

Laporan Tugas Kecil 3 IF2211 - Strategi Algoritma
Semester II Tahun Akademik 2024/2025

Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding



Disusun oleh:

M. Fariz Rifqi Rizqulloh 13523069

Nayaka Ghana Subrata 13523090

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
2025

DAFTAR ISI

DAFTAR ISI.....	2
DAFTAR GAMBAR.....	5
DAFTAR TABEL.....	6
BAB I: DESKRIPSI MASALAH.....	7
1.1 Deskripsi Masalah.....	7
1.2 Ilustrasi Kasus.....	8
BAB II: TEORI, PENYELESAIAN, DAN ALGORITMA.....	12
2.1 Algoritma Pencari Jalur (Pathfinding Algorithm).....	12
2.2 Algoritma Uniform Cost Search (UCS).....	13
2.2.1 Analisis Algoritma Uniform Cost Search (UCS).....	13
2.2.2 Langkah Penyelesaian.....	14
2.2.3 Pseudocode.....	15
2.3 Algoritma Greedy Best First Search (GBFS).....	17
2.3.1 Analisis Algoritma Greedy Best First Search (GBFS).....	17
2.3.2 Langkah Penyelesaian.....	17
2.3.3 Pseudocode.....	18
2.4 Algoritma A* (A-Star).....	20
2.4.1 Analisis Algoritma A* (A-Star).....	20
2.4.2 Langkah Penyelesaian.....	21
2.4.3 Pseudocode.....	21
2.5 Algoritma Dijkstra.....	24
2.5.1 Analisis Algoritma Dijkstra.....	24
2.5.2 Langkah Penyelesaian.....	25
2.5.3 Pseudocode.....	26
2.6 Algoritma Beam-Search.....	27
2.6.1 Analisis Algoritma Beam-Search.....	27
2.6.2 Langkah Penyelesaian.....	28
2.6.3 Pseudocode.....	29
2.7 Algoritma A* Iterative Deepening Search.....	30
2.7.1 Analisis Algoritma A* Iterative Deepening Search.....	30
2.7.2 Langkah Penyelesaian.....	31
2.7.3 Pseudocode.....	32
2.8 Strategi Heuristik.....	33
2.8.1 Manhattan Distance.....	33
2.8.2 Direct Distance Count.....	34
2.8.3 Blocking Count.....	34
2.8.4 Clearing Count.....	34

2.9 Analisis Kompleksitas Algoritma.....	35
BAB III: STRUKTUR REPOSITORY DAN SOURCE CODE.....	37
3.1. Struktur Repository.....	37
3.2. Source Code.....	42
3.2.1 Board.java.....	42
3.2.2 CompoundMove.java.....	56
3.2.3 Exit.java.....	57
3.2.4 FileParser.java.....	57
3.2.5 Main.java.....	66
3.2.6 Move.java.....	76
3.2.7 Orientation.java.....	77
3.2.8 Piece.java.....	77
3.2.9 Position.java.....	82
3.2.10 Solution.java.....	83
3.2.11 Solver.java.....	85
3.2.12 AnimationController.java.....	104
3.2.13 InputController.java.....	117
3.2.14 MainController.java.....	127
3.2.15 MatrixInputWindowController.java.....	160
3.2.16 VisualizationController.java.....	165
3.2.17 ZoomController.java.....	172
3.2.18 MainView.fxml.....	183
3.2.19 MatrixInputWindow.fxml.....	189
3.2.20 VisualizationView.fxml.....	189
3.2.21 GuiMain.java.....	192
3.2.22 styles.css.....	195
3.2.23 KessokuNoOwari.java.....	207
BAB IV: TEST CASES.....	212
4.1 Solusi Berhasil Ditemukan.....	212
4.1.1 Windows.....	212
4.1.2 Linux.....	214
4.2 Solusi Gagal Ditemukan.....	215
4.2.1 Windows.....	215
4.2.2 Linux.....	217
4.3 User Input Error.....	218
4.4 Perbandingan Algoritma yang digunakan.....	222
4.4.1 Testcase default.txt.....	222
4.4.2 Testcase misteri4.txt.....	225
4.4.3 Testcase tebaktebakanyuk_adahasilnyaapaenggakya.txt.....	228

4.4.4 Testcase wahgaknormalini😁.txt.....	231
4.5 Perbandingan Heuristik.....	235
4.5.1 Testcase default.txt.....	235
4.5.2 Testcase misteri4.txt.....	237
4.5.3 Testcase tebaktebakanyuk_adahasilnyaapaenggakya.txt.....	240
4.5.4 Testcase wahgaknormalini😁.txt.....	242
4.6 Kasus Spesial.....	245
4.6.1 Testcase wlntr.txt (7x7).....	245
4.6.2 Testcase katousignature.txt (8x8).....	248
BAB V: HASIL ANALISIS PERCOBAAN.....	253
BAB VI: BONUS.....	255
6.1 Algoritma Pencarian Jalur Alternatif.....	255
6.3 Graphical User Interface (GUI).....	255
6.3.1 Input.....	255
6.3.2 Algorithm Selection dan Solving.....	258
6.3.3 Canvas.....	259
6.3.4 Musik.....	261
LAMPIRAN.....	263
DAFTAR PUSTAKA.....	264
AKHIR KATA.....	265

DAFTAR GAMBAR

Gambar 1. Rush Hour Puzzle.....	7
Gambar 2. Awal permainan game Rush Hour.....	9
Gambar 3. Gerakan pertama game Rush Hour.....	10
Gambar 4. Gerakan seterusnya dan pemain menyelesaikan permainan.....	11
Gambar 5. Contoh Pencarian rute di antara dua titik.....	12
Gambar 6. Contoh Pencarian rute dari titik S ke titik G.....	14
Gambar 7. Contoh Pencarian rute dengan menggunakan algoritma Dijkstra.....	25
Gambar 8. Testcase 1.txt.....	212
Gambar 9. Testcase 2.txt.....	213
Gambar 10. Testcase 3.txt.....	213
Gambar 11. Testcase 1.txt.....	214
Gambar 12. Testcase 2.txt.....	214
Gambar 13. Testcase 3.txt.....	215
Gambar 14. Testcase sempit.txt.....	215
Gambar 15. Testcase gaksejajar.txt.....	216
Gambar 16. Testcase kenablokirnjir.txt.....	216
Gambar 17. Testcase sempit.txt.....	217
Gambar 18. Testcase gaksejajar.txt.....	217
Gambar 19. Testcase kenablokirnjir.txt.....	218
Gambar 20. File input.....	256
Gambar 21. Text input.....	257
Gambar 22. Matrix input.....	258
Gambar 23. Algorithm selection dan solving.....	259
Gambar 24. Tampilan canvas.....	260
Gambar 25. Implementasi zoom.....	261
Gambar 26. Daftar musik yang ada pada program.....	262

DAFTAR TABEL

Tabel 1 Perbandingan kompleksitas antar algoritma.....	35
Tabel 2 Poin yang dikerjakan.....	263

BAB I: DESKRIPSI MASALAH

1.1 Deskripsi Masalah



Gambar 1. Rush Hour Puzzle

(Sumber: <https://www.thinkfun.com/en-US/products/educational-games/rush-hour-76582>)

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

Komponen penting dari permainan Rush Hour terdiri dari:

1. **Papan** – Papan merupakan tempat permainan dimainkan.

Papan terdiri atas *cell*, yaitu sebuah *singular point* dari papan. Sebuah *piece* akan menempati *cell-cell* pada papan. Ketika permainan dimulai, semua *piece* telah diletakkan di dalam papan dengan konfigurasi tertentu berupa lokasi *piece* dan orientasi, antara horizontal atau vertikal.

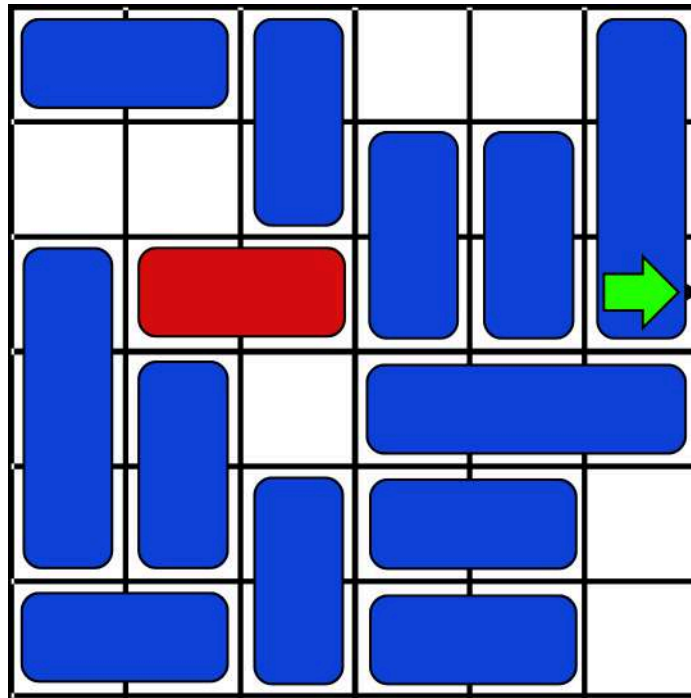
Hanya ***primary piece*** yang dapat digerakkan **keluar papan melewati pintu keluar**. *Piece* yang bukan *primary piece* tidak dapat digerakkan keluar

papan. Papan memiliki satu pintu keluar yang pasti berada di dinding papan dan sejajar dengan orientasi *primary piece*.

2. **Piece** – *Piece* adalah sebuah kendaraan di dalam papan. Setiap *piece* memiliki posisi, ukuran, dan orientasi. Orientasi sebuah *piece* hanya dapat berupa horizontal atau vertikal–tidak mungkin diagonal. *Piece* dapat memiliki beragam ukuran, yaitu jumlah *cell* yang ditempati oleh *piece*. Secara standar, variasi ukuran sebuah *piece* adalah *2-piece* (menempati 2 *cell*) atau *3-piece* (menempati 3 *cell*). Suatu *piece* tidak dapat digerakkan melewati/menembus *piece* yang lain.
3. **Primary Piece** – Primary piece adalah kendaraan utama yang harus dikeluarkan dari papan (biasanya berwarna merah). Hanya boleh terdapat satu *primary piece*.
4. **Pintu Keluar** – *Pintu keluar* adalah tempat *primary piece* dapat digerakkan keluar untuk menyelesaikan permainan.
5. **Gerakan** — *Gerakan* yang dimaksudkan adalah pergeseran *piece* di dalam permainan. *Piece* hanya dapat bergerak/bergeser lurus sesuai orientasinya (atas-bawah jika vertikal dan kiri-kanan jika horizontal). Suatu *piece* tidak dapat digerakkan melewati/menembus *piece* yang lain.

1.2 Ilustrasi Kasus

Diberikan sebuah papan berukuran 6 x 6 dengan 12 *piece* kendaraan dengan 1 *piece* merupakan *primary piece*. *Piece* ditempatkan pada papan dengan posisi dan orientasi sebagai berikut.

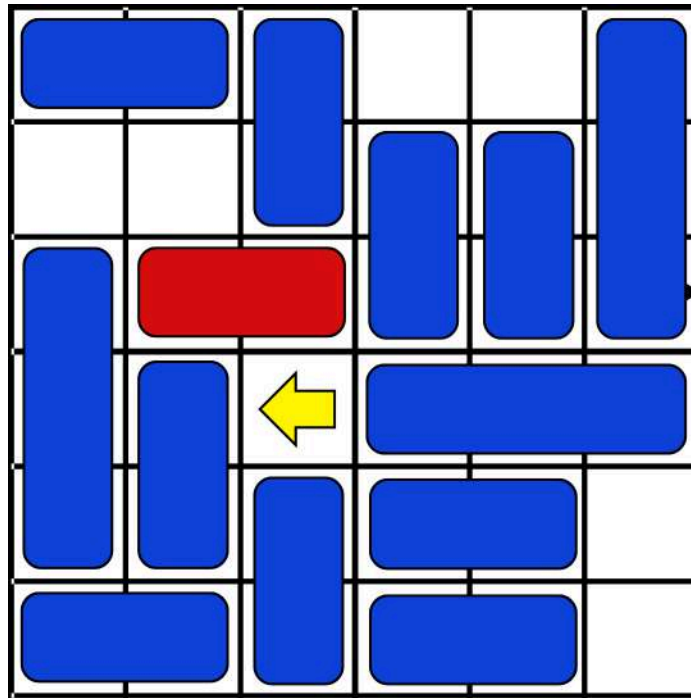


Gambar 2. Awal permainan game Rush Hour

(Sumber:

https://docs.google.com/document/d/1NXyjtIHs2_tWDD37MYtc0VhWtoU2wIH8A95ImttmMXk/edit?tab=t.0)

Pemain dapat menggeser-geser *piece* (termasuk *primary piece*) untuk membentuk jalan lurus antara *primary piece* dan pintu keluar.

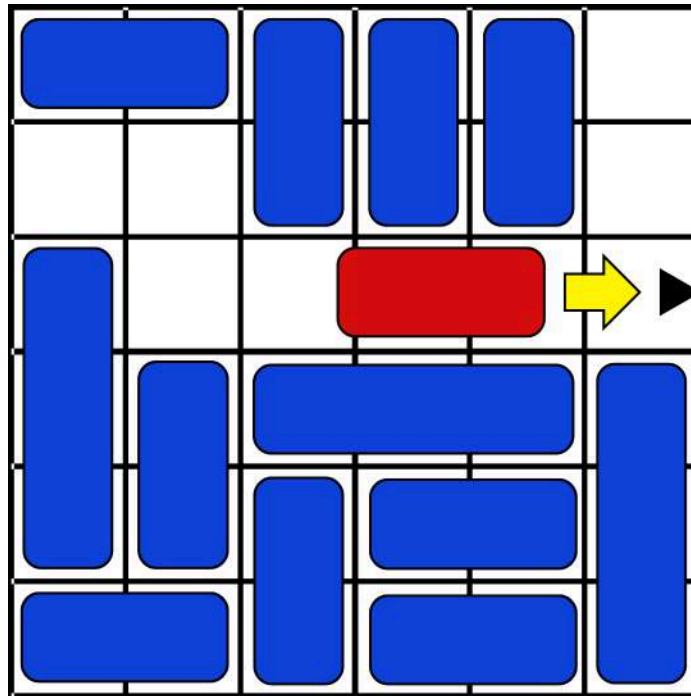


Gambar 3. Gerakan pertama game Rush Hour

(Sumber:

https://docs.google.com/document/d/1NXyjtIHs2_tWDD37MYtc0VhWtoU2wIH8A95ImttmMXk/edit?tab=t.0)

Puzzle berikut dinyatakan telah selesai apabila *primary piece* dapat digeser keluar papan melalui pintu keluar.



Gambar 4. Gerakan seterusnya dan pemain menyelesaikan permainan

(Sumber:

https://docs.google.com/document/d/1NXyjtIHs2_tWDD37MYtc0VhWtoU2wIH8A95ImttmMXk/edit?tab=t.0)

Agar lebih jelas, silahkan amati video cara bermain berikut:

<https://www.youtube.com/watch?v=IWigXwmfcNY&t=40s>

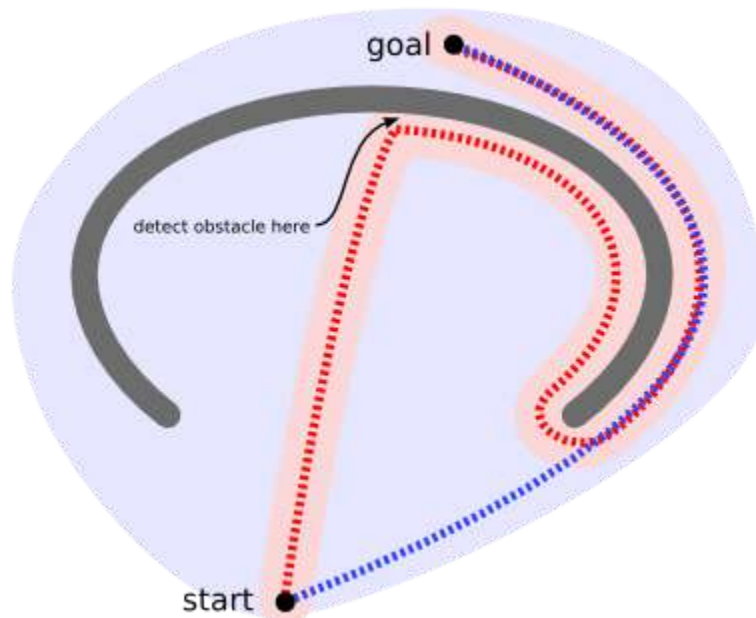
Anda juga dapat melihat gif berikut untuk melihat contoh permainan [Rush Hour Solution](#).

BAB II: TEORI, PENYELESAIAN, DAN ALGORITMA

2.1 Algoritma Pencari Jalur (Pathfinding Algorithm)

Algoritma pencari jalur (*pathfinding algorithm*) adalah teknik pemecahan masalah yang biasa digunakan pada persoalan graf. Algoritma ini akan mencari jalur terpendek dan paling efisien di antara dua titik, dengan mempertimbangkan hambatan, rintangan, dan tantangan lainnya yang terdefinisi pada masalah.

Selama bertahun-tahun, banyak algoritma pencarian jalur telah dikembangkan, dengan kelebihan dan keterbatasannya sendiri. Beberapa algoritma yang paling terkenal adalah algoritma Dijkstra, algoritma *Beam Search*, algoritma *Iterative Deepening A**, algoritma *A**, algoritma *Uniform Cost Search*, dan algoritma *Greedy Best First Search*. Masing-masing dari algoritma tersebut akan dijelaskan pada bagian selanjutnya, yakni analisis algoritma dan penjelasan langkah penyelesaian.



Gambar 5. Contoh Pencarian rute di antara dua titik

(Sumber: <https://theory.stanford.edu/~amitp/GameProgramming/concave1.png>)

2.2 Algoritma Uniform Cost Search (UCS)

2.2.1 Analisis Algoritma Uniform Cost Search (UCS)

Algoritma **Uniform Cost Search** (UCS) sering disebut sebagai salah satu algoritma *uninformed search* (*blind search*) karena tidak memiliki informasi tambahan ketika melakukan pencarian *path* ke titik tujuan (tidak menggunakan heuristik). Algoritma UCS adalah algoritma yang melakukan pencarian rute berdasarkan biaya (*cost*) dari setiap *edges*-nya. Jika seluruh *edges* pada graf tidak memiliki *cost* yang sama, maka algoritma *Breadth-First Search* (BFS) dapat digeneralisasikan menjadi algoritma UCS.

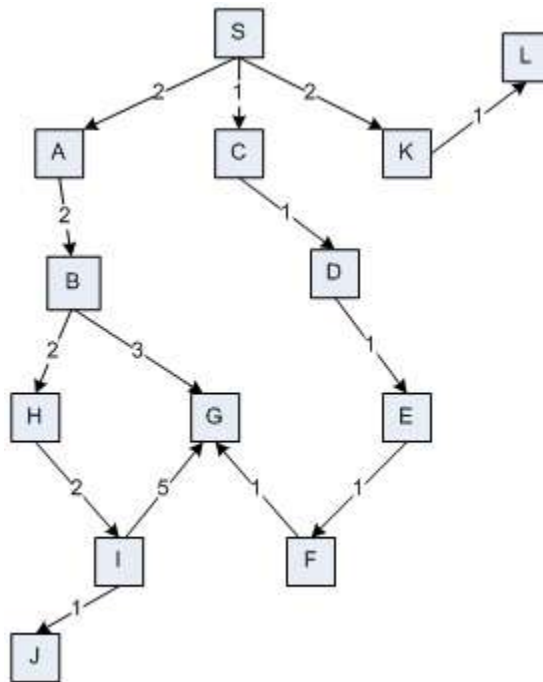
Tetapi, dalam permasalahan permainan Rush Hour, setiap langkah dari *primary piece* memiliki *cost* sebesar satu (baik *single step* maupun *compound step*), menyebabkan hasil dari rute yang dikembangkan mungkin sama dengan hasil yang dikembangkan jika menggunakan algoritma BFS, tetapi secara garis besar hasilnya akan sama (dalam artian urutan *node* yang dibangkitkan dan *path* yang dihasilkan sama).

Karena algoritma UCS tidak menggunakan heuristik, dan juga melakukan pencarian secara *uninformed*, maka algoritma UCS tidak memiliki suatu fungsi heuristik ($h(n)$), yang menyebabkan algoritma UCS dievaluasi menggunakan fungsi $f(n)$ yakni:

$$f(n) = g(n)$$

di mana $f(n)$ merupakan nilai estimasi biaya total yang diperlukan dari n untuk mencapai *goal state* (titik tujuan) nya, dengan $g(n)$ adalah biaya untuk mencapai n .

Selain itu, juga terlampir contoh graf yang merepresentasikan pencarian dua titik menggunakan algoritma UCS.



Gambar 6. Contoh Pencarian rute dari titik S ke titik G
(Sumber: <https://socs.binus.ac.id/files/2017/08/rini-2.jpg>)

2.2.2 Langkah Penyelesaian

Berikut adalah langkah penyelesaian permainan Rush Hour dengan menggunakan algoritma *Uniform Cost Search* (UCS).

1. Pertama, pilih simpul akar (*root*), yakni posisi awal dari *primary piece*, lalu masukkan simpul tersebut ke dalam sebuah *Priority Queue*.
2. Keluarkan elemen terdepan dari *Priority Queue*, dan tandai elemen tersebut dengan suatu penanda untuk menunjukkan bahwa elemen tersebut telah dikunjungi.
3. Jika elemen terdepan yang dikeluarkan adalah simpul tujuan, rekonstruksi jalur dari posisi awal *primary piece* ke posisi tujuan (*exit*) sebagai solusi.
4. Jika elemen terdepan yang dikeluarkan bukan simpul tujuan, eksplorasi semua simpul tetangga dari elemen tersebut. hitung *total cost* dari simpul awal (posisi awal *primary piece*) ke posisi tetangga dari elemen tersebut. Jika simpul tersebut belum dikunjungi, masukkan simpul tersebut beserta *total cost*-nya.

5. Ulangi langkah 3 dan 4 sampai *Priority Queue* kosong atau tujuan ditemukan. *Priority Queue* akan secara otomatis mengurutkan simpul berdasarkan $f(n)$.
6. Jika *Priority Queue* kosong dan tujuan belum ditemukan, maka tidak ada solusi yang mungkin untuk *primary piece* mencapai *exit state*.

2.2.3 Pseudocode

```
function solveUCS(initialBoard: Board, isCompound: boolean) →
Solution
{ Menyelesaikan puzzle Rush Hour menggunakan algoritma Uniform
Cost Search (UCS). }

Deklarasi
    frontier: PriorityQueue           { Antrian prioritas untuk
menyimpan node berdasarkan biaya }
    visited: Set                     { Set untuk melacak state
papan yang sudah dikunjungi }
    statesExamined: integer          { Jumlah state yang telah
diperiksa }
    current: Node                    { Node yang sedang
diproses }
    stateString: string              { Representasi string
dari state papan }
    compoundMoves: List              { Daftar gerakan yang
mungkin }
    newBoard: Board                  { Papan baru setelah
gerakan }
    newStateString: string           { Representasi string
papan baru }
    newCost: integer                 { Biaya untuk node baru }

Algoritma:
    frontier ← createPriorityQueue(comparingByCost) { Buat
priority queue dengan prioritas berdasarkan biaya }
    visited ← createEmptySet()           {
Inisialisasi set visited kosong }
    statesExamined ← 0                   { Mulai
penghitung dari 0 }

    startNode ← createNode(initialBoard, null, null, 0) { Buat
```

```

node awal dengan biaya 0 }
    frontier.add(startNode) {
Tambahkan node awal ke frontier }

    while not frontier.isEmpty() do
        current ← frontier.poll() { Ambil
node dengan biaya terkecil }
        stateString ← current.board.getStateString() {
Dapatkan representasi string dari papan }

        if stateString ∈ visited then { Jika
state sudah dikunjungi, lanjutkan }
            continue
        endif

        visited.add(stateString) { Tandai
state sebagai dikunjungi }
        statesExamined ← statesExamined + 1 { Tambah
counter state yang diperiksa }

        if current.board.isSolved() then { Cek
apakah puzzle terselesaikan }
            return reconstructSolution(current, statesExamined)
        { Kembalikan solusi }
        endif

        compoundMoves ← generateCompoundMoves(current.board,
isCompound) { Dapatkan semua gerakan yang mungkin }

        for each move in compoundMoves do
            newBoard ← makeCompoundMove(current.board, move) {
Buat papan baru setelah gerakan }
            newStateString ← newBoard.getStateString() {
Dapatkan representasi string papan baru }

            if newStateString ∉ visited then {
Jika state baru belum dikunjungi }
                newCost ← current.cost + 1 {
Setiap gerakan memiliki biaya 1 }
                newNode ← createNode(newBoard, move, current,
newCost) { Buat node baru }
                frontier.add(newNode) {
Tambahkan node baru ke frontier }
            endif
        endfor
    endwhile
endfunction

```



```
        endfor
    endwhile

    return null { Tidak ada solusi yang ditemukan }
```

2.3 Algoritma Greedy Best First Search (GBFS)

2.3.1 Analisis Algoritma Greedy Best First Search (GBFS)

Greedy Best-First Search (GBFS) merupakan salah satu algoritma dalam kategori *Informed Search*, yaitu algoritma pencarian yang memanfaatkan informasi tambahan berupa fungsi heuristik untuk mengestimasi seberapa dekat suatu *state* (simpul) terhadap tujuan. GBFS bekerja dengan selalu memilih simpul yang memiliki nilai heuristik terkecil, yakni simpul yang tampak paling dekat ke goal, tanpa mempertimbangkan biaya jalur yang telah ditempuh dari simpul awal.

Pendekatan ini menjadikan GBFS sangat efisien dalam beberapa kasus karena dapat dengan cepat mengarahkan pencarian ke arah yang menjanjikan. Namun, kelemahan utamanya adalah tidak adanya jaminan optimalitas dari solusi yang ditemukan, karena keputusan pencarian semata-mata didasarkan pada estimasi heuristik. Keberhasilan dan efisiensi algoritma ini sangat tergantung pada akurasi dan kualitas fungsi heuristik yang digunakan dalam konteks masalah yang dihadapi. Selain itu, GBFS rentan terhadap siklus atau eksplorasi berulang apabila tidak disertai strategi penandaan node yang telah dikunjungi (*visited set*). Dalam kasus seperti ini, algoritma dapat terjebak dalam loop atau gagal mencapai tujuan meskipun sebenarnya terdapat jalur yang valid.

Secara formal, setiap simpul n dalam GBFS dievaluasi menggunakan fungsi:

$$f(n) = h(n)$$

di mana $h(n)$ adalah nilai heuristik yang mengestimasi biaya dari simpul n menuju goal. Simpul dengan nilai $f(n)$ terkecil akan dipilih sebagai simpul yang diekspansi berikutnya dalam proses pencarian.

2.3.2 Langkah Penyelesaian

Berikut adalah langkah penyelesaian permainan Rush Hour dengan menggunakan algoritma *Greedy Best First Search* (GBFS)

1. Pertama, pilih simpul akar (*root*), yakni posisi awal dari *primary piece*, lalu masukkan simpul tersebut ke dalam sebuah *Priority Queue*.
2. Keluarkan elemen terdepan dari *Priority Queue*, dan tandai elemen tersebut dengan suatu penanda untuk menunjukkan bahwa elemen tersebut telah dikunjungi.
3. Jika elemen terdepan yang dikeluarkan adalah simpul tujuan, rekonstruksi jalur dari posisi awal *primary piece* ke posisi tujuan (*exit*) sebagai solusi.
4. Jika elemen terdepan yang dikeluarkan bukan simpul tujuan, eksplorasi semua simpul tetangga dari elemen tersebut. hitung *total cost* dari simpul awal (posisi awal *primary piece*) ke posisi tetangga dari elemen tersebut. Jika simpul tersebut belum dikunjungi, masukkan simpul tersebut beserta *total cost*-nya.
5. Ulangi langkah 3 dan 4 sampai *Priority Queue* kosong atau tujuan ditemukan. *Priority Queue* akan secara otomatis mengurutkan simpul berdasarkan $f(n)$. Dalam kasus GBFS, $f(n) = h(n)$, yakni estimasi heuristik dari node tersebut.
6. Jika *Priority Queue* kosong dan tujuan belum ditemukan, maka tidak ada solusi yang mungkin untuk *primary piece* mencapai *exit state*.

2.3.3 Pseudocode

```
function solveGreedy(initialBoard: Board, heuristic: string,
isCompound: boolean) → Solution
{ Menyelesaikan puzzle Rush Hour menggunakan algoritma Greedy
Best First Search. }
Deklarasi
    frontier: PriorityQueue          { Antrian prioritas untuk
menyimpan node berdasarkan nilai heuristik }
    visited: Set                    { Set untuk melacak state
papan yang sudah dikunjungi }
    statesExamined: integer         { Jumlah state yang telah
diperiksa }
    current: Node                   { Node yang sedang
diproses }
    stateString: string              { Representasi string
dari state papan }
    compoundMoves: List              { Daftar gerakan yang
mungkin }
    newBoard: Board                  { Papan baru setelah
gerakan }
```

```

    newStateString: string          { Representasi string
papan baru }
    h: integer                      { Nilai heuristik }

Algoritma:
    frontier ← createPriorityQueue(comparingByHeuristic) {
    Buat priority queue dengan prioritas berdasarkan heuristik }
    visited ← createEmptySet() {
    Inisialisasi set visited kosong }
    statesExamined ← 0 {
    Mulai penghitung dari 0 }

    h ← calculateHeuristic(initialBoard, heuristic) {
    Hitung nilai heuristik untuk papan awal }
    startNode ← createNode(initialBoard, null, null, 0, h, h)
    { Buat node awal }
    frontier.add(startNode) {
    Tambahkan node awal ke frontier }

    while not frontier.isEmpty() do
        current ← frontier.poll() { Ambil
node dengan nilai heuristik terkecil }
        stateString ← current.board.getStateString() {
    Dapatkan representasi string dari papan }

        if stateString ∈ visited then { Jika
state sudah dikunjungi, lanjutkan }
            continue
        endif

        visited.add(stateString) { Tandai
state sebagai dikunjungi }
        statesExamined ← statesExamined + 1 { Tambah
counter state yang diperiksa }

        if current.board.isSolved() then { Cek
apakah puzzle terselesaikan }
            return reconstructSolution(current, statesExamined)
        { Kembalikan solusi }
        endif

        compoundMoves ← generateCompoundMoves(current.board,
isCompound) { Dapatkan semua gerakan yang mungkin }

```

```

        for each move in compoundMoves do
            newBoard ← makeCompoundMove(current.board, move) {
Buat papan baru setelah gerakan }
            newStateString ← newBoard.getStateString() {
Dapatkan representasi string papan baru }

            if newStateString ∉ visited then {
Jika state baru belum dikunjungi }
                newH ← calculateHeuristic(newBoard, heuristic)
{ Hitung nilai heuristik untuk papan baru }
                newNode ← createNode(newBoard, move, current,
current.cost + 1, newH, newH) { Buat node baru }
                frontier.add(newNode) {
Tambahkan node baru ke frontier }
            endif
        endfor
    endwhile

    return null { Tidak ada solusi yang ditemukan }

```

2.4 Algoritma A* (A-Star)

2.4.1 Analisis Algoritma A* (A-Star)

A* (A Star) adalah salah satu algoritma dalam kategori *Informed Search* yang menggabungkan dua komponen penting dalam proses pencarian, yaitu biaya aktual dari titik awal dan estimasi jarak ke tujuan. Informasi ini dihitung melalui fungsi evaluasi

$$f(n) = g(n) + h(n)$$

di mana $g(n)$ merupakan total biaya dari simpul awal ke simpul saat ini, dan $h(n)$ adalah fungsi heuristik yang memperkirakan jarak dari simpul saat ini ke simpul tujuan. Dengan menggunakan kombinasi dua komponen ini, A* mampu membuat keputusan yang seimbang antara eksplorasi dan efisiensi.

Kelebihan utama dari A* dibanding algoritma pencarian lain seperti GBFS atau Uniform Cost Search terletak pada kemampuannya untuk menjamin optimalitas dan kelengkapan, selama fungsi heuristik yang digunakan bersifat admissible (tidak melebihi biaya sebenarnya ke goal) dan konsisten. Hal ini menjadikan A* sebagai algoritma yang sangat efektif dalam menyelesaikan masalah pencarian jalur atau perencanaan, khususnya ketika ruang pencarian besar namun masih dapat dibantu dengan informasi heuristik.

Secara formal, setiap simpul n akan diberi nilai evaluasi $f(n) = g(n) + h(n)$. Simpul yang memiliki nilai $f(n)$ terkecil akan diprioritaskan untuk diekspansi terlebih dahulu. Mekanisme ini memungkinkan A* untuk menemukan solusi dengan biaya minimum secara efisien, sambil tetap menghindari eksplorasi yang tidak perlu terhadap simpul-simpul yang tidak relevan.

2.4.2 Langkah Penyelesaian

Berikut adalah langkah penyelesaian permainan Rush Hour dengan menggunakan algoritma A* (A-Star)

1. Pertama, pilih simpul akar (*root*), yakni posisi awal dari *primary piece*, lalu masukkan simpul tersebut ke dalam sebuah *Priority Queue*.
2. Keluarkan elemen terdepan dari *Priority Queue*, dan tandai elemen tersebut dengan suatu penanda untuk menunjukkan bahwa elemen tersebut telah dikunjungi.
3. Jika elemen terdepan yang dikeluarkan adalah simpul tujuan, rekonstruksi jalur dari posisi awal *primary piece* ke posisi tujuan (*exit*) sebagai solusi.
4. Jika elemen terdepan yang dikeluarkan bukan simpul tujuan, eksplorasi semua simpul tetangga dari elemen tersebut. hitung *total cost* dari simpul awal (posisi awal *primary piece*) ke posisi tetangga dari elemen tersebut. Jika simpul tersebut belum dikunjungi, masukkan simpul tersebut beserta *total cost*-nya.
5. Ulangi langkah 3 dan 4 sampai *Priority Queue* kosong atau tujuan ditemukan. *Priority Queue* akan secara otomatis mengurutkan simpul berdasarkan $f(n)$. Dalam kasus GBFS, $f(n) = g(n) + h(n)$, yakni jarak node saat ini ditambah dengan nilai estimasi heuristik dari node tersebut.
6. Jika *Priority Queue* kosong dan tujuan belum ditemukan, maka tidak ada solusi yang mungkin untuk *primary piece* mencapai *exit state*.

2.4.3 Pseudocode

```
function solveAStar(initialBoard: Board, heuristic: string,
isCompound: boolean) → Solution
{ Menyelesaikan puzzle Rush Hour menggunakan algoritma A*
Search. }
Deklarasi
    frontier: PriorityQueue          { Antrian prioritas untuk
menyimpan node berdasarkan nilai f }
    visited: Set                    { Set untuk melacak state
```

```

papan yang sudah dikunjungi }
    nodeMap: Map                { Map untuk menyimpan
node terbaik untuk setiap state }
    statesExamined: integer      { Jumlah state yang telah
diperiksa }
    current: Node                { Node yang sedang
diproses }
    stateString: string          { Representasi string
dari state papan }
    compoundMoves: List          { Daftar gerakan yang
mungkin }
    newBoard: Board              { Papan baru setelah
gerakan }
    newStateString: string       { Representasi string
papan baru }
    newG: integer                { Biaya dari awal ke node
baru }
    newH: integer                { Nilai heuristik untuk
node baru }
    newF: integer                {  $f(n) = g(n) + h(n)$  }

Algoritma:
    frontier ← createPriorityQueue(comparingByF)    { Buat
priority queue dengan prioritas berdasarkan f }
    visited ← createEmptySet()                      {
Inisialisasi set visited kosong }
    nodeMap ← createEmptyMap()                      {
Inisialisasi map untuk node terbaik }
    statesExamined ← 0                             { Mulai
penghitung dari 0 }

    h ← calculateHeuristic(initialBoard, heuristic) { Hitung
nilai heuristik untuk papan awal }
    startNode ← createNode(initialBoard, null, null, 0, h, h)
{ Buat node awal dengan g=0, h=h, f=h }
    frontier.add(startNode)                          {
Tambahkan node awal ke frontier }
    nodeMap.put(initialBoard.getStateString(), startNode)
{ Simpan node awal dalam map }

    while not frontier.isEmpty() do
        current ← frontier.poll()                    { Ambil
node dengan nilai f terkecil }
        stateString ← current.board.getStateString() {

```

```

Dapatkan representasi string dari papan }

    if stateString ∈ visited then                { Jika
state sudah dikunjungi, lanjutkan }
        continue
    endif

    visited.add(stateString)                      { Tandai
state sebagai dikunjungi }
    statesExamined ← statesExamined + 1          { Tambah
counter state yang diperiksa }

    if current.board.isSolved() then              { Cek
apakah puzzle terselesaikan }
        return reconstructSolution(current, statesExamined)
    { Kembalikan solusi }
    endif

    compoundMoves ← generateCompoundMoves(current.board,
isCompound) { Dapatkan semua gerakan yang mungkin }

    for each move in compoundMoves do
        newBoard ← makeCompoundMove(current.board, move) {
Buat papan baru setelah gerakan }
        newStateString ← newBoard.getStateString()        {
Dapatkan representasi string papan baru }

        if newStateString ∉ visited then                  {
Jika state baru belum dikunjungi }
            newG ← current.cost + 1                      {
Biaya g(n) = biaya dari awal + 1 }
            newH ← calculateHeuristic(newBoard, heuristic)
{ Hitung nilai heuristik }
            newF ← newG + newH                            {
f(n) = g(n) + h(n) }

            if newStateString ∉ nodeMap OR
nodeMap.get(newStateString).f > newF then
                { Jika state baru belum ada dalam nodeMap
atau ditemukan jalur lebih baik }
                newNode ← createNode(newBoard, move,
current, newG, newH, newF) { Buat node baru }
                frontier.add(newNode)                    {
Tambahkan node baru ke frontier }

```

```

                                nodeMap.put(newStateString, newNode)      {
Simpan/update node dalam map }
                                endif
                                endif
                                endfor
                                endwhile

                                return null { Tidak ada solusi yang ditemukan }

```

2.5 Algoritma Dijkstra

2.5.1 Analisis Algoritma Dijkstra

Dijkstra adalah salah satu algoritma pencarian jalur terpendek yang termasuk dalam kategori *Uninformed Search*, karena tidak menggunakan informasi tambahan seperti heuristik. Dijkstra bertujuan untuk mencari jalur dengan total biaya minimum dari simpul awal ke semua simpul lain, atau ke satu simpul tujuan tertentu, dengan cara mengevaluasi simpul berdasarkan biaya riil yang telah ditempuh sejauh ini, tanpa memperkirakan biaya ke depan seperti dalam A* atau GBFS.

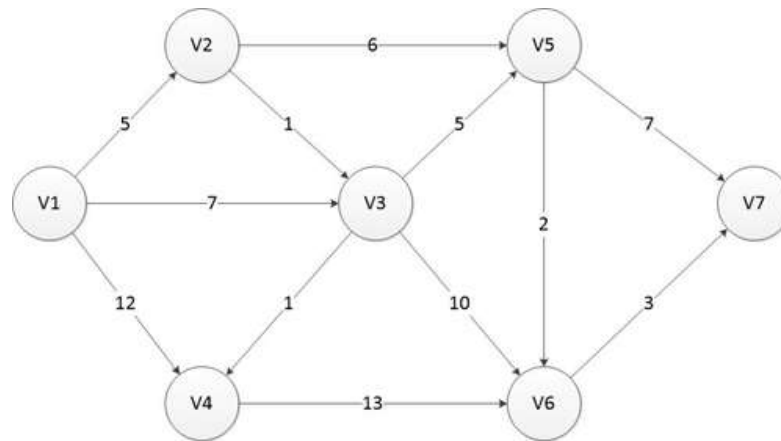
Algoritma ini menjamin solusi yang optimal dan lengkap, karena selalu memilih simpul dengan biaya terkecil dari simpul awal $g(n)$ sebagai simpul berikutnya untuk diekspansi. Dijkstra bekerja secara iteratif dengan menyimpan nilai biaya terkecil ke setiap simpul dan memperbarui jalur terbaik jika ditemukan jalur alternatif yang lebih murah. Algoritma ini sangat cocok digunakan pada graf berbobot non-negatif, baik untuk graf terhubung secara penuh maupun sebagian.

Dalam kasus penyelesaian puzzle *Rush Hour* dengan algoritma Dijkstra, setiap gerakan diasumsikan memiliki cost sebesar 1, sama seperti algoritma UCS. Akan tetapi, ada sedikit perbedaan dengan algoritma UCS, yakni pada algoritma UCS, suatu simpul yang sudah dikunjungi tidak akan dikunjungi lagi, sementara pada algoritma Dijkstra, suatu simpul masih mungkin untuk dikunjungi lagi apabila didapatkan suatu *distance* dengan jarak yang lebih optimal. Artinya, suatu simpul bisa dimasukkan kembali ke dalam *priority queue* jika nilai jarak terbaiknya diperbarui. Mekanisme ini dikenal sebagai relaksasi (*relaxation*) dan umum dijumpai dalam implementasi Dijkstra.

Secara formal, setiap simpul n dalam Dijkstra memiliki nilai evaluasi $f(n) = g(n)$, di mana $g(n)$ adalah total biaya dari simpul awal ke n . Simpul dengan nilai $g(n)$ terkecil akan dipilih sebagai pusat ekspansi selanjutnya. Karena tidak menggunakan heuristik, Dijkstra tidak bersifat directional seperti A*. Dengan kata lain, Dijkstra "menyebar merata" ke segala arah secara sistematis berdasarkan total biaya minimum, bukan

berdasarkan posisi relatif terhadap goal. Dengan demikian, algoritma Dijkstra menjamin menemukan jalur terpendek melalui eksplorasi menyeluruh yang efisien terhadap simpul-simpul dengan bobot minimum terlebih dahulu.

Berikut adalah visualisasi sederhana bagaimana suatu graf diproses pada algoritma Dijkstra



Iteration	Unvisited (Q)	Visited (S)	Current	Node : Min = (dis[node] prev[node] iteration)						
				V1	V2	V3	V4	V5	V6	V7
	Initialization {V1,V2,V3,V4,V5,V6,V7}	{-}		(0,-0)	(∞,-0)	(∞,-0)	(∞,-0)	(∞,-0)	(∞,-0)	(∞,-0)
1	{V2,V3,V4,V5,V6,V7}	{V1}	V1	(3,V1)1	(7,V1)1	(12,V1)1	(∞,V1)1	(∞,V1)1	(∞,V1)1	(∞,V1)1
2	{V3,V4,V5,V6,V7}	{V1,V2}	V2		(6,V2)2	(12,V1)1	(11,V2)2	(∞,V2)2	(∞,V2)2	(∞,V2)2
3	{V4,V5,V6,V7}	{V1,V2,V3}	V3			(7,V3)3	(16,V3)3	(∞,V3)3	(∞,V3)3	(∞,V3)3
4	{V5,V6,V7}	{V1,V2,V3,V4}	V4				(11,V3)3	(16,V3)3	(∞,V3)3	(∞,V3)3
5	{V6,V7}	{V1,V2,V3,V4,V5}	V5					(13,V5)5	(18,V5)5	(18,V5)5
6	{V7}	{V1,V2,V3,V4,V5,V6}	V6						(16,V6)6	(16,V6)6

Gambar 7. Contoh Pencarian rute dengan menggunakan algoritma Dijkstra
(Sumber : <https://mti.binus.ac.id/2017/11/28/algoritma-dijkstra/>)

2.5.2 Langkah Penyelesaian

Berikut adalah langkah penyelesaian permainan Rush Hour dengan menggunakan algoritma *Dijkstra*

1. Pertama, pilih simpul akar (*root*), yakni posisi awal dari *primary piece*, lalu masukkan simpul tersebut ke dalam sebuah *Priority Queue* dengan *cost* awal bernilai 0.
2. Keluarkan elemen terdepan dari *Priority Queue*, dan tandai elemen tersebut dengan suatu penanda untuk menunjukkan bahwa elemen tersebut telah dikunjungi.

3. Jika elemen terdepan yang dikeluarkan adalah simpul tujuan, lakukan rekonstruksi jalur dari posisi awal *primary piece* ke posisi tujuan (*exit*) sebagai solusi.
4. Jika elemen terdepan bukan simpul tujuan, eksplorasi semua simpul tetangga yang mungkin dari posisi tersebut (misalnya pergeseran kendaraan lain yang sah). Hitung total biaya aktual (misalnya jumlah langkah atau perpindahan) dari simpul awal ke simpul tetangga. Jika tetangga tersebut belum dikunjungi atau ditemukan jalur dengan *cost* lebih kecil dari sebelumnya, simpan jalur tersebut ke *Priority Queue* dengan nilai *cost*-nya.
5. Ulangi langkah 3 dan 4 sampai *Priority Queue* kosong atau tujuan ditemukan. Dalam kasus algoritma Dijkstra, simpul dengan nilai biaya $g(n)$ terkecil akan selalu diproses terlebih dahulu.
6. Jika *Priority Queue* kosong dan tujuan belum ditemukan, maka tidak ada solusi yang mungkin bagi *primary piece* untuk mencapai *exit state*.

2.5.3 Pseudocode

```
function solveDijkstra(initialBoard: Board, isCompound:
boolean) → Solution
Deklarasi:
    frontier: PriorityQueue          { Antrian prioritas
berdasarkan cost }
    visited: Set                    { State yang sudah dikunjungi
}
    costSoFar: Map                  { Menyimpan cost terbaik
untuk setiap state }
    current: Node                   { Node saat ini }
    compoundMoves: List             { Daftar gerakan compound }

Algoritma:
    frontier ← PriorityQueue(Comparator.cost)
    visited ← new Set()
    costSoFar ← new Map()
    statesExamined ← 0

    startNode ← Node(initialBoard, null, null, 0)
    frontier.add(startNode)
    costSoFar[initialBoard] ← 0
```

```

while frontier not empty:
    current ← frontier.poll()
    stateString ← current.board.getStateString()

    if stateString ∈ visited: continue
    visited.add(stateString)
    statesExamined++

    if current.board.isSolved():
        return reconstructSolution(current, statesExamined)

    compoundMoves ← generateCompoundMoves(current.board,
isCompound)

    for each move in compoundMoves:
        newBoard ← makeCompoundMove(current.board, move)
        newStateString ← newBoard.getStateString()
        newCost ← current.cost + 1

        if newCost < costSoFar.get(newStateString, ∞):
            costSoFar[newStateString] ← newCost
            newNode ← Node(newBoard, move, current,
newCost)
            frontier.add(newNode)

return null

```

2.6 Algoritma Beam-Search

2.6.1 Analisis Algoritma *Beam-Search*

Beam Search adalah salah satu variasi dari algoritma pencarian *Informed Search* yang menggabungkan prinsip dasar dari *Greedy Best-First Search* (GBFS) namun dengan batasan eksplorasi yang ketat untuk menghemat sumber daya komputasi. Berbeda dengan A* atau GBFS yang dapat menyimpan dan mengevaluasi semua *node* terbuka, *Beam Search* hanya menyimpan sejumlah *node* terbaik terbatas (beam width) pada setiap langkah pencarian.

Algoritma ini bekerja dengan cara memilih simpul awal, kemudian pada setiap level eksplorasi, hanya k simpul dengan nilai heuristik terbaik (yakni yang paling menjanjikan menuju *goal*) yang akan dipertahankan dan diekspansi lebih lanjut. Nilai k

ini disebut sebagai *beam width*, dan menjadi parameter utama yang mengontrol seberapa luas ruang pencarian yang dijelajahi pada tiap level. Dalam kasus penyelesaian permainan Rush Hour dengan *Beam Search*, digunakan nilai k sebesar 50, artinya hanya boleh terdapat 50 *node* yang dipertahankan dan diekspansi lebih lanjut.

Karena *Beam Search* sangat bergantung pada nilai *beam width*, ada *trade-off* (pertukaran) yang signifikan antara kualitas solusi dan efisiensi pencarian. *Beam width* kecil dapat membuat algoritma lebih cepat dan hemat memori, tetapi berisiko melewatkan solusi terbaik. *Beam Search* tidak menjamin solusi optimal, dan bahkan bisa gagal menemukan solusi jika jalur menuju goal tidak berada dalam jalur yang "tampak menjanjikan" dari perspektif heuristik awal. Analisis mengenai *trade-off* tersebut akan dibahas lebih lanjut pada bab selanjutnya.

Secara formal, *Beam Search* juga menggunakan fungsi heuristik $h(n)$ untuk mengevaluasi simpul, sama seperti GBFS. Namun, pada setiap langkah, hanya k simpul dengan nilai heuristik terendah yang dipilih dari semua kandidat hasil ekspansi dan dilanjutkan ke iterasi berikutnya. Proses ini diulang sampai *goal* ditemukan atau tidak ada lagi *node* untuk diekspansi.

2.6.2 Langkah Penyelesaian

Berikut adalah langkah penyelesaian permainan Rush Hour dengan menggunakan algoritma *Beam-Search*

1. Pertama, pilih simpul akar (*root*), yakni posisi awal dari **primary piece**, lalu masukkan simpul tersebut ke dalam sebuah **list frontier** yang menyimpan semua simpul yang akan dievaluasi, beserta nilai heuristik awal yang dihitung dari posisi tersebut. Inisialisasi juga nilai **beam width** sebagai batas jumlah maksimum simpul yang boleh diproses pada setiap level pencarian.
2. Keluarkan seluruh elemen dalam *frontier* untuk iterasi saat ini, lalu periksa satu per satu. Jika suatu simpul belum pernah dikunjungi, tandai simpul tersebut sebagai telah dikunjungi, dan periksa apakah simpul tersebut merupakan simpul tujuan. Jika ya, lakukan rekonstruksi jalur dari posisi awal *primary piece* ke posisi tujuan (*exit*) sebagai solusi.
3. Jika simpul yang dievaluasi bukan tujuan, lakukan eksplorasi terhadap semua simpul tetangga yang dapat dicapai melalui pergerakan sah (termasuk pergerakan *compound* jika diperbolehkan). Untuk setiap tetangga yang belum dikunjungi, hitung nilai heuristiknya dan simpan sebagai simpul baru dalam daftar `nextLevel`.
4. Setelah semua simpul dalam *frontier* dievaluasi, seluruh simpul yang dihasilkan pada level tersebut disortir berdasarkan nilai heuristik

$h(n)h(n)h(n)$ dari yang paling kecil. Hanya **beam width** simpul terbaik yang akan dipertahankan dan dimasukkan kembali ke frontier untuk iterasi berikutnya.

5. Ulangi langkah 2 hingga 5 sampai simpul tujuan ditemukan atau frontier kosong. Jika frontier kosong dan tujuan belum ditemukan, maka tidak ada solusi yang mungkin bagi *primary piece* untuk mencapai *exit state* dalam batasan *beam width* yang digunakan.

2.6.3 Pseudocode

```
function solveBeam(initialBoard: Board, heuristic: string,
isCompound: boolean) → Solution
Deklarasi:
    beamWidth: integer ← 50          { Ukuran beam }
    frontier: List                    { Daftar node saat ini }
    visited: Set                      { State yang sudah dikunjungi }
    nextLevel: List                   { Node untuk level berikutnya }

Algoritma:
    frontier ← [Node(initialBoard, null, null, 0, h, h)]
    visited ← new Set()
    statesExamined ← 0

    while frontier not empty:
        nextLevel ← []

        for each current in frontier:
            stateString ← current.board.getStateString()

            if stateString ∈ visited: continue
            visited.add(stateString)
            statesExamined++

            if current.board.isSolved():
                return reconstructSolution(current,
statesExamined)

        compoundMoves ←
generateCompoundMoves(current.board, isCompound)
```

```

        for each move in compoundMoves:
            newBoard ← makeCompoundMove(current.board,
move)
            newStateString ← newBoard.getStateString()

            if newStateString ∉ visited:
                newH ← calculateHeuristic(newBoard,
heuristic)
                newNode ← Node(newBoard, move, current,
current.cost+1, newH, newH)
                nextLevel.add(newNode)

        { Urutkan berdasarkan heuristik dan pilih terbaik }
        sort(nextLevel by h ascending)
        frontier ← nextLevel[0:min(beamWidth,
length(nextLevel))]

    return null

```

2.7 Algoritma A* Iterative Deepening Search

2.7.1 Analisis Algoritma A* Iterative Deepening Search

A* Iterative Deepening (IDA*) adalah salah satu variasi dari algoritma pencarian *Informed Search* yang menggabungkan keunggulan dari algoritma A* dan *Depth-First Iterative Deepening* (DFID). Berbeda dengan A* yang menggunakan struktur *priority queue* dan menyimpan semua *node* terbuka dalam memori, IDA* menghindari konsumsi memori besar dengan hanya menyimpan jejak pencarian yang sedang berlangsung, seperti pada *Depth-First Search* (DFS), sambil tetap mempertahankan evaluasi fungsi $f(n)$ milik A*.

IDA* bekerja dengan cara melakukan pencarian *depth-first* yang dibatasi oleh sebuah ambang batas nilai fungsi evaluasi $f(n)=g(n)+h(n)$. Pada iterasi pertama, ambang batas ini diatur berdasarkan nilai f dari simpul awal. Jika pencarian tidak menemukan solusi dalam batas tersebut, maka algoritma akan mengulang pencarian dengan meningkatkan ambang ke nilai terkecil $f(n)$ yang melebihi batas sebelumnya. Proses ini diulang hingga ditemukan solusi atau seluruh ruang pencarian habis dieksplorasi. Dalam kasus penyelesaian permainan Rush Hour dengan IDA*, setiap langkah gerakan diasumsikan memiliki *cost* sebesar 1, dan fungsi heuristik $h(n)$ digunakan untuk mengarahkan pencarian ke posisi tujuan secara efisien.

Karena IDA* menggunakan pendekatan iteratif dan menghindari penyimpanan node secara luas, algoritma ini sangat hemat memori, bahkan dalam ruang pencarian yang besar. Namun, pendekatan ini memiliki *trade-off* yaitu potensi pengulangan pencarian pada simpul-simpul yang sama pada iterasi berbeda. Dengan demikian, meskipun IDA* menjamin solusi optimal selama heuristiknya bersifat *admissible*, efisiensinya bisa lebih rendah dibandingkan A* pada beberapa kasus. Analisis mengenai pengaruh kedalaman dan kualitas heuristik terhadap efisiensi IDA* akan dibahas lebih lanjut pada bab selanjutnya.

Secara formal, IDA* mengevaluasi setiap simpul menggunakan fungsi $f(n)=g(n)+h(n)$, dan hanya akan menelusuri jalur dengan nilai $f(n)$ yang tidak melebihi ambang batas saat ini. Setelah satu iterasi selesai, ambang batas diperbarui berdasarkan simpul yang memiliki nilai $f(n)$ terkecil yang melebihi ambang sebelumnya. Proses ini diulang secara bertahap sampai ditemukan solusi atau seluruh ruang pencarian telah dijelajahi tanpa hasil.

2.7.2 Langkah Penyelesaian

Berikut adalah langkah penyelesaian permainan Rush Hour dengan menggunakan algoritma IDA*:

1. Pertama, pilih simpul akar (*root*), yakni posisi awal dari *primary piece*, lalu hitung nilai awal fungsi evaluasi $f(n)=g(n)+h(n)$, di mana $g(n) = 0$ dan $h(n)$ adalah nilai heuristik dari posisi tersebut ke posisi tujuan. Nilai $f(n)$ ini digunakan sebagai *threshold* awal untuk iterasi pertama
2. Lakukan pencarian secara *depth-first* dengan membatasi eksplorasi hanya pada simpul-simpul yang memiliki nilai $f(n) < THRESHOLD$. Jika nilai $f(n)$ dari suatu simpul melebihi *THRESHOLD* saat ini, simpul tersebut tidak diekspansi dan nilai $f(n)$ -nya dicatat sebagai kandidat *THRESHOLD* berikutnya.
3. Jika simpul yang sedang diekspansi adalah simpul tujuan, maka proses rekonstruksi jalur dilakukan dari posisi awal *primary piece* ke posisi tujuan (*exit*) sebagai solusi.
4. Jika simpul yang diekspansi bukan simpul tujuan, eksplorasi semua simpul tetangga dari simpul tersebut. Untuk setiap tetangga, hitung nilai $f(n)=g(n)+h(n)$. Jika nilai ini masih berada dalam batas *threshold* saat ini, lanjutkan pencarian secara rekursif ke simpul tersebut.
5. Jika seluruh pencarian dalam batas *threshold* selesai dan tujuan belum ditemukan, naikkan nilai *THRESHOLD* ke nilai minimum dari $f(n)$ yang sebelumnya melebihi batas, dan ulangi proses pencarian dengan *threshold* baru tersebut.

6. Ulangi proses ini sampai simpul tujuan ditemukan, atau tidak ada lagi *threshold* yang dapat digunakan. Jika pencarian berakhir tanpa menemukan solusi, maka tidak ada jalur yang memungkinkan bagi *primary piece* untuk mencapai *exit state* dalam ruang pencarian yang diberikan.

2.7.3 Pseudocode

```
function solveIDAStar(initialBoard: Board, heuristic: string,
isCompound: boolean) → Solution
Deklarasi:
    threshold: integer
    result: Result

Algoritma:
    h ← calculateHeuristic(initialBoard, heuristic)
    root ← Node(initialBoard, null, null, 0, h, h)
    threshold ← root.f

    loop:
        visited ← new Set()
        result ← dfsIDA(root, heuristic, isCompound, threshold,
visited)

        if result.found:
            return reconstructSolution(result.node,
statesExamined)
        if result.nextThreshold = ∞:
            return null

        threshold ← result.nextThreshold

function dfsIDA(current: Node, heuristic: string, isCompound:
boolean, threshold: int, visited: Set) → Result
Deklarasi:
    minThreshold: integer ← ∞

Algoritma:
    stateString ← current.board.getStateString()
    if stateString ∈ visited: return Result(false, null, ∞)
    visited.add(stateString)
```



```

statesExamined++

f ← current.cost + current.h
if f > threshold: return Result(false, null, f)
if current.board.isSolved(): return Result(true, current,
threshold)

for each move in generateCompoundMoves(current.board,
isCompound):
    newBoard ← makeCompoundMove(current.board, move)
    newH ← calculateHeuristic(newBoard, heuristic)
    child ← Node(newBoard, move, current, current.cost+1,
newH, current.cost+1+newH)

    result ← dfsIDA(child, heuristic, isCompound,
threshold, visited)
    if result.found: return result
    minThreshold ← min(minThreshold, result.nextThreshold)

visited.remove(stateString)
return Result(false, null, minThreshold)

```

2.8 Strategi Heuristik

2.8.1 Manhattan Distance

Jarak Manhattan antara dua titik adalah jumlah dari panjang ruas garis kedua titik tersebut terhadap tiap sumbu dalam koordinat Kartesius. Dalam kasus ini, manhattan distance digunakan sebagai “pedoman” dalam beberapa algoritma yang memanfaatkan strategi heuristik. Nilai dari $h(n)$ untuk suatu *state* adalah jumlah langkah minimum secara horizontal dan vertikal dari posisi *primary piece* saat ini menuju *exit goal*, tanpa mempertimbangkan hambatan. Dalam konteks permainan Rush Hour, ini berarti menghitung selisih baris dan kolom (tergantung orientasi mobil) antara posisi bagian terdepan dari *primary piece* dengan sel tujuan. Heuristik ini dapat dipastikan merupakan strategi yang *Admissible*, yakni nilai $h(n)$ pasti kurang dari atau sama dengan nilai $h^*(n)$.

