJ();
            }
            double x = pieceJ * gridSize + GRID_BORDER;
            double y = pieceI * gridSize + GRID_BORDER;
            rect.setX(x);
            rect.setY(y);
        }
        updateStepCounterLabel();
    }
}

private void updateStepCounterLabel() {
    if (solutionSteps != null && !solutionSteps.isEmpty() && isSolved) {
        stepCounterLabel.setText("Step: " + (currentStep) + " out of " +
(solutionSteps.size()-1));
    } else {
        stepCounterLabel.setText("Step: -");
    }
}

```

```
}
```

```
//
```

```
=====
```

```
// Dragging Logic
```

```
//
```

```
=====
```

```
private void makeDraggable(javafx.scene.Node node) {  
    node.setOnMousePressed(event -> onRectPressed(event, node));  
    node.setOnMouseDragged(event -> onRectDragged(event, node));  
    node.setOnMouseReleased(event -> onRectReleased(event, node));  
}
```

```
private void onRectDragged(MouseEvent event, javafx.scene.Node node) {  
    if (!(node instanceof PieceRectangle)) return;  
    PieceRectangle boardRectangle = (PieceRectangle) node;
```

```
    if (boardRectangle.isVertical()) {  
        // Vertical piece - move along Y axis  
        double deltaY = event.getSceneY() - dragStartY;  
        double newY = rectStartY + deltaY;  
  
        if (deltaY < 0) {  
            double topmostY = calculateTopmostY(boardRectangle);  
            newY = Math.max(topmostY, newY);  
        } else {  
            double bottommostY = calculateBottommostY(boardRectangle);  
            newY = Math.min(bottommostY, newY);  
        }  
        boardRectangle.setY(newY);  
    } else {  
        // Horizontal piece - move along X axis  
        double deltaX = event.getSceneX() - dragStartX;  
        double newX = rectStartX + deltaX;  
        if (deltaX < 0) {  
            double leftmostX = calculateLeftmostX(boardRectangle);  
            newX = Math.max(leftmostX, newX);  
        } else {  
            double rightmostX = calculateRightmostX(boardRectangle);  
            newX = Math.min(rightmostX, newX);  
        }  
        boardRectangle.setX(newX);  
    }  
}
```

```

}

private void onRectPressed(MouseEvent event, javafx.scene.Node node) {
    if (!(node instanceof PieceRectangle)) return;
    PieceRectangle boardRectangle = (PieceRectangle) node;

    dragStartX = event.getSceneX();
    dragStartY = event.getSceneY();
    rectStartX = boardRectangle.getX();
    rectStartY = boardRectangle.getY();
    boardRectangle.setFill(javafx.scene.paint.Color.YELLOW);
}

private void onRectReleased(MouseEvent event, javafx.scene.Node node) {
    if (!(node instanceof PieceRectangle)) return;
    PieceRectangle boardRectangle = (PieceRectangle) node;

    boardRectangle.setFill(boardRectangle.getColor());

    // Snap to grid
    double finalX = snapToGrid(boardRectangle.getX());
    double finalY = snapToGrid(boardRectangle.getY());

    boardRectangle.setX(finalX);
    boardRectangle.setY(finalY);
}

```

```
//
```

```
=====
// Collision detection and snapping logic
//
=====
```

```

private double calculateLeftmostX(PieceRectangle rect) {

    List<PieceRectangle> sortedRects = new ArrayList<>(boardRectangles);
    sortedRects.sort((r1, r2) -> Double.compare(r1.getX(), r2.getX()));

    int index = sortedRects.indexOf(rect);
    if (index == -1) return rectStartX;

    for (int i = index - 1; i >= 0; i--) {
        PieceRectangle otherRect = sortedRects.get(i);
    }
}

```

```

        if (rect.getY() >= otherRect.getY() && rect.getY() <= otherRect.getY()
+ otherRect.getHeight()) {
            return otherRect.getX() + otherRect.getWidth();
        }
    }

    if (rect.isPrimaryPiece() && board.getWinPosJ() == 0) {
        return -rect.getWidth() + gridSize;
    }

    return 0;
}

private double calculateTopmostY(PieceRectangle rect) {
    List<PieceRectangle> sortedRects = new ArrayList<>(boardRectangles);
    sortedRects.sort((r1, r2) -> Double.compare(r1.getY(), r2.getY()));

    int index = sortedRects.indexOf(rect);
    if (index == -1) return rectStartY;

