LAPORAN TUGAS KECIL 3
IF2211 STRATEGI ALGORITMA

Penyelesaian Puzzle Rush Hour
Menggunakan Algoritma *Pathfinding*

Disusun oleh :

Carlo Angkisan	13523091
Heleni Gratia M Tampubolon	13523107

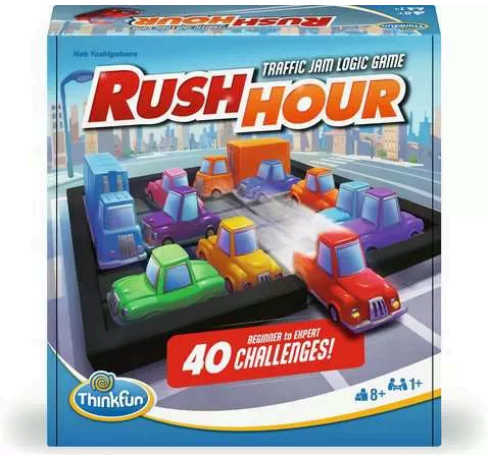
PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
JL. GANESHA 10, BANDUNG 40132

2025

DAFTAR ISI

BAB I
DESKRIPSI MASALAH DAN ALGORITMA

1.1 Deskripsi Masalah Rush Hour



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 menguji kemampuan pemecahan masalah pemain dengan cara menggeser kendaraan dalam sebuah papan permainan berukuran tetap, dengan tujuan utama mengeluarkan satu mobil utama (*primary piece*) dari kemacetan lalu lintas melalui pintu keluar yang terletak di tepi papan. Setiap kendaraan (*piece*) memiliki ukuran dan orientasi tertentu (horizontal atau vertikal), serta hanya dapat bergerak lurus sesuai orientasinya, tanpa dapat berputar atau menembus kendaraan lain. Tantangan utama permainan ini terletak pada bagaimana mengatur pergeseran kendaraan-kendaraan lain untuk membentuk jalur bebas bagi mobil utama agar dapat keluar papan dengan jumlah langkah seminimal mungkin. *Primary piece* adalah satu-satunya kendaraan yang diizinkan untuk keluar papan melalui pintu keluar, yang posisinya selalu sejajar dengan orientasi *primary piece* dan berada tepat di dinding papan. Permainan dinyatakan selesai jika *primary piece* berhasil digeser keluar papan melalui pintu keluar tersebut.

1.2 Algoritma UCS (*Uniform Cost Search*)

Uniform Cost Search (UCS) merupakan algoritma *pathfinding* (pencarian) solusi optimal yang bertujuan menemukan jalur dengan biaya total (*cumulative cost*) terendah yang terhitung dari simpul awal (start node) ke simpul tujuan (goal node) dengan mempertimbangkan biaya (*cost*) kumulatif dari langkah-langkah yang telah diambil. Hal ini dapat dikatakan juga, bahwa algoritma ini didasarkan pada fungsi evaluasi f(n) untuk setiap simpul, dengan f(n) = g(n), dimana g(n) adalah biaya kumulatifnya. Sehingga ciri

dari UCS ialah melakukan perluasan simpul dengan biaya jalur terendah terlebih dahulu. Akibat hal ini juga, UCS menggunakan struktur priority queue yang dapat mengurutkan simpul berdasarkan total biaya dari simpul awal ke simpul tersebut. Simpul yang memiliki biaya total terendah akan diekspansi terlebih dahulu.

1. Inisialisasi
   UCS diawali dari simpul awal(*root node*). Simpul ini kemudian ditambahkan ke *priority queue* dengan biaya kumulatif adalah nol karena belum ada langkah yang diambil.
2. Ekspansi Simpul
   Simpul dengan biaya jalur terendah diambil dari *priority queue*. Simpul ini kemudian diekspansi dan semua tetangganya (simpul yang terhubung langsung) dieksplorasi.
3. Eksplorasi Tetangga
   Untuk setiap tetangga dari simpul yang diekspansi, algoritma menghitung biaya total dari simpul awal ke tetangga tersebut melalui simpul saat ini.
   - Jika simpul tetangga belum ada di dalam priority *queue*, maka simpul tersebut ditambahkan ke antrian (*queue*) dengan biaya yang telah dihitung.
   - Jika simpul tetangga sudah ada di dalam antrian, tetapi ditemukan jalur baru dengan biaya yang lebih rendah, maka biaya simpul tersebut diperbarui di dalam antrian.
4. Pemeriksaan Tujuan
   Setelah suatu simpul diekspansi, algoritma akan memeriksa apakah simpul tersebut merupakan simpul tujuan. Jika iya, maka algoritma akan mengembalikan total biaya untuk mencapai simpul tersebut beserta jalur yang ditempuh.
5. Pengulangan
   Proses ini akan diulang terus-menerus, hingga:
   - *Priority Queue* kosong, atau
   - Simpul suatu tujuan telah ditemukan.

Algoritma tersebut dapat dituliskan dalam Pseudocode sebagai berikut.

```
procedure UCS(input start: integer, input goal: integer)
{
  Menemukan jalur dengan biaya minimum dari simpul start ke goal
  Masukan: start dan goal adalah simpul
  Keluaran: semua simpul yang dikunjungi ditulis ke layar dan jalur
minimum ditemukan
}

Deklarasi
  antrian_prioritas : struktur data (biaya, simpul)
  biaya[0..n] : array of integer
  asal[0..n] : array of integer
  current, neighbor, cost, total_cost : integer
```

```
Algoritma:
  inisialisasi semua biaya[i] ← tak_hingga
  biaya[start] ← 0
  asal[start] ← -1
  masukkan (0, start) ke dalam antrian_prioritas

  while antrian_prioritas tidak kosong do
    keluarkan (current_cost, current) dengan biaya terkecil dari
antrian_prioritas

    write(current)

    if current = goal then
      keluarkan jalur dari asal[] mulai dari goal ke start
      stop
    endif

    for setiap tetangga neighbor dari current do
      cost ← biaya dari current ke neighbor
      total_cost ← current_cost + cost

      if total_cost < biaya[neighbor] then
        biaya[neighbor] ← total_cost
        asal[neighbor] ← current
        masukkan (total_cost, neighbor) ke dalam antrian_prioritas
      endif
    endfor
  endwhile
```

1.3 Algoritma *Greedy Best First Search*

Greedy Best First Search (GBFS) adalah algoritma pencarian jalur yang menggunakan pendekatan heuristik untuk memilih simpul yang paling menjanjikan (menurut fungsi heuristik/h(n)) untuk diperluas terlebih dahulu. Hal ini dapat dikatakan juga, bahwa algoritma ini didasarkan pada fungsi evaluasi f(n) untuk setiap simpul, dengan f(n) = h(n). Tujuan GBFS adalah mencapai simpul tujuan dengan meminimalkan estimasi jarak ke tujuan, bukan total biaya seperti UCS. Algoritma ini memilih simpul dengan nilai heuristik terkecil terlebih dahulu, sehingga digunakan struktur *priority queue* untuk mengatur urutan ekspansi simpul. Karena pemilihan jalur didasarkan sepenuhnya pada nilai heuristik, algoritma GBFS tidak menjamin solusi yang dihasilkan merupakan solusi optima**l**.

Terdapat beberapa tahapan proses dalam algoritma GBFS:

1. Inisialisasi
   GBFS diawali dari simpul awal(*root node*). Simpul ini kemudian ditambahkan ke *priority queue* dengan nilai heuristic sebagai prioritas.
2. Ekspansi Simpul

Simpul dengan nilai heuristic terkecil diambil dari *priority queue* (paling atas). Simpul ini kemudian diekspansi dan semua tetangganya (simpul yang terhubung langsung) dieksplorasi.

3. Eksplorasi Tetangga
   Untuk setiap tetangga dari simpul yang diekspansi, algoritma menghitung nilai heuristic dari simpul awal ke tetangga tersebut melalui simpul saat ini.
   - Jika simpul tetangga belum pernah dikunjungi, antrian ditambahkan dengan prioritas berdasarkan nilai heuristik.
4. Pemeriksaan Tujuan
   Setelah suatu simpul diekspansi, algoritma akan memeriksa apakah simpul tersebut merupakan simpul tujuan. Jika iya, maka algoritma akan mengembalikan total biaya untuk mencapai simpul tersebut beserta jalur yang ditempuh.
5. Pengulangan
   Proses ini akan diulang terus-menerus, hingga:
   - *Priority Queue* kosong, atau
   - Simpul suatu tujuan telah ditemukan.

Algoritma tersebut dapat dituliskan dalam Pseudocode sebagai berikut.

```
procedure GBFS(input start: integer, input goal: integer)
{
  Menemukan jalur dari simpul start ke goal berdasarkan nilai heuristik
terkecil
  Masukan: start dan goal adalah simpul
  Keluaran: semua simpul yang dikunjungi ditulis ke layar dan jalur
ditemukan
}

Deklarasi
  antrian_prioritas : struktur data (heuristik, simpul)
  dikunjungi[0..n] : array of boolean
  asal[0..n] : array of integer
  current, neighbor : integer

Algoritma:
  inisialisasi semua dikunjungi[i] ← false
  asal[start] ← -1
  masukkan (h[start], start) ke dalam antrian_prioritas

  while antrian_prioritas tidak kosong do
    keluarkan (h_now, current) dengan h terkecil dari antrian_prioritas

    if dikunjungi[current] = true then
      lanjutkan ke iterasi berikutnya
    endif

    dikunjungi[current] ← true
    write(current)
```

```
    if current = goal then
      keluarkan jalur dari asal[] mulai dari goal ke start
      stop
    endif

    for setiap tetangga neighbor dari current do
      if not dikunjungi[neighbor] then
        asal[neighbor] ← current
        masukkan (h[neighbor], neighbor) ke dalam antrian_prioritas
      endif
    endfor
  endwhile
```

1.4 Algoritma A*

A* adalah algoritma pencarian jalur yang menggabungkan dua pendekatan: Uniform Cost Search (UCS) dan Greedy Best First Search (GBFS). Tujuannya adalah untuk menemukan jalur optimal ke tujuan dengan meminimalkan total estimasi biaya dari simpul awal ke simpul tujuan.

Algoritma ini menggunakan fungsi evaluasi f(n) untuk menentukan simpul mana yang akan diperluas terlebih dahulu, dengan rumus:

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

Dengan:

- g(n) adalah biaya dari simpul awal ke simpul n.
- h(n) adalah estimasi biaya dari n ke tujuan dengan menggunakan fungsi heuristic.
- f(n) adalah estimasi total biaya terpendek melalui n.

A* akan selalu memilih simpul dengan nilai f(n) terkecil terlebih dahulu. Karena memperhitungkan total biaya aktual (g) dan estimasi ke depan (h), A* dapat menghasilkan solusi optimal asalkan fungsi heuristiknya *admissible* (tidak melebihi biaya sebenarnya ke tujuan). Beberapa fungsi heuristik yang umum digunakan, yaitu Manhattan, Euclidean, dan Diagonal Distance.

Terdapat beberapa tahapan proses dalam algoritma A*:

1. Inisialisasi
   Algoritma A* dimulai dari simpul awal yang dimasukkan ke dalam *priority queue*. Nilai g untuk simpul awal diatur ke 0, dan h dihitung menggunakan fungsi heuristik. Simpul awal menjadi simpul pertama yang diproses.
2. Ekspansi Simpul
   Simpul dengan nilai f terkecil diambil dari *priority queue* (paling atas). Simpul ini kemudian diekspansi dan semua tetangganya (simpul yang terhubung langsung) dieksplorasi.
3. Eksplorasi Tetangga

Untuk setiap tetangga dari simpul yang diekspansi, algoritma menghitung nilai heuristic dan biaya (*cost*) nyata dari simpul awal ke tetangga tersebut melalui simpul saat ini.

- Jika simpul tetangga belum pernah dikunjungi, antrian ditambahkan dengan prioritas berdasarkan nilai heuristik.

4. Pemeriksaan Tujuan
   Setelah suatu simpul diekspansi, algoritma akan memeriksa apakah simpul tersebut merupakan simpul tujuan. Jika iya, maka algoritma akan mengembalikan total biaya untuk mencapai simpul tersebut beserta jalur yang ditempuh.
5. Pengulangan
   Proses ini akan diulang terus-menerus, hingga:
   - *Priority Queue* kosong, atau
   - Simpul suatu tujuan telah ditemukan.

Algoritma tersebut dapat dituliskan dalam Pseudocode sebagai berikut.

```
procedure AStar(input start: integer, input goal: integer)
{
  Menemukan jalur dari simpul start ke goal berdasarkan nilai f = g + h
  Masukan: start dan goal adalah simpul
  Keluaran: semua simpul yang dikunjungi ditulis ke layar dan jalur
ditemukan
}

Deklarasi
  antrian_prioritas : struktur data (f[n], simpul)
  dikunjungi[0..n] : array of boolean
  asal[0..n] : array of integer
  g[0..n] : array of integer
  current, neighbor : integer

Algoritma:
  inisialisasi semua dikunjungi[i] ← false
  inisialisasi semua g[i] ← ∞
  asal[start] ← -1
  g[start] ← 0
  masukkan (g[start] + h[start], start) ke dalam antrian_prioritas

  while antrian_prioritas tidak kosong do
    keluarkan (f_now, current) dengan f terkecil dari antrian_prioritas

    if dikunjungi[current] = true then
      lanjutkan ke iterasi berikutnya
    endif

    dikunjungi[current] ← true
    write(current)

    if current = goal then
      keluarkan jalur dari asal[] mulai dari goal ke start
      stop
```

```
    endif

    for setiap tetangga neighbor dari current do
      biaya ← cost(current, neighbor)
      if not dikunjungi[neighbor] and g[current] + biaya < g[neighbor]
then
        g[neighbor] ← g[current] + biaya
        asal[neighbor] ← current
        masukkan (g[neighbor] + h[neighbor], neighbor) ke dalam
antrian_prioritas
      endif
    endfor
  endwhile
```

A* sangat efektif dalam pencarian jalur karena mempertimbangkan biaya total aktual dan estimasi biaya ke depan. Algoritma ini lebih unggul dibandingkan GBFS karena:

- Tidak hanya memikirkan seberapa dekat dengan tujuan (h), tetapi juga mempertimbangkan seberapa mahal jalur yang sudah ditempuh (g)
- Menjamin solusi optimal jika fungsi heuristik yang digunakan adalah *admissible* dan *consisten.*

1.5 Algoritma *Beam Search*

Beam Search adalah algoritma pencarian berbasis heuristik yang mirip dengan Breadth-First Search tetapi tidak menyimpan semua simpul. Beam Search memperluas hanya k simpul terbaik (berdasarkan heuristik) di setiap level pencarian, dengan parameter *beam width* sebesar k. Ini mengurangi kebutuhan memori dan waktu, tetapi bisa mengabaikan solusi optimal jika tidak berada di antara k node terbaik.

Terdapat beberapa tahapan proses dalam algoritma *Beam Search*:

1. Inisialisasi
   Beam Search dimulai dari simpul awal yang dimasukkan ke dalam *list frontier*. Simpul awal dihitung nilai heuristiknya menggunakan fungsi h(n).
   beam width (k) ditentukan di awal, yang menjadi batas maksimum jumlah simpul yang disimpan di setiap iterasi.
   Simpul awal menjadi satu-satunya elemen dalam frontier pada awal pencarian.
2. Ekspansi Simpul
   Simpul-simpul dalam frontier (maksimal sebanyak k) diekspansi satu per satu. Untuk setiap simpul:
   - Diperiksa apakah simpul tersebut merupakan simpul tujuan.
   - Jika bukan, maka semua tetangga (anak) dari simpul ini akan dihasilkan.
3. Eksplorasi Tetangga
   Untuk setiap tetangga dari simpul yang diekspansi, algoritma menghitung nilai heuristic dari simpul awal ke tetangga tersebut melalui simpul saat ini.

- Jika simpul tetangga belum pernah dikunjungi, antrian ditambahkan dengan prioritas berdasarkan nilai heuristik untuk iterasi berikutnya.

4. Seleksi Simpul Terbaik
   Setelah semua simpul tetangga dari seluruh *frontier* saat ini dihasilkan, maka dicek:
   - Semua simpul dalam daftar kandidat disortir berdasarkan nilai heuristiknya (semakin kecil akan semakin baik).
   - Hanya k simpul terbaik yang dipilih untuk menjadi frontier pada iterasi berikutnya.
   - Simpul-simpul yang tidak masuk dalam k teratas dibuang (tidak dilanjutkan).
5. Pemeriksaan Tujuan
   Setelah suatu simpul diekspansi, algoritma akan memeriksa apakah simpul tersebut merupakan simpul tujuan. Jika iya, maka algoritma akan mengembalikan total biaya untuk mencapai simpul tersebut beserta jalur yang ditempuh.
6. Pengulangan
   Proses ini akan diulang terus-menerus, hingga:
   - *List frontier* kosong, atau
   - Simpul suatu tujuan telah ditemukan.

Algoritma tersebut dapat dituliskan dalam Pseudocode sebagai berikut.

```
procedure AStar(input start: integer, input goal: integer)
{
  Menemukan jalur dari simpul start ke goal menggunakan biaya terkecil +
heuristik
  Masukan: start dan goal adalah simpul
  Keluaran: semua simpul yang dikunjungi ditulis ke layar dan jalur
ditemukan
}

Deklarasi
  antrian_prioritas : struktur data (f_score, simpul)
  dikunjungi[0..n] : array of boolean
  asal[0..n] : array of integer
  g_score[0..n] : array of integer // biaya dari start ke node
  f_score[0..n] : array of integer // estimasi total biaya (g + h)
  current, neighbor : integer

Algoritma:
  inisialisasi semua dikunjungi[i] ← false
  untuk semua i: g_score[i] ← ∞
  untuk semua i: f_score[i] ← ∞
  asal[start] ← -1
  g_score[start] ← 0
  f_score[start] ← h[start]
  masukkan (f_score[start], start) ke dalam antrian_prioritas
```

```
  while antrian_prioritas tidak kosong do
    keluarkan (f_now, current) dengan f_score terkecil dari
antrian_prioritas

    if dikunjungi[current] = true then
      lanjutkan ke iterasi berikutnya
    endif

    dikunjungi[current] ← true
    write(current)

    if current = goal then
      keluarkan jalur dari asal[] mulai dari goal ke start
      stop
    endif

    for setiap tetangga neighbor dari current do
      if dikunjungi[neighbor] = false then
        temp_g_score ← g_score[current] + biaya(current, neighbor)

        if temp_g_score < g_score[neighbor] then
          asal[neighbor] ← current
          g_score[neighbor] ← temp_g_score
          f_score[neighbor] ← g_score[neighbor] + h[neighbor]
          masukkan (f_score[neighbor], neighbor) ke dalam
antrian_prioritas
        endif
      endif
    endfor
  endwhile
```

BAB II
ANALISIS ALGORITMA

2.1 Definisi f(n) dan g(n)

Dalam algoritma *pathfinding,* terdapat 3 hal yang mempengaruhi pencarian yang akan dilakukan untuk menemukan jalur terpendek, yaitu biaya aktual (g(n)), estimasi biaya (h(n), dan total dari biaya aktual dan estimasi (f(n)).

Fungsi g(n) akan menghasilkan biaya total (*cost*) yang telah ditempuh untuk mencapai simpul n dari simpul awal (start). Biaya ini dihitung berdasarkan penjumlahan bobot/biaya dari setiap langkah yang sudah dilakukan hingga mencapai simpul n.

Fungsi h(n) atau dapat disebut sebagai fungsi heuristik, yaitu menghasilkan perkiraan biaya dari simpul n ke simpul *goal*. Heuristik yang digunakan dapat berbeda-beda dan tidak selalu dijamin akurat, namun harus tidak melebih-lebihkan biaya sebenarnya (agar algoritma A* optimal). Hal ini disebut sebagai heuristik yang *admissible*. Pada *rush hour solver*, digunakan beberapa pilihan heuristik, seperti *manhattan distance* dan *blocking*. *Manhattan Distance* menggunakan jumlah langkah horizontal dan vertikal minimum untuk mencapai titik tujuan dari titik asal. Sedangkan, *Blocking Heuristic* akan menghitung jumlah hambatan (*piece car non-primary*) yang berada di jalur menuju tujuan.

Fungsi f(n) merupakan fungsi penjumlahan dari biaya aktual (g(n)) dan estimasi biaya (h(n)). Nilai yang diperoleh adalah nilai evaluasi total dari simpul n.

2.2 Pemeriksaan Heuristik A*

Heuristik yang digunakan dalam algoritma A* diharapkan adalah heuristik yang admissible. Jika heuristik selalu underestimate atau tepat sama dengan biaya sebenarnya, maka A* dijamin akan menemukan jalur optimal dan inilah yang disebut admissible. Admissible heuristik dapat dituliskan juga, bahwa untuk setiap simpul n,

$$h(n) \leq h^*(n)$$

Di mana:

- h(n) adalah nilai heuristik dari simpul n (perkiraan biaya dari n ke tujuan).
- h*(n) adalah biaya sebenarnya jalur terpendek dari n ke tujuan.

Heuristik yang digunakan pada rush hour solver ini adalah Manhattan Distance dan Blocking Heuristic. *Manhattan Distance* menggunakan jumlah langkah horizontal dan vertikal minimum untuk mencapai titik tujuan dari titik asal. Hal ini akan selalu admissible dan sekaligus tepat untuk algoritma *rush hour solver* karena gerakan piece car pada rush hour hanya bisa berpindah secara vertikal dan horizontal. Sebab itu, perhitungan hasilnya merupakan jarak terpendek yang mungkin ditempuh di grid tanpa

gerakan diagonal dan ini tidak pernah melebih-lebihkan biaya sebenarnya (nilai estimasi tidak lebih besar daripada biaya jalur terpendek sesungguhnya yang harus ditempuh).

Heuristik lainnya ialah Blocking Heuristic yang akan menghitung jumlah hambatan (*piece car non-primary*) yang berada di jalur menuju tujuan. Fungsi ini juga admissible dan tepat digunakan untuk *rush hour solver* karena setiap mobil penghalang harus digeser minimal satu kali agar jalan mobil target (*primary*) terbuka. Sehingga, jumlah mobil penghalang adalah perkiraan langkah minimal tambahan yang harus dilakukan.

2.3 Perbandingan UCS dengan BFS pada Rush Hour

Pada permainan Rush Hour, algoritma *Uniform Cost Search* (UCS) dan *Breadth First Search* (BFS) memiliki perilaku yang serupa, namun tetap terdapat perbedaan mendasar dari sisi penanganan bobot biaya. BFS menelusuri seluruh tetangga dari node yang sedang dieksekusi secara menyeluruh dan dalam urutan yang tetap, tanpa mempertimbangkan bobot atau jarak. Sementara itu, UCS memilih jalur berdasarkan akumulasi biaya terkecil, sehingga urutan eksplorasi dapat berbeda tergantung nilai biaya total dari setiap jalur.

Dalam implementasi Rush Hour Solver yang kami rancang, setiap aksi pergerakan mobil diberi bobot biaya **sebesar jarak (*distance*)** yang ditempuh. Dengan demikian, **algoritma UCS akan mempertimbangkan perbedaan biaya antar aksi berdasarkan seberapa jauh mobil bergerak.**

Konsekuensinya, UCS akan mengantri dan mengeksekusi node berdasarkan total biaya terkecil, bukan sekadar urutan eksplorasi seperti pada BFS. Hal ini menyebabkan urutan eksplorasi node pada UCS bisa berbeda dari BFS, karena UCS memprioritaskan jalur dengan akumulasi biaya terendah, sedangkan BFS memperlakukan semua aksi seolah memiliki biaya yang sama. **Sebenarnya, UCS dapat sama dengan BFS bila mempertimbangkan sejauh apapun gerakannya, akan dihitung sebagai cost = 1 (menjadi 1 step).**

2.4 Analisis Teoritis A* dengan UCS

Penerapan algoritma A* dan UCS pada penyelesaian Rush Hour menunjukkan perbedaan dari segi jumlah node yang dieksplorasi, waktu penyelesaian, dan kompleksitas. Dari sisi jumlah node, A* cenderung mengeksplorasi lebih sedikit node dibandingkan UCS. Hal ini disebabkan karena A* mempertimbangkan total biaya kumulatif dari simpul awal ke simpul saat ini (g(n)) dan estimasi biaya ke tujuan (h(n)), sehingga pencarian lebih terarah dan efisien.

Dampaknya, waktu yang dibutuhkan oleh A* untuk menemukan solusi juga umumnya lebih singkat dibandingkan UCS, karena jumlah node yang diproses lebih sedikit. UCS sendiri hanya mempertimbangkan biaya aktual (g(n)) tanpa mempertimbangkan estimasi ke tujuan, sehingga dapat menjelajahi lebih banyak jalur yang tidak menjanjikan.

Dari sisi kompleksitas, dalam kasus terburuk, kedua algoritma ini akan menjelajahi semua node berdasarkan biaya terkecil dengan $O(b^m)$, dengan b adalah *branching factor*, yaitu jumlah rata-rata cabang yang dihasilkan pada setiap simpul, dan d merupakan *depth* (kedalaman) dari solusi optimal, yaitu jumlah langkah dari simpul awal (start) ke simpul tujuan (*goal*) melalui jalur terbaik.

Namun, keunggulan A* terletak pada kemampuan heuristiknya. Jika fungsi heuristik yang digunakan cukup baik (admissible dan konsisten), A* mampu menyaring banyak simpul yang tidak relevan dan lebih fokus pada jalur yang berpotensi memberikan solusi optimal, sehingga kompleksitas aktualnya bisa jauh lebih kecil dibandingkan UCS.

2.5 Jaminan Solusi oleh Greedy Best First Search

Greedy Best First Search (Greedy BFS) tidak selalu menemukan solusi yang optimal dan bahkan tidak menjamin menemukan solusi sama sekali, terutama jika tidak ditangani dengan benar, seperti tidak mencatat node yang telah dikunjungi (*visited*). Greedy BFS hanya berfokus pada nilai heuristik (h(n)) untuk menentukan node mana yang akan dieksplorasi berikutnya, tanpa memperhitungkan biaya aktual yang telah ditempuh (g(n)).

Hal ini membuat Greedy BFS cenderung "serakah", karena hanya memilih jalur yang tampak paling dekat ke tujuan berdasarkan estimasi, meskipun bisa jadi jalur tersebut lebih panjang atau bahkan buntu.

Berbeda dengan A* dan UCS, keduanya mempertimbangkan total biaya yang telah ditempuh secara aktual. UCS menggunakan g(n) (biaya dari start ke node saat ini) untuk menjamin solusi optimal, sedangkan A* menggabungkan g(n) dengan estimasi h(n) sehingga mampu mencari solusi optimal dengan lebih efisien jika heuristiknya *admissible* dan konsisten.