2.8.2 Direct Distance Count

Direct Distance estimasi berapa langkah minimal yang diperlukan oleh suatu *Primary Piece* untuk mencapai titik exit, dengan asumsi bahwa *piece* tersebut dapat “menembus” seluruh halangan yang ada. Estimasi ini dapat dipastikan merupakan strategi yang *Admissible*, yakni nilai $h(n)$ pasti kurang dari atau sama dengan nilai $h^*(n)$.

2.8.3 Blocking Count

Blocking Count adalah heuristik yang menghitung jumlah kendaraan lain yang secara langsung menghalangi jalan *primary piece* menuju exit. Dalam konteks Rush Hour, ini berarti menghitung berapa banyak kendaraan yang menempati sel di antara *primary piece* dan goal di baris yang sama. Heuristik ini memberikan informasi yang sangat relevan terhadap tingkat kesulitan sesungguhnya dari posisi saat ini, karena mengindikasikan berapa banyak kendaraan yang harus digeser terlebih dahulu. Heuristik ini dapat dipastikan merupakan strategi yang *Admissible*, dikarenakan sebelum kita dalam mengarahkan *primary piece* menuju exit, kita sangat perlu untuk “menggeser” *piece* yang menghalangi. Sehingga nilai $h(n)$ pasti kurang dari atau sama dengan nilai $h^*(n)$.

2.8.4 Clearing Count

Clearing Count adalah heuristik lanjutan yang tidak hanya menghitung jumlah penghalang, tetapi juga memperkirakan jumlah langkah atau pergerakan yang diperlukan untuk membersihkan jalur menuju *exit*. Heuristik ini menganalisis apakah kendaraan penghalang tersebut bisa digerakkan, dan dalam banyak kasus juga mempertimbangkan konfigurasi sekunder (misalnya, apakah kendaraan penghalang sendiri terhalang oleh kendaraan lain). Clearing Count lebih realistis daripada Blocking Count maupun Manhattan Distance, dan bisa lebih informatif jika diimplementasikan dengan benar. Namun, ia cenderung lebih mahal secara komputasi dan memiliki potensi untuk tidak *admissible*, tergantung bagaimana estimasi langkah pembersihan dilakukan.

2.9 Analisis Kompleksitas Algoritma

Setiap algoritma memiliki karakteristik efisiensi dan keoptimalan yang berbeda-beda, tergantung pada bagaimana mereka mengekspansi *node*, mempertimbangkan biaya, dan menggunakan heuristik. Berikut adalah analisis kompleksitas, dan performa tiap algoritma secara teoritikal untuk digunakan dalam penyelesaian masalah puzzle Rush Hour :

Tabel 1 Perbandingan kompleksitas antar algoritma

No.	Algoritma	Analisis
1.	UCS	Kompleksitas Waktu : $O(b^d)$ Kompleksitas Ruang : $O(b^d)$ Keoptimalan : Optimal Kelengkapan : Lengkap Keterangan : UCS memperluas <i>node</i> berdasarkan biaya terkecil dari <i>start</i> . Pada kasus terburuk, UCS menjelajahi seluruh ruang hingga kedalaman solusi d , di mana b adalah <i>branching factor</i> .
2.	GBFS	Kompleksitas Waktu : $O(b^m)$ Kompleksitas Ruang : $O(b^m)$ Keoptimalan : Belum Tentu Optimal Kelengkapan : Tidak Lengkap Keterangan : m adalah jumlah maksimum <i>node</i> yang memiliki nilai heuristik mendekati optimal. Sangat cepat jika heuristik bagus, tetapi mudah tersesat jika heuristik menipu.
3.	A*	Kompleksitas Waktu : $O(b^d)$ Kompleksitas Ruang : $O(b^d)$ Keoptimalan : Optimal Kelengkapan : Lengkap
4.	Dijkstra	Kompleksitas Waktu : $O((V+E)\log V)$ Kompleksitas Ruang : $O(V)$ Keoptimalan : Optimal Kelengkapan : Lengkap Keterangan: Dijkstra hampir identik dengan UCS.
5.	Beam Search	Kompleksitas Waktu : $O(kdb)$

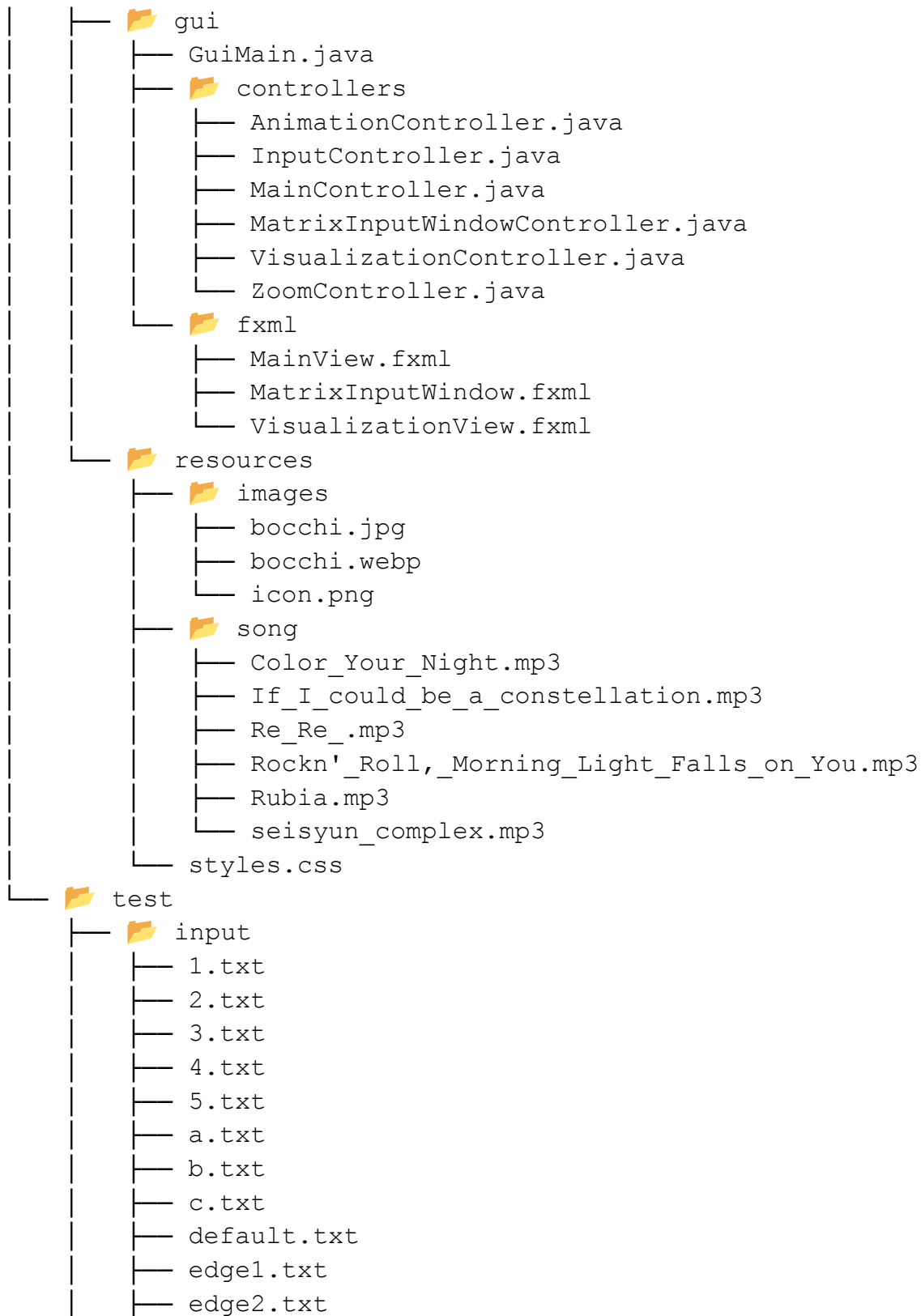
		<p>Kompleksitas Ruang : $O(k)$</p> <p>Keoptimalan : Tidak Optimal</p> <p>Kelengkapan : Tidak Lengkap</p> <p>Keterangan : k adalah <i>Beam Width</i>, d adalah kedalaman, dan b adalah <i>branching factor</i>. Beam Search mengurangi penggunaan memori dan waktu pencarian secara sangat drastis dengan hanya menyimpan k node terbaik pada setiap level. Algoritma ini merupakan <i>trade-off</i> antara efisiensi dan akurasi solusi.</p>
6.	IDA*	<p>Kompleksitas Waktu : $O(b^d)$</p> <p>Kompleksitas Ruang : $O(d)$</p> <p>Keoptimalan : Optimal (Jika heuristik <i>admissible</i>)</p> <p>Kelengkapan : Lengkap</p> <p>Keterangan : IDA* dan A* dapat dianggap sebagai 2 algoritma yang sama persis. Perbedaan dari 2 algoritma ini hanya terletak pada penggunaan memori. Strategi ini merupakan <i>trade-off</i> antara efisiensi memori dengan pencarian yang lebih lama.</p>

BAB III: STRUKTUR REPOSITORY DAN SOURCE CODE

3.1. Struktur Repository

```

Tucil3_13523069_13523090
├── README.md
├── bin
│   ├── KessokuNoOwari.class
│   └── KessokuNoOwari.jar
├── build.gradle
├── doc
│   └── Tucil3_13523069_13523090.pdf
├── env.bat
├── gradle
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradle.properties
├── gradlew
├── gradlew.bat
├── kessoku
├── kessoku.bat
├── settings.gradle
├── src
│   ├── KessokuNoOwari.java
│   └── cli
│       ├── Board.java
│       ├── CompoundMove.java
│       ├── Exit.java
│       ├── FileParser.java
│       ├── Main.java
│       ├── Move.java
│       ├── Orientation.java
│       ├── Piece.java
│       ├── Position.java
│       ├── Solution.java
│       └── Solver.java
```



├─ edge3.txt
├─ edge4.txt
├─ edge5.txt
├─ edge6.txt
├─ edge7.txt
├─ edge8.txt
├─ edge9.txt
├─ error1.txt
├─ error10.txt
├─ error11.txt
├─ error12.txt
├─ error13.txt
├─ error14.txt
├─ error15.txt
├─ error2.txt
├─ error3.txt
├─ error4.txt
├─ error5.txt
├─ error6.txt
├─ error7.txt
├─ error8.txt
├─ error9.txt
├─ gaksejajar.txt
├─ katousignature.txt
├─ kenablokirnjir.txt
├─ misteril.txt
├─ misteri2.txt
├─ misteri3.txt
├─ misteri4.txt
├─ misteri5.txt
├─ sempit.txt
├─ tebaktebakanyuk_adahasilnyaapaenggakya.txt
├─ test.txt
├─ tst.txt
├─ v.txt
├─ wlntr.txt
├─ waduh.txt
├─ wahgaknormalini😂.txt
└─ 📁 output

|— algo_default_Dijkstra.txt
|— algo_default_GBFS.txt
|— algo_default_IDA.txt
|— algo_default_astar.txt
|— algo_default_beam.txt
|— algo_default_ucs.txt
|— algo_misteri4_Dijkstra.txt
|— algo_misteri4_GBFS.txt
|— algo_misteri4_IDA.txt
|— algo_misteri4_astar.txt
|— algo_misteri4_beam.txt
|— algo_misteri4_ucs.txt
|— algo_normalkok_Dijkstra.txt
|— algo_normalkok_GBFS.txt
|— algo_normalkok_IDA.txt
|— algo_normalkok_astar.txt
|— algo_normalkok_beam.txt
|— algo_normalkok_ucs.txt
|— algo_tebak_Dijkstra.txt
|— algo_tebak_GBFS.txt
|— algo_tebak_IDA.txt
|— algo_tebak_astar.txt
|— algo_tebak_beam.txt
|— algo_tebak_ucs.txt
|— block_default.txt
|— block_misteri4.txt
|— block_normalkok.txt
|— block_tebak.txt
|— clear_default.txt
|— clear_misteri4.txt
|— clear_normalkok.txt
|— clear_tebak.txt
|— direct_default.txt
|— direct_misteri4.txt
|— direct_normalkok.txt
|— direct_tebak.txt
|— katou_Dijkstra.txt
|— katou_GBFS.txt
|— katou_IDA.txt


```

├── katou_astar.txt
├── katou_beam.txt
├── katou_ucs.txt
├── manhattan_default.txt
├── manhattan_misteri4.txt
├── manhattan_normalkok.txt
├── manhattan_tebak.txt
├── wlnttr_Dijkstra.txt
├── wlnttr_GBFS.txt
├── wlnttr_IDA.txt
├── wlnttr_astar.txt
├── wlnttr_beam.txt
└── wlnttr_ucs.txt

```

Pada skema di atas, sudah diberikan visualisasi dari struktur repositori dari program ini. Berikut merupakan penjelasan struktur dari isi *repository* tugas kecil *path finding algorithm*.

1. Folder **src** yang berisi source code program.
2. Folder **bin** yang berisi executable file.
3. Folder **test** yang berisi solusi jawaban dari data uji yang digunakan dalam laporan.
4. Folder **doc** yang berisi laporan tugas kecil dalam bentuk PDF.
5. Folder **gradle** yang berisi *wrapper* dari gradle.
6. **README.md** yang berisi:
 - a. Penjelasan singkat program yang dibuat.
 - b. Requirement program dan instalasi tertentu bila ada.
 - c. Cara mengkompilasi program bila perlu dikompilasi (pastikan dengan langkah yang **jelas dan benar**).
 - d. Cara menjalankan dan menggunakan program (pastikan dengan langkah yang **jelas dan benar**).
 - e. Author / identitas pembuat.
7. **settings.gradle**, **build.gradle**, **gradle.properties**, **gradlew**, dan **gradlew.bat** yang merupakan konfigurasi untuk gradle.
8. **kessoku.bat** (Windows) dan **kessoku** (Linux) yang berisi instruksi untuk *construct* program dan *executable file*, dan juga sebagai *runner* dari program.

3.2. Source Code

Berikut adalah source code dari program ini yang berupa kode asli dari program, untuk lebih lengkapnya, source code dapat diakses pada [tautan ini](#). Untuk cara menjalankan program, silakan untuk me-refer ke file “README.md” yang ada pada repository, atau bisa diakses pada [tautan ini](#).

3.2.1 Board.java

```
package cli;

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

/**
 * Board class representing the Rush Hour puzzle grid state
 */
public class Board {
    private int width;
    private int height;
    private char[][] grid;
    private List<Piece> pieces;
    private Piece primaryPiece;
    private Position exitPosition;
    private Exit exitSide;

    public Board(int width, int height) {
        this.width = width;
        this.height = height;
        this.grid = new char[height][width];
        this.pieces = new ArrayList<>();

        // Initialize grid with empty cells
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                grid[i][j] = '.';
            }
        }
    }

    // Copy constructor for creating board states
    public Board(Board other) {
        this.width = other.width;
        this.height = other.height;
        this.grid = new char[height][width];
        this.pieces = new ArrayList<>();
        this.exitSide = other.exitSide;

        // Copy grid
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
```

```

        grid[i][j] = other.grid[i][j];
    }
}

// Copy pieces
for (Piece piece : other.pieces) {
    Piece newPiece = new Piece(piece);
    pieces.add(newPiece);
    if (piece == other.primaryPiece) {
        this.primaryPiece = newPiece;
    }
}

this.exitPosition = new Position(other.exitPosition);
}

public static Board readFromFile(String filename) throws IOException
{
    FileParser.ParsedBoard parsed = FileParser.parseFile(filename);

    Board board = new Board(parsed.cols, parsed.rows);

    // Copy grid
    for (int i = 0; i < parsed.rows; i++) {
        for (int j = 0; j < parsed.cols; j++) {
            board.grid[i][j] = parsed.grid[i][j];
        }
    }

    // Set exit position (normalize to board coordinates for
algorithm)
    if (parsed.exitPosition.row == -1) {
        // Top exit
        board.exitPosition = new Position(0,
parsed.exitPosition.col);
        board.exitSide = Exit.TOP;
    } else if (parsed.exitPosition.row == parsed.rows) {
        // Bottom exit
        board.exitPosition = new Position(parsed.rows - 1,
parsed.exitPosition.col);
        board.exitSide = Exit.BOTTOM;
    } else if (parsed.exitPosition.col == -1) {
        // Left exit
        board.exitPosition = new Position(parsed.exitPosition.row,
0);
        board.exitSide = Exit.LEFT;
    } else if (parsed.exitPosition.col == parsed.cols) {
        // Right exit
        board.exitPosition = new Position(parsed.exitPosition.row,
parsed.cols - 1);
        board.exitSide = Exit.RIGHT;
    } else {
        // Exit is within the grid (should not normally happen)

```

```

        board.exitPosition = new Position(parsed.exitPosition);
        board.exitSide = Exit.NONE;
    }

    // Parse pieces from grid
    Map<Character, List<Position>> piecePositions = new HashMap<>();

    for (int i = 0; i < parsed.rows; i++) {
        for (int j = 0; j < parsed.cols; j++) {
            char c = board.grid[i][j];
            if (c != '.' && c != 'K') { // Exclude K and dots from
pieces
                // Validate character
                if (!Character.isLetter(c)) {
                    throw new IOException("Invalid character '" + c
+ "'" at position (" + i + ", " + j + "). Only letters are allowed for
pieces.");
                }
                if (!Character.isUpperCase(c)) {
                    throw new IOException("Invalid character '" + c
+ "'" at position (" + i + ", " + j + "). Only uppercase letters (A-Z)
are allowed for pieces.");
                }
                piecePositions.computeIfAbsent(c, k -> new
ArrayList<>())
                    .add(new Position(i, j));
            }
        }
    }

    // Create pieces from positions
    int actualPieceCount = 0;
    for (Map.Entry<Character, List<Position>> entry :
piecePositions.entrySet()) {
        char id = entry.getKey();
        List<Position> positions = entry.getValue();

        try {
            // Piece constructor performs validation
            Piece piece = new Piece(id, positions);
            board.pieces.add(piece);
            actualPieceCount++;

            if (id == 'P') {
                board.primaryPiece = piece;
            }
        } catch (IllegalArgumentException e) {
            throw new IOException("Invalid piece configuration: " +
e.getMessage());
        }
    }

    // Validate

```

```

        if (board.primaryPiece == null) {
            throw new IOException("No primary piece (P) found in the
board configuration");
        }

        // Validate number of pieces (excluding primary piece)
        int actualNonPrimaryPieces = actualPieceCount - 1;
        if (actualNonPrimaryPieces != parsed.numPieces) {
            throw new IOException("Number of non-primary pieces
mismatch. Expected " + parsed.numPieces + ", but found " +
actualNonPrimaryPieces);
        }

        System.out.println("Board loaded successfully:");
        System.out.println("- Primary piece: " +
board.primaryPiece.getId());
        System.out.println("- Exit position: " + board.exitPosition);
        System.out.println("- Exit on: " + board.exitSide);
        System.out.println("- Total pieces: " + board.pieces.size());

        return board;
    }

    public List<Move> getPossibleMoves() {
        List<Move> moves = new ArrayList<>();

        for (Piece piece : pieces) {
            // Try moving forward (down for vertical, right for
horizontal)
            if (canMovePiece(piece, 1)) {
                moves.add(new Move(piece, piece.getOrientation() ==
Orientation.HORIZONTAL ? "right" : "down"));
            }

            // Try moving backward (up for vertical, left for
horizontal)
            if (canMovePiece(piece, -1)) {
                moves.add(new Move(piece, piece.getOrientation() ==
Orientation.HORIZONTAL ? "left" : "up"));
            }
        }

        return moves;
    }

    /**
     * Check if a piece can move in a given direction
     * With improved exit detection
     */
    private boolean canMovePiece(Piece piece, int direction) {
        List<Position> positions = piece.getPositions();
        Set<Position> currentPositionsSet = new HashSet<>(positions);

```

```

// Calculate all new positions
List<Position> newPositions = new ArrayList<>();
for (Position pos : positions) {
    Position newPos;

    if (piece.getOrientation() == Orientation.HORIZONTAL) {
        newPos = new Position(pos.row, pos.col + direction);
    } else {
        newPos = new Position(pos.row + direction, pos.col);
    }

    newPositions.add(newPos);
}

// Check if any new position is invalid
for (Position newPos : newPositions) {
    // Special case: primary piece exiting
    if (piece == primaryPiece && exitPosition != null) {
        // Check if this move would reach or exit through the
exit
        if (piece.getOrientation() == Orientation.HORIZONTAL) {
            if (newPos.row == exitPosition.row) {
                if ((exitSide == Exit.RIGHT && newPos.col >=
width) ||
                    (exitSide == Exit.LEFT && newPos.col < 0)) {
                    return true; // Allow exit
                }
            }
        } else { // VERTICAL
            if (newPos.col == exitPosition.col) {
                if ((exitSide == Exit.BOTTOM && newPos.row >=
height) ||
                    (exitSide == Exit.TOP && newPos.row < 0)) {
                    return true; // Allow exit
                }
            }
        }
    }
}

// Normal bounds check
if (newPos.row < 0 || newPos.row >= height ||
    newPos.col < 0 || newPos.col >= width) {
    return false;
}

// Check collision with other pieces
char cellContent = grid[newPos.row][newPos.col];

// Cell must be empty or occupied by the current piece
if (cellContent != '.' && cellContent != piece.getId()) {
    return false;
}

```

```

        // If occupied by the same piece, make sure it's a position
        that will be vacated
        if (cellContent == piece.getId() &&
!currentPositionsSet.contains(newPos)) {
            return false;
        }
    }

    return true;
}

public Piece getPieceAt(int row, int col) {
    for (Piece piece : pieces) {
        for (Position pos : piece.getPositions()) {
            if (pos.row == row && pos.col == col) {
                return piece;
            }
        }
    }
    return null;
}

/**
 * Make a move on the board, with improved exit handling
 */
public Board makeMove(Move move) {
    Board newBoard = new Board(this);
    Piece piece = newBoard.getPieceById(move.getPiece().getId());
    int direction = move.getDirectionValue();

    // Check if this is an exit move for the primary piece
    boolean isExitMove = false;
    if (piece == newBoard.primaryPiece) {
        if (piece.getOrientation() == Orientation.HORIZONTAL) {
            if ((exitSide == Exit.RIGHT && direction > 0) ||
                (exitSide == Exit.LEFT && direction < 0)) {
                // Check if this move would position the primary
                piece at the exit
                for (Position pos : piece.getPositions()) {
                    Position newPos = new Position(pos.row, pos.col
+ direction);
                    if ((exitSide == Exit.RIGHT && newPos.col >=
width && pos.row == exitPosition.row) ||
                        (exitSide == Exit.LEFT && newPos.col < 0 &&
pos.row == exitPosition.row)) {
                        isExitMove = true;
                        break;
                    }
                }
            }
        } else { // VERTICAL
            if ((exitSide == Exit.BOTTOM && direction > 0) ||
                (exitSide == Exit.TOP && direction < 0)) {

```

```

        // Check if this move would position the primary
piece at the exit
        for (Position pos : piece.getPositions()) {
            Position newPos = new Position(pos.row +
direction, pos.col);
            if ((exitSide == Exit.BOTTOM && newPos.row >=
height && pos.col == exitPosition.col) ||
                (exitSide == Exit.TOP && newPos.row < 0 &&
pos.col == exitPosition.col)) {
                isExitMove = true;
                break;
            }
        }
    }
}

// Clear current positions
for (Position pos : piece.getPositions()) {
    if (pos.row >= 0 && pos.row < height && pos.col >= 0 &&
pos.col < width) {
        newBoard.grid[pos.row][pos.col] = '.';
    }
}

// Update piece positions
piece.move(direction);

// Only set new positions if it's not an exit move for primary
piece
if (!isExitMove) {
    // Set new positions (only if within bounds)
    for (Position pos : piece.getPositions()) {
        if (pos.row >= 0 && pos.row < height && pos.col >= 0 &&
pos.col < width) {
            newBoard.grid[pos.row][pos.col] = piece.getId();
        }
    }
} else {
    // For exit moves, leave primary piece positions empty
    // Remove primary piece from the list to ensure it won't be
considered again
    newBoard.pieces.remove(piece);
    newBoard.primaryPiece = null; // Clear primary piece
reference
}

    return newBoard;
}

public boolean isSolved() {
    if (primaryPiece == null || exitPosition == null) {
        return false;
    }
}

```



```

    }

    // Get positions of the primary piece
    List<Position> primaryPositions = primaryPiece.getPositions();

    // Check based on exit side and primary piece orientation
    if (primaryPiece.getOrientation() == Orientation.HORIZONTAL) {
        // For horizontal primary piece, check if it's at exit
        if (exitSide == Exit.RIGHT) {
            // Find rightmost position of the piece
            Position rightmost = primaryPositions.stream()
                .max(Comparator.comparingInt(p -> p.col))
                .orElse(primaryPositions.get(0));

            // Primary piece solved if rightmost position is at the
            rightmost cell AND same row as exit
            return rightmost.col == width - 1 && rightmost.row ==
            exitPosition.row;
        }
        else if (exitSide == Exit.LEFT) {
            // Find leftmost position of the piece
            Position leftmost = primaryPositions.stream()
                .min(Comparator.comparingInt(p -> p.col))
                .orElse(primaryPositions.get(0));

            // Primary piece solved if leftmost position is at the
            leftmost cell AND same row as exit
            return leftmost.col == 0 && leftmost.row ==
            exitPosition.row;
        }
    }
    else if (primaryPiece.getOrientation() == Orientation.VERTICAL)
    {
        // For vertical primary piece, check if it's at exit
        if (exitSide == Exit.BOTTOM) {
            // Find bottommost position of the piece
            Position bottommost = primaryPositions.stream()
                .max(Comparator.comparingInt(p -> p.row))
                .orElse(primaryPositions.get(0));

            // Primary piece solved if bottommost position is at the
            bottom cell AND same column as exit
            return bottommost.row == height - 1 && bottommost.col ==
            exitPosition.col;
        }
        else if (exitSide == Exit.TOP) {
            // Find topmost position of the piece
            Position topmost = primaryPositions.stream()
                .min(Comparator.comparingInt(p -> p.row))
                .orElse(primaryPositions.get(0));

            // Primary piece solved if topmost position is at the
            top cell AND same column as exit

```

```

        return topmost.row == 0 && topmost.col ==
exitPosition.col;
    }

    return false;
}

public void display() {
    // Use ANSI colors for better visualization
    String RESET = "\u001B[0m";
    String RED = "\u001B[31m";    // Primary piece
    String GREEN = "\u001B[32m";  // Exit

    // Display top exit if exists
    if (exitSide == Exit.TOP) {
        for (int j = 0; j < width; j++) {
            if (j == exitPosition.col) {
                System.out.print(GREEN + "K" + RESET);
            } else {
                System.out.print(" ");
            }
        }
        System.out.println();
    }

    // Display board content
    for (int i = 0; i < height; i++) {
        // Left exit
        if (exitSide == Exit.LEFT && i == exitPosition.row) {
            System.out.print(GREEN + "K" + RESET);
        }

        // Board content
        for (int j = 0; j < width; j++) {
            char c = grid[i][j];
            if (c == 'P') {
                System.out.print(RED + c + RESET);
            } else {
                System.out.print(c);
            }
        }

        // Right exit
        if (exitSide == Exit.RIGHT && i == exitPosition.row) {
            System.out.print(GREEN + "K" + RESET);
        }

        System.out.println();
    }

    // Display bottom exit if exists
    if (exitSide == Exit.BOTTOM) {

```

```

        for (int j = 0; j < width; j++) {
            if (j == exitPosition.col) {
                System.out.print(GREEN + "K" + RESET);
            } else {
                System.out.print(" ");
            }
        }
        System.out.println();
    }
}

public void displayWithMove(Move lastMove) {
    String RESET = "\u001B[0m";
    String RED = "\u001B[31m";    // Primary piece
    String GREEN = "\u001B[32m";  // Exit
    String YELLOW = "\u001B[33m"; // Moving piece

    char movingPieceId = lastMove != null ?
lastMove.getPiece().getId() : '\0';

    // Display top exit if exists
    if (exitSide == Exit.TOP) {
        for (int j = 0; j < width; j++) {
            if (j == exitPosition.col) {
                System.out.print(GREEN + "K" + RESET);
            } else {
                System.out.print(" ");
            }
        }
        System.out.println();
    }

    // Display board content
    for (int i = 0; i < height; i++) {
        // Left exit
        if (exitSide == Exit.LEFT && i == exitPosition.row) {
            System.out.print(GREEN + "K" + RESET);
        }

        // Board content
        for (int j = 0; j < width; j++) {
            char c = grid[i][j];
            if (c == 'P') {
                // Primary piece always stays red
                System.out.print(RED + c + RESET);
            } else if (c == movingPieceId) {
                // Other moving pieces are yellow
                System.out.print(YELLOW + c + RESET);
            } else {
                System.out.print(c);
            }
        }
    }
}

```

```

        // Right exit
        if (exitSide == Exit.RIGHT && i == exitPosition.row) {
            System.out.print(GREEN + "K" + RESET);
        }

        System.out.println();
    }

    // Display bottom exit if exists
    if (exitSide == Exit.BOTTOM) {
        for (int j = 0; j < width; j++) {
            if (j == exitPosition.col) {
                System.out.print(GREEN + "K" + RESET);
            } else {
                System.out.print(" ");
            }
        }
        System.out.println();
    }
}

public String getStateString() {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            sb.append(grid[i][j]);
        }
    }
    return sb.toString();
}

public Piece getPieceById(char id) {
    for (Piece piece : pieces) {
        if (piece.getId() == id) {
            return piece;
        }
    }
    return null;
}

public boolean isPrimaryPieceAlignedWithExit() {
    if (primaryPiece == null || exitPosition == null) {
        return false;
    }

    // Check alignment based on primary piece orientation and exit
    position
    if (primaryPiece.getOrientation() == Orientation.HORIZONTAL) {
        // For horizontal primary piece, exit must be on left or
        right AND same row
        if ((exitSide == Exit.LEFT || exitSide == Exit.RIGHT)) {
            // Check if primary piece shares the same row with exit
            for (Position pos : primaryPiece.getPositions()) {

```

```

        if (pos.row == exitPosition.row) {
            return true;
        }
    }
} else { // VERTICAL
    // For vertical primary piece, exit must be on top or bottom
    AND same column
    if ((exitSide == Exit.TOP || exitSide == Exit.BOTTOM)) {
        // Check if primary piece shares the same column with
        exit
        for (Position pos : primaryPiece.getPositions()) {
            if (pos.col == exitPosition.col) {
                return true;
            }
        }
    }
}

return false;
}

/**
 * Display board with improved visualization for exit state
 */
public void displayWithFinalState() {
    // Use ANSI colors for better visualization
    String RESET = "\u001B[0m";
    String RED = "\u001B[31m";    // Primary piece
    String GREEN = "\u001B[32m";  // Exit

    // Display top exit if exists
    if (exitSide == Exit.TOP) {
        for (int j = 0; j < width; j++) {
            if (j == exitPosition.col) {
                // Show primary piece in the exit
                if (primaryPiece != null &&
                    primaryPiece.getOrientation() == Orientation.VERTICAL) {
                    System.out.print(RED + "P" + RESET);
                } else {
                    System.out.print(GREEN + "K" + RESET);
                }
            } else {
                System.out.print(" ");
            }
        }
        System.out.println();
    }

    // Display board content
    for (int i = 0; i < height; i++) {
        // Left exit
        if (exitSide == Exit.LEFT && i == exitPosition.row) {

```

```

        // Show primary piece in the exit
        if (primaryPiece != null &&
primaryPiece.getOrientation() == Orientation.HORIZONTAL) {
            System.out.print(RED + "P" + RESET);
        } else {
            System.out.print(GREEN + "K" + RESET);
        }
    }

    // Board content
    for (int j = 0; j < width; j++) {
        char c = grid[i][j];
        if (c == 'P') {
            System.out.print(RED + c + RESET);
        } else {
            System.out.print(c);
        }
    }

    // Right exit
    if (exitSide == Exit.RIGHT && i == exitPosition.row) {
        // Show primary piece in the exit for the final state
        if (this.isSolved() || primaryPiece == null) {
            System.out.print(RED + "P" + RESET);
        } else {
            System.out.print(GREEN + "K" + RESET);
        }
    }

    System.out.println();
}

// Display bottom exit if exists
if (exitSide == Exit.BOTTOM) {
    for (int j = 0; j < width; j++) {
        if (j == exitPosition.col) {
            // Show primary piece in the exit
            if (primaryPiece != null &&
primaryPiece.getOrientation() == Orientation.VERTICAL) {
                System.out.print(RED + "P" + RESET);
            } else {
                System.out.print(GREEN + "K" + RESET);
            }
        } else {
            System.out.print(" ");
        }
    }
    System.out.println();
}

}

/**
 * Display board with primary piece shown next to exit K

```

```

    */
    public void displayWithExitedPiece() {
        // Use ANSI colors for better visualization
        String RESET = "\u001B[0m";
        String RED = "\u001B[31m";    // Primary piece
        String GREEN = "\u001B[32m";  // Exit

        // Display top exit if exists
        if (exitSide == Exit.TOP) {
            for (int j = 0; j < width; j++) {
                if (j == exitPosition.col) {
                    // Show K with P outside
                    System.out.print(GREEN + "K" + RED + "P" + RESET);
                    j++; // Skip one column as we printed two characters
                } else {
                    System.out.print(" ");
                }
            }
            System.out.println();
        }

        // Display board content
        for (int i = 0; i < height; i++) {
            // Left exit
            if (exitSide == Exit.LEFT && i == exitPosition.row) {
                // Show P outside, then K
                System.out.print(RED + "P" + GREEN + "K" + RESET);
            }

            // Board content
            for (int j = 0; j < width; j++) {
                char c = grid[i][j];
                // Don't show P in the board for final state (it's
exited)
                if (c == 'P' && this.isSolved()) {
                    System.out.print('.');
                } else if (c == 'P') {
                    System.out.print(RED + c + RESET);
                } else {
                    System.out.print(c);
                }
            }

            // Right exit
            if (exitSide == Exit.RIGHT && i == exitPosition.row) {
                // Show K then P outside
                System.out.print(GREEN + "K" + RED + "P" + RESET);
            }

            System.out.println();
        }

        // Display bottom exit if exists

```

```

        if (exitSide == Exit.BOTTOM) {
            for (int j = 0; j < width; j++) {
                if (j == exitPosition.col) {
                    // Show K with P outside
                    System.out.print(GREEN + "K" + RED + "P" + RESET);
                    j++; // Skip one column as we printed two characters
                } else {
                    System.out.print(" ");
                }
            }
            System.out.println();
        }
    }

    // Getters
    public int getWidth() { return width; }
    public int getHeight() { return height; }
    public List<Piece> getPieces() { return pieces; }
    public Piece getPrimaryPiece() { return primaryPiece; }
    public Position getExitPosition() { return exitPosition; }
    public Exit getExitSide() { return exitSide; }
    public char[][] getGrid() { return grid; }

    public char getGridAt(int row, int col) {
        if (row >= 0 && row < height && col >= 0 && col < width) {
            return grid[row][col];
        }
        return ' ';
    }
}

```

3.2.2 CompoundMove.java

```

package cli;

/**
 * Represents a compound move (multi-cell movement in one direction)
 * This is used to represent moves like "A+3" in the solution sequence
 */
public class CompoundMove extends Move {
    private int distance;

    /**
     * Create a compound move
     * @param piece The piece to move
     * @param direction The direction ("up", "down", "left", "right")
     * @param distance The number of cells to move
     */
    public CompoundMove(Piece piece, String direction, int distance) {

```



```

        super(piece, direction);
        this.distance = distance;
    }

    /**
     * Get the distance of this compound move
     */
    public int getDistance() {
        return distance;
    }

    /**
     * Format the move as a string like "UP×3" or "DOWN×2"
     */
    @Override
    public String toString() {
        return getDirection().toUpperCase() + " (" + distance + ")";
    }
}

```

3.2.3 *Exit.java*

```

package cli;

public enum Exit {
    TOP,
    BOTTOM,
    LEFT,
    RIGHT,
    NONE
}

```

3.2.4 *FileParser.java*

```

package cli;

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

/**
 * FileParser class for parsing Rush Hour puzzle files with specific
 * exit placement rules
 */
public class FileParser {

```

```

        public static ParsedBoard parseFile(String filename) throws
IOException {
            BufferedReader reader = new BufferedReader(new
FileReader(filename));

            try {
                // Dimensions
                String firstLine = reader.readLine();
                if (firstLine == null) {
                    throw new IOException("File is empty");
                }
                firstLine = firstLine.trim();

                String[] dimensions = firstLine.split("\\s+");
                if (dimensions.length != 2) {
                    throw new IOException("First line must contain exactly 2
integers for board dimensions");
                }

                int rows, cols;
                try {
                    rows = Integer.parseInt(dimensions[0]);
                    cols = Integer.parseInt(dimensions[1]);
                } catch (NumberFormatException e) {
                    throw new IOException("First line must contain valid
integers for board dimensions");
                }

                // Validate dimensions
                if (rows <= 0) {
                    throw new IOException("Number of rows must be greater
than 0, got: " + rows);
                }
                if (cols <= 0) {
                    throw new IOException("Number of columns must be greater
than 0, got: " + cols);
                }

                if (rows == 1 && cols == 1) {
                    throw new IOException("Board size must be at least 1x2
or 2x1. Current size: " + rows + "x" + cols);
                }

                // Read number of pieces
                String secondLine = reader.readLine();
                if (secondLine == null) {
                    throw new IOException("Missing second line for number of
pieces");
                }
                secondLine = secondLine.trim();

                int numPieces;

```

```

        try {
            numPieces = Integer.parseInt(secondLine);
        } catch (NumberFormatException e) {
            throw new IOException("Second line must contain a valid
integer for number of pieces");
        }

        // Validate number of pieces
        if (numPieces < 1) {
            throw new IOException("Number of pieces cannot be less
than 1, got: " + numPieces);
        }

        // Read all remaining lines
        List<String> allLines = new ArrayList<>();
        String line;
        while ((line = reader.readLine()) != null) {
            allLines.add(line);
        }

        // Exit position and count
        Position exitPosition = null;
        Orientation exitOrientation = null;
        int exitCount = 0;
        List<String> exitPositionDescriptions = new ArrayList<>();
        boolean hasLeftK = false;
        int leftKRow = -1;

        // First pass: Check for exit positions and count total
exits
        for (int i = 0; i < allLines.size(); i++) {
            String currentLine = allLines.get(i);
            if (currentLine.trim().isEmpty()) continue;

            if (currentLine.contains("K")) {
                // Check if this is a top K (before board)
                if (i == 0) {
                    // If the first line has only K or starts with K
and has no other letters, it's a top exit
                    if (currentLine.trim().equals("K") ||
                        (currentLine.indexOf('K') >= 0 &&
!containsAnyPiece(currentLine))) {

                        int kPosition = currentLine.indexOf('K');

                        // Check if K is within the board's column
range (0 to cols-1)
                        if (kPosition >= cols) {
                            throw new IOException("Invalid exit
position 'K' at position (" +
                                i + ", " + kPosition
+ "). Top K must be within the board's column range (0 to " +
                                (cols-1) + ").");

```

```

        }

        exitCount++;
        exitPositionDescriptions.add("TOP (column "
+ kPosition + ")");
        exitPosition = new Position(-1, kPosition);
        exitOrientation = Orientation.VERTICAL;
    }
    // Otherwise, treat it as a normal board row
    that could have left or right K
    else {
        if (currentLine.startsWith("K")) {
            exitCount++;
            exitPositionDescriptions.add("LEFT (row
" + i + ")");
            exitPosition = new Position(i, -1);
            exitOrientation =
Orientation.HORIZONTAL;

            hasLeftK = true;
            leftKRow = i;
        }
        // Check for right K (including corner
cases)
        else if (currentLine.endsWith("K") ||
                (currentLine.length() > cols &&
currentLine.charAt(cols) == 'K') ||
                (currentLine.length() >= cols &&
currentLine.charAt(currentLine.length() - 1) == 'K')) {
            exitCount++;
            // Check if it's a corner K
            if (currentLine.length() >= cols &&
currentLine.charAt(currentLine.length() - 1) == 'K') {
                String cornerType = "UPPER RIGHT
CORNER";
                exitPositionDescriptions.add(cornerType + " (row " + i + ")");
            } else {
                exitPositionDescriptions.add("RIGHT
(row " + i + ")");
            }

            exitPosition = new Position(i, cols);
            exitOrientation =
Orientation.HORIZONTAL;
        }
        // K inside the board (invalid)
        else {
            int kPosition =
currentLine.indexOf('K');
            throw new IOException("Invalid exit
position 'K' found inside the board at position (" +
                i + ", " + kPosition

```

```

+ "). Exit must be placed on the edge of the board.");
        }
    }
    // Check if this is a bottom K (after all board
rows)
    else if (i >= rows) {
        int kPosition = currentLine.indexOf('K');

        // Check if K is within the board's column range
(0 to cols-1)
        if (kPosition >= cols) {
            throw new IOException("Invalid exit position
'K' at position (" +
                                i + ", " + kPosition +
"). Bottom K must be within the board's column range (0 to " +
                                (cols-1) + ").");
        }

        exitCount++;
        exitPositionDescriptions.add("BOTTOM (column " +
kPosition + ")");

        exitPosition = new Position(rows, kPosition);
        exitOrientation = Orientation.VERTICAL;
    }
    // Check for left K
    else if (currentLine.startsWith("K")) {
        exitCount++;
        exitPositionDescriptions.add("LEFT (row " + i +
")");

        exitPosition = new Position(i, -1);
        exitOrientation = Orientation.HORIZONTAL;
        hasLeftK = true;
        leftKRow = i;
    }
    // Check for right K (including corner cases)
    else if (currentLine.endsWith("K") ||
            (currentLine.length() > cols &&
currentLine.charAt(cols) == 'K') ||
            (currentLine.length() == cols &&
currentLine.charAt(cols - 1) == 'K')) {

        exitCount++;
        // Check if it's a corner K
        if (currentLine.length() == cols &&
currentLine.charAt(cols - 1) == 'K') {
            String cornerType = (i == 0) ? "UPPER RIGHT
CORNER" :
                                (i == rows - 1) ? "LOWER
RIGHT CORNER" : "RIGHT CORNER";
            exitPositionDescriptions.add(cornerType + "
(row " + i + ")");
        } else {

```

```

                                exitPositionDescriptions.add("RIGHT (row " +
i + ")");
                                }

                                exitPosition = new Position(i, cols);
                                exitOrientation = Orientation.HORIZONTAL;
                                }
                                // K inside the board (invalid)
                                else {
                                    int kPosition = currentLine.indexOf('K');
                                    throw new IOException("Invalid exit position 'K'
found inside the board at position (" +
                                                i + ", " + kPosition + ").
Exit must be placed on the edge of the board.");
                                }
                                }
                                }

                                // Check if we found an exit
                                if (exitPosition == null) {
                                    throw new IOException("No exit position (K) found in the
board configuration");
                                }

                                // Check if we have more than one exit
                                if (exitCount > 1) {
                                    throw new IOException("Multiple exit positions (K)
found: " + String.join(", ", exitPositionDescriptions) +
                                                ". The Rush Hour puzzle must have
exactly one exit position.");
                                }

                                // Second pass: Process the board
                                char[][] grid = new char[rows][cols];
                                int currentRow = 0;
                                int lineIndex = 0;

                                // Skip top exit line if it exists
                                if (exitPosition.row == -1) {
                                    lineIndex = 1;
                                }

                                // Initialize grid with empty cells
                                for (int i = 0; i < rows; i++) {
                                    for (int j = 0; j < cols; j++) {
                                        grid[i][j] = '.';
                                    }
                                }

                                // Process the board rows
                                while (currentRow < rows && lineIndex < allLines.size()) {
                                    String currentLine = allLines.get(lineIndex);

```

```

        // Skip empty lines
        if (currentLine.trim().isEmpty()) {
            lineIndex++;
            continue;
        }

        String processedLine = currentLine;
        boolean hasRightK = false;

        // Special handling for left K
        if (hasLeftK) {
            // If this is the row with K, remove the K for grid
            if (lineIndex == leftKRow) {
                processedLine = currentLine.substring(1);
            }
            // All other rows should have exactly one leading
            space
            else {
                if (!currentLine.startsWith(" ") ||
                currentLine.startsWith("  ")) {
                    throw new IOException("Invalid format: When
                    K is at the left, all other rows must have exactly one leading space.
                    Row " +
                    (lineIndex + 1) + " has
                    incorrect spacing.");
                }
                processedLine = currentLine.substring(1); //
            Remove leading space
            }
        }

        // If this line has a right K, remove it for grid
        processing
        if (currentLine.endsWith("K")) {
            processedLine = currentLine.substring(0,
            currentLine.length() - 1);
            hasRightK = true;
        } else if (currentLine.length() > cols &&
        currentLine.charAt(cols) == 'K') {
            processedLine = currentLine.substring(0, cols);
            hasRightK = true;
        }

        // Special case: Check for corner K (K at the end of
        line)
        if (!hasRightK && currentLine.length() == cols &&
        currentLine.charAt(cols - 1) == 'K') {
            // This is a corner K - it's treated as a right edge
            exit
            processedLine = processedLine.substring(0, cols - 1)
            + ".";
            hasRightK = true;

```

```

        }

        // Validate processed line length
        if (processedLine.length() > cols) {
            processedLine = processedLine.substring(0, cols); //
Truncate if too long
        }
        else if (processedLine.length() < cols) {
            throw new IOException("Row " + (lineIndex + 1) + "
has incorrect length. Expected " + cols +
                                ", got " +
processedLine.length() + ". Line: \"" + processedLine + "\"");
        }

        // Map the grid cells
        for (int j = 0; j < processedLine.length(); j++) {
            char c = processedLine.charAt(j);

            // Validate character
            if (c != '.' && !Character.isLetter(c)) {
                throw new IOException("Invalid character '" + c
+ "'" at row " + (lineIndex + 1) +
                                ", column " + (j + 1) + ".
Only letters (A-Z) and dots (.) are allowed.");
            }

            // Ensure uppercase letters
            if (Character.isLetter(c) &&
!Character.isUpperCase(c)) {
                throw new IOException("Invalid character '" + c
+ "'" at row " + (lineIndex + 1) +
                                ", column " + (j + 1) + ".
Only uppercase letters (A-Z) are allowed.");
            }

            // K should have been handled earlier (replaced with
            .)
            if (c == 'K' && !hasRightK) {
                throw new IOException("Invalid exit position 'K'
found inside the board at position (" +
                                (lineIndex + 1) + ", " + (j
+ 1) + "). Exit must be placed on the edge of the board.");
            }

            grid[currentRow][j] = c;
        }

        currentRow++;
        lineIndex++;
    }

    // Validate that we have enough rows
    if (currentRow < rows) {

```



```

        throw new IOException("Not enough rows in file. Expected
" + rows + ", got " + currentRow);
    }

    // No additional validation for K attachment to pieces
needed

    return new ParsedBoard(rows, cols, grid, exitPosition,
exitOrientation, numPieces);

    } finally {
        reader.close();
    }
}

/**
 * Helper method to check if a line contains any piece character
 */
private static boolean containsAnyPiece(String line) {
    for (char c : line.toCharArray()) {
        if (Character.isLetter(c) && c != 'K') {
            return true;
        }
    }
    return false;
}

// Inner class to hold parsed data
public static class ParsedBoard {
    public final int rows;
    public final int cols;
    public final char[][] grid;
    public final Position exitPosition;
    public final Orientation exitOrientation;
    public final int numPieces;

    public ParsedBoard(int rows, int cols, char[][] grid, Position
exitPosition,
        Orientation exitOrientation, int numPieces) {
        this.rows = rows;
        this.cols = cols;
        this.grid = grid;
        this.exitPosition = exitPosition;
        this.exitOrientation = exitOrientation;
        this.numPieces = numPieces;
    }
}
}

```

3.2.5 Main.java

```
package cli;

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

/**
 * Main class for the Rush Hour puzzle solver
 * This version supports compound moves as shown in the sample image
 */
public class Main {
    // ANSI color codes
    private static final String RESET = "\u001B[0m";
    private static final String YELLOW = "\u001B[33m";
    private static final String RED = "\u001B[31m";

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.println("=== Rush Hour Puzzle Solver ===");
        System.out.print("Enter the filename (without .txt extension):");
        String filename = scanner.nextLine().trim();

        // Add .txt extension if not present
        if (!filename.endsWith(".txt")) {
            filename = filename + ".txt";
        }

        // Use relative path to test/input folder
        String filepath = "test/input/" + filename;

        try {
            // Read the puzzle configuration
            System.out.println("Reading file: " + filepath);
            Board board = Board.readFromFile(filepath);
            System.out.println("\nInitial Board State:");
            board.display();

            // Check if primary piece is aligned with exit
            if (!board.isPrimaryPieceAlignedWithExit()) {
                System.out.println();
                System.out.println(YELLOW + "WARNING: Primary piece (P) and exit position (K) are not aligned!" + RESET);
                System.out.println(YELLOW + "The primary piece may not be able to reach the exit with this configuration." + RESET);
                System.out.print("Do you want to continue anyway? (y/n):");
            }
        }
    }
}
```

```

        String continueChoice = getYesNoChoice(scanner);

        if (!continueChoice.equalsIgnoreCase("y")) {
            System.out.println("Exiting program...");
            scanner.close();
            return;
        }
    }

    System.out.println("\nChoose algorithm:");
    System.out.println("1. Uniform Cost Search (UCS)");
    System.out.println("2. Greedy Best First Search");
    System.out.println("3. A* Search");
    System.out.print("Enter your choice (1-3): ");

    int algorithmChoice = getAlgorithmChoice(scanner);

    // Ask for heuristic if using Greedy or A*
    String heuristic = "manhattan";
    if (algorithmChoice == 2 || algorithmChoice == 3) {
        System.out.println("\nChoose heuristic:");
        System.out.println("1. Manhattan Distance");
        System.out.println("2. Direct Distance");
        System.out.println("3. Blocking Count");
        System.out.print("Enter your choice (1-3): ");

        int heuristicChoice = getHeuristicChoice(scanner);
        switch (heuristicChoice) {
            case 1: heuristic = "manhattan"; break;
            case 2: heuristic = "direct"; break;
            case 3: heuristic = "blocking"; break;
            default: heuristic = "manhattan";
        }
    }

    // Solve the puzzle
    Solver solver = new Solver();
    Solution solution = null;
    String algorithmUsed = "";

    long startTime = System.currentTimeMillis();

    switch (algorithmChoice) {
        case 1:
            solution = solver.solveUCS(board, false);
            algorithmUsed = "Uniform Cost Search (UCS)";
            break;
        case 2:
            solution = solver.solveGreedy(board, heuristic,
false);
            algorithmUsed = "Greedy Best First Search with " +
heuristic + " heuristic";
            break;
    }

```

```

        case 3:
            solution = solver.solveAStar(board, heuristic,
false);
            algorithmUsed = "A* Search with " + heuristic + "
heuristic";
            break;
        default:
            System.out.println("Using UCS as default.");
            solution = solver.solveUCS(board, false);
            algorithmUsed = "Uniform Cost Search (UCS)";
    }

    long endTime = System.currentTimeMillis();

    // Display results
    if (solution != null) {
        System.out.println("\nSolution found!");
        System.out.println("Algorithm used: " + algorithmUsed);
        System.out.println("Number of states examined: " +
solution.getStatesExamined());
        System.out.println("Number of moves: " +
solution.getMoves().size());
        System.out.println("Execution time: " + (endTime -
startTime) + " ms");

        System.out.println("\nSolution steps:");
        solution.displaySolution();

        // Ask if user wants to save the solution
        System.out.print("\nDo you want to save the solution to
a file? (y/n): ");
        String saveChoice = getYesNoChoice(scanner);

        if (saveChoice.equalsIgnoreCase("y")) {
            // Save solution to file
            String outputFilename = filename.replace(".txt",
""");
            String algoPrefix = algorithmChoice == 1 ? "ucs_" :
(algorithmChoice == 2 ? "greedy_"
: "astar_");
            String outputPath = "test/output/" + algoPrefix +
"output_" + outputFilename + ".txt";

            // Check if file already exists
            File outputFile = new File(outputPath);
            boolean shouldSave = true;

            if (outputFile.exists()) {
                System.out.print("File " + outputPath + "
already exists. Do you want to overwrite it? (y/n): ");
                String overwriteChoice =
getYesNoChoice(scanner);

```

```

        if (!overwriteChoice.equalsIgnoreCase("y")) {
            shouldSave = false;
            System.out.println("File not saved.");
        }
    }

    if (shouldSave) {
        try {
            saveSolutionToFile(board, solution, endTime
- startTime, outputPath, algorithmUsed);
            System.out.println("Solution saved to: " +
outputPath);
        } catch (IOException e) {
            System.err.println("Error saving solution to
file: " + e.getMessage());
        }
    }
    } else {
        System.out.println("Solution not saved.");
    }
} else {
    System.out.println("\nNo solution found!");
}

} catch (IOException e) {
    String errorMessage = e.getMessage();
    if (errorMessage.contains("No such file or directory") ||
errorMessage.contains("The system cannot find the file")) {
        System.err.println("Error: File not found - " +
filepath);
        System.err.println("Make sure the file exists in the
test/input/ directory");
    } else {
        System.err.println("Error reading file: " +
errorMessage);
    }
} catch (Exception e) {
    System.err.println("Error: " + e.getMessage());
    e.printStackTrace();
}

scanner.close();
}

/**
 * Write final board state to file with primary piece shown next to
exit
 */
private static void writeFinalBoardWithExitedPiece(PrintWriter
writer, Board board) {
    int width = board.getWidth();
    int height = board.getHeight();

```

```

Exit exitSide = board.getExitSide();
Position exitPosition = board.getExitPosition();

// Top exit if exists
if (exitSide == Exit.TOP) {
    for (int j = 0; j < width; j++) {
        if (j == exitPosition.col) {
            writer.print("KP"); // Show K with P outside
            j++; // Skip one column as we printed two characters
        } else {
            writer.print(" ");
        }
    }
    writer.println();
}

// Board content
for (int i = 0; i < height; i++) {
    // Left exit
    if (exitSide == Exit.LEFT && i == exitPosition.row) {
        writer.print("PK"); // Show P outside, then K
    }

    // Board cells
    for (int j = 0; j < width; j++) {
        char c = board.getGridAt(i, j);
        // Don't show P in the board for final state (it's
exited)
        if (c == 'P') {
            writer.print('.');
        } else {
            writer.print(c);
        }
    }

    // Right exit
    if (exitSide == Exit.RIGHT && i == exitPosition.row) {
        writer.print("KP"); // Show K then P outside
    }

    writer.println();
}

// Bottom exit if exists
if (exitSide == Exit.BOTTOM) {
    for (int j = 0; j < width; j++) {
        if (j == exitPosition.col) {
            writer.print("KP"); // Show K with P outside
            j++; // Skip one column as we printed two characters
        } else {
            writer.print(" ");
        }
    }
}

```

```

        writer.println();
    }
}

/**
 * Save solution to a file with P shown next to exit K
 */
private static void saveSolutionToFile(Board initialBoard, Solution
solution, long executionTime,
                                     String outputPath, String
algorithmUsed) throws IOException {
    // Create output directory if it doesn't exist
    File outputDir = new File("test/output");
    if (!outputDir.exists()) {
        outputDir.mkdirs();
    }

    // Write solution to file
    try (PrintWriter writer = new PrintWriter(new
FileWriter(outputPath))) {
        // Write execution info
        writer.println("=== Rush Hour Solution ===");
        writer.println("Algorithm used: " + algorithmUsed);
        writer.println("Number of states examined: " +
solution.getStatesExamined());
        writer.println("Number of moves: " +
solution.getMoves().size());
        writer.println("Execution time: " + executionTime + " ms");
        writer.println();

        // Write move sequence
        writer.println("Move sequence:");
        writer.println(formatMoveSequence(solution.getMoves()));
        writer.println();

        // Write initial board
        writer.println("Papan Awal");
        writeBoard(writer, solution.getStates().get(0));

        // Write each move except the last
        List<Move> moves = solution.getMoves();
        List<Board> states = solution.getStates();

        for (int i = 0; i < moves.size() - 1; i++) {
            writer.println();
            writer.println("Gerakan " + (i + 1) + ": " +
moves.get(i));
            writeBoard(writer, states.get(i + 1));
        }

        // Write the final move with special visualization
        if (moves.size() > 0) {
            int lastIndex = moves.size() - 1;

```

```

        writer.println();
        writer.println("Gerakan " + (lastIndex + 1) + ": " +
moves.get(lastIndex));
        writeFinalBoardWithExitedPiece(writer,
states.get(states.size() - 1));
    }

    writer.println();
    writer.println("[Primary piece has reached the exit!]");
}
}

/**
 * Write final board state to file with primary piece shown exiting
 */
public static void writeFinalBoard(PrintWriter writer, Board board)
{
    int width = board.getWidth();
    int height = board.getHeight();

    Exit exitSide = board.getExitSide();
    Position exitPosition = board.getExitPosition();

    // Top exit if exists
    if (exitSide == Exit.TOP) {
        for (int j = 0; j < width; j++) {
            if (j == exitPosition.col) {
                writer.print("P"); // Show primary piece exiting at
top
            } else {
                writer.print(" ");
            }
        }
        writer.println();
    }

    // Board content
    for (int i = 0; i < height; i++) {
        // Left exit
        if (exitSide == Exit.LEFT && i == exitPosition.row) {
            writer.print("P"); // Show primary piece exiting at left
        }

        // Board cells
        for (int j = 0; j < width; j++) {
            writer.print(board.getGridAt(i, j));
        }

        // Right exit
        if (exitSide == Exit.RIGHT && i == exitPosition.row) {
            writer.print("P"); // Show primary piece exiting at
right
        }
    }
}

```



```

        writer.println();
    }

    // Bottom exit if exists
    if (exitSide == Exit.BOTTOM) {
        for (int j = 0; j < width; j++) {
            if (j == exitPosition.col) {
                writer.print("P"); // Show primary piece exiting at
bottom
            } else {
                writer.print(" ");
            }
        }
        writer.println();
    }
}

/**
 * Format move sequence for display
 */
private static String formatMoveSequence(List<Move> moves) {
    StringBuilder sb = new StringBuilder();
    int count = 0;

    for (Move move : moves) {
        sb.append(move.toString()).append(" ");
        count++;

        // Add newline every 16 moves for readability
        if (count % 16 == 0) {
            sb.append("\n");
        }
    }

    sb.append("(").append(moves.size()).append(" moves)");
    return sb.toString();
}

/**
 * Write board to a file
 */
private static void writeBoard(PrintWriter writer, Board board) {
    // Write board without colors (plain text for file)
    int width = board.getWidth();
    int height = board.getHeight();

    // Top exit if exists
    Exit exitSide = board.getExitSide();
    Position exitPosition = board.getExitPosition();

    if (exitSide == Exit.TOP) {
        for (int j = 0; j < width; j++) {