    for (int i = index - 1; i >= 0; i--) {
        PieceRectangle otherRect = sortedRects.get(i);
        if (rect.getX() >= otherRect.getX() && rect.getX() <= otherRect.getX()
+ otherRect.getWidth()) {
            return otherRect.getY() + otherRect.getHeight();
        }
    }

    if (rect.isPrimaryPiece() && board.getWinPosI() == 0) {
        return -rect.getHeight() + gridSize;
    }

    return 0;
}

private double calculateRightmostX(PieceRectangle rect) {
    List<PieceRectangle> sortedRects = new ArrayList<>(boardRectangles);
    sortedRects.sort((r1, r2) -> Double.compare(r1.getX(), r2.getX()));

    int index = sortedRects.indexOf(rect);
    if (index == -1) return rectStartX;

    for (int i = index + 1; i < sortedRects.size(); i++) {
        PieceRectangle otherRect = sortedRects.get(i);
        if (rect.getY() >= otherRect.getY() && rect.getY() <= otherRect.getY()
+ otherRect.getHeight()) {
            return otherRect.getX() - rect.getWidth();

```

```

    }
}

// if is primary piece and the goal is to the right
if (rect.isPrimaryPiece() && board.getWinPosJ() == board.getWidth() - 1) {
    return boardPane.getWidth() - gridSize;
}

return boardPane.getWidth() - rect.getWidth();
}

private double calculateBottommostY(PieceRectangle rect) {
    List<PieceRectangle> sortedRects = new ArrayList<>(boardRectangles);
    sortedRects.sort((r1, r2) -> Double.compare(r1.getY(), r2.getY()));

    int index = sortedRects.indexOf(rect);
    if (index == -1) return rectStartY;

    for (int i = index + 1; i < sortedRects.size(); i++) {
        PieceRectangle otherRect = sortedRects.get(i);
        if (rect.getX() >= otherRect.getX() && rect.getX() <= otherRect.getX()
+ otherRect.getWidth()) {
            return otherRect.getY() - rect.getHeight();
        }
    }

    if (rect.isPrimaryPiece() && board.getWinPosI() == board.getHeight() - 1) {
        return boardPane.getHeight() - gridSize;
    }

    return boardPane.getHeight() - gridSize;
}

private double snapToGrid(double value) {
    return Math.round(value / gridSize) * gridSize + GRID_BORDER;
}

//
=====
// Button actions
//
=====

@FXML
private void onClickApplyConfiguration() {

```



```

clearAlerts();
clearBoard();

try {
    initializeBoard();
    showAlert("Configuration applied successfully!", "SUCCESS");

    isConfigured = true;
} catch (IllegalArgumentException e) {
    showAlert("Invalid board configuration: " + e.getMessage(), "ERROR");
    playButton.setDisable(true);
    nextButton.setDisable(true);
    previousButton.setDisable(true);
}
}

@FXML
private void onClickUploadFile() {
    clearAlerts();
    clearBoard();
    clearFileName();
    FileChooser fileChooser = new FileChooser();
    fileChooser.setInitialDirectory(new File("../test"));
    fileChooser.setTitle("Open Rush Hour Board File");
    fileChooser.getExtensionFilters().add(
        new FileChooser.ExtensionFilter("Text Files", "*.txt"));

    File file = fileChooser.showOpenDialog(new Stage());
    if (file != null) {
        try {
            String content = new String(Files.readAllBytes(file.toPath()));
            if(validateBoardInput(content)) {
                boardTextArea.setText(content);
                showFileName(file.getName());
                currentFile = file;
                showAlert("File loaded successfully!", "SUCCESS");
            }
        } catch (Exception e) {
            showAlert("Error reading file: " + e.getMessage(), "ERROR");
        }
    }
}

private String formatDuration(long milliseconds) {
    if (milliseconds < 1000) {
        return milliseconds + "ms";
    } else if (milliseconds < 60000) {

```

```

        return String.format("%.2fs", milliseconds / 1000.0);
    } else {
        long minutes = milliseconds / 60000;
        long seconds = (milliseconds % 60000) / 1000;
        return String.format("%dm %ds", minutes, seconds);
    }
}

@FXML
private void onClickSolve() {
    if (!isConfigured) {
        showAlert("Please configure the board first!", "ERROR");
        return;
    }

    playButton.setDisable(true);
    nextButton.setDisable(true);
    previousButton.setDisable(true);
    toStartButton.setDisable(true);
    toEndButton.setDisable(true);
    stepCounterLabel.setVisible(false);

    // Reset isSolved if you intend to allow re-solving
    isSolved = false;

    clearAlerts();