Dengan demikian, A* dan UCS menjamin solusi yang lengkap dan optimal, sedangkan Greedy BFS hanya efisien pada beberapa kasus terbatas namun tidak menjamin optimalitas dan tidak selalu lengkap.

BAB III

STRUKTUR DAN SOURCE CODE PROGRAM UTAMA

3.1 Struktur Program

Program memiliki struktur folder sebagai berikut.

```
├── README.md
├── RushHourSolver.zip
├── build
│   ├── RushHourSolver.jar
│   └── RushHourSolverCLI.jar
├── doc
│   ├── img
│   └── Tucil3_13523091_13523107.pdf
├── src
│   ├── algorithm
│   │   ├── AStar.java
│   │   ├── Algorithm.java
│   │   ├── BeamSearch.java
│   │   ├── GBFS.java
│   │   ├── SolutionPath.java
│   │   └── UCS.java
│   ├── cli
│   │   └── CLI.java
│   ├── gui
│   │   ├── LoadingController.java
│   │   ├── LoadingScreen.fxml
│   │   ├── MainController.java
│   │   ├── MainView.fxml
│   │   ├── RushHourApp.java
│   │   └── assets
│   ├── heuristic
│   │   ├── BlockingHeuristic.java
│   │   ├── Heuristic.java
│   │   └── ManhattanDistance.java
│   ├── model
│   │   ├── Board.java
│   │   ├── Move.java
│   │   ├── Piece.java
│   │   ├── Position.java
│   │   └── State.java
│   └── util
│       ├── BoardParser.java
│       └── Constants.java
```

```
│       └── FileHandler.java
└── test
    ├── input
    └── output
```

Keterangan :

- **src** : berisi source code program dalam bentuk file **.java**.
- **build** : berisi file **.jar** hasil kompilasi program GUI dan CLI yang siap dijalankan.
- **test** : berisi hasil pengujian yang dicantumkan pada laporan.
- **doc** : berisi laporan tugas kecil dan dokumentasi program.

3.2 Source Code

Dalam implementasi program penyelesaian *Rush Hour Puzzle* menggunakan algoritma *pathfinding* dengan bahasa Java, pendekatan paradigma Object-Oriented Programming (OOP) digunakan karena Java sendiri berbasis OOP. Dengan pendekatan ini, setiap komponen diorganisir dalam bentuk kelas (*class*) yang memiliki tanggung jawab spesifik, sehingga meningkatkan modularitas dan kemudahan dalam pengelolaan kode. Program ini tersedia dalam dua antarmuka, yaitu Command Line Interface (CLI) dan Graphical User Interface (GUI). Struktur program dibagi ke dalam beberapa package sesuai tanggung jawabnya, yaitu `model` berisi representasi objek-objek utama seperti *board*, *piece*, *position*, *move*, dan *state*, `algorithm` untuk implementasi algoritma pencarian seperti UCS, *Greedy Best First Search*, A*, dan *Beam Search*, `heuristic` untuk fungsi-fungsi heuristik, `util` sebagai kumpulan utilitas seperti pembaca file dan parser papan, `cli` untuk antarmuka berbasis terminal, serta `gui` yang menampilkan animasi visual. Berikut source code implementasi program.

3.2.1 Class Algorithm

```java
package algorithm;

import heuristic.Heuristic;
import model.Board;

public interface Algorithm {
    SolutionPath findSolution(Board initialBoard);

    String getName();

    default void setHeuristic(Heuristic h) {
    }
}
```

3.2.2 Class AStar

```java
package algorithm;

import heuristic.Heuristic;
import java.util.Comparator;
import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Set;
import model.Board;
import model.State;
import util.Constants;

public class AStar implements Algorithm {

    private Heuristic heuristic;

    @Override
    public SolutionPath findSolution(Board initialBoard) {
        long startTime = System.currentTimeMillis();
        int nodesVisited = 0;

        State initialState = new State(initialBoard);
        PriorityQueue<State> frontier = new PriorityQueue<>(
                Comparator.comparing(s -> s.getHeuristicValue() + s.getCost()));

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

        frontier.add(initialState);

        while (!frontier.isEmpty()) {
            State currentState = frontier.poll();
            nodesVisited++;

            if (currentState.getBoard().isSolved()) {
                long endTime = System.currentTimeMillis();
                List<State> path = currentState.getSolutionPath();
                return new SolutionPath(path, nodesVisited, endTime - startTime);
            }

            String boardStr = currentState.getBoard().toString();
            if (visited.contains(boardStr)) {
                continue;
            }

            visited.add(boardStr);

            List<State> childStates = currentState.generateChildStates();

            for (State childState : childStates) {
                String childBoardStr = childState.getBoard().toString();
                if (!visited.contains(childBoardStr)) {
                    childState.setHeuristicValue(heuristic.calculate(childState.getBoard()));
                    frontier.add(childState);
                }
            }
        }

        // No solution found
        long endTime = System.currentTimeMillis();
        return new SolutionPath(nodesVisited, endTime - startTime);
    }

    @Override
    public void setHeuristic(Heuristic heuristic) {
        this.heuristic = heuristic;
    }

    @Override
    public String getName() {
        return Constants.ASTAR;
    }
}
```

3.2.3 Class GBFS

```java
package algorithm;

import heuristic.Heuristic;
import java.util.Comparator;
import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Set;
import model.Board;
import model.State;
import util.Constants;

public class GBFS implements Algorithm {

    private Heuristic heuristic;

    @Override
    public SolutionPath findSolution(Board initialBoard) {
        long startTime = System.currentTimeMillis();
        int nodesVisited = 0;

        State initialState = new State(initialBoard);
        initialState.setHeuristicValue(heuristic.calculate(initialBoard));

        PriorityQueue<State> frontier = new PriorityQueue<>(Comparator.comparing(State::getHeuristicValue));
        Set<String> visited = new HashSet<>();

        frontier.add(initialState);

        while (!frontier.isEmpty()) {
            State currentState = frontier.poll();
            nodesVisited++;

            if (currentState.getBoard().isSolved()) {
                long endTime = System.currentTimeMillis();
                List<State> path = currentState.getSolutionPath();
                return new SolutionPath(path, nodesVisited, endTime - startTime);
            }

            String boardStr = currentState.getBoard().toString();
            if (visited.contains(boardStr)) {
                continue;
            }

            visited.add(boardStr);

            List<State> childStates = currentState.generateChildStates();

            for (State childState : childStates) {
                String childBoardStr = childState.getBoard().toString();
                if (!visited.contains(childBoardStr)) {
                    childState.setHeuristicValue(heuristic.calculate(childState.getBoard()));
                    frontier.add(childState);
                }
            }
        }

        // No solution found
        long endTime = System.currentTimeMillis();
        return new SolutionPath(nodesVisited, endTime - startTime);
    }
    @Override
    public void setHeuristic(Heuristic heuristic) {
        this.heuristic = heuristic;
    }

    @Override
    public String getName() {
        return Constants.GBFS;
    }
}
```

3.2.4 Class SolutionPath

```java
package algorithm;

import java.util.ArrayList;
import java.util.List;
import model.Move;
import model.State;

public class SolutionPath {
    private List<State> path; // Urutan state dari awal sampai solusi
    private int nodesVisited;
    private long executionTimeMs;
    private boolean solutionFound;

    public SolutionPath(List<State> path, int nodesVisited, long executionTimeMs) {
        this.path = path;
        this.nodesVisited = nodesVisited;
        this.executionTimeMs = executionTimeMs;
        this.solutionFound = true;
    }
```

```java
    // Constructor untuk solusi tidak ditemukan
    public SolutionPath(int nodesVisited, long executionTimeMs) {
        this.path = new ArrayList<>();
        this.nodesVisited = nodesVisited;
        this.executionTimeMs = executionTimeMs;
        this.solutionFound = false;
    }

    public List<State> getPath() {
        return path;
    }

    public int getStepCount() {
        // Kurangi 1 karena path termasuk state awal
        return path.size() > 0 ? path.size() - 1 : 0;
    }

    public int getNodesVisited() {
        return nodesVisited;
    }

    public long getExecutionTimeMs() {
        return executionTimeMs;
    }

    public boolean isSolutionFound() {
        return solutionFound;
    }
```

```
    public int getGroupedStepCount() {
        if (path.size() <= 1)
            return 0;

        int count = 1; // Setidaknya satu grup
        Move prevMove = path.get(1).getLastMove();

        for (int i = 2; i < path.size(); i++) {
            Move currentMove = path.get(i).getLastMove();
            if (!currentMove.equals(prevMove)) {
                count++;
                prevMove = currentMove;
            }
        }

        return count; // Tidak perlu tambah count++ lagi karena sudah dimulai dari 1
    }

    public List<String> getGroupedMoveDescriptions() {
        List<String> result = new ArrayList<>();
        if (path.size() <= 1)
            return result;

        Move prevMove = path.get(1).getLastMove();
        int repeat = 1;

        for (int i = 2; i < path.size(); i++) {
            Move currentMove = path.get(i).getLastMove();
            if (currentMove.equals(prevMove)) {
                repeat++;
            } else {
                result.add(formatMove(prevMove, repeat));
                prevMove = currentMove;
                repeat = 1;
            }
        }

        result.add(formatMove(prevMove, repeat)); // Tambahkan langkah terakhir
        return result;
    }

    private String formatMove(Move move, int count) {
        return move.toString() + (count > 1 ? " x" + count : "");
    }

}
```

3.2.5 Class UCS

```java
package algorithm;

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

import model.Board;
import model.State;
import util.Constants;

public class UCS implements Algorithm {

    @Override
    public SolutionPath findSolution(Board initialBoard) {
        long startTime = System.currentTimeMillis();
        int nodesVisited = 0;

        State initialState = new State(initialBoard);
        PriorityQueue<State> frontier = new PriorityQueue<>(Comparator.comparing(State::getCost));

        // Use a set for visited states based on board configuration
        Set<String> visited = new HashSet<>();

        frontier.add(initialState);

        while (!frontier.isEmpty()) {
            State currentState = frontier.poll();
            nodesVisited++;

            if (currentState.getBoard().isSolved()) {
                long endTime = System.currentTimeMillis();
                List<State> path = currentState.getSolutionPath();
                return new SolutionPath(path, nodesVisited, endTime - startTime);
            }

            String boardStr = currentState.getBoard().toString();

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

            visited.add(boardStr);

            List<State> childStates = currentState.generateChildStates();

            for (State childState : childStates) {
                String childBoardStr = childState.getBoard().toString();

                if (!visited.contains(childBoardStr)) {
                    frontier.add(childState);
                }
            }
        }

        // No solution found
        long endTime = System.currentTimeMillis();
        return new SolutionPath(nodesVisited, endTime - startTime);
    }

    @Override
    public String getName() {
        return Constants.UCS;
    }
}
```

3.2.6 Class Heuristic

```java
package heuristic;

import model.Board;

public interface Heuristic {
    int calculate(Board board);

    String getName();
}
```

3.2.7 Class ManhattanDistance

```java
package heuristic;

import model.*;
import util.Constants;

public class ManhattanDistance implements Heuristic {
    @Override
    public int calculate(Board board) {
        if (board.getExitPosition() == null || board.getPrimaryPieceId() == 0
            || !board.getPieces().containsKey(board.getPrimaryPieceId())) {
            return Integer.MAX_VALUE; // cek kondisi, heuristic gabisa dihitung
        }

        Piece primary = board.getPrimaryPiece();
        Position anchor = primary.getAnchor();

        int anchorRow = anchor.getRow();
        int anchorCol = anchor.getCol();

        Position exitPosition = board.getExitPosition();
        int exitRow = exitPosition.getRow();
        int exitCol = exitPosition.getCol();

        int rows = board.getRows();
        int cols = board.getCols();

        if (exitRow == -1) { // atas
            return Math.abs(anchorCol - exitCol) + (anchorRow + 1);
        } else if (exitRow == rows) { // bawah
            return Math.abs(anchorCol - exitCol) + (rows - anchorRow);
        } else if (exitCol == -1) { // kiri
            return Math.abs(anchorRow - exitRow) + (anchorCol + 1);
        } else if (exitCol == cols) { // kanan
            return Math.abs(anchorRow - exitRow) + (cols - anchorCol);
        } else { // jaga-jaga aja
            return Math.abs(anchorRow - exitRow) + Math.abs(anchorCol - exitCol);
        }
    }

    @Override
    public String getName() {
        return Constants.MANHATTAN_HEURISTIC;
    }
}
```

3.2.8 Class BlockingHeuristic

```java
package heuristic;

import model.Board;
import model.Piece;
import model.Position;
import util.Constants;
```

```java
public class BlockingHeuristic implements Heuristic {

    @Override
    public int calculate(Board board) { // kalau ga valid
        if (board.getExitPosition() == null || board.getPrimaryPieceId() == 0
            || !board.getPieces().containsKey(board.getPrimaryPieceId())) {
            return Integer.MAX_VALUE;
        }

        Piece primary = board.getPrimaryPiece();
        Position anchor = primary.getAnchor();
        int row = anchor.getRow();
        int col = anchor.getCol();
        int length = primary.getSize();

        Position exit = board.getExitPosition();
        int exitRow = exit.getRow();
        int exitCol = exit.getCol();

        int blockingCount = 0;

        // Arah exit gate
        int dirRow = 0, dirCol = 0;

        if (exitRow < 0)
            dirRow = -1; // atas
        else if (exitRow >= board.getRows())
            dirRow = 1; // bawah
        else if (exitCol < 0)
            dirCol = -1; // kiri
        else if (exitCol >= board.getCols())
            dirCol = 1; // kanan
        else
            return Integer.MAX_VALUE; // ga valid

        // posisi awal untuk cek
        int startRow = row;
        int startCol = col;
```

```java
        if (!primary.isHorizontal()) {
            if (dirRow > 0)
                startRow = row + length - 1;
        } else {
            if (dirCol > 0)
                startCol = col + length - 1;
        }

        int currentRow = startRow + dirRow;
        int currentCol = startCol + dirCol;

        while (currentRow >= 0 && currentRow < board.getRows() &&
                currentCol >= 0 && currentCol < board.getCols()) {

            int cellValue = board.getBoardArray()[currentRow][currentCol];
            if (cellValue != 0 && cellValue != board.getPrimaryPieceId()) { // ada mobil lain
                blockingCount++;
            }

            currentRow += dirRow;
            currentCol += dirCol;
        }

        return blockingCount;
    }

    @Override
    public String getName() {
        return Constants.BLOCKING_HEURISTIC;
    }
}
```

3.2.9 Class Board

```java
package model;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import util.Constants;

public class Board {
    private int rows;
    private int cols;
    private char[][] boardArray;
    private Position exitPosition;
    private Map<Character, Piece> pieces;
    private char primaryPieceId;

    public Board(int rows, int cols, char[][] boardArray) {
        this.rows = rows;
        this.cols = cols;
        this.boardArray = boardArray;
        this.pieces = new HashMap<>();
    }

    public Board(Board other) {
        this.rows = other.rows;
        this.cols = other.cols;
        this.boardArray = new char[rows][cols];

        for (int i = 0; i < rows; i++) {
            System.arraycopy(other.boardArray[i], srcPos:0, this.boardArray[i], destPos:0, cols);
        }

        if (other.exitPosition != null) {
            this.exitPosition = new Position(other.exitPosition);
        }

        this.pieces = new HashMap<>();
        for (Map.Entry<Character, Piece> entry : other.pieces.entrySet()) {
            this.pieces.put(entry.getKey(), new Piece(entry.getValue()));
        }

        this.primaryPieceId = other.primaryPieceId;
    }
```

```java
    public int getRows() {
        return rows;
    }

    public int getCols() {
        return cols;
    }

    public char[][] getBoardArray() {
        return boardArray;
    }

    public Position getPrimaryPiecePosition() {
        Piece primaryPiece = pieces.get(primaryPieceId);
        if (primaryPiece != null) {
            return primaryPiece.getAnchor();
        }
        return null;
    }

    public Position getExitPosition() {
        return exitPosition;
    }
```

```java
public void setExitPosition(Position exitPosition) {
    this.exitPosition = exitPosition;
}

public char getPrimaryPieceId() {
    return primaryPieceId;
}

public Piece getPrimaryPiece() {
    return pieces.get(primaryPieceId);
}

public Map<Character, Piece> getPieces() {
    return pieces;
}

public int getPieceCount() {
    return pieces.size();
}

public void addPiece(char id, Piece piece) {
    pieces.put(id, piece);
}
```

```java
public void setPrimaryPieceId(char primaryPieceId) {
    this.primaryPieceId = primaryPieceId;
}

public boolean isExitPosition(int row, int col) {
    return exitPosition != null && exitPosition.getRow() == row && exitPosition.getCol() == col;
}

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

    for (Map.Entry<Character, Piece> entry : pieces.entrySet()) {
        char id = entry.getKey();
        Piece piece = entry.getValue();

        int[] directions = piece.isHorizontal() ? new int[] { Move.LEFT, Move.RIGHT }
                : new int[] { Move.UP, Move.DOWN };

        for (int direction : directions) {
            int maxDistance = piece.canMove(this, direction);
            for (int dist = 1; dist <= maxDistance; dist++) {
                possibleMoves.add(new Move(id, direction, dist));
            }
        }
    }

    return possibleMoves;
}

public boolean movePiece(char id, int direction, int distance) {
    Piece piece = pieces.get(id);
    if (piece == null)
        return false;

    for (Position pos : piece.getAllPositions()) {
        int row = pos.getRow();
        int col = pos.getCol();
        if (row >= 0 && row < boardArray.length && col >= 0 && col < boardArray[0].length) {
            boardArray[row][col] = '.';
        }
    }

    piece.move(direction, distance);
```

```
        for (Position pos : piece.getAllPositions()) {
            int row = pos.getRow();
            int col = pos.getCol();

            if (row >= 0 && row < boardArray.length && col >= 0 && col < boardArray[0].length) {
                boardArray[row][col] = id;
            }
        }
        return true;
    }

    public int getPieceCountNoPrimary() {
        int count = 0;
        for (Piece piece : pieces.values()) {
            if (!piece.isPrimary()) {
                count++;
            }
        }
        return count;
    }

    public boolean validatePrimaryPieceAlignedWithExit() {
        if (exitPosition == null || primaryPieceId == 0 || !pieces.containsKey(primaryPieceId)) {
            return false;
        }

        Piece primary = getPrimaryPiece();
        Position anchor = primary.getAnchor();

        if (exitPosition.getRow() == -1 || exitPosition.getRow() == rows) {
            return !primary.isHorizontal() && exitPosition.getCol() >= anchor.getCol()
                    && exitPosition.getCol() < anchor.getCol() + 1;
        } else if (exitPosition.getCol() == -1 || exitPosition.getCol() == cols) {
            return primary.isHorizontal() && exitPosition.getRow() >= anchor.getRow()
                    && exitPosition.getRow() < anchor.getRow() + 1;
        }

        return false;
    }

    public boolean isSolved() {
        if (exitPosition == null || primaryPieceId == 0 || !pieces.containsKey(primaryPieceId)) {
            return false;
        }
```

```
        Piece primary = getPrimaryPiece();
        if (primary == null) {
            return false;
        }

        Position anchor = primary.getAnchor();
        int size = primary.getSize();

        if (primary.isHorizontal()) {
            int endCol = anchor.getCol() + size - 1;

            if (exitPosition.getCol() == cols) {
                return anchor.getCol() >= cols;
            }

            if (exitPosition.getCol() == -1) {
                return endCol < 0;
            }
        } else {
            int endRow = anchor.getRow() + size - 1;

            if (exitPosition.getRow() == rows) {
                return anchor.getRow() >= rows;
            }
```

```
            if (exitPosition.getRow() == -1) {
                return endRow < 0;
            }
        }

        return false;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();

        boolean hasLeftExit = (exitPosition != null && exitPosition.getCol() == -1);
        boolean hasRightExit = (exitPosition != null && exitPosition.getCol() == cols);
        boolean hasTopExit = (exitPosition != null && exitPosition.getRow() == -1);
        boolean hasBottomExit = (exitPosition != null && exitPosition.getRow() == rows);
```

```
        if (hasTopExit) {
            if (hasLeftExit)
                sb.append(str:" ");
            for (int j = 0; j < cols; j++) {
                if (exitPosition.getCol() == j) {
                    sb.append(Constants.EXIT_CHAR);
                } else {
                    sb.append(c:' ');
                }
            }
            if (hasRightExit)
                sb.append(str:" ");
            sb.append(str:"\n");
        }

        for (int i = 0; i < rows; i++) {
            if (hasLeftExit) {
                if (exitPosition != null && exitPosition.getRow() == i && exitPosition.getCol() == -1) {
                    sb.append(Constants.EXIT_CHAR);
                } else {
                    sb.append(c:' ');
                }
            }

            for (int j = 0; j < cols; j++) {
                sb.append(boardArray[i][j]);
            }

            if (hasRightExit) {
                if (exitPosition != null && exitPosition.getRow() == i && exitPosition.getCol() == cols) {
                    sb.append(Constants.EXIT_CHAR);
                } else {
                    sb.append(c:' ');
                }
            }

            sb.append(str:"\n");
        }
```

```
        if (hasBottomExit) {
            if (hasLeftExit)
                sb.append(str:" ");
            for (int j = 0; j < cols; j++) {
                if (exitPosition.getCol() == j) {
                    sb.append(Constants.EXIT_CHAR);
                } else {
                    sb.append(c:' ');
                }
            }
            if (hasRightExit)
                sb.append(str:" ");
            sb.append(str:"\n");
        }
```

```
    return sb.toString();
}

public String toStringWithColor() {
    StringBuilder sb = new StringBuilder();

    boolean hasLeftExit = (exitPosition != null && exitPosition.getCol() == -1);
    boolean hasRightExit = (exitPosition != null && exitPosition.getCol() == cols);
    boolean hasTopExit = (exitPosition != null && exitPosition.getRow() == -1);
    boolean hasBottomExit = (exitPosition != null && exitPosition.getRow() == rows);

    if (hasTopExit) {
        if (hasLeftExit)
            sb.append(str:" ");
        for (int j = 0; j < cols; j++) {
            if (exitPosition.getCol() == j) {
                sb.append(Constants.GREEN).append(Constants.EXIT_CHAR).append(Constants.RESET);
            } else {
                sb.append(str:" ");
            }
        }
        if (hasRightExit)
            sb.append(str:" ");
        sb.append(str:"\n");
    }
```

```
    for (int i = 0; i < rows; i++) {
        if (hasLeftExit) {
            if (exitPosition != null && exitPosition.getRow() == i && exitPosition.getCol() == -1) {
                sb.append(Constants.GREEN).append(Constants.EXIT_CHAR).append(Constants.RESET);
            } else {
                sb.append(str:" ");
            }
        }

        for (int j = 0; j < cols; j++) {
            char c = boardArray[i][j];
            if (c == Constants.PRIMARY_PIECE_CHAR) {
                sb.append(Constants.RED).append(c).append(Constants.RESET);
            } else {
                sb.append(c);
            }
        }

        if (hasRightExit) {
            if (exitPosition != null && exitPosition.getRow() == i && exitPosition.getCol() == cols) {
                sb.append(Constants.GREEN).append(Constants.EXIT_CHAR).append(Constants.RESET);
            } else {
                sb.append(str:" ");
            }
        }

        sb.append(str:"\n");
    }

    if (hasBottomExit) {
        if (hasLeftExit)
            sb.append(str:" ");
        for (int j = 0; j < cols; j++) {
            if (exitPosition.getCol() == j) {
                sb.append(Constants.GREEN).append(Constants.EXIT_CHAR).append(Constants.RESET);
            } else {
                sb.append(str:" ");
            }
        }
        if (hasRightExit)
            sb.append(str:" ");
        sb.append(str:"\n");
    }

    return sb.toString();
}
```

```
public String toStringWithColor(Move lastMove, Board beforeMoveBoard) {
    StringBuilder sb = new StringBuilder();

    boolean hasLeftExit = (exitPosition != null && exitPosition.getCol() == -1);
    boolean hasRightExit = (exitPosition != null && exitPosition.getCol() == cols);
    boolean hasTopExit = (exitPosition != null && exitPosition.getRow() == -1);
    boolean hasBottomExit = (exitPosition != null && exitPosition.getRow() == rows);

    List<Position> movePath = new ArrayList<>();
    if (lastMove != null) {
        Piece before = beforeMoveBoard.getPieces().get(lastMove.getPieceId());
        for (Position pos : before.getAllPositions()) {
            for (int d = 1; d <= lastMove.getDistance(); d++) {
                int row = pos.getRow();
                int col = pos.getCol();
                switch (lastMove.getDirection()) {
                    case Move.LEFT -> movePath.add(new Position(row, col - d));
                    case Move.RIGHT -> movePath.add(new Position(row, col + d));
                    case Move.UP -> movePath.add(new Position(row - d, col));
                    case Move.DOWN -> movePath.add(new Position(row + d, col));
                }
            }
        }
    }

    if (hasTopExit) {
        if (hasLeftExit)
            sb.append(str:" ");
        for (int j = 0; j < cols; j++) {
            if (exitPosition.getCol() == j) {
                sb.append(Constants.GREEN).append(Constants.EXIT_CHAR).append(Constants.RESET);
            } else {
                sb.append(str:" ");
            }
        }
        if (hasRightExit)
            sb.append(str:" ");
        sb.append(str:"\n");
    }

    for (int i = 0; i < rows; i++) {
        if (hasLeftExit) {
            if (exitPosition.getRow() == i) {
                sb.append(Constants.GREEN).append(Constants.EXIT_CHAR).append(Constants.RESET);
            } else {
                sb.append(str:" ");
            }
        }
```

```
        for (int j = 0; j < cols; j++) {
            char c = boardArray[i][j];
            Position current = new Position(i, j);
            boolean isMoved = movePath.contains(current);
            boolean isPrimary = (c == Constants.PRIMARY_PIECE_CHAR);
            boolean isMovingPiece = (lastMove != null && c == lastMove.getPieceId());

            if (isMoved) {
                if (isPrimary) {
                    sb.append(Constants.YELLOW).append(Constants.RED).append(c).append(Constants.RESET);
                } else if (isMovingPiece) {
                    sb.append(Constants.YELLOW).append(Constants.BLUE).append(c).append(Constants.RESET);
                } else {
                    sb.append(Constants.YELLOW).append(c).append(Constants.RESET);
                }
            } else if (isMovingPiece) {
                if (isPrimary) {
                    sb.append(Constants.RED).append(c).append(Constants.RESET);
                } else {
                    sb.append(Constants.BLUE).append(c).append(Constants.RESET);
                }
            } else if (isPrimary) {
                sb.append(Constants.RED).append(c).append(Constants.RESET);
```

```
                } else {
                    sb.append(c);
                }
            }

            if (hasRightExit) {
                if (exitPosition.getRow() == i) {
                    sb.append(Constants.GREEN).append(Constants.EXIT_CHAR).append(Constants.RESET);
                } else {
                    sb.append(str:" ");
                }
            }

            sb.append(str:"\n");
        }
```

```
        if (hasBottomExit) {
            if (hasLeftExit)
                sb.append(str:" ");
            for (int j = 0; j < cols; j++) {
                if (exitPosition.getCol() == j) {
                    sb.append(Constants.GREEN).append(Constants.EXIT_CHAR).append(Constants.RESET);
                } else {
                    sb.append(str:" ");
                }
            }
            if (hasRightExit)
                sb.append(str:" ");
            sb.append(str:"\n");
        }

        return sb.toString();
    }
}
```