```

```

        if (j == exitPosition.col) {
            writer.print("K");
        } else {
            writer.print(" ");
        }
    }
    writer.println();
}

// Board content
for (int i = 0; i < height; i++) {
    // Left exit
    if (exitSide == Exit.LEFT && i == exitPosition.row) {
        writer.print("K");
    }

    // Board cells
    for (int j = 0; j < width; j++) {
        writer.print(board.getGridAt(i, j));
    }

    // Right exit
    if (exitSide == Exit.RIGHT && i == exitPosition.row) {
        writer.print("K");
    }

    writer.println();
}

// Bottom exit if exists
if (exitSide == Exit.BOTTOM) {
    for (int j = 0; j < width; j++) {
        if (j == exitPosition.col) {
            writer.print("K");
        } else {
            writer.print(" ");
        }
    }
    writer.println();
}
}

/**
 * Get yes/no choice from user
 */
private static String getYesNoChoice(Scanner scanner) {
    String choice = scanner.nextLine().trim();

    while (!choice.equalsIgnoreCase("y") &&
!choice.equalsIgnoreCase("n")) {
        System.out.print(RED + "Invalid input! Please enter 'y' or
'n': " + RESET);
        choice = scanner.nextLine().trim();
    }
}

```

```

        }

        return choice;
    }

    /**
     * Get algorithm choice from user
     */
    private static int getAlgorithmChoice(Scanner scanner) {
        String input = scanner.nextLine().trim();

        while (true) {
            try {
                int choice = Integer.parseInt(input);
                if (choice >= 1 && choice <= 3) {
                    return choice;
                } else {
                    System.out.print(RED + "Invalid choice! Please enter
a number between 1 and 3: " + RESET);
                    input = scanner.nextLine().trim();
                }
            } catch (NumberFormatException e) {
                System.out.print(RED + "Invalid input! Please enter a
number (1-3): " + RESET);
                input = scanner.nextLine().trim();
            }
        }
    }

    /**
     * Get heuristic choice from user
     */
    private static int getHeuristicChoice(Scanner scanner) {
        String input = scanner.nextLine().trim();

        while (true) {
            try {
                int choice = Integer.parseInt(input);
                if (choice >= 1 && choice <= 3) {
                    return choice;
                } else {
                    System.out.print(RED + "Invalid choice! Please enter
a number between 1 and 3: " + RESET);
                    input = scanner.nextLine().trim();
                }
            } catch (NumberFormatException e) {
                System.out.print(RED + "Invalid input! Please enter a
number (1-3): " + RESET);
                input = scanner.nextLine().trim();
            }
        }
    }
}

```

3.2.6 *Move.java*

```
package cli;

public class Move {
    private Piece piece;
    private String direction;

    public Move(Piece piece, String direction) {
        this.piece = piece;
        this.direction = direction;
    }

    public Piece getPiece() { return piece; }
    public String getDirection() { return direction; }

    /**
     * Returns the direction as an integer value
     * @return 1 for right/down, -1 for left/up
     */
    public int getDirectionValue() {
        if (direction.equals("right") || direction.equals("down")) {
            return 1;
        } else {
            return -1;
        }
    }

    @Override
    public String toString() {
        // Convert the direction to uppercase
        return direction.toUpperCase();
    }
}
```

3.2.7 *Orientation.java*

```
package cli;

public enum Orientation {
    HORIZONTAL,
    VERTICAL
}
```

3.2.8 Piece.java

```
package cli;

public class Piece {
    private char id;
    private java.util.List<Position> positions;
    private Orientation orientation;

    public Piece(char id, java.util.List<Position> positions) {
        this.id = id;
        this.positions = new java.util.ArrayList<>(positions);

        // Validate piece shape before determining orientation
        validatePieceShape();

        determineOrientation();
        sortPositions();
    }

    /**
     * Validates that the piece has a proper elongated form (1×N or N×1)
     * @throws IllegalArgumentException if the piece shape is invalid
     */
    private void validatePieceShape() {
        if (positions == null || positions.isEmpty()) {
            throw new IllegalArgumentException("Piece '" + id + "' has no positions");
        }

        if (positions.size() < 2) {
            throw new IllegalArgumentException("Piece '" + id + "' must occupy at least 2 cells, but only has " + positions.size());
        }

        // Get the bounding box
        int minRow = positions.stream().mapToInt(p -> p.row).min().orElse(0);
        int maxRow = positions.stream().mapToInt(p -> p.row).max().orElse(0);
        int minCol = positions.stream().mapToInt(p -> p.col).min().orElse(0);
        int maxCol = positions.stream().mapToInt(p -> p.col).max().orElse(0);

        int rowSpan = maxRow - minRow + 1;
        int colSpan = maxCol - minCol + 1;

        // Check if piece is linear (either horizontal or vertical)
    }
}
```

```

        boolean isHorizontal = (rowSpan == 1 && colSpan > 1);
        boolean isVertical = (colSpan == 1 && rowSpan > 1);

        // Special case: single cell allowed only for primary piece (P)
        if (positions.size() == 1 && id == 'P') {
            return; // Allow single cell for primary piece (for testing)
        }

        if (!isHorizontal && !isVertical) {
            throw new IllegalArgumentException("Piece '" + id + "' is
not linear. It spans " +
                rowSpan + " rows and " + colSpan + " columns. Pieces
must be either 1xN or Nx1.");
        }

        // Verify that all positions are contiguous
        if (isHorizontal) {
            // For horizontal pieces, all positions should have the same
row
            int expectedRow = positions.get(0).row;
            for (Position pos : positions) {
                if (pos.row != expectedRow) {
                    throw new IllegalArgumentException("Piece '" + id +
"' is not properly aligned horizontally. " +
                        "Position (" + pos.row + ", " + pos.col + ") is
not in row " + expectedRow);
                }
            }

            // Check for gaps in columns
            if (positions.size() != colSpan) {
                throw new IllegalArgumentException("Piece '" + id + "'
has gaps. Expected " +
                    colSpan + " consecutive cells but found " +
positions.size());
            }

            // Verify no gaps between min and max columns
            java.util.Set<Integer> columns = new java.util.HashSet<>();
            for (Position pos : positions) {
                columns.add(pos.col);
            }
            for (int col = minCol; col <= maxCol; col++) {
                if (!columns.contains(col)) {
                    throw new IllegalArgumentException("Piece '" + id +
"' has a gap at column " + col);
                }
            }
        } else { // isVertical
            // For vertical pieces, all positions should have the same
column
            int expectedCol = positions.get(0).col;

```

```

        for (Position pos : positions) {
            if (pos.col != expectedCol) {
                throw new IllegalArgumentException("Piece '" + id +
"' is not properly aligned vertically. " +
                "Position (" + pos.row + ", " + pos.col + ") is
not in column " + expectedCol);
            }
        }

        // Check for gaps in rows
        if (positions.size() != rowSpan) {
            throw new IllegalArgumentException("Piece '" + id + "'
has gaps. Expected " +
            rowSpan + " consecutive cells but found " +
positions.size());
        }

        // Verify no gaps between min and max rows
        java.util.Set<Integer> rows = new java.util.HashSet<>();
        for (Position pos : positions) {
            rows.add(pos.row);
        }
        for (int row = minRow; row <= maxRow; row++) {
            if (!rows.contains(row)) {
                throw new IllegalArgumentException("Piece '" + id +
"' has a gap at row " + row);
            }
        }
    }

    private void determineOrientation() {
        if (positions.size() < 2) {
            this.orientation = Orientation.HORIZONTAL;
            return;
        }

        // Since validation has already been done, we know the piece is
linear
        // Just check if it's horizontal or vertical based on the span
        int minRow = positions.stream().mapToInt(p ->
p.row).min().orElse(0);
        int maxRow = positions.stream().mapToInt(p ->
p.row).max().orElse(0);
        int minCol = positions.stream().mapToInt(p ->
p.col).min().orElse(0);
        int maxCol = positions.stream().mapToInt(p ->
p.col).max().orElse(0);

        int rowSpan = maxRow - minRow + 1;
        int colSpan = maxCol - minCol + 1;

        // Single cell piece defaults to horizontal (shouldn't happen in

```

```

normal cases)
    if (rowSpan == 1 && colSpan == 1) {
        this.orientation = Orientation.HORIZONTAL;
    } else if (rowSpan == 1) {
        this.orientation = Orientation.HORIZONTAL;
    } else {
        this.orientation = Orientation.VERTICAL;
    }
}

/**
 * Copy constructor
 */
public Piece(Piece other) {
    this.id = other.id;
    this.positions = new java.util.ArrayList<>();
    for (Position pos : other.positions) {
        this.positions.add(new Position(pos));
    }
    this.orientation = other.orientation;
}

int getLength() {
    return positions.size();
}

/**
 * Move the piece in the specified direction
 * @param direction 1 for forward (right/down), -1 for backward
(left/up)
 */
public void move(int direction) {
    for (Position pos : positions) {
        if (orientation == Orientation.HORIZONTAL) {
            pos.col += direction;
        } else {
            pos.row += direction;
        }
    }
    sortPositions();
}

/**
 * Sort positions for consistent ordering
 */
private void sortPositions() {
    positions.sort((p1, p2) -> {
        if (orientation == Orientation.HORIZONTAL) {
            if (p1.row != p2.row) {
                return Integer.compare(p1.row, p2.row);
            }
            return Integer.compare(p1.col, p2.col);
        } else {

```



```

        if (p1.col != p2.col) {
            return Integer.compare(p1.col, p2.col);
        }
        return Integer.compare(p1.row, p2.row);
    }
}

});

// Getters
public char getId() { return id; }
public java.util.List<Position> getPositions() { return positions; }
public Orientation getOrientation() { return orientation; }
public int getSize() { return positions.size(); }

// Add these methods to the Piece class

public int getLeftmostCol() {
    if (orientation == Orientation.HORIZONTAL) {
        // First position in sorted list has smallest column
        return positions.get(0).col;
    }
    // Vertical pieces have same column for all positions
    return positions.get(0).col;
}

public int getRightmostCol() {
    if (orientation == Orientation.HORIZONTAL) {
        // Last position in sorted list has largest column
        return positions.get(positions.size() - 1).col;
    }
    // Vertical pieces have same column for all positions
    return positions.get(0).col;
}

public int getTopmostRow() {
    if (orientation == Orientation.VERTICAL) {
        // First position in sorted list has smallest row
        return positions.get(0).row;
    }
    // Horizontal pieces have same row for all positions
    return positions.get(0).row;
}

public int getBottommostRow() {
    if (orientation == Orientation.VERTICAL) {
        // Last position in sorted list has largest row
        return positions.get(positions.size() - 1).row;
    }
    // Horizontal pieces have same row for all positions
    return positions.get(0).row;
}

public Position getFirstPosition() {

```

```

        return positions.get(0);
    }

    @Override
    public String toString() {
        return "Piece{id=" + id + ", orientation=" + orientation + ",
positions=" + positions + "}";
    }
}

```

3.2.9 Position.java

```

package cli;

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

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

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

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Position position = (Position) obj;
        return row == position.row && col == position.col;
    }

    @Override
    public int hashCode() {
        return row * 100 + col;
    }

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

```

3.2.10 Solution.java

```
package cli;

import java.util.*;

/**
 * Solution class with improved visualization showing primary piece next
 * to exit
 */
public class Solution {
    private List<Move> moves;
    private List<Board> states;
    private int statesExamined;

    public Solution(List<Move> moves, List<Board> states, int
statesExamined) {
        this.moves = moves;
        this.states = states;
        this.statesExamined = statesExamined;
    }

    public List<Move> getMoves() { return moves; }
    public List<Board> getStates() { return states; }
    public int getStatesExamined() { return statesExamined; }

    /**
     * Display the solution step by step with primary piece shown next
     * to exit
     */
    public void displaySolution() {
        System.out.println("\nPapan Awal");
        states.get(0).display();

        // Display all moves except the last one
        for (int i = 0; i < moves.size() - 1; i++) {
            Move move = moves.get(i);
            System.out.println("\nGerakan " + (i + 1) + ": " +
move.toString());

            Board nextState = states.get(i + 1);

            // If it's a compound move, highlight the moving piece
            if (move instanceof CompoundMove) {
                nextState.displayWithMove(move);
            } else {
                nextState.display();
            }
        }
    }
}
```

```

        // Display final move with P shown outside next to exit
        if (moves.size() > 0) {
            int lastIndex = moves.size() - 1;
            Move lastMove = moves.get(lastIndex);
            System.out.println("\nGerakan " + (lastIndex + 1) + ": " +
lastMove.toString());

            // Get final state
            Board finalState = states.get(states.size() - 1);

            // Use the visualization showing P next to K
            finalState.displayWithExitedPiece();
        }

        System.out.println("\n[Primary piece has reached the exit!]");

        // Display the complete move sequence
        System.out.println("\nMove sequence:");
        displayMoveSequence();
    }

    /**
     * Display the move sequence in a compact format
     */
    private void displayMoveSequence() {
        StringBuilder sequence = new StringBuilder();
        int count = 0;

        for (Move move : moves) {
            sequence.append(move.toString()).append(" ");
            count++;

            // Add newline every 16 moves for readability
            if (count % 16 == 0) {
                sequence.append("\n");
            }
        }

        sequence.append("(").append(moves.size()).append(" moves)");
        System.out.println(sequence.toString());
    }
}

```

3.2.11 Solver.java

```

package cli;

import java.util.ArrayList;
import java.util.Comparator;

```

```

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

/**
 * Solver class implementing different pathfinding algorithms for the
 * Rush Hour puzzle
 * with support for compound moves (multi-cell movements in one
 * direction count as one move)
 * and tracking of examined node count
 */
public class Solver {

    // Added to track nodes examined even when no solution is found
    private int lastNodesExamined = 0;

    /**
     * Get the number of states examined in the last solving attempt
     */
    public int getLastNodesExamined() {
        return lastNodesExamined;
    }

    private static class Result {
        boolean found;
        Node node;
        int nextThreshold;

        Result(boolean found, Node node, int nextThreshold) {
            this.found = found;
            this.node = node;
            this.nextThreshold = nextThreshold;
        }
    }

    /**
     * UCS Implementation with compound moves
     */
    public Solution solveUCS(Board initialBoard, boolean isCompound) {
        System.out.println("Searching for solution using UCS");
        PriorityQueue<Node> frontier = new
        PriorityQueue<>(Comparator.comparingInt(n -> n.cost));
        Set<String> visited = new HashSet<>();
        lastNodesExamined = 0; // Reset counter

        Node startNode = new Node(initialBoard, null, null, 0);
        frontier.add(startNode);

        while (!frontier.isEmpty()) {
            Node current = frontier.poll();

```

```

        String stateString = current.board.getStateString();

        if (visited.contains(stateString)) {
            continue;
        }

        visited.add(stateString);
        lastNodesExamined++; // Increment counter

        // Check if solved
        if (current.board.isSolved()) {
            return reconstructSolution(current, lastNodesExamined);
        }

        // Generate compound moves (multi-cell movements)
        List<CompoundMove> compoundMoves =
generateCompoundMoves(current.board, isCompound);

        for (CompoundMove move : compoundMoves) {
            Board newBoard = makeCompoundMove(current.board, move);
            String newStateString = newBoard.getStateString();

            if (!visited.contains(newStateString)) {
                int newCost = current.cost + 1; // Each compound
move costs 1
                Node newNode = new Node(newBoard, move, current,
newCost);
                frontier.add(newNode);
            }
        }

        return null; // No solution found, but lastNodesExamined has
been updated
    }

    /**
     * A* Search Implementation with compound moves
     */
    public Solution solveAStar(Board initialBoard, String heuristic,
boolean isCompound) {
        System.out.println("Searching for solution using A* with
heuristic: " + heuristic);
        PriorityQueue<Node> frontier = new
PriorityQueue<>(Comparator.comparingInt(n -> n.f));
        Set<String> visited = new HashSet<>();
        Map<String, Node> nodeMap = new HashMap<>();
        lastNodesExamined = 0; // Reset counter

        int h = calculateHeuristic(initialBoard, heuristic);
        Node startNode = new Node(initialBoard, null, null, 0, h, h);
        frontier.add(startNode);
        nodeMap.put(initialBoard.getStateString(), startNode);

```

```

        while (!frontier.isEmpty()) {
            Node current = frontier.poll();
            String stateString = current.board.getStateString();

            if (visited.contains(stateString)) {
                continue;
            }

            visited.add(stateString);
            lastNodesExamined++; // Increment counter

            // Check if solved
            if (current.board.isSolved()) {
                return reconstructSolution(current, lastNodesExamined);
            }

            // Generate compound moves (multi-cell movements)
            List<CompoundMove> compoundMoves =
generateCompoundMoves(current.board, isCompound);

            for (CompoundMove move : compoundMoves) {
                Board newBoard = makeCompoundMove(current.board, move);
                String newStateString = newBoard.getStateString();

                if (!visited.contains(newStateString)) {
                    int newG = current.cost + 1; // Each compound move
costs 1
                    int newH = calculateHeuristic(newBoard, heuristic);
                    int newF = newG + newH;

                    if (!nodeMap.containsKey(newStateString) ||
nodeMap.get(newStateString).f > newF) {
                        Node newNode = new Node(newBoard, move, current,
newG, newH, newF);
                        frontier.add(newNode);
                        nodeMap.put(newStateString, newNode);
                    }
                }
            }

            return null; // No solution found, but lastNodesExamined has
been updated
        }

/**
 * Greedy Best First Search Implementation with compound moves
 */
public Solution solveGreedy(Board initialBoard, String heuristic,
boolean isCompound) {
    System.out.println("Searching for solution using Greedy Best
First Search with heuristic: " + heuristic);

```

```

        PriorityQueue<Node> frontier = new
PriorityQueue<>(Comparator.comparingInt(n -> n.h));
        Set<String> visited = new HashSet<>();
        lastNodesExamined = 0; // Reset counter

        int h = calculateHeuristic(initialBoard, heuristic);
        Node startNode = new Node(initialBoard, null, null, 0, h, h);
        frontier.add(startNode);

        while (!frontier.isEmpty()) {
            Node current = frontier.poll();
            String stateString = current.board.getStateString();

            if (visited.contains(stateString)) {
                continue;
            }

            visited.add(stateString);
            lastNodesExamined++; // Increment counter

            // Check if solved
            if (current.board.isSolved()) {
                return reconstructSolution(current, lastNodesExamined);
            }

            // Generate compound moves (multi-cell movements)
            List<CompoundMove> compoundMoves =
generateCompoundMoves(current.board, isCompound);

            for (CompoundMove move : compoundMoves) {
                Board newBoard = makeCompoundMove(current.board, move);
                String newStateString = newBoard.getStateString();

                if (!visited.contains(newStateString)) {
                    int newH = calculateHeuristic(newBoard, heuristic);
                    Node newNode = new Node(newBoard, move, current,
current.cost + 1, newH, newH);
                    frontier.add(newNode);
                }
            }

            return null; // No solution found, but lastNodesExamined has
been updated
        }

        /**
         * Dijkstra's algorithm implementation - similar to UCS but with
different node mapping
         */
        public Solution solveDijkstra(Board initialBoard, boolean
isCompound) {
            System.out.println("Searching for solution using Dijkstra's

```



```

algorithm");
    PriorityQueue<Node> frontier = new
PriorityQueue<>(Comparator.comparingInt(n -> n.cost));
    Set<String> visited = new HashSet<>();
    Map<String, Integer> costSoFar = new HashMap<>();
    lastNodesExamined = 0; // Reset counter

    Node startNode = new Node(initialBoard, null, null, 0);
    frontier.add(startNode);
    costSoFar.put(initialBoard.getStateString(), 0);

    while (!frontier.isEmpty()) {
        Node current = frontier.poll();
        String stateString = current.board.getStateString();

        if (visited.contains(stateString)) {
            continue;
        }

        visited.add(stateString);
        lastNodesExamined++; // Increment counter

        // Check if solved
        if (current.board.isSolved()) {
            return reconstructSolution(current, lastNodesExamined);
        }

        // Generate compound moves
        List<CompoundMove> compoundMoves =
generateCompoundMoves(current.board, isCompound);

        for (CompoundMove move : compoundMoves) {
            Board newBoard = makeCompoundMove(current.board, move);
            String newStateString = newBoard.getStateString();

            // For Dijkstra, treat all moves as cost 1
            int newCost = current.cost + 1;

            if (!costSoFar.containsKey(newStateString) || newCost <
costSoFar.get(newStateString)) {
                costSoFar.put(newStateString, newCost);
                Node newNode = new Node(newBoard, move, current,
newCost);
                frontier.add(newNode);
            }
        }

        return null; // No solution found, but lastNodesExamined has
been updated
    }

    public Solution solveBeam(Board initialBoard, String heuristic,

```

```

boolean isCompound) {
    System.out.println("Searching for solution using Beam Search
with heuristic: " + heuristic);
    int beamWidth = 50;
    List<Node> frontier = new ArrayList<>();
    Set<String> visited = new HashSet<>();
    lastNodesExamined = 0;

    int h = calculateHeuristic(initialBoard, heuristic);
    Node startNode = new Node(initialBoard, null, null, 0, h, h);
    frontier.add(startNode);

    while (!frontier.isEmpty()) {
        List<Node> nextLevel = new ArrayList<>();

        for (Node current : frontier) {
            String stateString = current.board.getStateString();

            if (visited.contains(stateString)) {
                continue;
            }

            visited.add(stateString);
            lastNodesExamined++;

            // Goal check
            if (current.board.isSolved()) {
                return reconstructSolution(current,
lastNodesExamined);
            }

            // Generate children
            List<CompoundMove> compoundMoves =
generateCompoundMoves(current.board, isCompound);

            for (CompoundMove move : compoundMoves) {
                Board newBoard = makeCompoundMove(current.board,
move);

                String newStateString = newBoard.getStateString();

                if (!visited.contains(newStateString)) {
                    int newH = calculateHeuristic(newBoard,
heuristic);

                    Node newNode = new Node(newBoard, move, current,
current.cost + 1, newH, newH);
                    nextLevel.add(newNode);
                }
            }

            // Sort all next level nodes by heuristic value and select
top beamWidth
            nextLevel.sort(Comparator.comparingInt(n -> n.h));

```

```

        frontier = nextLevel.subList(0, Math.min(beamWidth,
nextLevel.size()));
    }

    return null; // No solution found
}

public Solution solveIDAStar(Board initialBoard, String heuristic,
boolean isCompound) {
    // IDA* Search Implementation with compound moves
    System.out.println("Searchinig for solution using IDA* with
heuristic: " + heuristic);
    lastNodesExamined = 0;

    int h = calculateHeuristic(initialBoard, heuristic);
    Node root = new Node(initialBoard, null, null, 0, h, h);
    int threshold = root.f;

    while (true) {
        Set<String> visited = new HashSet<>();
        Result result = dfsIDA(root, heuristic, isCompound,
threshold, visited);
        if (result.found) {
            return reconstructSolution(result.node,
lastNodesExamined);
        }
        if (result.nextThreshold == Integer.MAX_VALUE) {
            return null; // No solution
        }
        threshold = result.nextThreshold;
    }
}

private Result dfsIDA(Node current, String heuristic, boolean
isCompound, int threshold, Set<String> visited) {
    lastNodesExamined++;
    String stateString = current.board.getStateString();
    if (visited.contains(stateString)) return new Result(false,
null, Integer.MAX_VALUE);
    visited.add(stateString);

    int f = current.cost + current.h;
    if (f > threshold) return new Result(false, null, f);
    if (current.board.isSolved()) return new Result(true, current,
f);

    int minThreshold = Integer.MAX_VALUE;

    for (CompoundMove move : generateCompoundMoves(current.board,
isCompound)) {
        Board newBoard = makeCompoundMove(current.board, move);
        String newStateString = newBoard.getStateString();
        if (visited.contains(newStateString)) continue;
    }
}

```

```

        int newH = calculateHeuristic(newBoard, heuristic);
        Node child = new Node(newBoard, move, current, current.cost
+ 1, newH, current.cost + 1 + newH);
        Result result = dfsIDA(child, heuristic, isCompound,
threshold, visited);

        if (result.found) return result;
        minThreshold = Math.min(minThreshold, result.nextThreshold);
    }

    visited.remove(stateString);
    return new Result(false, null, minThreshold);
}

private List<CompoundMove> generateCompoundMoves(Board board) {
    return generateCompoundMoves(board, true); // default isCompound
= true
}

/**
 * Generate all possible compound moves (multi-cell movements) for a
board
 */
private List<CompoundMove> generateCompoundMoves(Board board,
boolean isCompound) {
    List<CompoundMove> compoundMoves = new ArrayList<>();

    if (isCompound) {
        for (Piece piece : board.getPieces()) {
            // Try all possible moves for each piece
            if (piece.getOrientation() == Orientation.HORIZONTAL) {
                // Try moving right
                int maxRight = findMaximumDistance(board, piece,
"right");
                if (maxRight > 0) {
                    compoundMoves.add(new CompoundMove(piece,
"right", maxRight));
                }

                // Try moving left
                int maxLeft = findMaximumDistance(board, piece,
"left");
                if (maxLeft > 0) {
                    compoundMoves.add(new CompoundMove(piece,
"left", maxLeft));
                }
            } else {
                // Try moving down
                int maxDown = findMaximumDistance(board, piece,
"down");
                if (maxDown > 0) {

```

```

        compoundMoves.add(new CompoundMove(piece,
"down", maxDown));
    }

    // Try moving up
    int maxUp = findMaximumDistance(board, piece, "up");
    if (maxUp > 0) {
        compoundMoves.add(new CompoundMove(piece, "up",
maxUp));
    }
}
}
else {
    for (Piece piece : board.getPieces()) {
        // Try all possible moves for each piece
        if (piece.getOrientation() == Orientation.HORIZONTAL) {
            // Try moving right
            int maxRight = findMaximumDistance(board, piece,
"right");
            if (maxRight > 0) {
                compoundMoves.add(new CompoundMove(piece,
"right", 1));
            }

            // Try moving left
            int maxLeft = findMaximumDistance(board, piece,
"left");
            if (maxLeft > 0) {
                compoundMoves.add(new CompoundMove(piece,
"left", 1));
            }
        } else {
            // Try moving down
            int maxDown = findMaximumDistance(board, piece,
"down");
            if (maxDown > 0) {
                compoundMoves.add(new CompoundMove(piece,
"down", 1));
            }

            // Try moving up
            int maxUp = findMaximumDistance(board, piece, "up");
            if (maxUp > 0) {
                compoundMoves.add(new CompoundMove(piece, "up",
1));
            }
        }
    }
}

return compoundMoves;
}

```

```

/**
 * Find the maximum distance a piece can move in a given direction
 * Stop if primary piece reaches exit
 */
private int findMaximumDistance(Board board, Piece piece, String
direction) {
    int distance = 0;
    Board currentBoard = board;
    boolean isPrimaryPiece = piece.getId() == 'P';

    // Keep moving until we can't move anymore
    while (true) {
        // Check if we can move one more step
        List<Move> possibleMoves = currentBoard.getPossibleMoves();
        boolean canMove = false;

        for (Move move : possibleMoves) {
            if (move.getPiece().getId() == piece.getId() &&
move.getDirection().equals(direction)) {
                canMove = true;
                currentBoard = currentBoard.makeMove(move);
                distance++;

                // Check if primary piece has reached exit
                if (isPrimaryPiece && currentBoard.isSolved()) {
                    return distance; // Stop at exit for primary
piece
                }

                break;
            }
        }

        if (!canMove) {
            break;
        }
    }

    return distance;
}

/**
 * Apply a compound move to a board with proper exit detection
 */
private Board makeCompoundMove(Board board, CompoundMove move) {
    Board currentBoard = board;
    String direction = move.getDirection();
    char pieceId = move.getPiece().getId();
    boolean isPrimaryPiece = pieceId == 'P';

    // Apply the move step by step
    for (int i = 0; i < move.getDistance(); i++) {

```

```

        // Find the piece in the current board
        Piece targetPiece = null;
        for (Piece p : currentBoard.getPieces()) {
            if (p.getId() == pieceId) {
                targetPiece = p;
                break;
            }
        }

        if (targetPiece == null) {
            break; // Piece not found (might happen if primary piece
exits)
        }

        // Create a simple move and apply it
        Move simpleMove = new Move(targetPiece, direction);
        Board nextBoard = currentBoard.makeMove(simpleMove);

        // Check if the primary piece has reached the exit
        if (isPrimaryPiece && nextBoard.isSolved()) {
            return nextBoard; // Stop movement once primary piece
reaches exit
        }

        currentBoard = nextBoard;
    }

    return currentBoard;
}

/**
 * Reconstruct solution from final node
 */
private Solution reconstructSolution(Node goalNode, int
statesExamined) {
    List<Move> moves = new ArrayList<>();
    List<Board> states = new ArrayList<>();

    Node current = goalNode;

    // Trace back from goal to start
    while (current != null) {
        if (current.board != null) {
            states.add(0, current.board);
        }
        if (current.move != null) {
            moves.add(0, current.move);
        }
        current = current.parent;
    }

    return new Solution(moves, states, statesExamined);
}

```

```

/**
 * Calculate heuristic value
 */
private int calculateHeuristic(Board board, String heuristic) {
    switch (heuristic.toLowerCase()) {
        case "manhattan distance":
        case "manhattan":
            return calculateManhattanDistance(board);
        case "direct distance":
        case "direct":
            return calculateDirectDistance(board);
        case "blocking count":
        case "blocking":
            return calculateBlockingCount(board);
        case "clearing moves":
        case "clearing":
            return calculateClearingMoves(board);
        default:
            return calculateManhattanDistance(board);
    }
}

/**
 * Manhattan distance heuristic
 */
private int calculateManhattanDistance(Board board) {
    Piece primaryPiece = board.getPrimaryPiece();
    if (primaryPiece == null) return Integer.MAX_VALUE;

    cli.Position exitPos = board.getExitPosition();
    if (exitPos == null) return Integer.MAX_VALUE;

    // Get the position of primary piece that's closest to exit
    int minDistance = Integer.MAX_VALUE;
    for (cli.Position pos : primaryPiece.getPositions()) {
        int distance = Math.abs(pos.row - exitPos.row) +
Math.abs(pos.col - exitPos.col);
        minDistance = Math.min(minDistance, distance);
    }

    return minDistance;
}

/**
 * Direct distance to exit (considering orientation)
 */
private int calculateDirectDistance(Board board) {
    Piece primaryPiece = board.getPrimaryPiece();
    if (primaryPiece == null) return Integer.MAX_VALUE;

    cli.Position exitPos = board.getExitPosition();
    if (exitPos == null) return Integer.MAX_VALUE;

```



```

        // We need to account for the orientation of the piece and the
        exit side
        Exit exitSide = board.getExitSide();

        if (primaryPiece.getOrientation() == Orientation.HORIZONTAL) {
            // For horizontal piece
            if (exitSide == Exit.RIGHT) {
                // Need to get rightmost position of piece
                cli.Position rightmost =
primaryPiece.getPositions().stream()
                    .max(Comparator.comparingInt(p -> p.col))
                    .orElse(primaryPiece.getPositions().get(0));
                return board.getWidth() - rightmost.col - 1; // Distance
to right edge
            } else if (exitSide == Exit.LEFT) {
                // Need to get leftmost position of piece
                cli.Position leftmost =
primaryPiece.getPositions().stream()
                    .min(Comparator.comparingInt(p -> p.col))
                    .orElse(primaryPiece.getPositions().get(0));
                return leftmost.col; // Distance to left edge
            } else {
                // Exit is not aligned with piece orientation
                return Integer.MAX_VALUE;
            }
        } else {
            // For vertical piece
            if (exitSide == Exit.BOTTOM) {
                // Need to get bottommost position of piece
                cli.Position bottommost =
primaryPiece.getPositions().stream()
                    .max(Comparator.comparingInt(p -> p.row))
                    .orElse(primaryPiece.getPositions().get(0));
                return board.getHeight() - bottommost.row - 1; //
Distance to bottom edge
            } else if (exitSide == Exit.TOP) {
                // Need to get topmost position of piece
                cli.Position topmost =
primaryPiece.getPositions().stream()
                    .min(Comparator.comparingInt(p -> p.row))
                    .orElse(primaryPiece.getPositions().get(0));
                return topmost.row; // Distance to top edge
            } else {
                // Exit is not aligned with piece orientation
                return Integer.MAX_VALUE;
            }
        }
    }

    /**
     * Count pieces blocking the path to exit
     */

```

```

private int calculateBlockingCount(Board board) {
    Piece primaryPiece = board.getPrimaryPiece();
    if (primaryPiece == null) return Integer.MAX_VALUE;

    cli.Position exitPos = board.getExitPosition();
    if (exitPos == null) return Integer.MAX_VALUE;

    int blockingCount = 0;
    char[][] grid = board.getGrid();

    // Get the path from primary piece to exit
    if (primaryPiece.getOrientation() == Orientation.HORIZONTAL) {
        int row = primaryPiece.getPositions().get(0).row;

        // Find the range to check based on exit position
        int startCol, endCol;
        if (board.getExitSide() == Exit.RIGHT) {
            // Rightmost position of primary piece
            cli.Position rightmost =
primaryPiece.getPositions().stream()
                .max(Comparator.comparingInt(p -> p.col))
                .orElse(primaryPiece.getPositions().get(0));

            startCol = rightmost.col + 1;
            endCol = board.getWidth();
        } else {
            // Leftmost position of primary piece
            cli.Position leftmost =
primaryPiece.getPositions().stream()
                .min(Comparator.comparingInt(p -> p.col))
                .orElse(primaryPiece.getPositions().get(0));

            startCol = 0;
            endCol = leftmost.col;
        }

        // Check for blocking pieces
        Set<Character> blockingPieces = new HashSet<>();
        for (int col = startCol; col < endCol; col++) {
            if (row >= 0 && row < grid.length && col >= 0 && col <
grid[row].length) {
                char cell = grid[row][col];
                if (cell != '.' && cell != 'P' &&
!blockingPieces.contains(cell)) {
                    blockingPieces.add(cell);
                    blockingCount++;
                }
            }
        }
    } else {
        // Vertical orientation
        int col = primaryPiece.getPositions().get(0).col;

```

```

        // Find the range to check based on exit position
        int startRow, endRow;
        if (board.getExitSide() == Exit.BOTTOM) {
            // Bottommost position of primary piece
            cli.Position bottommost =
primaryPiece.getPositions().stream()
                .max(Comparator.comparingInt(p -> p.row))
                .orElse(primaryPiece.getPositions().get(0));

            startRow = bottommost.row + 1;
            endRow = board.getHeight();
        } else {
            // Topmost position of primary piece
            cli.Position topmost =
primaryPiece.getPositions().stream()
                .min(Comparator.comparingInt(p -> p.row))
                .orElse(primaryPiece.getPositions().get(0));

            startRow = 0;
            endRow = topmost.row;
        }

        // Check for blocking pieces
        Set<Character> blockingPieces = new HashSet<>();
        for (int row = startRow; row < endRow; row++) {
            if (row >= 0 && row < grid.length && col >= 0 && col <
grid[row].length) {
                char cell = grid[row][col];
                if (cell != '.' && cell != 'P' &&
!blockingPieces.contains(cell)) {
                    blockingPieces.add(cell);
                    blockingCount++;
                }
            }
        }

        return blockingCount;
    }

    private int calculateClearingMoves(Board board) {
        Piece primary = board.getPrimaryPiece();
        if (primary == null) return Integer.MAX_VALUE;

        // 1. Calculate direct exit distance
        int directDistance = calculateDirectDistance(board);

        // 2. Identify critical path blockers
        Set<Piece> blockers = getCriticalBlockers(board);

        // 3. Calculate minimal clearing moves for each blocker
        int totalMoves = 0;
        for (Piece blocker : blockers) {

```

```

        int moves = calculateBlockerMoves(blocker, primary, board);
        if (moves == Integer.MAX_VALUE) return Integer.MAX_VALUE;
        totalMoves += moves;
    }

    return directDistance + totalMoves;
}

private Set<Piece> getCriticalBlockers(Board board) {
    Set<Piece> blockers = new HashSet<>();
    Piece primary = board.getPrimaryPiece();
    Exit exit = board.getExitSide();

    if (primary.getOrientation() == Orientation.HORIZONTAL) {
        int row = primary.getFirstPosition().row;
        int startCol = exit == Exit.RIGHT ?
            primary.getRightmostCol() + 1 : 0;
        int endCol = exit == Exit.RIGHT ?
            board.getWidth() : primary.getLeftmostCol();

        // Check primary's row and adjacent vertical blockers
        for (int col = startCol; col < endCol; col++) {
            // Direct blockers in path
            if (board.getGrid()[row][col] != '.' &&
                board.getGrid()[row][col] != primary.getId()) {
                blockers.add(board.getPieceAt(row, col));
            }

            // Vertical blockers crossing the path
            for (Piece p : board.getPieces()) {
                if (p.getOrientation() == Orientation.VERTICAL &&
                    p.getLeftmostCol() == col &&
                    p.getTopmostRow() <= row &&
                    p.getBottommostRow() >= row) {
                    blockers.add(p);
                }
            }
        }
    } else { // Vertical primary
        int col = primary.getFirstPosition().col;
        int startRow = exit == Exit.BOTTOM ?
            primary.getBottommostRow() + 1 : 0;
        int endRow = exit == Exit.BOTTOM ?
            board.getHeight() : primary.getTopmostRow();

        for (int row = startRow; row < endRow; row++) {
            // Direct blockers in path
            if (board.getGrid()[row][col] != '.' &&
                board.getGrid()[row][col] != primary.getId()) {
                blockers.add(board.getPieceAt(row, col));
            }

            // Horizontal blockers crossing the path

```

```

        for (Piece p : board.getPieces()) {
            if (p.getOrientation() == Orientation.HORIZONTAL &&
                p.getTopmostRow() == row &&
                p.getLeftmostCol() <= col &&
                p.getRightmostCol() >= col) {
                blockers.add(p);
            }
        }
    }
    return blockers;
}

private int calculateBlockerMoves(Piece blocker, Piece primary,
Board board) {
    // Determine available space in both possible directions
    int forwardSpace = calculateClearSpace(blocker, true, board);
    int backwardSpace = calculateClearSpace(blocker, false, board);

    // Minimum moves needed to clear the path
    int requiredClearance = getRequiredClearance(blocker, primary,
board);

    int forwardMoves = (requiredClearance <= forwardSpace) ?
        (blocker.getLength() + requiredClearance) :
Integer.MAX_VALUE;
    int backwardMoves = (requiredClearance <= backwardSpace) ?
        (blocker.getLength() + requiredClearance) :
Integer.MAX_VALUE;

    return Math.min(forwardMoves, backwardMoves);
}

private int calculateClearSpace(Piece blocker, boolean moveForward,
Board board) {
    int space = 0;
    if (blocker.getOrientation() == Orientation.HORIZONTAL) {
        int checkCol = moveForward ?
            blocker.getRightmostCol() + 1 : blocker.getLeftmostCol()
- 1;
        while (moveForward ? checkCol < board.getWidth() : checkCol
>= 0) {
            boolean columnClear = true;
            for (int r = blocker.getTopmostRow(); r <=
blocker.getBottommostRow(); r++) {
                if (board.getGrid()[r][checkCol] != '.') {
                    columnClear = false;
                    break;
                }
            }
            if (!columnClear) break;
            space++;
            checkCol += moveForward ? 1 : -1;
        }
    }
}

```

```

        }
    } else { // Vertical
        int checkRow = moveForward ?
            blocker.getBottommostRow() + 1 : blocker.getTopmostRow()
- 1;
        while (moveForward ? checkRow < board.getHeight() : checkRow
>= 0) {
            boolean rowClear = true;
            for (int c = blocker.getLeftmostCol(); c <=
blocker.getRightmostCol(); c++) {
                if (board.getGrid()[checkRow][c] != '.') {
                    rowClear = false;
                    break;
                }
            }
            if (!rowClear) break;
            space++;
            checkRow += moveForward ? 1 : -1;
        }
    }
    return space;
}

private int getRequiredClearance(Piece blocker, Piece primary, Board
board) {
    // Calculate how far the blocker needs to move to clear the path
    if (primary.getOrientation() == Orientation.HORIZONTAL) {
        int blockerColSpan = blocker.getLeftmostCol() +
blocker.getLength() - 1;
        return Math.max(0, blockerColSpan -
primary.getRightmostCol() + 1);
    } else {
        int blockerRowSpan = blocker.getTopmostRow() +
blocker.getLength() - 1;
        return Math.max(0, blockerRowSpan -
primary.getBottommostRow() + 1);
    }
}

/**
 * Inner class representing a search node
 */
private static class Node {
    Board board;
    Move move;
    Node parent;
    int cost; // g(n) - cost from start
    int h; // h(n) - heuristic value
    int f; // f(n) = g(n) + h(n)

    Node(Board board, Move move, Node parent, int cost) {
        this.board = board;
        this.move = move;
    }
}

```

```

        this.parent = parent;
        this.cost = cost;
        this.h = 0;
        this.f = cost;
    }

    Node(Board board, Move move, Node parent, int cost, int h, int
f) {
        this.board = board;
        this.move = move;
        this.parent = parent;
        this.cost = cost;
        this.h = h;
        this.f = f;
    }
}

```

3.2.12 AnimationController.java

```

package gui.controllers;

import cli.*;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.Slider;
import javafx.scene.paint.Color;
import javafx.util.Duration;
import java.util.*;

/**
 * Controller for smooth animation of Rush Hour puzzle states
 * Implements state-by-state animation with interpolation between states
 * and special handling for primary piece exiting the board
 */
public class AnimationController {
    private Canvas boardCanvas;
    private GraphicsContext gc;
    private Timeline animation;
    private List<Board> states;
    private List<Move> moves;
    private int currentStateIndex = 0;
    private int animationSteps = 10; // Number of interpolation steps
    between states
}

```

```

private int currentAnimationStep = 0;
private boolean isPlaying = false;
private Label moveLabel;
private Label statsLabel;
private Slider speedSlider;
private Button playButton;
private Button pauseButton;
private Button resetButton;
private int width;
private int height;

// Constants for visualization
private static final int CELL_SIZE = 60;
private static final int PADDING = 40;
private Map<Character, Color> pieceColors = new HashMap<>();

/**
 * Initialize the controller
 */
public AnimationController(Canvas canvas, Label moveLabel, Label
statsLabel,
                                Slider speedSlider, Button playButton,
Button pauseButton,
                                Button resetButton, Solution solution) {
    this.boardCanvas = canvas;
    this.gc = canvas.getGraphicsContext2D();
    this.moveLabel = moveLabel;
    this.statsLabel = statsLabel;
    this.speedSlider = speedSlider;
    this.playButton = playButton;
    this.pauseButton = pauseButton;
    this.resetButton = resetButton;
    this.states = solution.getStates();
    this.moves = solution.getMoves();

    // Initialize piece colors
    initializePieceColors();

    // Set up the canvas size based on board dimensions
    Board initialBoard = states.get(0);
    this.width = initialBoard.getWidth();
    this.height = initialBoard.getHeight();
    boardCanvas.setWidth(width * CELL_SIZE + PADDING * 2);
    boardCanvas.setHeight(height * CELL_SIZE + PADDING * 2);

    // Set up animation
    setupAnimation();

    // Draw initial state
    drawBoard(initialBoard, null, 0);
    updateInfo();
}

```



```

/**
 * Initialize colors for pieces
 */
private void initializePieceColors() {
    // Define a set of visually distinct colors
    Color[] colors = {
        Color.CRIMSON,           // Primary piece (P)
        Color.ROYALBLUE,
        Color.LIMEGREEN,
        Color.GOLD,
        Color.DARKORANGE,
        Color.MEDIUMPURPLE,
        Color.DEEPSKYBLUE,
        Color.HOTPINK,
        Color.FORESTGREEN,
        Color.CHOCOLATE,
        Color.SLATEBLUE,
        Color.TOMATO,
        Color.TEAL,
        Color.INDIANRED,
        Color.MEDIUMSEAGREEN,
        Color.STEELBLUE,
        Color.DARKVIOLET,
        Color.SANDYBROWN,
        Color.CORAL,
        Color.DARKORANGE,
        Color.DARKCYAN,
        Color.MEDIUMORCHID,
        Color.LIGHTCORAL,
        Color.LIGHTSEAGREEN,
        Color.LIGHTSLATEGRAY,
        Color.LIGHTSTEELBLUE
    };

    // Assign colors to pieces in initial board
    Board initialBoard = states.get(0);
    int colorIndex = 1; // Start at 1, reserve 0 for primary piece

    // Assign primary piece color first
    pieceColors.put('P', colors[0]);

    // Assign colors to other pieces
    for (Piece piece : initialBoard.getPieces()) {
        if (piece.getId() != 'P') {
            pieceColors.put(piece.getId(), colors[colorIndex %
colors.length]);
            colorIndex++;
        }
    }
}

/**
 * Set up animation timeline

```

```

    */
    private void setupAnimation() {
        animation = new Timeline(
            new KeyFrame(Duration.millis(1000 / (speedSlider.getValue()
* animationSteps)), e -> {
                // Update animation step
                currentAnimationStep++;

                // If we completed all animation steps for this move
                if (currentAnimationStep >= animationSteps) {
                    currentAnimationStep = 0;
                    currentStateIndex++;

                    // If we reached the end, stop
                    if (currentStateIndex >= states.size()) {
                        currentStateIndex = states.size() - 1;
                        pause();
                    }
                }

                // Update the visualization
                updateVisualization();
            })
        );
        animation.setCycleCount(Timeline.INDEFINITE);

        // Set up slider to adjust animation speed
        speedSlider.valueProperty().addListener((obs, oldVal, newVal) ->
        {
            double millis = 1000 / (newVal.doubleValue() *
animationSteps);
            animation.getKeyFrames().clear();
            animation.getKeyFrames().add(new
KeyFrame(Duration.millis(millis), e -> {
                currentAnimationStep++;
                if (currentAnimationStep >= animationSteps) {
                    currentAnimationStep = 0;
                    currentStateIndex++;
                    if (currentStateIndex >= states.size()) {
                        currentStateIndex = states.size() - 1;
                        pause();
                    }
                }
            }
            updateVisualization();
        }));
    });
}

/**
 * Update visualization based on current state and animation step
 */
private void updateVisualization() {
    // Get current board state

```

```

        Board currentState = states.get(currentStateIndex);

        // Get the move that led to this state (if not the initial
state)
        Move currentMove = null;
        if (currentStateIndex > 0 && currentStateIndex <= moves.size())
        {
            currentMove = moves.get(currentStateIndex - 1);
        }

        // Draw the board with animation
drawBoard(currentState, currentMove, currentAnimationStep);

        // Update info labels
updateInfo();
    }

    /**
     * Update info labels
     */
    private void updateInfo() {
        // Update move information
        if (currentStateIndex == 0) {
            moveLabel.setText("Initial Board Configuration");
        } else if (currentStateIndex <= moves.size()) {
            Move move = moves.get(currentStateIndex - 1);
            moveLabel.setText(String.format("Move %d of %d: %s",
                currentStateIndex, moves.size(), move.toString()));
        }
    }

    /**
     * Draw board with animation
     * @param board The current board state
     * @param move The current move (may be null for initial state)
     * @param animStep Animation step (0 to animationSteps-1)
     */
    private void drawBoard(Board board, Move move, int animStep) {
        // Clear canvas
        gc.clearRect(0, 0, boardCanvas.getWidth(),
boardCanvas.getHeight());

        // Draw board background
        gc.setFill(Color.rgb(240, 240, 240));
        gc.fillRect(PADDING, PADDING, width * CELL_SIZE, height *
CELL_SIZE);

        // Draw grid lines
        gc.setStroke(Color.DARKGRAY);
        gc.setLineWidth(1);
        for (int i = 0; i <= width; i++) {
            gc.strokeLine(PADDING + i * CELL_SIZE, PADDING,
                PADDING + i * CELL_SIZE, PADDING + height *

```

```

CELL_SIZE);
    }
    for (int i = 0; i <= height; i++) {
        gc.strokeLine(PADDING, PADDING + i * CELL_SIZE,
                      PADDING + width * CELL_SIZE, PADDING + i *
CELL_SIZE);
    }

    // Draw exit
    drawExit(board);

    // Draw pieces
    for (Piece piece : board.getPieces()) {
        // For the piece being moved, draw it with animation
        if (move != null && piece.getId() ==
move.getPiece().getId()) {
            drawAnimatedPiece(piece, move, animStep);
        } else {
            drawPiece(piece);
        }
    }

    // Special case: primary piece exiting in last move
    if (currentStateIndex == states.size() - 1 &&
        currentAnimationStep > 0 &&
        move != null && move.getPiece().getId() == 'P') {
        drawExitingPrimaryPiece(board, move, currentAnimationStep);
    }
}

/**
 * Draw exit on the board
 */
private void drawExit(Board board) {
    Exit exitSide = board.getExitSide();
    Position exitPos = board.getExitPosition();

    gc.setFill(Color.LIGHTGREEN);
    gc.setStroke(Color.DARKGREEN);
    gc.setLineWidth(2);

    switch (exitSide) {
        case TOP:
            gc.fillRect(PADDING + exitPos.col * CELL_SIZE, PADDING -
20, CELL_SIZE, 20);
            gc.strokeRect(PADDING + exitPos.col * CELL_SIZE, PADDING
- 20, CELL_SIZE, 20);

            // Draw exit arrow
            gc.setFill(Color.DARKGREEN);
            double xTop = PADDING + exitPos.col * CELL_SIZE +
CELL_SIZE / 2;
            double yTop = PADDING - 5;

```

```

        gc.fillPolygon(
            new double[] {xTop, xTop - 10, xTop + 10},
            new double[] {yTop - 10, yTop, yTop},
            3
        );
        break;

    case BOTTOM:
        gc.fillRect(PADDING + exitPos.col * CELL_SIZE,
                    PADDING + height * CELL_SIZE, CELL_SIZE, 20);
        gc.strokeRect(PADDING + exitPos.col * CELL_SIZE,
                    PADDING + height * CELL_SIZE, CELL_SIZE,
20);

        // Draw exit arrow
        gc.setFill(Color.DARKGREEN);
        double xBottom = PADDING + exitPos.col * CELL_SIZE +
CELL_SIZE / 2;
        double yBottom = PADDING + height * CELL_SIZE + 15;
        gc.fillPolygon(
            new double[] {xBottom, xBottom - 10, xBottom + 10},
            new double[] {yBottom + 10, yBottom, yBottom},
            3
        );
        break;

    case LEFT:
        gc.fillRect(PADDING - 20, PADDING + exitPos.row *
CELL_SIZE, 20, CELL_SIZE);
        gc.strokeRect(PADDING - 20, PADDING + exitPos.row *
CELL_SIZE, 20, CELL_SIZE);

        // Draw exit arrow
        gc.setFill(Color.DARKGREEN);
        double xLeft = PADDING - 10;
        double yLeft = PADDING + exitPos.row * CELL_SIZE +
CELL_SIZE / 2;
        gc.fillPolygon(
            new double[] {xLeft - 10, xLeft, xLeft},
            new double[] {yLeft, yLeft - 10, yLeft + 10},
            3
        );
        break;

    case RIGHT:
        gc.fillRect(PADDING + width * CELL_SIZE,
                    PADDING + exitPos.row * CELL_SIZE, 20,
CELL_SIZE);
        gc.strokeRect(PADDING + width * CELL_SIZE,
                    PADDING + exitPos.row * CELL_SIZE, 20,
CELL_SIZE);

        // Draw exit arrow

```

```

        gc.setFill(Color.DARKGREEN);
        double xRight = PADDING + width * CELL_SIZE + 10;
        double yRight = PADDING + exitPos.row * CELL_SIZE +
CELL_SIZE / 2;
        gc.fillPolygon(
            new double[] {xRight + 10, xRight, xRight},
            new double[] {yRight, yRight - 10, yRight + 10},
            3
        );
        break;
    }

    // Draw exit label
    gc.setFill(Color.BLACK);
    gc.fillText("EXIT", PADDING + width * CELL_SIZE + 25, PADDING +
15);
}

/**
 * Draw a piece on the board
 */
private void drawPiece(Piece piece) {
    List<Position> positions = piece.getPositions();

    // Skip if no positions
    if (positions.isEmpty()) return;

    // Get color for this piece
    Color color = pieceColors.getOrDefault(piece.getId(),
Color.GRAY);

    // Calculate bounding box
    int minRow = Integer.MAX_VALUE;
    int maxRow = Integer.MIN_VALUE;
    int minCol = Integer.MAX_VALUE;
    int maxCol = Integer.MIN_VALUE;

    for (Position pos : positions) {
        minRow = Math.min(minRow, pos.row);
        maxRow = Math.max(maxRow, pos.row);
        minCol = Math.min(minCol, pos.col);
        maxCol = Math.max(maxCol, pos.col);
    }

    // Calculate dimensions
    double x = PADDING + minCol * CELL_SIZE + 5;
    double y = PADDING + minRow * CELL_SIZE + 5;
    double width = (maxCol - minCol + 1) * CELL_SIZE - 10;
    double height = (maxRow - minRow + 1) * CELL_SIZE - 10;

    // Draw piece with rounded corners and shadows
    gc.setFill(Color.rgb(0, 0, 0, 0.3));
    gc.fillRoundRect(x + 3, y + 3, width, height, 10, 10);

```

```

        gc.setFill(color);
        gc.fillRoundRect(x, y, width, height, 10, 10);

        gc.setStroke(color.darker());
        gc.setLineWidth(2);
        gc.strokeRoundRect(x, y, width, height, 10, 10);

        // Add a highlight effect
        gc.setFill(Color.rgb(255, 255, 255, 0.3));
        gc.fillRoundRect(x + 5, y + 5, width - 10, height / 3, 10, 10);

        // Draw piece ID
        gc.setFill(Color.WHITE);
        gc.fillText(String.valueOf(piece.getId()), x + width / 2 - 5, y
+ height / 2 + 5);
    }

    /**
     * Draw a piece with animation during movement
     */
    private void drawAnimatedPiece(Piece piece, Move move, int animStep)
    {
        List<Position> positions = piece.getPositions();

        // Skip if no positions
        if (positions.isEmpty()) return;

        // Get color for this piece
        Color color = pieceColors.getOrDefault(piece.getId(),
Color.GRAY);

        // Calculate bounding box
        int minRow = Integer.MAX_VALUE;
        int maxRow = Integer.MIN_VALUE;
        int minCol = Integer.MAX_VALUE;
        int maxCol = Integer.MIN_VALUE;

        for (Position pos : positions) {
            minRow = Math.min(minRow, pos.row);
            maxRow = Math.max(maxRow, pos.row);
            minCol = Math.min(minCol, pos.col);
            maxCol = Math.max(maxCol, pos.col);
        }

        // Calculate movement offset
        double offsetX = 0, offsetY = 0;
        int distance = 1; // Default distance is 1

        if (move instanceof CompoundMove) {
            distance = ((CompoundMove)move).getDistance();
        }
    }

```

```

        // Calculate fraction of movement
        double fraction = (double)animStep / animationSteps;

        // Determine direction and apply offset
        if (move.getDirection().equals("right")) {
            offsetX = fraction * distance * CELL_SIZE;
        } else if (move.getDirection().equals("left")) {
            offsetX = -fraction * distance * CELL_SIZE;
        } else if (move.getDirection().equals("down")) {
            offsetY = fraction * distance * CELL_SIZE;
        } else if (move.getDirection().equals("up")) {
            offsetY = -fraction * distance * CELL_SIZE;
        }

        // Calculate dimensions with offset
        double x = PADDING + minCol * CELL_SIZE + 5 + offsetX;
        double y = PADDING + minRow * CELL_SIZE + 5 + offsetY;
        double width = (maxCol - minCol + 1) * CELL_SIZE - 10;
        double height = (maxRow - minRow + 1) * CELL_SIZE - 10;

        // Draw piece with rounded corners and shadows
        gc.setFill(Color.rgb(0, 0, 0, 0.3));
        gc.fillRoundRect(x + 3, y + 3, width, height, 10, 10);

        gc.setFill(color);
        gc.fillRoundRect(x, y, width, height, 10, 10);

        gc.setStroke(color.darker());
        gc.setLineWidth(2);
        gc.strokeRoundRect(x, y, width, height, 10, 10);

        // Add a highlight effect
        gc.setFill(Color.rgb(255, 255, 255, 0.3));
        gc.fillRoundRect(x + 5, y + 5, width - 10, height / 3, 10, 10);

        // Draw piece ID
        gc.setFill(Color.WHITE);
        gc.fillText(String.valueOf(piece.getId()), x + width / 2 - 5, y
+ height / 2 + 5);
    }

    /**
     * Draw primary piece exiting the board
     */
    private void drawExitingPrimaryPiece(Board board, Move lastMove, int
animStep) {
        // Only proceed if it's a primary piece move
        if (lastMove.getPiece().getId() != 'P') return;

        // Get information about the primary piece
        int pieceSize = lastMove.getPiece().getSize();
        Exit exitSide = board.getExitSide();
        Position exitPos = board.getExitPosition();

```



```

        // Get the distance moved
        int distance = 1;
        if (lastMove instanceof CompoundMove) {
            distance = ((CompoundMove)lastMove).getDistance();
        }

        // Calculate how much of the piece is outside
        double fractionOutside = (double)animStep / animationSteps;
        int cellsOutside = (int)Math.ceil(fractionOutside *
Math.min(pieceSize, distance));

        // Draw the exiting piece based on exit side
        Color color = pieceColors.getOrDefault('P', Color.CRIMSON);

        switch (exitSide) {
            case RIGHT:
                for (int i = 0; i < cellsOutside; i++) {
                    double x = PADDING + board.getWidth() * CELL_SIZE +
5 + i * CELL_SIZE;
                    double y = PADDING + exitPos.row * CELL_SIZE + 5;

                    // Draw exiting piece cell
                    gc.setFill(color);
                    gc.fillRoundRect(x, y, CELL_SIZE - 10, CELL_SIZE -
10, 10, 10, 10);

                    gc.setStroke(color.darker());
                    gc.strokeRoundRect(x, y, CELL_SIZE - 10, CELL_SIZE -
10, 10, 10, 10);

                    // Add highlight
                    gc.setFill(Color.rgb(255, 255, 255, 0.3));
                    gc.fillRoundRect(x + 5, y + 5, CELL_SIZE - 20,
(CELL_SIZE - 10) / 3, 8, 8);

                    // Draw P if it's the first exiting cell
                    if (i == 0) {
                        gc.setFill(Color.WHITE);
                        gc.fillText("P", x + CELL_SIZE / 2 - 5, y +
CELL_SIZE / 2 + 5);
                    }
                }
                break;

            case LEFT:
                for (int i = 0; i < cellsOutside; i++) {
                    double x = PADDING - (i + 1) * CELL_SIZE + 5;
                    double y = PADDING + exitPos.row * CELL_SIZE + 5;

                    // Draw exiting piece cell
                    gc.setFill(color);
                    gc.fillRoundRect(x, y, CELL_SIZE - 10, CELL_SIZE -
10, 10, 10, 10);