    SearchMode searchMode;
    switch (algorithmChoiceBox.getValue()) {
        case "UCS":
            searchMode = SearchMode.UCS;
            break;
        case "A*":
            searchMode = SearchMode.A_STAR;
            break;
        case "GBFS":
            searchMode = SearchMode.GREEDY;
            break;
        default:
            showAlert("Invalid algorithm selected", "ERROR");
            return;
    }

    String heuristic = heuristicChoiceBox.getValue();

    Task<Boolean> solveTask = new Task<Boolean>() {

```

```

@Override
protected Boolean call() throws Exception {
    Heuristics.heuristicType = heuristic.toUpperCase().replace(" ",
" _");

    // solutionSteps = new ArrayList<>();
    // solutionSteps.add(new rushhour.State(board, pieces, null, null,
0, primaryPiece));
    // solutionSteps.addAll(solver.solve(searchMode));

    solutionSteps = solver.solve(searchMode);

    boolean foundSolution = solver.hasFoundSolution();
    return foundSolution;
}
};

setupSolveTask(solveTask);
solvingStartTime = System.currentTimeMillis();
new Thread(solveTask).start();
}

// Modify setupSolveTask to handle Task<Boolean>
private void setupSolveTask(Task<Boolean> solveTask) {
    solveTask.setOnSucceeded(e -> {
        boolean foundSolution = solveTask.getValue();
        isSolved = foundSolution;

        if (foundSolution) {
            long solvingDuration = System.currentTimeMillis() -
solvingStartTime;
            String durationMessage = formatDuration(solvingDuration);
            long steps = solutionSteps.size()-1;

            if (heuristicChoiceBox.getValue().isBlank()) {
                showAlert("Solution found using " +
algorithmChoiceBox.getValue() + "!\n" +
                    "Steps: " + steps + "\nTime: " + durationMessage,
"SUCCESS");
            } else {
                showAlert("Solution found using " +
algorithmChoiceBox.getValue() + " with " + (heuristicChoiceBox.getValue()) + "
heuristic!\n" +
                    "Steps: " + steps + "\nTime: " + durationMessage,
"SUCCESS");
            }
        }
    });
}

```

```

        currentStep = 0;
        playButton.setDisable(false);
        nextButton.setDisable(false);
        previousButton.setDisable(false);
        toStartButton.setDisable(false);
        toEndButton.setDisable(false);
        stepCounterLabel.setVisible(true);
        updateStepCounterLabel();
    } else {
        showAlert("No solution found.", "ERROR");
        playButton.setDisable(true);
        nextButton.setDisable(true);
        previousButton.setDisable(true);
        toStartButton.setDisable(true);
        toEndButton.setDisable(true);
        stepCounterLabel.setVisible(false);
    }
});

solveTask.setOnFailed(e -> {
    isSolved = false;
    Throwable exception = solveTask.getException();
    showAlert("Error during solving: " + exception.getMessage(), "ERROR");
    playButton.setDisable(true);
    nextButton.setDisable(true);
    previousButton.setDisable(true);
    stepCounterLabel.setVisible(false);
    exception.printStackTrace();
});
}

private boolean validateBoardInput(String input) {
    if(input == null || input.trim().isEmpty()) {
        showAlert("Board configuration cannot be empty", "ERROR");
        return false;
    }

    return true;
}

private void showAlert(String message, String type) {
    switch(type.toUpperCase()) {
        case "ERROR":
            alertMessageText.setFill(javafx.scene.paint.Color.RED);
            break;
        case "SUCCESS":
            alertMessageText.setFill(javafx.scene.paint.Color.GREEN);

```

```

        break;
    default:
        alertMessageText.setFill(javafx.scene.paint.Color.BLACK);
    }
    alertMessageText.setText(message);
}

private void showFileName(String fileName) {
    filenameText.setText("File: " + fileName);
}

private void clearAlerts() {
    alertMessageText.setText("");
}

private void clearBoard() {
    try {
        if (boardPane != null) boardPane.getChildren().clear();
        if (boardRectangles != null) boardRectangles.clear();
        if (solutionSteps != null) solutionSteps.clear();
        isSolved = false;
        isPlaying = false;
        isConfigured = false;
        playButton.setDisable(true);
        nextButton.setDisable(true);
        previousButton.setDisable(true);
        toStartButton.setDisable(true);
        toEndButton.setDisable(true);
        stepCounterLabel.setVisible(isSolved);
    } catch (Exception e) {
        System.err.println("Error: " + e);
    }
}

private void clearFileName() {
    filenameText.setText("");
}

private int calculateGridSize(int row, int col) {
    return Math.min(BOARD_PANE_WIDTH / col, BOARD_PANE_HEIGHT / row);
}