3.2.10 Class Move

```
package model;

import util.Constants;

public class Move {
    public static final int UP = 0;
    public static final int RIGHT = 1;
    public static final int DOWN = 2;
    public static final int LEFT = 3;

    private char pieceId;
    private int direction;
    private int distance;

    public Move(char pieceId, int direction, int distance) {
        this.pieceId = pieceId;
        this.direction = direction;
        this.distance = distance;
    }

    public char getPieceId() {
        return pieceId;
    }
```

```java
    public int getDirection() {
        return direction;
    }

    public int getDistance() {
        return distance;
    }

    public String getDirectionString() {
        switch (direction) {
            case UP:
                return Constants.DIRECTION_STRINGS[0];
            case RIGHT:
                return Constants.DIRECTION_STRINGS[1];
            case DOWN:
                return Constants.DIRECTION_STRINGS[2];
            case LEFT:
                return Constants.DIRECTION_STRINGS[3];
            default:
                return "unknown";
        }
    }

    @Override
    public String toString() {
        return pieceId + "-" + getDirectionString();
    }
```

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

        Move move = (Move) obj;
        return pieceId == move.pieceId &&
                direction == move.direction &&
                distance == move.distance;
    }

    @Override
    public int hashCode() {
        int result = Character.hashCode(pieceId);
        result = 31 * result + direction;
        result = 31 * result + distance;
        return result;
    }
}
```

3.2.11 Class Piece

```java
package model;

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

public class Piece {
    private char id;
    private Position anchor;
    private int size;
    private boolean isPrimary;
    private boolean isHorizontal;
```

```java
public Piece(char id, Position anchor, int size, boolean isPrimary, boolean isHorizontal) {
    this.id = id;
    this.anchor = new Position(anchor);
    this.size = size;
    this.isPrimary = isPrimary;
    this.isHorizontal = isHorizontal;
}

public Piece(Piece other) {
    this.id = other.id;
    this.anchor = new Position(other.anchor);
    this.size = other.size;
    this.isPrimary = other.isPrimary;
    this.isHorizontal = other.isHorizontal;
}

public char getId() {
    return id;
}

public boolean isPrimary() {
    return isPrimary;
}

public boolean isHorizontal() {
    return isHorizontal;
}

public Position getAnchor() {
    return anchor;
}

public int getSize() {
    return size;
}
```

```java
public void move(int direction, int distance) {
    switch (direction) {
        case Move.UP:
            anchor.setRow(anchor.getRow() - distance);
            break;
        case Move.RIGHT:
            anchor.setCol(anchor.getCol() + distance);
            break;
        case Move.DOWN:
            anchor.setRow(anchor.getRow() + distance);
            break;
        case Move.LEFT:
            anchor.setCol(anchor.getCol() - distance);
            break;
    }
}

public int canMove(Board board, int direction) {
    if (isHorizontal && (direction == Move.UP || direction == Move.DOWN)) {
        return 0;
    }
    if (!isHorizontal && (direction == Move.LEFT || direction == Move.RIGHT)) {
        return 0;
    }

    int maxDistance = 0;
    char[][] boardArray = board.getBoardArray();
    int rowCount = board.getRows();
    int colCount = board.getCols();

    int row = anchor.getRow();
    int col = anchor.getCol();

    Position exitPos = board.getExitPosition();
```

```
        switch (direction) {
            case Move.UP:
                row--;
                if (isPrimary && row < 0 && exitPos.getRow() == -1 && exitPos.getCol() == col) {
                    return 1;
                }

                while (row >= 0 && boardArray[row][col] == '.') {
                    maxDistance++;
                    row--;
                }
                break;
```

```
            case Move.RIGHT:
                col += size;
                if (isPrimary && col >= colCount && exitPos.getRow() == row && exitPos.getCol() == colCount) {
                    return 1;
                }

                while (col < colCount && boardArray[row][col] == '.') {
                    maxDistance++;
                    col++;
                }
                break;

            case Move.DOWN:
                row += size;
                if (isPrimary && row >= rowCount && exitPos.getRow() == rowCount && exitPos.getCol() == col) {
                    return 1;
                }

                while (row < rowCount && boardArray[row][col] == '.') {
                    maxDistance++;
                    row++;
                }
                break;

            case Move.LEFT:
                col--;
                if (isPrimary && col < 0 && exitPos.getRow() == row && exitPos.getCol() == -1) {
                    return 1;
                }

                while (col >= 0 && boardArray[row][col] == '.') {
                    maxDistance++;
                    col--;
                }
                break;
        }

        return maxDistance;
    }

    public List<Position> getAllPositions() {
        List<Position> result = new ArrayList<>();
        for (int i = 0; i < size; i++) {
            int row = anchor.getRow() + (isHorizontal ? 0 : i);
            int col = anchor.getCol() + (isHorizontal ? i : 0);
            result.add(new Position(row, col));
        }
        return result;
    }
```

```
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append(str:"Piece ").append(id);
        if (isPrimary)
            sb.append(str:" (Primary)");
        sb.append(str:": [");

        for (int i = 0; i < size; i++) {
            int row = anchor.getRow() + (isHorizontal ? 0 : i);
            int col = anchor.getCol() + (isHorizontal ? i : 0);
            sb.append(str:"(").append(row).append(str:", ").append(col).append(str:")");
            if (i < size - 1)
                sb.append(str:", ");
        }

        sb.append(str:"]");
        return sb.toString();
    }
}
```

3.2.12 Class Position

```
package model;

public class Position {
    private int row;
    private 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;
    }

    public int getRow() {
        return row;
    }

    public int getCol() {
        return col;
    }

    public void setRow(int row) {
        this.row = row;
    }

    public void setCol(int col) {
        this.col = 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 31 * row + col;
    }
```

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

3.2.13 Class State

```
package model;

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

public class State implements Comparable<State> {
    private Board board;
    private int cost;
    private int heuristicValue;
    private Move lastMove;
    private State parent;
    private List<Move> moveHistory;

    public State(Board board) {
        this.board = board;
        this.cost = 0;
        this.heuristicValue = 0;
        this.lastMove = null;
        this.parent = null;
        this.moveHistory = new ArrayList<>();
    }

    public State(Board board, State parent, Move lastMove, int cost, int heuristicValue) {
        this.board = board;
        this.parent = parent;
        this.lastMove = lastMove;
        this.cost = cost;
        this.heuristicValue = heuristicValue;

        this.moveHistory = new ArrayList<>();
        if (parent != null) {
            this.moveHistory.addAll(parent.getMoveHistory());
        }

        if (lastMove != null) {
            this.moveHistory.add(lastMove);
        }
    }

    public int getTotalValue() {
        return cost + heuristicValue;
    }

    @Override
    public int compareTo(State other) {
        return Integer.compare(this.getTotalValue(), other.getTotalValue());
    }
```

```
    public List<State> generateChildStates(int heuristicValue) {
        List<State> childStates = new ArrayList<>();
        List<Move> possibleMoves = this.board.getPossibleMoves();

        for (Move move : possibleMoves) {
            Board newBoard = new Board(this.board);

            boolean moveSuccess = newBoard.movePiece(
                    move.getPieceId(),
                    move.getDirection(),
                    move.getDistance());
```

```
            if (moveSuccess) {
                State childState = new State(
                        newBoard,
                        this,
                        move,
                        this.cost + move.getDistance(),
                        heuristicValue);

                childStates.add(childState);
            }
        }

        return childStates;
    }

    public List<State> generateChildStates() {
        return generateChildStates(heuristicValue:0);
    }

    public List<State> getSolutionPath() {
        List<State> path = new ArrayList<>();
        State current = this;
        List<State> tempPath = new ArrayList<>();

        while (current != null) {
            tempPath.add(current);
            current = current.getParent();
        }

        if (!tempPath.isEmpty()) {
            State lastAddedState = tempPath.get(tempPath.size() - 1);
            path.add(lastAddedState);

            char lastPieceId = 0;
            int lastDirection = -1;
```

```
            for (int i = tempPath.size() - 2; i >= 0; i--) {
                State state = tempPath.get(i);
                Move move = state.getLastMove();

                if (move != null) {
                    if (move.getPieceId() != lastPieceId || move.getDirection() != lastDirection) {
                        path.add(state);
                        lastPieceId = move.getPieceId();
                        lastDirection = move.getDirection();
                    } else {
                        path.set(path.size() - 1, state);
                    }
                } else {
                    path.add(state);
                }
            }
        }

        return path;
    }

    public Board getBoard() {
        return board;
    }

    public void setBoard(Board board) {
        this.board = board;
    }

    public int getCost() {
        return cost;
    }

    public void setCost(int cost) {
        this.cost = cost;
    }
```

```java
    public int getHeuristicValue() {
        return heuristicValue;
    }

    public void setHeuristicValue(int heuristicValue) {
        this.heuristicValue = heuristicValue;
    }

    public Move getLastMove() {
        return lastMove;
    }

    public void setLastMove(Move lastMove) {
        this.lastMove = lastMove;
    }

    public State getParent() {
        return parent;
    }

    public void setParent(State parent) {
        this.parent = parent;
    }

    public List<Move> getMoveHistory() {
        return moveHistory;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;
        State state = (State) o;
        return Objects.equals(board, state.board);
    }

    @Override
    public int hashCode() {
        return Objects.hash(board);
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("State {\n");
        sb.append("  cost: ").append(cost).append(",\n");
        sb.append("  heuristic: ").append(heuristicValue).append(",\n");
        sb.append("  total: ").append(getTotalValue()).append(",\n");
        if (lastMove != null) {
            sb.append("  lastMove: ").append(lastMove).append(",\n");
        }
        sb.append("  board: \n").append(board.toString()).append("\n");
        sb.append("}");
        return sb.toString();
    }
}
```

3.2.14 Class BoardParser

```java
package util;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
```

```java
import model.*;

public class BoardParser {
    public static Board parseBoard(int rows, int cols, char[][] boardArray, Position exitPosition) {
        Board board = new Board(rows, cols, boardArray);
        board.setExitPosition(exitPosition);

        Map<Character, List<Position>> piecesPositions = findPiecesPositions(boardArray);
        createPieces(board, piecesPositions);

        return board;
    }

    private static Map<Character, List<Position>> findPiecesPositions(char[][] boardArray) {
        Map<Character, List<Position>> piecesPositions = new HashMap<>();

        int rows = boardArray.length;
        int cols = boardArray[0].length;

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                char cellChar = boardArray[i][j];

                if (cellChar == Constants.EMPTY_CELL_CHAR || cellChar == Constants.EXIT_CHAR) {
                    continue;
                }

                piecesPositions.computeIfAbsent(cellChar, k -> new ArrayList<>()).add(new Position(i, j));
            }
        }

        return piecesPositions;
    }

    private static void createPieces(Board board, Map<Character, List<Position>> piecesPositions) {
        for (Map.Entry<Character, List<Position>> entry : piecesPositions.entrySet()) {
            char id = entry.getKey();
            List<Position> positions = entry.getValue();

            Position anchor = positions.get(index:0);
            boolean isHorizontal = positions.stream().allMatch(p -> p.getRow() == anchor.getRow());
            int size = positions.size();

            boolean isPrimary = id == Constants.PRIMARY_PIECE_CHAR;
            Piece piece = new Piece(id, anchor, size, isPrimary, isHorizontal);
            board.addPiece(id, piece);

            if (isPrimary) {
                board.setPrimaryPieceId(id);
            }
        }
    }
}
```

3.2.15 Class Constants

```java
package util;

public class Constants {
    // Direction Constants
    public static final int UP = 0;
    public static final int RIGHT = 1;
    public static final int DOWN = 2;
    public static final int LEFT = 3;

    public static final String[] DIRECTION_STRINGS = { "atas", "kanan", "bawah", "kiri" };
```

```
    // Character Constants
    public static final char PRIMARY_PIECE_CHAR = 'P';
    public static final char EXIT_CHAR = 'K';
    public static final char EMPTY_CELL_CHAR = '.';

    // Box drawing characters for borders
    public static final char TOP_LEFT = '╔';
    public static final char TOP_RIGHT = '╗';
    public static final char BOTTOM_LEFT = '╚';
    public static final char BOTTOM_RIGHT = '╝';
    public static final char HORIZONTAL = '═';
    public static final char VERTICAL = '║';

    // ANSI colors for console output
    public static final String RESET = "\u001B[0m";
    public static final String RED = "\u001B[31m";
    public static final String GREEN = "\u001B[32m";
    public static final String WHITE = "\u001B[37m";
    public static final String BLACK = "\u001B[30m";
    public static final String YELLOW = "\u001B[33m";
    public static final String BLUE = "\u001B[34m";
    public static final String MAGENTA = "\u001B[35m";
    public static final String CYAN = "\u001B[36m";
    public static final String BRIGHT_YELLOW = "\u001B[93m";
    public static final String BRIGHT_GREEN = "\u001B[92m";
    public static final String BRIGHT_MAGENTA = "\u001B[95m";
    public static final String BRIGHT_CYAN = "\u001B[96m";
    public static final String BRIGHT_BLUE = "\u001B[94m";
    public static final String BRIGHT_RED = "\u001B[91m";
    public static final String BRIGHT_WHITE = "\u001B[97m";
    public static final String BOLD = "\u001B[1m";
    public static final String UNDERLINE = "\u001B[4m";
    public static final String BLINK = "\u001B[5m";
    public static final String INVERSE = "\u001B[7m";
    public static final String BG_BLACK = "\u001B[40m";
    public static final String BG_RED = "\u001B[41m";
    public static final String BG_GREEN = "\u001B[42m";
    public static final String BG_YELLOW = "\u001B[43m";
    public static final String BG_BLUE = "\u001B[44m";
    public static final String BG_PURPLE = "\u001B[45m";
    public static final String BG_CYAN = "\u001B[46m";
    public static final String BG_WHITE = "\u001B[47m";

    // Algorithm Labels
    public static final String UCS = "UCS";
    public static final String GBFS = "Greedy Best First Search";
    public static final String ASTAR = "A*";
    public static final String BEAM = "Beam Search";

    // Heuristic Labels
    public static final String BLOCKING_HEURISTIC = "Blocking Heuristic";
    public static final String MANHATTAN_HEURISTIC = "Manhattan Distance";

    // Helper functions
    public static String getAlgorithmName(int id) {
        return switch (id) {
            case 1 -> UCS;
            case 2 -> GBFS;
            case 3 -> ASTAR;
            case 4 -> BEAM;
            default -> "Unknown";
        };
    }

    public static String getHeuristicName(int id) {
        return switch (id) {
            case 1 -> BLOCKING_HEURISTIC;
            case 2 -> MANHATTAN_HEURISTIC;
            default -> "Unknown";
        };
    }
}
```

3.2.16 Class FileHandler

```java
package util;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.List;

import model.*;

import java.util.ArrayList;

public class FileHandler {
    public static Board readInputFile(String filePath) throws IOException {
        try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
            String[] dimensions = reader.readLine().trim().split(regex:"\\s+");
            int expectedRows = Integer.parseInt(dimensions[0]);
            int expectedCols = Integer.parseInt(dimensions[1]);

            int numPiecesNoPrimary = Integer.parseInt(reader.readLine().trim());

            List<String> allLines = new ArrayList<>();
            String line;
            while ((line = reader.readLine()) != null) {
                if (!line.trim().isEmpty()) {
                    allLines.add(line);
                }
            }

            int maxHeight = allLines.size();
            int maxWidth = 0;
            for (String s : allLines) {
                maxWidth = Math.max(maxWidth, s.length());
            }

            char[][] rawBoard = new char[maxHeight][maxWidth];
            for (int i = 0; i < maxHeight; i++) {
                String currentLine = allLines.get(i);
                for (int j = 0; j < maxWidth; j++) {
                    rawBoard[i][j] = (j < currentLine.length()) ? currentLine.charAt(j) : ' ';
                }
            }

            Position exitPosition = new Position(-1, -1);
            boolean hasTopK = false, hasBottomK = false, hasLeftK = false, hasRightK = false;
```

```java
            if (maxHeight > 0) {
                for (int j = 0; j < maxWidth; j++) {
                    if (rawBoard[0][j] == Constants.EXIT_CHAR) {
                        exitPosition = new Position(-1, j);
                        hasTopK = true;
                        break;
                    }
                }
            }

            if (!hasTopK && maxHeight > 0) {
                for (int j = 0; j < maxWidth; j++) {
                    if (rawBoard[maxHeight - 1][j] == Constants.EXIT_CHAR) {
                        exitPosition = new Position(expectedRows, j);
                        hasBottomK = true;
                        break;
                    }
                }
            }

            if (!hasTopK && !hasBottomK) {
                for (int i = 0; i < maxHeight; i++) {
                    if (maxWidth > 0 && rawBoard[i][0] == Constants.EXIT_CHAR) {
                        exitPosition = new Position(i, -1);
                        hasLeftK = true;
                        break;
                    }
                }
            }

            if (!hasTopK && !hasBottomK && !hasLeftK && maxWidth > 0) {
                for (int i = 0; i < maxHeight; i++) {
                    if (rawBoard[i][maxWidth - 1] == Constants.EXIT_CHAR) {
                        exitPosition = new Position(i, expectedCols);
                        hasRightK = true;
                        break;
                    }
                }
            }
```

```
        int actualRows = maxHeight - (hasTopK ? 1 : 0) - (hasBottomK ? 1 : 0);
        int actualCols = maxWidth - (hasLeftK ? 1 : 0) - (hasRightK ? 1 : 0);

        if (actualRows != expectedRows) {
            throw new IOException(
                "Error: Row count mismatch. Expected: " + expectedRows + ", Found: " + actualRows);
        }

        if (actualCols != expectedCols) {
            throw new IOException(
                "Error: Column count mismatch. Expected: " + expectedCols + ", Found: " + actualCols);
        }

        char[][] boardArray = new char[expectedRows][expectedCols];
        for (int i = 0; i < expectedRows; i++) {
            for (int j = 0; j < expectedCols; j++) {
                int sourceRow = i + startRow;
                int sourceCol = j + startCol;
                char cell = (sourceRow < maxHeight && sourceCol < maxWidth) ? rawBoard[sourceRow][sourceCol] : ' ';
                boardArray[i][j] = (cell == Constants.EXIT_CHAR) ? ' ' : cell;
            }
        }

        if (hasLeftK)
            exitPosition = new Position(exitPosition.getRow() - startRow, -1);
        if (hasRightK)
            exitPosition = new Position(exitPosition.getRow() - startRow, expectedCols);

        if (exitPosition.getRow() == -1 && exitPosition.getCol() == -1) {
            throw new IOException(message:"Error: No exit position (K) found on board.");
        }

        Board board = BoardParser.parseBoard(expectedRows, expectedCols, boardArray, exitPosition);

        if (board.getPieceCountNoPrimary() != numPiecesNoPrimary) {
            throw new IOException(message:"Error: Non-primary piece count mismatch.");
        }

        if (!board.validatePrimaryPieceAlignedWithExit()) {
            throw new IOException(message:"Error: Primary piece is not aligned with exit.");
        }

        return board;
    }
}
```

```
public static void writeSolutionToFile(
        String outputPath,
        Board initialBoard,
        List<State> solutionPath,
        String algorithm,
        int totalNodesVisited,
        long executionTime) throws IOException {

    try (BufferedWriter writer = new BufferedWriter(new FileWriter(outputPath))) {
        writer.write("Algorithm: " + algorithm);
        writer.newLine();
        writer.write("Step count: " + (solutionPath.size() - 1));
        writer.newLine();
        writer.write("Nodes visited: " + totalNodesVisited);
        writer.newLine();
        writer.write("Execution time: " + executionTime + " ms");
        writer.newLine();
        writer.newLine();

        writer.write(str:"Initial Board");
        writer.newLine();
        writer.write(initialBoard.toString());
        writer.newLine();

        for (int i = 1; i < solutionPath.size(); i++) {
            State state = solutionPath.get(i);
            Move move = state.getLastMove();

            writer.write("Move " + i + ": " +
                    move.getPieceId() + "-" +
                    move.getDirectionString());
            writer.newLine();
            writer.write(state.getBoard().toString());
            writer.newLine();
        }
    }
}
}
```

3.2.17 Class CLI

```java
package cli;

import util.Constants;
import util.FileHandler;
import algorithm.*;
import heuristic.*;
import model.*;

import java.io.File;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.List;
import java.util.concurrent.TimeUnit;

import static util.Constants.*;

public class CLI {

    private static final BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
    private static final int DEFAULT_WIDTH = 60;
    private static final String DEFAULT_OUTPUT_PATH = "test/output/solution.txt";

    Run | Debug
    public static void main(String[] args) {
        clearScreen();
        displayStartupAnimation();


        Board board = promptBoardPath();


        int algorithmChoice = promptAlgorithm();


        int heuristicChoice = 0;
        if (algorithmChoice != 1) {
            heuristicChoice = promptHeuristic();
        }


        printStartMessage(board, algorithmChoice, heuristicChoice);


        showProgressBar(message:"Solving puzzle");
```

```java
        try {

            Heuristic heuristic = null;
            Algorithm solver;
            switch (algorithmChoice) {
                case 1 -> {
                    solver = new UCS();
                }
                case 2 -> {
                    heuristic = (heuristicChoice == 1) ? new BlockingHeuristic() : new ManhattanDistance();
                    GBFS gbfs = new GBFS();
                    gbfs.setHeuristic(heuristic);
                    solver = gbfs;
                }
                case 3 -> {
                    heuristic = (heuristicChoice == 1) ? new BlockingHeuristic() : new ManhattanDistance();
                    AStar aStar = new AStar();
                    aStar.setHeuristic(heuristic);
                    solver = aStar;
                }
                case 4 -> {
                    heuristic = (heuristicChoice == 1) ? new BlockingHeuristic() : new ManhattanDistance();
                    BeamSearch beam = new BeamSearch();
                    beam.setHeuristic(heuristic);
                    solver = beam;
                }
                default -> throw new UnsupportedOperationException(message:"Algoritma tidak dikenali.");
            }


            SolutionPath solution = solver.findSolution(board);
```

```
            if (solution.isSolutionFound()) {
                clearScreen();
                printSolutionFoundAnimation();
                printSolutionToTerminal(board, solution.getPath(), solver.getName(), solution.getNodesVisited(),
                        solution.getExecutionTimeMs());


                boolean saveResult = promptSaveOption();

                if (saveResult) {
                    String outputPath = promptOutputPath();
                    FileHandler.writeSolutionToFile(outputPath, board, solution.getPath(), solver.getName(),
                            solution.getNodesVisited(), solution.getExecutionTimeMs());
                    System.out.println(BRIGHT_GREEN + BOLD + "Solution successfully written to: " + outputPath + RESET);
                } else {
                    System.out.println(BRIGHT_YELLOW + "Solution was not saved to file." + RESET);
                }
```

```
            } else {
                System.out.println("\n" + BG_RED + WHITE + BOLD + " NO SOLUTION FOUND " + RESET);
                System.out.println(RED + "The puzzle appears to be unsolvable with the selected algorithm." + RESET);
            }
        } catch (Exception e) {
            System.out.println("\n" + BG_RED + WHITE + BOLD + " ERROR " + RESET);
            System.out.println(RED + "An error occurred: " + e.getMessage() + RESET);
            e.printStackTrace();
        }


        System.out.println("\n" + BRIGHT_CYAN + "Press Enter to exit..." + RESET);
        try {
            reader.readLine();
        } catch (IOException e) {

        }
    }

    private static void clearScreen() {
        System.out.print(s:"\033[H\033[2J");
        System.out.flush();
    }

    private static void displayStartupAnimation() {
        String[] frames = {
                "   R U S H   H O U R   S O L V E R   ",
                "   R-U-S-H   H-O-U-R   S-O-L-V-E-R   ",
                "   R-U-S-H---H-O-U-R---S-O-L-V-E-R   ",
                "   R U S H - H O U R - S O L V E R   "
        };

        try {
            for (int i = 0; i < 3; i++) {
                for (String frame : frames) {
                    clearScreen();
                    String[] colors = { BRIGHT_RED, BRIGHT_YELLOW, BRIGHT_GREEN, BRIGHT_CYAN, BRIGHT_MAGENTA };


                    for (int j = 0; j < frame.length(); j++) {
                        String color = colors[j % colors.length];
                        System.out.print(color + frame.charAt(j) + RESET);
                    }
                    System.out.println("\n\n" + BRIGHT_CYAN + "Starting application..." + RESET);
                    TimeUnit.MILLISECONDS.sleep(timeout:150);
                }
            }
        } catch (InterruptedException e) {

        }

        printBanner();
    }
```

```java
private static void showProgressBar(String message) {
    int width = 30;
    System.out.print(s:"\n");

    try {
        for (int i = 0; i <= width; i++) {
            StringBuilder progressBar = new StringBuilder();
            progressBar.append(BRIGHT_CYAN);
            progressBar.append(str:"\r [");


            for (int j = 0; j < i; j++) {
                progressBar.append(BRIGHT_GREEN + "");
            }


            for (int j = i; j < width; j++) {
                progressBar.append(BLACK + "");
            }

            int percent = (i * 100) / width;
            progressBar.append(BRIGHT_CYAN + "] " + BRIGHT_WHITE + percent + "% " + message + "..." + RESET);

            System.out.print(progressBar);
            TimeUnit.MILLISECONDS.sleep(timeout:50);
        }


        TimeUnit.MILLISECONDS.sleep(timeout:500);
        System.out.println("\r" + " ".repeat(message.length() + 50));
    } catch (InterruptedException e) {

    }
}

private static void printSolutionFoundAnimation() {
    String message = "Solution found!";
    try {
        for (int i = 0; i < message.length(); i++) {
            clearScreen();
            System.out.println(x:"\n\n");
            System.out.print(BRIGHT_GREEN + " ".repeat((80 - message.length()) / 2));
            for (int j = 0; j <= i; j++) {
                System.out.print(BOLD + message.charAt(j) + RESET);
            }
            TimeUnit.MILLISECONDS.sleep(timeout:100);
        }