```

```

gc.setStroke(color.darker());
gc.strokeRoundRect(x, y, CELL_SIZE - 10, CELL_SIZE -
10, 10, 10);

// Add highlight
gc.setFill(Color.rgb(255, 255, 255, 0.3));
gc.fillRoundRect(x + 5, y + 5, CELL_SIZE - 20,
(CELL_SIZE - 10) / 3, 8, 8);

// Draw P if it's the first exiting cell
if (i == 0) {
    gc.setFill(Color.WHITE);
    gc.fillText("P", x + CELL_SIZE / 2 - 5, y +
CELL_SIZE / 2 + 5);
}
}
break;

case BOTTOM:
    for (int i = 0; i < cellsOutside; i++) {
        double x = PADDING + exitPos.col * CELL_SIZE + 5;
        double y = PADDING + board.getHeight() * CELL_SIZE +
5 + i * CELL_SIZE;

        // Draw exiting piece cell
        gc.setFill(color);
        gc.fillRoundRect(x, y, CELL_SIZE - 10, CELL_SIZE -
10, 10, 10);

        gc.setStroke(color.darker());
        gc.strokeRoundRect(x, y, CELL_SIZE - 10, CELL_SIZE -
10, 10, 10);

        // Add highlight
        gc.setFill(Color.rgb(255, 255, 255, 0.3));
        gc.fillRoundRect(x + 5, y + 5, CELL_SIZE - 20,
(CELL_SIZE - 10) / 3, 8, 8);

        // Draw P if it's the first exiting cell
        if (i == 0) {
            gc.setFill(Color.WHITE);
            gc.fillText("P", x + CELL_SIZE / 2 - 5, y +
CELL_SIZE / 2 + 5);
        }
    }
    break;

case TOP:
    for (int i = 0; i < cellsOutside; i++) {
        double x = PADDING + exitPos.col * CELL_SIZE + 5;
        double y = PADDING - (i + 1) * CELL_SIZE + 5;

        // Draw exiting piece cell
        gc.setFill(color);

```

```

        gc.fillRoundRect(x, y, CELL_SIZE - 10, CELL_SIZE -
10, 10, 10);
        gc.setStroke(color.darker());
        gc.strokeRoundRect(x, y, CELL_SIZE - 10, CELL_SIZE -
10, 10, 10);

        // Add highlight
        gc.setFill(Color.rgb(255, 255, 255, 0.3));
        gc.fillRoundRect(x + 5, y + 5, CELL_SIZE - 20,
(CELL_SIZE - 10) / 3, 8, 8);

        // Draw P if it's the first exiting cell
        if (i == 0) {
            gc.setFill(Color.WHITE);
            gc.fillText("P", x + CELL_SIZE / 2 - 5, y +
CELL_SIZE / 2 + 5);
        }
        break;
    }
}

/**
 * Play the animation
 */
public void play() {
    // Only play if not at the end
    if (currentStateIndex < states.size() - 1 ||
currentAnimationStep < animationSteps - 1) {
        animation.play();
        isPlaying = true;
        playButton.setDisable(true);
        pauseButton.setDisable(false);
    }
}

/**
 * Pause the animation
 */
public void pause() {
    if (isPlaying) {
        animation.pause();
        isPlaying = false;
        playButton.setDisable(false);
        pauseButton.setDisable(true);
    }
}

/**
 * Reset the animation to the beginning
 */
public void reset() {
    pause();

```

```

        currentStateIndex = 0;
        currentAnimationStep = 0;
        drawBoard(states.get(0), null, 0);
        updateInfo();
        playButton.setDisable(false);
    }

    /**
     * Clean up resources
     */
    public void cleanup() {
        if (animation != null) {
            animation.stop();
        }
    }
}

```

3.2.13 *InputController.java*

```

package gui.controllers;

import cli.*;
import javafx.fxml.FXML;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.image.Image;
import javafx.scene.layout.*;
import javafx.scene.text.Text;
import javafx.stage.FileChooser;
import javafx.stage.Stage;
import javafx.application.Platform;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class InputController {

    @FXML private TabPane inputTabPane;
    @FXML private Tab fileTab;
    @FXML private Tab textTab;
}

```

```

@FXML private Tab matrixTab;
@FXML private VBox rootContainer;

// File input components
@FXML private TextField filePathField;
@FXML private Button browseButton;
@FXML private Button loadFileButton;

// Text input components
@FXML private TextArea configTextArea;
@FXML private Button parseTextButton;

// Matrix input components
@FXML private Spinner<Integer> rowSpinner;
@FXML private Spinner<Integer> colSpinner;
@FXML private Spinner<Integer> piecesSpinner;
@FXML private GridPane matrixGrid;
@FXML private Button createMatrixButton;
@FXML private Button solveMatrixButton;
@FXML private VBox matrixInputContainer;

// Algorithm selection
@FXML private ComboBox<String> algorithmComboBox;
@FXML private ComboBox<String> heuristicComboBox;
@FXML private Button solveButton;

// Status components
@FXML private Text statusText;
@FXML private ScrollPane statusScrollPane;

private Board currentBoard;
private List<TextField> matrixCells;
private boolean isCompound = false;

@FXML
public void initialize() {
    // Initialize spinners
    rowSpinner.setValueFactory(new
SpinnerValueFactory.IntegerSpinnerValueFactory(4, 10, 6));
    colSpinner.setValueFactory(new
SpinnerValueFactory.IntegerSpinnerValueFactory(4, 10, 6));
    piecesSpinner.setValueFactory(new
SpinnerValueFactory.IntegerSpinnerValueFactory(1, 20, 12));

    // Initialize algorithm dropdown
    algorithmComboBox.getItems().addAll(
        "Uniform Cost Search (UCS)",
        "Greedy Best First Search",
        "A* Search",
        "Dijkstra's Algorithm"
    );
    algorithmComboBox.getSelectionModel().selectFirst();
}

```

```

// Initialize heuristic dropdown
heuristicComboBox.getItems().addAll(
    "Manhattan Distance",
    "Direct Distance",
    "Blocking Count"
);
heuristicComboBox.getSelectionModel().selectFirst();

// Initially disable heuristic for UCS
heuristicComboBox.setDisable(true);
algorithmComboBox.setOnAction(e -> {
    String selected = algorithmComboBox.getValue();
    boolean needsHeuristic = selected.contains("Greedy") ||
selected.contains("A*");
    heuristicComboBox.setDisable(!needsHeuristic);
});

matrixCells = new ArrayList<>();
matrixInputContainer.setVisible(false);

// Add sample configuration
configTextArea.setText("6
6\n11\nAAB..F\n..BCDF\nGPPCDFK\nGH.III\nGHJ...\nLLJMM.");

setBackground();
}

private void setBackground() {
    try {
        // Try to load background image
        String backgroundPath = "resources/images/bocchi.jpg";
        BackgroundImage backgroundImage = null;

        try {
            Image image = new
Image(getClass().getClassLoader().getResourceAsStream(backgroundPath));
            if (!image.isError()) {
                backgroundImage = new BackgroundImage(
                    image,
                    BackgroundRepeat.REPEAT,
BackgroundRepeat.REPEAT,
                    BackgroundPosition.DEFAULT,
                    new BackgroundSize(1.0, 1.0, true, true, false,
false)
                );
            }
        } catch (Exception e) {
            System.err.println("Error loading background image: " +
e.getMessage());
            // Try alternative path
            try {
                File file = new File(backgroundPath);
                if (file.exists()) {

```

```

        Image image = new
Image(file.toURI().toString());
        backgroundImage = new BackgroundImage(
            image,
            BackgroundRepeat.REPEAT,
BackgroundRepeat.REPEAT,
            BackgroundPosition.DEFAULT,
            new BackgroundSize(1.0, 1.0, true, true,
false, false)
        );
    }
    } catch (Exception ex) {
        System.err.println("Error loading background from
file: " + ex.getMessage());
    }
}

    if (backgroundImage != null) {
        rootContainer.setBackground(new
Background(backgroundImage));
    } else {
        // Use dark color as fallback
        rootContainer.setStyle("-fx-background-color:
#121212;");
    }
    } catch (Exception e) {
        System.err.println("Error setting background: " +
e.getMessage());
    }
}

@FXML
private void handleToggleCompound() {
    isCompound = !isCompound;
    if (isCompound) {
        updateStatus("Compound moves enabled", false);
    } else {
        updateStatus("Compound moves disabled", false);
    }
}

@FXML
private void handleBrowse() {
    FileChooser fileChooser = new FileChooser();
    fileChooser.setTitle("Select Rush Hour Configuration File");
    fileChooser.getExtensionFilters().add(
        new FileChooser.ExtensionFilter("Text Files", "*.txt")
    );

    // Set initial directory to test/input if it exists
    File testInputDir = new File("test/input");
    if (testInputDir.exists() && testInputDir.isDirectory()) {
        fileChooser.setInitialDirectory(testInputDir);
    }
}

```

```

    }

    File file =
fileChooser.showOpenDialog(browseButton.getScene().getWindow());
    if (file != null) {
        filePathField.setText(file.getAbsolutePath());
        handleLoadFile(); // Auto-load when file is selected
    }
}

@FXML
private void handleLoadFile() {
    String filePath = filePathField.getText();
    if (filePath.isEmpty()) {
        updateStatus("Error: Please select a file first", true);
        return;
    }

    try {
        currentBoard = Board.readFromFile(filePath);

        // Check if primary piece is aligned with exit
        if (!currentBoard.isPrimaryPieceAlignedWithExit()) {
            boolean shouldContinue = showConfirmation(
                "Warning",
                "Primary piece and exit are not aligned. The puzzle
may not be solvable. Continue anyway?"
            );
            if (!shouldContinue) {
                currentBoard = null;
                return;
            }
        }

        updateStatus("Board loaded successfully!", false);
        solveButton.setDisable(false);
    } catch (IOException e) {
        updateStatus("Error: " + e.getMessage(), true);
        currentBoard = null;
        solveButton.setDisable(true);
    }
}

@FXML
private void handleParseText() {
    String configText = configTextArea.getText();
    if (configText.trim().isEmpty()) {
        updateStatus("Error: Please enter a configuration", true);
        return;
    }

    try {
        // Save to temporary file and parse

```



```

        File tempFile = File.createTempFile("rushHour", ".txt");
        try (FileWriter writer = new FileWriter(tempFile)) {
            writer.write(configText);
        }

        currentBoard =
Board.readFromFile(tempFile.getAbsolutePath());
        tempFile.delete();

        // Check if primary piece is aligned with exit
        if (!currentBoard.isPrimaryPieceAlignedWithExit()) {
            boolean shouldContinue = showConfirmation(
                "Warning",
                "Primary piece and exit are not aligned. The puzzle
may not be solvable. Continue anyway?"
            );
            if (!shouldContinue) {
                currentBoard = null;
                return;
            }
        }

        updateStatus("Configuration parsed successfully!", false);
        solveButton.setDisable(false);
    } catch (IOException e) {
        updateStatus("Error: " + e.getMessage(), true);
        currentBoard = null;
        solveButton.setDisable(true);
    }
}

@FXML
private void handleCreateMatrix() {
    int rows = rowSpinner.getValue();
    int cols = colSpinner.getValue();

    matrixGrid.getChildren().clear();
    matrixCells.clear();

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            TextField cell = new TextField();
            cell.setPrefWidth(40);
            cell.setPrefHeight(40);
            cell.setMaxWidth(40);
            cell.setMaxHeight(40);
            cell.getStyleClass().add("matrix-cell");
            cell.setAlignment(javafx.geometry.Pos.CENTER);
            cell.setTextFormatter(new TextFormatter<>(change -> {
                String newText = change.getControlNewText();
                if (newText.length() <= 1 && (newText.isEmpty() ||
                    newText.matches("[A-Z.]")) ||
newText.equals("K"))) {

```

```

        return change;
    }
    return null;
}));

matrixGrid.add(cell, j, i);
matrixCells.add(cell);
}
}

matrixInputContainer.setVisible(true);
solveMatrixButton.setDisable(false);
}

@FXML
private void handleSolveMatrix() {
    int rows = rowSpinner.getValue();
    int cols = colSpinner.getValue();
    int numPieces = piecesSpinner.getValue();

    // Build configuration string from matrix
    StringBuilder config = new StringBuilder();
    config.append(rows).append(" ").append(cols).append("\n");
    config.append(numPieces).append("\n");

    boolean foundP = false;
    boolean foundK = false;

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            int index = i * cols + j;
            String value = matrixCells.get(index).getText();
            if (value.isEmpty()) value = ".";

            if (value.equals("P")) foundP = true;
            if (value.equals("K")) foundK = true;

            config.append(value);
        }
        if (i < rows - 1) config.append("\n");
    }

    if (!foundP) {
        updateStatus("Error: Primary piece 'P' not found in matrix",
true);
        return;
    }

    if (!foundK) {
        updateStatus("Error: Exit 'K' not found in matrix", true);
        return;
    }
}

```

```

        // Parse the configuration
        try {
            File tempFile = File.createTempFile("rushHourMatrix",
".txt");
            try (FileWriter writer = new FileWriter(tempFile)) {
                writer.write(config.toString());
            }

            currentBoard =
Board.readFromFile(tempFile.getAbsolutePath());
            tempFile.delete();

            updateStatus("Matrix configuration loaded successfully!",
false);
            solveButton.setDisable(false);
        } catch (IOException e) {
            updateStatus("Error: " + e.getMessage(), true);
            currentBoard = null;
            solveButton.setDisable(true);
        }
    }

    @FXML
    private void handleSolve() {
        if (currentBoard == null) {
            updateStatus("Error: Please load a board configuration
first", true);
            return;
        }

        String algorithm = algorithmComboBox.getValue();
        String heuristic = heuristicComboBox.getValue();

        // Disable UI during solving
        solveButton.setDisable(true);
        updateStatus("Solving puzzle...", false);

        // Run solving in background thread
        new Thread(() -> {
            try {
                Solver solver = new Solver();
                Solution solution = null;

                long startTime = System.currentTimeMillis();

                // Select algorithm based on combo box selection
                if (algorithm.contains("UCS")) {
                    solution = solver.solveUCS(currentBoard,
isCompound);
                } else if (algorithm.contains("Dijkstra")) {
                    solution = solver.solveDijkstra(currentBoard,
isCompound);
                } else if (algorithm.contains("Greedy")) {

```

```

        solution = solver.solveGreedy(currentBoard,
        heuristic, isCompound);
    } else if (algorithm.contains("A*")) {
        solution = solver.solveAStar(currentBoard,
        heuristic, isCompound);
    } else if (algorithm.contains("Beam")) {
        solution = solver.solveBeam(currentBoard, heuristic,
        isCompound);
    } else if (algorithm.contains("Iterative")) {
        solution = solver.solveIDAStar(currentBoard,
        heuristic, isCompound);
    }

    long endTime = System.currentTimeMillis();

    final Solution finalSolution = solution;
    final long executionTime = endTime - startTime;

    // Update UI on JavaFX thread
    Platform.runLater(() -> {
        solveButton.setDisable(false);

        if (finalSolution != null) {
            updateStatus("Solution found! Opening
visualization...", false);
            openVisualization(finalSolution, currentBoard,
            algorithm, heuristic, executionTime);
        } else {
            updateStatus("No solution found for this
configuration", true);
        }
    });

    } catch (Exception e) {
        Platform.runLater(() -> {
            solveButton.setDisable(false);
            updateStatus("Error solving puzzle: " +
e.getMessage(), true);
            e.printStackTrace();
        });
    }
    }).start();
}

private void openVisualization(Solution solution, Board
initialBoard, String algorithm,
                                String heuristic, long executionTime) {
    try {
        FXXMLLoader loader = new
FXXMLLoader(getClass().getResource("/gui/fxml/VisualizationView.fxml"));
        Parent root = loader.load();

        VisualizationController controller = loader.getController();

```

```

        controller.initialize(solution, initialBoard, algorithm,
        heuristic, executionTime);

        Stage stage = new Stage();
        stage.setTitle("Rush Hour Solution Visualization");
        Scene scene = new Scene(root, 800, 650);

        // Try to load CSS
        try {

scene.getStylesheets().add(getClass().getResource("/resources/styles.css
").toExternalForm());
            } catch (Exception e) {
                System.out.println("Could not load dark-mode-styles.css,
trying styles.css");
                try {

scene.getStylesheets().add(getClass().getResource("/resources/styles.css
").toExternalForm());
                    } catch (Exception ex) {
                        System.out.println("Could not load styles.css");
                    }
                }

            stage.setScene(scene);
            stage.setMinWidth(800);
            stage.setMinHeight(650);
            stage.show();

        } catch (IOException e) {
            updateStatus("Error opening visualization: " +
e.getMessage(), true);
            e.printStackTrace();
        }
    }

    private void updateStatus(String message, boolean isError) {
        statusText.setText(message);

        if (isError) {
            statusText.getStyleClass().add("error");
        } else {
            statusText.getStyleClass().removeAll("error");
        }
    }

    private boolean showConfirmation(String title, String content) {
        Alert alert = new Alert(Alert.AlertType.CONFIRMATION);
        alert.setTitle(title);
        alert.setHeaderText(null);
        alert.setContentText(content);

        // Apply dark mode to alert

```

```

        DialogPane dialogPane = alert.getDialogPane();

        dialogPane.getStylesheets().add(getClass().getResource("/resources/style
s.css").toExternalForm());
        dialogPane.getStyleClass().add("dark-dialog");

        ButtonType result =
        alert.showAndWait().orElse(ButtonType.CANCEL);
        return result == ButtonType.OK;
    }
}

```

3.2.14 MainController.java

```

package gui.controllers;

import cli.*;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.application.Platform;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.fxml.FXML;
import javafx.fxml.FXMLLoader;
import javafx.geometry.Insets;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;
import javafx.scene.paint.Color;
import javafx.scene.text.Text;
import javafx.stage.FileChooser;
import javafx.stage.Modality;
import javafx.stage.Stage;
import javafx.util.Duration;
import javafx.scene.Node;

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

/**
 * MainController class with zoom support for large board sizes
 * Compatible with the provided FXML layout
 */

```

```

public class MainController {

    // FXML components from your layout
    @FXML private BorderPane rootContainer;
    @FXML private VBox sidebarContainer;
    @FXML private TabPane inputTabPane;
    @FXML private TextField filePathField;
    @FXML private Button browseButton;
    @FXML private Button loadFileButton;
    @FXML private TextArea configTextArea;
    @FXML private Button parseTextButton;
    @FXML private Spinner<Integer> rowSpinner;
    @FXML private Spinner<Integer> colSpinner;
    @FXML private Spinner<Integer> piecesSpinner;
    @FXML private GridPane matrixGrid;
    @FXML private Button createMatrixButton;
    @FXML private VBox matrixInputContainer;
    @FXML private ComboBox<String> algorithmComboBox;
    @FXML private ComboBox<String> heuristicComboBox;
    @FXML private Button solveButton;
    @FXML private Text statusText;
    @FXML private ProgressBar progressBar;
    @FXML private Button compoundButton;

    // Board visualization elements
    @FXML private StackPane canvasContainer;
    @FXML private VBox boardDisplay;
    @FXML private VBox welcomePane;
    @FXML private Canvas boardCanvas;
    @FXML private HBox animationControls;
    @FXML private Button playButton;
    @FXML private Button pauseButton;
    @FXML private Button resetButton;
    @FXML private Button saveButton;
    @FXML private Slider speedSlider;
    @FXML private Label moveLabel;
    @FXML private Label statsLabel;
    @FXML private ComboBox<String> songComboBox;
    @FXML private Button pauseSongButton;
    @FXML private BorderPane boardContainer;
    @FXML private VBox zoomControlsVBox;

    // Zoom-related components (will be created programmatically)
    private HBox zoomControls;
    private Button zoomInButton;
    private Button zoomOutButton;
    private Button resetZoomButton;
    private Slider zoomSlider;
    private Label zoomPercentLabel;

    // Board state and data
    private Board currentBoard;

```

```

// Solution and animation state
private Solution solution;
private List<Board> boardStates;
private List<Move> moves;
private int currentStateIndex = 0;
private Timeline animation;
private boolean isPlaying = false;
private long executionTime;
private int nodesExamined;
private boolean isCompound = true;

// Zoom controller
private ZoomController zoomController;

// Visualization constants
private static final int CELL_SIZE = 60;
private static final int PADDING = 40;
private final Map<Character, Color> pieceColors = new HashMap<>();

// Misc
private MediaPlayer mediaPlayer;
private final String SONG_PATH = "src/resources/song/";
private String playedSong = "If_I_could_be_a_constellation.mp3";

/**
 * Initialize the controller
 */
@FXML
public void initialize() {
    // Initialize songs
    ObservableList<String> songs =
FXCollections.observableArrayList(
        "If_I_could_be_a_constellation.mp3",
        "Re_Re_.mp3",
        "Rockn' Roll, Morning Light Falls on You.mp3",
        "seisyun_complex.mp3",
        "Color_Your_Night.mp3",
        "Rubia.mp3"
    );
    songComboBox.setItems(songs);
    songComboBox.getSelectionModel().selectFirst();

    // Optionally play the first song by default
    playBackgroundMusic(songs.get(0));

    // Initialize compound button
    updateCompoundButtonText();

    // Initialize input components
    initializeInputs();

    // Initialize zoom controls
    createZoomControls();
}

```



```

        // Set initial status
        updateStatus("Ready to load configuration");
    }

    /**
     * Create zoom controls programmatically since they're not in the
     provided FXML
     */
    private void createZoomControls() {
        // Create vertical zoom slider container that will go on the
right side of the canvas
        VBox verticalZoomControls = new VBox();
        verticalZoomControls.setSpacing(10);
        verticalZoomControls.setAlignment(javafx.geometry.Pos.CENTER);

verticalZoomControls.getStyleClass().add("zoom-controls-vertical");
        verticalZoomControls.setVisible(false);
        verticalZoomControls.setManaged(false);
        verticalZoomControls.setPrefWidth(50); // Fixed width for the
control panel

        // Create zoom buttons
        zoomInButton = new Button("+");
        zoomInButton.getStyleClass().add("zoom-button");
        zoomInButton.setTooltip(new Tooltip("Zoom In"));
        zoomInButton.setMaxWidth(Double.MAX_VALUE); // Make button fill
width

        zoomOutButton = new Button("-");
        zoomOutButton.getStyleClass().add("zoom-button");
        zoomOutButton.setTooltip(new Tooltip("Zoom Out"));
        zoomOutButton.setMaxWidth(Double.MAX_VALUE); // Make button
fill width

        resetZoomButton = new Button("↺"); // Reset symbol
        resetZoomButton.getStyleClass().add("zoom-button");
        resetZoomButton.setTooltip(new Tooltip("Reset Zoom"));
        resetZoomButton.setMaxWidth(Double.MAX_VALUE); // Make button
fill width

        // Create vertical zoom slider
        zoomSlider = new Slider(0.25, 3.0, 1.0);
        // Use JavaFX Orientation, not CLI Orientation
        zoomSlider.setOrientation(javafx.geometry.Orientation.VERTICAL);
// Make slider vertical
        zoomSlider.setShowTickMarks(true);
        zoomSlider.setShowTickLabels(true);
        zoomSlider.setMajorTickUnit(0.5);
        zoomSlider.setMinorTickCount(4);
        zoomSlider.setPrefHeight(200.0); // Make slider adequately tall

        // Create zoom percentage label

```

```

        zoomPercentLabel = new Label("100%");
        zoomPercentLabel.getStyleClass().add("zoom-percent-label");

        // Add all components to the container with proper spacing
        verticalZoomControls.getChildren().addAll(
            zoomInButton,
            zoomSlider,
            zoomOutButton,
            resetZoomButton,
            zoomPercentLabel
        );

        // We need an HBox to match the original zoomControls type
        HBox zoomControlsWrapper = new HBox();
        zoomControlsWrapper.setAlignment(javafx.geometry.Pos.CENTER);
        zoomControlsWrapper.setVisible(false);
        zoomControlsWrapper.setManaged(false);

        // Store reference to zoom controls
        this.zoomControls = zoomControlsWrapper;

        // IMPORTANT: Rather than modifying the canvas container
        directly,
        // we need to put the canvas in a separate container that can
        scroll/zoom
        // while keeping the labels and controls fixed

        // First, check if we've already set up our custom container
        boolean alreadySetup = false;
        for (javafx.scene.Node node : canvasContainer.getChildren()) {
            if (node instanceof VBox && node.getId() != null &&
node.getId().equals("zoomableCanvasContainer")) {
                alreadySetup = true;
                break;
            }
        }

        if (!alreadySetup) {
            // Create a container for the zoomable canvas
            StackPane zoomableArea = new StackPane();
            zoomableArea.setId("zoomableCanvasContainer");

            zoomableArea.getStyleClass().add("zoomable-canvas-container");

            // Save the original parent of the canvas
            Parent originalParent = boardCanvas.getParent();

            // If the canvas has a parent, remove it from there
            if (originalParent instanceof Pane) {
                ((Pane)
originalParent).getChildren().remove(boardCanvas);
            }
        }

```

```

        // Add canvas to zoomable area
        zoomableArea.getChildren().add(boardCanvas);

        // Add our vertical zoom controls to the zoomable area
        zoomableArea.getChildren().add(verticalZoomControls);

        // Position the zoom controls to the right of the canvas
        StackPane.setAlignment(verticalZoomControls,
javafx.geometry.Pos.CENTER_RIGHT);
        // Add some margin to avoid overlap with the canvas
        StackPane.setMargin(verticalZoomControls, new Insets(0, 20,
0, 0));

        // Create a VBox to hold the zoomable area and keep the
labels below fixed
        VBox boardContainerWithLabels = new VBox(10); // 10px
spacing

boardContainerWithLabels.setAlignment(javafx.geometry.Pos.CENTER);

        // Add the zoomable area to the top of the VBox
        boardContainerWithLabels.getChildren().add(zoomableArea);

        // Keep track of the original children that were below the
canvas
        List<javafx.scene.Node> belowCanvasNodes = new
ArrayList<>();

        // If canvasContainer already has children, identify which
ones are below the canvas
        // and add them to our new container in the same order
        if (canvasContainer.getChildren().contains(boardCanvas)) {
            int canvasIndex =
canvasContainer.getChildren().indexOf(boardCanvas);
            for (int i = canvasIndex + 1; i <
canvasContainer.getChildren().size(); i++) {
belowCanvasNodes.add(canvasContainer.getChildren().get(i));
            }

            // Remove all nodes that we've identified
            canvasContainer.getChildren().removeAll(belowCanvasNodes);

            // Add them to our new container

boardContainerWithLabels.getChildren().addAll(belowCanvasNodes);
        }

        // Add our new container to the canvas container
        canvasContainer.getChildren().add(boardContainerWithLabels);
    }

```

```

        // Set up event handlers for buttons
        resetZoomButton.setOnAction(e -> {
            if (zoomController != null) {
                zoomController.resetZoom();
                updateZoomPercentLabel();
            }
        });

        zoomInButton.setOnAction(e -> {
            if (zoomController != null) {
                zoomController.zoomIn();
                updateZoomPercentLabel();
            }
        });

        zoomOutButton.setOnAction(e -> {
            if (zoomController != null) {
                zoomController.zoomOut();
                updateZoomPercentLabel();
            }
        });

        // Set up listener for slider changes
        zoomSlider.valueProperty().addListener((obs, oldVal, newVal) ->
        {
            updateZoomPercentLabel();
            if (zoomController != null) {
                zoomController.zoom(newVal.doubleValue());
            }
        });

    }

    /**
     * Update zoom percentage label
     */
    private void updateZoomPercentLabel() {
        if (zoomPercentLabel != null && zoomSlider != null) {
            int percentage = (int) (zoomSlider.getValue() * 100);
            zoomPercentLabel.setText(percentage + "%");
        }
    }

    /**
     * Initialize input elements
     */
    private void initializeInputs() {
        // Initialize spinners with a minimum value of 1
        rowSpinner.setValueFactory(new
SpinnerValueFactory.IntegerSpinnerValueFactory(1, 25, 6));
        colSpinner.setValueFactory(new
SpinnerValueFactory.IntegerSpinnerValueFactory(1, 25, 6));
        piecesSpinner.setValueFactory(new

```

```

SpinnerValueFactory.IntegerSpinnerValueFactory(1, 50, 12));

    // Add a listener to prevent 1x1 boards
    rowSpinner.valueProperty().addListener((obs, oldVal, newVal) ->
    {
        if (newVal == 1 && colSpinner.getValue() == 1) {
            colSpinner.getValueFactory().setValue(2);
            updateStatus("Board size must be at least 1x2 or 2x1",
true);
        }

        // Show zoom controls for larger boards
        checkAndShowZoomControls();
    });

    colSpinner.valueProperty().addListener((obs, oldVal, newVal) ->
    {
        if (newVal == 1 && rowSpinner.getValue() == 1) {
            rowSpinner.getValueFactory().setValue(2);
            updateStatus("Board size must be at least 1x2 or 2x1",
true);
        }

        // Show zoom controls for larger boards
        checkAndShowZoomControls();
    });

    // Initialize algorithm dropdown
    algorithmComboBox.getItems().addAll(
        "Uniform Cost Search (UCS)",
        "Greedy Best First Search",
        "A* Search",
        "Dijkstra's Algorithm",
        "Beam Search [Not Complete Search]",
        "Iterative Deepening A*"
    );
    algorithmComboBox.getSelectionModel().selectFirst();

    // Initialize heuristic dropdown
    heuristicComboBox.getItems().addAll(
        "Manhattan Distance",
        "Direct Distance",
        "Blocking Count",
        "Clearing Moves"
    );
    heuristicComboBox.getSelectionModel().selectFirst();

    // Initially disable heuristic for UCS
    heuristicComboBox.setDisable(true);
    algorithmComboBox.setOnAction(e -> {
        String selected = algorithmComboBox.getValue();
        boolean needsHeuristic = selected.contains("Greedy") ||
selected.contains("A*") || selected.contains("Beam") ||

```

```

selected.contains("Iterative");
    heuristicComboBox.setDisable(!needsHeuristic);
});

    // Setup text area with sample configuration
    configTextArea.setText("6
6\n11\nAAB..F\n..BCDF\nGPPCDFK\nGH.III\nGHJ...\nLLJMM.");

    // Hide matrix input initially
    matrixInputContainer.setVisible(false);

    // Disable solve button initially
    solveButton.setDisable(true);
}

/**
 * Check if zoom controls should be shown based on board size
 */
private void checkAndShowZoomControls() {
    boolean largeBoard = rowSpinner.getValue() > 10 ||
colSpinner.getValue() > 10;

    if (zoomControls != null) {
        zoomControls.setVisible(largeBoard);
        zoomControls.setManaged(largeBoard);
    }
}

/**
 * Initialize canvas and animation
 */
private void initializeCanvas() {
    // Setup animation
    animation = new Timeline(
        new KeyFrame(Duration.seconds(1.0 / speedSlider.getValue()),
e -> {
            if (currentStateIndex < boardStates.size() - 1) {
                currentStateIndex++;
                drawBoard(boardStates.get(currentStateIndex));
                updateMoveInfo();

                // If solved
                if (currentStateIndex == boardStates.size() - 1) {
                    updateStatus("Puzzle solved successfully!");
                }
            } else {
                pauseAnimation();
            }
        })
    );
    animation.setCycleCount(Timeline.INDEFINITE);

    // Update animation speed when slider changes

```

```

        speedSlider.valueProperty().addListener((obs, oldVal, newVal) ->
{
    animation.setRate(newVal.doubleValue());
});

// Initialize zoom controller
zoomController = new ZoomController(
    boardCanvas,
    canvasContainer,
    zoomInButton,
    zoomOutButton,
    zoomSlider
);

// Show zoom controls for large boards
checkAndShowZoomControls();
}

private void updateCompoundButtonText() {
    if (compoundButton != null) {
        compoundButton.setText(isCompound ? "ON" : "OFF");
    }
}

/**
 * Adjust canvas size based on board dimensions
 */
private void adjustCanvasSize(Board board) {
    int width = board.getWidth();
    int height = board.getHeight();

    // Calculate required canvas size
    double canvasWidth = width * CELL_SIZE + PADDING * 2;
    double canvasHeight = height * CELL_SIZE + PADDING * 2;

    // Set canvas size
    boardCanvas.setWidth(canvasWidth);
    boardCanvas.setHeight(canvasHeight);

    // Determine if zoom controls should be shown based on board
size
    boolean largeBoard = width > 10 || height > 10;

    // Find the vertical zoom controls (now they're siblings of the
canvas)
    StackPane zoomableArea = null;
    if (boardCanvas.getParent() instanceof StackPane) {
        zoomableArea = (StackPane) boardCanvas.getParent();
    }

    VBox verticalZoomControls = null;
    if (zoomableArea != null) {
        for (javafx.scene.Node node : zoomableArea.getChildren()) {

```

```

        if (node instanceof VBox &&
node.getStyleClass().contains("zoom-controls-vertical")) {
            verticalZoomControls = (VBox) node;
            break;
        }
    }

    // Show or hide zoom controls based on board size
    if (verticalZoomControls != null) {
        verticalZoomControls.setVisible(largeBoard);
        verticalZoomControls.setManaged(largeBoard);
    }

    // Check if we need to scale the board to fit in the view
    if (largeBoard && zoomController != null && zoomableArea !=
null) {
        // Get the available space for the canvas
        // Use the parent of the zoomable area to determine
available space
        double availableWidth = zoomableArea.getWidth() -
(largeBoard ? 70 : 0); // 50px for controls + 20px margin
        double availableHeight = zoomableArea.getHeight();

        // If the board is larger than available space, calculate
scaling
        if (canvasWidth > availableWidth || canvasHeight >
availableHeight) {
            double widthRatio = availableWidth / canvasWidth;
            double heightRatio = availableHeight / canvasHeight;
            double scaleFactor = Math.min(widthRatio, heightRatio) *
0.9; // 90% of max scale to leave a margin

            // Ensure scaling is within our zoom limits
            scaleFactor = Math.max(0.25, Math.min(scaleFactor,
1.0));

            // Apply zoom
            zoomController.zoom(scaleFactor);

            // Update slider value
            if (zoomSlider != null) {
                zoomSlider.setValue(scaleFactor);
            }

            // Update zoom percentage label
            updateZoomPercentLabel();
        } else {
            // Board fits, reset zoom
            zoomController.resetZoom();
        }
    }
}

```



```

/**
 * Display initial board
 */
private void displayInitialBoard(Board board) {
    // Hide welcome pane, show board container
    welcomePane.setVisible(false);
    welcomePane.setManaged(false);
    boardContainer.setVisible(true);
    boardContainer.setManaged(true);
    boardCanvas.setVisible(true);

    // Initialize colors
    initializePieceColors(board);

    // Initialize canvas and zoom controller if not yet initialized
    if (animation == null) {
        initializeCanvas();
    }

    // Set appropriate canvas size based on board size
    adjustCanvasSize(board);

    // Draw the board
    drawBoard(board);

    // Keep animation controls hidden initially
    animationControls.setVisible(false);

    // Update move info
    moveLabel.setText("Initial Board Configuration");
    statsLabel.setText("");
}

@FXML
private void handleToggleCompound() {
    isCompound = !isCompound;
    updateCompoundButtonText();
}
private void resetBoardState() {
    // Reset solution and board state
    this.solution = null;
    this.boardStates = null;
    this.moves = null;
    this.currentStateIndex = 0;
    this.nodesExamined = 0;

    // Reset animation state
    if (animation != null) {
        animation.stop();
        isPlaying = false;
    }
}

```

```

        // Reset UI controls
        if (playButton != null) playButton.setDisable(false);
        if (pauseButton != null) pauseButton.setDisable(true);
        if (resetButton != null) resetButton.setDisable(false);

        // Reset labels
        if (moveLabel != null) moveLabel.setText("");
        if (statsLabel != null) statsLabel.setText("");

        // Reset zoom if controller exists
        if (zoomController != null) {
            zoomController.resetZoom();
            updateZoomPercentLabel();
        }
    }

    /**
     * Handle browse button click
     */
    @FXML
    private void handleBrowse() {
        FileChooser fileChooser = new FileChooser();
        fileChooser.setTitle("Select Rush Hour Configuration File");
        fileChooser.getExtensionFilters().add(
            new FileChooser.ExtensionFilter("Text Files", "*.txt")
        );

        // Set initial directory to test/input if it exists
        File testInputDir = new File("test/input");
        if (testInputDir.exists() && testInputDir.isDirectory()) {
            fileChooser.setInitialDirectory(testInputDir);
        }

        File file =
fileChooser.showOpenDialog(rootContainer.getScene().getWindow());
        if (file != null) {
            filePathField.setText(file.getAbsolutePath());
            handleLoadFile(); // Auto-load when file is selected
        }
    }

    /**
     * Handle load file button click
     */
    @FXML
    private void handleLoadFile() {
        String filePath = filePathField.getText();
        if (filePath.isEmpty()) {
            updateStatus("Error: Please select a file first", true);
            return;
        }

        resetBoardState();
    }

```

```

        updateStatus("Loading file...");
        progressBar.setVisible(true);

progressBar.setProgress(ProgressIndicator.INDETERMINATE_PROGRESS);

        // Run loading in background
        new Thread(() -> {
            try {
                currentBoard = Board.readFromFile(filePath);

                Platform.runLater(() -> {
                    progressBar.setVisible(false);

                    // Show zoom controls for large boards
                    boolean largeBoard = currentBoard.getWidth() > 10 ||
currentBoard.getHeight() > 10;
                    if (zoomControls != null) {
                        zoomControls.setVisible(largeBoard);
                        zoomControls.setManaged(largeBoard);
                    }

                    // Check if primary piece is aligned with exit
                    if (!currentBoard.isPrimaryPieceAlignedWithExit()) {
                        updateStatus("Warning: Primary piece and exit
are not aligned. The puzzle may not be solvable.", true);
                    } else {
                        updateStatus("Board loaded successfully!");
                    }

                    // Enable solve button
                    solveButton.setDisable(false);

                    // Show the board
                    displayInitialBoard(currentBoard);
                });

            } catch (IOException e) {
                Platform.runLater(() -> {
                    progressBar.setVisible(false);
                    updateStatus("Error loading file: " +
e.getMessage(), true);
                    currentBoard = null;
                    solveButton.setDisable(true);
                });
            }
        }).start();
    }

    /**
     * Handle parse text button click - Uses FileParser for consistent
    parsing
    */

```

```

@FXML
private void handleParseText() {
    String configText = configTextArea.getText();
    if (configText.trim().isEmpty()) {
        updateStatus("Error: Please enter a configuration", true);
        return;
    }

    // Make sure we have at least 3 lines (dimensions, pieces count,
    and at least one board row)
    String[] lines = configText.split("\\n");
    if (lines.length < 3) {
        updateStatus("Error: Configuration must have at least 3
lines", true);
        return;
    }

    // Parse dimensions to determine board size
    String[] dimensions = lines[0].trim().split("\\s+");
    if (dimensions.length != 2) {
        updateStatus("Error: First line must contain exactly 2
integers for board dimensions", true);
        return;
    }

    int rows, cols;
    try {
        rows = Integer.parseInt(dimensions[0]);
        cols = Integer.parseInt(dimensions[1]);
    } catch (NumberFormatException e) {
        updateStatus("Error: Invalid board dimensions, must be
integers", true);
        return;
    }

    // Validate minimum board size (at least 1x2 or 2x1)
    if (rows < 1 || cols < 1 || (rows == 1 && cols == 1)) {
        updateStatus("Error: Board size must be at least 1x2 or 2x1.
Current size: " + rows + "x" + cols, true);
        return;
    }

    resetBoardState();
    updateStatus("Parsing configuration...");
    progressBar.setVisible(true);

    progressBar.setProgress(ProgressIndicator.INDETERMINATE_PROGRESS);

    // Run parsing in background
    new Thread(() -> {
        try {
            // Save to temporary file and parse using FileParser
            File tempFile = File.createTempFile("rushHour", ".txt");

```

```

        try (FileWriter writer = new FileWriter(tempFile)) {
            writer.write(configText);
        }

        // Use the same FileParser logic used in
Board.readFromFile
        currentBoard =
Board.readFromFile(tempFile.getAbsolutePath());
        tempFile.delete();

        Platform.runLater(() -> {
            progressBar.setVisible(false);

            // Show zoom controls for large boards
            boolean largeBoard = currentBoard.getWidth() > 10 ||
currentBoard.getHeight() > 10;
            if (zoomControls != null) {
                zoomControls.setVisible(largeBoard);
                zoomControls.setManaged(largeBoard);
            }

            // Check if primary piece is aligned with exit
            if (!currentBoard.isPrimaryPieceAlignedWithExit()) {
                updateStatus("Warning: Primary piece and exit
are not aligned. The puzzle may not be solvable.", true);
            } else {
                updateStatus("Configuration parsed
successfully!");
            }

            // Enable solve button
            solveButton.setDisable(false);

            // Show the board
            displayInitialBoard(currentBoard);
        });

    } catch (Exception e) {
        Platform.runLater(() -> {
            progressBar.setVisible(false);
            updateStatus("Error parsing configuration: " +
e.getMessage(), true);
            currentBoard = null;
            solveButton.setDisable(true);
        });
    }
}).start();
}

/**
 * Handle create matrix button click
 */
@FXML

```

```

private void handleCreateMatrix() {
    try {
        FXMLLoader loader = new
FXMLLoader(getClass().getResource("/gui/fxml/MatrixInputWindow.fxml"));
        Parent root = loader.load();

        if (root == null) {
            updateStatus("Error loading matrix input window", true);
            return;
        }

        MatrixInputWindowController controller =
loader.getController();

        int rows = rowSpinner.getValue();
        int cols = colSpinner.getValue();
        int numPieces = piecesSpinner.getValue();

        Stage popupStage = new Stage();
        popupStage.setTitle("Fill Matrix");
        popupStage.initModality(Modality.APPLICATION_MODAL);
        popupStage.setScene(new Scene(root));
        controller.initMatrix(rows, cols, popupStage);

        popupStage.showAndWait();

        // After closing, retrieve matrix
        String finalString = controller.getFinalString();
        if (finalString != null && !finalString.isEmpty()) {
            handleParseMatrix(rows, cols, numPieces, finalString);
        }
    } catch (IOException e) {
        updateStatus("Error creating matrix: " + e.getMessage(),
true);
        e.printStackTrace();
    }
}

private void handleParseMatrix(int rows, int cols, int numPieces,
String configText) {
    String finalConfigText = rows + " " + cols + "\n" + numPieces +
"\n" + configText;

    System.out.println("Matrix configuration: " + finalConfigText);
    if (finalConfigText.trim().isEmpty()) {
        updateStatus("Error: Please enter a configuration", true);
        return;
    }

    resetBoardState();
    updateStatus("Parsing matrix configuration...");
    progressBar.setVisible(true);
}

```

```

progressBar.setProgress(ProgressIndicator.INDETERMINATE_PROGRESS);

    // Run parsing in background
    new Thread(() -> {
        try {
            // Save to temporary file and parse using FileParser
            File tempFile = File.createTempFile("rushHour", ".txt");
            try (FileWriter writer = new FileWriter(tempFile)) {
                writer.write(finalConfigText);
            }

            // Use the same FileParser logic used in
Board.readFromFile
            currentBoard =
Board.readFromFile(tempFile.getAbsolutePath());
            tempFile.delete();

            Platform.runLater(() -> {
                progressBar.setVisible(false);

                // Show zoom controls for large boards
                boolean largeBoard = currentBoard.getWidth() > 10 ||
currentBoard.getHeight() > 10;
                if (zoomControls != null) {
                    zoomControls.setVisible(largeBoard);
                    zoomControls.setManaged(largeBoard);
                }

                // Check if primary piece is aligned with exit
                if (!currentBoard.isPrimaryPieceAlignedWithExit()) {
                    updateStatus("Warning: Primary piece and exit
are not aligned. The puzzle may not be solvable.", true);
                } else {
                    updateStatus("Matrix configuration parsed
successfully!");
                }

                // Enable solve button
                solveButton.setDisable(false);

                // Show the board
                displayInitialBoard(currentBoard);
            });

        } catch (Exception e) {
            Platform.runLater(() -> {
                progressBar.setVisible(false);
                updateStatus("Error parsing matrix: " +
e.getMessage(), true);
                currentBoard = null;
                solveButton.setDisable(true);
            });
        }
    });
}

```

```

        }).start();
    }

    /**
     * Handle solve button click
     */
    @FXML
    private void handleSolve() {
        if (currentBoard == null) {
            updateStatus("Error: Please load a board configuration
first", true);
            return;
        }

        String algorithm = algorithmComboBox.getValue();
        String heuristic = heuristicComboBox.getValue();

        // Disable UI during solving
        solveButton.setDisable(true);
        progressBar.setVisible(true);

progressBar.setProgress(ProgressIndicator.INDETERMINATE_PROGRESS);
        updateStatus("Solving puzzle using " + algorithm + "...");

        // Run solving in background thread
        new Thread(() -> {
            try {
                Solver solver = new Solver();
                Solution solution = null;

                long startTime = System.currentTimeMillis();

                // Select algorithm based on combo box selection
                if (algorithm.contains("UCS")) {
                    solution = solver.solveUCS(currentBoard,
isCompound);
                } else if (algorithm.contains("Dijkstra")) {
                    solution = solver.solveDijkstra(currentBoard,
isCompound);
                } else if (algorithm.contains("Greedy")) {
                    solution = solver.solveGreedy(currentBoard,
heuristic, isCompound);
                } else if (algorithm.contains("A*")) {
                    solution = solver.solveAStar(currentBoard,
heuristic, isCompound);
                } else if (algorithm.contains("Beam")) {
                    solution = solver.solveBeam(currentBoard, heuristic,
isCompound);
                } else if (algorithm.contains("Iterative")) {
                    solution = solver.solveIDAStar(currentBoard,
heuristic, isCompound);
                }
            }

```



```

        long endTime = System.currentTimeMillis();
        final long executionTime = endTime - startTime;

        final Solution finalSolution = solution;

        // Update UI on JavaFX thread
        Platform.runLater(() -> {
            progressBar.setVisible(false);
            solveButton.setDisable(false);

            // Store the execution time and solution as class
members
            this.executionTime = executionTime;
            this.solution = finalSolution;

            if (finalSolution != null) {
                displaySolution(finalSolution, algorithm,
heuristic, executionTime);
            } else {
                // Even when no solution is found, show examined
nodes and time
                int nodesExamined =
solver.getLastNodesExamined();
                updateStatus("No solution found for this
configuration. Examined " + nodesExamined + " states in " +
executionTime + " ms.", true);

                // Store the nodes examined for saving
                this.nodesExamined = nodesExamined;

                // Enable save button to allow saving the
no-solution result
                animationControls.setVisible(true);
                playButton.setDisable(true);
                pauseButton.setDisable(true);
                resetButton.setDisable(true);

                // Update stats label with the info
                statsLabel.setText(String.format(
                    "Algorithm: %s | Heuristic: %s | States
Examined: %d | Time: %d ms | No Solution Found",
                    algorithm,
                    (algorithm.contains("UCS") ||
algorithm.contains("Dijkstra")) ? "-" : heuristic,
                    nodesExamined,
                    executionTime
                ));
            }
        });

    } catch (Exception e) {
        Platform.runLater(() -> {
            progressBar.setVisible(false);

```

```

        solveButton.setDisable(false);
        updateStatus("Error solving puzzle: " +
e.getMessage(), true);
        e.printStackTrace();
    });
    }
    }).start();
}

private void displaySolution(Solution solution, String algorithm,
String heuristic, long executionTime) {
    this.solution = solution;
    this.boardStates = solution.getStates();
    this.moves = solution.getMoves();
    this.currentStateIndex = 0;

    // Initialize canvas animation if needed
    if (animation == null) {
        initializeCanvas();
    }

    // Show animation controls
    animationControls.setVisible(true);

    // Reset controls state
    playButton.setDisable(false);
    pauseButton.setDisable(true);

    // Set appropriate canvas size based on board size
    adjustCanvasSize(boardStates.get(0));

    // Draw initial state
    drawBoard(boardStates.get(0));

    // For UCS and Dijkstra, display "-" as the heuristic since they
    don't use heuristics
    String displayHeuristic = heuristic;
    if (algorithm.contains("UCS") || algorithm.contains("Dijkstra"))
    {
        displayHeuristic = "-";
    }

    // Update stats and info
    statsLabel.setText(String.format(
        "Algorithm: %s | Heuristic: %s | States Examined: %d |
Moves: %d | Time: %d ms",
        algorithm,
        displayHeuristic,
        solution.getStatesExamined(),
        moves.size(),
        executionTime
    ));
}

```

```

        updateMoveInfo();
        updateStatus("Solution found! Use controls to view the steps.");
    }

    /**
     * Initialize colors for pieces
     */
    private void initializePieceColors(Board board) {
        pieceColors.clear();

        Color[] colors = {
            Color.CRIMSON,          // Primary piece (P)
            Color.ROYALBLUE,
            Color.LIMEGREEN,
            Color.GOLD,
            Color.DARKORANGE,
            Color.MEDIUMPURPLE,
            Color.DEEPPINK,
            Color.TURQUOISE,
            Color.CORAL,
            Color.DARKGREEN,
            Color.INDIGO,
            Color.MAROON,
            Color.OLIVE,
            Color.TEAL,
            Color.NAVY,
            Color.CHOCOLATE
        };

        // Assign colors to pieces
        int colorIndex = 1; // Start from 1, reserve 0 for primary piece
        for (Piece piece : board.getPieces()) {
            if (piece.getId() == 'P') {
                pieceColors.put('P', colors[0]); // Red for primary
            } else {
                pieceColors.put(piece.getId(), colors[colorIndex %
                    colors.length]);
                colorIndex++;
            }
        }
    }

    /**
     * Draw the board on canvas with zoom support
     */
    private void drawBoard(Board board) {
        GraphicsContext gc = boardCanvas.getGraphicsContext2D();
        int width = board.getWidth();
        int height = board.getHeight();

        // Calculate required canvas size
        double canvasWidth = width * CELL_SIZE + PADDING * 2;

```

```

        double canvasHeight = height * CELL_SIZE + PADDING * 2;

        // If canvas size doesn't match, update it
        if (boardCanvas.getWidth() < canvasWidth ||
boardCanvas.getHeight() < canvasHeight) {
            boardCanvas.setWidth(canvasWidth);
            boardCanvas.setHeight(canvasHeight);
        }

        // Clear canvas
        gc.clearRect(0, 0, boardCanvas.getWidth(),
boardCanvas.getHeight());

        // Draw board background with rounded corners
        gc.setFill(Color.rgb(240, 240, 240, 0.9));
        gc.fillRoundRect(PADDING - 10, PADDING - 10,
            width * CELL_SIZE + 20, height * CELL_SIZE + 20,
15, 15);

        // Draw grid background
        gc.setFill(Color.LIGHTGRAY);
        gc.fillRoundRect(PADDING, PADDING, width * CELL_SIZE, height *
CELL_SIZE, 10, 10);

        // Draw grid lines
        gc.setStroke(Color.DARKGRAY);
        gc.setLineWidth(1);
        for (int i = 0; i <= width; i++) {
            gc.strokeLine(PADDING + i * CELL_SIZE, PADDING,
                PADDING + i * CELL_SIZE, PADDING + height *
CELL_SIZE);
        }
        for (int i = 0; i <= height; i++) {
            gc.strokeLine(PADDING, PADDING + i * CELL_SIZE,
                PADDING + width * CELL_SIZE, PADDING + i *
CELL_SIZE);
        }

        // Draw exit
        Position exitPos = board.getExitPosition();
        gc.setFill(Color.LIGHTGREEN);
        gc.setStroke(Color.DARKGREEN);
        gc.setLineWidth(3);

        switch (board.getExitSide()) {
            case TOP:
                gc.fillRect(PADDING + exitPos.col * CELL_SIZE, PADDING -
20, CELL_SIZE, 20);
                gc.strokeRect(PADDING + exitPos.col * CELL_SIZE, PADDING
- 20, CELL_SIZE, 20);
                break;
            case BOTTOM:
                gc.fillRect(PADDING + exitPos.col * CELL_SIZE, PADDING +

```

```

height * CELL_SIZE,
                                CELL_SIZE, 20);
        gc.strokeRect(PADDING + exitPos.col * CELL_SIZE, PADDING
+ height * CELL_SIZE,
                                CELL_SIZE, 20);
        break;
    case LEFT:
        gc.fillRect(PADDING - 20, PADDING + exitPos.row *
CELL_SIZE, 20, CELL_SIZE);
        gc.strokeRect(PADDING - 20, PADDING + exitPos.row *
CELL_SIZE, 20, CELL_SIZE);
        break;
    case RIGHT:
        gc.fillRect(PADDING + width * CELL_SIZE, PADDING +
exitPos.row * CELL_SIZE,
                                20, CELL_SIZE);
        gc.strokeRect(PADDING + width * CELL_SIZE, PADDING +
exitPos.row * CELL_SIZE,
                                20, CELL_SIZE);
        break;
    }

    // Draw exit arrow
    drawExitArrow(gc, board);

    // Draw pieces
    for (Piece piece : board.getPieces()) {
        drawPiece(gc, piece);
    }

    // Draw mini-map for large boards
    if (board.getWidth() > 12 || board.getHeight() > 12) {
        drawMiniMap(gc, board);
    }
}

/**
 * Draw a mini-map in the corner for large boards
 */
private void drawMiniMap(GraphicsContext gc, Board board) {
    int width = board.getWidth();
    int height = board.getHeight();

    // Mini-map settings
    double miniMapSize = 150;
    double miniMapRatio = Math.min(miniMapSize / width, miniMapSize
/ height);
    double miniMapWidth = width * miniMapRatio;
    double miniMapHeight = height * miniMapRatio;
    double miniMapX = boardCanvas.getWidth() - miniMapWidth - 20;
    double miniMapY = 20;

    // Draw mini-map background

```

```

        gc.setFill(Color.rgb(30, 30, 30, 0.7));
        gc.fillRoundRect(miniMapX - 5, miniMapY - 5, miniMapWidth + 10,
miniMapHeight + 10, 10, 10);

        // Draw mini-map grid
        gc.setFill(Color.rgb(60, 60, 60, 0.8));
        gc.fillRect(miniMapX, miniMapY, miniMapWidth, miniMapHeight);

        // Draw pieces in mini-map
        for (Piece piece : board.getPieces()) {
            List<Position> positions = piece.getPositions();

            // Calculate bounding box
            int minRow = positions.stream().mapToInt(p ->
p.row).min().orElse(0);
            int maxRow = positions.stream().mapToInt(p ->
p.row).max().orElse(0);
            int minCol = positions.stream().mapToInt(p ->
p.col).min().orElse(0);
            int maxCol = positions.stream().mapToInt(p ->
p.col).max().orElse(0);

            // Draw piece in mini-map
            Color color = pieceColors.get(piece.getId());
            if (color == null) {
                color = Color.GRAY;
            }

            gc.setFill(color);
            gc.fillRect(
                miniMapX + minCol * miniMapRatio,
                miniMapY + minRow * miniMapRatio,
                (maxCol - minCol + 1) * miniMapRatio,
                (maxRow - minRow + 1) * miniMapRatio
            );
        }

        // Draw view port if zoomed
        if (zoomController != null && zoomController.getZoomScale() >
1.0) {
            double viewportWidth = miniMapWidth /
zoomController.getZoomScale();
            double viewportHeight = miniMapHeight /
zoomController.getZoomScale();

            // Calculate center of viewport based on translation
            double centerX = miniMapX + miniMapWidth / 2;
            double centerY = miniMapY + miniMapHeight / 2;

            gc.setStroke(Color.WHITE);
            gc.setLineWidth(2);
            gc.strokeRect(
                centerX - viewportWidth / 2,

```

```

        centerY - viewportHeight / 2,
        viewportWidth,
        viewportHeight
    );
}
}

/**
 * Draw exit arrow
 */
private void drawExitArrow(GraphicsContext gc, Board board) {
    Position exitPos = board.getExitPosition();
    gc.setFill(Color.DARKGREEN);

    double arrowSize = 15;

    switch (board.getExitSide()) {
        case TOP:
            // Draw arrow pointing up
            double xTop = PADDING + exitPos.col * CELL_SIZE +
CELL_SIZE / 2;
            double yTop = PADDING - 10;
            gc.fillPolygon(
                new double[] {xTop, xTop - arrowSize, xTop +
arrowSize},
                new double[] {yTop - arrowSize, yTop, yTop},
                3
            );
            break;
        case BOTTOM:
            // Draw arrow pointing down
            double xBottom = PADDING + exitPos.col * CELL_SIZE +
CELL_SIZE / 2;
            double yBottom = PADDING + board.getHeight() * CELL_SIZE
+ 10;
            gc.fillPolygon(
                new double[] {xBottom, xBottom - arrowSize, xBottom
+ arrowSize},
                new double[] {yBottom + arrowSize, yBottom,
yBottom},
                3
            );
            break;
        case LEFT:
            // Draw arrow pointing left
            double xLeft = PADDING - 10;
            double yLeft = PADDING + exitPos.row * CELL_SIZE +
CELL_SIZE / 2;
            gc.fillPolygon(
                new double[] {xLeft - arrowSize, xLeft, xLeft},
                new double[] {yLeft, yLeft - arrowSize, yLeft +
arrowSize},
                3
            );
            break;
        case RIGHT:
            // Draw arrow pointing right
            double xRight = PADDING + board.getWidth() * CELL_SIZE -
CELL_SIZE / 2;
            double yRight = PADDING + exitPos.row * CELL_SIZE +
CELL_SIZE / 2;
            gc.fillPolygon(
                new double[] {xRight, xRight + arrowSize, xRight -
arrowSize},
                new double[] {yRight, yRight - arrowSize, yRight +
arrowSize},
                3
            );
            break;
    }
}

```

```

        );
        break;
    case RIGHT:
        // Draw arrow pointing right
        double xRight = PADDING + board.getWidth() * CELL_SIZE +
10;
        double yRight = PADDING + exitPos.row * CELL_SIZE +
CELL_SIZE / 2;
        gc.fillPolygon(
            new double[] {xRight + arrowSize, xRight, xRight},
            new double[] {yRight, yRight - arrowSize, yRight +
arrowSize},
            3
        );
        break;
    }
}

/**
 * Draw piece with 3D effect
 */
private void drawPiece(GraphicsContext gc, Piece piece) {
    Color color = pieceColors.get(piece.getId());
    if (color == null) {
        color = Color.GRAY; // Default color if not found
    }

    gc.setFill(color);
    gc.setStroke(color.darker());
    gc.setLineWidth(3);

    List<Position> positions = piece.getPositions();

    // Calculate bounding box
    int minRow = positions.stream().mapToInt(p ->
p.row).min().orElse(0);
    int maxRow = positions.stream().mapToInt(p ->
p.row).max().orElse(0);
    int minCol = positions.stream().mapToInt(p ->
p.col).min().orElse(0);
    int maxCol = positions.stream().mapToInt(p ->
p.col).max().orElse(0);

    // Draw piece as rounded rectangle with shadow
    double x = PADDING + minCol * CELL_SIZE + 5;
    double y = PADDING + minRow * CELL_SIZE + 5;
    double width = (maxCol - minCol + 1) * CELL_SIZE - 10;
    double height = (maxRow - minRow + 1) * CELL_SIZE - 10;

    // Draw shadow
    gc.setFill(Color.rgb(0, 0, 0, 0.3));
    gc.fillRoundRect(x + 3, y + 3, width, height, 15, 15);