//
=====
// PieceRectangle class
//

```

```

=====
private static class PieceRectangle extends Rectangle {
    private Color color;
    private boolean isPrimaryPiece;

    public PieceRectangle(double width, double height) {
        super(width, height);
        this.color = Color.BLUE;
        this.isPrimaryPiece = false;
        setFill(color);
    }

    public void setColor(Color color) {
        this.color = color;
        setFill(color);
    }

    public Color getColor() {
        return color;
    }

    public void setPrimaryPiece(boolean isPrimaryPiece) {
        this.isPrimaryPiece = isPrimaryPiece;
    }

    public boolean isPrimaryPiece() {
        return isPrimaryPiece;
    }

    public boolean isVertical() {
        return getHeight() > getWidth();
    }
}

```

```
//
```

```
=====
// GoalTriangle class

```

```
//
```

```

=====
private static class GoalTriangle extends Polygon {
    private final Rotate rotate;

    public GoalTriangle(double base, double height) {
        super(0.0, 0.0,
              0.0, height,
              base, height / 2.0);
    }
}

```

```
        rotate = new Rotate();
        rotate.setPivotX(base / 2);
        rotate.setPivotY(height / 2);
        this.getTransforms().add(rotate);
    }

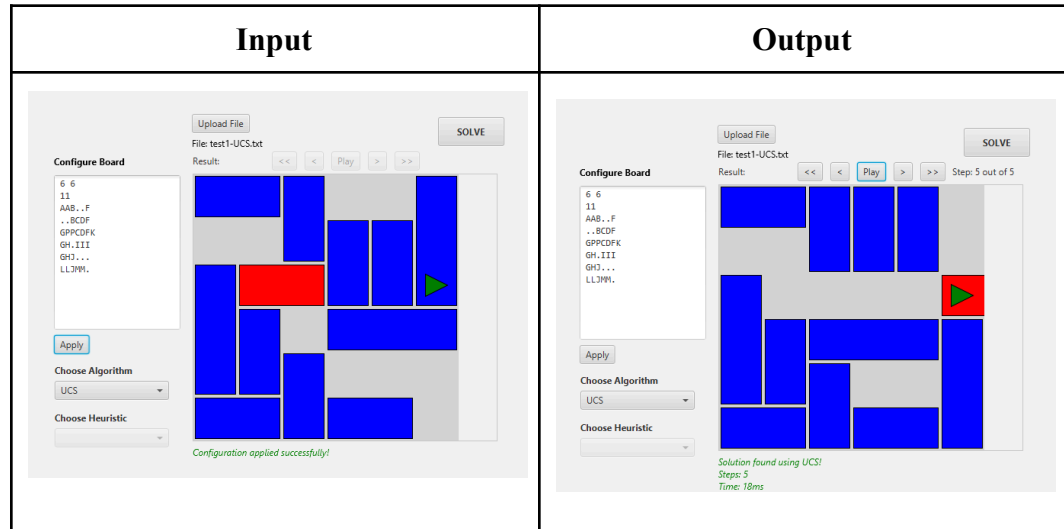
    public void setAngle(double angle) {
        rotate.setAngle(angle);
    }
}
}
```

BAB IV

PERCOBAAN DAN ANALISIS

4.1. Percobaan

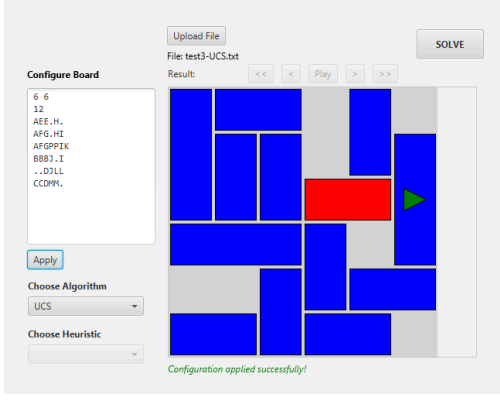
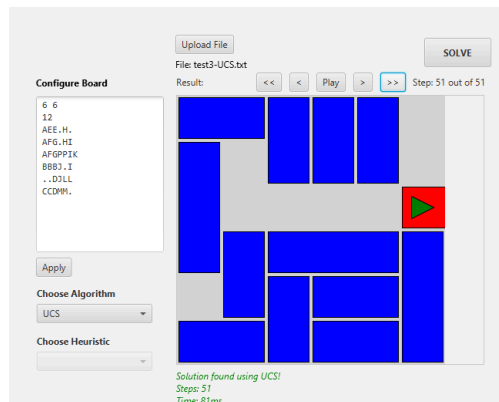
4.1.1. Test Case 1: UCS dengan Solusi di Kanan




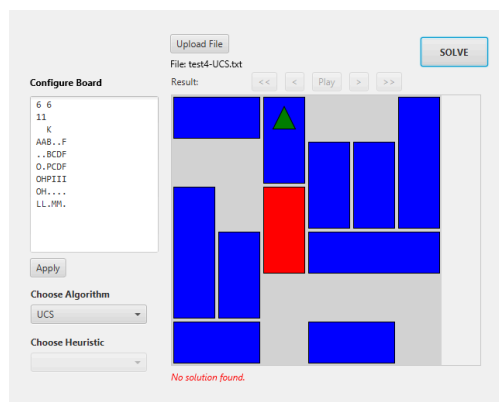
4.1.2. Test Case 2: UCS dengan Solusi di Bawah



4.1.3. Test Case 3: UCS dengan Solusi Panjang

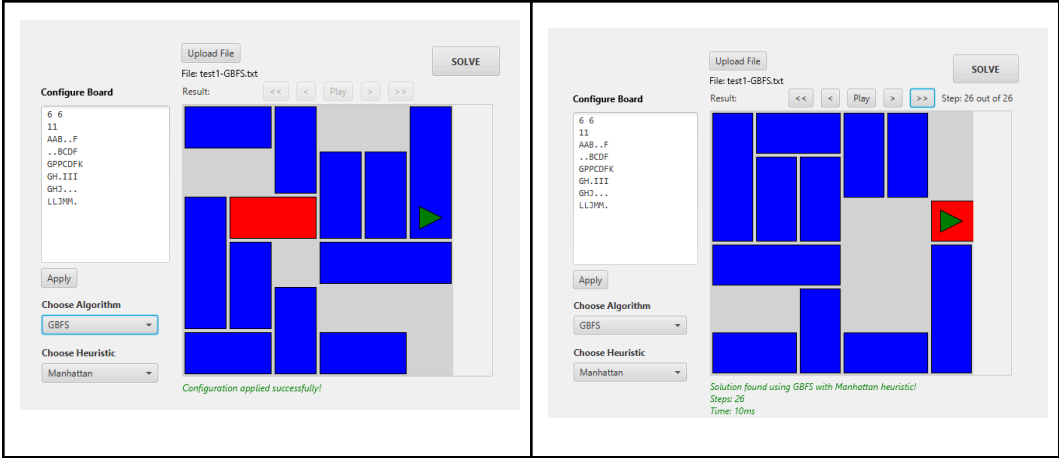
Input	Output
	