        TimeUnit.MILLISECONDS.sleep(timeout:500);
    } catch (InterruptedException e) {

    }
}
```

```java
private static void printBanner() {
    String title = " RUSH HOUR SOLVER CLI ";
    int width = DEFAULT_WIDTH;
    int padding = (width - title.length()) / 2;

    StringBuilder banner = new StringBuilder();
    banner.append(BRIGHT_CYAN + BOLD);
    banner.append(str:"\n");


    banner.append(" " + TOP_LEFT);
    for (int i = 0; i < width - 2; i++) {
        banner.append(HORIZONTAL);
    }
    banner.append(TOP_RIGHT + "\n");


    banner.append(" " + VERTICAL);
    banner.append(" ".repeat(padding));
    banner.append(BRIGHT_YELLOW + title + BRIGHT_CYAN);
    banner.append(" ".repeat(width - 2 - padding - title.length()));
    banner.append(VERTICAL + "\n");
```

```
        banner.append(" " + BOTTOM_LEFT);
        for (int i = 0; i < width - 2; i++) {
            banner.append(HORIZONTAL);
        }
        banner.append(BOTTOM_RIGHT);
        banner.append(RESET);

        System.out.println(banner.toString());
        System.out.println("\n" + BRIGHT_GREEN + "Welcome to the Rush Hour Puzzle Solver!" + RESET);
        System.out.println(BRIGHT_WHITE
                + "This application will help you find the optimal solution to Rush Hour puzzles." + RESET);
    }

    private static Board promptBoardPath() {
        Board board = null;
        while (board == null) {
            printBoxedPrompt(title:"FILE CONFIGURATION", message:"Enter the absolute path to the board configuration file:");

            try {
                String path = reader.readLine();
                if (path.trim().isEmpty()) {
                    printErrorMessage(message:"Path cannot be empty. Please enter a valid file path.");
                    continue;
                }

                showProgressBar(message:"Loading board");
                board = FileHandler.readInputFile(path);
                printSuccessMessage(message:"Board loaded successfully!");
```

```
                TimeUnit.SECONDS.sleep(timeout:1);
            } catch (IOException | InterruptedException e) {
                printErrorMessage("Invalid path or file cannot be read: " + e.getMessage());
            }
        }
        return board;
    }

    private static int promptAlgorithm() {
        int choice = -1;
        while (choice < 1 || choice > 4) {
            clearScreen();

            String[] options = {
                    "Uniform Cost Search (UCS)",
                    "Greedy Best First Search",
                    "A* Search",
                    "Beam Search"
            };

            printBoxedMenu(title:"ALGORITHM SELECTION", options);

            try {
                System.out.print(BRIGHT_YELLOW + "Enter your choice (1-4): " + RESET);
                String input = reader.readLine();

                if (input.trim().isEmpty()) {
                    printErrorMessage(message:"Input cannot be empty. Please enter a number between 1 and 4.");
                    TimeUnit.SECONDS.sleep(timeout:1);
                    continue;
                }

                try {
                    choice = Integer.parseInt(input);
                    if (choice < 1 || choice > 4) {
                        printErrorMessage(message:"Invalid choice. Please enter a number between 1 and 4.");
                        TimeUnit.SECONDS.sleep(timeout:1);
                    }
                } catch (NumberFormatException e) {
                    printErrorMessage(message:"Invalid input. Please enter a valid number.");
                    TimeUnit.SECONDS.sleep(timeout:1);
                }
            } catch (IOException | InterruptedException e) {
                printErrorMessage("An error occurred: " + e.getMessage());
            }
        }

        printSuccessMessage("Algorithm selected: " + Constants.getAlgorithmName(choice));
        return choice;
    }
```

```
private static int promptHeuristic() {
    int choice = -1;
    while (choice < 1 || choice > 2) {
        clearScreen();

        String[] options = {
            "Blocking Vehicles Heuristic",
            "Manhattan Distance Heuristic",
        };

        printBoxedMenu(title:"HEURISTIC SELECTION", options);

        try {
            System.out.print(BRIGHT_YELLOW + "Enter your choice (1-2): " + RESET);
            String input = reader.readLine();

            if (input.trim().isEmpty()) {
                printErrorMessage(message:"Input cannot be empty. Please enter a number between 1 and 2.");
                TimeUnit.SECONDS.sleep(timeout:1);
                continue;
            }

            try {
                choice = Integer.parseInt(input);
                if (choice < 1 || choice > 2) {
                    printErrorMessage(message:"Invalid choice. Please enter a number between 1 and 2.");
                    TimeUnit.SECONDS.sleep(timeout:1);
                }
            } catch (NumberFormatException e) {
                printErrorMessage(message:"Invalid input. Please enter a valid number.");
                TimeUnit.SECONDS.sleep(timeout:1);
            }
        } catch (IOException | InterruptedException e) {
            printErrorMessage("An error occurred: " + e.getMessage());
        }
    }

    printSuccessMessage("Heuristic selected: " + Constants.getHeuristicName(choice));
    return choice;
}

private static void printBoxedPrompt(String title, String message) {
    clearScreen();
    int width = Math.max(title.length() + 4, message.length() + 4);

    System.out.println(BRIGHT_CYAN);

    System.out.print(" " + TOP_LEFT);
    for (int i = 0; i < width; i++) {
        System.out.print(HORIZONTAL);
    }
    System.out.println(TOP_RIGHT);
```

```
    System.out.print(" " + VERTICAL + " " + BRIGHT_YELLOW + BOLD + title + RESET + BRIGHT_CYAN);
    System.out.print(" ".repeat(width - title.length() - 1));
    System.out.println(VERTICAL);


    System.out.print(" " + VERTICAL);
    for (int i = 0; i < width; i++) {
        System.out.print(s:"-");
    }
    System.out.println(VERTICAL);


    System.out.print(" " + VERTICAL + " " + BRIGHT_WHITE + message + RESET + BRIGHT_CYAN);
    System.out.print(" ".repeat(width - message.length() - 1));
    System.out.println(VERTICAL);


    System.out.print(" " + BOTTOM_LEFT);
    for (int i = 0; i < width; i++) {
        System.out.print(HORIZONTAL);
    }
    System.out.println(BOTTOM_RIGHT);
    System.out.println(RESET);
}
```

```
private static void printBoxedMenu(String title, String[] options) {
    int width = Math.max(title.length() + 4, b:60);
    for (String option : options) {
        width = Math.max(width, option.length() + 8);
    }

    System.out.println(BRIGHT_CYAN);

    System.out.print(" " + TOP_LEFT);
    for (int i = 0; i < width; i++) {
        System.out.print(HORIZONTAL);
    }
    System.out.println(TOP_RIGHT);


    System.out.print(" " + VERTICAL + " " + BRIGHT_YELLOW + BOLD + title + RESET + BRIGHT_CYAN);
    System.out.print(" ".repeat(width - title.length() - 1));
    System.out.println(VERTICAL);


    System.out.print(" " + VERTICAL);
    for (int i = 0; i < width; i++) {
        System.out.print(s:"-");
    }
    System.out.println(VERTICAL);
```

```
    for (int i = 0; i < options.length; i++) {
        System.out.print(" " + VERTICAL + " " + BRIGHT_WHITE + (i + 1) + ". " + options[i] + RESET + BRIGHT_CYAN);
        System.out.print(" ".repeat(width - options[i].length() - 4));
        System.out.println(VERTICAL);
    }


    System.out.print(" " + BOTTOM_LEFT);
    for (int i = 0; i < width; i++) {
        System.out.print(HORIZONTAL);
    }
    System.out.println(BOTTOM_RIGHT);
    System.out.println(RESET);
}

private static void printErrorMessage(String message) {
    System.out.println(BG_RED + WHITE + BOLD + " ERROR " + RESET + " " + RED + message + RESET);
}

private static void printSuccessMessage(String message) {
    System.out.println(BG_GREEN + BLACK + BOLD + " SUCCESS " + RESET + " " + BRIGHT_GREEN + message + RESET);
}

private static void printStartMessage(Board board, int algorithm, int heuristic) {
    clearScreen();
    int width = DEFAULT_WIDTH;

    System.out.println(BRIGHT_CYAN + BOLD);

    System.out.print(" " + TOP_LEFT);
    for (int i = 0; i < width; i++) {
        System.out.print(HORIZONTAL);
    }
    System.out.println(TOP_RIGHT);


    String title = " CONFIGURATION SUMMARY ";
    int padding = (width - title.length()) / 2;
    System.out.print(" " + VERTICAL);
    System.out.print(" ".repeat(padding));
    System.out.print(BRIGHT_YELLOW + BOLD + title + RESET + BRIGHT_CYAN + BOLD);
    System.out.print(" ".repeat(width - padding - title.length()));
    System.out.println(VERTICAL);


    System.out.print(" " + VERTICAL);
    for (int i = 0; i < width; i++) {
        System.out.print(s:"-");
    }
    System.out.println(VERTICAL);
```

```
        System.out.print(" " + VERTICAL + " " + BRIGHT_WHITE + "Board size: " + board.getRows() + "x"
                + board.getCols() + RESET + BRIGHT_CYAN + BOLD);
        System.out.print(" ".repeat(
                width - 13 - String.valueOf(board.getRows()).length() - String.valueOf(board.getCols()).length()));
        System.out.println(VERTICAL);

        System.out.print(" " + VERTICAL + " " + BRIGHT_WHITE + "Number of vehicles: " + board.getPieces().size()
                + RESET + BRIGHT_CYAN + BOLD);
        System.out.print(" ".repeat(width - 20 - String.valueOf(board.getPieces().size()).length()));
        System.out.println(VERTICAL);


        System.out.print(" " + VERTICAL + " " + BRIGHT_WHITE + "Selected algorithm: "
                + Constants.getAlgorithmName(algorithm) + RESET + BRIGHT_CYAN + BOLD);
        System.out.print(" ".repeat(width - 20 - Constants.getAlgorithmName(algorithm).length()));
        System.out.println(VERTICAL);

        if (algorithm >= 2 && algorithm <= 4) {
            System.out.print(" " + VERTICAL + " " + BRIGHT_WHITE + "Selected heuristic: "
                    + Constants.getHeuristicName(heuristic) + RESET + BRIGHT_CYAN + BOLD);
            System.out.print(" ".repeat(width - 20 - Constants.getHeuristicName(heuristic).length()));
            System.out.println(VERTICAL);
        }


        System.out.print(" " + BOTTOM_LEFT);
        for (int i = 0; i < width; i++) {
            System.out.print(HORIZONTAL);
        }
        System.out.println(BOTTOM_RIGHT);
        System.out.println(RESET);


        System.out.println(BRIGHT_YELLOW + BOLD + "\nInitial Board State:" + RESET);
        System.out.println(board.toStringWithColor());
    }

    private static boolean promptSaveOption() {
        while (true) {
            System.out.println("\n" + BRIGHT_CYAN + "Do you want to save the solution to a file?" + RESET);
            System.out.println(BRIGHT_WHITE + "1. " + BRIGHT_GREEN + "Yes" + RESET);
            System.out.println(BRIGHT_WHITE + "2. " + BRIGHT_RED + "No" + RESET);

            try {
                System.out.print(BRIGHT_YELLOW + "Enter your choice (1-2): " + RESET);
                String input = reader.readLine();

                if (input.trim().isEmpty()) {
                    printErrorMessage(message:"Input cannot be empty. Please enter 1 for Yes or 2 for No.");
                    continue;
                }
```

```
                if (input.equals(anObject:"1")) {
                    return true;
                } else if (input.equals(anObject:"2")) {
                    return false;
                } else {
                    printErrorMessage(message:"Invalid choice. Please enter 1 for Yes or 2 for No.");
                }
            } catch (IOException e) {
                printErrorMessage("Error reading your input: " + e.getMessage());
            }
        }
    }

    private static String promptOutputPath() {
        while (true) {
            printBoxedPrompt(title:"SAVE SOLUTION", message:"Enter the path where you want to save the solution file:");
            System.out.println(BRIGHT_CYAN + "Default path: " + BRIGHT_WHITE + DEFAULT_OUTPUT_PATH + RESET);
            System.out.println(BRIGHT_CYAN + "Press Enter to use default path or type a new path." + RESET);

            try {
                System.out.print(BRIGHT_YELLOW + "Path: " + RESET);
                String input = reader.readLine();

                if (input.trim().isEmpty()) {
                    return DEFAULT_OUTPUT_PATH;
                }
```

```
            File file = new File(input);
            File parentDir = file.getParentFile();

            if (parentDir != null && !parentDir.exists()) {
                System.out.println(
                        BRIGHT_YELLOW + "Directory doesn't exist. Do you want to create it? (Y/N)" + RESET);
                String createDir = reader.readLine();

                if (createDir.equalsIgnoreCase(anotherString:"Y")) {
                    if (parentDir.mkdirs()) {
                        printSuccessMessage(message:"Directory created successfully!");
                        return input;
                    } else {
                        printErrorMessage(message:"Failed to create directory. Using default path instead.");
                        return DEFAULT_OUTPUT_PATH;
                    }
                } else {
                    printErrorMessage(message:"Directory not created. Using default path instead.");
                    return DEFAULT_OUTPUT_PATH;
                }
            }

            return input;
        } catch (IOException e) {
            printErrorMessage("Error reading your input: " + e.getMessage());
            return DEFAULT_OUTPUT_PATH;
        }
```

```
    public static void printSolutionToTerminal(
            Board initialBoard,
            List<State> solutionPath,
            String algorithm,
            int totalNodesVisited,
            long executionTime) {

        int width = DEFAULT_WIDTH;

        System.out.println(BRIGHT_CYAN + BOLD);

        System.out.print(" " + TOP_LEFT);
        for (int i = 0; i < width; i++) {
            System.out.print(HORIZONTAL);
        }
        System.out.println(TOP_RIGHT);

        String title = "SOLUTION FOUND";
        int padding = (width - title.length()) / 2;
        System.out.print(" " + VERTICAL);
        System.out.print(" ".repeat(padding));
        System.out.print(BRIGHT_GREEN + BOLD + title + RESET + BRIGHT_CYAN + BOLD);
        System.out.print(" ".repeat(width - padding - title.length()));
        System.out.println(VERTICAL);
```

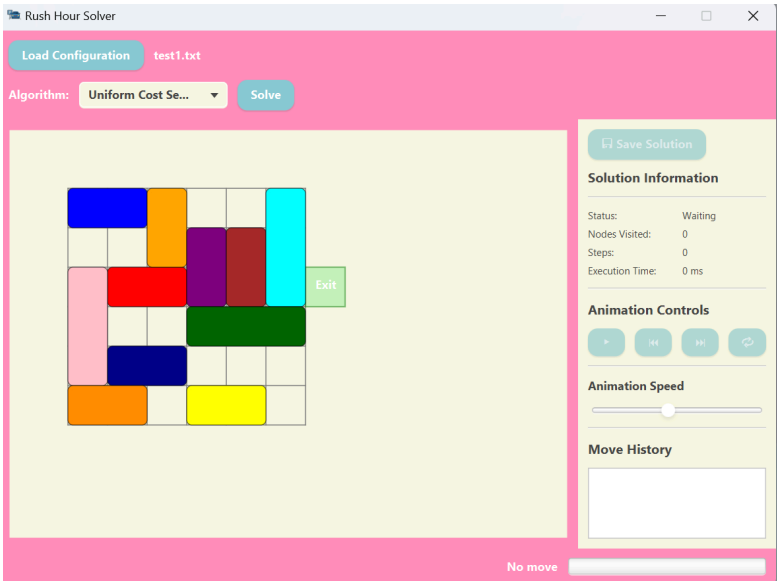
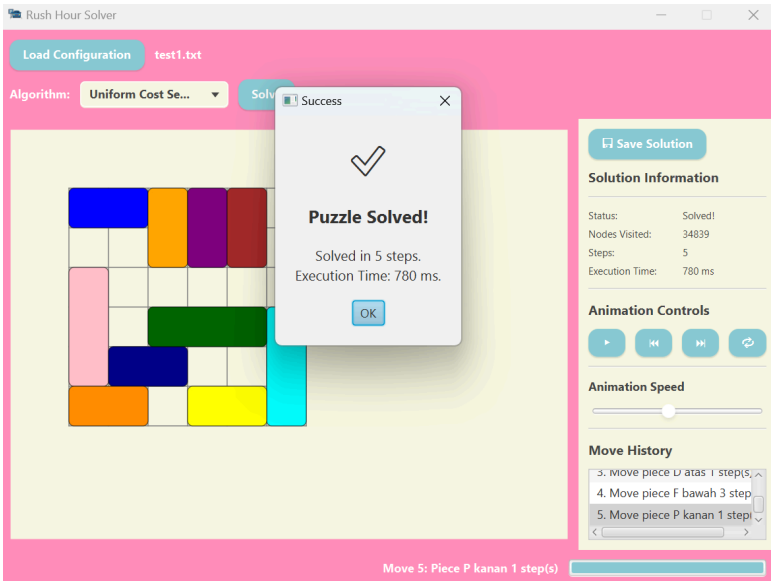
```
        System.out.print(" " + VERTICAL + " " + BRIGHT_WHITE + "Algorithm: " + algorithm + RESET + BRIGHT_CYAN + BOLD);
        System.out.print(" ".repeat(width - 12 - algorithm.length()));
        System.out.println(VERTICAL);

        System.out.print(" " + VERTICAL + " " + BRIGHT_WHITE + "Step count: " + (solutionPath.size() - 1) + RESET
                + BRIGHT_CYAN + BOLD);
        System.out.print(" ".repeat(width - 13 - String.valueOf(solutionPath.size() - 1).length()));
        System.out.println(VERTICAL);

        System.out.print(" " + VERTICAL + " " + BRIGHT_WHITE + "Nodes visited: " + totalNodesVisited + RESET
                + BRIGHT_CYAN + BOLD);
        System.out.print(" ".repeat(width - 16 - String.valueOf(totalNodesVisited).length()));
        System.out.println(VERTICAL);

        System.out.print(" " + VERTICAL + " " + BRIGHT_WHITE + "Execution time: " + executionTime + " ms" + RESET
                + BRIGHT_CYAN + BOLD);
        System.out.print(" ".repeat(width - 20- String.valueOf(executionTime).length()));
        System.out.println(VERTICAL);
```

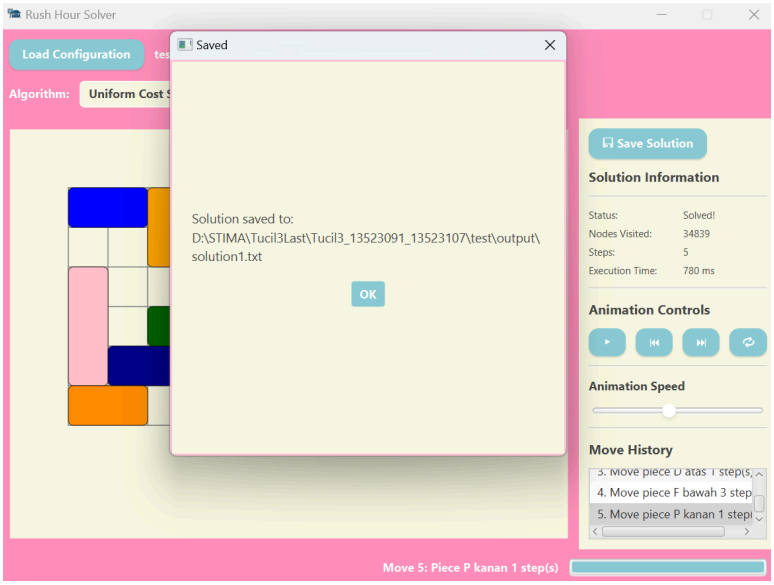
```
        System.out.print(" " + BOTTOM_LEFT);
        for (int i = 0; i < width; i++) {
            System.out.print(HORIZONTAL);
        }
        System.out.println(BOTTOM_RIGHT);
        System.out.println(RESET);

        System.out.println(BRIGHT_YELLOW + BOLD + "\nInitial Board" + RESET);
        System.out.println(initialBoard.toStringWithColor());


        if (solutionPath.size() > 1) {
            System.out.println(BRIGHT_CYAN + BOLD + "\nSolution Steps:" + RESET);

            Board previousBoard = initialBoard;

            for (int i = 1; i < solutionPath.size(); i++) {
                State state = solutionPath.get(i);
                Move move = state.getLastMove();
                Board currentBoard = state.getBoard();

                System.out.println(BRIGHT_GREEN + "Step " + i + ": " + BRIGHT_YELLOW + "Move " +
                        BRIGHT_WHITE + move.getPieceId() + BRIGHT_YELLOW + " " +
                        move.getDirectionString() + RESET);

                try {
                    TimeUnit.MILLISECONDS.sleep(timeout:300);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }


                System.out.println(currentBoard.toStringWithColor(move, previousBoard));

                previousBoard = currentBoard;
            }
        }
    }
}
```

BAB IV
HASIL PENGUJIAN DAN ANALISIS

4.1 Hasil Pengujian

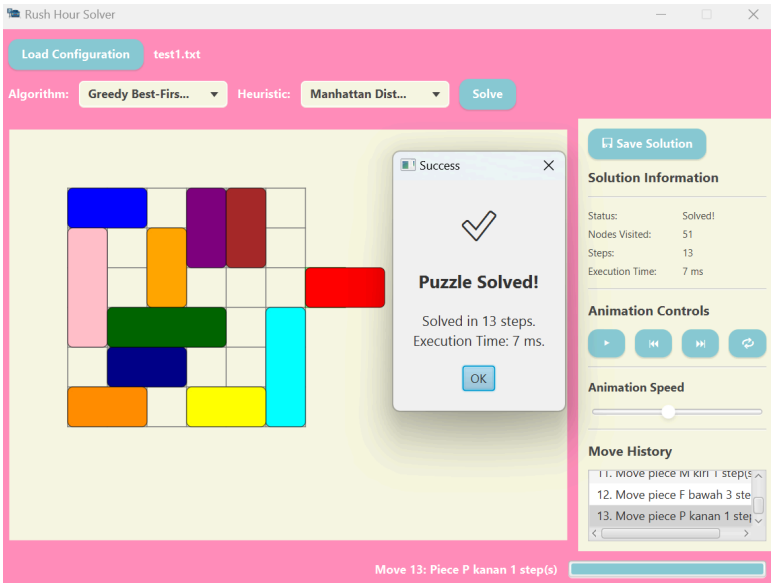
4.1.1 Test Case 1

Kondisi: Pengecekan kondisi bila *exit gate* berada di sebelah kanan.
Input

Output: Algoritma UCS


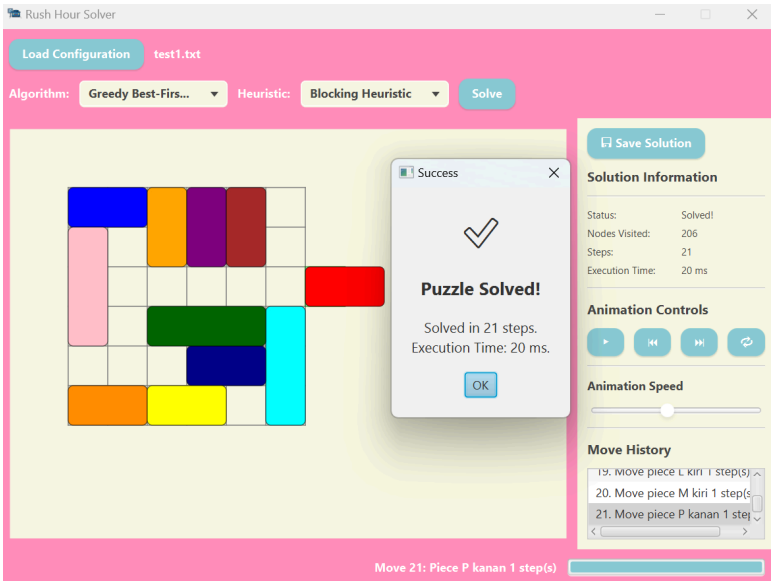



```
Algorithm: Uniform Cost Search
Step count: 5
Nodes visited: 34839
Execution time: 780 ms

Initial Board
AAB..F
..BCDF
GPPCDFK
G..III
GHH...
LL.MM.

Move 1: C-atas
AABC.F
..BCDF
GPP.DFK
G..III
GHH...
LL.MM.

Move 2: I-kiri
AABC.F
..BCDF
GPP.DFK
G.III.
GHH...
LL.MM.

Move 3: D-atas
AABCDF
..BCDF
GPP..FK
G.III.
GHH...
LL.MM.