```



```

        // Draw piece
        gc.setFill(color);
        gc.fillRoundRect(x, y, width, height, 15, 15);
        gc.setStroke(color.darker());
        gc.strokeRoundRect(x, y, width, height, 15, 15);

        // Add highlight effect
        gc.setFill(Color.rgb(255, 255, 255, 0.3));
        gc.fillRoundRect(x + 3, y + 3, width - 6, height/2 - 6, 10, 10);

        // Draw piece ID
        gc.setFill(Color.WHITE);
        gc.setFont(javafx.scene.text.Font.font("Arial",
javafx.scene.text.FontWeight.BOLD, 24));
        double textX = x + width / 2 - 10;
        double textY = y + height / 2 + 8;
        gc.fillText(String.valueOf(piece.getId()), textX, textY);
    }

    /**
     * Update move information
     */
    private void updateMoveInfo() {
        if (currentStateIndex == 0) {
            moveLabel.setText("Initial Board Configuration");
        } else if (currentStateIndex > moves.size()) {
            moveLabel.setText("Puzzle Solved!");
        } else {
            Move currentMove = moves.get(currentStateIndex - 1);
            moveLabel.setText(String.format("Move %d of %d: %s",
currentMove.toString()));
        }
    }

    /**
     * Update status message
     */
    private void updateStatus(String message) {
        updateStatus(message, false);
    }

    /**
     * Update status message with error flag
     */
    private void updateStatus(String message, boolean isError) {
        statusText.setText(message);
        if (isError) {
            statusText.setStyle("-fx-fill: #e74c3c;"); // Red for errors
        } else {
            statusText.setStyle("-fx-fill: #80d8ff;"); // Normal color
        }
    }
}

```

```

/**
 * Handle play button click
 */
@FXML
private void handlePlay() {
    if (!isPlaying && currentStateIndex < boardStates.size() - 1) {
        animation.play();
        isPlaying = true;
        playButton.setDisable(true);
        pauseButton.setDisable(false);
    }
}

/**
 * Handle pause button click
 */
@FXML
private void handlePause() {
    pauseAnimation();
}

/**
 * Pause animation
 */
private void pauseAnimation() {
    if (isPlaying) {
        animation.pause();
        isPlaying = false;
        playButton.setDisable(false);
        pauseButton.setDisable(true);
    }
}

/**
 * Handle reset button click
 */
@FXML
private void handleReset() {
    pauseAnimation();
    currentStateIndex = 0;
    drawBoard(boardStates.get(0));
    updateMoveInfo();
    playButton.setDisable(false);
    updateStatus("Reset to initial configuration");
}

/**
 * Handle save button click
 */
@FXML
private void handleSave() {
    // Make sure we have either a solution or at least examined some

```

```

nodes
    if (solution == null && nodesExamined == 0) {
        updateStatus("No results to save", true);
        return;
    }

    FileChooser fileChooser = new FileChooser();
    fileChooser.setTitle("Save Solution");
    fileChooser.getExtensionFilters().add(
        new FileChooser.ExtensionFilter("Text Files", "*.txt")
    );

    // Use algorithm name in suggested filename
    String algorithm = algorithmComboBox.getValue().split("
")[0].toLowerCase();
    String resultType = (solution != null) ? "solution" :
    "no_solution";
    fileChooser.setInitialFileName("rush_hour_" + algorithm + "_" +
    resultType + ".txt");

    // Try to set initial directory to test/output if it exists
    File outputDir = new File("test/output");
    if (!outputDir.exists()) {
        outputDir.mkdirs();
    }
    fileChooser.setInitialDirectory(outputDir);

    File file =
    fileChooser.showSaveDialog(rootContainer.getScene().getWindow());
    if (file != null) {
        try {
            saveSolution(file);
            if (solution != null) {
                updateStatus("Solution saved to: " +
    file.getName());
            } else {
                updateStatus("No solution results saved to: " +
    file.getName());
            }
        } catch (IOException e) {
            updateStatus("Error saving solution: " + e.getMessage(),
    true);
        }
    }

    /**
     * Save solution to file
     */
    private void saveSolution(File file) throws IOException {
        try (FileWriter writer = new FileWriter(file)) {
            // For UCS and Dijkstra, display "-" as the heuristic
            String displayHeuristic = heuristicComboBox.getValue();

```

```

        if (algorithmComboBox.getValue().contains("UCS") ||
            algorithmComboBox.getValue().contains("Dijkstra")) {
            displayHeuristic = "-";
        }

        writer.write("=== Rush Hour Solution ===\n");
        writer.write("Algorithm: " + algorithmComboBox.getValue() +
"\n");

        // Write heuristic info if applicable
        writer.write("Heuristic: " + displayHeuristic + "\n");

        // Check if we have a solution
        if (solution != null) {
            writer.write("Number of states examined: " +
solution.getStatesExamined() + "\n");
            writer.write("Number of moves: " +
solution.getMoves().size() + "\n");
        } else {
            // No solution found case
            writer.write("Number of states examined: " +
nodesExamined + "\n");
            writer.write("Number of moves: NO SOLUTION FOUND\n");
        }

        writer.write("Execution time: " + executionTime + " ms\n");
        writer.write("\n");

        // Only write solution steps if a solution was found
        if (solution != null) {
            writer.write("Papan Awal\n");
            writeBoard(writer, boardStates.get(0));
            writer.write("\n");

            for (int i = 0; i < moves.size(); i++) {
                writer.write("Gerakan " + (i + 1) + ": " +
moves.get(i) + "\n");
                writeBoard(writer, boardStates.get(i + 1));
                writer.write("\n");
            }

            writer.write("[Primary piece has reached the exit!]\n");
        } else {
            // Write initial board only
            writer.write("Papan Awal\n");
            writeBoard(writer, currentBoard);
            writer.write("\n");
            writer.write("[No solution found for this
configuration]\n");
        }
    }
}

```

```

/**
 * Write board to file
 */
private void writeBoard(FileWriter writer, Board board) throws
IOException {
    int width = board.getWidth();
    int height = board.getHeight();

    // Write top exit if exists
    Exit exitSide = board.getExitSide();
    Position exitPosition = board.getExitPosition();

    if (exitSide == Exit.TOP) {
        for (int j = 0; j < width; j++) {
            if (j == exitPosition.col) {
                writer.write("K");
            } else {
                writer.write(" ");
            }
        }
        writer.write("\n");
    }

    // Write board content
    for (int i = 0; i < height; i++) {
        // Write left exit
        if (exitSide == Exit.LEFT && i == exitPosition.row) {
            writer.write("K");
        }

        // Write board cells
        for (int j = 0; j < width; j++) {
            writer.write(board.getGridAt(i, j));
        }

        // Write right exit
        if (exitSide == Exit.RIGHT && i == exitPosition.row) {
            writer.write("K");
        }

        writer.write("\n");
    }

    // Write bottom exit if exists
    if (exitSide == Exit.BOTTOM) {
        for (int j = 0; j < width; j++) {
            if (j == exitPosition.col) {
                writer.write("K");
            } else {
                writer.write(" ");
            }
        }
        writer.write("\n");
    }
}

```

```

        }
    }

    /**
     * Handle pause song button click
     */
    @FXML
    private void handlePauseSong() {
        if (mediaPlayer != null) {
            MediaPlayer.Status status = mediaPlayer.getStatus();
            if (status == MediaPlayer.Status.PLAYING) {
                mediaPlayer.pause();
                pauseSongButton.setText("Play");
            }
            else if (status == MediaPlayer.Status.PAUSED || status ==
MediaPlayer.Status.STOPPED) {
                if (songComboBox.getValue() != null &&
!songComboBox.getValue().equals(playedSong)) {
                    mediaPlayer.stop();
                    playBackgroundMusic(songComboBox.getValue());
                    pauseSongButton.setText("Pause");
                    playedSong = songComboBox.getValue();
                    return;
                }
                mediaPlayer.play();
                pauseSongButton.setText("Pause");
            }
        }
    }

    /**
     * Play background music
     */
    public void playBackgroundMusic(String filename) {
        try {
            System.out.println("Playing background music: " + filename);
            if (mediaPlayer != null) {
                mediaPlayer.stop();
            }

            File file = new File(SONG_PATH + filename);
            if (!file.exists()) {
                System.err.println("Audio file not found: " +
file.getAbsolutePath());
                return;
            }

            Media media = new Media(file.toURI().toString());
            mediaPlayer = new MediaPlayer(media);
            mediaPlayer.setCycleCount(MediaPlayer.INDEFINITE); // loop
the song
            mediaPlayer.play();
            playedSong = filename;
        }
    }

```

```

        pauseSongButton.setText("Pause");
    } catch (Exception e) {
        System.err.println("Error playing music: " +
e.getMessage());
        e.printStackTrace();
    }
}
}

```

3.2.15 *MatrixInputWindowController.java*

```

package gui.controllers;

import javafx.fxml.FXML;
import javafx.scene.control.Button;
import javafx.scene.control.TextField;
import javafx.scene.control.TextFormatter;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;
import java.util.ArrayList;
import java.util.List;
import javafx.scene.control.TextField;
import javafx.scene.control.TextFormatter.Change;
import javafx.scene.input.MouseEvent;
import javafx.geometry.Pos;
import javafx.scene.control.Alert;

public class MatrixInputWindowController {
    @FXML private GridPane matrixGrid;
    @FXML private Button okButton;

    private int rows;
    private int cols;
    private List<TextField> matrixCells = new ArrayList<>();
    private Stage stage;

    public TextField selectedExitCell = null;
    public final String EXIT_STYLE = "-fx-background-color: #ffe082;";
    // yellow highlight
    public final String NORMAL_STYLE = ""; // reset to default
    public String EXIT_POS = "";
    public int EXIT_ROW;
    public int EXIT_COL;

    String FinalString = "";

    public void initialize() {
        okButton.setOnAction(e -> {
            System.err.println("EXIT_POS " + EXIT_POS);

```

```

System.err.println("EXIT_ROW " + EXIT_ROW);
System.err.println("EXIT_COL " + EXIT_COL);

if (selectedExitCell == null) {
    Alert alert = new Alert(Alert.AlertType.WARNING);
    alert.setTitle("Missing Exit");
    alert.setHeaderText("No Exit Selected");
    alert.setContentText("Please click one of the border cells
to mark it as the exit (K).");
    alert.showAndWait();
    return;
}

if (EXIT_POS == "TOP") {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < EXIT_COL - 1; i++) {
        sb.append(" ");
    }
    sb.append("K");
    for (int i = EXIT_COL + 1; i < cols; i++) {
        sb.append(" ");
    }
    sb.append('\n');

    for (int i = 0; i < rows; i++) {
        for (TextField cell : matrixCells.subList(i * cols, (i +
1) * cols)) {
            String text = cell.getText();
            if (text.isEmpty()) {
                sb.append(".");
            } else {
                sb.append(text);
            }
        }
        sb.append('\n');
    }
    FinalString = sb.toString();
}
else if (EXIT_POS == "BOTTOM") {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < rows; i++) {
        for (TextField cell : matrixCells.subList(i * cols, (i +
1) * cols)) {
            String text = cell.getText();
            if (text.isEmpty()) {
                sb.append(".");
            } else {
                sb.append(text);
            }
        }
        sb.append('\n');
    }
    for (int i = 0; i < EXIT_COL - 1; i++) {

```



```

        sb.append(" ");
    }
    sb.append("K");
    for (int i = EXIT_COL + 1; i < cols; i++) {
        sb.append(" ");
    }
    FinalString = sb.toString();
}
else if (EXIT_POS == "LEFT") {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < rows; i++) {
        if (i == EXIT_ROW) {
            sb.append("K");
        }
        else {
            sb.append(" ");
        }

        for (TextField cell : matrixCells.subList(i * cols, (i +
1) * cols)) {
            String text = cell.getText();
            if (text.isEmpty()) {
                sb.append(".");
            } else {
                sb.append(text);
            }
        }
        sb.append('\n');
    }
    FinalString = sb.toString();
}
else if (EXIT_POS == "RIGHT") {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < rows; i++) {
        for (TextField cell : matrixCells.subList(i * cols, (i +
1) * cols)) {
            String text = cell.getText();
            if (text.isEmpty()) {
                sb.append(".");
            } else {
                sb.append(text);
            }
        }
        if (i == EXIT_ROW) {
            sb.append("K");
        }
        sb.append('\n');
    }
    FinalString = sb.toString();
}

System.out.println("Final String: " + FinalString);

```

```

        // Exit is selected - close the window
        stage.close();
    });
}

private void handleExitClick(TextField cell, String exitPos, int
row, int col) {
    if (selectedExitCell != null) {
        selectedExitCell.setStyle("-fx-background-color: #eeeeee;");
        selectedExitCell.setText(""); // clear old exit
    }

    selectedExitCell = cell;
    selectedExitCell.setStyle(EXIT_STYLE);
    selectedExitCell.setText("K");

    EXIT_POS = exitPos;
    EXIT_ROW = row;
    EXIT_COL = col;
}

public void initMatrix(int rows, int cols, Stage stage) {
    int paddedRows = rows + 2;
    int paddedCols = cols + 2;

    this.rows = rows;
    this.cols = cols;
    this.stage = stage;

    matrixGrid.getChildren().clear();
    matrixCells.clear();

    for (int i = 0; i < paddedRows; i++) {
        for (int j = 0; j < paddedCols; j++) {

            // Skip corners
            if ((i == 0 || i == paddedRows - 1) && (j == 0 || j ==
paddedCols - 1)) {
                continue;
            }

            TextField cell = new TextField();
            cell.setPrefWidth(40);
            cell.setPrefHeight(40);
            cell.setAlignment(Pos.CENTER);
            cell.setTextFormatter(new TextFormatter<>(change -> {
                String newText = change.getControlNewText();
                if (newText.length() <= 1 && (newText.isEmpty() ||
                    newText.matches("[A-Z.]") ||
!newText.equals("K"))) {
                    return change;
                }
            }
            return null;

```

```

        ));

        // Top, bottom, left, or right edges (not corners): make
        them clickable exits
        boolean isTop = i == 0 && j > 0 && j < paddedCols - 1;
        boolean isBottom = i == paddedRows - 1 && j > 0 && j <
        paddedCols - 1;
        boolean isLeft = j == 0 && i > 0 && i < paddedRows - 1;
        boolean isRight = j == paddedCols - 1 && i > 0 && i <
        paddedRows - 1;
        if (isTop) {
            EXIT_POS = "TOP";
        } else if (isBottom) {
            EXIT_POS = "BOTTOM";
        } else if (isLeft) {
            EXIT_POS = "LEFT";
        } else if (isRight) {
            EXIT_POS = "RIGHT";
        }

        if (isTop || isBottom || isLeft || isRight) {
            cell.setEditable(false);
            cell.setStyle("-fx-background-color: #eeeeee;"); //
            lighter background
            final int exitRow = (isTop) ? i : i-1;
            final int exitCol = j;
            String exitPos = isTop ? "TOP" : isBottom ? "BOTTOM"
            : isLeft ? "LEFT" : "RIGHT";
            cell.setOnMouseClicked(e -> handleExitClick(cell,
            exitPos, exitRow, exitCol));
        } else {
            matrixCells.add(cell); // only center cells are
            actual matrix input
        }

        matrixGrid.add(cell, j, i);
    }
}

public String getFinalString() {
    return FinalString;
}
}

```

3.2.16 VisualizationController.java

```

package gui.controllers;

```

```

import cli.*;
import javafx.fxml.FXML;
import javafx.scene.canvas.Canvas;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.Slider;
import javafx.stage.FileChooser;
import javafx.stage.Stage;

import java.io.*;
import java.util.List;

/**
 * Controller for the visualization view that integrates with the
 * AnimationController
 * to display smooth animations of the Rush Hour puzzle solution
 */
public class VisualizationController {

    @FXML private Canvas boardCanvas;
    @FXML private Button playButton;
    @FXML private Button pauseButton;
    @FXML private Button resetButton;
    @FXML private Button saveButton;
    @FXML private Slider speedSlider;
    @FXML private Label statusLabel;
    @FXML private Label statsLabel;
    @FXML private Label currentMoveLabel;

    private Solution solution;
    private Board initialBoard;
    private AnimationController animationController;
    private String algorithmUsed;
    private String heuristicUsed;
    private long executionTime;

    /**
     * Initialize the controller with solution data
     */
    public void initialize(Solution solution, Board initialBoard, String
algorithm,
                        String heuristic, long executionTime) {
        this.solution = solution;
        this.initialBoard = initialBoard;
        this.algorithmUsed = algorithm;
        this.heuristicUsed = heuristic;
        this.executionTime = executionTime;

        // For UCS and Dijkstra, display "-" as the heuristic since they
        don't use heuristics
        String displayHeuristic = heuristic;
        if (algorithm.contains("UCS") || algorithm.contains("Dijkstra"))

```

```

{
    displayHeuristic = "-";
}

// Set stats label with algorithm info
statsLabel.setText(String.format(
    "Algorithm: %s | Heuristic: %s | States Examined: %d |
Moves: %d | Time: %d ms",
    algorithm,
    displayHeuristic,
    solution.getStatesExamined(),
    solution.getMoves().size(),
    executionTime
));

// Initialize the animation controller
animationController = new AnimationController(
    boardCanvas, currentMoveLabel, statusLabel, speedSlider,
    playButton, pauseButton, resetButton, solution
);

// Set button actions
playButton.setOnAction(e -> {
    animationController.play();
});

pauseButton.setOnAction(e -> {
    animationController.pause();
});

resetButton.setOnAction(e -> {
    animationController.reset();
});

saveButton.setOnAction(e -> {
    handleSave();
});

// Initial state
pauseButton.setDisable(true);
}

/**
 * Handle save button click
 */
@FXML
private void handleSave() {
    FileChooser fileChooser = new FileChooser();
    fileChooser.setTitle("Save Solution");
    fileChooser.getExtensionFilters().add(
        new FileChooser.ExtensionFilter("Text Files", "*.txt")
    );
    fileChooser.setInitialFileName("rush_hour_solution.txt");
}

```

```

        // Try to use test/output directory if it exists
        File outputDir = new File("test/output");
        if (outputDir.exists() && outputDir.isDirectory()) {
            fileChooser.setInitialDirectory(outputDir);
        }

        File file =
fileChooser.showSaveDialog(saveButton.getScene().getWindow());
        if (file != null) {
            try {
                saveSolution(file);
                showInfo("Solution saved successfully!");
            } catch (IOException e) {
                showError("Error saving solution: " + e.getMessage());
            }
        }
    }

    /**
     * Save solution to a file with visualization of primary piece
    exiting
    */
    private void saveSolution(File file) throws IOException {
        try (PrintWriter writer = new PrintWriter(new FileWriter(file)))
        {
            // Write header
            writer.println("Rush Hour Solution");
            writer.println("=====\n");

            // Write stats
            writer.println("Algorithm: " + algorithmUsed);

            // For UCS and Dijkstra, display "-" as the heuristic
            String displayHeuristic = heuristicUsed;
            if (algorithmUsed.contains("UCS") ||
algorithmUsed.contains("Dijkstra")) {
                displayHeuristic = "-";
            }

            if (displayHeuristic != null && !displayHeuristic.isEmpty()
&& !displayHeuristic.equals("-")) {
                writer.println("Heuristic: " + displayHeuristic);
            } else {
                writer.println("Heuristic: -");
            }

            writer.println("States examined: " +
solution.getStatesExamined());
            writer.println("Total moves: " +
solution.getMoves().size());
            writer.println("Execution time: " + executionTime + " ms");
            writer.println();
        }
    }
}

```

```

        // Write move sequence
        writer.println("Move Sequence:");
        writer.println(formatMoveSequence(solution.getMoves()));
        writer.println();

        // Write initial board
        writer.println("Papan Awal");
        writeBoard(writer, solution.getStates().get(0));

        // Write each move
        List<Move> moves = solution.getMoves();
        List<Board> states = solution.getStates();

        for (int i = 0; i < moves.size() - 1; i++) {
            writer.println();
            writer.println("Gerakan " + (i + 1) + ": " +
moves.get(i));
            writeBoard(writer, states.get(i + 1));
        }

        // Write final move with visualization of exiting primary
piece
        if (moves.size() > 0) {
            int lastIndex = moves.size() - 1;
            writer.println();
            writer.println("Gerakan " + (lastIndex + 1) + ": " +
moves.get(lastIndex));
            writeFinalBoard(writer, states.get(states.size() - 1),
moves.get(lastIndex));
        }

        writer.println();
        writer.println("[Primary piece has reached the exit!]");
    }
}

/**
 * Format move sequence as a compact string
 */
private String formatMoveSequence(List<Move> moves) {
    StringBuilder sb = new StringBuilder();
    int count = 0;

    for (Move move : moves) {
        sb.append(move.toString()).append(" ");
        count++;

        // Add newline every 16 moves for readability
        if (count % 16 == 0) {
            sb.append("\n");
        }
    }
}

```

```

        sb.append("(").append(moves.size()).append(" moves)");
        return sb.toString();
    }

    /**
     * Write board to file
     */
    private void writeBoard(PrintWriter writer, Board board) {
        int width = board.getWidth();
        int height = board.getHeight();

        // Write top exit if exists
        Exit exitSide = board.getExitSide();
        Position exitPosition = board.getExitPosition();

        if (exitSide == Exit.TOP) {
            for (int j = 0; j < width; j++) {
                if (j == exitPosition.col) {
                    writer.print("K");
                } else {
                    writer.print(" ");
                }
            }
            writer.println();
        }

        // Write board content
        for (int i = 0; i < height; i++) {
            // Left exit
            if (exitSide == Exit.LEFT && i == exitPosition.row) {
                writer.print("K");
            }

            // Board cells
            for (int j = 0; j < width; j++) {
                writer.print(board.getGridAt(i, j));
            }

            // Right exit
            if (exitSide == Exit.RIGHT && i == exitPosition.row) {
                writer.print("K");
            }

            writer.println();
        }

        // Write bottom exit if exists
        if (exitSide == Exit.BOTTOM) {
            for (int j = 0; j < width; j++) {
                if (j == exitPosition.col) {
                    writer.print("K");
                } else {

```



```

        writer.print(" ");
    }
}
writer.println();
}

/**
 * Write final board state with visualization of exiting primary
piece
 */
private void writeFinalBoard(PrintWriter writer, Board board, Move
lastMove) {
    int width = board.getWidth();
    int height = board.getHeight();

    Exit exitSide = board.getExitSide();
    Position exitPosition = board.getExitPosition();

    // Write top exit with exiting piece if applicable
    if (exitSide == Exit.TOP) {
        for (int j = 0; j < width; j++) {
            if (j == exitPosition.col) {
                writer.print("K");
                // Show primary piece outside
                if (lastMove.getPiece().getId() == 'P') {
                    writer.print("P");
                }
            } else {
                writer.print(" ");
            }
        }
        writer.println();
    }

    // Write board content
    for (int i = 0; i < height; i++) {
        // Left exit with exiting piece if applicable
        if (exitSide == Exit.LEFT && i == exitPosition.row) {
            // Show primary piece outside
            if (lastMove.getPiece().getId() == 'P') {
                writer.print("P");
            }
            writer.print("K");
        }

        // Board cells - don't show primary piece as it's now
outside
        for (int j = 0; j < width; j++) {
            char cell = board.getGridAt(i, j);
            if (cell == 'P' && lastMove.getPiece().getId() == 'P') {
                writer.print('.'); // Primary piece has exited
            } else {

```

```

        writer.print(cell);
    }
}

// Right exit with exiting piece if applicable
if (exitSide == Exit.RIGHT && i == exitPosition.row) {
    writer.print("K");
    // Show primary piece outside
    if (lastMove.getPiece().getId() == 'P') {
        writer.print("P");
    }
}

writer.println();
}

// Write bottom exit with exiting piece if applicable
if (exitSide == Exit.BOTTOM) {
    for (int j = 0; j < width; j++) {
        if (j == exitPosition.col) {
            writer.print("K");
            // Show primary piece outside
            if (lastMove.getPiece().getId() == 'P') {
                writer.print("P");
            }
        } else {
            writer.print(" ");
        }
    }
    writer.println();
}

}

/**
 * Show info message
 */
private void showInfo(String message) {
    javafx.scene.control.Alert alert = new
javafx.scene.control.Alert(
        javafx.scene.control.Alert.AlertType.INFORMATION);
    alert.setTitle("Information");
    alert.setHeaderText(null);
    alert.setContentText(message);
    alert.showAndWait();
}

/**
 * Show error message
 */
private void showError(String message) {
    javafx.scene.control.Alert alert = new
javafx.scene.control.Alert(
        javafx.scene.control.Alert.AlertType.ERROR);

```

```

        alert.setTitle("Error");
        alert.setHeaderText(null);
        alert.setContentText(message);
        alert.showAndWait();
    }

    /**
     * Clean up resources before closing
     */
    public void cleanup() {
        if (animationController != null) {
            animationController.cleanup();
        }
    }
}

```

3.2.17 ZoomController.java

```

package cli;

public enum Exit {
    TOP,
    BOTTOM,
    LEFT,
    RIGHT,
    NONE
}package gui.controllers;

import javafx.scene.canvas.Canvas;
import javafx.scene.control.Button;
import javafx.scene.control.Slider;
import javafx.scene.input.ScrollEvent;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.StackPane;
import javafx.geometry.Point2D;
import javafx.scene.Cursor;
import javafx.application.Platform;

/**
 * Enhanced controller for handling zoom and pan functionality on a
 * canvas
 * Works with existing FXML layout and supports more intuitive
 * dragging/panning
 * with special optimizations for very large boards (24x24 and above)
 */
public class ZoomController {
    private final Canvas canvas;
    private final StackPane canvasContainer;
}

```

```

private final Button zoomInButton;
private final Button zoomOutButton;
private final Slider zoomSlider;

// Zoom state
private double zoomScale = 1.0;
private double minZoom = 0.25;
private double maxZoom = 3.0;
private double zoomFactor = 1.1;

// Pan state
private double translateX = 0;
private double translateY = 0;
private Point2D lastMousePosition;
private boolean isPanning = false;

// Mouse states
private boolean leftButtonDragging = false;
private boolean hasResizedCanvas = false;

/**
 * Create a new zoom controller
 * @param canvas The canvas to be zoomed
 * @param canvasContainer The container holding the canvas
 * @param zoomInButton Button for zooming in (optional, can be null)
 * @param zoomOutButton Button for zooming out (optional, can be
null)
 * @param zoomSlider Slider for zoom control (optional, can be null)
 */
public ZoomController(Canvas canvas, StackPane canvasContainer,
                        Button zoomInButton, Button zoomOutButton,
Slider zoomSlider) {
    this.canvas = canvas;
    this.canvasContainer = canvasContainer;
    this.zoomInButton = zoomInButton;
    this.zoomOutButton = zoomOutButton;
    this.zoomSlider = zoomSlider;

    setupEventHandlers();
    setupButtons();
    setupSlider();
}

/**
 * Set up mouse event handlers for zoom and pan
 */
private void setupEventHandlers() {
    // Scroll zoom
    canvas.setOnScroll(this::handleScroll);

    // Pan with mouse drag
    canvas.setOnMousePressed(this::handleMousePressed);
    canvas.setOnMouseDragged(this::handleMouseDragged);
}

```

```

        canvas.setOnMouseReleased(this::handleMouseReleased);

        // Set cursor based on board size
        updateCursor();

        // Listen for canvas size changes
        canvas.widthProperty().addListener((obs, oldVal, newVal) -> {
            updateCursor();
            // For very large boards, automatically set an initial zoom
level
            if (newVal.doubleValue() > 1000 && !hasResizedCanvas) {
                hasResizedCanvas = true;
                Platform.runLater(() -> {
                    // Use a lower initial zoom for very large boards
                    double initialZoom = calculateOptimalInitialZoom();
                    zoom(initialZoom);
                });
            }
        });
        canvas.heightProperty().addListener((obs, oldVal, newVal) ->
updateCursor());
    }

    /**
     * Calculate optimal initial zoom level based on canvas size and
     container
     */
    private double calculateOptimalInitialZoom() {
        // Safety check - if we can't get container dimensions, use a
default
        if (canvasContainer == null || canvasContainer.getWidth() <= 0
|| canvasContainer.getHeight() <= 0) {
            // Default small scale for very large boards
            if (canvas.getWidth() > 1400 || canvas.getHeight() > 1400) {
                return 0.3;
            } else if (canvas.getWidth() > 1000 || canvas.getHeight() >
1000) {
                return 0.5;
            } else {
                return 0.8;
            }
        }

        // Calculate zoom that would fit the canvas in the container
with some padding
        double containerWidth = canvasContainer.getWidth() - 40; // 20px
padding on each side
        double containerHeight = canvasContainer.getHeight() - 40;

        double widthRatio = containerWidth / canvas.getWidth();
        double heightRatio = containerHeight / canvas.getHeight();

        // Choose the smaller ratio to ensure complete visibility

```

```

        double fitZoom = Math.min(widthRatio, heightRatio);

        // Limit to a reasonable range
        return Math.max(0.25, Math.min(fitZoom, 1.0));
    }

    /**
     * Update cursor based on board size and zoom level
     */
    private void updateCursor() {
        boolean isVeryLargeBoard = canvas.getWidth() > 1000 ||
        canvas.getHeight() > 1000;

        if (zoomScale > 1.0 || isLargeBoard() || isVeryLargeBoard) {
            canvas.setCursor(Cursor.HAND); // Indicate draggable
        } else {
            canvas.setCursor(Cursor.DEFAULT);
        }
    }

    /**
     * Set up zoom buttons
     */
    private void setupButtons() {
        if (zoomInButton != null) {
            zoomInButton.setOnAction(e -> zoomIn());
        }

        if (zoomOutButton != null) {
            zoomOutButton.setOnAction(e -> zoomOut());
        }
    }

    /**
     * Set up zoom slider
     */
    private void setupSlider() {
        if (zoomSlider != null) {
            // Set slider range and initial value
            zoomSlider.setMin(minZoom);
            zoomSlider.setMax(maxZoom);
            zoomSlider.setValue(zoomScale);

            // Update zoom when slider changes
            zoomSlider.valueProperty().addListener((obs, oldVal, newVal)
-> {
                // Only update if the change is significant
                if (Math.abs(newVal.doubleValue() - zoomScale) > 0.01) {
                    zoom(newVal.doubleValue());
                }
            });
        }
    }
}

```

```

/**
 * Handle scroll events for zooming
 */
private void handleScroll(ScrollEvent event) {
    // Get mouse position relative to canvas
    double mouseX = event.getX();
    double mouseY = event.getY();

    // Calculate new scale with adaptive zoom rate for large boards
    boolean isVeryLargeBoard = canvas.getWidth() > 1000 ||
canvas.getHeight() > 1000;

    // Use a smaller zoom factor for very large boards - makes
zooming more gradual
    double effectiveZoomFactor = isVeryLargeBoard ? 1.05 :
zoomFactor;

    double delta = event.getDeltaY() > 0 ? effectiveZoomFactor : 1 /
effectiveZoomFactor;
    double newScale = zoomScale * delta;
    newScale = Math.max(minZoom, Math.min(maxZoom, newScale));

    // Only apply zoom if scale has changed
    if (newScale != zoomScale) {
        // Calculate zoom center (mouse position)
        Point2D mousePoint = new Point2D(mouseX, mouseY);
        zoom(newScale, mousePoint);
    }

    event.consume();
}

/**
 * Handle mouse pressed event for panning
 */
private void handleMousePressed(MouseEvent event) {
    // Special handling for very large boards - always allow
dragging
    boolean isVeryLargeBoard = canvas.getWidth() > 1000 ||
canvas.getHeight() > 1000;

    // Start panning with middle button or right button
    if (event.isMiddleButtonDown() || event.isSecondaryButtonDown())
{
        lastMousePosition = new Point2D(event.getX(), event.getY());
        isPanning = true;
        canvas.setCursor(Cursor.MOVE);
    }

    // Start panning with left button based on board size and zoom
    else if (event.isPrimaryButtonDown()) {
        // Always enable left-button dragging for large boards, very

```

```

large boards, or when zoomed
    if (zoomScale > 1.0 || isLargeBoard() || isVeryLargeBoard) {
        lastMousePosition = new Point2D(event.getX(),
event.getY());
        leftButtonDragging = true;
        canvas.setCursor(Cursor.MOVE);
    }
}

/**
 * Check if the board is considered "large" and should allow
dragging
 */
private boolean isLargeBoard() {
    // Always consider boards with dimensions over 10x10 as large
    return canvas.getWidth() > 400 || canvas.getHeight() > 400;
}

/**
 * Handle mouse dragged event for panning
 */
private void handleMouseDragged(MouseEvent event) {
    // Handle traditional panning with middle/right mouse button
    if (isPanning) {
        handlePanning(event);
    }

    // Handle left-button drag panning
    else if (leftButtonDragging) {
        // Always allow dragging once initiated
        handlePanning(event);
    }
}

/**
 * Common panning logic for any button drag
 */
private void handlePanning(MouseEvent event) {
    if (lastMousePosition == null) {
        lastMousePosition = new Point2D(event.getX(), event.getY());
        return;
    }

    // Calculate delta in screen space
    double deltaX = event.getX() - lastMousePosition.getX();
    double deltaY = event.getY() - lastMousePosition.getY();

    // Special handling for very large boards - apply a higher
sensitivity factor
    boolean isVeryLargeBoard = canvas.getWidth() > 1000 ||
canvas.getHeight() > 1000;
    double sensitivityFactor = isVeryLargeBoard ? 2.0 : 1.0;

```



```

        // Convert delta to canvas space (accounting for zoom)
        // We divide by zoomScale because when zoomed in, a small mouse
movement
        // translates to a larger canvas movement
        double canvasDeltaX = (deltaX / zoomScale) * sensitivityFactor;
        double canvasDeltaY = (deltaY / zoomScale) * sensitivityFactor;

        // Update translation
        translateX += canvasDeltaX;
        translateY += canvasDeltaY;

        lastMousePosition = new Point2D(event.getX(), event.getY());

        // Apply transformation
        applyTransform();
    }

    /**
     * Handle mouse released event for panning
     */
    private void handleMouseReleased(MouseEvent event) {
        if (isPanning) {
            isPanning = false;
            updateCursor();
        }

        if (leftButtonDragging) {
            leftButtonDragging = false;
            updateCursor();
        }
    }

    /**
     * Zoom in by one step
     */
    public void zoomIn() {
        double newScale = zoomScale * zoomFactor;
        newScale = Math.min(newScale, maxZoom);
        zoom(newScale);
    }

    /**
     * Zoom out by one step
     */
    public void zoomOut() {
        double newScale = zoomScale / zoomFactor;
        newScale = Math.max(newScale, minZoom);
        zoom(newScale);
    }

    /**
     * Reset zoom and pan to default

```

```

    */
    public void resetZoom() {
        zoomScale = 1.0;
        translateX = 0;
        translateY = 0;

        // Update slider if available
        if (zoomSlider != null) {
            zoomSlider.setValue(zoomScale);
        }

        // Update cursor based on zoom level
        updateCursor();

        // Apply transformation
        applyTransform();
    }

    /**
     * Zoom to specific scale
     */
    public void zoom(double newScale) {
        // Zoom to center of canvas
        double centerX = canvas.getWidth() / 2;
        double centerY = canvas.getHeight() / 2;
        zoom(newScale, new Point2D(centerX, centerY));
    }

    /**
     * Zoom to specific scale at a specific point
     */
    public void zoom(double newScale, Point2D center) {
        if (newScale < minZoom || newScale > maxZoom) {
            return;
        }

        // Get relative position of zoom center
        double centerX = center.getX();
        double centerY = center.getY();

        // Calculate new translation to maintain center point
        double factor = newScale / zoomScale;
        translateX = factor * (translateX - centerX) + centerX;
        translateY = factor * (translateY - centerY) + centerY;

        // Update zoom scale
        zoomScale = newScale;

        // Update slider if available
        if (zoomSlider != null) {
            zoomSlider.setValue(zoomScale);
        }
    }

```

```

        // Update cursor based on new zoom level
        updateCursor();

        // Apply transformation
        applyTransform();
    }

    /**
     * Apply current zoom and translation to canvas
     */
    private void applyTransform() {
        // Store the canvas dimensions before transform
        double canvasWidth = canvas.getWidth();
        double canvasHeight = canvas.getHeight();

        // Reset canvas transform
        canvas.setScaleX(1);
        canvas.setScaleY(1);
        canvas.setTranslateX(0);
        canvas.setTranslateY(0);

        // Apply scale transform
        canvas.setScaleX(zoomScale);
        canvas.setScaleY(zoomScale);

        // Try to get container dimensions
        double containerWidth = 0;
        double containerHeight = 0;

        if (canvasContainer != null) {
            containerWidth = canvasContainer.getWidth();
            containerHeight = canvasContainer.getHeight();

            // If container dimensions are not available yet, use
            // reasonable defaults
            if (containerWidth <= 0) containerWidth = 800;
            if (containerHeight <= 0) containerHeight = 600;
        } else {
            // Default fallback values if container is null
            containerWidth = 800;
            containerHeight = 600;
        }

        // Calculate bounds to prevent excessive dragging for very large
        // boards
        double scaledWidth = canvasWidth * zoomScale;
        double scaledHeight = canvasHeight * zoomScale;

        // For very large boards, use a more aggressive bounding
        // approach
        boolean isVeryLargeBoard = canvasWidth > 1000 || canvasHeight >
        1000;
        double boundingFactor = isVeryLargeBoard ? 0.3 : 0.5; // Tighter

```

```

bounds for very large boards

    // Limit translation to keep canvas mostly within view
    if (scaledWidth > containerWidth) {
        // Calculate max translation with extra padding for large
boards
        double extraPadding = isVeryLargeBoard ? 100 : 0;
        double maxTranslateX = (scaledWidth - containerWidth +
extraPadding) / 2 / zoomScale * boundingFactor;
        translateX = Math.max(-maxTranslateX,
Math.min(maxTranslateX, translateX));
    } else {
        // Center smaller canvas horizontally
        translateX = 0;
    }

    if (scaledHeight > containerHeight) {
        // Calculate max translation with extra padding for large
boards
        double extraPadding = isVeryLargeBoard ? 100 : 0;
        double maxTranslateY = (scaledHeight - containerHeight +
extraPadding) / 2 / zoomScale * boundingFactor;
        translateY = Math.max(-maxTranslateY,
Math.min(maxTranslateY, translateY));
    } else {
        // Center smaller canvas vertically
        translateY = 0;
    }

    // Apply translation with adaptive scaling based on board size
    double translationFactor = 1 - zoomScale;
    canvas.setTranslateX(translateX * translationFactor);
    canvas.setTranslateY(translateY * translationFactor);
}

/**
 * Set zoom limits
 */
public void setZoomLimits(double minZoom, double maxZoom) {
    this.minZoom = minZoom;
    this.maxZoom = maxZoom;

    // Update slider limits if available
    if (zoomSlider != null) {
        zoomSlider.setMin(minZoom);
        zoomSlider.setMax(maxZoom);
    }
}

/**
 * Get current zoom scale
 */
public double getZoomScale() {

```

```

        return zoomScale;
    }

    /**
     * Transform a point in canvas space to zoomed space
     */
    public Point2D canvasToZoomed(double x, double y) {
        double zoomedX = x * zoomScale + translateX * (1 - zoomScale);
        double zoomedY = y * zoomScale + translateY * (1 - zoomScale);
        return new Point2D(zoomedX, zoomedY);
    }

    /**
     * Transform a point in zoomed space to canvas space
     */
    public Point2D zoomedToCanvas(double x, double y) {
        double canvasX = (x - translateX * (1 - zoomScale)) / zoomScale;
        double canvasY = (y - translateY * (1 - zoomScale)) / zoomScale;
        return new Point2D(canvasX, canvasY);
    }
}

```

3.2.18 MainView.fxml

```

<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.Insets?>
<?import javafx.scene.canvas.Canvas?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.ComboBox?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.ProgressBar?>
<?import javafx.scene.control.ScrollPane?>
<?import javafx.scene.control.Slider?>
<?import javafx.scene.control.Spinner?>
<?import javafx.scene.control.Tab?>
<?import javafx.scene.control.TabPane?>
<?import javafx.scene.control.TextArea?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.control.ToggleButton?>
<?import javafx.scene.layout.BorderPane?>
<?import javafx.scene.layout.GridPane?>
<?import javafx.scene.layout.HBox?>
<?import javafx.scene.layout.AnchorPane?>
<?import javafx.scene.layout.StackPane?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.text.Text?>

<StackPane prefHeight="720.0" prefWidth="1200.0"

```

```

xmlns="http://javafx.com/javafx/11.0.1"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="gui.controllers.MainController">
    <children>
        <!-- Full-screen background image is handled by CSS in .root -->
        <BorderPane styleClass="root">
            <!-- Dark overlay for the entire screen -->
            <center>
                <AnchorPane styleClass="overlay">
                    <children>
                        <!-- Main content -->
                        <BorderPane fx:id="rootContainer"
AnchorPane.bottomAnchor="0.0" AnchorPane.leftAnchor="0.0"
AnchorPane.rightAnchor="0.0" AnchorPane.topAnchor="0.0">
                            <left>
                                <VBox fx:id="sidebarContainer" prefWidth="400.0"
spacing="15.0" styleClass="sidebar" visible="true">
                                    <children>
                                        <Label styleClass="sidebar-title"
text="Yahallo, Minna!" />
                                        <Label styleClass="section-header"
text="What do you wanna vibe to? :3" />
                                        <HBox alignment="CENTER_LEFT"
spacing="10.0">
                                            <children>
                                                <ComboBox fx:id="songComboBox"
prefWidth="260.0" promptText="Select a song..." />
                                                <Button fx:id="pauseSongButton"
text="Pause" onAction="#handlePauseSong" />
                                            </children>
                                        </HBox>
                                        <TabPane fx:id="inputTabPane"
prefHeight="320.0" prefWidth="200.0" styleClass="config-tabs"
tabClosingPolicy="UNAVAILABLE">
                                            <tabs>
                                                <Tab fx:id="fileTab" text="File
Input">
                                                    <content>
                                                        <VBox spacing="10.0">
                                                            <children>
                                                                <Label
styleClass="section-header" text="Select Puzzle File" />
                                                                <HBox spacing="5.0">
                                                                    <children>
                                                                        <TextField
fx:id="filePathField" promptText="Choose a file..." HBox.hgrow="ALWAYS"
/>
                                                                        <Button
fx:id="browseButton" mnemonicParsing="false" onAction="#handleBrowse"
styleClass="action-button" text="Browse" />
                                                                    </children>
                                                                </HBox>
                                                                <Button

```



```

spacing="10.0">
    <HBox alignment="CENTER"
        <children>
            <Label
text="Non-primary pieces:" />
            <Spinner
fx:id="piecesSpinner" editable="true" prefWidth="70.0" />
        </children>
    </HBox>
    <Button
fx:id="createMatrixButton" mnemonicParsing="false"
onAction="#handleCreateMatrix" styleClass="primary-button" text="Create
Matrix" />
    <VBox
fx:id="matrixInputContainer" spacing="10.0">
        <children>
            <ScrollPane
prefHeight="160.0" styleClass="transparent-scroll-pane">
                <content>
                    <GridPane
fx:id="matrixGrid" alignment="CENTER" hgap="2.0" vgap="2.0" />
                </content>
            </ScrollPane>
            <Label text="Use
'P' for primary piece, 'K' for exit, other uppercase letters for pieces,
'.' for empty cells" wrapText="true" />
            <Button
fx:id="solveButton" mnemonicParsing="false" onAction="#handleSolve"
styleClass="primary-button" text="Load Matrix" />
        </children>
    </VBox>
    </children>
    <padding>
        <Insets bottom="10.0"
left="10.0" right="10.0" top="10.0" />
    </padding>
    </VBox>
</content>
</Tab>
</tabs>
</TabPage>
<Label styleClass="section-header"
text="Algorithm Selection" />
<HBox alignment="CENTER_LEFT"
spacing="10.0">
    <children>
        <Label text="Algorithm:"
style="-fx-text-fill: white;" />
        <ComboBox fx:id="algorithmComboBox"
prefWidth="260.0" />
    </children>
</HBox>
<HBox alignment="CENTER_LEFT"

```



```

spacing="10.0">
    <children>
        <Label text="Heuristic:"
style="-fx-text-fill: white;" />
        <ComboBox fx:id="heuristicComboBox"
prefWidth="260.0" />
    </children>
</HBox>
<HBox alignment="CENTER_LEFT"
spacing="10.0">
    <children>
        <Label text="Compound Move:"
style="-fx-text-fill: white;" />
        <Button fx:id="compoundButton"
text="OFF" onAction="#handleToggleCompound" />
    </children>
</HBox>
<Button fx:id="solveButton"
mnemonicParsing="false" onAction="#handleSolve"
styleClass="solve-button" text="Solve Puzzle" />
    <VBox styleClass="status-section">
        <children>
            <Text strokeType="OUTSIDE"
strokeWidth="0.0" styleClass="status-title" text="STATUS:" />
            <Text fx:id="statusText"
strokeType="OUTSIDE" strokeWidth="0.0" styleClass="status-text"
text="Ready to solve" wrappingWidth="350.0" />
            <ProgressBar fx:id="progressBar"
prefWidth="350.0" progress="0.0" visible="false" />
        </children>
    </VBox>
</children>
<padding>
    <Insets bottom="20.0" left="20.0"
right="20.0" top="20.0" />
</padding>
</VBox>
</left>
<center>
    <VBox fx:id="boardDisplay" alignment="CENTER"
spacing="10.0">
        <!-- Welcome Screen shown initially -->
        <VBox fx:id="welcomePane" alignment="CENTER"
spacing="20.0" styleClass="welcome-pane" VBox.vgrow="ALWAYS">
            <children>
                <Label styleClass="welcome-title"
text="Muri muri muri muri &gt;_&lt;" />
                <Label styleClass="welcome-subtitle"
text="Load a configuration to get started :3" />
                <Label styleClass="welcome-text"
text="Configure settings in the sidebar and click 'Solve Puzzle'"
wrapText="true" />
            </children>
        </VBox>
    </center>
</HBox>

```

```

        </VBox>

        <!-- Board area with zoom controls - hidden
initially -->
        <BorderPane fx:id="boardContainer"
visible="false" VBox.vgrow="ALWAYS">
            <center>
                <StackPane fx:id="canvasContainer"
styleClass="zoomable-canvas-container">
                    <Canvas fx:id="boardCanvas"
height="500.0" width="600.0" />
                </StackPane>
            </center>
            <!-- Remove the right section with the
zoom controls -->
        </BorderPane>

        <!-- Controls and info area - always visible
but components may be hidden -->
        <VBox alignment="CENTER" spacing="10.0"
styleClass="board-controls-area">
            <!-- Animation controls -->
            <HBox fx:id="animationControls"
alignment="CENTER" spacing="15.0" visible="false">
                <Button fx:id="playButton"
onAction="#handlePlay" styleClass="control-button" text="Play"
prefWidth="80.0" prefHeight="40.0" />
                <Button fx:id="pauseButton"
onAction="#handlePause" styleClass="control-button" text="Pause"
prefWidth="80.0" prefHeight="40.0" />
                <Button fx:id="resetButton"
onAction="#handleReset" styleClass="control-button" text="Reset"
prefWidth="80.0" prefHeight="40.0" />
                <Slider fx:id="speedSlider"
blockIncrement="0.25" majorTickUnit="0.5" max="3.0" min="0.5"
prefWidth="180.0" value="1.0" />
                <Label text="Speed"
style="-fx-font-size: 14px; -fx-text-fill: white;" />
                <Button fx:id="saveButton"
onAction="#handleSave" styleClass="control-button" text="Save"
prefWidth="80.0" prefHeight="40.0" />
            <padding>
                <Insets top="10.0" bottom="10.0" />
            </padding>
        </HBox>

            <!-- Information labels -->
            <Label fx:id="moveLabel"
styleClass="move-label" style="-fx-font-size: 20px; -fx-text-fill:
white;" />
            <Label fx:id="statsLabel"
styleClass="stats-label" style="-fx-font-size: 16px; -fx-text-fill:
white;" />

```

```

                                <padding>
                                <Insets bottom="10.0" left="15.0"
right="15.0" top="10.0" />
                                </padding>
                                </VBox>
                                <padding>
                                <Insets bottom="15.0" left="15.0"
right="15.0" top="15.0" />
                                </padding>
                                </VBox>
                                </center>
                                </BorderPane>
                                </children>
                                </AnchorPane>
                                </center>
                                </BorderPane>
                                </children>
                                </StackPane>

```

3.2.19 *MatrixInputWindow.fxml*

```

<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<VBox xmlns="http://javafx.com/javafx"
      xmlns:fx="http://javafx.com/fxml"
      fx:controller="gui.controllers.MatrixInputWindowController"
      spacing="10" alignment="CENTER" style="-fx-padding: 20;">

    <Label text="Enter Matrix Cells" />

    <ScrollPane prefViewportHeight="300" prefViewportWidth="300">
        <content>
            <GridPane fx:id="matrixGrid" hgap="2" vgap="2" />
        </content>
    </ScrollPane>

    <HBox spacing="10">
        <Button text="OK" fx:id="okButton" />
    </HBox>
</VBox>

```

3.2.20 VisualizationView.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.Insets?>
<?import javafx.scene.canvas.Canvas?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.Slider?>
<?import javafx.scene.layout.BorderPane?>
<?import javafx.scene.layout.HBox?>
<?import javafx.scene.layout.StackPane?>
<?import javafx.scene.layout.VBox?>

<BorderPane fx:id="rootPane" prefHeight="650.0" prefWidth="800.0"
styleClass="root" xmlns="http://javafx.com/javafx/11.0.1"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="gui.controllers.VisualizationController">
    <top>
        <HBox alignment="CENTER" prefHeight="50.0"
styleClass="board-display">
            <children>
                <Label styleClass="welcome-title" text="Rush Hour Puzzle
Solution" />
            </children>
            <padding>
                <Insets bottom="10.0" left="10.0" right="10.0" top="10.0" />
            </padding>
            <BorderPane.margin>
                <Insets bottom="10.0" left="10.0" right="10.0" top="10.0" />
            </BorderPane.margin>
        </HBox>
    </top>

    <center>
        <!-- Main board container with zoom controls -->
        <BorderPane styleClass="canvas-container">
            <center>
                <StackPane fx:id="canvasContainer"
styleClass="zoomable-canvas-container">
                    <Canvas fx:id="boardCanvas" height="400.0" width="480.0"
/>
                </StackPane>
            </center>
            <right>
                <!-- Vertical zoom controls -->
                <VBox alignment="CENTER" spacing="10.0"
styleClass="zoom-controls-vertical">
                    <Button fx:id="zoomInButton" styleClass="zoom-button"
text="+" />
                </VBox>
            </right>
        </BorderPane>
    </center>
</BorderPane>
```



```

        <padding>
            <Insets bottom="5.0" left="5.0" right="5.0" top="5.0"
/>

        </padding>
    </HBox>
    <Label fx:id="statsLabel" alignment="CENTER"
styleClass="stats-label" text="Algorithm: | States Examined: | Moves: |
Time:" textAlignment="CENTER" wrapText="true" />
    <Label fx:id="statusLabel" alignment="CENTER"
styleClass="status-label" text="" textAlignment="CENTER" />
</children>
<padding>
    <Insets bottom="15.0" left="15.0" right="15.0" top="15.0" />
</padding>
<BorderPane.margin>
    <Insets bottom="10.0" left="10.0" right="10.0" top="10.0" />
</BorderPane.margin>
</VBox>
</bottom>
</BorderPane>

```

3.2.21 *GuiMain.java*

```

package gui;

import java.io.File;
import java.net.URL;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.stage.Stage;

/**
 * Main JavaFX Application class for the Rush Hour Puzzle Solver GUI
 */
public class GuiMain extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        try {
            // Create necessary directories
            createDirectories();

            // Try to load FXML
            URL fxmlUrl = findResource("gui/fxml/MainView.fxml");
            if (fxmlUrl == null) {

```

```

        System.err.println("Could not find MainView.fxml");
        System.err.println("Trying alternative paths...");

        // List available resources
        System.err.println("Available resources in classpath:");
        URL rootResource = getClass().getResource("/");
        if (rootResource != null) {
            System.err.println("Root resource: " +
rootResource);
        } else {
            System.err.println("Root resource not found");
        }

        throw new RuntimeException("Cannot find MainView.fxml.
Make sure it's in src/gui/fxml/");
    }

    FXMLLoader loader = new FXMLLoader(fxmlUrl);
    Parent root = loader.load();

    Scene scene = new Scene(root, 1100, 750);

    // Load CSS
    URL cssUrl = findResource("resources/styles.css");
    if (cssUrl != null) {
        scene.getStylesheets().add(cssUrl.toExternalForm());
        System.out.println("Loaded CSS from: " +
cssUrl.toExternalForm());
    } else {
        System.out.println("Warning: Could not find styles.css,
using default styles");

        // Try to load from file directly
        File cssFile = new File("src/resources/styles.css");
        if (cssFile.exists()) {

scene.getStylesheets().add(cssFile.toURI().toString());
            System.out.println("Loaded CSS from file: " +
cssFile.getAbsolutePath());
        }
    }

    // Set application icon
    try {
        File iconFile = new
File("src/resources/images/icon.png");
        if (iconFile.exists()) {
            primaryStage.getIcons().add(new
Image(iconFile.toURI().toString()));
            System.out.println("Loaded icon from: " +
iconFile.getAbsolutePath());
        } else {
            System.out.println("Icon file not found, using

```

```

default icon");
    }
    } catch (Exception e) {
        System.err.println("Error loading icon (non-critical): "
+ e.getMessage());
    }

    primaryStage.setTitle("Kessoku No Owari ♪");
    primaryStage.setScene(scene);
    primaryStage.setMinWidth(1000);
    primaryStage.setMinHeight(700);

    primaryStage.show();

    } catch (Exception e) {
        System.err.println("Error loading GUI:");
        e.printStackTrace();
        throw e;
    }
}

/**
 * Find a resource file by trying multiple paths
 */
private URL findResource(String path) {
    // Try different class loaders and paths
    URL resource = null;

    // Try with class loader
    resource = getClass().getClassLoader().getResource(path);
    if (resource != null) return resource;

    // Try with / prefix
    resource = getClass().getResource("/") + path);
    if (resource != null) return resource;

    // Try without / prefix
    resource = getClass().getResource(path);
    if (resource != null) return resource;

    // Try as file
    File file = new File("src/" + path);
    if (file.exists()) {
        try {
            return file.toURI().toURL();
        } catch (Exception e) {
            // Ignore
        }
    }

    return null;
}

```



```

/**
 * Create necessary directories for resources
 */
private void createDirectories() {
    try {
        // Create test/input and test/output directories
        new File("test/input").mkdirs();
        new File("test/output").mkdirs();

        // Create resources directories
        new File("src/resources/images").mkdirs();

        System.out.println("Created resource directories");
    } catch (Exception e) {
        System.err.println("Error creating directories
(non-critical): " + e.getMessage());
    }
}

public static void main(String[] args) {
    launch(args);
}
}

```

3.2.22 styles.css

```

/* Dark mode base styling with anime background */
.root {
    -fx-font-family: "Segoe UI", "Arial", sans-serif;
    -fx-background-image: url('/resources/images/bocchi.jpg');
    -fx-background-size: cover;
    -fx-background-position: center;
    -fx-background-repeat: no-repeat;
}

/* Dark overlay for the entire application */
.overlay {
    -fx-background-color: rgba(0, 0, 0, 0.7);
    -fx-background-insets: 0;
}

/* Welcome pane styling to match the image */
.welcome-pane {
    -fx-padding: 50;
    -fx-alignment: center;
    -fx-background-color: transparent;
}

.welcome-title {

```

```

        -fx-font-size: 36px;
        -fx-font-weight: bold;
        -fx-text-fill: white;
        -fx-effect: dropshadow(gaussian, rgba(0, 0, 0, 0.7), 5, 0, 0, 1);
    }

    .welcome-subtitle {
        -fx-font-size: 20px;
        -fx-text-fill: #e0e0e0;
        -fx-effect: dropshadow(gaussian, rgba(0, 0, 0, 0.7), 3, 0, 0, 1);
    }

    .welcome-text {
        -fx-font-size: 16px;
        -fx-text-fill: #cccccc;
        -fx-effect: dropshadow(gaussian, rgba(0, 0, 0, 0.7), 2, 0, 0, 1);
    }

    /* Sidebar styling */
    .sidebar {
        -fx-background-color: rgba(30, 30, 30, 0.85);
        -fx-effect: dropshadow(three-pass-box, rgba(0, 0, 0, 0.5), 15, 0, 0, 0);
        -fx-background-radius: 0 12 12 0;
        -fx-min-width: 380px;
    }

    .sidebar-title {
        -fx-font-size: 28px;
        -fx-font-weight: bold;
        -fx-text-fill: #e0e0e0;
        -fx-effect: dropshadow(gaussian, rgba(0, 0, 0, 0.4), 2, 0, 0, 1);
    }

    .section-header {
        -fx-font-size: 18px;
        -fx-font-weight: bold;
        -fx-text-fill: #bbbbbb;
    }

    /* Canvas container styling */
    .canvas-container {
        -fx-background-color: rgba(40, 40, 40, 0.85);
        -fx-background-radius: 12;
        -fx-effect: dropshadow(three-pass-box, rgba(0, 0, 0, 0.7), 15, 0, 0, 0);
        -fx-padding: 20;
        -fx-min-height: 500px;
    }

    .board-display {
        -fx-background-color: rgba(18, 18, 18, 0.85);
        -fx-background-radius: 8;
    }

```

```

}

/* Modern minimalist button styling */
.button {
    -fx-background-color: #444444;
    -fx-text-fill: #e0e0e0;
    -fx-font-weight: normal;
    -fx-background-radius: 20;
    -fx-padding: 8 20;
    -fx-cursor: hand;
    -fx-border-width: 0;
    -fx-effect: dropshadow(three-pass-box, rgba(0, 0, 0, 0.3), 3, 0, 0,
1);
    -fx-font-size: 13px;
}

.button:hover {
    -fx-background-color: #555555;
    -fx-effect: dropshadow(three-pass-box, rgba(0, 0, 0, 0.4), 5, 0, 0,
2);
}

.button:pressed {
    -fx-background-color: #333333;
    -fx-effect: dropshadow(three-pass-box, rgba(0, 0, 0, 0.2), 2, 0, 0,
0);
}

.primary-button {
    -fx-background-color: #2a6fdb;
    -fx-text-fill: white;
}

.primary-button:hover {
    -fx-background-color: #3a7feb;
}

.primary-button:pressed {
    -fx-background-color: #1a5fcb;
}

.action-button {
    -fx-background-color: #444444;
    -fx-text-fill: #e0e0e0;
}

.action-button:hover {
    -fx-background-color: #555555;
}

.solve-button {
    -fx-background-color: #27ae60;
    -fx-text-fill: white;
}

```

```

        -fx-font-size: 18px;
        -fx-padding: 12 24;
        -fx-background-radius: 25;
    }

    .solve-button:hover {
        -fx-background-color: #2ecc71;
    }

    .control-button {
        -fx-background-color: #2a6fdb;
        -fx-text-fill: white;
        -fx-padding: 6 15;
        -fx-font-weight: normal;
        -fx-background-radius: 20;
    }

    .control-button:hover {
        -fx-background-color: #3a7feb;
    }

    .icon-button {
        -fx-background-color: transparent;
        -fx-text-fill: #bbbbbb;
        -fx-padding: 5;
        -fx-cursor: hand;
        -fx-background-radius: 100;
    }

    .icon-button:hover {
        -fx-background-color: rgba(255, 255, 255, 0.15);
        -fx-background-radius: 100;
    }

    .icon-button:selected {
        -fx-text-fill: #ff5252;
    }

    /* Text field styling */
    .text-field, .text-area {
        -fx-background-color: #2a2a2a;
        -fx-border-color: #555555;
        -fx-border-radius: 5;
        -fx-background-radius: 5;
        -fx-text-fill: #e0e0e0;
        -fx-prompt-text-fill: #888888;
    }