4.1.4. Test Case 4: UCS Tanpa Solusi

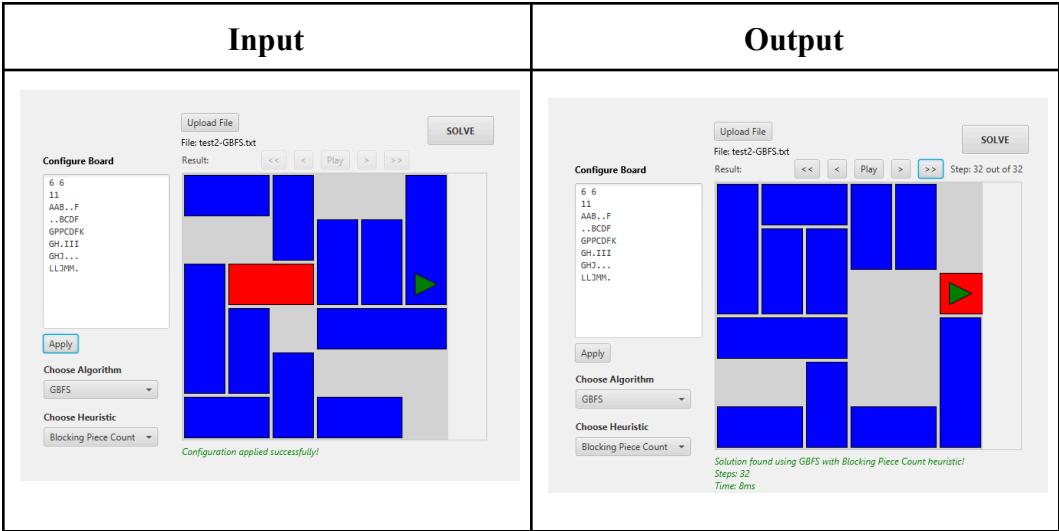
Input	Output
	

4.1.5. Test Case 1: GBFS dengan Heuristik Manhattan

Input	Output
-------	--------



4.1.6. Test Case 2: GBFS dengan Heuristik Blocking Piece Count



4.1.7. Test Case 3: GBFS dengan Solusi Panjang

Input	Output
-------	--------

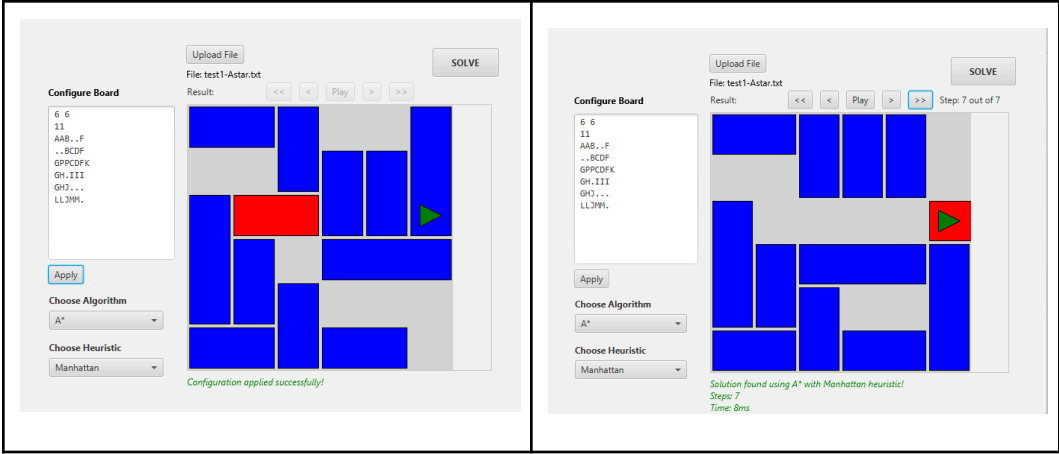


4.1.8. Test Case 4: GBFS Tanpa Solusi

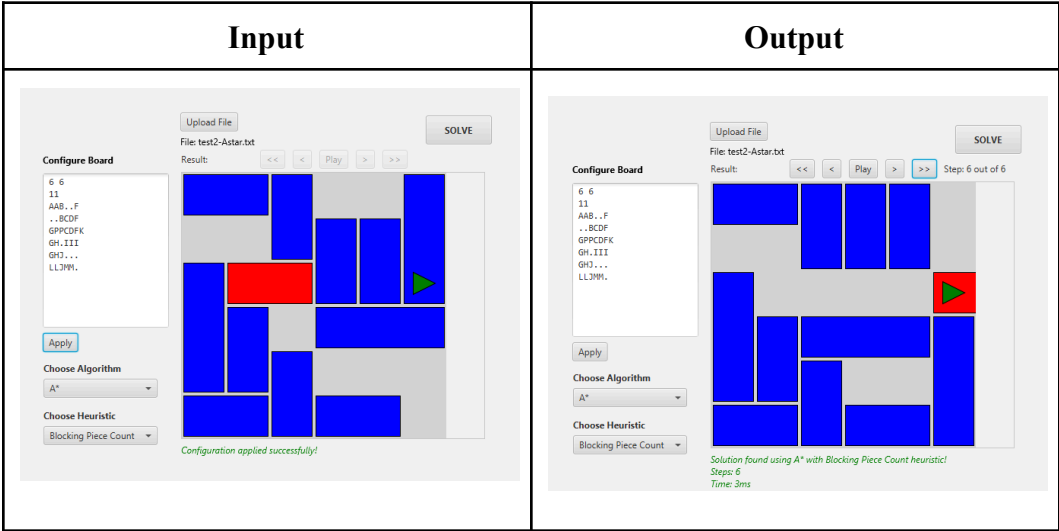


4.1.9. Test Case 1: A* dengan Heuristik Manhattan





4.1.10. Test Case 2: A* dengan Heuristik Blocking Piece Count

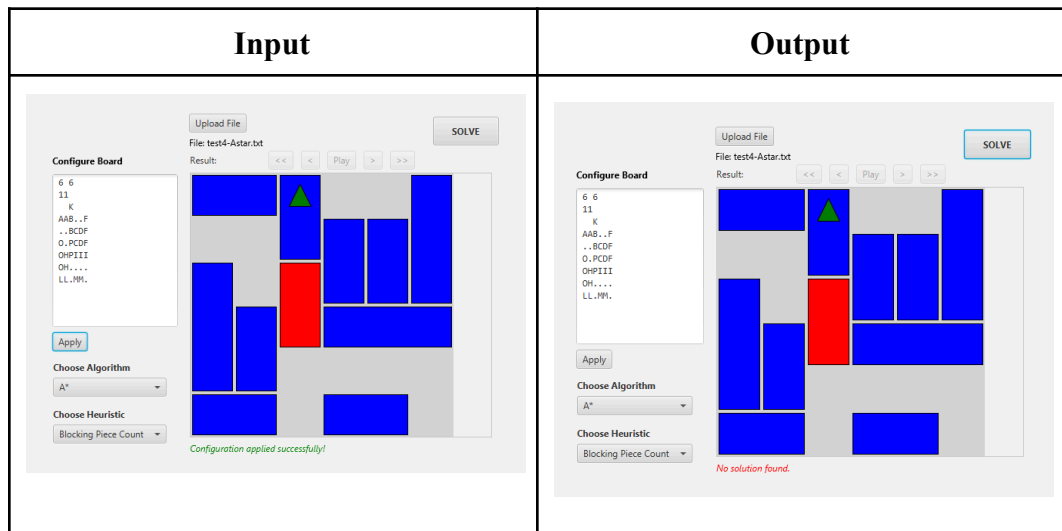


4.1.11. Test Case 3: A* dengan Solusi Panjang





4.1.12. Test Case 4: A* Tanpa Solusi



4.2. Analisis

4.2.1. Kompleksitas Algoritma

Setelah melakukan implementasi dan pengujian terhadap algoritma Uniform Cost Search (UCS), Greedy Best-First Search (Greedy BFS), dan A* untuk menyelesaikan permainan Rush Hour, analisis terhadap kompleksitas algoritma-algoritma tersebut dalam konteks program yang dikembangkan menjadi penting. Kompleksitas ini akan ditinjau dari segi waktu dan ruang.

Secara umum, semua algoritma yang diimplementasikan (UCS, Greedy BFS, dan A*) menggunakan struktur data PriorityQueue untuk openSet, dan HashSet untuk closedSet. Operasi dasar pada PriorityQueue (seperti add dan poll) memiliki kompleksitas waktu rata-rata $O(\log V)$, di mana V adalah jumlah state dalam antrian. Operasi pada HashSet (seperti add dan contains) memiliki kompleksitas waktu rata-rata $O(1)$, dengan asumsi fungsi hash yang baik dan tidak banyak terjadi kolisi.