Move 4: F-bawah
AABCD.
..BCD.
GPP...K
G.IIIF
GHH..F
LL.MMF

Move 5: P-kanan
AABCD.
..BCD.
G.....K
G.IIIF
GHH..F
LL.MMF
```

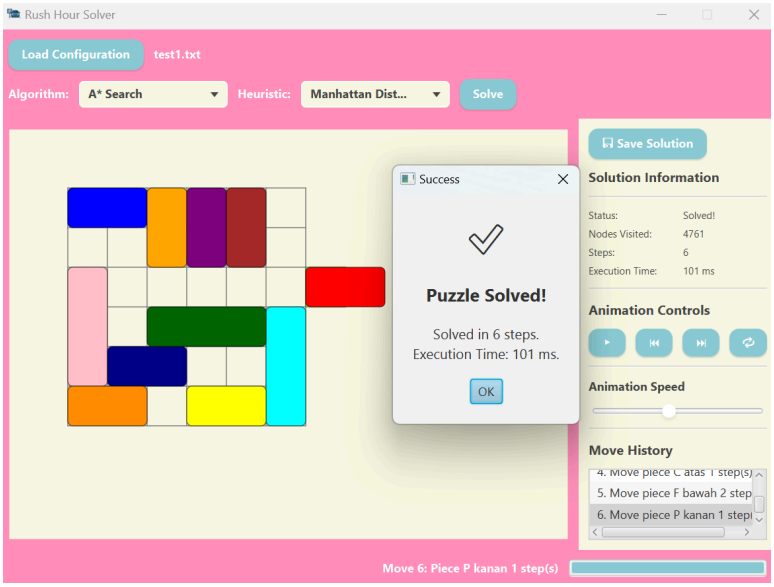
Output: Algoritma Greedy BFS dengan Heuristic Manhattan Distance



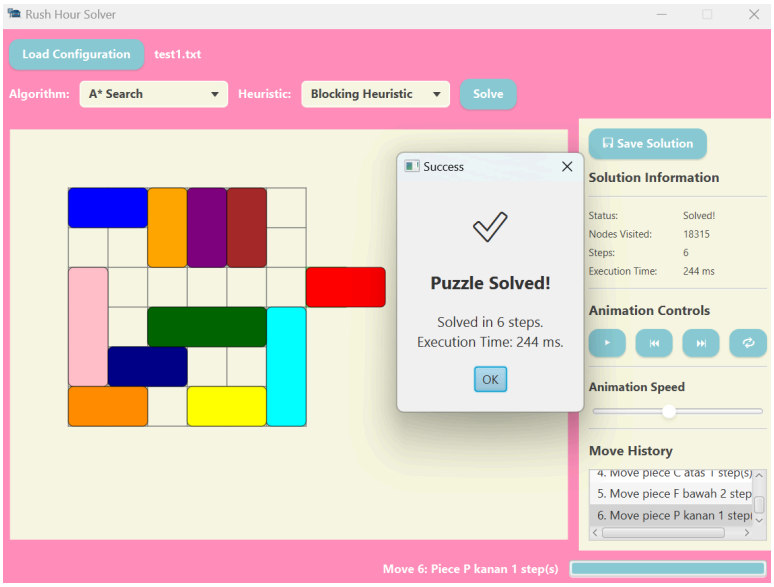
Output: Algoritma Greedy BFS dengan Blocking Heuristic



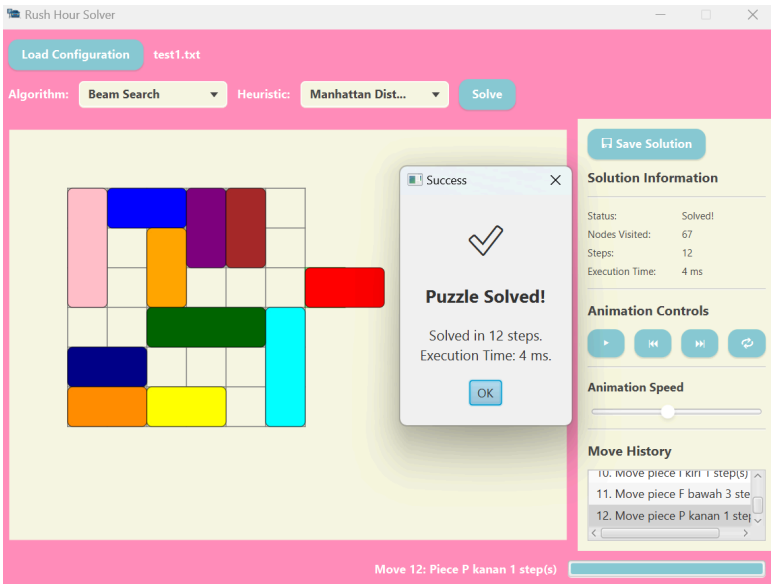
Output: Algoritma A* dengan Heuristic Manhattan Distance



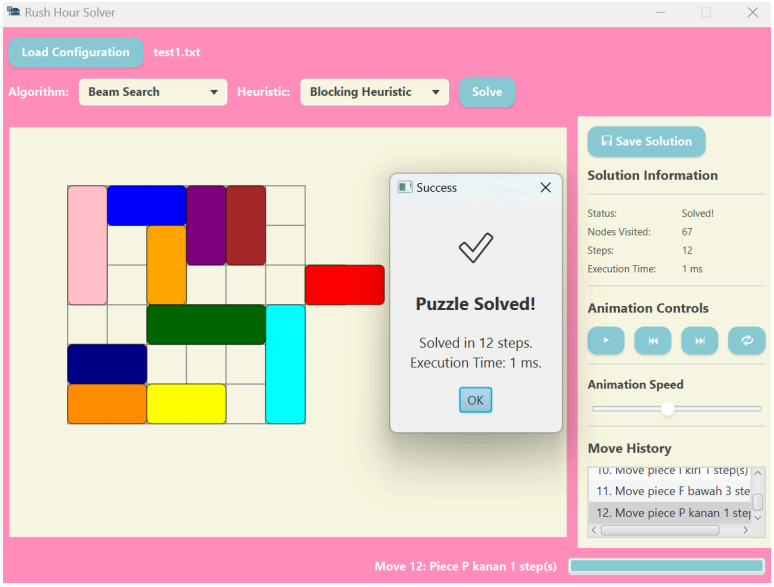
Output: Algoritma Greedy A* dengan Blocking Heuristic



Output: Algoritma Beam Search dengan Heuristic Manhattan Distance



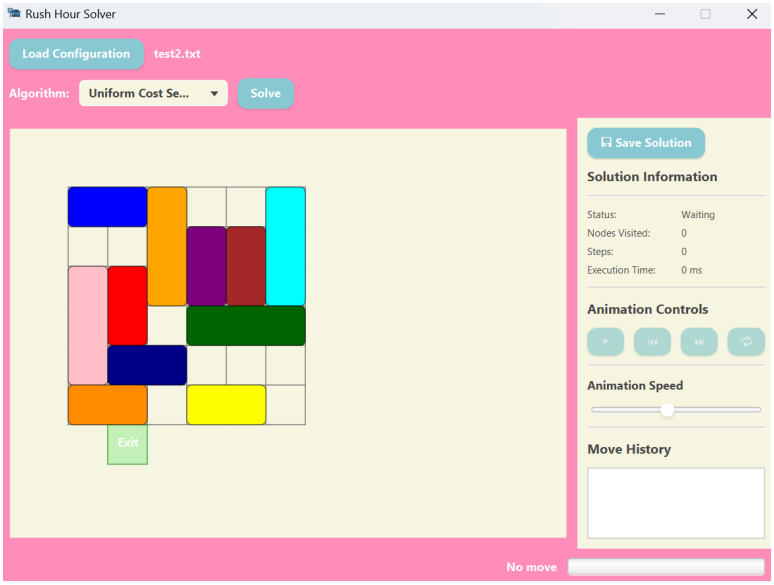
Output: Algoritma Beam Search dengan dengan Blocking Heuristic



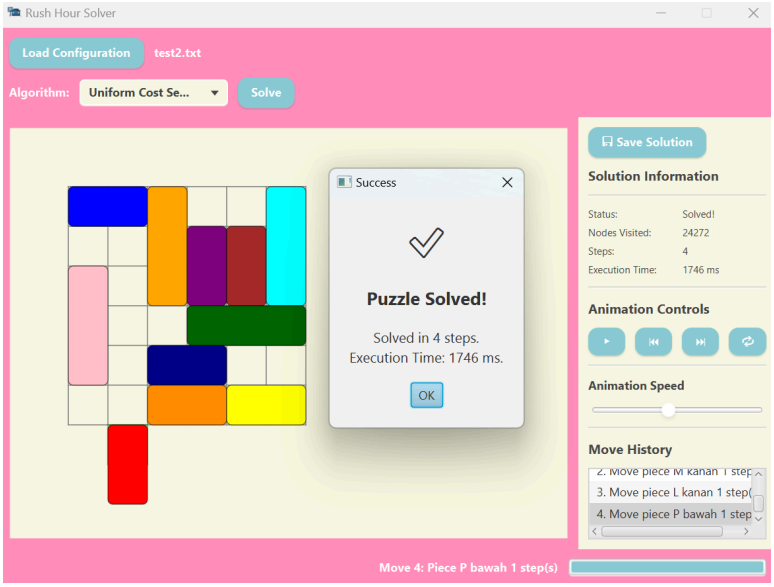
4.1.2 Test Case 2

Kondisi: Pengecekan kondisi bila *exit gate* berada di sebelah bawah.

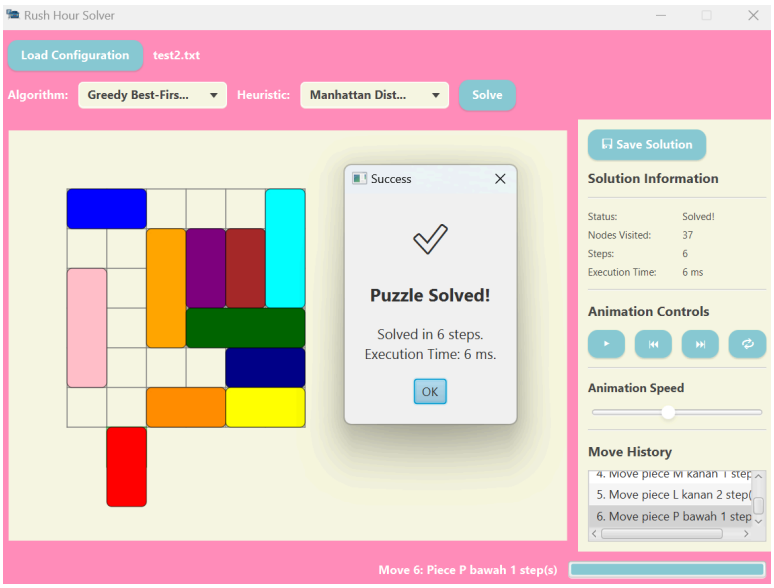
Input



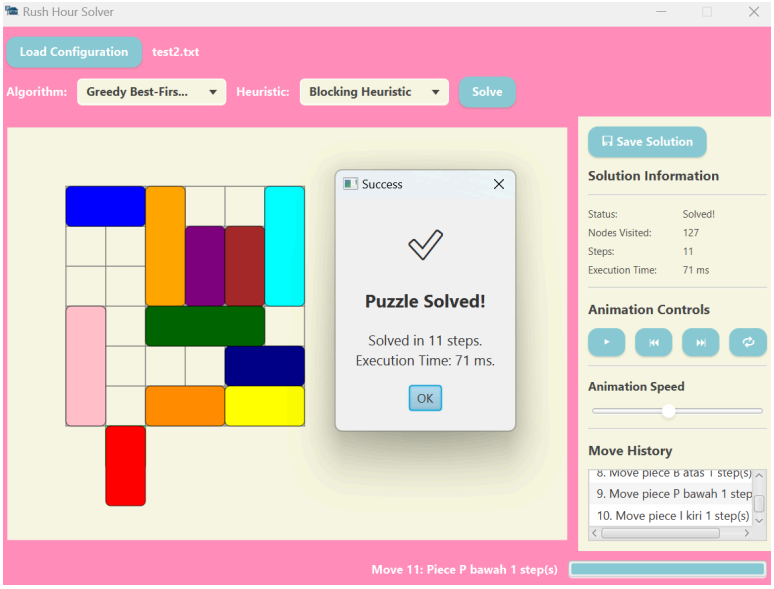
Output: Algoritma UCS



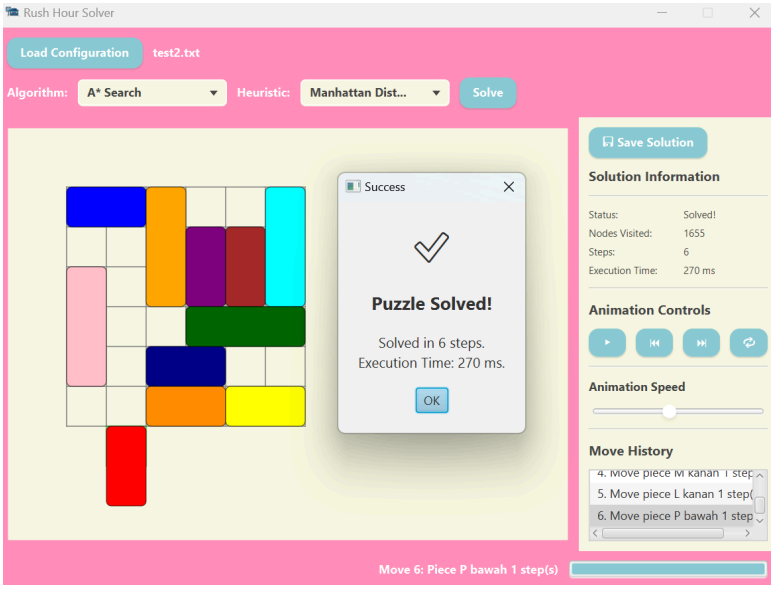
Output: Algoritma Greedy BFS dengan Heuristic Manhattan Distance



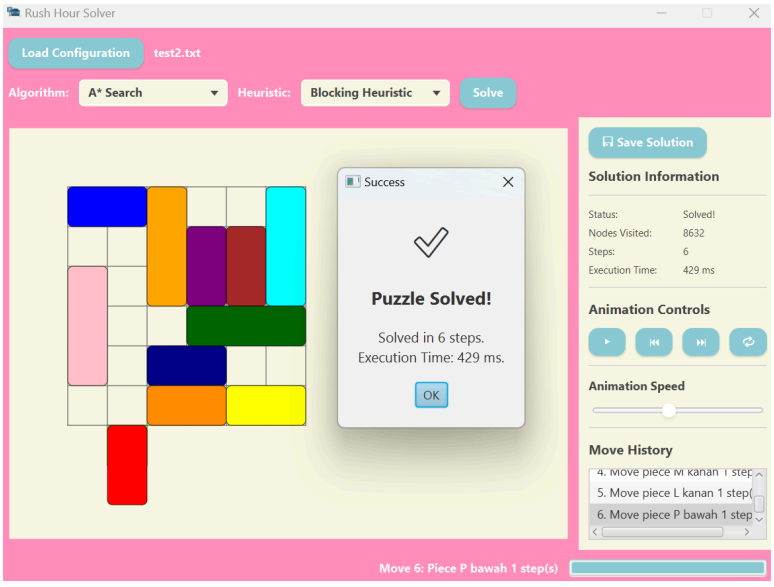
Output: Algoritma Greedy BFS dengan Blocking Heuristic



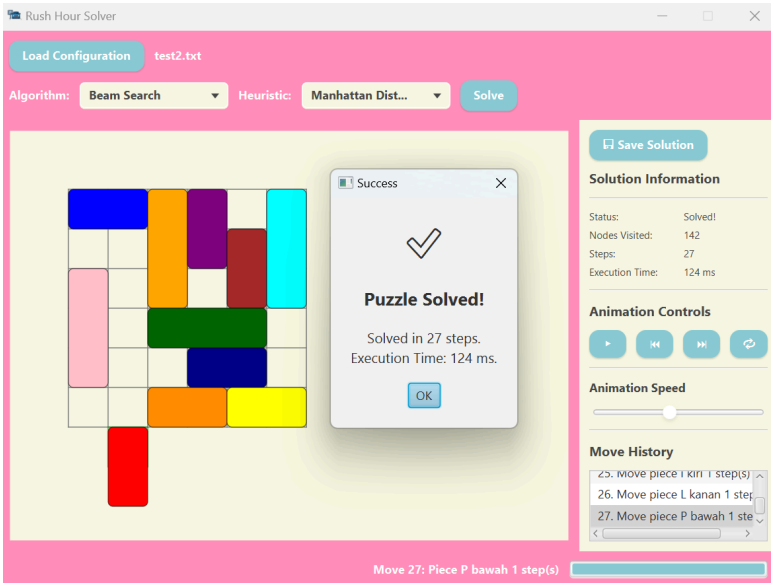
Output: Algoritma A* dengan Heuristic Manhattan Distance



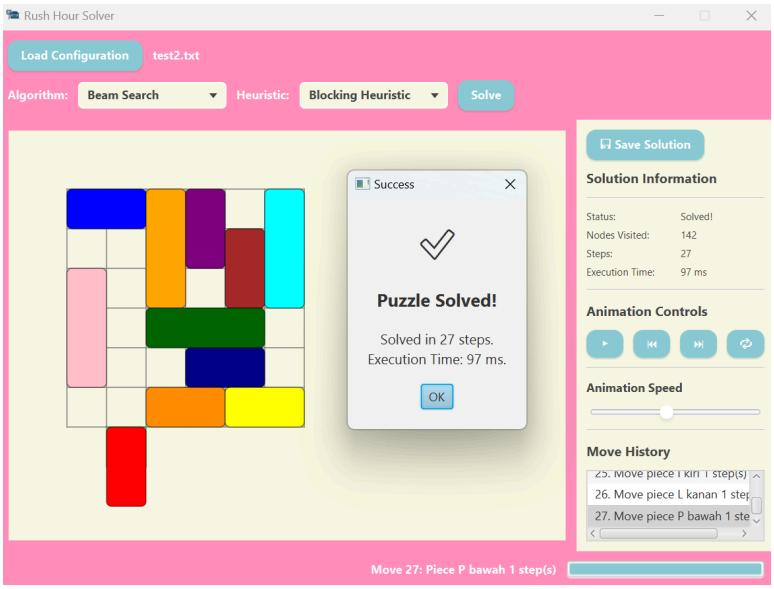
Output: Algoritma Greedy A* dengan Blocking Heuristic



Output: Algoritma Beam Search dengan Heuristic Manhattan Distance



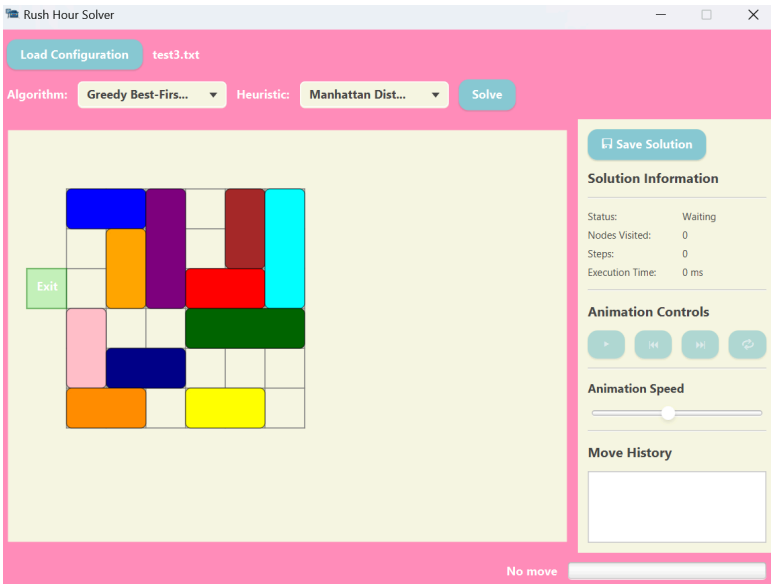
Output: Algoritma Beam Search dengan dengan Blocking Heuristic



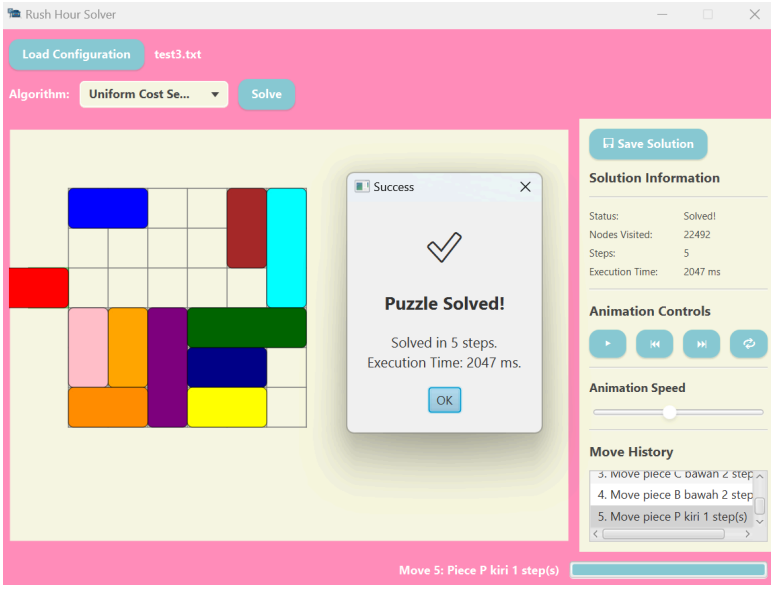
4.1.3 Test Case 3

Kondisi: Pengecekan kondisi bila *exit gate* berada di sebelah kiri.

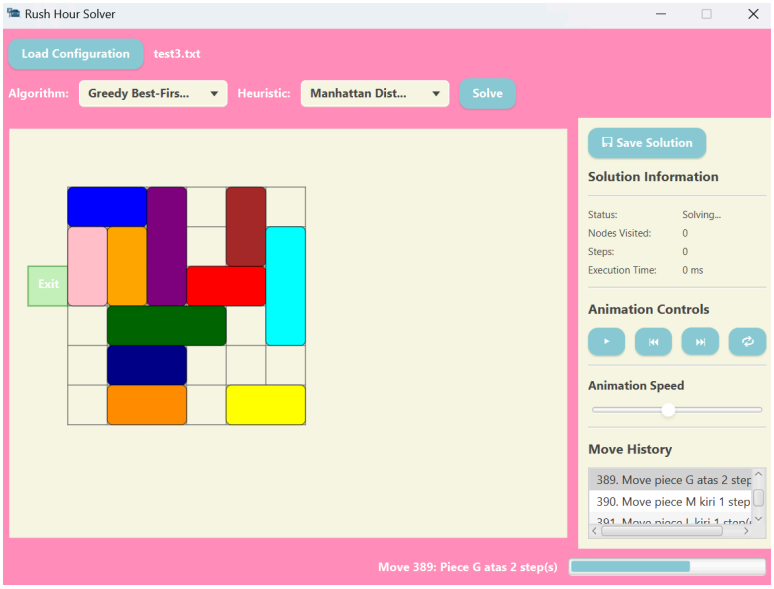
Input



Output: Algoritma UCS

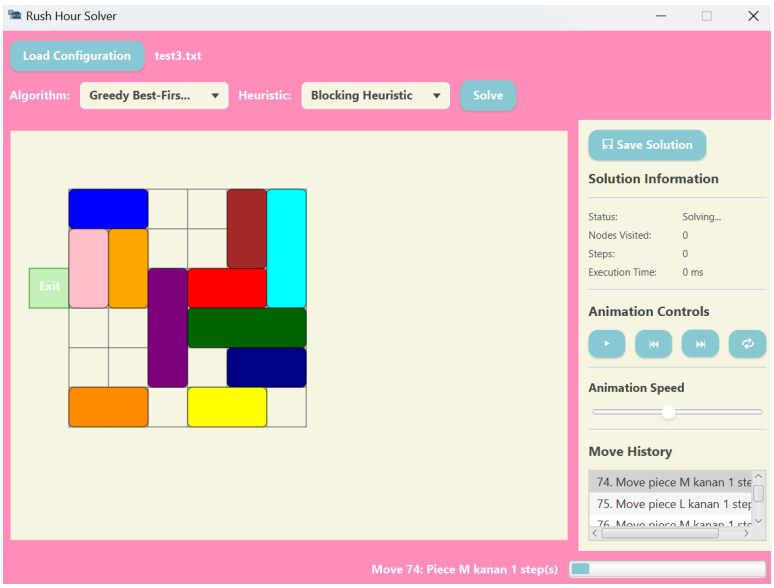


Output: Algoritma Greedy BFS dengan Heuristic Manhattan Distance



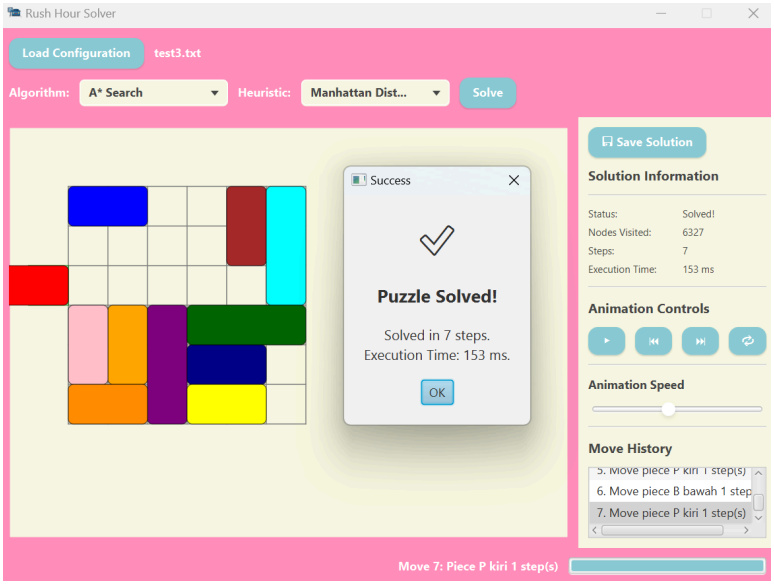
(langkah solusi masih terus dicek dan dibangun, tapi dijamin akan menghasilkan solusi)

Output: Algoritma Greedy BFS dengan Blocking Heuristic

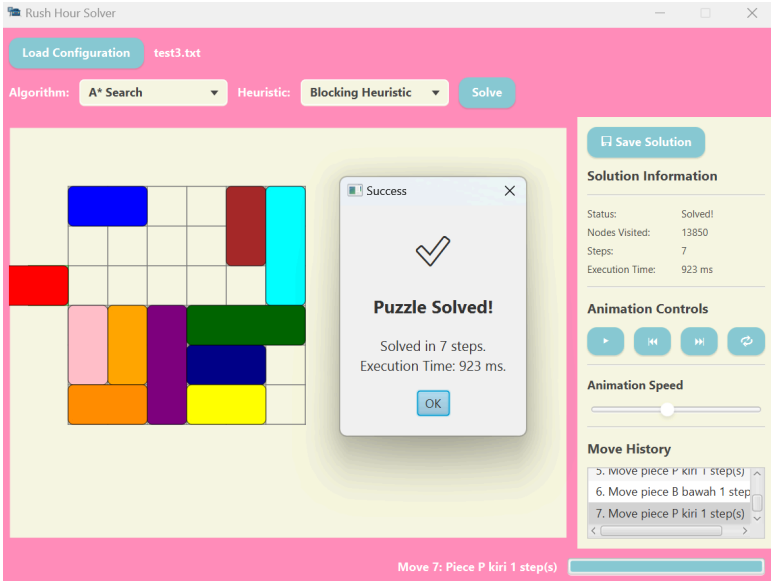


(langkah solusi masih terus dicek dan dibangun, tapi dijamin akan menghasilkan solusi)

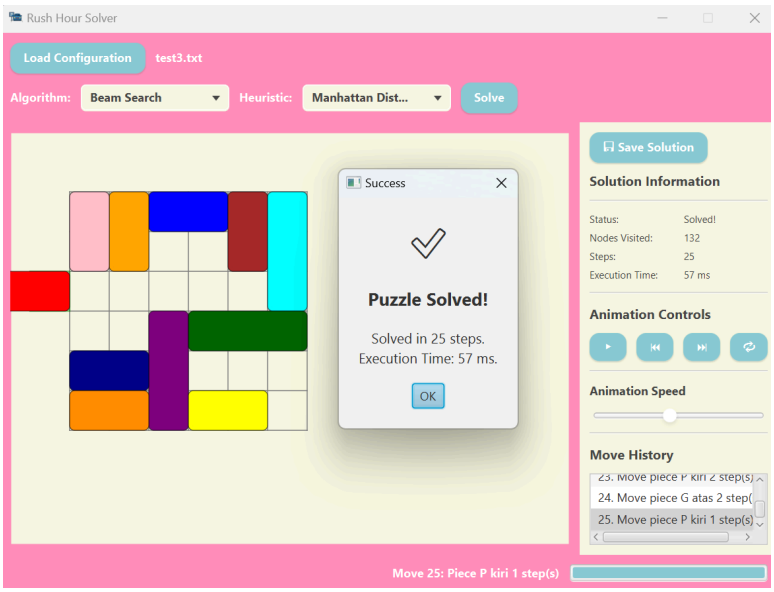
Output: Algoritma A* dengan Heuristic Manhattan Distance



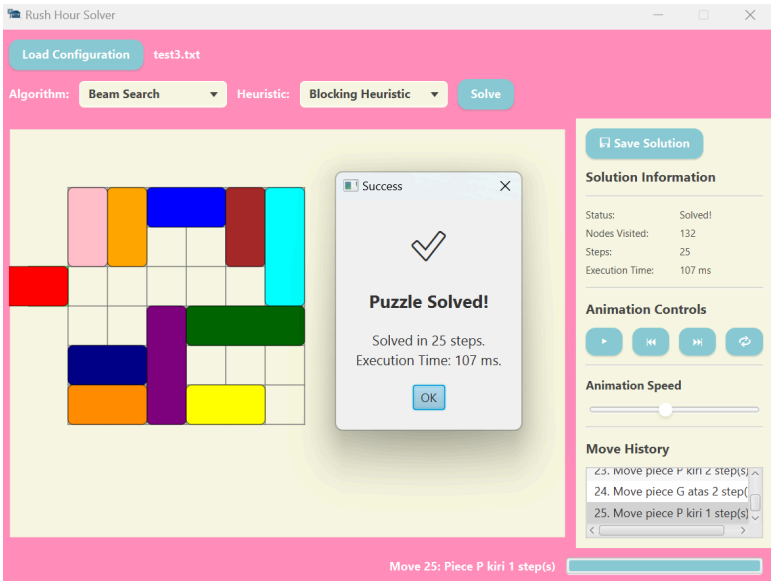
Output: Algoritma Greedy A* dengan Blocking Heuristic



Output: Algoritma Beam Search dengan Heuristic Manhattan Distance



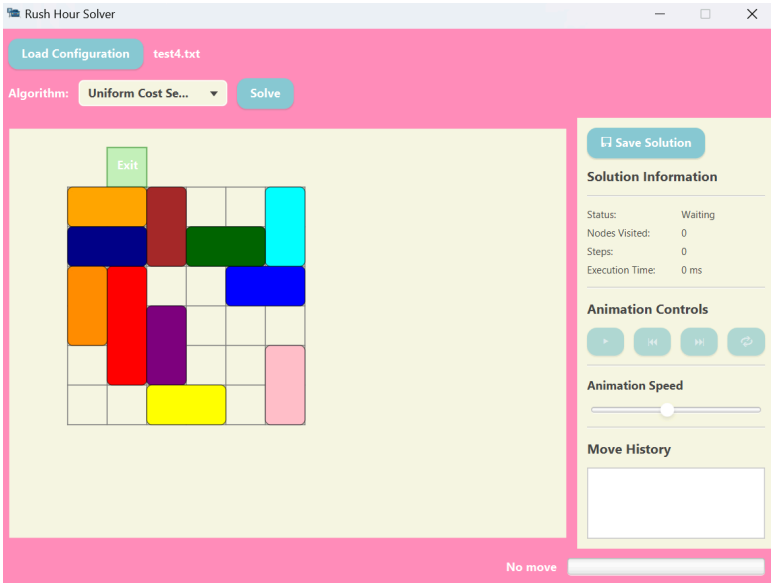
Output: Algoritma Beam Search dengan dengan Blocking Heuristic



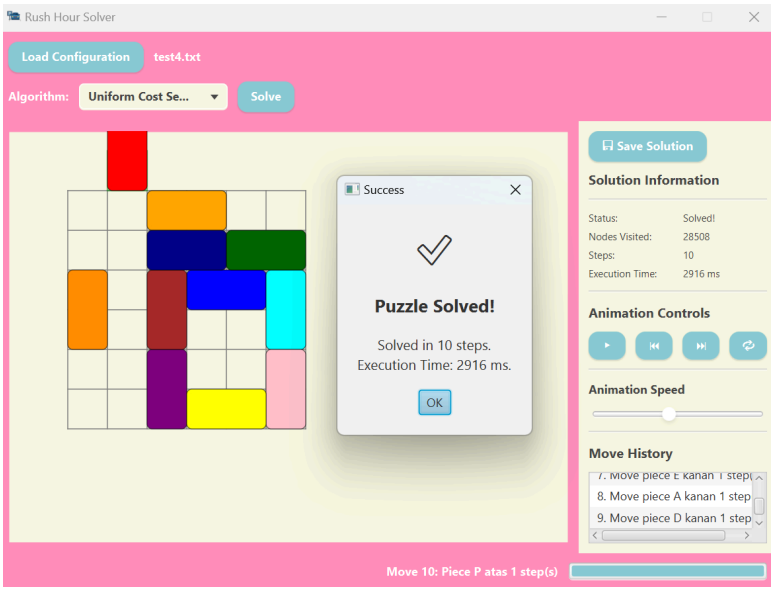
4.1.4 Test Case 4

Kondisi: Pengecekan kondisi bila *exit gate* berada di sebelah atas.

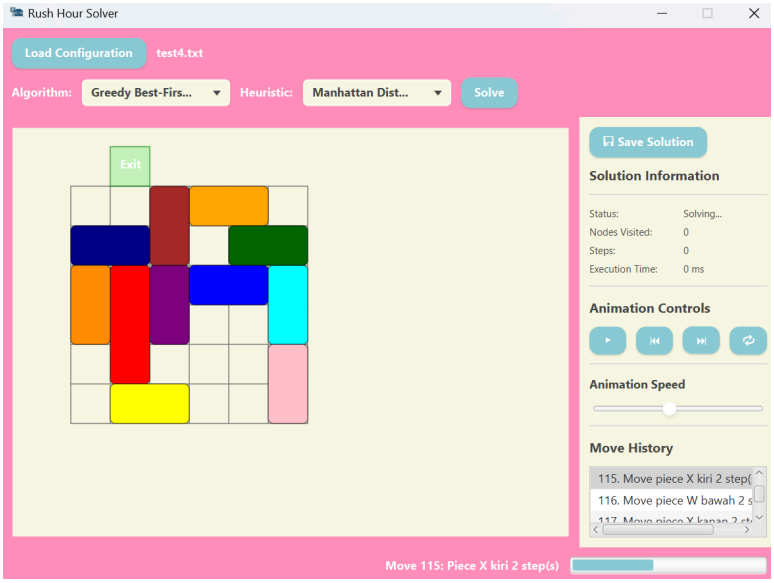
Input



Output: Algoritma UCS

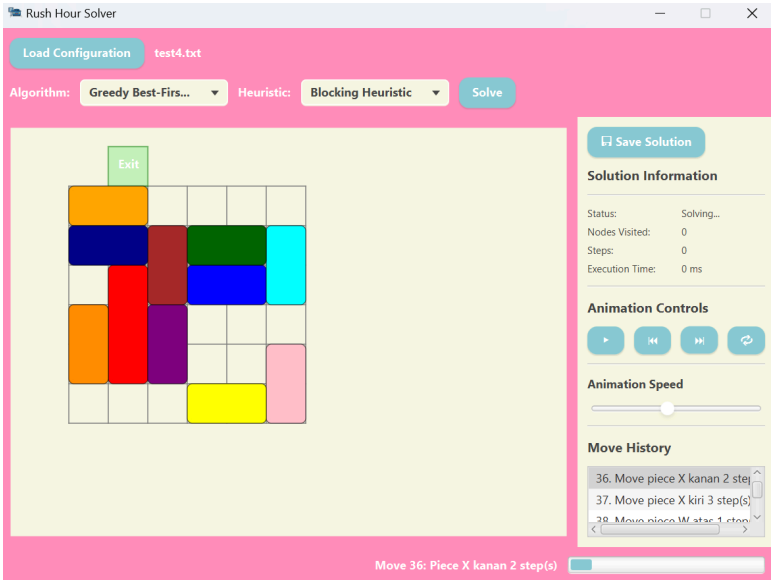


Output: Algoritma Greedy BFS dengan Heuristic Manhattan Distance



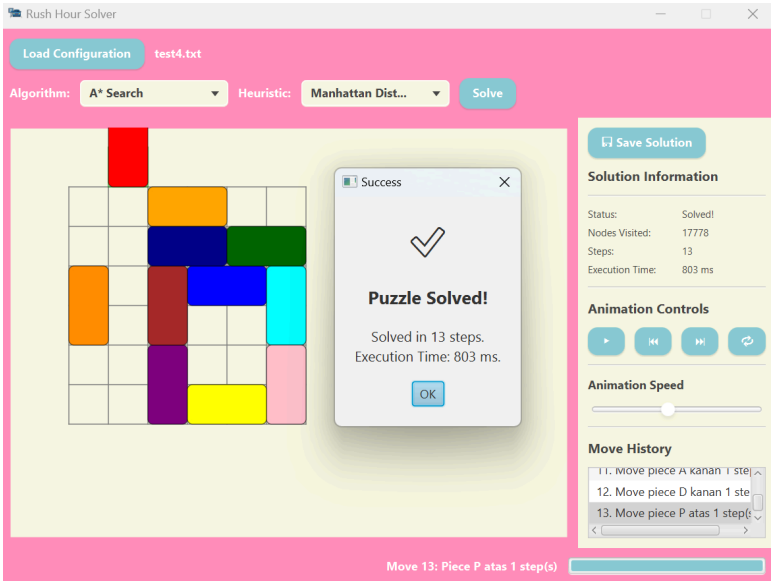
(langkah solusi masih terus dicek dan dibangun, tapi dijamin akan menghasilkan solusi)

Output: Algoritma Greedy BFS dengan Blocking Heuristic

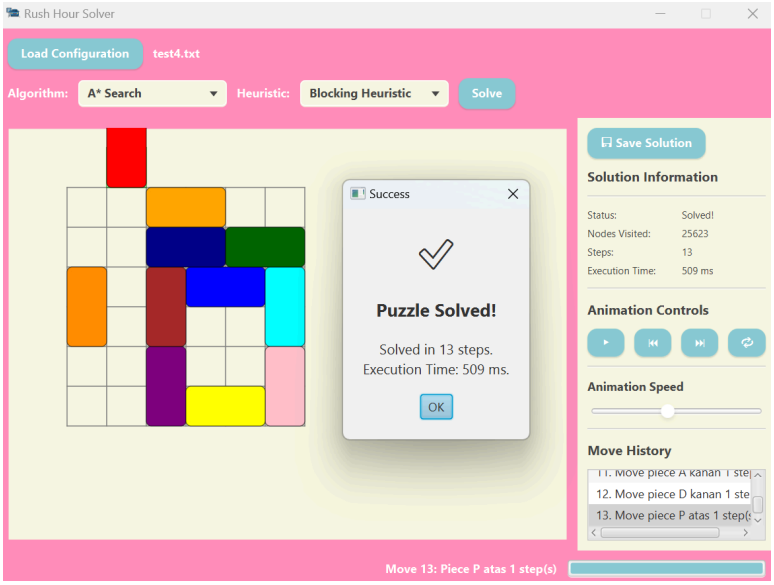


(langkah solusi masih terus dicek dan dibangun, tapi dijamin akan menghasilkan solusi)

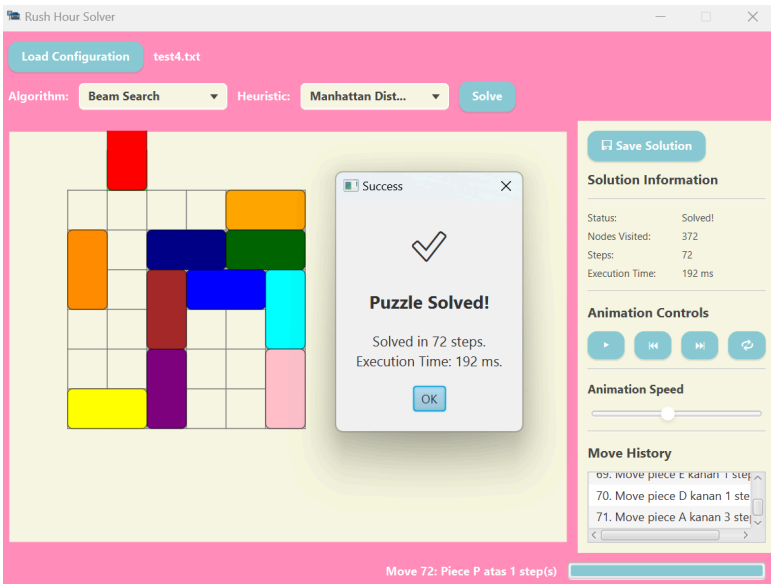
Output: Algoritma A* dengan Heuristic Manhattan Distance



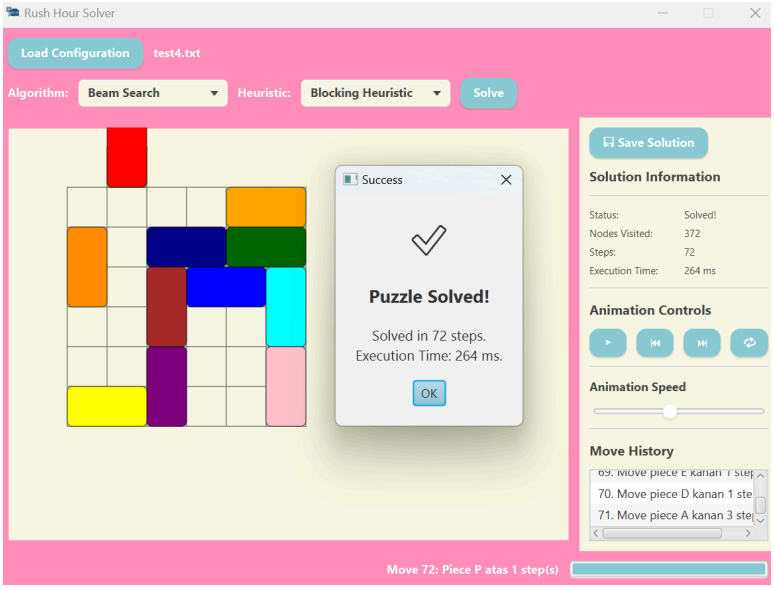
Output: Algoritma Greedy A* dengan Blocking Heuristic



Output: Algoritma Beam Search dengan Heuristic Manhattan Distance



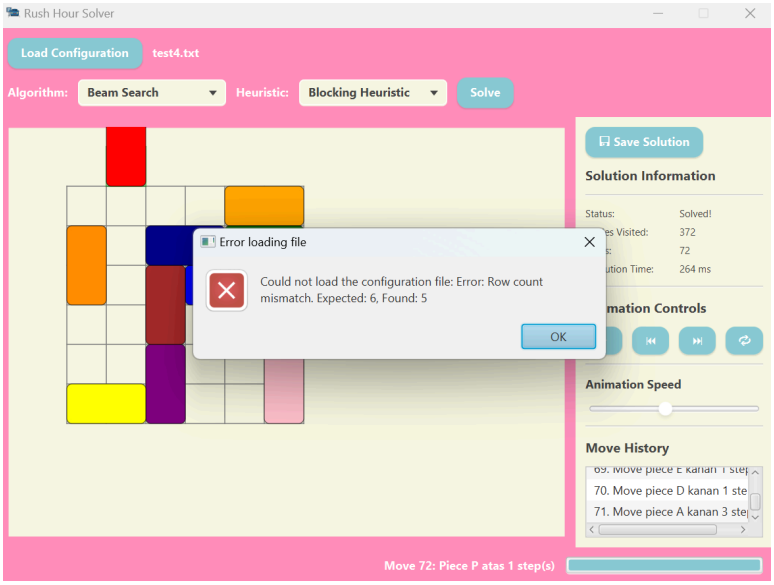
Output: Algoritma Beam Search dengan dengan Blocking Heuristic



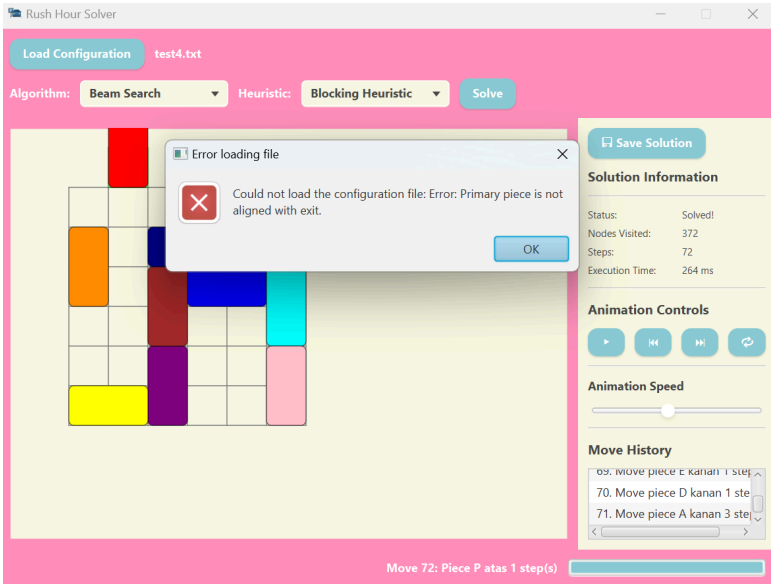
4.1.5 Test Case 5

Kondisi: Pengecekan kondisi bila *input board* tidak valid.

Output

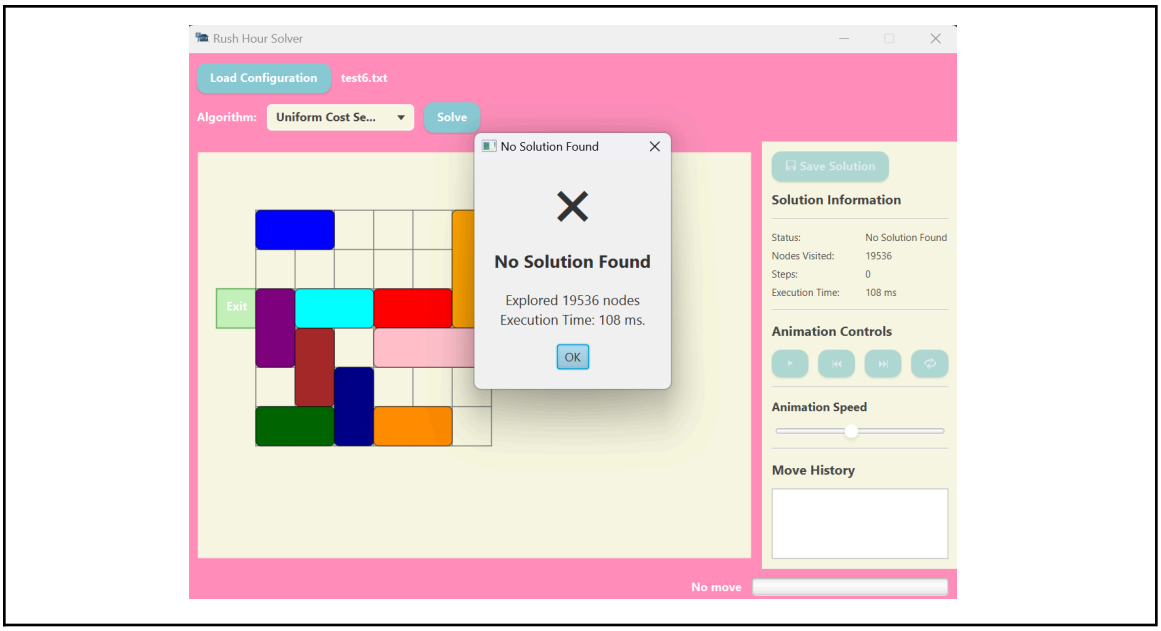


4.1.6 Test Case 6

Kondisi: Pengecekan kondisi bila *exit gate* tidak valid (tidak sejajar *primary car*)
Output


4.1.7 Test Case 7

Kondisi: Pengecekan kondisi bila solusi tidak ditemukan
Output



4.2 Analisis Pengujian

Berdasarkan pengujian yang telah dilakukan terhadap berbagai algoritma pencarian jalur (*pathfinding*), dapat disimpulkan bahwa algoritma A* merupakan salah satu algoritma paling optimal dalam menemukan solusi menuju titik akhir. Berbeda dengan Uniform Cost Search (UCS) yang hanya mempertimbangkan biaya aktual (*g(n)*) tanpa memperhatikan estimasi ke tujuan, A* mengombinasikan keunggulan UCS dan Greedy Best First Search (GBFS) dengan mempertimbangkan dua faktor utama: biaya aktual (*g(n)*) dan estimasi biaya ke tujuan (*h(n)*). Kombinasi ini memungkinkan A* untuk memilih jalur yang tidak hanya murah secara biaya saat ini, tetapi juga menjanjikan secara estimasi ke depan.

Dalam pengujian, algoritma GBFS beberapa kali gagal menemukan solusi atau membutuhkan waktu sangat lama hingga antrian prioritasnya habis. Hal ini terjadi karena GBFS hanya berfokus pada nilai estimasi heuristik (Manhattan Distance atau Blocking Heuristic), tanpa mempertimbangkan biaya aktual. Akibatnya, eksplorasi simpul bisa menjadi tidak efisien dan berulang.

Sementara itu, Algoritma Beam Search bekerja serupa dengan GBFS, namun dengan pembatasan jumlah simpul yang dieksplorasi pada setiap tingkat pencarian. Dalam implementasi kami, nilai *beam width* ditetapkan sebesar k = 5, sehingga hanya lima simpul dengan estimasi terbaik yang diperluas di setiap iterasi. Pendekatan ini secara signifikan mengurangi kompleksitas waktu dan ruang, meskipun dengan risiko melewatkan solusi optimal jika simpul yang menjanjikan tidak termasuk dalam lima teratas.

Dari segi efisiensi waktu, A* secara umum lebih cepat dibandingkan UCS karena mampu memproses lebih sedikit simpul. UCS, dengan hanya memperhatikan *g(n)*, cenderung menjelajahi banyak jalur yang sebenarnya tidak efektif menuju solusi.

Dalam kasus terburuk, kompleksitas waktu untuk UCS dan A* adalah $O(b^m)$, di mana *b* adalah *branching factor* (jumlah rata-rata cabang dari setiap simpul), dan *m* adalah kedalaman solusi optimal. Untuk Beam Search, kompleksitas waktu berada pada $O(k^d)$, dengan *k* adalah lebar beam (jumlah simpul yang diperluas di setiap tingkat).

BAB V
IMPLEMENTASI BONUS

5.1 Algoritma *Beam Search*

Beam Search adalah algoritma pencarian berbasis heuristik yang menyerupai *Breadth-First Search* (BFS), tetapi dengan penghematan memori dan waktu. Berbeda dengan BFS yang memperluas semua simpul pada setiap level, Beam Search hanya memperluas k simpul terbaik berdasarkan nilai heuristik. Parameter ini dikenal sebagai *beam width*.

Dalam penerapannya, kami menetapkan *beam width* (k) sebesar 5, agar tidak terlalu kecil sehingga berisiko melewatkan solusi optimal, namun juga tidak terlalu besar agar tetap efisien dalam eksekusi. Program akan berhenti jika solusi tidak ditemukan, yaitu isi *priority queue* telah kosong.

Secara keseluruhan, pendekatan *Beam Search* mirip dengan *Greedy Best-First Search* (GBFS), namun dengan jumlah simpul yang dieksekusi per level yang lebih terbatas, sehingga menjadikannya lebih ringan dan cepat, meskipun tetap memiliki risiko melewatkan solusi optimal.

Berikut source code program untuk Algoritma *Beam Search*.