    .text-field:focus, .text-area:focus {
        -fx-border-color: #2a6fdb;
    }

    .text-area .content {

```

```

        -fx-background-color: #2a2a2a;
    }

    /* Combo box styling */
    .combo-box {
        -fx-background-color: #2a2a2a;
        -fx-border-color: #555555;
        -fx-background-radius: 5;
        -fx-border-radius: 5;
        -fx-text-fill: #e0e0e0;
    }

    .combo-box:focused {
        -fx-border-color: #2a6fdb;
    }

    .combo-box .list-cell {
        -fx-padding: 5 8;
        -fx-background-color: #2a2a2a;
        -fx-text-fill: #e0e0e0;
    }

    .combo-box .list-view {
        -fx-background-color: #2a2a2a;
        -fx-border-color: #555555;
    }

    .combo-box .list-view .list-cell:filled:selected, .combo-box .list-view
    .list-cell:filled:selected:hover {
        -fx-background-color: #2a6fdb;
        -fx-text-fill: white;
    }

    .combo-box .list-view .list-cell:filled:hover {
        -fx-background-color: #3a3a3a;
    }

    .combo-box .arrow-button {
        -fx-background-color: transparent;
    }

    .combo-box .arrow {
        -fx-background-color: #bbbbbb;
    }

    /* Tabs styling */
    .config-tabs {
        -fx-tab-min-width: 100px;
    }

    .config-tabs .tab {
        -fx-background-color: #333333;
        -fx-background-radius: 5 5 0 0;
    }

```

```

        -fx-padding: 5 15;
    }

.config-tabs .tab-header-area .tab-header-background {
    -fx-background-color: #222222;
}

.config-tabs .tab .tab-label {
    -fx-text-fill: #bbbbbb;
    -fx-font-size: 14px;
}

.config-tabs .tab:selected {
    -fx-background-color: #444444;
    -fx-border-width: 0;
}

.config-tabs .tab:selected .tab-label {
    -fx-text-fill: #e0e0e0;
}

.config-tabs .tab-pane:focused > .tab-header-area > .headers-region >
.tab:selected .focus-indicator {
    -fx-border-color: transparent;
}

.tab-pane > .tab-header-area > .headers-region > .tab > .tab-container >
.tab-close-button {
    -fx-background-color: #bbbbbb;
}

.tab-pane .tab-content-area {
    -fx-background-color: #333333;
}

/* Matrix cell styling */
.matrix-cell {
    -fx-pref-width: 40px;
    -fx-pref-height: 40px;
    -fx-max-width: 40px;
    -fx-max-height: 40px;
    -fx-alignment: center;
    -fx-font-family: monospace;
    -fx-font-weight: bold;
    -fx-font-size: 18px;
    -fx-background-color: #2a2a2a;
    -fx-text-fill: #e0e0e0;
}

/* Status styling */
.status-section {
    -fx-background-color: rgba(40, 40, 40, 0.9);
    -fx-background-radius: 8;
}

```

```

        -fx-padding: 10;
        -fx-min-height: 90px; /* Increased height to accommodate the title
*/
        -fx-pref-height: 90px;
        -fx-max-height: 90px;
    }

/* Status title styling */
.status-title {
    -fx-font-size: 16px;
    -fx-font-weight: bold;
    -fx-fill: white;
    -fx-underline: false;
}

/* Status text styling - light blue for normal status */
.status-text {
    -fx-font-size: 14px;
    -fx-fill: #aee6ff; /* Light blue color for better visibility */
    -fx-wrap-text: true;
    -fx-padding: 5 0 0 0;
}

/* Error status styling - light red */
.status-text.error {
    -fx-fill: #ff8080; /* Light red for errors */
}

/* Info labels */
.move-label {
    -fx-font-size: 20px; /* Increased from 16px */
    -fx-font-weight: bold;
    -fx-text-fill: white; /* Changed from #e0e0e0 to white for better
visibility */
}

.stats-label {
    -fx-font-size: 16px; /* Increased from 12px */
    -fx-text-fill: white; /* Changed from #bbbbbb to white for better
visibility */
}

/* Separator styling */
.separator *.line {
    -fx-border-color: #555555;
    -fx-border-width: 1px;
}

/* Slider styling */
.slider .track {
    -fx-background-color: #555555;
}

```

```

.slider .thumb {
    -fx-background-color: #2a6fdb;
}

/* Spinner styling */
.spinner {
    -fx-background-color: #2a2a2a;
    -fx-border-color: #555555;
    -fx-border-radius: 5;
    -fx-background-radius: 5;
}

.spinner:focused {
    -fx-border-color: #2a6fdb;
}

.spinner .text-field {
    -fx-border-width: 0;
    -fx-background-radius: 0;
}

.spinner .increment-arrow-button, .spinner .decrement-arrow-button {
    -fx-background-color: #444444;
}

.spinner .increment-arrow, .spinner .decrement-arrow {
    -fx-background-color: #bbbbbb;
}

/* Progress bar styling */
.progress-bar {
    -fx-accent: #2a6fdb;
}

.progress-bar .track {
    -fx-background-color: #444444;
}

/* Scroll pane styling */
.scroll-pane {
    -fx-background-color: transparent;
}

.scroll-pane > .viewport {
    -fx-background-color: transparent;
}

.scroll-pane .scroll-bar:vertical,
.scroll-pane .scroll-bar:horizontal {
    -fx-background-color: transparent;
}

.scroll-pane .increment-button,

```



```

.scroll-pane .decrement-button {
    -fx-background-color: transparent;
    -fx-border-color: transparent;
}

.scroll-pane .scroll-bar .thumb {
    -fx-background-color: #555555;
}

.transparent-scroll-pane {
    -fx-background: transparent;
    -fx-background-color: transparent;
    -fx-padding: 0;
    -fx-background-insets: 0;
    -fx-border-color: transparent;
}

.transparent-scroll-pane > .viewport {
    -fx-background-color: transparent;
    -fx-background: transparent;
}

.transparent-scroll-pane .scroll-bar:vertical,
.transparent-scroll-pane .scroll-bar:horizontal {
    -fx-opacity: 0.5;
}

/* Dialog styling */
.dark-dialog {
    -fx-background-color: #2a2a2a;
}

.dark-dialog > *.button-bar > *.container {
    -fx-background-color: #2a2a2a;
}

.dark-dialog > *.label.content {
    -fx-font-size: 14px;
    -fx-text-fill: #e0e0e0;
}

.dark-dialog:header *.header-panel {
    -fx-background-color: #333333;
}

.dark-dialog:header *.header-panel *.label {
    -fx-font-size: 18px;
    -fx-text-fill: #e0e0e0;
}

.dark-dialog .button {
    -fx-background-color: #444444;
    -fx-text-fill: #e0e0e0;
}

```

```

        -fx-background-radius: 20;
    }

    .dark-dialog .button:hover {
        -fx-background-color: #555555;
    }

    .dark-dialog .button:default {
        -fx-background-color: #2a6fdb;
        -fx-text-fill: white;
    }

    .dark-dialog .button:default:hover {
        -fx-background-color: #3a7feb;
    }

    /* Status label */
    .status-label {
        -fx-font-size: 16px; /* Increased from 14px */
        -fx-text-fill: white; /* Changed from #e0e0e0 to white for better
visibility */
        -fx-font-weight: bold;
    }

    /* Grid pane */
    .grid-pane {
        -fx-background-color: transparent;
    }

    /* Tooltip styling */
    .tooltip {
        -fx-background-color: #2a2a2a;
        -fx-text-fill: #e0e0e0;
        -fx-font-size: 12px;
        -fx-background-radius: 5;
        -fx-padding: 5 10;
    }

    /* Make sidebar label text white for better visibility */
    .sidebar Label {
        -fx-text-fill: white;
    }

    /* Algorithm and heuristic related elements */
    HBox Label {
        -fx-text-fill: white;
    }

    /* Board container with zoom area */
    .board-container {
        -fx-background-color: rgba(30, 30, 30, 0.7);
        -fx-background-radius: 12;
        -fx-effect: dropshadow(three-pass-box, rgba(0, 0, 0, 0.7), 15, 0, 0,

```

```

0);
}

/* Zoomable canvas container */
.zoomable-canvas-container {
    -fx-background-color: transparent; /* More transparent */
    -fx-background-radius: 8;
    -fx-padding: 5;
    -fx-border-color: transparent;
    -fx-border-radius: 8;
    -fx-border-width: 1px;
    -fx-min-height: 400px;
}

/* Improved vertical zoom controls */
.zoom-controls-vertical {
    -fx-background-color: rgba(40, 40, 40, 0.8);
    -fx-background-radius: 8;
    -fx-padding: 10;
    -fx-spacing: 12;
    -fx-effect: dropshadow(three-pass-box, rgba(0, 0, 0, 0.5), 10, 0, 0,
3);
    -fx-alignment: center;
    -fx-pref-width: 50px;
    -fx-max-width: 50px;
}

/* Zoom buttons with distinct styling */
.zoom-button {
    -fx-background-color: #2a6fdb;
    -fx-text-fill: white;
    -fx-font-weight: bold;
    -fx-min-width: 30px;
    -fx-min-height: 30px;
    -fx-max-width: 30px;
    -fx-max-height: 30px;
    -fx-background-radius: 15;
    -fx-font-size: 16px;
    -fx-padding: 0;
    -fx-alignment: center;
}

.zoom-button:hover {
    -fx-background-color: #3a7feb;
    -fx-effect: dropshadow(three-pass-box, rgba(255, 255, 255, 0.3), 5,
0, 0, 0);
}

/* Vertical slider styling */
.zoom-controls-vertical .slider {
    -fx-orientation: vertical;
    -fx-pref-height: 160px;
}

```

```

.zoom-controls-vertical .slider .track {
    -fx-background-color: #555555;
}

.zoom-controls-vertical .slider .thumb {
    -fx-background-color: #2a6fdb;
    -fx-background-radius: 10;
    -fx-pref-height: 15px;
    -fx-pref-width: 15px;
}

.zoom-percent-label {
    -fx-text-fill: white;
    -fx-font-weight: bold;
    -fx-font-size: 14px;
    -fx-padding: 3 0;
}

/* Controls area below the board */
.board-controls-area {
    -fx-background-color: transparent;
    -fx-background-radius: 8;
    -fx-padding: 10;
    -fx-spacing: 5;
    -fx-effect: dropshadow(three-pass-box, rgba(0, 0, 0, 0.5), 10, 0, 0,
3);
}

/* Improved control buttons */
.control-button {
    -fx-background-color: #2a6fdb;
    -fx-text-fill: white;
    -fx-padding: 8 15;
    -fx-font-weight: normal;
    -fx-background-radius: 20;
    -fx-cursor: hand;
}

.control-button:hover {
    -fx-background-color: #3a7feb;
    -fx-effect: dropshadow(three-pass-box, rgba(0, 0, 0, 0.4), 5, 0, 0,
2);
}

.control-button:pressed {
    -fx-background-color: #1a5fcb;
}

/* Improved move label with shadow for better visibility */
.move-label {
    -fx-font-size: 20px;
    -fx-font-weight: bold;
}

```

```

        -fx-text-fill: white;
        -fx-effect: dropshadow(gaussian, rgba(0, 0, 0, 0.7), 2, 0, 0, 1);
    }

    /* Stats label with better visibility */
    .stats-label {
        -fx-font-size: 16px;
        -fx-text-fill: #aee6ff;
        -fx-effect: dropshadow(gaussian, rgba(0, 0, 0, 0.5), 1, 0, 0, 1);
    }

```

3.2.23 KessokuNoOwari.java

```

import java.io.File;
import java.util.ArrayList;
import java.util.List;

/**
 * KessokuNoOwari - A simple Gradle launcher that handles Java version
 * issues
 * Usage:
 *   - To build: kessoku build
 *   - To run Gradle commands: kessoku run [gradle-command]
 */
public class KessokuNoOwari {

    // Possible Java installation locations
    private static final String[] WINDOWS_JAVA_PATHS = {
        "C:\\Program Files\\Java\\jdk-17",
        "C:\\Program Files\\Java\\jdk-11",
        "C:\\Program Files\\Java\\jdk1.8.0",
        "C:\\Program Files (x86)\\Java\\jdk-17",
        "C:\\Program Files (x86)\\Java\\jdk-11"
    };

    private static final String[] UNIX_JAVA_PATHS = {
        "/usr/lib/jvm/java-17-openjdk",
        "/usr/lib/jvm/java-11-openjdk",
        "/usr/lib/jvm/java-8-openjdk",
        "/usr/local/openjdk-17",
        "/usr/local/openjdk-11",
        "/Library/Java/JavaVirtualMachines/jdk-17.jdk/Contents/Home",
        "/Library/Java/JavaVirtualMachines/jdk-11.jdk/Contents/Home"
    };

    public static void main(String[] args) {
        try {
            System.out.println("KessokuNoOwari - Gradle Launcher");
            System.out.println("-----");
        }
    }
}

```

```

        // Find Java Home
        String javaHome = findJavaHome();

        if (javaHome != null) {
            System.out.println("Using Java Home: " + javaHome);
        } else {
            System.out.println("Warning: Could not find Java Home.
Using system Java.");
        }

        // Find Gradle wrapper
        String gradleWrapper = findGradleWrapper();
        System.out.println("Using Gradle: " + gradleWrapper);

        // Create command
        List<String> command = buildCommand(javaHome, gradleWrapper,
args);

        // Display command
        System.out.println("Running command: " + String.join(" ",
command));

        // Execute from the project root directory, not the bin
directory
        ProcessBuilder pb = new ProcessBuilder(command);

        // Set working directory to project root
        File projectRoot = findProjectRoot();
        if (projectRoot != null) {
            pb.directory(projectRoot);
            System.out.println("Working directory: " +
projectRoot.getAbsolutePath());
        }

        pb.inheritIO();
        Process process = pb.start();

        // Wait for process to complete
        int exitCode = process.waitFor();
        System.exit(exitCode);

    } catch (Exception e) {
        System.err.println("Error: " + e.getMessage());
        e.printStackTrace();
        System.exit(1);
    }
}

private static String findJavaHome() {
    // Check environment variable first
    String javaHome = System.getenv("JAVA_HOME");
    if (javaHome != null && !javaHome.isEmpty()) && new

```

```

File(javaHome).exists()) {
    return javaHome;
}

// Try to find Java installation
String[] paths = isWindows() ? WINDOWS_JAVA_PATHS :
UNIX_JAVA_PATHS;
for (String path : paths) {
    File javaDir = new File(path);
    if (javaDir.exists() && new File(javaDir, "bin/java" +
(isWindows() ? ".exe" : "")).exists()) {
        return path;
    }
}

// Try to infer from java command
try {
    String javaCommand = isWindows() ? "where java" : "which
java";
    Process process = Runtime.getRuntime().exec(javaCommand);

    // Read the output
    java.io.InputStream is = process.getInputStream();
    byte[] buffer = new byte[1024];
    int bytesRead = is.read(buffer);
    String javaPath = "";
    if (bytesRead > 0) {
        javaPath = new String(buffer, 0, bytesRead).trim();
    }

    if (!javaPath.isEmpty()) {
        File javaFile = new File(javaPath);
        if (javaFile.exists()) {
            // Try to get parent directory of bin
            File binDir = javaFile.getParentFile();
            if (binDir != null &&
binDir.getName().equals("bin")) {
                return binDir.getParent();
            }
        }
    }
} catch (Exception e) {
    // Ignore errors in this step
}

return null;
}

private static String findGradleWrapper() {
    // Check for gradlew in current directory
    String wrapper = isWindows() ? "gradlew.bat" : "./gradlew";
    File wrapperFile = new File(isWindows() ? "gradlew.bat" :
"gradlew");

```

```

        if (wrapperFile.exists()) {
            return wrapper;
        }

        // Look in parent directories
        File current = new File(".").getAbsoluteFile();
        while (current != null) {
            File gradleWrapperFile = new File(current, isWindows() ?
"gradlew.bat" : "gradlew");
            if (gradleWrapperFile.exists()) {
                return gradleWrapperFile.getAbsolutePath();
            }
            current = current.getParentFile();
        }

        // Fallback to system gradle
        return isWindows() ? "gradle.bat" : "gradle";
    }

    /**
     * Find the project root directory (where the gradlew file is
    located)
     */
    private static File findProjectRoot() {
        // Start with current directory
        File current = new File(".").getAbsoluteFile();

        // Check for gradlew in current directory
        if (new File(current, isWindows() ? "gradlew.bat" :
"gradlew").exists()) {
            return current;
        }

        // Look in parent directories
        while (current != null) {
            File gradleWrapperFile = new File(current, isWindows() ?
"gradlew.bat" : "gradlew");
            if (gradleWrapperFile.exists()) {
                return current;
            }
            current = current.getParentFile();
        }

        // If we can't find it, just return the current directory
        return new File(".").getAbsoluteFile();
    }

    private static List<String> buildCommand(String javaHome, String
gradleWrapper, String[] gradleArgs) {
        List<String> command = new ArrayList<>();

        if (isWindows()) {

```



```

        command.add("cmd");
        command.add("/c");

        if (javaHome != null && !javaHome.isEmpty()) {
            command.add("set");
            command.add("JAVA_HOME=" + javaHome);
            command.add("&&");
        }

        command.add(gradleWrapper);

        // Add Gradle arguments
        for (String arg : gradleArgs) {
            command.add(arg);
        }

    } else {
        command.add("/bin/sh");
        command.add("-c");

        StringBuilder shellCommand = new StringBuilder();

        if (javaHome != null && !javaHome.isEmpty()) {
            shellCommand.append("export
JAVA_HOME=").append(javaHome).append("' && ");
        }

        shellCommand.append(gradleWrapper);

        // Add Gradle arguments
        for (String arg : gradleArgs) {
            shellCommand.append("' ").append(arg.replace("'",
"\\'")).append("'");
        }

        command.add(shellCommand.toString());
    }

    return command;
}

private static boolean isWindows() {
    return
System.getProperty("os.name").toLowerCase().contains("win");
}
}