Kompleksitas waktu dari algoritma-algoritma ini sangat dipengaruhi oleh jumlah state yang diekspansi dan operasi yang dilakukan untuk setiap state. Misalkan b adalah faktor percabangan (branching factor) atau jumlah suksesor rata-rata yang dapat dihasilkan dari satu state, dan d adalah kedalaman solusi (jumlah langkah dalam solusi). Untuk setiap state yang diambil dari openSet:

1. Operasi poll dari PriorityQueue: $O(\log|\text{openSet}|)$.
2. Generasi suksesor: Setiap pemanggilan fungsi `generateSuccessor()`, dilakukan penyalinan penyalinan (*copy constructor*) setiap *piece* dan *board* untuk setiap kemungkinan pergerakan *piece* yang ada. Jika ada P jumlah pieces, dan ukuran *board* adalah $A \times B$, maka kompleksitasnya adalah $O(P \cdot (P + AB)) = O(P^2 + PAB)$.
3. Iterasi melalui suksesor: Untuk setiap suksesor (sebanyak b):
 - Pengecekan `closedSet.contains(successor)`: Rata-rata $O(1)$
 - Penambahan ke `openSet.add(successor)`: $O(\log|\text{openSet}|)$
 - Penambahan ke `closedSet.add(successor)`: Rata-rata $O(1)$.

Jika M adalah jumlah total state (node) yang diekspansi hingga solusi ditemukan atau openSet kosong, maka kompleksitas waktu kasar dapat diestimasi sebagai $M \times \text{KompleksitasPerState}$.

- Untuk UCS dan A*, dalam kasus terburuk, kompleksitas waktunya bisa menjadi eksponensial, yaitu sekitar $O(b^d)$ jika seluruh ruang harus dijelajahi. Operasi dominan per state adalah poll dan add pada

PriorityQueue, serta generasi suksesor. Jadi, kompleksitas waktunya adalah $O(b^d \cdot \log|openState| + b^d \cdot (P^2 + PAB))$

- Untuk Greedy BFS, meskipun seringkali menemukan solusi lebih cepat, dalam kasus terburuk ia juga bisa menjelajahi banyak state, dan kompleksitas waktunya juga bisa menjadi eksponensial dalam beberapa skenario. Namun, karena ia tidak menjamin optimalitas, ia mungkin berhenti lebih awal.