```java
package algorithm;

import heuristic.Heuristic;
import java.util.*;
import model.Board;
import model.State;
import util.Constants;

public class BeamSearch implements Algorithm {

    private Heuristic heuristic;
    private final int beamWidth = 5;

    @Override
    public SolutionPath findSolution(Board initialBoard) {
        long startTime = System.currentTimeMillis();
        int nodesVisited = 0;

        State initialState = new State(initialBoard);
        initialState.setHeuristicValue(heuristic.calculate(initialBoard));

        PriorityQueue<State> frontier = new PriorityQueue<>(Comparator.comparing(State::getHeuristicValue));

        frontier.add(initialState);

        Set<String> visited = new HashSet<>();
        visited.add(initialBoard.toString());

        while (!frontier.isEmpty()) {
            List<State> current = new ArrayList<>(); // Ambil sesuai beamWitdth terbaik
            int count = 0;
            while (!frontier.isEmpty() && count < beamWidth) {
                current.add(frontier.poll());
                count++;
            }
```

```
            List<State> nextCandidates = new ArrayList<>();

            for (State currentState : current) {
                nodesVisited++;
                if (currentState.getBoard().isSolved()) {
                    long endTime = System.currentTimeMillis();
                    List<State> path = currentState.getSolutionPath();
                    return new SolutionPath(path, nodesVisited, endTime - startTime);
                }

                List<State> childStates = currentState.generateChildStates();

                for (State childState : childStates) {
                    String boardKey = childState.getBoard().toString();
                    if (!visited.contains(boardKey)) {
                        childState.setHeuristicValue(heuristic.calculate(childState.getBoard()));
                        nextCandidates.add(childState);
                        visited.add(boardKey);
                    }
                }
            }

            nextCandidates.sort(Comparator.comparing(State::getHeuristicValue));

            frontier.clear();
            for (int i = 0; i < Math.min(beamWidth * 2, nextCandidates.size()); i++) {
                frontier.add(nextCandidates.get(i));
            }
        }

        long endTime = System.currentTimeMillis();
        return new SolutionPath(nodesVisited, endTime - startTime);
    }

    @Override
    public void setHeuristic(Heuristic heuristic) {
        this.heuristic = heuristic;
    }

    @Override
    public String getName() {
        return Constants.BEAM;
    }
}
```

5.2 Heuristic: *Manhattan Distance*

Manhattan Distance menghitung jumlah langkah minimum secara horizontal dan vertikal untuk mencapai titik tujuan dari titik asal. Heuristik ini selalu *admissible* karena tidak pernah melebih-lebihkan biaya sebenarnya, yakni jarak sesungguhnya yang diperlukan untuk mencapai tujuan.

Heuristik ini sangat tepat diterapkan dalam Rush Hour Solver, karena pergerakan mobil dalam permainan hanya diperbolehkan secara horizontal atau vertikal, tanpa gerakan diagonal. Oleh karena itu, hasil perhitungan *Manhattan Distance* mencerminkan jarak terpendek yang realistis di *board* permainan.

Dalam implementasinya, perhitungan dilakukan dengan menentukan posisi dari *primary piece car* (mobil utama) dan pintu keluar. Selanjutnya, estimasi jarak dihitung dengan menjumlahkan selisih absolut posisi baris dan kolom antara keduanya (misalnya: $|x_1 - x_2| + |y_1 - y_2|$).

Berikut source code program untuk Heuristic: *Manhattan Distance*.