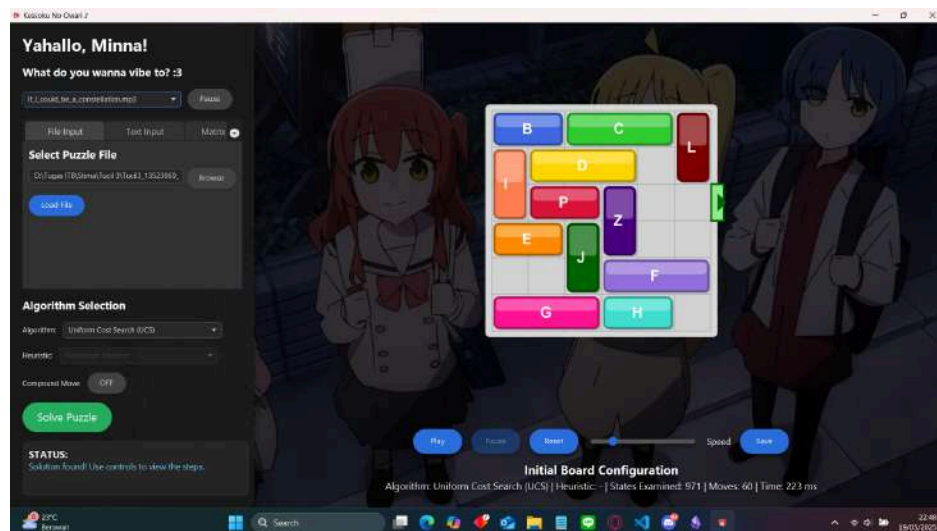
```


BAB IV: TEST CASES

Berikut merupakan output dari percobaan masing-masing *test cases* dengan parameter berbeda yang kami lakukan. *Input* dari *test case* berada di dalam folder [../test/input](#) sedangkan hasil dari *test case* berada di dalam folder [../test/output](#). Untuk *input file*, harus dalam bentuk .txt.

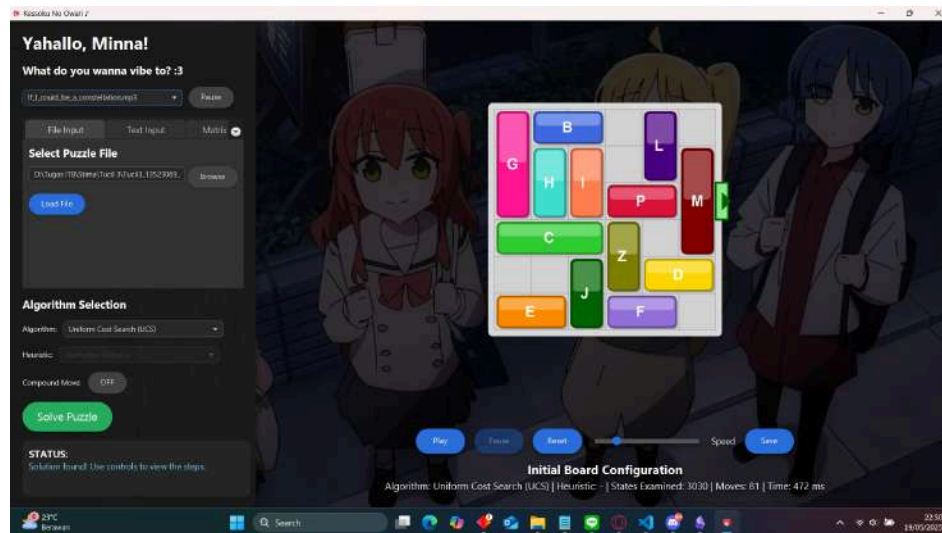
4.1 Solusi Berhasil Ditemukan

4.1.1 Windows



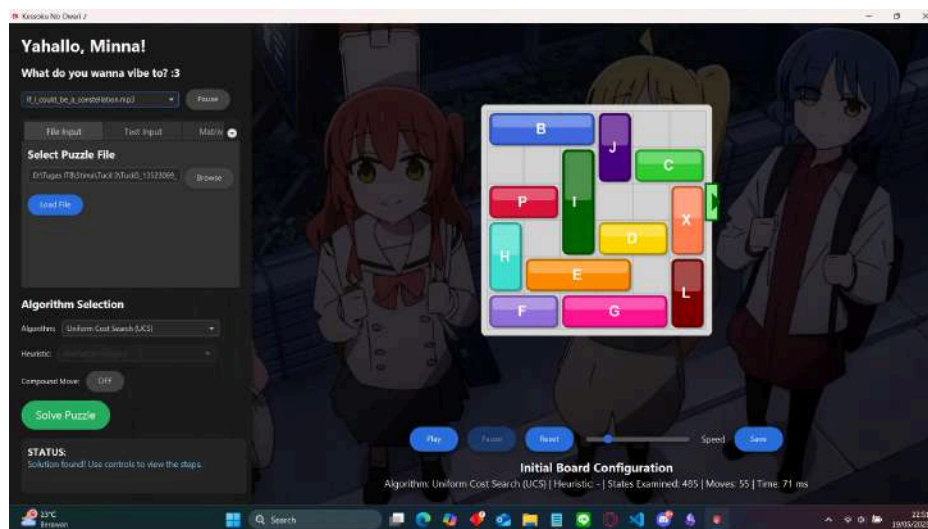
Gambar 8. Testcase 1.txt

(Sumber: arsip penulis)



Gambar 9. Testcase 2.txt

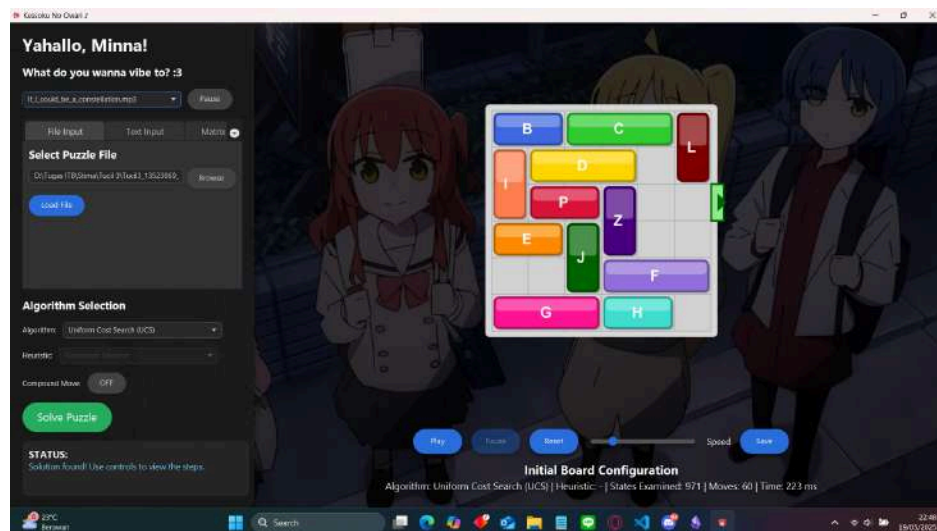
(Sumber: arsip penulis)



Gambar 10. Testcase 3.txt

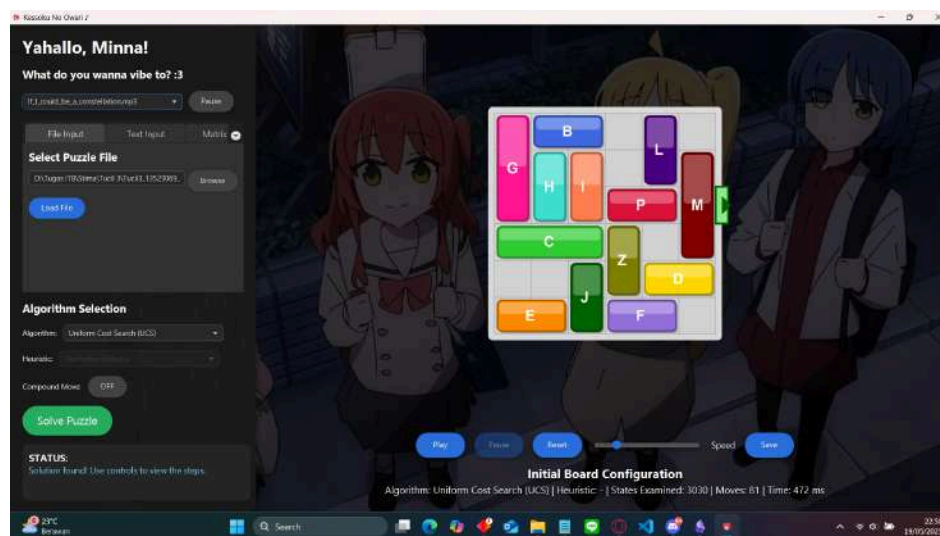
(Sumber: arsip penulis)

4.1.2 Linux



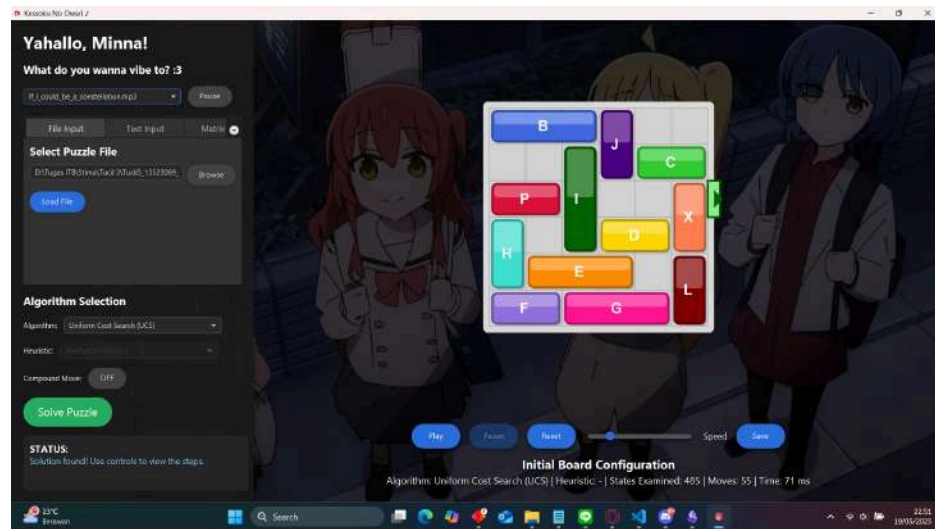
Gambar 11. Testcase 1.txt

(Sumber: arsip penulis)



Gambar 12. Testcase 2.txt

(Sumber: arsip penulis)

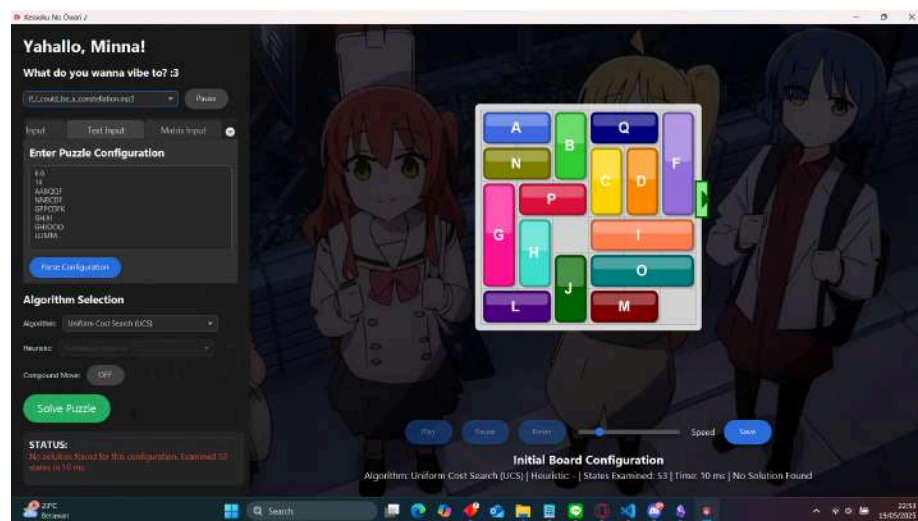


Gambar 13. Testcase 3.txt

(Sumber: arsip penulis)

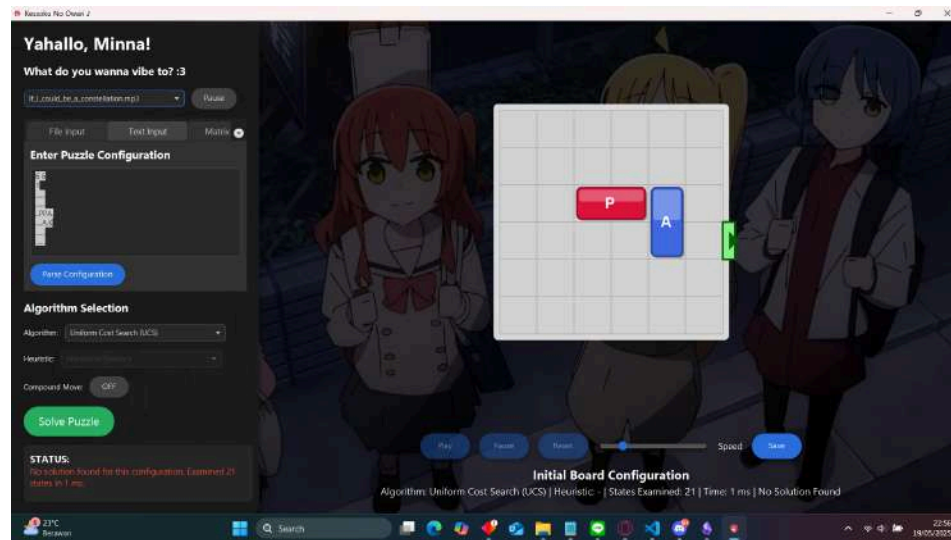
4.2 Solusi Gagal Ditemukan

4.2.1 Windows



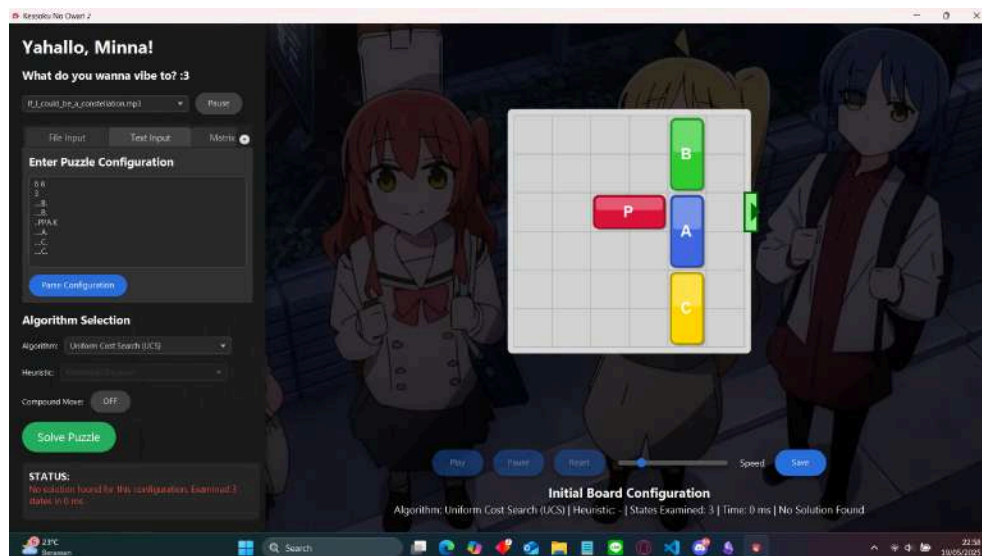
Gambar 14. Testcase sempit.txt

(Sumber: arsip penulis)



Gambar 15. Testcase gaksejajar.txt

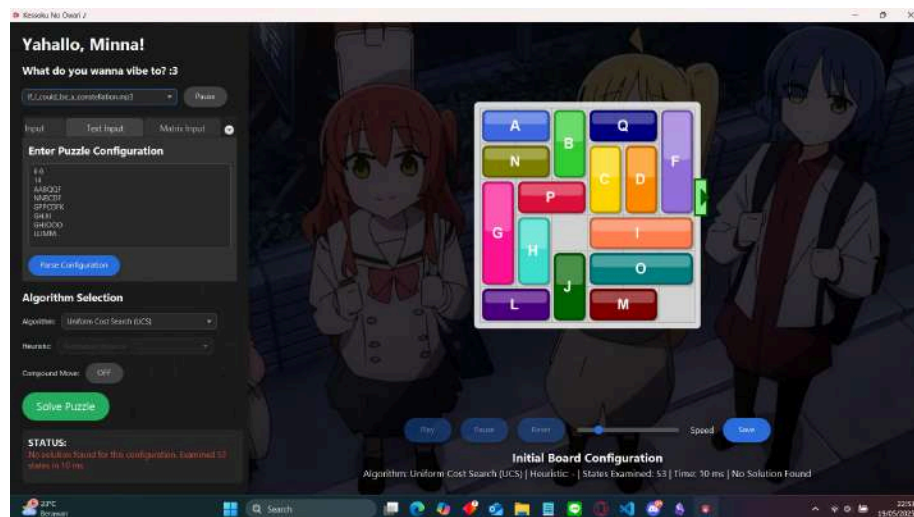
(Sumber: arsip penulis)



Gambar 16. Testcase kenablokirnjir.txt

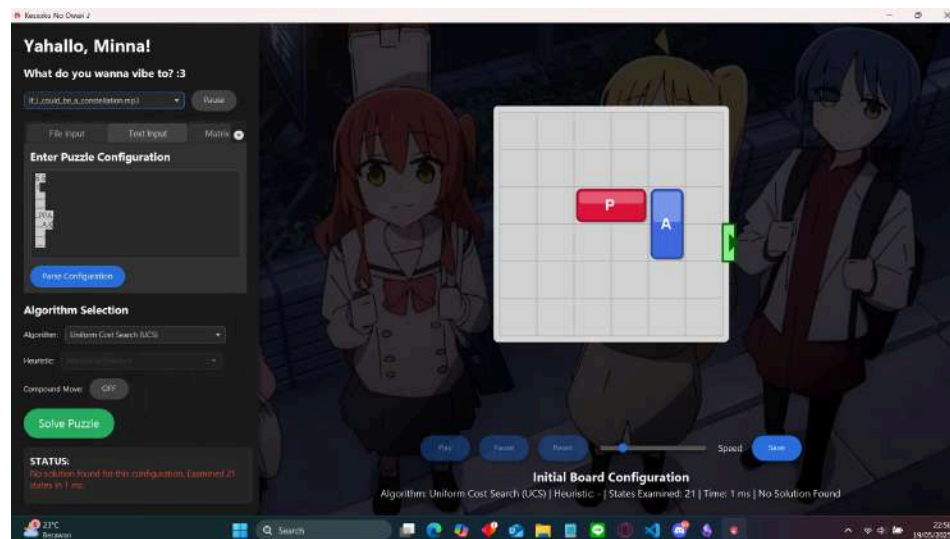
(Sumber: arsip penulis)

4.2.2 Linux



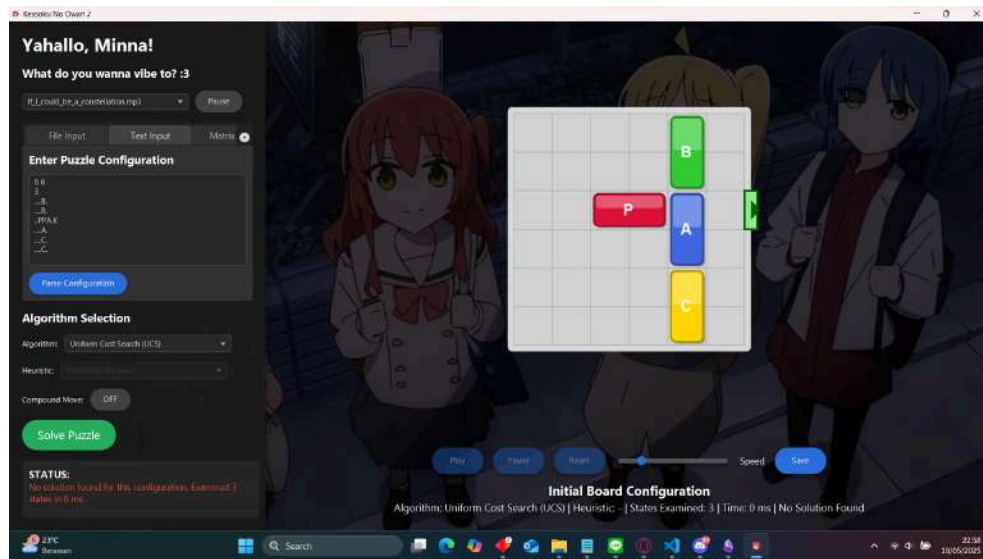
Gambar 17. Testcase sempit.txt

(Sumber: arsip penulis)



Gambar 18. Testcase gaksejajar.txt

(Sumber: arsip penulis)



Gambar 19. Testcase kenablockinjr.txt

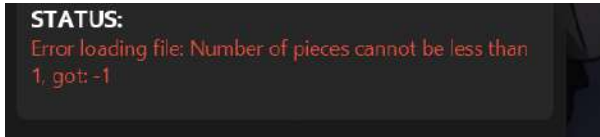

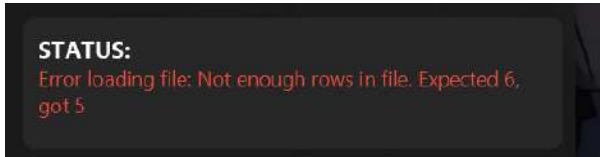
(Sumber: arsip penulis)

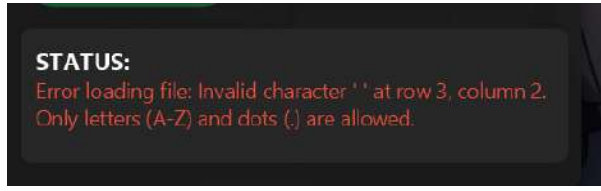


4.3 User Input Error

Kasus Error	Hasil
<p>Format input tidak sesuai jika posisi exit berada di kiri</p> <p>Testcase : 4.txt</p> <pre>6 6 11 AAB..F ..BCDF KGPPCDF GH.III GHJ... LLJMM.</pre>	<p>STATUS:</p> <p>Error loading file: Invalid format: When K is at the left, all other rows must have exactly one leading space. Row 1 has incorrect spacing.</p>
<p>Terdapat piece yang tidak linear</p> <p>Testcase : 5.txt</p> <pre>8 9 9 PP.AABB..K ..CCDEEF. GGHHDEEF.</pre>	<p>STATUS:</p> <p>Error loading file: Invalid piece configuration: Piece 'E' is not linear. It spans 2 rows and 2 columns. Pieces must be either 1xN or Nx1.</p>

<pre> I I H H D J J L . M M N N O J J L . M M Q Q O S S R . T T U U V V W R . T T U U V V W W . </pre>	
<p>Terdapat pieces yang terpisah atau piece dengan ID duplikat</p> <p>Testcase : a.txt</p> <pre> 6 6 11 K A A B . . F . . B C D F G P P C D F G H . I I I G H J . . . L L J A A . </pre>	<p>STATUS:</p> <p>Error loading file: Invalid piece configuration: Piece 'A' is not linear. It spans 6 rows and 5 columns. Pieces must be either 1×N or N×1.</p>
<p>Terdapat lebih dari 1 exit</p> <p>Testcase : edge1.txt</p> <pre> 6 6 12 G B B . L . G H I . L M G H I P P M K C C C Z . M . . J Z D D E E J F F K K </pre>	<p>STATUS:</p> <p>Error loading file: Multiple exit positions (K) found: RIGHT (row 2), RIGHT (row 5). The Rush Hour puzzle must have exactly one exit position.</p>
<p>Terdapat lebih dari 1 exit</p> <p>Testcase : edge3.txt</p> <pre> 6 6 12 G B B . L . G H I . L M G H I P P M K C C C Z . M . . J Z D D K E E J F F . </pre>	<p>STATUS:</p> <p>Error loading file: Multiple exit positions (K) found: RIGHT (row 2), RIGHT (row 5). The Rush Hour puzzle must have exactly one exit position.</p>

<p>Jumlah non-primary pieces tidak sesuai dengan yang diberikan pada input</p> <p>Testcase : edge6.txt</p> <pre>6 6 13 GBB.L. GHI.LM GHIPPMKYY CCCZ.M ..JZDD EEJFF.</pre>	<p>STATUS:</p> <p>Error loading file: Number of non-primary pieces mismatch. Expected 13, but found 12</p>
<p>Tidak ditemukan <i>primary piece</i> (P) pada input</p> <p>Testcase : edge7.txt</p> <pre>6 6 12 GBB.L. GHI.LM GHI..MKPP CCCZ.M ..JZDD EEJFF.</pre>	<p>STATUS:</p> <p>Error loading file: No primary piece (P) found in the board configuration</p>
<p>Format input salah : baris pertama tidak memuat 2 buah integer yang merupakan row dan column dari papan permainan atau baris pertama kosong</p> <p>Testcase : error1.txt</p> <pre>11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.</pre>	<p>STATUS:</p> <p>Error loading file: First line must contain exactly 2 integers for board dimensions</p>
<p>Format input salah : input bilangan negatif</p> <p>Testcase : error2.txt</p> <pre>6 -69 11 AAB..F ..BCDF</pre>	<p>STATUS:</p> <p>Error loading file: Number of columns must be greater than 0, got: -69</p>

GPPCDFK GH.III GHJ... LLJMM.	
<p>Banyak non-primary piece kurang dari 1 atau negatif</p> <p>Testcase : error4.txt</p> <pre> 6 6 -1 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM. </pre>	
<p>Format input salah : baris kedua tidak memuat 2 buah integer yang merupakan row dan column dari papan permainan atau baris pertama kosong</p> <p>Testcase : error5.txt</p> <pre> 6 6 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM. </pre>	
<p>Format input salah : Banyak baris tidak sesuai</p> <p>Testcase : error7.txt</p> <pre> 6 6 9 K AAB..F ..BCDF GPPCDF GH.III GHJJ.. </pre>	

<p>Format input salah : Terdapat karakter yang dilarang pada input papan</p> <p>Testcase : error9.txt</p> <pre>6 6 11 AAB..F ..BCDF G CDFK GH.III GHJPP. LLJMM.</pre>	
<p>Format input salah : Posisi k salah</p> <p>Testcase : error12.txt</p> <pre>6 6 10 AAB..F ..BCDF G..CDF GH.III GH.PP. JJJMM. K</pre>	
<p>File masukan kosong</p> <p>Testcase : error13.txt</p>	

4.4 Perbandingan Algoritma yang digunakan

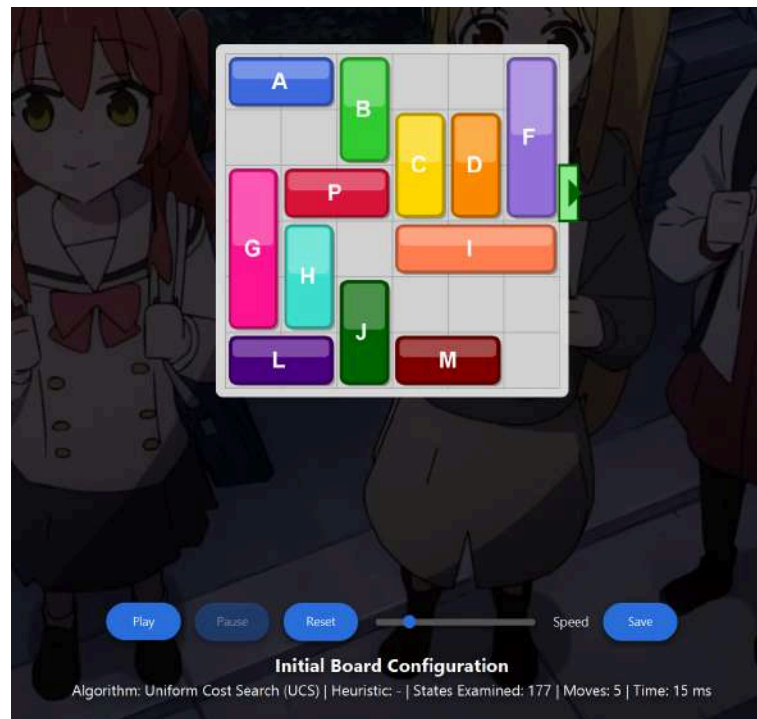
4.4.1 Testcase default.txt

Algoritma	Hasil
-----------	-------

Algoritma : Uniform Cost Search
 Heuristik : -
 Test Case:
 6 6
 11
 AAB..F
 ..BCDF
 GPPCDFK
 GH.III
 GHJ...
 LLJMM.

States : 177
 Moves: 5
 Time : 15 ms

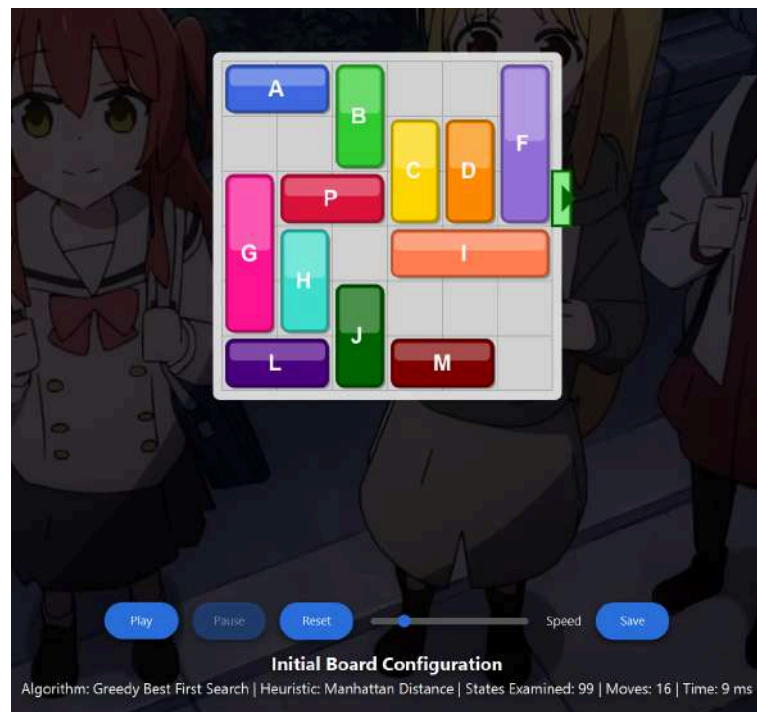
Output file : algo_default_ucs.txt



Algoritma : Greedy Best First Search
 Heuristik : Manhattan Distance
 Test Case:
 6 6
 11
 AAB..F
 ..BCDF
 GPPCDFK
 GH.III
 GHJ...
 LLJMM.

States : 99
 Moves: 16
 Time : 9 ms

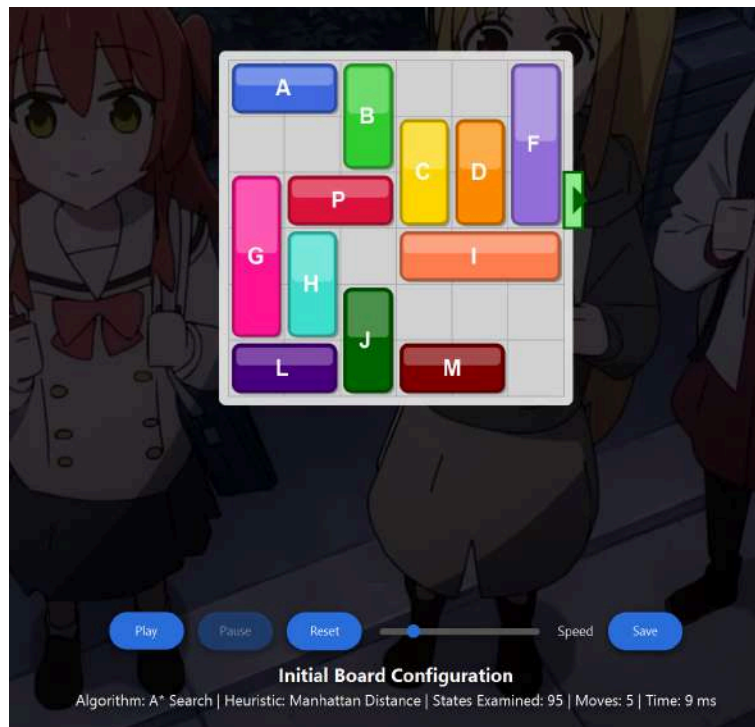
Output file :
 algo_default_GBFS.txt



Algoritma : A*
 Heuristik : Manhattan Distance
 Test Case:
 6 6
 11
 AAB..F
 ..BCDF
 GPPCDFK
 GH.III
 GHJ...
 LLJMM.

States : 95
 Moves: 5
 Time : 9 ms

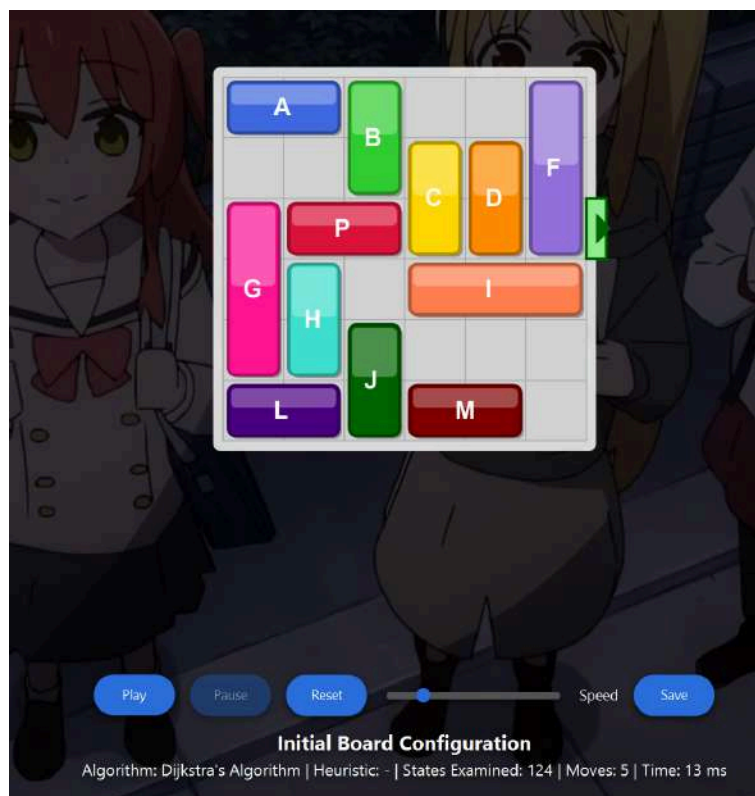
Output file : algo_default_astar.txt

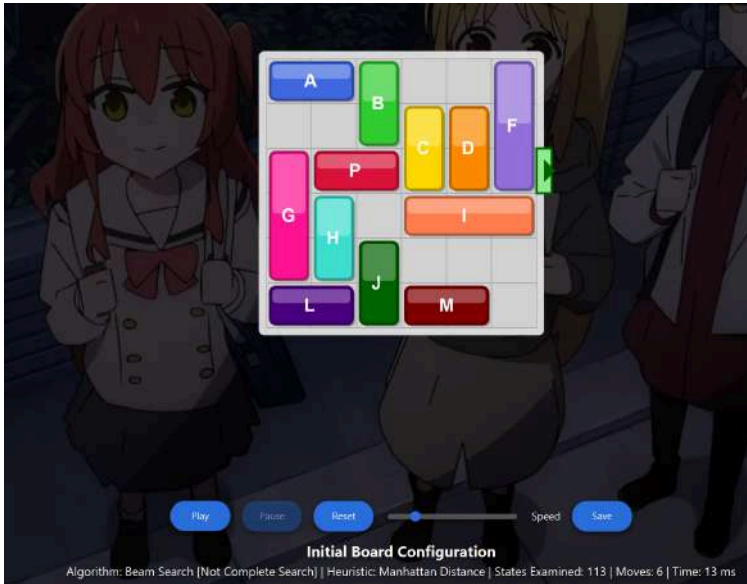
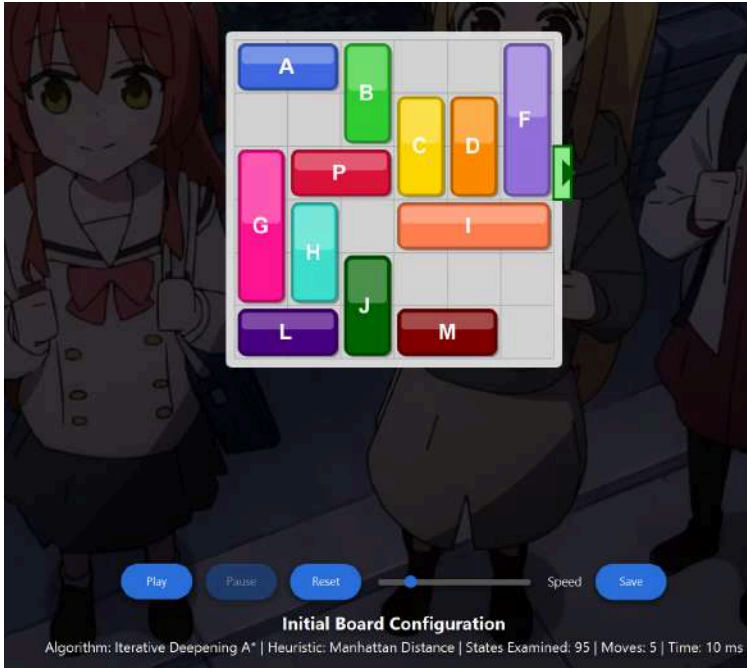


Algoritma : Dijkstra
 Heuristik : -
 Test Case:
 6 6
 11
 AAB..F
 ..BCDF
 GPPCDFK
 GH.III
 GHJ...
 LLJMM.

States : 124
 Moves: 5
 Time : 13 ms

Output file :
 algo_default_Dijkstra.txt



<p> Algoritma : Beam Search Heuristik : Manhattan Distance Test Case: 6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM. </p> <p> States : 113 Moves: 6 Time : 13 ms </p> <p>Output file : algo_default_beam.txt</p>	
<p> Algoritma : IDA* Heuristik : Manhattan Distance Test Case: 6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM. </p> <p> States : 95 Moves: 5 Time : 10 ms </p> <p>Output file : algo_default_IDA.txt</p>	

4.4.2 Testcase misteri4.txt

Algoritma	Hasil
-----------	-------

Algoritma : Uniform Cost Search

Heuristik : -

Test Case:

6 6

12

BBBZLM

HCCZLM

H.PPLMK

DDJ...

.IJEE.

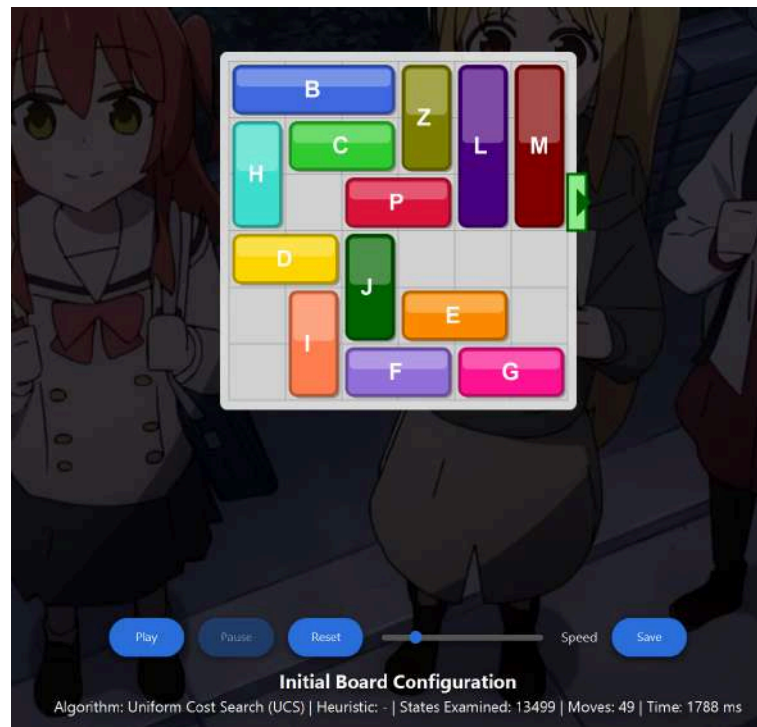
.IFFGG

States : 13499

Moves: 49

Time : 1788 ms

Output file : algo_misteri4_ucs.txt



Algoritma : Greedy Best First Search

Heuristik : Manhattan Distance

Test Case:

6 6

12

BBBZLM

HCCZLM

H.PPLMK

DDJ...

.IJEE.

.IFFGG

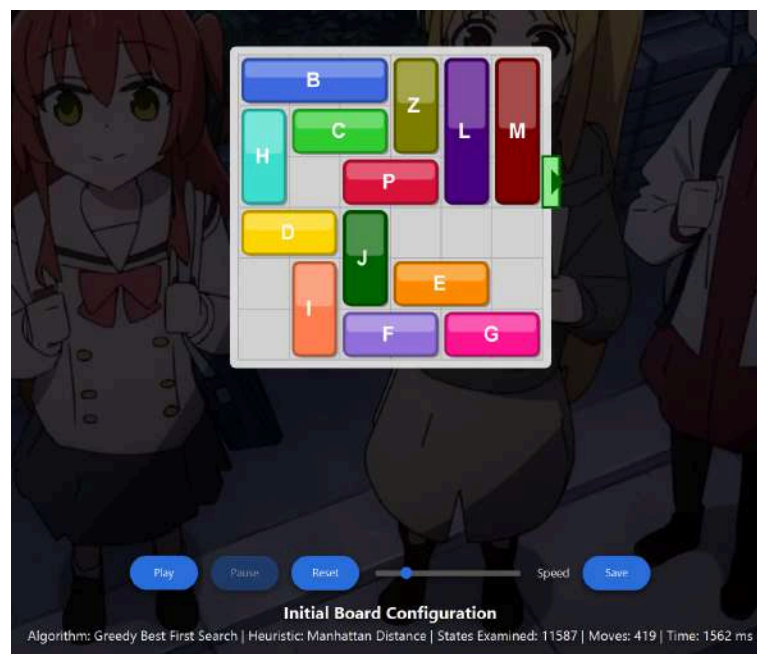
States : 11587

Moves: 419

Time : 1562 ms

Output file :

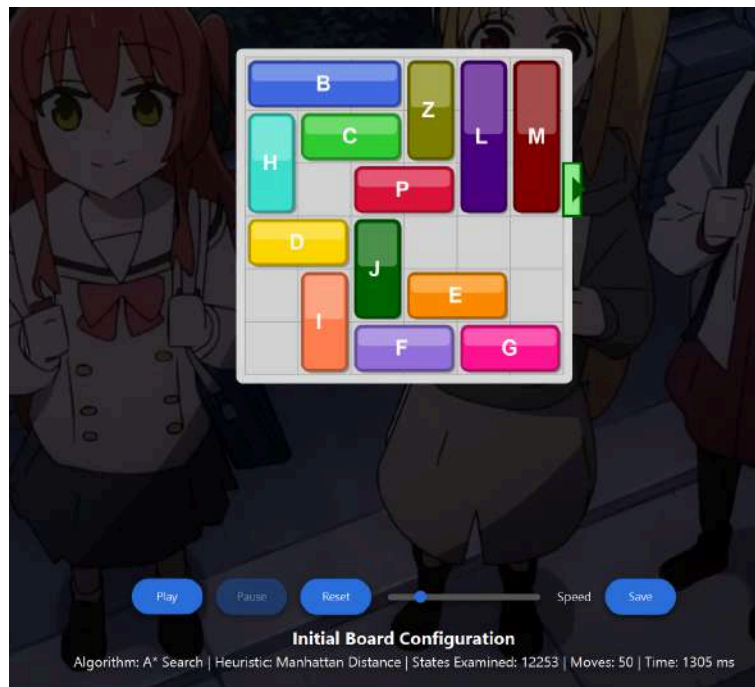
algo_misteri4_GBFS.txt



Algoritma : A*
 Heuristik : Manhattan Distance
 Test Case:
 6 6
 12
 BBBZLM
 HCCZLM
 H.PPLMK
 DDJ...
 .IJEE.
 .IFFGG

States : 12253
 Moves: 50
 Time : 1305 ms

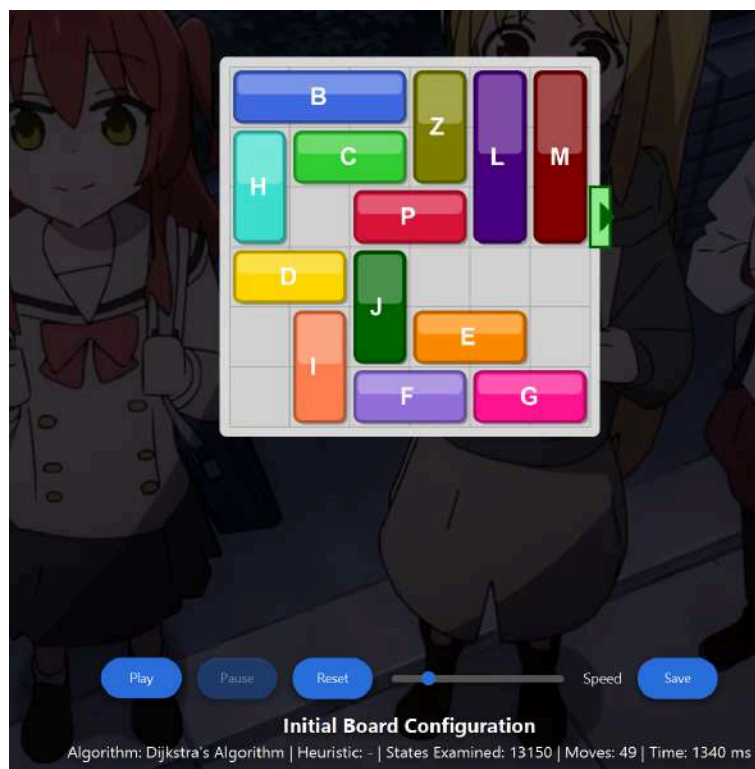
Output file :
 algo_misteri4_astar.txt

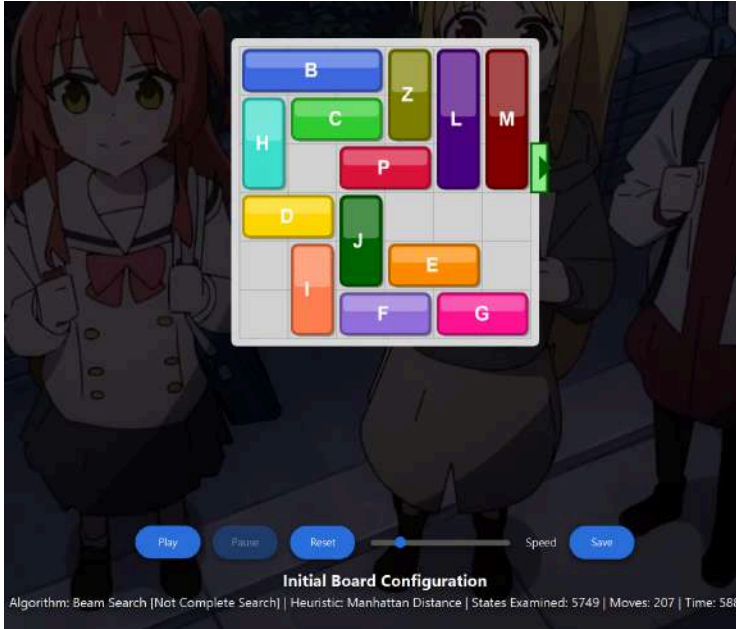
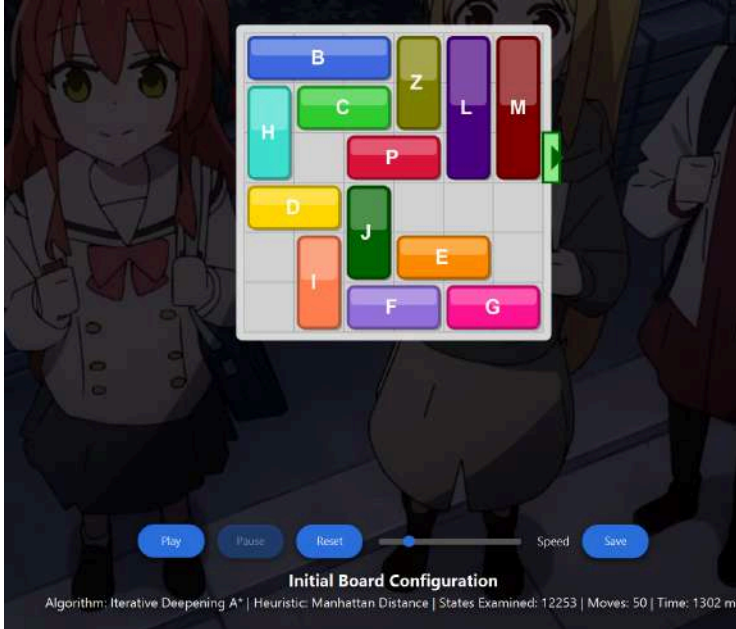


Algoritma : Dijkstra
 Heuristik : -
 Test Case:
 6 6
 12
 BBBZLM
 HCCZLM
 H.PPLMK
 DDJ...
 .IJEE.
 .IFFGG

States : 13150
 Moves: 49
 Time : 1340 ms

Output file :
 algo_misteri4_Dijkstra.txt



<p> Algoritma : Beam Search Heuristik : Manhattan Distance Test Case: 6 6 12 BBBZLM HCCZLM H.PPLMK DDJ... .IJEE. .IFFGG </p> <p> States : 5749 Moves: 207 Time : 588 ms </p> <p> Output file : algo_misteri4_beam.txt </p>	
<p> Algoritma : IDA* Heuristik : Manhattan Distance Test Case: 6 6 12 BBBZLM HCCZLM H.PPLMK DDJ... .IJEE. .IFFGG </p> <p> States : 12253 Moves: 50 Time : 1302 ms </p> <p> Output file : algo_misteri4_IDA.txt </p>	

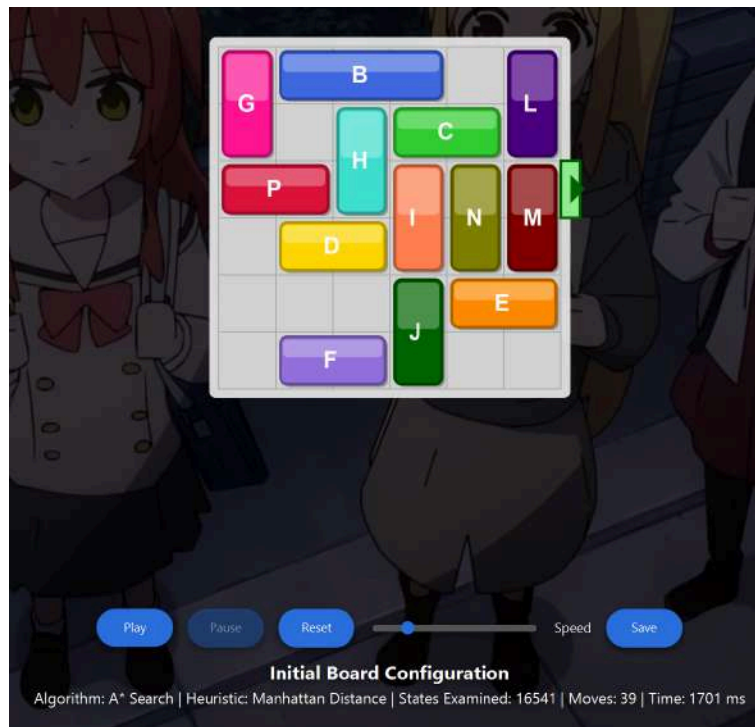
4.4.3 Testcase tebaktebakanyuk_adahasilnyaapaenggakyya.txt

Algoritma	Hasil
<p>Algoritma : Uniform Cost Search</p> <p>Heuristik : -</p> <p>Test Case:</p> <p>6 6</p> <p>12</p> <p>G B B B . L</p> <p>G . H C C L</p> <p>P P H I N M K</p> <p>. D D I N M</p> <p>... J E E</p> <p>. F F J . .</p> <p>States : 21168</p> <p>Moves: 38</p> <p>Time : 2202 ms</p> <p>Output file : algo_tebak_ucs.txt</p>	
<p>Algoritma : Greedy Best First Search</p> <p>Heuristik : Manhattan Distance</p> <p>Test Case:</p> <p>6 6</p> <p>12</p> <p>G B B B . L</p> <p>G . H C C L</p> <p>P P H I N M K</p> <p>. D D I N M</p> <p>... J E E</p> <p>. F F J . .</p> <p>States : 5466</p> <p>Moves: 933</p> <p>Time : 607 ms</p> <p>Output file : algo_tebak_GBFS.txt</p>	

Algoritma : A*
 Heuristik : Manhattan Distance
 Test Case:
 6 6
 12
 GBBB.L
 G.HCCL
 PPHINMK
 .DDINM
 ...JEE
 .FFJ..

States : 16541
 Moves: 39
 Time : 1701 ms

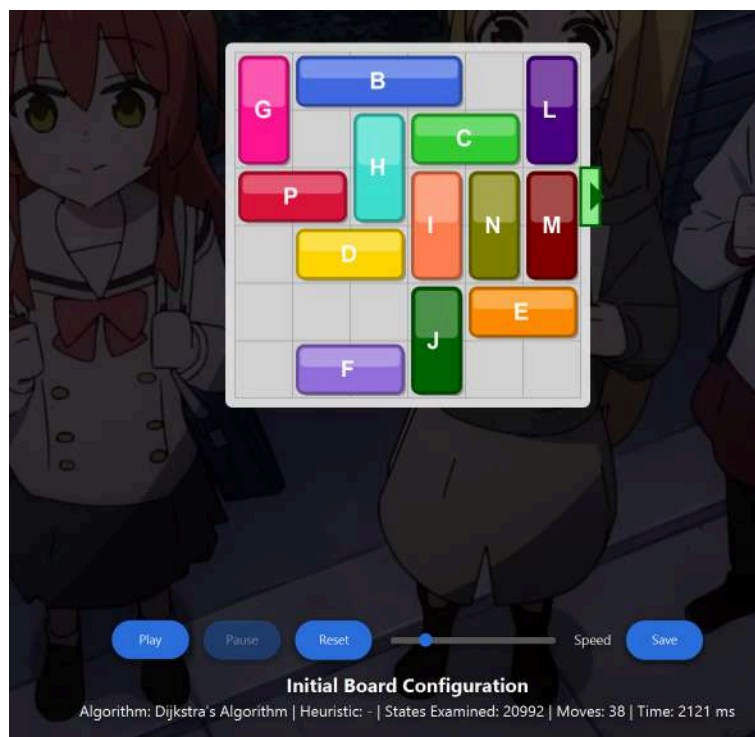
Output file : algo_tebak_astar.txt



Algoritma : Dijkstra
 Heuristik : -
 Test Case:
 6 6
 12
 GBBB.L
 G.HCCL
 PPHINMK
 .DDINM
 ...JEE
 .FFJ..

States : 20992
 Moves: 38
 Time : 2121 ms

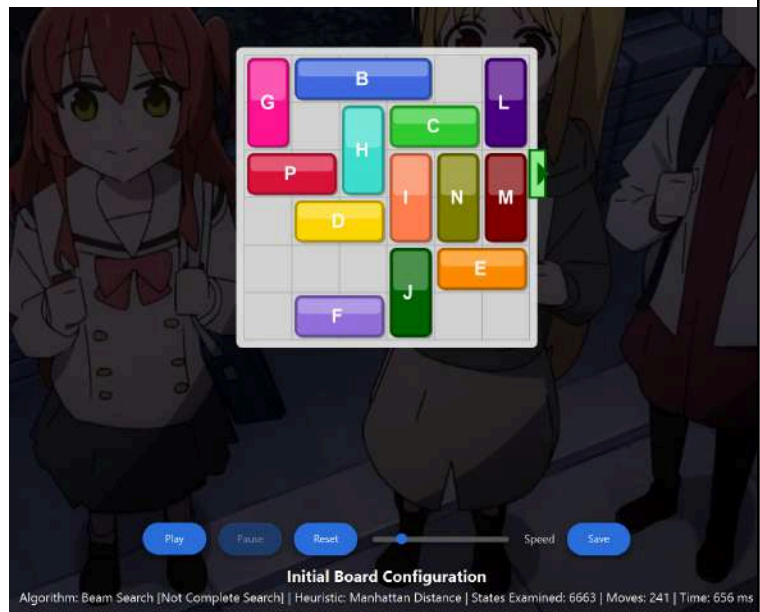
Output file :
 algo_tebak_Dijkstra.txt



Algoritma : Beam Search
 Heuristik : Manhattan Distance
 Test Case:
 6 6
 12
 GBBB.L
 G.HCCL
 PPHINMK
 .DDINM
 ...JEE
 .FFJ..

States : 6663
 Moves: 241
 Time : 656 ms

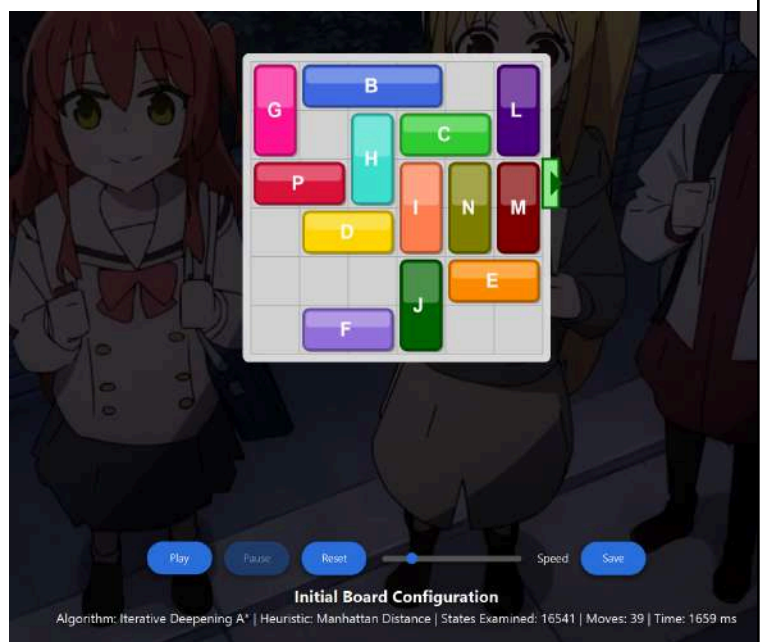
Output file : algo_tebak_beam.txt



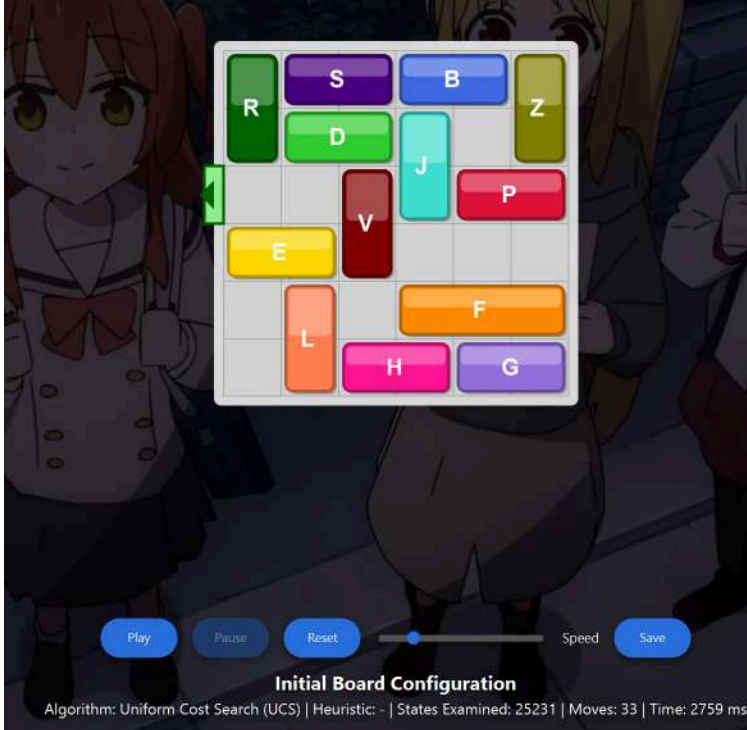
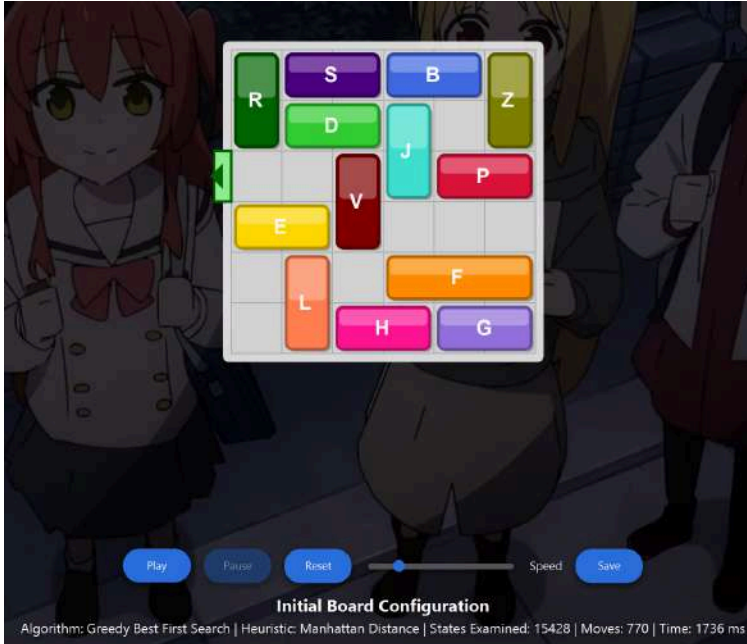
Algoritma : IDA*
 Heuristik : Manhattan Distance
 Test Case:
 6 6
 12
 GBBB.L
 G.HCCL
 PPHINMK
 .DDINM
 ...JEE
 .FFJ..

States : 16541
 Moves: 39
 Time : 1659 ms

Output file : algo_tebak_IDA.txt



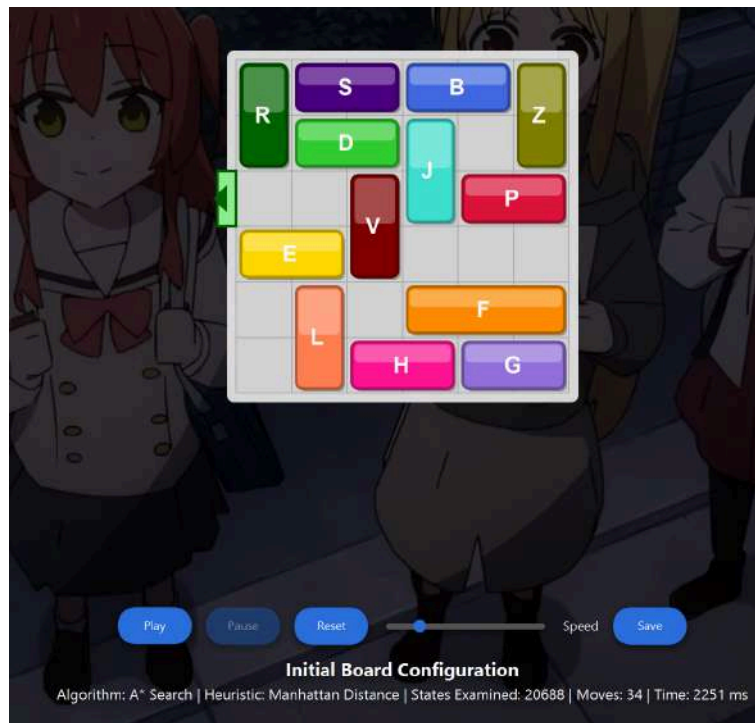
4.4.4 Testcase wahgaknormalini😂.txt

Algoritma	Hasil
<p>Algoritma : Uniform Cost Search Heuristik : - Test Case: 6 6 12 RSSBBZ RDDJ.Z K..VJPP EEV... .L.FFF .LHHGG</p> <p>States : 25231 Moves: 33 Time : 2759 ms</p> <p>Output file : algo_normalkok_ucs.txt</p>	
<p>Algoritma : Greedy Best First Search Heuristik : Manhattan Distance Test Case: 6 6 12 RSSBBZ RDDJ.Z K..VJPP EEV... .L.FFF .LHHGG</p> <p>States : 15428 Moves: 770 Time : 1736 ms</p> <p>Output file : algo_normalkok_GBFS.txt</p>	

Algoritma : A*
 Heuristik : Manhattan Distance
 Test Case:
 6 6
 12
 RSSBBZ
 RDDJ.Z
 K..VJPP
 EEV...
 .L.FFF
 .LHHGG

States : 20688
 Moves: 34
 Time : 2251 ms

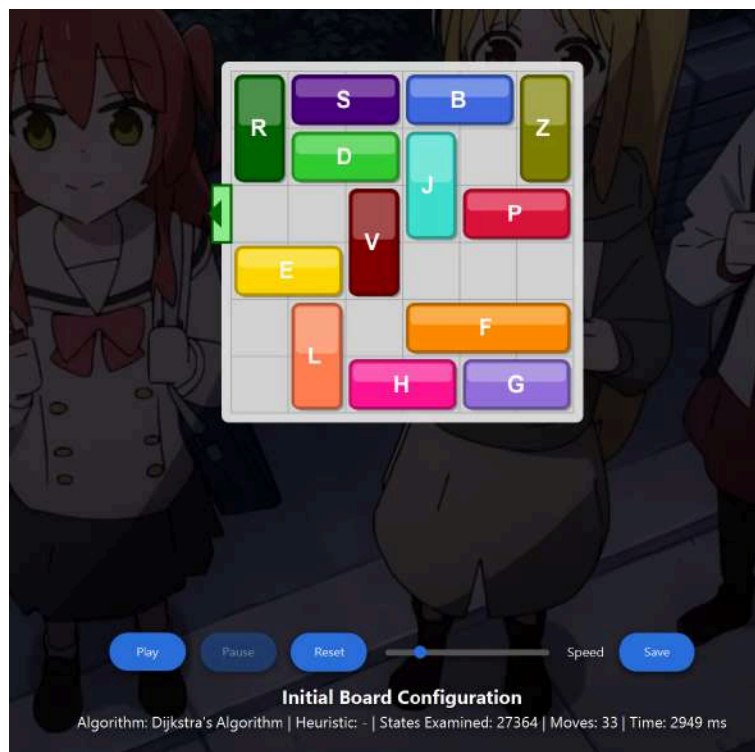
Output file :
 algo_normalkok_astar.txt



Algoritma : Dijkstra
 Heuristik : -
 Test Case:
 6 6
 12
 RSSBBZ
 RDDJ.Z
 K..VJPP
 EEV...
 .L.FFF
 .LHHGG

States : 27364
 Moves: 33
 Time : 2949 ms

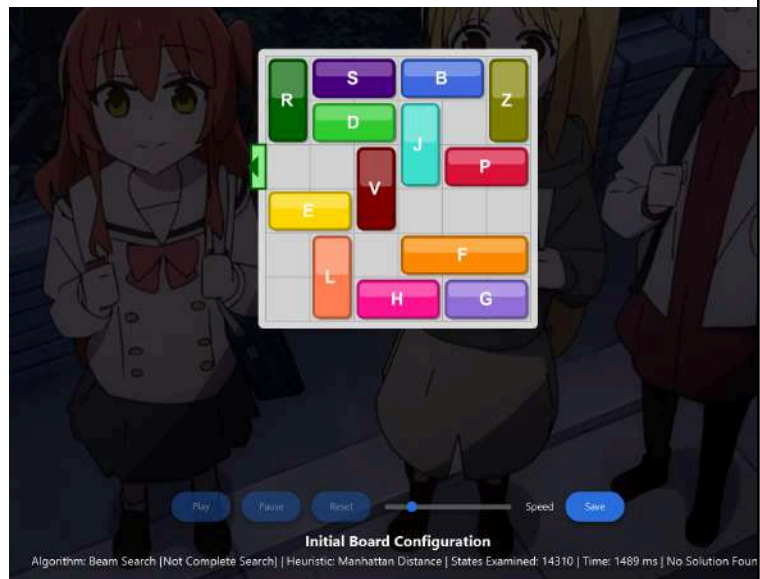
Output file :
 algo_normalkok_Dijkstra.txt



Algoritma : Beam Search
 Heuristik : Manhattan Distance
 Test Case:
 6 6
 12
 RSSBBZ
 RDDJ.Z
 K..VJPP
 EEV...
 .L.FFF
 .LHHGG

States : 14310
 Moves: No solution
 Time : 1489 ms

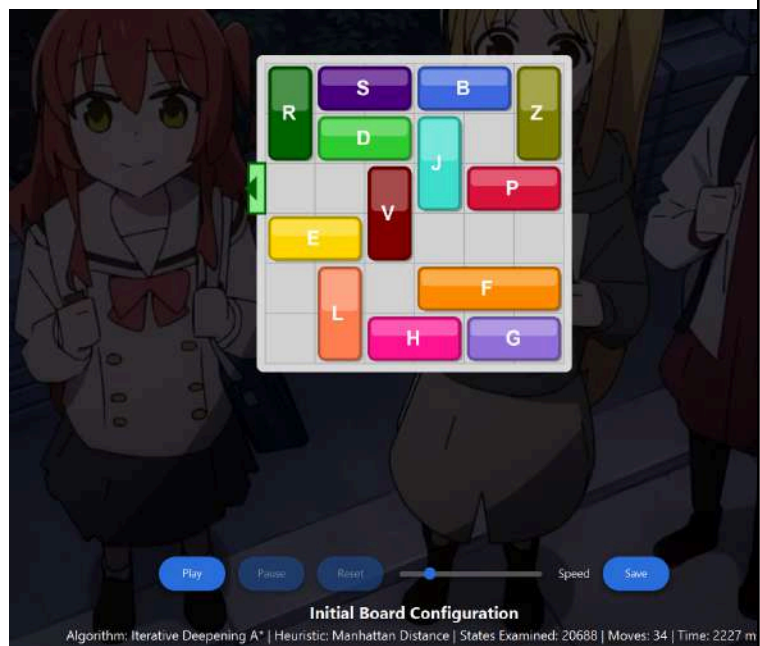
Output file :
 algo_normalkok_beam.txt



Algoritma : IDA*
 Heuristik : Manhattan Distance
 Test Case:
 6 6
 12
 RSSBBZ
 RDDJ.Z
 K..VJPP
 EEV...
 .L.FFF
 .LHHGG

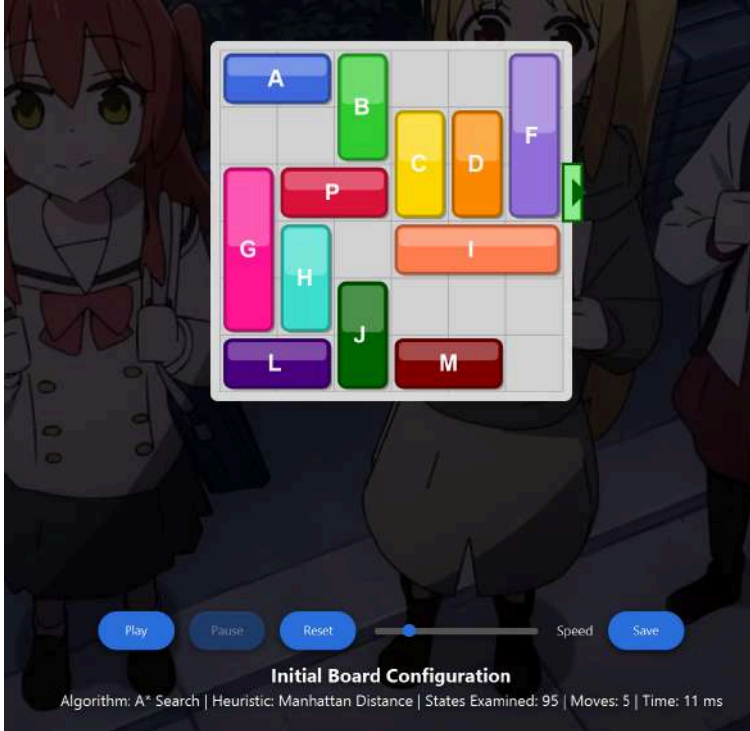
States : 20688
 Moves: 34
 Time : 2227 ms

Output file :
 algo_normalkok_IDA.txt



4.5 Perbandingan Heuristik

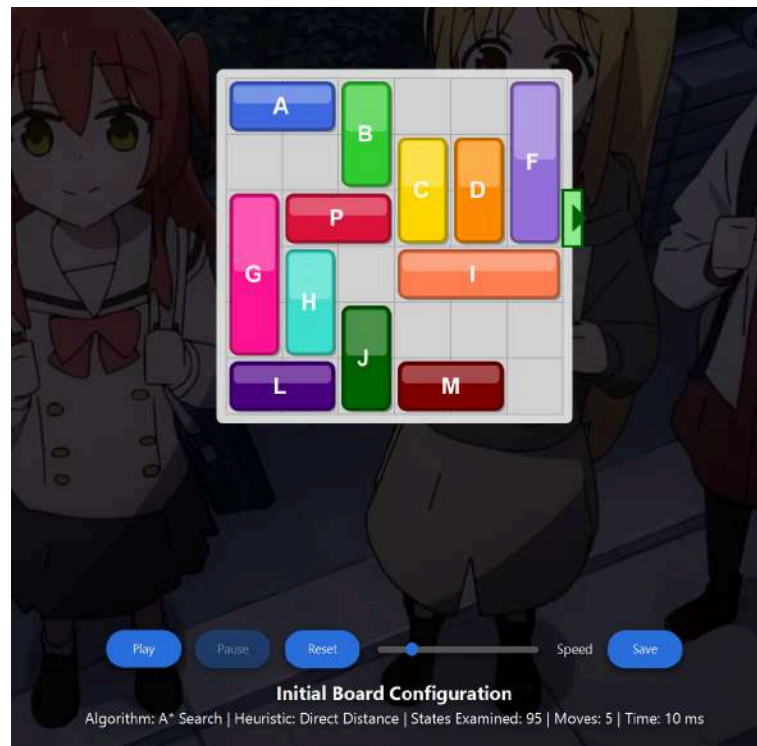
4.5.1 Testcase default.txt

Heuristik	Hasil
<p>Algoritma : A*</p> <p>Heuristik : Manhattan Distance</p> <p>Test Case:</p> <p>6 6</p> <p>11</p> <p>AAB..F</p> <p>..BCDF</p> <p>GPPCDFK</p> <p>GH.III</p> <p>GHJ...</p> <p>LLJMM.</p>	

Algoritma : A*
 Heuristik : Direct Distance
 Test Case:
 6 6
 11
 AAB..F
 ..BCDF
 GPPCDFK
 GH.III
 GHJ...
 LLJMM.

States : 95
 Moves: 5
 Time : 10 ms

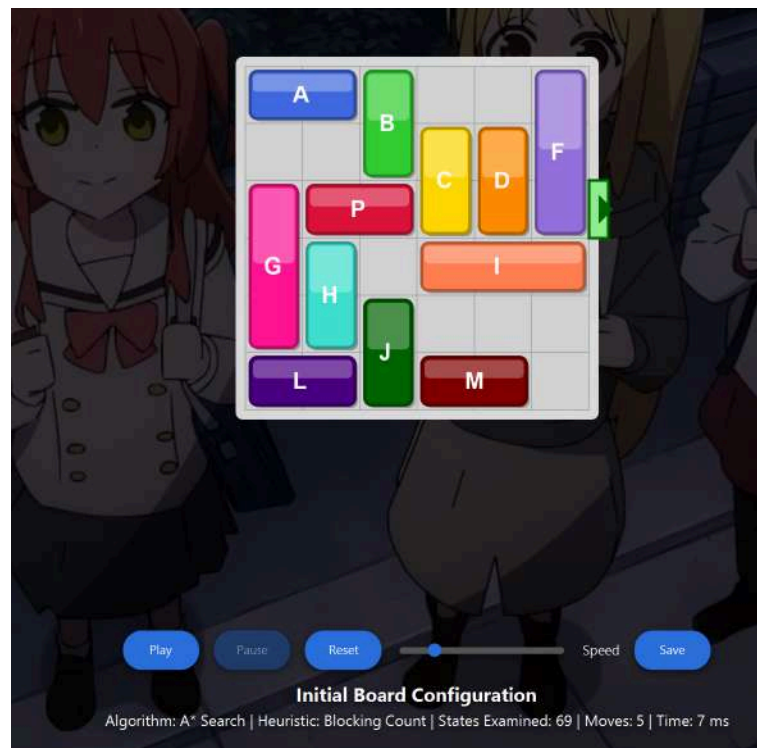
Output file : direct_default.txt



Algoritma : A*
 Heuristik : Blocking Count
 Test Case:
 6 6
 11
 AAB..F
 ..BCDF
 GPPCDFK
 GH.III
 GHJ...
 LLJMM.

States : 69
 Moves: 5
 Time : 7 ms

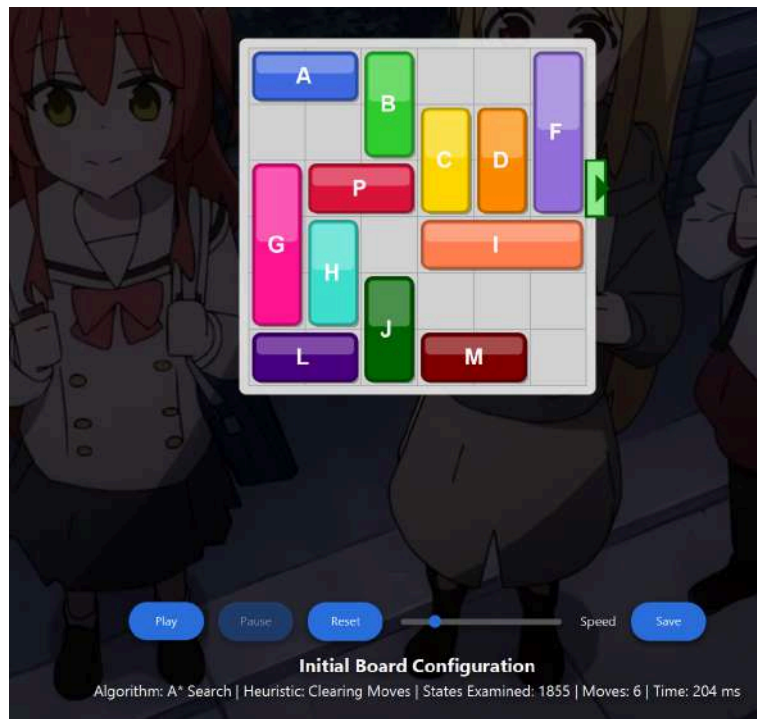
Output file : block_default.txt



Algoritma : A*
 Heuristik : Clearing Moves
 Test Case:
 6 6
 11
 AAB..F
 ..BCDF
 GPPCDFK
 GH.III
 GHJ...
 LLJMM.

States : 1855
 Moves: 6
 Time : 204 ms

Output file : clear_default.txt



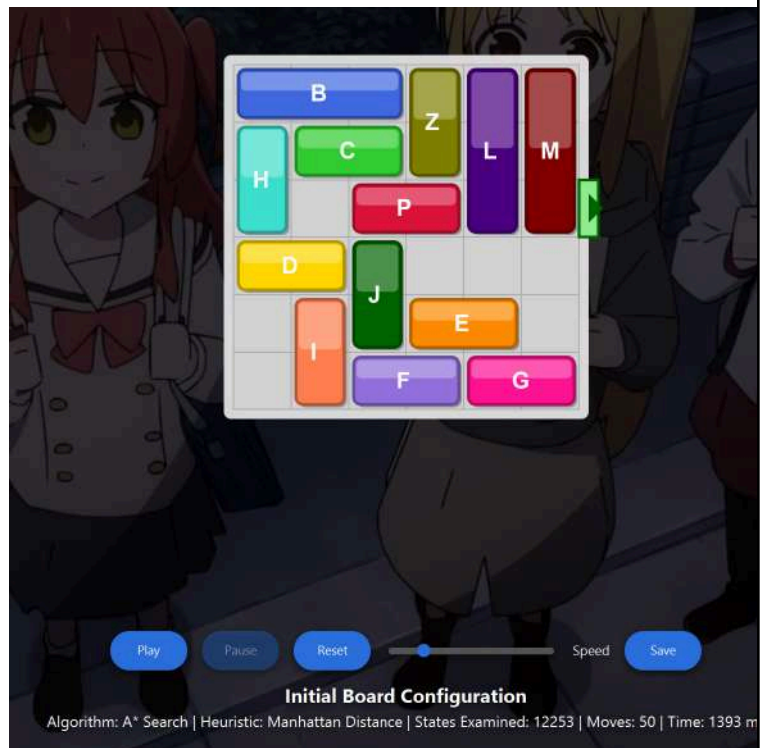
4.5.2 Testcase misteri4.txt

Heuristik	Hasil
-----------	-------

Algoritma : A*
 Heuristik : Manhattan Distance
 Test Case:
 6 6
 12
 BBBZLM
 HCCZLM
 H.PPLMK
 DDJ...
 .IJEE.
 .IFFGG

States : 12253
 Moves: 50
 Time : 1393 ms

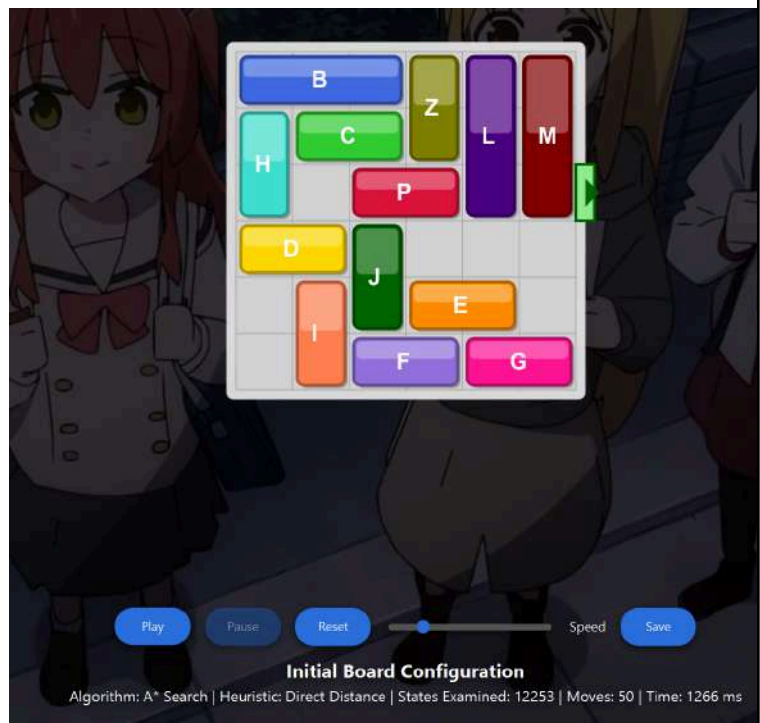
Output file :
 manhattan_misteri4.txt



Algoritma : A*
 Heuristik : Direct Distance
 Test Case:
 6 6
 12
 BBBZLM
 HCCZLM
 H.PPLMK
 DDJ...
 .IJEE.
 .IFFGG

States : 12253
 Moves: 50
 Time : 1266 ms

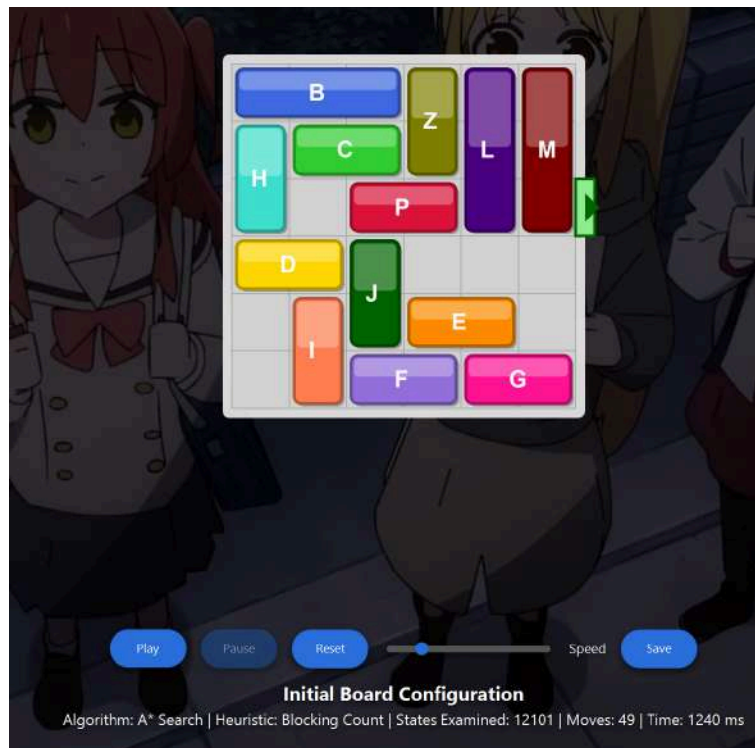
Output file : direct_misteri4.txt



Algoritma : A*
 Heuristik : Blocking Count
 Test Case:
 6 6
 12
 BBBZLM
 HCCZLM
 H.PPLMK
 DDJ...
 .IJEE.
 .IFFGG

States : 12101
 Moves: 49
 Time : 1240 ms

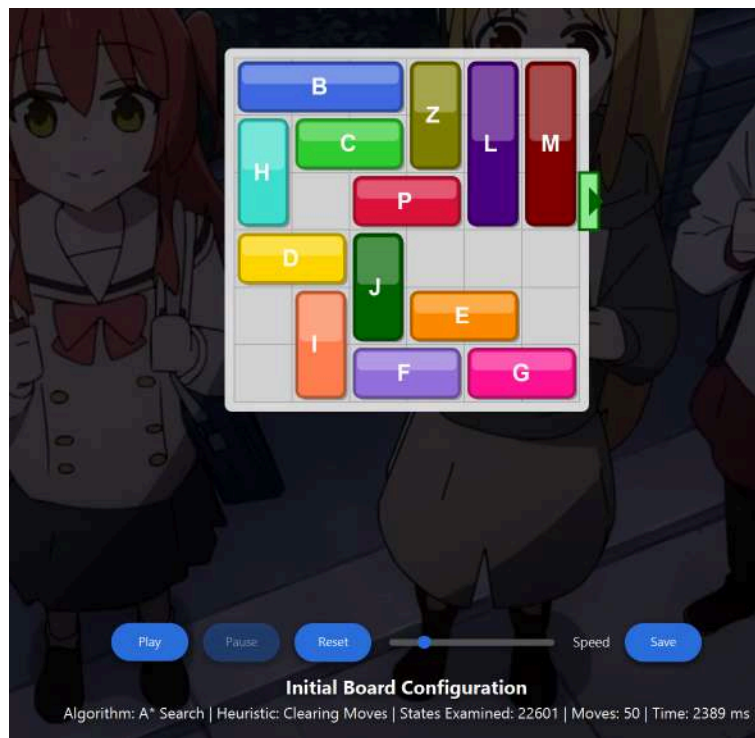
Output file : block_misteri4.txt



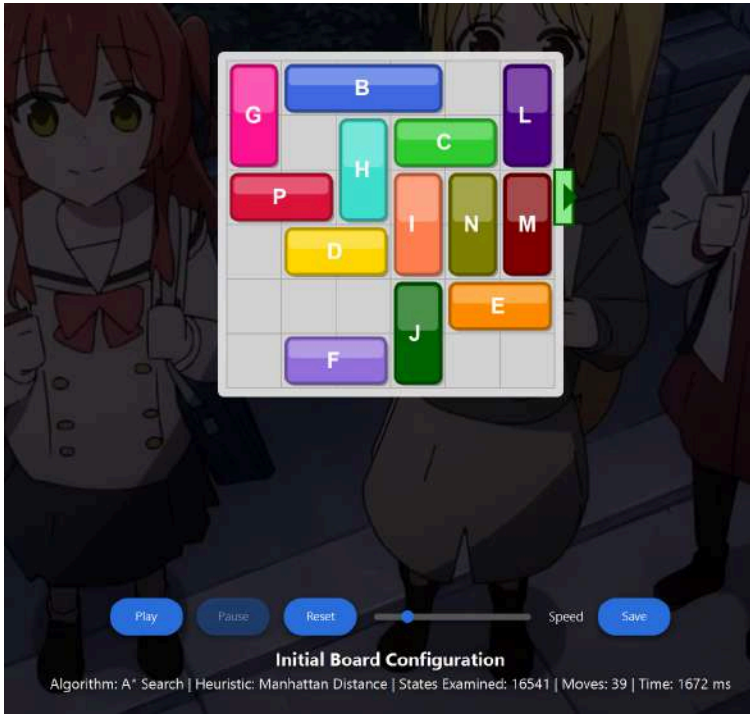
Algoritma : A*
 Heuristik : Clearing Moves
 Test Case:
 6 6
 12
 BBBZLM
 HCCZLM
 H.PPLMK
 DDJ...
 .IJEE.
 .IFFGG

States : 22601
 Moves: 50
 Time : 2389 ms

Output file : clear_misteri4.txt



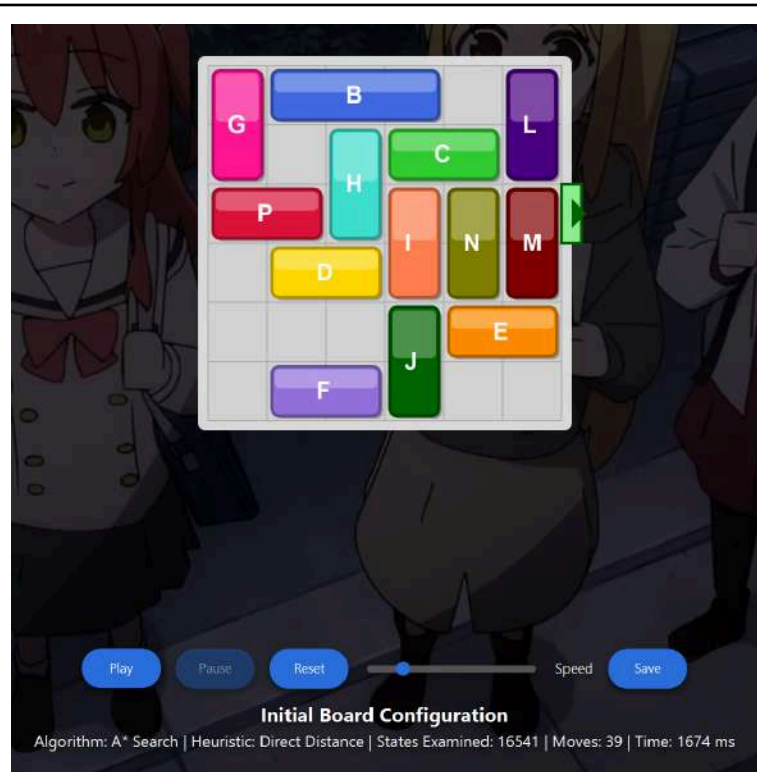
4.5.3 Testcase tebaktebakanyuk_adahasilnyaapaenggakyyaa.txt

Heuristik	Hasil
<p>Algoritma : A*</p> <p>Heuristik : Manhattan Distance</p> <p>Test Case:</p> <p>6 6</p> <p>12</p> <p>GBBB.L</p> <p>G.HCCL</p> <p>PPHINMK</p> <p>.DDINM</p> <p>...JEE</p> <p>.FFJ..</p> <p>States : 16541</p> <p>Moves: 39</p> <p>Time : 1672 ms</p> <p>Output file : manhattan_tebak.txt</p>	

Algoritma : A*
 Heuristik : Direct Distance
 Test Case:
 6 6
 12
 GBBB.L
 G.HCCL
 PPHINMK
 .DDINM
 ...JEE
 .FFJ..

States : 16541
 Moves: 39
 Time : 1674 ms

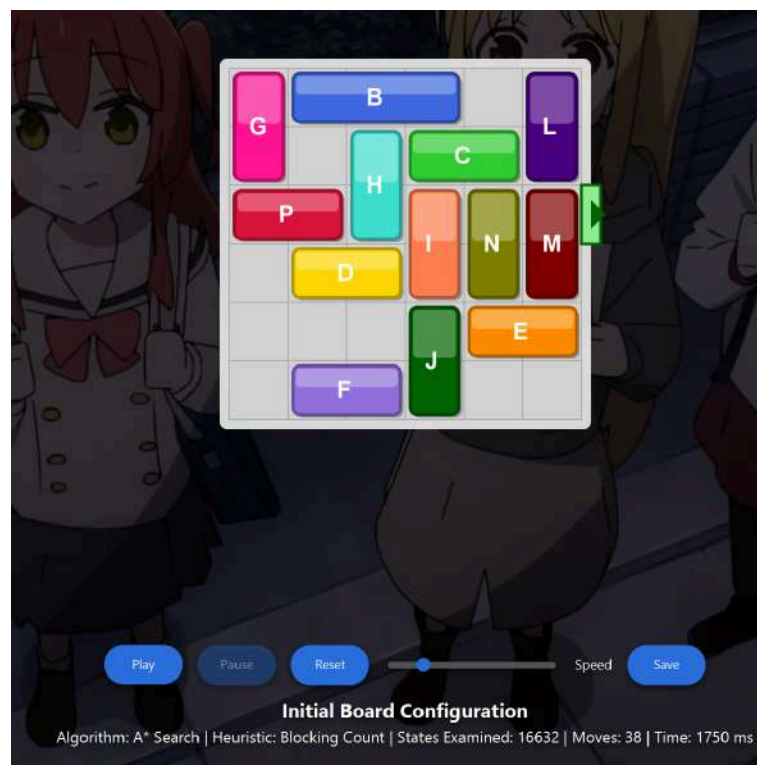
Output file : direct_tebak.txt



Algoritma : A*
 Heuristik : Blocking Count
 Test Case:
 6 6
 12
 GBBB.L
 G.HCCL
 PPHINMK
 .DDINM
 ...JEE
 .FFJ..

States : 16632
 Moves: 38
 Time : 1750 ms

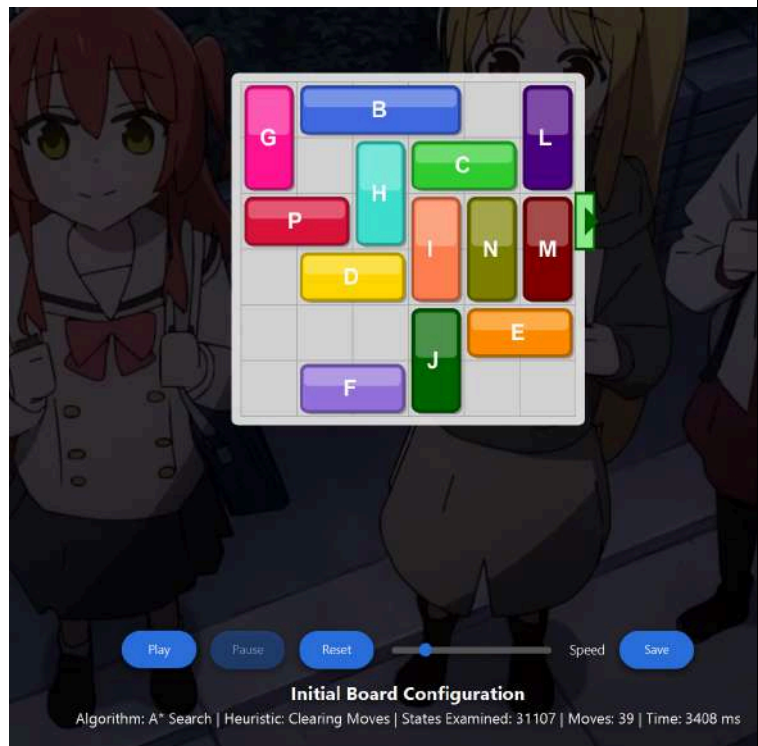
Output file : block_tebak.txt



Algoritma : A*
 Heuristik : Clearing Moves
 Test Case:
 6 6
 12
 GBBB.L
 G.HCCL
 PPHINMK
 .DDINM
 ...JEE
 .FFJ..

States : 31107
 Moves: 39
 Time : 3408 ms

Output file : clear_tebak.txt



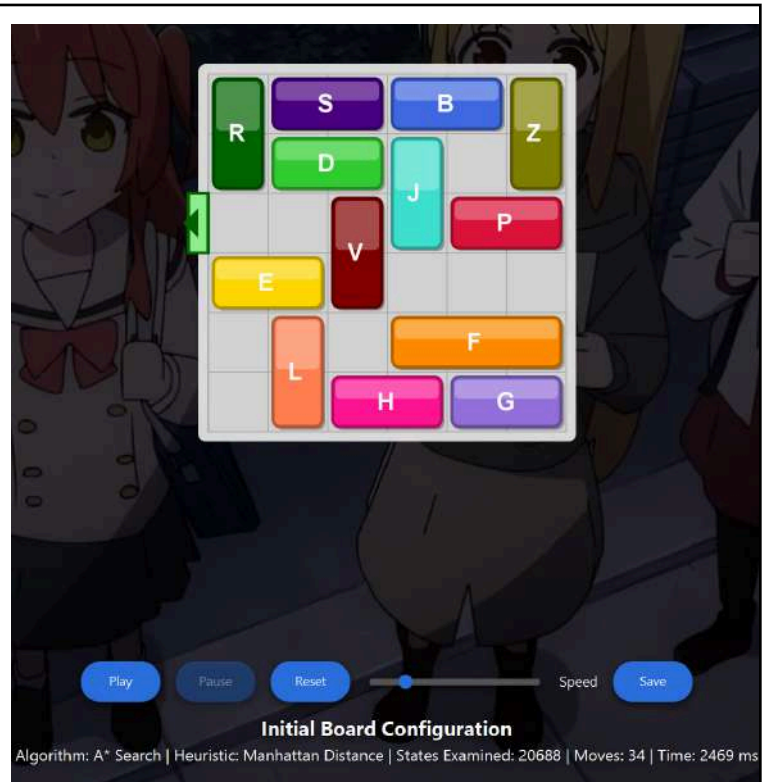
4.5.4 Testcase wahgaknormalini😂.txt

Heuristik	Hasil
-----------	-------

Algoritma : A*
 Heuristik : Manhattan Distance
 Test Case:
 6 6
 12
 RSSBBZ
 RDDJ.Z
 K..VJPP
 EEV...
 .L.FFF
 .LHHGG

States : 20688
 Moves: 34
 Time : 2469 ms

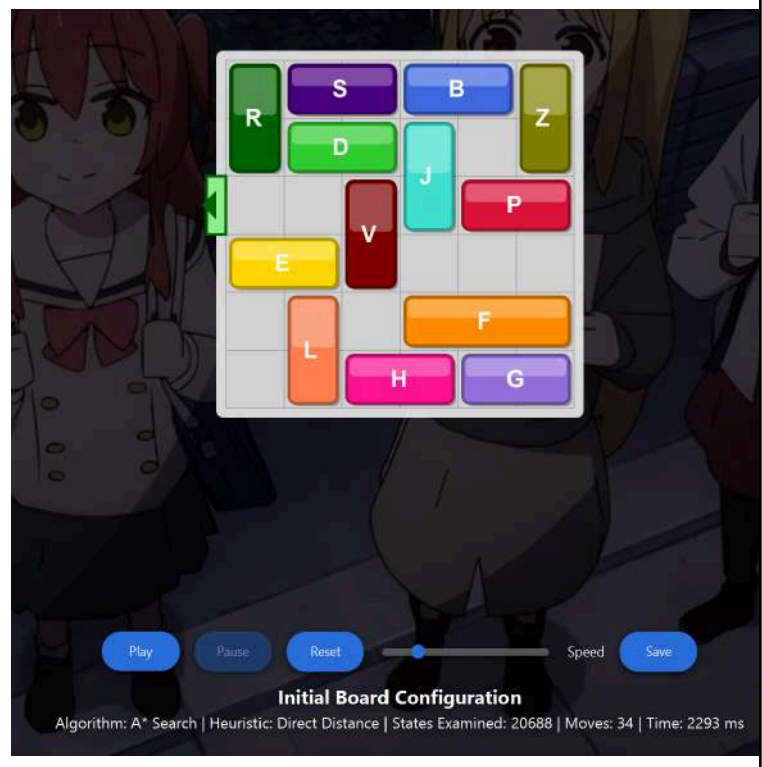
Output file :
 manhattan_normalkok.txt



Algoritma : A*
 Heuristik : Direct Distance
 Test Case:
 6 6
 12
 RSSBBZ
 RDDJ.Z
 K..VJPP
 EEV...
 .L.FFF
 .LHHGG

States : 20688
 Moves: 34
 Time : 2293 ms

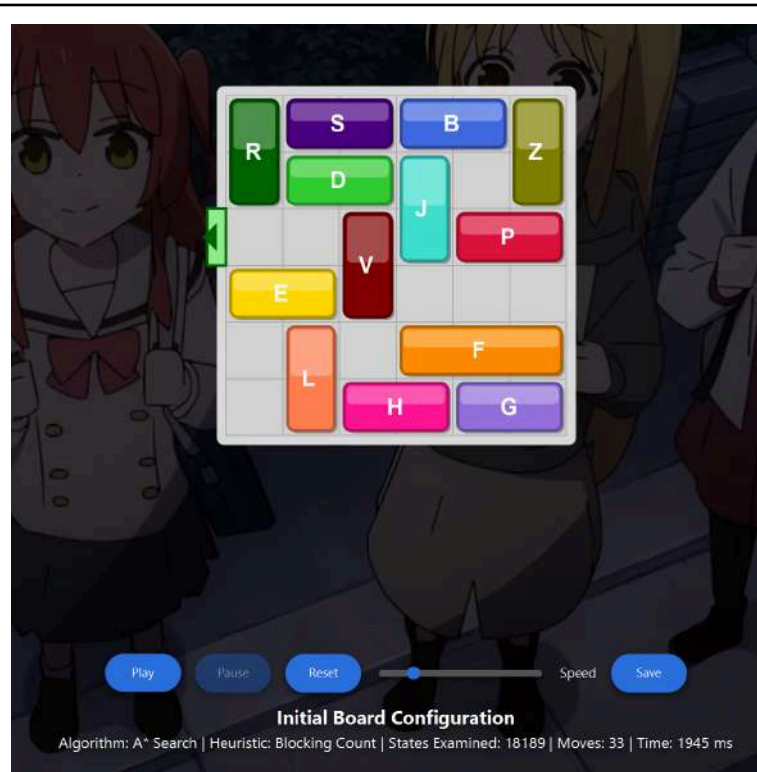
Output file : direct_normalkok.txt



Algoritma : A*
 Heuristik : Blocking Count
 Test Case:
 6 6
 12
 RSSBBZ
 RDDJ.Z
 K..VJPP
 EEV...
 .L.FFF
 .LHHGG

States : 18189
 Moves: 33
 Time : 1945 ms

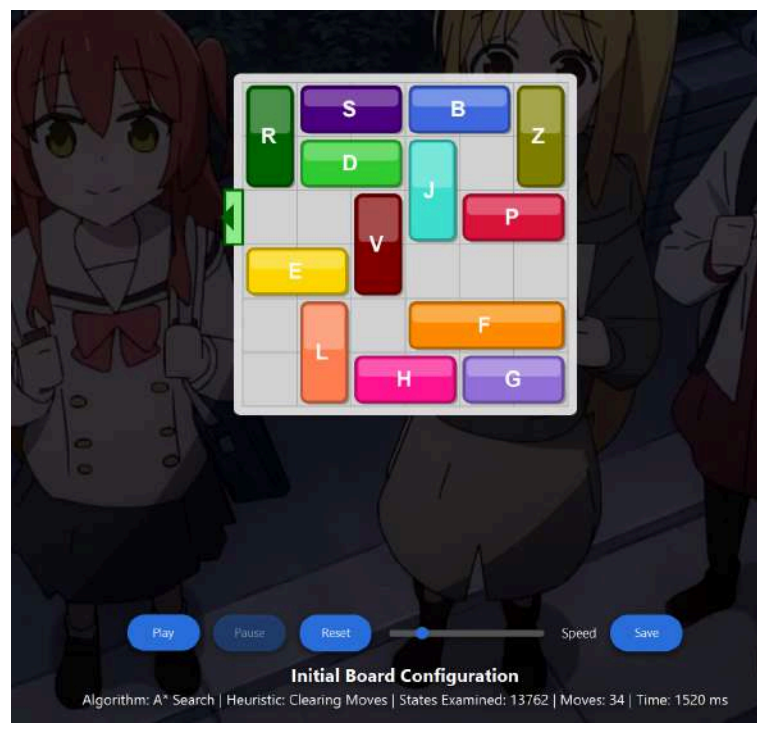
Output file : block_normalkok.txt



Algoritma : A*
 Heuristik : Clearing Moves
 Test Case:
 6 6
 12
 RSSBBZ
 RDDJ.Z
 K..VJPP
 EEV...
 .L.FFF
 .LHHGG

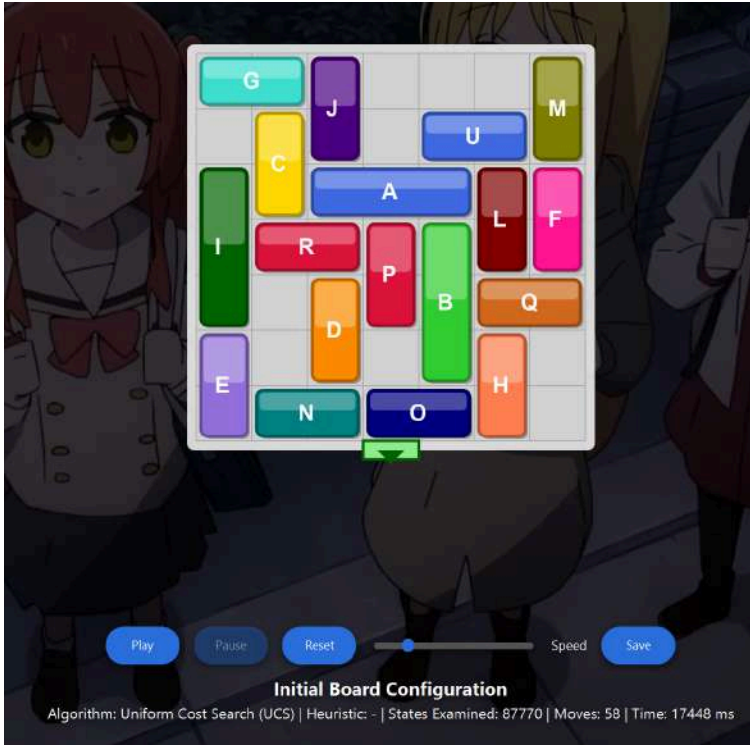
States : 13762
 Moves: 34
 Time : 1520 ms