- A* memiliki sifat bahwa ia tidak akan mengeksplorasi lebih banyak node daripada UCS jika heuristiknya konsisten (dan admissible). Dengan heuristik yang baik, A* bisa jauh lebih cepat.

Kemudian, kompleksitas ruang dari algoritma utamanya ditentukan oleh jumlah state yang disimpan dalam openSet dan closedSet. Dalam kasus terburuk, kedua struktur data ini bisa menyimpan hingga $O(b^d)$ state. Setiap *state* menyimpan salinan *Board* dan *List of Pieces*. Maka, kompleksitas ruangnya adalah $O(b^d \cdot P + b^d \cdot AB)$ dengan P adalah jumlah *piece*, serta $A \times B$ adalah ukuran *board*.

4.2.2. Analisis Percobaan

Secara umum, berdasarkan hasil percobaan, program dapat mencari solusi pada permainan Rush Hour dengan baik dan dapat menentukan apakah dari konfigurasi yang diberikan dapat ditemukan solusi atau tidak. Hasil pengukuran performa berdasarkan waktu menunjukkan bahwa waktu berjalannya algoritma cukup cepat, yakni di bawah 30 milidetik untuk permasalahan dengan solusi pendek (dapat dicapai dengan kurang dari 10 langkah) dan di bawah 100 milidetik untuk permasalahan dengan kompleksitas yang lebih tinggi (solusi memiliki minimal lebih dari 50 langkah). Secara praktis, tidak ada perbedaan yang signifikan dari segi waktu untuk setiap algoritma. Namun, terlihat sedikit peningkatan waktu yang dibutuhkan bagi algoritma UCS dibandingkan yang lainnya.

Hasil yang membedakan setiap algoritma adalah jumlah langkah pada solusi rute yang ditemukannya. UCS secara umum berhasil menemukan solusi dengan jumlah langkah terkecil dibandingkan yang lain. Di sisi lain, GBFS memperoleh solusi yang secara umum memiliki jumlah langkah yang lebih banyak, yakni hingga lebih dari 20 langkah untuk permasalahan yang sederhana dan lebih dari 100 langkah untuk permasalahan kompleks. Hal ini terjadi karena UCS tidak menggunakan