```java
package heuristic;

import model.*;
import util.Constants;

public class ManhattanDistance implements Heuristic {
    @Override
    public int calculate(Board board) {
        if (board.getExitPosition() == null || board.getPrimaryPieceId() == 0
            || !board.getPieces().containsKey(board.getPrimaryPieceId())) {
            return Integer.MAX_VALUE; // cek kondisi, heuristic gabisa dihitung
        }

        Piece primary = board.getPrimaryPiece();
        Position anchor = primary.getAnchor();

        int anchorRow = anchor.getRow();
        int anchorCol = anchor.getCol();

        Position exitPosition = board.getExitPosition();
        int exitRow = exitPosition.getRow();
        int exitCol = exitPosition.getCol();

        int rows = board.getRows();
        int cols = board.getCols();

        if (exitRow == -1) { // atas
            return Math.abs(anchorCol - exitCol) + (anchorRow + 1);
        } else if (exitRow == rows) { // bawah
            return Math.abs(anchorCol - exitCol) + (rows - anchorRow);
        } else if (exitCol == -1) { // kiri
            return Math.abs(anchorRow - exitRow) + (anchorCol + 1);
        } else if (exitCol == cols) { // kanan
            return Math.abs(anchorRow - exitRow) + (cols - anchorCol);
        } else { // jaga-jaga aja
            return Math.abs(anchorRow - exitRow) + Math.abs(anchorCol - exitCol);
        }
    }

    @Override
    public String getName() {
        return Constants.MANHATTAN_HEURISTIC;
    }
}
```

5.3 Heuristic: *Blocking*

Blocking Heuristic menghitung jumlah mobil yang menghalangi jalur langsung antara *primary piece car* (mobil utama) dan pintu keluar. Heuristik ini mengasumsikan bahwa setiap mobil penghalang harus dipindahkan setidaknya sekali untuk membebaskan jalur menuju tujuan, sehingga memberikan estimasi minimum terhadap upaya yang dibutuhkan untuk menyelesaikan permainan.

Heuristik ini sangat sesuai diterapkan dalam Rush Hour Solver, karena kemenangan hanya dapat dicapai jika mobil utama dapat bergerak lurus menuju pintu keluar tanpa terhalang. Oleh karena itu, menghitung jumlah mobil yang berada tepat di jalur tersebut menjadi indikator yang relevan terhadap tingkat kesulitan situasi saat ini.

Dalam implementasinya, proses perhitungan dilakukan dengan menentukan arah keluar dari mobil utama berdasarkan lokasi pintu keluar di papan. Setelah itu, sel-sel di jalur tersebut ditelusuri satu per satu. Setiap sel yang berisi mobil selain mobil utama akan dihitung sebagai penghalang. Hasil akhir dari heuristik ini adalah total jumlah mobil yang secara langsung menghambat jalan keluar mobil utama.

Berikut source code program untuk Heuristic: *Blocking*.

```
package heuristic;

import model.Board;
import model.Piece;
import model.Position;
import util.Constants;

public class BlockingHeuristic implements Heuristic {

    @Override
    public int calculate(Board board) { // kalau ga valid
        if (board.getExitPosition() == null || board.getPrimaryPieceId() == 0
            || !board.getPieces().containsKey(board.getPrimaryPieceId())) {
            return Integer.MAX_VALUE;
        }

        Piece primary = board.getPrimaryPiece();
        Position anchor = primary.getAnchor();
        int row = anchor.getRow();
        int col = anchor.getCol();
        int length = primary.getSize();

        Position exit = board.getExitPosition();
        int exitRow = exit.getRow();
        int exitCol = exit.getCol();

        int blockingCount = 0;

        // Arah exit gate
        int dirRow = 0, dirCol = 0;

        if (exitRow < 0)
            dirRow = -1; // atas
        else if (exitRow >= board.getRows())
            dirRow = 1; // bawah
        else if (exitCol < 0)
            dirCol = -1; // kiri
        else if (exitCol >= board.getCols())
            dirCol = 1; // kanan
        else
            return Integer.MAX_VALUE; // ga valid

        // posisi awal untuk cek
        int startRow = row;
        int startCol = col;
```

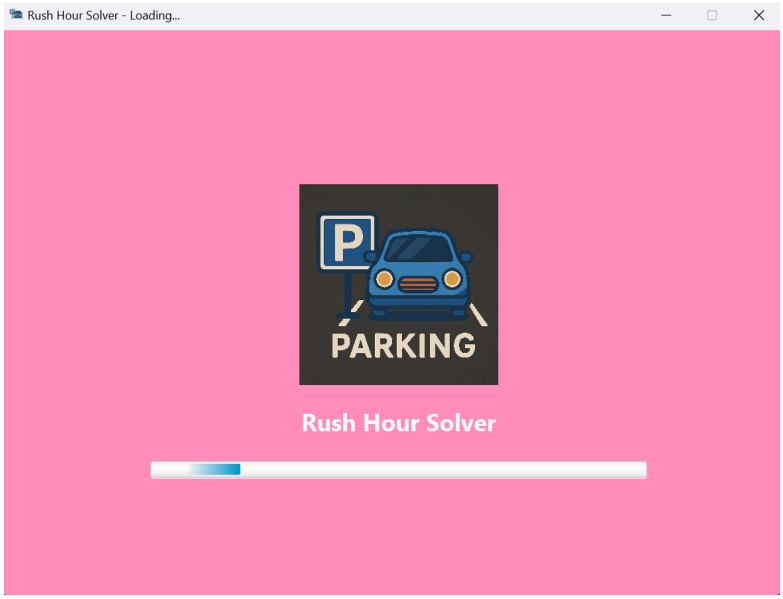
```
        if (!primary.isHorizontal()) {
            if (dirRow > 0)
                startRow = row + length - 1;
        } else {
            if (dirCol > 0)
                startCol = col + length - 1;
        }

        int currentRow = startRow + dirRow;
        int currentCol = startCol + dirCol;

        while (currentRow >= 0 && currentRow < board.getRows() &&
                currentCol >= 0 && currentCol < board.getCols()) {

            int cellValue = board.getBoardArray()[currentRow][currentCol];
            if (cellValue != 0 && cellValue != board.getPrimaryPieceId()) { // ada mobil lain
                blockingCount++;
            }

            currentRow += dirRow;
            currentCol += dirCol;
        }

        return blockingCount;
    }

    @Override
    public String getName() {
        return Constants.BLOCKING_HEURISTIC;
    }
}
```

5.4 GUI

Untuk mendukung kenyamanan dan kemudahan penggunaan, kami mengimplementasikan antarmuka pengguna grafis (GUI) menggunakan JavaFX. JavaFX dipilih karena menyediakan tools dan library yang cocok untuk membangun antarmuka modern dan terstruktur.

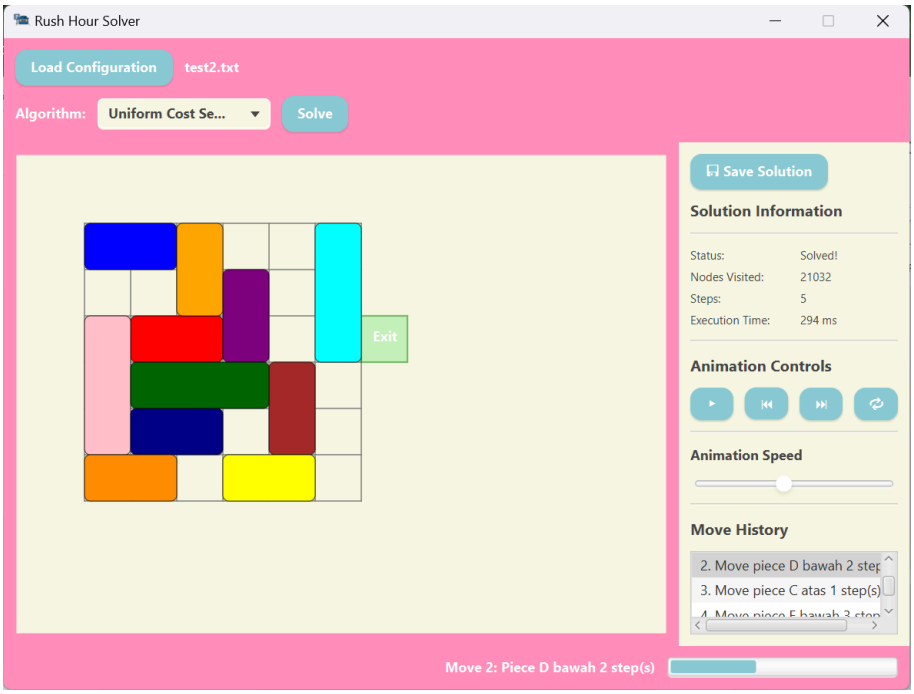
GUI terdiri atas dua tampilan utama, yaitu *Loading Screen* dan *Main Menu*.

Loading Screen



Tampilan awal aplikasi menggunakan latar pink dengan logo bergambar mobil dan rambu parkir di tengah layar. Di bawah logo terdapat teks “Rush Hour Solver” serta progress bar horizontal yang menunjukkan proses pemuatan aplikasi sebelum masuk ke menu utama.

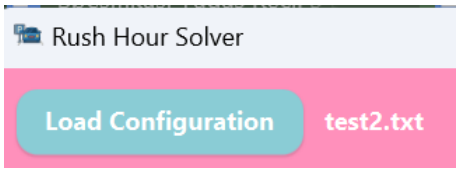
Main Menu



Tampilan utama terdiri dari tombol pemilihan file dan algoritma di bagian atas, papan permainan berbentuk grid di tengah, serta panel samping berisi kontrol dan informasi di sebelah kanan. Seluruh elemen disusun rapi dengan nuansa warna pastel yang konsisten.

Berikut fitur-fitur yang tersedia pada GUI kami.

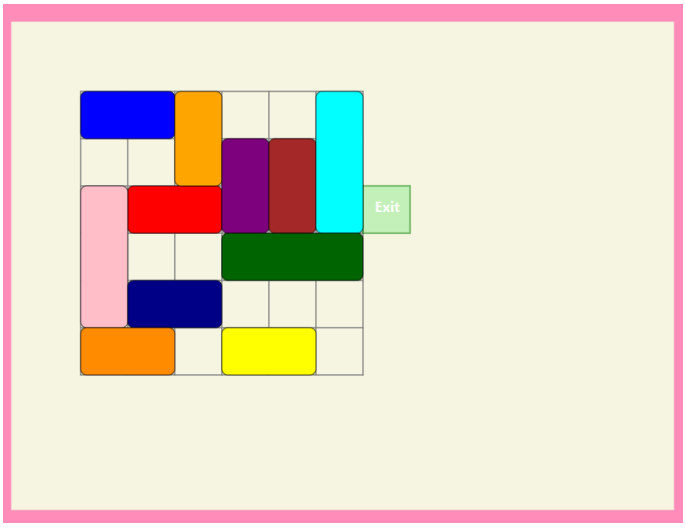
- ***File Configuration***


Pengguna dapat memilih file konfigurasi papan puzzle melalui tombol “Load Configuration”.

- **Pemilihan Algoritma dan Heuristik**

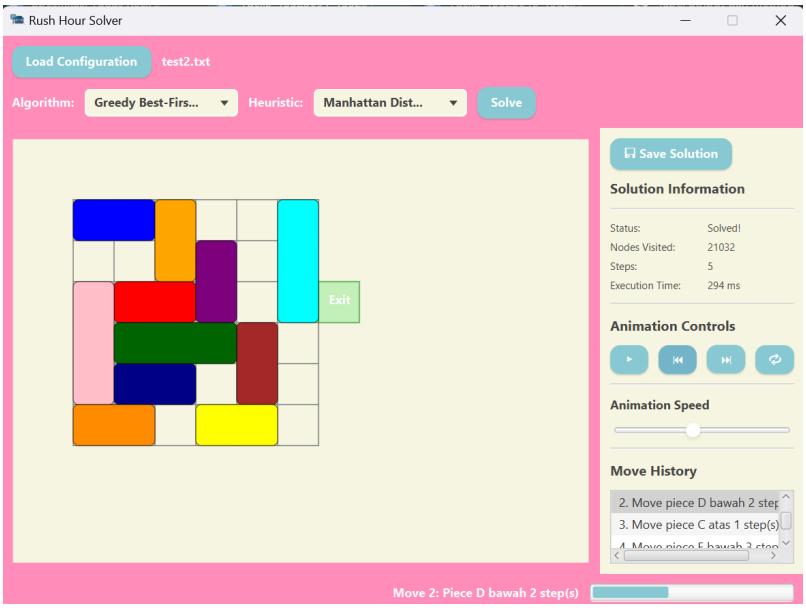

Tersedia pilihan algoritma Uniform Cost Search, Greedy Best First Search, A*, dan Beam Search. Jika algoritma memerlukan heuristik, dropdown heuristik akan muncul secara otomatis. Tersedia dua pilihan heuristik, yaitu Manhattan Distance dan Blocking Heuristic

- **Visualisasi Papan Puzzle**



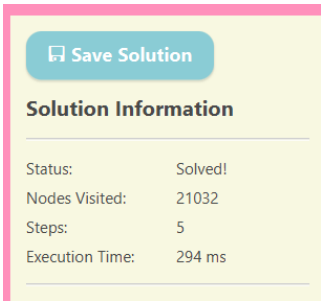
Papan permainan divisualisasikan dalam bentuk grid, dengan kendaraan ditampilkan berwarna-warni. Primary Piece atau kendaraan utama divisualisasikan dengan warna merah dan terdapat juga Exit Gate atau pintu keluar yang diimplementasikan dengan warna hijau bertuliskan “Exit”.

- **Solusi Animasi**



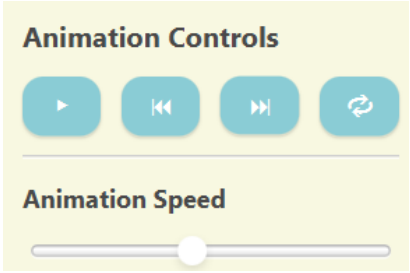
Setiap langkah solusi divisualisasikan secara animatif satu per satu untuk memudahkan pemahaman pengguna.

- **Informasi Solusi**



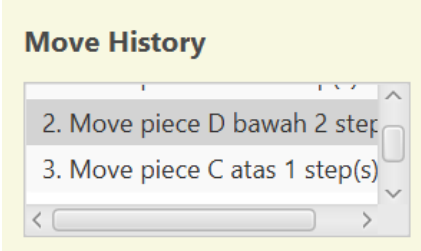
Panel samping menampilkan status solusi, jumlah simpul yang dikunjungi, langkah yang dibutuhkan, dan waktu eksekusi.

- **Kontrol Animasi**



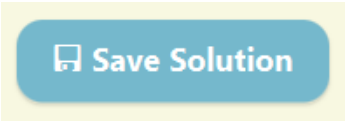
Pengguna dapat memutar, menjeda, mempercepat, mengatur ulang animasi, atau melihat animasi langkah demi langkah melalui tombol kontrol yang disediakan.

- **Riwayat Gerakan**



Seluruh langkah solusi dicatat dan ditampilkan dalam daftar riwayat yang dapat discroll.

- **Simpan Solusi**



Terdapat tombol “Save Solution” untuk menyimpan hasil solusi ke dalam file txt. Isi dari file solusi berupa informasi algoritma yang digunakan, jumlah langkah, jumlah simpul, waktu eksekusi, kondisi board awal, serta kondisi board langkah per langkah hingga solusi ditemukan.

Berikut source code program untuk GUI.

- **Class RushHourApp**

```java
package gui;

import javafx.animation.PauseTransition;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.util.Duration;

import java.io.IOException;

public class RushHourApp extends Application {

    @Override
    public void start(Stage primaryStage) {
        try {
            FXMLLoader loadingLoader = new FXMLLoader(getClass().getResource(name:"LoadingScreen.fxml"));
            Parent loadingRoot = loadingLoader.load();
            Scene loadingScene = new Scene(loadingRoot);

            primaryStage.setTitle("Rush Hour Solver - Loading...");
            primaryStage.getIcons()
                    .add(new javafx.scene.image.Image(getClass().getResourceAsStream(name:"/gui/assets/logo.png")));
            primaryStage.setScene(loadingScene);

            primaryStage.setWidth(800);
            primaryStage.setHeight(600);
            primaryStage.setResizable(false);
            primaryStage.show();

            PauseTransition delay = new PauseTransition(Duration.seconds(3));
            delay.setOnFinished(event -> {    The value of the lambda parameter event is not used
                try {
                    FXMLLoader mainLoader = new FXMLLoader(getClass().getResource(name:"MainView.fxml"));
                    Parent mainRoot = mainLoader.load();
                    Scene mainScene = new Scene(mainRoot);
                    mainScene.getStylesheets().add(getClass().getResource(name:"/gui/assets/style.css").toExternalForm());
                    primaryStage.setScene(mainScene);
                    primaryStage.setTitle("Rush Hour Solver");

                    primaryStage.setWidth(800);
                    primaryStage.setHeight(600);
                } catch (IOException ex) {
                    ex.printStackTrace();
                }
            });
            delay.play();

        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    Run | Debug
    public static void main(String[] args) {
        launch(args);
    }
}
```

- **Class LoadingController**

```java
package gui;

import javafx.animation.PauseTransition;
import javafx.fxml.FXML;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.control.ProgressBar;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.stage.Stage;
import javafx.util.Duration;

import java.io.IOException;

public class LoadingController {

    @FXML
    private ProgressBar progressBar;

    @FXML
    private ImageView carImage;

    @FXML
    public void initialize() {
        carImage.setImage(new Image(getClass().getResource(name:"/gui/assets/loading.png").toExternalForm()));

        PauseTransition delay = new PauseTransition(Duration.seconds(3));
        delay.setOnFinished(e -> {    The value of the lambda parameter e is not used
            try {
                Stage stage = (Stage) progressBar.getScene().getWindow();
                FXMLLoader loader = new FXMLLoader(getClass().getResource(name:"MainView.fxml"));
                Scene mainScene = new Scene(loader.load());
                stage.setScene(mainScene);
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        });
        delay.play();
    }
}
```

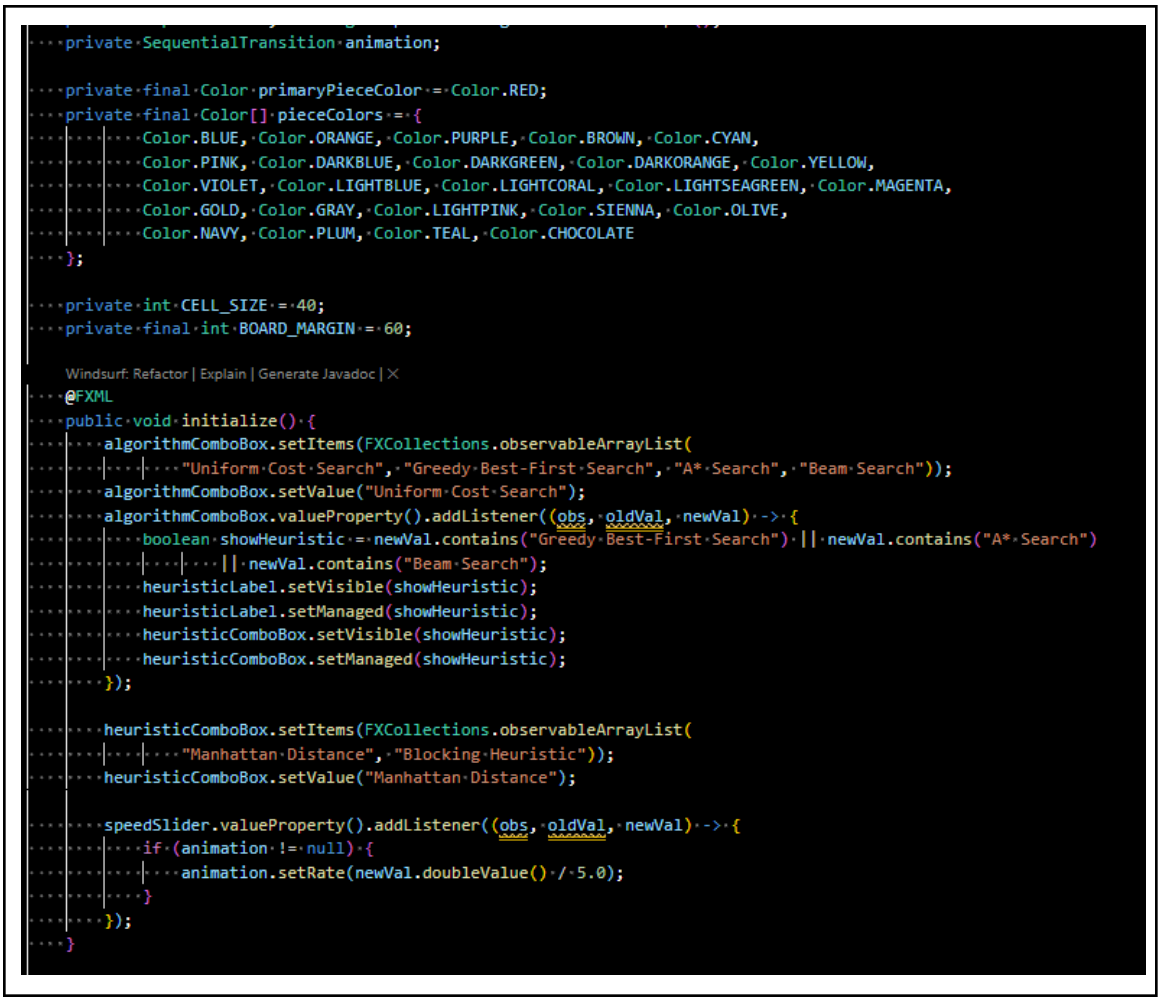
- **LoadingScreen.fxml**

```xml
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.ProgressBar?>
<?import javafx.scene.image.ImageView?>
<?import javafx.scene.layout.AnchorPane?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.control.Label?>
<?import javafx.geometry.Pos?>

<AnchorPane style="-fx-background-color: #FF90BB;" xmlns:fx="http://javafx.com/fxml"
    fx:controller="gui.LoadingController" prefWidth="800" prefHeight="600">

    <VBox alignment="CENTER" spacing="20" layoutX="0" layoutY="0" prefWidth="800" prefHeight="600">
        <ImageView fx:id="carImage" fitHeight="200" fitWidth="200" />
        <Label text="Rush Hour Solver"
            style="-fx-font-size: 24px; -fx-text-fill: white; -fx-font-weight: bold;" />
        <ProgressBar fx:id="progressBar" prefWidth="500" />
    </VBox>
</AnchorPane>
```

- **Class MainController**

```java
package gui;

import javafx.animation.*;
import javafx.application.Platform;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.fxml.FXML;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.Pane;
import javafx.scene.layout.StackPane;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Line;
import javafx.scene.shape.Rectangle;
import javafx.stage.FileChooser;
import javafx.stage.Modality;
import javafx.stage.Stage;
import javafx.util.Duration;
import model.*;

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

import algorithm.*;
import heuristic.*;
import util.*;

Windsurf: Refactor | Explain
public class MainController {

    @FXML
    private Button loadButton;
    @FXML
    private Label filePathLabel;
    @FXML
    private ComboBox<String> algorithmComboBox;
    @FXML
    private Label heuristicLabel;
```

```
    @FXML
    private ComboBox<String> heuristicComboBox;
    @FXML
    private Button solveButton;
    @FXML
    private StackPane boardContainer;
    @FXML
    private Pane boardPane;
    @FXML
    private Label statusLabel;
    @FXML
    private Label nodesVisitedLabel;
    @FXML
    private Label stepsLabel;
    @FXML
    private Label executionTimeLabel;
    @FXML
    private Button playButton;
    @FXML
    private Button resetButton;
    @FXML
    private Slider speedSlider;
    @FXML
    private ListView<String> moveHistoryListView;
    @FXML
    private Label currentMoveLabel;
    @FXML
    private ProgressBar solutionProgress;
    @FXML
    private Button stepBackButton;
    @FXML
    private Button stepForwardButton;
    @FXML
    private Button saveSolutionButton;

    private Board initialBoard;
    private SolutionPath solution;
    private List<State> statePath;
    private int currentStateIndex = 0;
    private Map<Character, Rectangle> pieceRectangles = new HashMap<>();
```



```
    private SequentialTransition animation;

    private final Color primaryPieceColor = Color.RED;
    private final Color[] pieceColors = {
        Color.BLUE, Color.ORANGE, Color.PURPLE, Color.BROWN, Color.CYAN,
        Color.PINK, Color.DARKBLUE, Color.DARKGREEN, Color.DARKORANGE, Color.YELLOW,
        Color.VIOLET, Color.LIGHTBLUE, Color.LIGHTCORAL, Color.LIGHTSEAGREEN, Color.MAGENTA,
        Color.GOLD, Color.GRAY, Color.LIGHTPINK, Color.SIENNA, Color.OLIVE,
        Color.NAVY, Color.PLUM, Color.TEAL, Color.CHOCOLATE
    };

    private int CELL_SIZE = 40;
    private final int BOARD_MARGIN = 60;