Output file : clear_normalkok.txt



4.6 Kasus Spesial

4.6.1 Testcase w1ntr.txt (7x7)

Algoritma	Hasil
<p>Algoritma : Uniform Cost Search Heuristik : - Test Case: 7 7 17 GGJ...M .CJ.UUM ICAAALF IRRPBLF I.DPBQQ E.D.BH. ENNOOH. K</p> <p>States : 87770 Moves: 58 Time : 17448 ms</p> <p>Output file : w1ntr_ucs.txt</p>	

Algoritma : Greedy Best First Search

Heuristik : Blocking Count

Test Case:

7 7

17

GGJ...M

.CJ.UUM

ICAAALF

IRRPBLF

I.DPBQQ

E.D.BH.

ENNOOH.

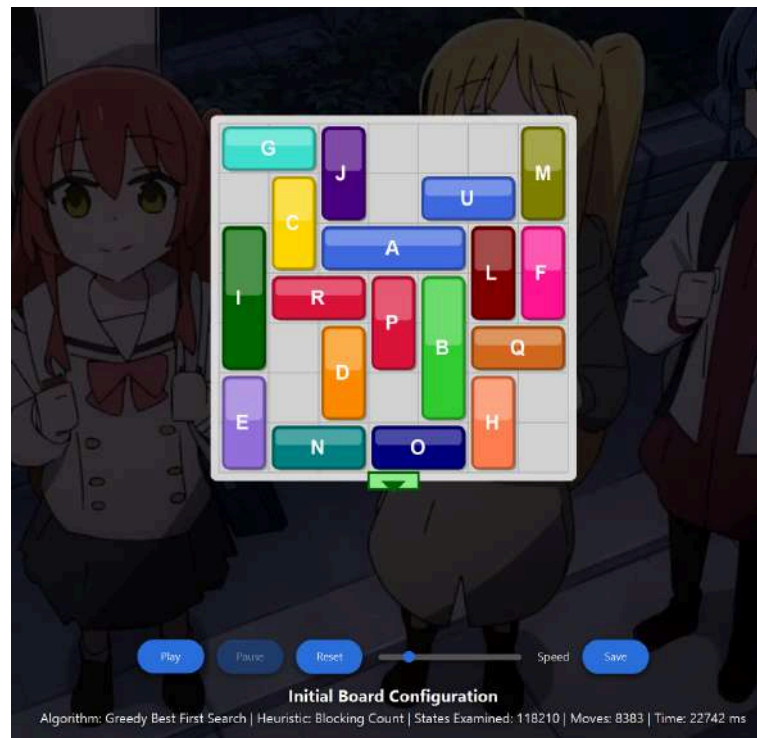
K

States : 118210

Moves: 8383

Time : 22742 ms

Output file : w1ntr_GBFS.txt



Algoritma : A*

Heuristik : Blocking Count

Test Case:

7 7

17

GGJ...M

.CJ.UUM

ICAAALF

IRRPBLF

I.DPBQQ

E.D.BH.

ENNOOH.

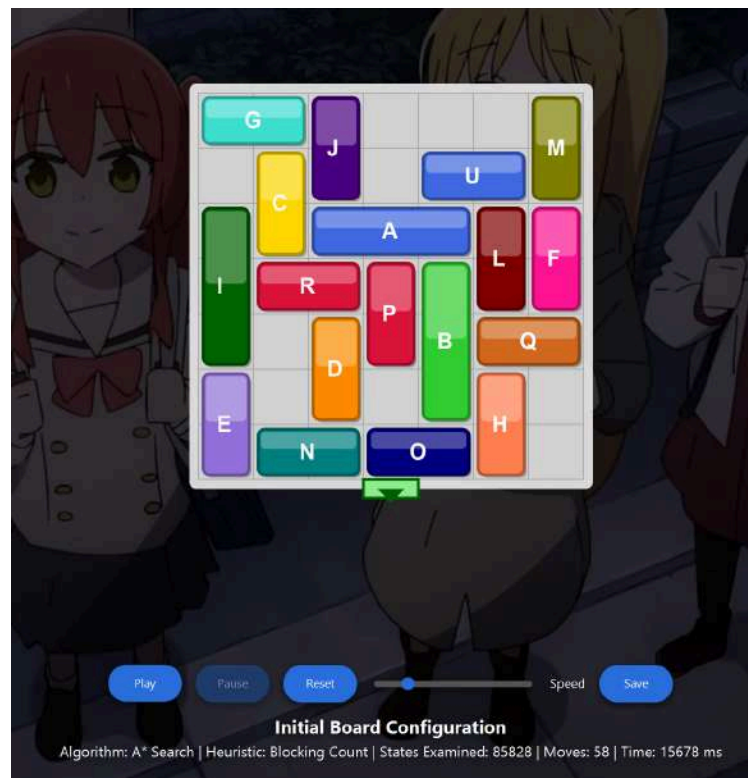
K

States : 85828

Moves: 58

Time : 15678 ms

Output file : w1ntr_astar.txt



Algoritma : Dijkstra

Heuristik : -

Test Case:

7 7

17

GGJ...M

.CJ.UUM

ICAAALF

IRRPBLF

I.DPBQQ

E.D.BH.

ENNOOH.

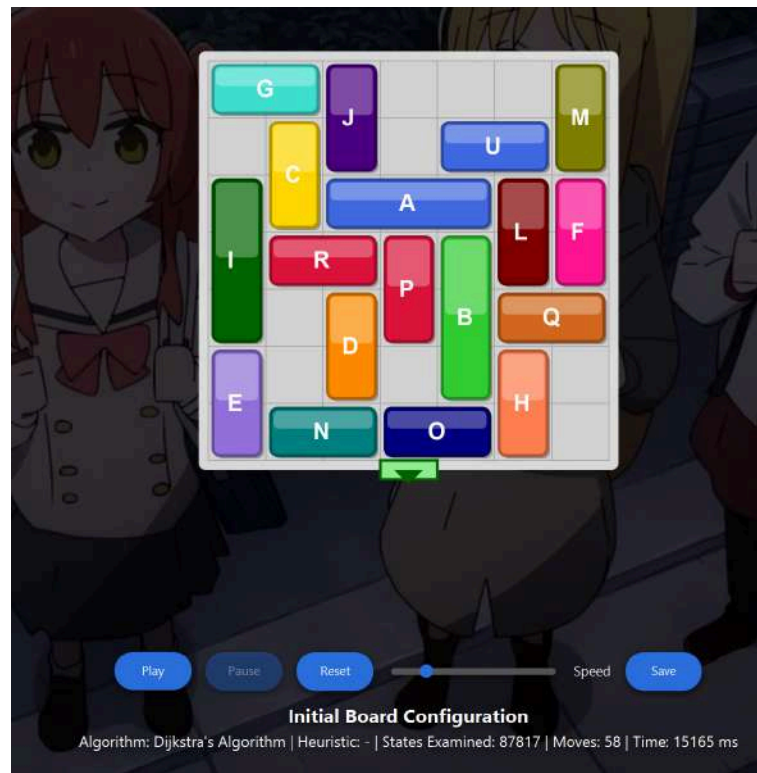
K

States : 87817

Moves: 58

Time : 15165 ms

Output file : w1ntr_Dijkstra.txt



Algoritma : Beam Search

Heuristik : Blocking Count

Test Case:

7 7

17

GGJ...M

.CJ.UUM

ICAAALF

IRRPBLF

I.DPBQQ

E.D.BH.

ENNOOH.

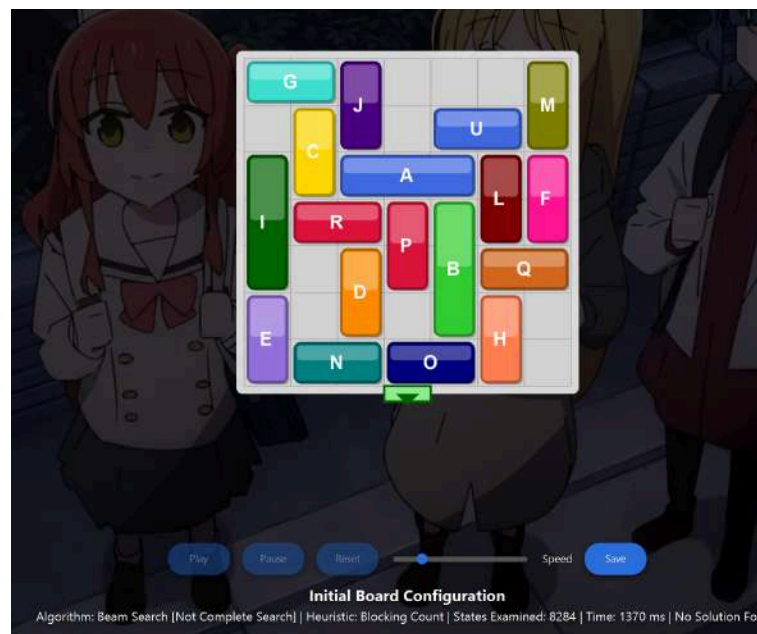
K

States : 8284

Moves: No solution

Time : 1370 ms

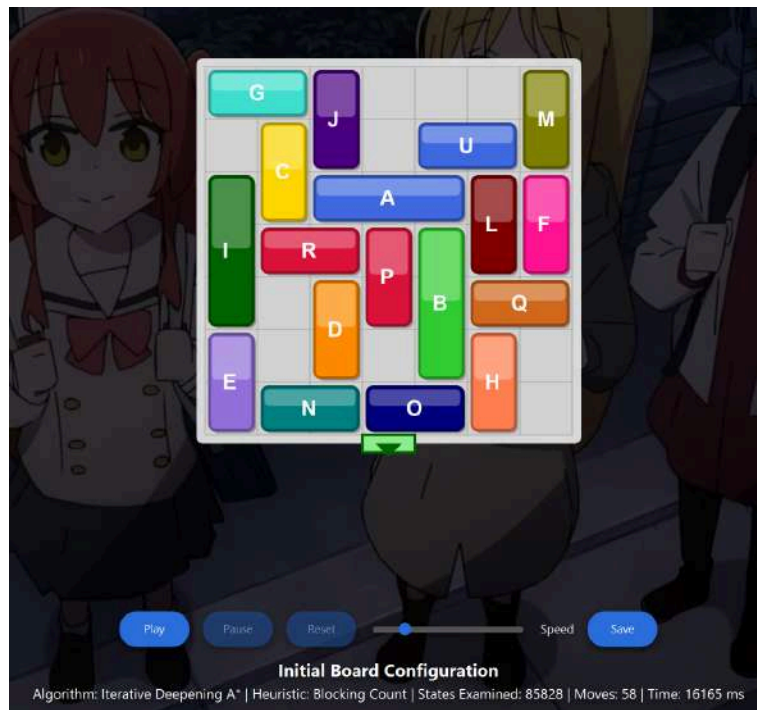
Output file : w1ntr_beam.txt



Algoritma : IDA*
 Heuristik : Blocking Count
 Test Case:
 7 7
 17
 GGJ...M
 .CJ.UUM
 ICAAALF
 IRRPBLF
 I.DPBQQ
 E.D.BH.
 ENNOOH.
 K

States : 85828
 Moves: 58
 Time : 16165 ms

Output file : w1ntr_IDA.txt



4.6.2 Testcase katousignature.txt (8x8)

Algoritma	Hasil
-----------	-------

Algoritma : Uniform Cost Search

Heuristik : -

Test Case:

8 8

23

BBQMSSSD

A.QM.OED

ACCC.OEV

GG...OHV

WX.PPPHUK

WX..FNNU

IXTTFLLU

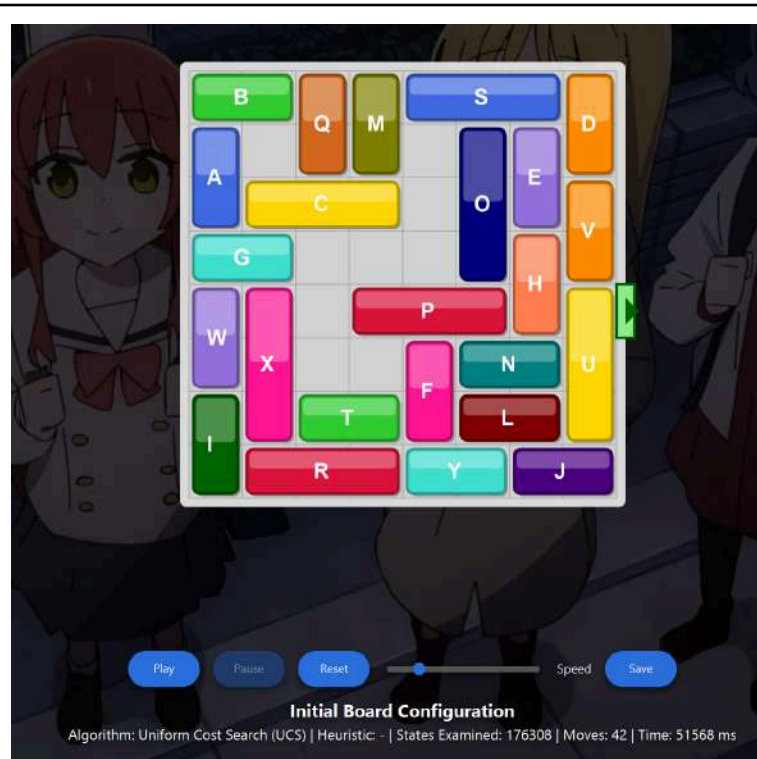
IRRRYYJJ

States : 176308

Moves: 42

Time : 51568 ms

Output file : katou_ucs.txt



Algoritma : Greedy Best First Search

Heuristik : Blocking Count

Test Case:

8 8

23

BBQMSSSD

A.QM.OED

ACCC.OEV

GG...OHV

WX.PPPHUK

WX..FNNU

IXTTFLLU

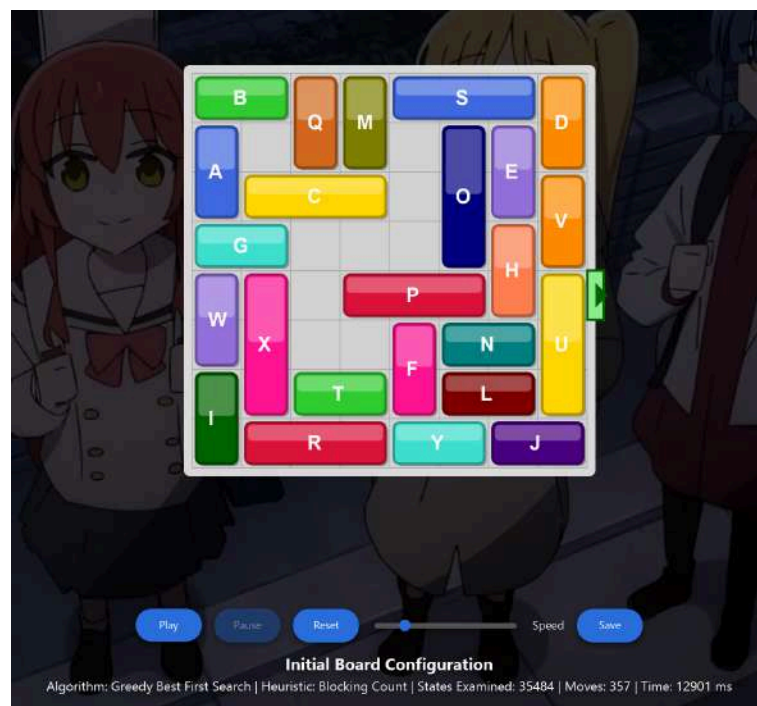
IRRRYYJJ

States : 35484

Moves: 357

Time : 12901 ms

Output file : katou_GBFS.txt



Algoritma : A*

Heuristik : Blocking Count

Test Case:

8 8

23

BBQMSSSD

A.QM.OED

ACCC.OEV

GG...OHV

WX.PPPHUK

WX..FNNU

IXTTFLLU

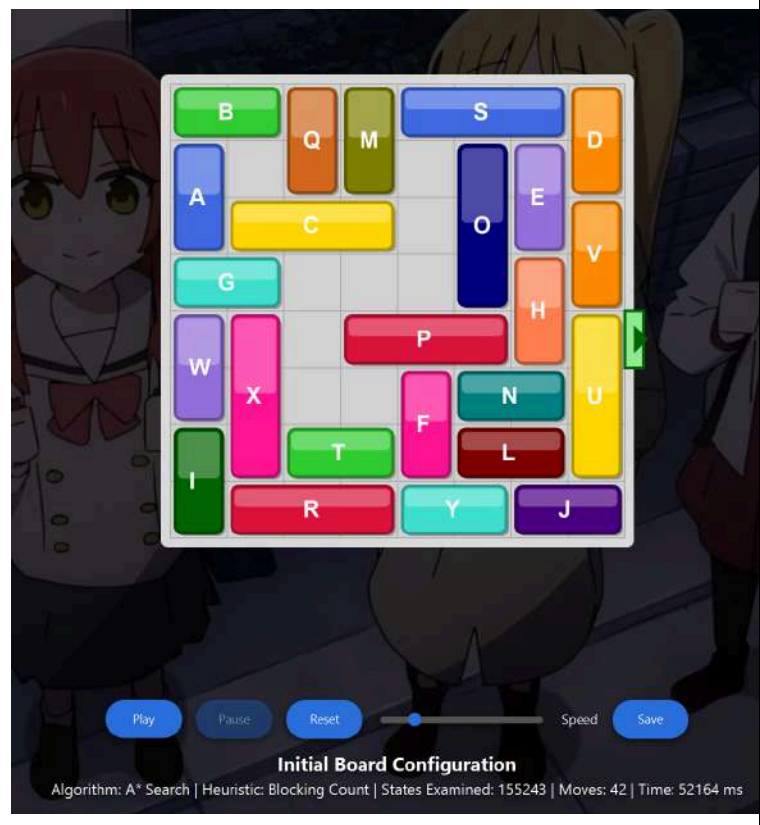
IRRRYYJJ

States : 155243

Moves: 42

Time : 52164 ms

Output file : katou_astar.txt



Algoritma : Dijkstra

Heuristik : -

Test Case:

8 8

23

BBQMSSSD

A.QM.OED

ACCC.OEV

GG...OHV

WX.PPPHUK

WX..FNNU

IXTTFLLU

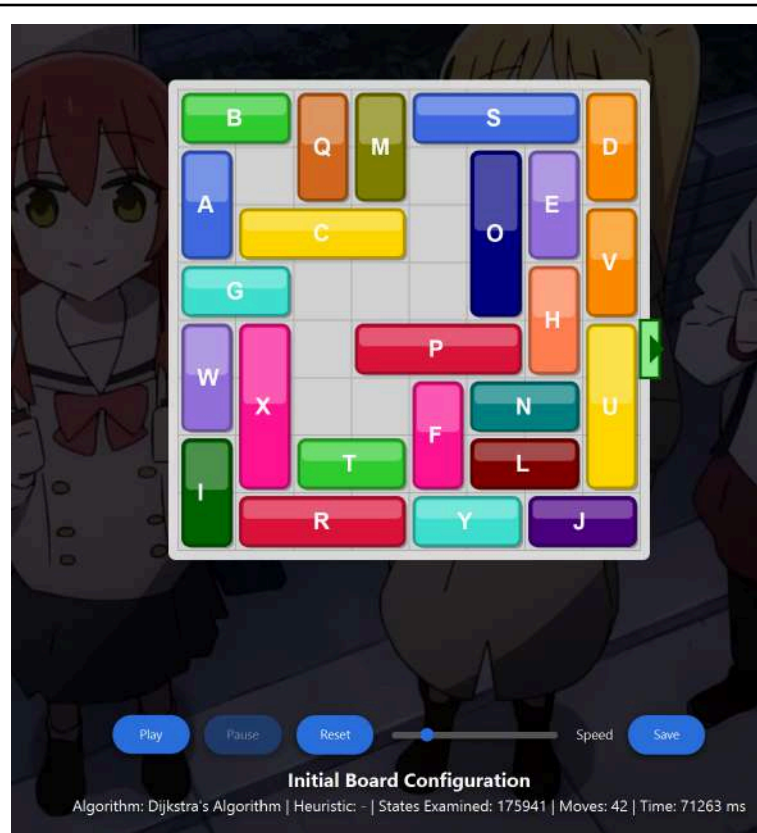
IRRRYYJJ

States : 175941

Moves: 42

Time : 71263 ms

Output file : katou_Dijkstra.txt



Algoritma : Beam Search

Heuristik : Blocking Count

Test Case:

8 8

23

BBQMSSSD

A.QM.OED

ACCC.OEV

GG...OHV

WX.PPPHUK

WX..FNNU

IXTTFLLU

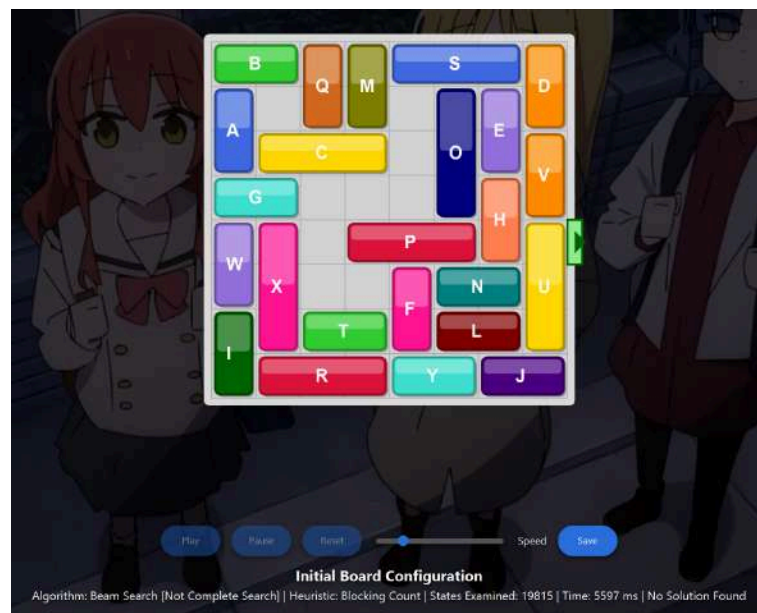
IRRRYYJJ

States : 19815

Moves: No solution

Time : 5597 ms

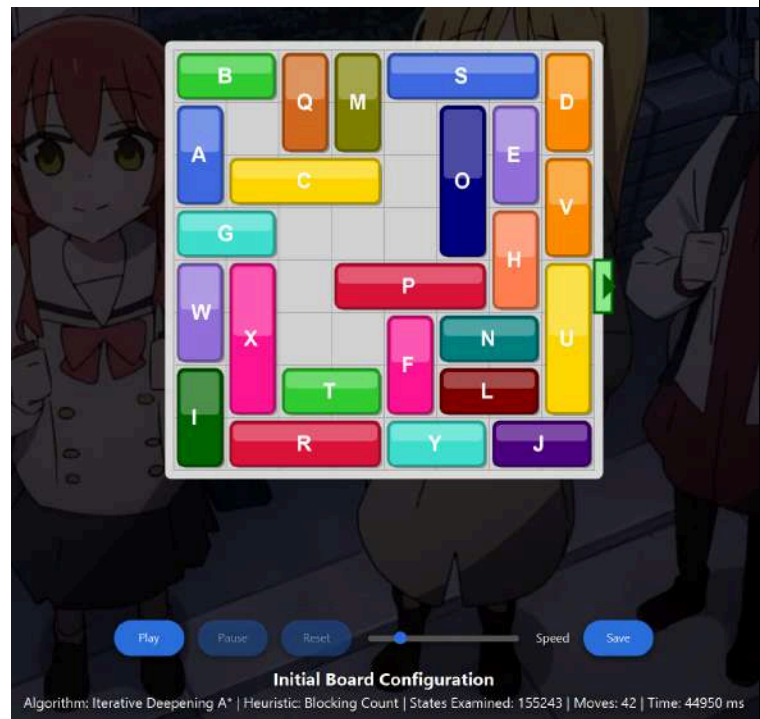
Output file : katou_beam.txt



Algoritma : IDA*
 Heuristik : Blocking Count
 Test Case:
 8 8
 23
 BBQMSSSD
 A.QM.OED
 ACCC.OEV
 GG...OHV
 WX.PPPHUK
 WX..FNNU
 IXTTFLLU
 IRRYYJJ

States : 155243
 Moves: 42
 Time : 44950 ms

Output file : katou_IDA.txt



BAB V: HASIL ANALISIS PERCOBAAN

Berdasarkan beberapa hasil percobaan, terlihat perbedaan baik dari segi *state* yang dikunjungi, optimalitas, dan juga waktu eksekusi dari setiap algoritma yang ada. Dari *test case* default.txt, didapatkan bahwa jalur yang dimiliki algoritma UCS, A*, Dijkstra, Beam Search dan IDA* tergolong sama (hampir sama). Algoritma GBFS memang tidak menghasilkan solusi yang optimum, karena algoritma GBFS pada suatu waktu akan terjebak di optimum lokal, tetapi waktu pencariannya cukup cepat dibandingkan dengan algoritma yang lainnya.

Pada pengujian *test case* misteri4.txt, terlihat bahwa algoritma UCS, A*, IDA*, dan Dijkstra memiliki optimalitas yang konsisten, hal ini mulai berbeda pada algoritma Beam Search dan GBFS. Pada algoritma Beam Search dan GBFS, solusi yang dihasilkan tidaklah optimum, karena pada dasarnya algoritma GBFS dan Beam Search adalah algoritma yang mirip yang dijalankan berdasarkan hanya pada heuristik ($h(n)$) yang diterapkan pada algoritma tersebut. Tetapi, dapat dikatakan bahwa kedua algoritma tersebut memiliki waktu eksekusi yang cukup cepat dibandingkan dengan keempat algoritma yang lainnya. Karena *test case* masih belum terlalu kompleks, maka perbedaan di antara keenam algoritma tersebut belum terlalu terlihat.

Perbedaan mulai terlihat ketika *test case* sudah mulai lebih kompleks. Sebagai contoh, pada *test case* tebaktebakanyuk_adahasilnyaapaenggakya.txt dan wahgaknormalini😄.txt, waktu eksekusi dari algoritma UCS dan Dijkstra mulai menjadi cukup lambat dibandingkan dengan algoritma yang lain, tetapi kedua algoritma tersebut memiliki optimalitas solusi yang sangat konsisten dan baik. Hal ini dikarenakan algoritma UCS dan Dijkstra memiliki *behaviour* yang mirip dengan BFS dalam konteks permasalahan permainan Rush Hour, sehingga solusi yang dihasilkan pasti optimal (karena BFS memiliki sifat *complete* dan *optimal*), hanya saja kompleksitas waktu yang digunakan lebih lambat dibandingkan dengan algoritma lain (sesuai dengan pembahasan kompleksitas secara teori).

Keunggulan dari algoritma GBFS dan Beam Search pun mulai terlihat pada *test case* tebaktebakanyuk_adahasilnyaapaenggakya.txt dan wahgaknormalini😄.txt, yakni waktu pencarian solusi yang sangat cepat, bahkan mencapai 2.83 kali lebih cepat dari algoritma A* dan IDA*, serta 3.67 kali lebih cepat dari algoritma UCS dan Dijkstra. Tetapi, seperti yang telah dibahas sebelumnya, kedua algoritma tersebut tidak memiliki optimalitas solusi yang baik dibandingkan dengan algoritma UCS, Dijkstra, A*, dan IDA*.

Selanjutnya, terdapat kasus unik, khususnya pada solusi dari algoritma A* maupun IDA*, yakni pada persoalan yang lebih kompleks. Kasus tersebut ada pada heuristik manhattan distance, direct distance, dan clearing count terkadang memiliki jumlah solusi $n+1$ (dengan n adalah solusi paling optimal yang bisa didapatkan pada suatu permasalahan). Contohnya pada *test case* wahgaknormalini😄.txt, dengan menggunakan heuristik manhattan distance, direct distance, dan clearing count, langkah yang ditemukan adalah sebanyak 34 langkah, sedangkan langkah paling optimumnya adalah langkah yang dihasilkan dari heuristik blocking count, yakni

33 langkah. Hal ini dapat disebabkan karena pada heuristik blocking count, heuristik menggunakan pendekatan jumlah *piece* yang melakukan blokir pada *exit*, sedangkan yang lainnya lebih berfokus pada jarak (ada hubungan antara *expected compound moves* dan penghalang yang ada), menyebabkan adanya $n+1$ solusi. Tetapi, karena perbedaannya insignifikan, dan dapat dipastikan hanya $n+1$, maka heuristik ini masih *admissible* dan masih bisa mencari solusi yang sangat mendekati optimum, bahkan optimum.

Selain itu, ada kasus unik selain dari heuristik, khususnya pada *test case* w1ntr.txt. Pada analisis sebelumnya, telah dijelaskan bahwa GBFS memiliki waktu eksekusi yang lebih singkat dibandingkan dengan algoritma yang lain. Tetapi, pada *test case* tersebut, dapat dilihat bahwa GBFS memiliki waktu eksekusi yang paling lama dibandingkan dengan kelima algoritma yang lain. Hal ini menunjukkan bahwa GBFS mungkin saja untuk terjebak terlalu lama dalam nilai optimum lokalnya, dan tidak kunjung menemukan nilai optimum globalnya, menyebabkan adanya langkah yang tidak perlu yang seharusnya tidak dilakukan oleh algoritma tersebut sehingga akan menambah waktu eksekusi dari algoritma tersebut.

Selanjutnya, pada *test case* katousignature.txt, algoritma GBFS dapat menemukan solusi lima kali lebih cepat dibandingkan dengan algoritma UCS, Dijkstra, A*, dan IDA*. *Test case* ini menunjukkan algoritma GBFS yang tidak terjebak dalam optimum lokal terlalu lama, sehingga *state* yang dicek juga tidak banyak. Selain itu, *behaviour* algoritma lainnya masih sama, hanya saja algoritma Dijkstra mengalami peningkatan waktu eksekusi yang signifikan karena *scope* permasalahan yang mulai kompleks.

Terlepas dari permasalahan tersebut, algoritma A* dan IDA* dapat selalu memberikan hasil yang optimal, dengan memori dan waktu eksekusi yang lebih sedikit dibandingkan dengan algoritma UCS dan Dijkstra. Di sisi lain, GBFS dan Beam Search tidak selalu memberikan hasil yang optimal, tetapi kedua algoritma tersebut dapat digunakan jika memang permasalahan tersebut sangat kompleks tetapi ingin mencari solusi dengan sangat cepat (tetap tidak disarankan untuk menggunakannya pada penyelesaian permainan Rush Hour). Untuk algoritma UCS dan Dijkstra, tentu bisa digunakan, namun harus dicatat jika permasalahannya cukup besar dan kompleks, maka waktu eksekusi dan memori harus mulai menjadi perhatian, tetapi solusinya terjamin ada.

Sehingga, berdasarkan percobaan dari *test case* yang ada beserta hasil analisis yang telah dibuat, dapat disimpulkan bahwa algoritma A* dan IDA* adalah algoritma yang paling optimal untuk memecahkan permasalahan pada permainan Rush Hour dengan solusi, jumlah *node* yang dikunjungi, dan penggunaan memori yang lebih *wise* dibandingkan dengan keempat algoritma yang lainnya.

BAB VI: BONUS

6.1 Algoritma Pencarian Jalur Alternatif

Bonus ini adalah bonus yang bertujuan untuk membuat variasi dari algoritma pencarian jalur selain algoritma UCS, GBFS, dan A* pada penyelesaian permainan Rush Hour. Algoritma lain yang penulis kembangkan antara lain adalah algoritma Dijkstra, algoritma *Beam Search*, dan algoritma IDA*, yang masing-masing sudah dijelaskan analisisnya pada bagian [2.5](#), [2.6](#), dan [2.7](#).

6.2 Heuristik Alternatif

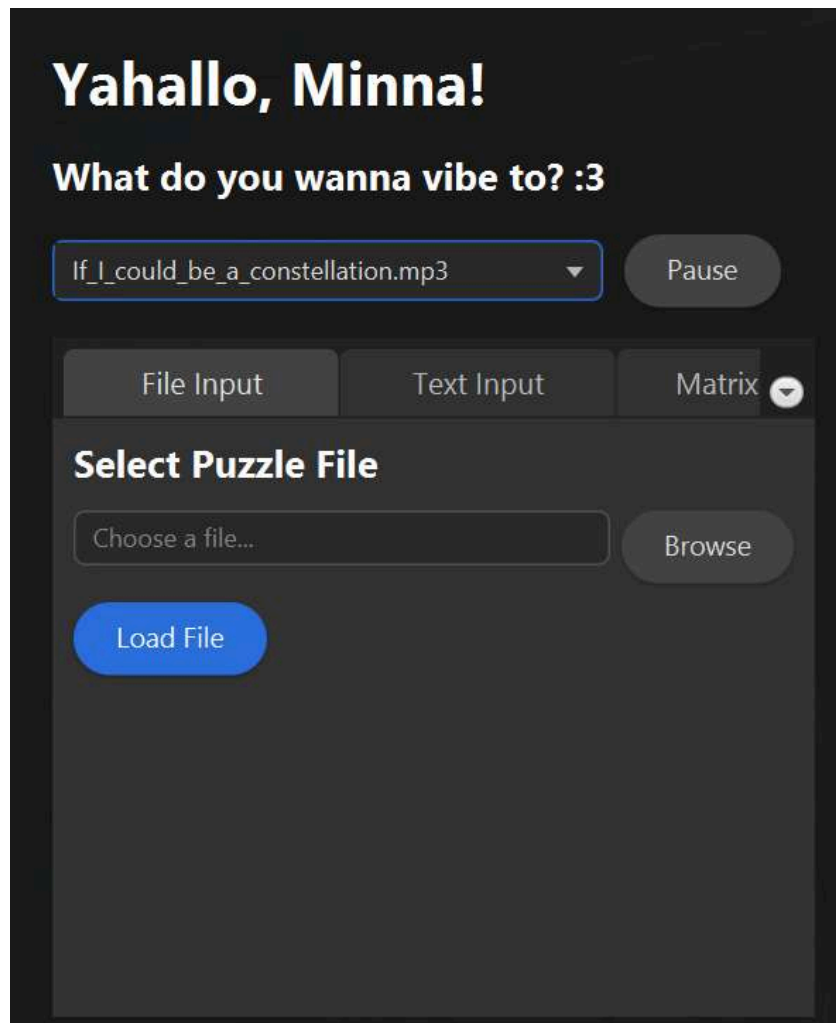
Bonus ini adalah bonus yang bertujuan untuk membantu beberapa algoritma pencarian jalur yang membutuhkan pendekatan heuristik pada implementasinya (sebagai contoh, algoritma A*). Pada implementasinya, penulis membuat empat pendekatan heuristik, yakni Manhattan Distance, Direct Distance, Blocking Count, dan Clearing Count. Masing-masing pendekatan heuristik telah dijelaskan pada bagian [2.8](#).

6.3 Graphical User Interface (GUI)

Pada program ini, kami mengimplementasikan GUI dengan menggunakan bantuan sebuah library, yakni [javaFX](#) yang diimplementasikan untuk bahasa Java. Berikut adalah penjelasan dari GUI yang telah penulis implementasikan.

6.3.1 Input

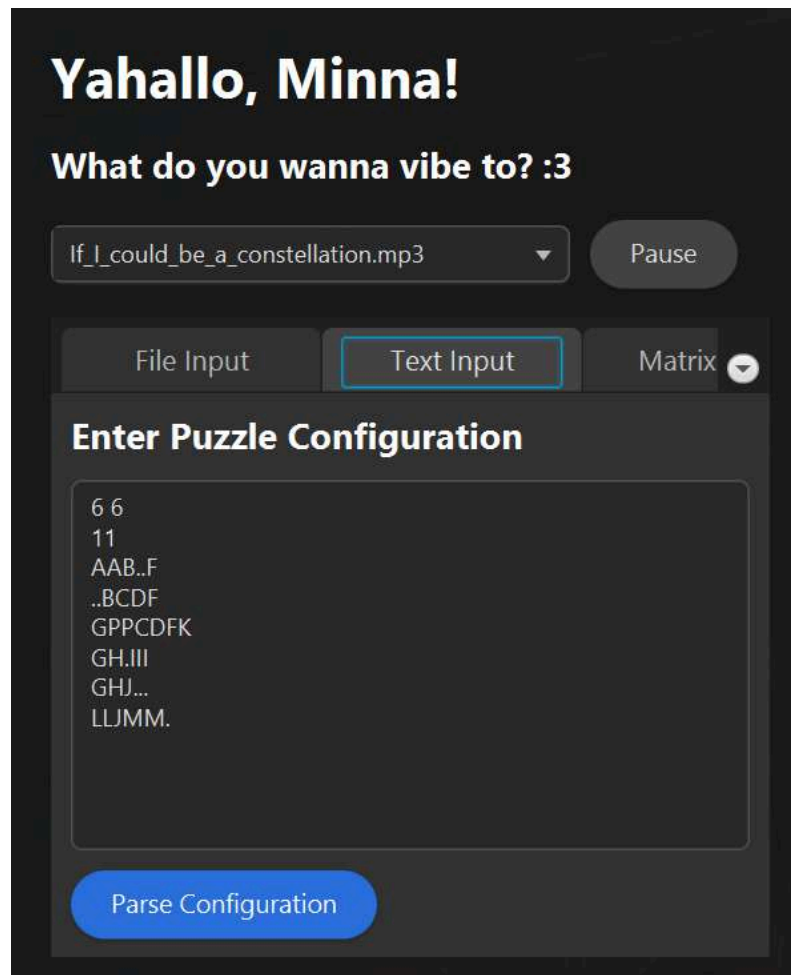
Penulis menyediakan tiga cara input, yakni dengan cara *load file*, *parse text*, dan *matrix input*. Untuk pilihan dari input, terletak pada pojok kiri atas, baik untuk sistem operasi Windows maupun Linux. Berikut adalah tangkapan layar dari opsi *input* yang ada pada program.



Gambar 20. File input

(Sumber: arsip penulis)

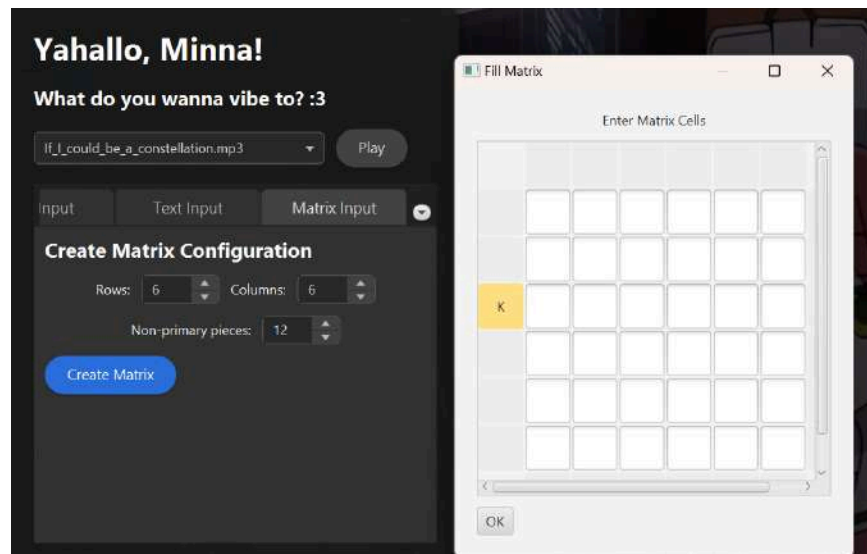
Untuk input dengan cara *load file*, cukup dengan menekan tombol *browse* untuk mencari *file* apa yang ingin di-*input* dan dicari solusinya. Setelah itu, tekan tombol *load file* untuk melakukan *load* pada file.



Gambar 21. Text input

(Sumber: arsip penulis)

Untuk input dengan cara *parse text*, cukup dengan mengetik konfigurasi dari permainan dengan aturan-aturan yang sudah ditentukan untuk di-*input* dan dicari solusinya. Setelah itu, tekan tombol *Parse Configuration* untuk melakukan *load* pada konfigurasi.



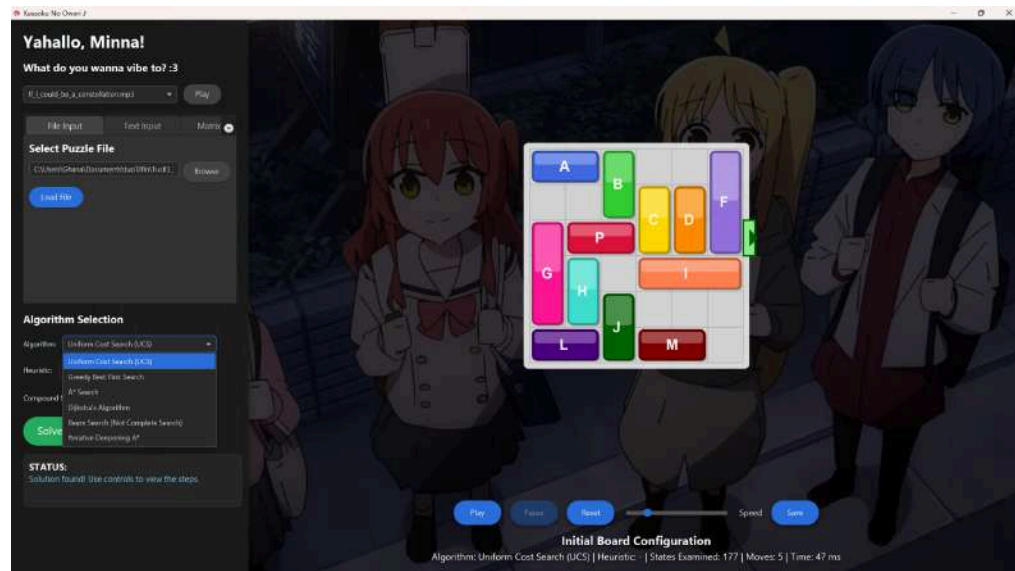
Gambar 22. Matrix input

(Sumber: arsip penulis)

Untuk input dengan cara *matrix input*, cukup dengan melakukan input ukuran dari papan (definisikan *row* dan *column*-nya), setelah itu dilanjutkan dengan jumlah *piece* yang bukan *primary piece*, dilanjutkan dengan menekan tombol *Create Matrix* untuk membuat *matrix input*-nya. Setelah menekan tombol, akan muncul sebuah *pop-up* dari konfigurasi matriks, kotak yang berwarna putih adalah representasi dari papan, dan kotak abu-abu merupakan letak dari *exit* (huruf K). Lakukan input *piece* pada kotak putih, dan tekan pada kotak abu-abu mana saja untuk meletakkan *exit*.

6.3.2 Algorithm Selection dan Solving

Penulis menyediakan enam algoritma pencarian jalur dan 4 pendekatan heuristik yang bisa dipilih oleh pengguna. Untuk memilih algoritma, pengguna diharuskan untuk menekan tombol *drop-down* untuk memilih algoritma apa yang ingin digunakan, begitu juga dengan heuristik. Setelah memilih algoritma dan heuristik apa yang ingin digunakan, pengguna akan bisa menekan tombol *Solve Puzzle* untuk memulai pencarian solusi. Berikut adalah skema dari pemilihan algoritma dan juga tahap penyelesaian.

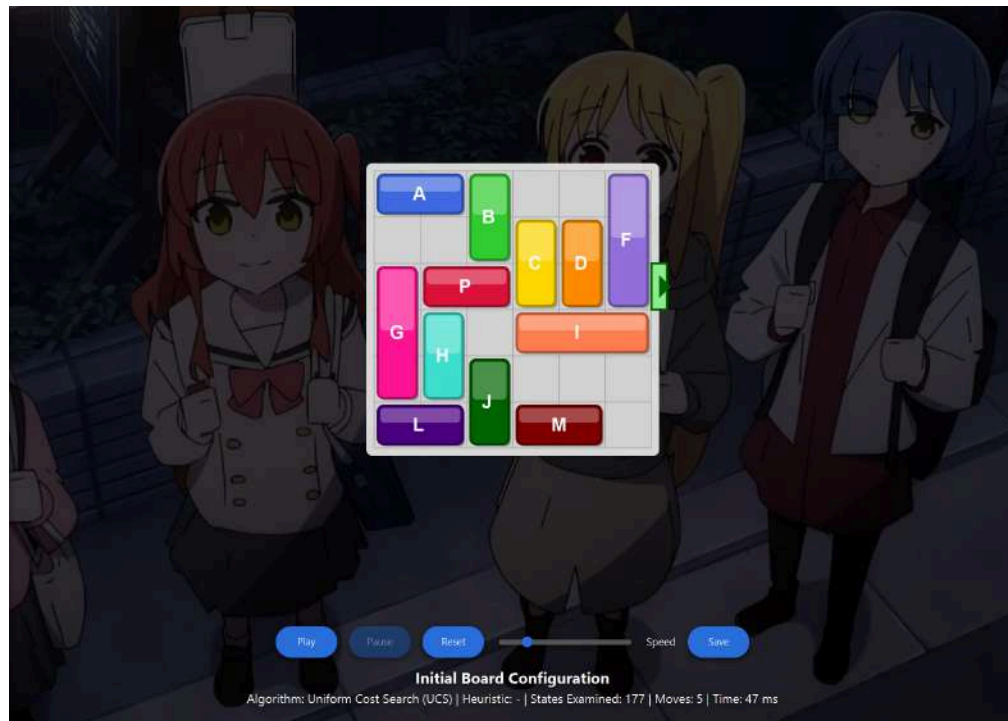


Gambar 23. Algorithm selection dan solving

(Sumber: arsip penulis)

6.3.3 Canvas

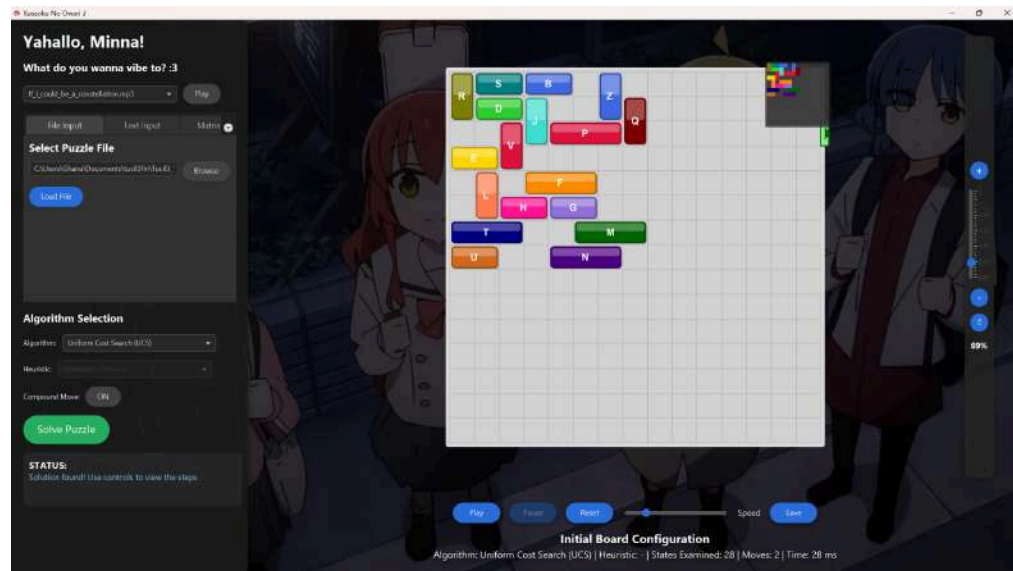
Canvas adalah bagian visualisasi dari penyelesaian permainan Rush Hour, di mana terdapat 5 fitur utama yang disediakan oleh penulis. Yang pertama adalah fitur *Play*, yakni fitur yang akan menjalankan visualisasi dari penyelesaian permainan Rush Hour secara bertahap tergantung apakah solusinya ada atau tidak. Yang kedua adalah fitur *Pause*, yakni fitur yang akan melakukan jeda pada visualisasi dari penyelesaian permainan Rush Hour. Yang ketiga adalah fitur *Reset*, yakni fitur yang akan melakukan reset pada *board* ke dalam *initial state* (kondisi awal sebelum visualisasi dilakukan). Yang keempat adalah fitur *Speed*, yakni *slider* yang berfungsi untuk mengatur animasi dari visualisasi yang dilakukan oleh program. Yang kelima adalah fitur *Save*, yakni fitur untuk menyimpan solusi yang ditemukan atau tidak ditemukan oleh aplikasi ke dalam file berekstensi txt (pengguna juga bisa secara bebas memilih *path* untuk menyimpan *file*). Berikut adalah implementasi dari *canvas* yang telah dibuat.



Gambar 24. Tampilan canvas

(Sumber: arsip penulis)

Di bawah fitur yang diimplementasikan, terdapat juga informasi dari solusi yang telah didapatkan. Informasi tersebut antara lain adalah algoritma yang digunakan, pendekatan heuristik yang dipilih, banyak *state* yang dicek, dan waktu program dalam menemukan suatu solusi. Selain itu, jika papan dirasa cukup besar, program secara otomatis akan menyediakan *slider zoom* untuk memperbesar atau mengecilkan papan. Contoh implementasi akan dilampirkan pada gambar di bawah.

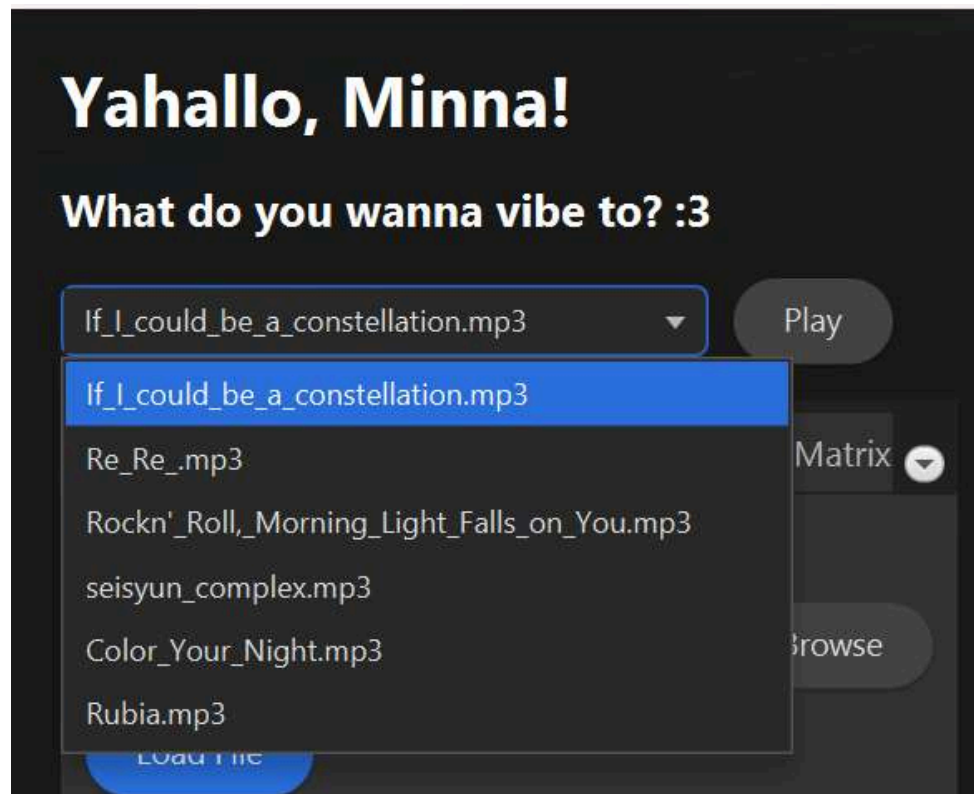


Gambar 25. Implementasi zoom

(Sumber: arsip penulis)

6.3.4 Musik

Agar membuat pengalaman pengguna lebih asyik dan menyenangkan. Penulis juga menambahkan beberapa lagu yang bisa disetel pada program. Secara umum, musik yang ada pada program adalah musik yang diambil dari Anime [Bocchi The Rock](#) (*please watch it, it's peak*). Tetapi, musik hanya tersedia pada sistem operasi Windows, sehingga untuk sistem operasi Linux, fitur ini tidak tersedia. Berikut adalah implementasi dari musik yang ada pada program.



Gambar 26. Daftar musik yang ada pada program

(Sumber: arsip penulis)

LAMPIRAN

Github Repository

Program dapat diakses pada https://github.com/Fariz36/Tucil3_13523069_13523090

Miscellaneous (Tabel Poin)

Tabel 2 Poin yang dikerjakan

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

DAFTAR PUSTAKA

M. Rinaldi. Route Planning (Bagian 1). Diakses pada 18 Mei 2025 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)

M. Rinaldi. Route Planning (Bagian 2). Diakses pada 18 Mei 2025 dari [https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-\(2025\)-Bagian2.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf)

F. Michael. Solving Rush Hour, the Puzzle. Diakses pada 18 Mei 2025 dari <https://www.michaelfogleman.com/rush/>

AKHIR KATA

Sekian, *deliverables* dari kami, semoga AC, dan semangat kak asisten :3!...



Gw ketika ngerjain 10 tubes dengan deadline di minggu yang sama.
Ada banyak sekali cara untuk menumbuhkan habit buruk, vibecoding is one of them

<https://www.youtube.com/watch?v=iqVhUX4Vel8>

-> Fariz.

Who else didn't sleep last night



“What is fourth semester even about 🧠 SOS SOS SOS”

[Lullaby](#)

-> Nayaka.