heuristik dan hanya bergantung pada nilai $g(n)$, sehingga ia mengeksplorasi semua *node* dengan biaya yang lebih rendah terlebih dahulu. Algoritma ini memastikan solusi yang diperoleh adalah optimal. Berbeda dengan UCS, A* dan GBFS menggunakan heuristik untuk mempercepat pencarian status kemenangan pada papan namun mengorbankan optimalitas. Heuristik menyebabkan kedua algoritma ini memiliki urutan penelusuran *node* yang lebih cepat mencapai status kemenangan. Namun, ketika mencapai solusi, jalur yang telah ditelusuri belum tentu merupakan jalur yang optimal. Dengan demikian, jumlah langkah yang didapatkan tidak seoptimal UCS.

LAMPIRAN

Poin	Ya	Tidak
1. Program berhasil dikompilasi tanpa kesalahan	✓	
2. Program berhasil dijalankan	✓	
3. Solusi yang diberikan program benar dan mematuhi aturan permainan	✓	
4. Program dapat membaca masukan berkas .txt dan menyimpan solusi berupa print board tahap per tahap dalam berkas .txt	✓	
5. [Bonus] Implementasi algoritma pathfinding alternatif		✓
6. [Bonus] Implementasi 2 atau lebih heuristik alternatif	✓	
7. [Bonus] Program memiliki GUI	✓	
8. Program dan laporan dibuat (kelompok) sendiri	✓	

REFERENSI

- Munir, R. (2025). *Penentuan Rute (Route/Path Planning) Bagian 1: BFS, DFS, UCS, Greedy Best First Search* [Materi kuliah]. Institut Teknologi Bandung. Diakses dari [https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-\(2025\)-Bagian1.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-(2025)-Bagian1.pdf)
- Munir, R. (2025). *Penentuan Rute (Route/Path Planning) Bagian 2: Algoritma A** [Materi kuliah]. Institut Teknologi Bandung. Diakses dari [https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-\(2025\)-Bagian1.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-(2025)-Bagian1.pdf)