    Windsurf: Refactor | Explain | Generate Javadoc | ×
    @FXML
    public void initialize() {
        algorithmComboBox.setItems(FXCollections.observableArrayList(
                "Uniform Cost Search", "Greedy Best-First Search", "A* Search", "Beam Search"));
        algorithmComboBox.setValue("Uniform Cost Search");
        algorithmComboBox.valueProperty().addListener((obs, oldVal, newVal) -> {
            boolean showHeuristic = newVal.contains("Greedy Best-First Search") || newVal.contains("A* Search")
                    || newVal.contains("Beam Search");
            heuristicLabel.setVisible(showHeuristic);
            heuristicLabel.setManaged(showHeuristic);
            heuristicComboBox.setVisible(showHeuristic);
            heuristicComboBox.setManaged(showHeuristic);
        });

        heuristicComboBox.setItems(FXCollections.observableArrayList(
                "Manhattan Distance", "Blocking Heuristic"));
        heuristicComboBox.setValue("Manhattan Distance");

        speedSlider.valueProperty().addListener((obs, oldVal, newVal) -> {
            if (animation != null) {
                animation.setRate(newVal.doubleValue() / 5.0);
            }
        });
    }
```

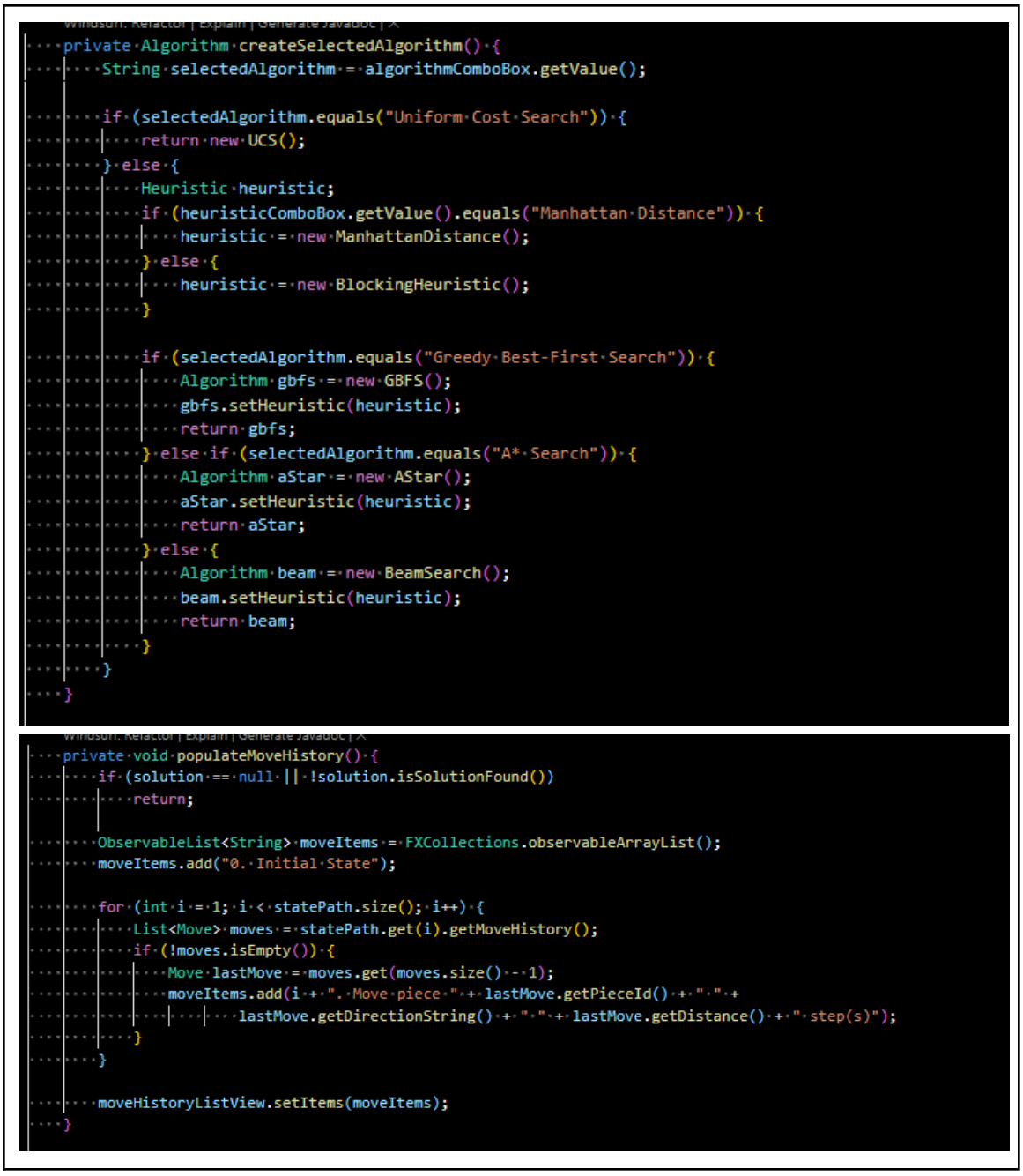
```java
    @FXML
    private void handleLoadFile() {
        FileChooser fileChooser = new FileChooser();
        fileChooser.setTitle("Open Rush Hour Configuration File");
        fileChooser.getExtensionFilters().add(
                new FileChooser.ExtensionFilter("Text Files", "*.txt"));

        File selectedFile = fileChooser.showOpenDialog(loadButton.getScene().getWindow());
        if (selectedFile != null) {
            try {
                initialBoard = FileHandler.readInputFile(selectedFile.getAbsolutePath());
                filePathLabel.setText(selectedFile.getName());
                solveButton.setDisable(false);
                resetUI();
                drawInitialBoard();
            } catch (Exception e) {
                showAlert("Error loading file", "Could not load the configuration file: " + e.getMessage());
                e.printStackTrace();
            }
        }
    }

    Windsurf: Refactor | Explain | Generate Javadoc | ×
    private void resetUI() {
        statusLabel.setText("Waiting");
        nodesVisitedLabel.setText("0");
        stepsLabel.setText("0");
        executionTimeLabel.setText("0 ms");
        playButton.setDisable(true);
        stepBackButton.setDisable(true);
        stepForwardButton.setDisable(true);
        resetButton.setDisable(true);
        solutionProgress.setProgress(0);
        moveHistoryListView.getItems().clear();
        currentMoveLabel.setText("No move");
        solution = null;
        statePath = null;
        currentStateIndex = 0;
        pieceRectangles.clear();
```

```
        boardPane.getChildren().clear();
    }

    Windsurf: Refactor | Explain | Generate Javadoc | ×
    private void drawInitialBoard() {
        if (initialBoard == null)
            return;

        int rows = initialBoard.getRows();
        int cols = initialBoard.getCols();

        double availableWidth = boardContainer.getWidth() - 2 * BOARD_MARGIN;
        double availableHeight = boardContainer.getHeight() - 2 * BOARD_MARGIN;

        if (availableWidth <= 0 || availableHeight <= 0) {
            Platform.runLater(this::drawInitialBoard);
            return;
        }

        double boardWidth = cols * CELL_SIZE + 2 * BOARD_MARGIN;
        double boardHeight = rows * CELL_SIZE + 2 * BOARD_MARGIN;

        boardPane.setPrefSize(boardWidth, boardHeight);

        for (int i = 0; i <= rows; i++) {
            Line hLine = new Line(BOARD_MARGIN, BOARD_MARGIN + i * CELL_SIZE,
                    BOARD_MARGIN + cols * CELL_SIZE, BOARD_MARGIN + i * CELL_SIZE);
            hLine.setStroke(Color.GRAY);
            boardPane.getChildren().add(hLine);
        }

        for (int i = 0; i <= cols; i++) {
            Line vLine = new Line(BOARD_MARGIN + i * CELL_SIZE, BOARD_MARGIN,
                    BOARD_MARGIN + i * CELL_SIZE, BOARD_MARGIN + rows * CELL_SIZE);
            vLine.setStroke(Color.GRAY);
            boardPane.getChildren().add(vLine);
        }
```

```
        Position exitPos = initialBoard.getExitPosition();
        double exitX = BOARD_MARGIN + exitPos.getCol() * CELL_SIZE;
        double exitY = BOARD_MARGIN + exitPos.getRow() * CELL_SIZE;

        if (exitPos.getCol() < 0)
            exitX = BOARD_MARGIN - CELL_SIZE;
        if (exitPos.getCol() >= cols)
            exitX = BOARD_MARGIN + cols * CELL_SIZE;
        if (exitPos.getRow() < 0)
            exitY = BOARD_MARGIN - CELL_SIZE;
        if (exitPos.getRow() >= rows)
            exitY = BOARD_MARGIN + rows * CELL_SIZE;

        Rectangle exitRect = new Rectangle(exitX, exitY, CELL_SIZE, CELL_SIZE);
        exitRect.setFill(Color.LIGHTGREEN);
        exitRect.setOpacity(0.5);
        exitRect.setStroke(Color.GREEN);
        exitRect.setStrokeWidth(1.5);
        boardPane.getChildren().add(exitRect);

        Label exitLabel = new Label("Exit");
        exitLabel.setTextFill(Color.DARKGREEN);
        exitLabel.setStyle("-fx-font-weight: bold;");
        exitLabel.setLayoutX(exitX + CELL_SIZE / 4.0);
        exitLabel.setLayoutY(exitY + CELL_SIZE / 4.0);

        boardPane.getChildren().add(exitLabel);

        Map<Character, Piece> pieces = initialBoard.getPieces();
        int colorIndex = 0;

        for (Map.Entry<Character, Piece> entry : pieces.entrySet()) {
            char pieceId = entry.getKey();
            Piece piece = entry.getValue();

            Rectangle rect = createPieceRectangle(piece, pieceColors[colorIndex % pieceColors.length], pieceId);
```

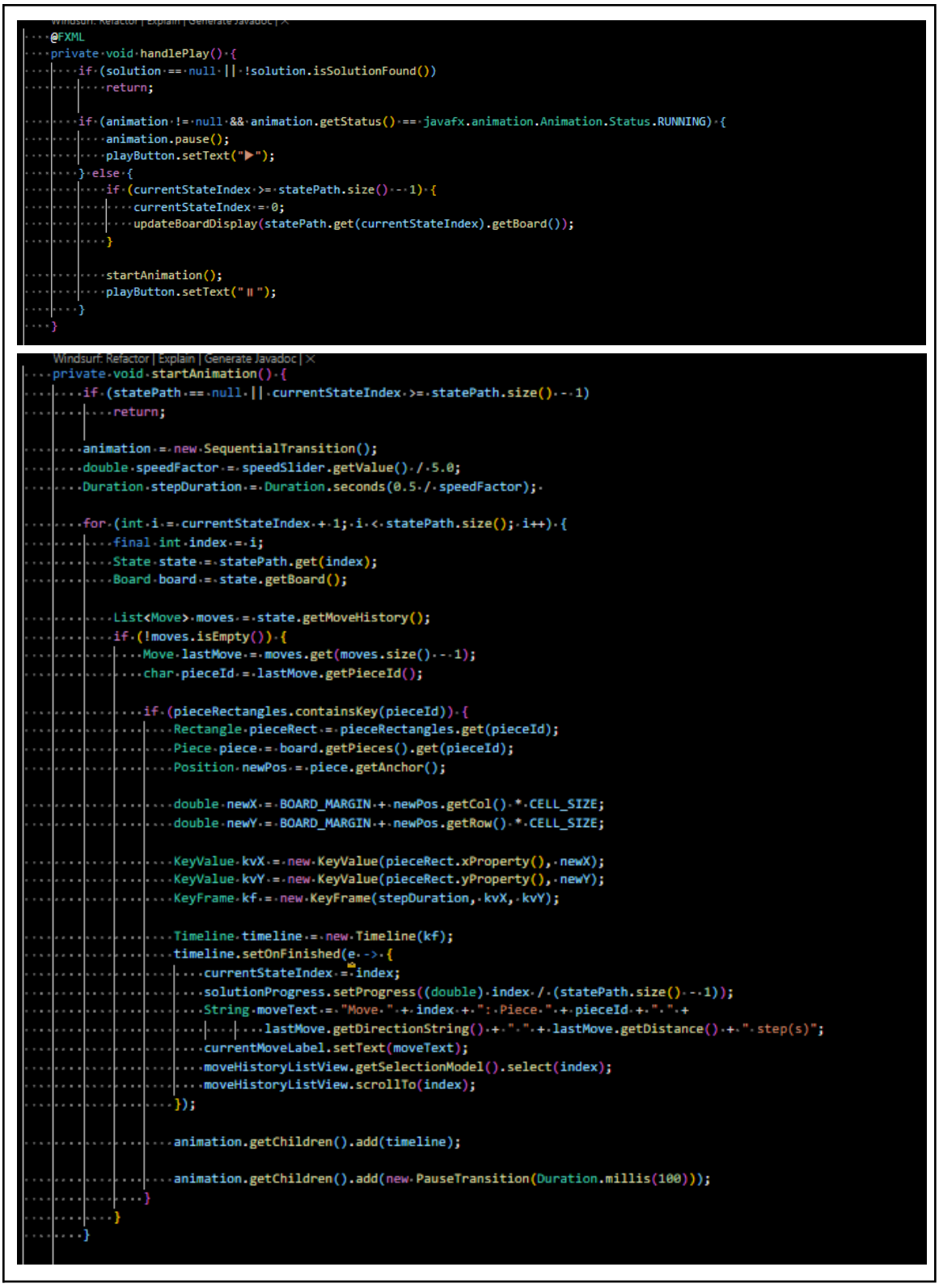
```
            Color pieceColor = piece.isPrimary() ? primaryPieceColor : pieceColors[colorIndex % pieceColors.length];
            if (!piece.isPrimary())
                colorIndex++;

            rect.setFill(pieceColor);
            pieceRectangles.put(pieceId, rect);
            boardPane.getChildren().add(rect);
        }

        Rectangle clip = new Rectangle();
        clip.widthProperty().bind(boardPane.widthProperty());
        clip.heightProperty().bind(boardPane.heightProperty());
        boardPane.setClip(clip);
    }

    Windsurf: Refactor | Explain | Generate Javadoc | ×
    private Rectangle createPieceRectangle(Piece piece, Color color, char id) {
        Position anchor = piece.getAnchor();
        int size = piece.getSize();
        boolean isHorizontal = piece.isHorizontal();

        Rectangle rect = new Rectangle(
                BOARD_MARGIN + anchor.getCol() * CELL_SIZE,
                BOARD_MARGIN + anchor.getRow() * CELL_SIZE,
                isHorizontal ? size * CELL_SIZE : CELL_SIZE,
                isHorizontal ? CELL_SIZE : size * CELL_SIZE);

        rect.setArcWidth(10);
        rect.setArcHeight(10);
        rect.setStroke(Color.BLACK);
        rect.setStrokeWidth(0.5);
        rect.setFill(color);
        rect.setId("piece-" + id);
        return rect;
    }
```

```
Windsurf: Refactor | Explain | Generate Javadoc | ×
    @FXML
    private void handleSolve() {
        if (initialBoard == null)
            return;

        resetUI();
        drawInitialBoard();

        solveButton.setDisable(true);
        statusLabel.setText("Solving...");

        new Thread(() -> {
            Algorithm algorithm = createSelectedAlgorithm();
            solution = algorithm.findSolution(initialBoard);

            Platform.runLater(() -> {
                solveButton.setDisable(false);

                if (solution != null && solution.isSolutionFound()) {
                    statePath = solution.getPath();
                    populateMoveHistory();

                    playButton.setDisable(false);
                    saveSolutionButton.setDisable(false);
                    stepBackButton.setDisable(false);
                    stepForwardButton.setDisable(false);
                    resetButton.setDisable(false);
                    handlePlay();
                } else {
                    statusLabel.setText("No Solution Found");
                }
            });
        }).start();
    }
```



```
private Algorithm createSelectedAlgorithm() {
    String selectedAlgorithm = algorithmComboBox.getValue();

    if (selectedAlgorithm.equals("Uniform Cost Search")) {
        return new UCS();
    } else {
        Heuristic heuristic;
        if (heuristicComboBox.getValue().equals("Manhattan Distance")) {
            heuristic = new ManhattanDistance();
        } else {
            heuristic = new BlockingHeuristic();
        }

        if (selectedAlgorithm.equals("Greedy Best-First Search")) {
            Algorithm gbfs = new GBFS();
            gbfs.setHeuristic(heuristic);
            return gbfs;
        } else if (selectedAlgorithm.equals("A* Search")) {
            Algorithm aStar = new AStar();
            aStar.setHeuristic(heuristic);
            return aStar;
        } else {
            Algorithm beam = new BeamSearch();
            beam.setHeuristic(heuristic);
            return beam;
        }
    }
}
```

```
private void populateMoveHistory() {
    if (solution == null || !solution.isSolutionFound())
        return;

    ObservableList<String> moveItems = FXCollections.observableArrayList();
    moveItems.add("0. Initial State");

    for (int i = 1; i < statePath.size(); i++) {
        List<Move> moves = statePath.get(i).getMoveHistory();
        if (!moves.isEmpty()) {
            Move lastMove = moves.get(moves.size() - 1);
            moveItems.add(i + ". Move piece " + lastMove.getPieceId() + " " +
                lastMove.getDirectionString() + " " + lastMove.getDistance() + " step(s)");
        }
    }

    moveHistoryListView.setItems(moveItems);
}
```



```
Windsurf: Refactor | Explain | Generate Javadoc | ×
@FXML
private void handlePlay(){
    if (solution == null || !solution.isSolutionFound())
        return;

    if (animation != null && animation.getStatus() == javafx.animation.Animation.Status.RUNNING) {
        animation.pause();
        playButton.setText("▶");
    }else {
        if (currentStateIndex >= statePath.size() - 1) {
            currentStateIndex = 0;
            updateBoardDisplay(statePath.get(currentStateIndex).getBoard());
        }

        startAnimation();
        playButton.setText("⏸");
    }
}
```

```
Windsurf: Refactor | Explain | Generate Javadoc | ×
private void startAnimation() {
    if (statePath == null || currentStateIndex >= statePath.size() - 1)
        return;

    animation = new SequentialTransition();
    double speedFactor = speedSlider.getValue() / 5.0;
    Duration stepDuration = Duration.seconds(0.5 / speedFactor);

    for (int i = currentStateIndex + 1; i < statePath.size(); i++) {
        final int index = i;
        State state = statePath.get(index);
        Board board = state.getBoard();

        List<Move> moves = state.getMoveHistory();
        if (!moves.isEmpty()) {
            Move lastMove = moves.get(moves.size() - 1);
            char pieceId = lastMove.getPieceId();

            if (pieceRectangles.containsKey(pieceId)) {
                Rectangle pieceRect = pieceRectangles.get(pieceId);
                Piece piece = board.getPieces().get(pieceId);
                Position newPos = piece.getAnchor();

                double newX = BOARD_MARGIN + newPos.getCol() * CELL_SIZE;
                double newY = BOARD_MARGIN + newPos.getRow() * CELL_SIZE;

                KeyValue kvX = new KeyValue(pieceRect.xProperty(), newX);
                KeyValue kvY = new KeyValue(pieceRect.yProperty(), newY);
                KeyFrame kf = new KeyFrame(stepDuration, kvX, kvY);

                Timeline timeline = new Timeline(kf);
                timeline.setOnFinished(e -> {
                    currentStateIndex = index;
                    solutionProgress.setProgress((double) index / (statePath.size() - 1));
                    String moveText = "Move " + index + ": Piece " + pieceId + " " +
                            lastMove.getDirectionString() + " " + lastMove.getDistance() + " step(s)";
                    currentMoveLabel.setText(moveText);
                    moveHistoryListView.getSelectionModel().select(index);
                    moveHistoryListView.scrollTo(index);
                });

                animation.getChildren().add(timeline);

                animation.getChildren().add(new PauseTransition(Duration.millis(100)));
            }
        }
    }
```

```
        animation.setOnFinished(e -> {
            playButton.setText("▶");

            Platform.runLater(() -> {
                statusLabel.setText("Solved!");
                nodesVisitedLabel.setText(String.valueOf(solution.getNodesVisited()));
                stepsLabel.setText(String.valueOf(solution.getStepCount()));
                executionTimeLabel.setText(solution.getExecutionTimeMs() + "ms");

                showSuccessDialog(solution.getStepCount(), solution.getExecutionTimeMs());
            });
        });

        animation.play();
    }

    Windsurf: Refactor | Explain | Generate Javadoc | ×
    @FXML
    private void handleStep() {
        if (solution == null || !solution.isSolutionFound())
            return;

        if (animation != null) {
            animation.stop();
            playButton.setText("▶");
        }

        if (currentStateIndex < statePath.size() - 1) {
            currentStateIndex++;
            State state = statePath.get(currentStateIndex);
            updateBoardDisplay(state.getBoard());

            solutionProgress.setProgress((double) currentStateIndex / (statePath.size() - 1));

            List<Move> moves = state.getMoveHistory();
            if (!moves.isEmpty()) {
                Move lastMove = moves.get(moves.size() - 1);
                String moveText = "Move " + currentStateIndex + ": Piece " + lastMove.getPieceId() + " " +
                        lastMove.getDirectionString() + " " + lastMove.getDistance() + " step(s)";
                currentMoveLabel.setText(moveText);

                moveHistoryListView.getSelectionModel().select(currentStateIndex);
                moveHistoryListView.scrollTo(currentStateIndex);
            }
        }
    }
```

```
    private void updateBoardDisplay(Board board) {
        Map<Character, Piece> pieces = board.getPieces();

        for (Map.Entry<Character, Piece> entry : pieces.entrySet()) {
            char pieceId = entry.getKey();
            Piece piece = entry.getValue();
            Position anchor = piece.getAnchor();

            if (pieceRectangles.containsKey(pieceId)) {
                Rectangle rect = pieceRectangles.get(pieceId);
                rect.setX(BOARD_MARGIN + anchor.getCol() * CELL_SIZE);
                rect.setY(BOARD_MARGIN + anchor.getRow() * CELL_SIZE);
            }
        }
    }

    Windsurf: Refactor | Explain | Generate Javadoc | ×
    @FXML
    private void handleReset() {
        if (solution == null)
            return;

        if (animation != null) {
            animation.stop();
            playButton.setText("▶");
        }

        currentStateIndex = 0;
        updateBoardDisplay(statePath.get(0).getBoard());
        solutionProgress.setProgress(0);
        currentMoveLabel.setText("Initial state");
        moveHistoryListView.getSelectionModel().select(0);
    }

    Windsurf: Refactor | Explain | Generate Javadoc | ×
    private void showAlert(String title, String message) {
        Alert alert = new Alert(Alert.AlertType.ERROR);
        alert.setTitle(title);
        alert.setHeaderText(null);
        alert.setContentText(message);
        alert.showAndWait();
    }
```

```
    private void showSuccessDialog(int steps, long timeMs) {
        Stage dialogStage = new Stage();
        dialogStage.initModality(Modality.APPLICATION_MODAL);
        dialogStage.setTitle("Success");

        Label icon = new Label("✅");
        icon.setStyle("-fx-font-size: 36px;");

        Label title = new Label("Puzzle Solved!");
        title.setStyle("-fx-font-size: 18px; -fx-font-weight: bold;");

        Label body = new Label("Solved in " + steps + " steps.\nExecution Time: " + timeMs + " ms.");
        body.setStyle("-fx-font-size: 14px; -fx-text-alignment: center;");

        Button okButton = new Button("OK");
        okButton.setDefaultButton(true);
        okButton.setOnAction(e -> dialogStage.close());

        VBox content = new VBox(15, icon, title, body, okButton);
        content.setAlignment(Pos.CENTER);
        content.setPadding(new Insets(20));

        Scene scene = new Scene(content);
        dialogStage.setScene(scene);
        dialogStage.setResizable(false);
        dialogStage.showAndWait();
    }

    Windsurf: Refactor | Explain | Generate Javadoc | ×
    @FXML
    private void handleStepForward() {
        if (solution == null || !solution.isSolutionFound())
            return;

        if (animation != null) {
            animation.stop();
            playButton.setText("▶");
        }

        if (currentStateIndex < statePath.size() - 1) {
            currentStateIndex++;
            updateBoardDisplay(statePath.get(currentStateIndex).getBoard());

            solutionProgress.setProgress((double) currentStateIndex / (statePath.size() - 1));

            Move lastMove = getLastMove(currentStateIndex);
            currentMoveLabel.setText("Move " + currentStateIndex + ": Piece " +
                    lastMove.getPieceId() + " " + lastMove.getDirectionString() + " " + lastMove.getDistance()
                    + " step(s)");
```

```
            moveHistoryListView.getSelectionModel().select(currentStateIndex);
            moveHistoryListView.scrollTo(currentStateIndex);
        }
    }

    Windsurf: Refactor | Explain | Generate Javadoc | ×
    @FXML
    private void handleStepBack() {
        if (solution == null || !solution.isSolutionFound())
            return;

        if (animation != null) {
            animation.stop();
            playButton.setText("▶");
        }

        if (currentStateIndex > 0) {
            currentStateIndex--;
            updateBoardDisplay(statePath.get(currentStateIndex).getBoard());

            solutionProgress.setProgress((double) currentStateIndex / (statePath.size() - 1));

            if (currentStateIndex > 0) {
                Move lastMove = getLastMove(currentStateIndex);
                currentMoveLabel.setText("Move " + currentStateIndex + ": Piece " +
                        lastMove.getPieceId() + " " + lastMove.getDirectionString() + " " + lastMove.getDistance()
                        + " step(s)");
            } else {
                currentMoveLabel.setText("Initial state");
            }

            moveHistoryListView.getSelectionModel().select(currentStateIndex);
            moveHistoryListView.scrollTo(currentStateIndex);
        }
    }

    Windsurf: Refactor | Explain | Generate Javadoc | ×
    private Move getLastMove(int index) {
        List<Move> moves = statePath.get(index).getMoveHistory();
        return moves.isEmpty() ? null : moves.get(moves.size() - 1);
    }
```

```
    Windsurf: Refactor | Explain | Generate Javadoc | ×
    @FXML
    private void handleSaveSolution() {
        if (solution == null || !solution.isSolutionFound())
            return;

        FileChooser fileChooser = new FileChooser();
        fileChooser.setTitle("Save Solution As");
        fileChooser.getExtensionFilters().add(new FileChooser.ExtensionFilter("Text Files", "*.txt"));
        fileChooser.setInitialFileName("solution.txt");

        File file = fileChooser.showSaveDialog(boardPane.getScene().getWindow());

        if (file != null) {
            try {
                String algoName = algorithmComboBox.getValue();
                FileHandler.writeSolutionToFile(
                        file.getAbsolutePath(),
                        initialBoard,
                        solution.getPath(),
                        algoName,
                        solution.getNodesVisited(),
                        solution.getExecutionTimeMs());
                showInfo("Saved", "Solution saved to:\n" + file.getAbsolutePath());
            } catch (Exception e) {
                showAlert("Save Failed", "Could not save file: " + e.getMessage());
                e.printStackTrace();
            }
        }
    }

    Windsurf: Refactor | Explain | Generate Javadoc | ×
    private void showInfo(String title, String message) {
        Stage dialogStage = new Stage();
        dialogStage.initModality(Modality.APPLICATION_MODAL);
        dialogStage.setTitle(title);

        Label label = new Label(message);
        label.setWrapText(true);
        label.setStyle("-fx-font-size: 13px; -fx-text-fill: #4A4A4A;");

        Button okButton = new Button("OK");
        okButton.setOnAction(e -> dialogStage.close());
        okButton.setStyle("-fx-background-color: #8ACCD5; -fx-text-fill: white; -fx-font-weight: bold;");

        VBox layout = new VBox(15, label, okButton);
        layout.setAlignment(Pos.CENTER);
        layout.setPadding(new Insets(20));
        layout.setStyle("-fx-background-color: #F8F8E1; -fx-border-color: #FFC1DA; -fx-border-width: 2;");

        Scene scene = new Scene(layout, 400, 400);
        dialogStage.setScene(scene);
        dialogStage.setResizable(false);
        dialogStage.showAndWait();
    }
}
```

- **MainView.fxml**

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<?import javafx.geometry.*?>

<BorderPane xmlns="http://javafx.com/javafx"
    xmlns:fx="http://javafx.com/fxml"
    fx:controller="gui.MainController"
    prefHeight="600" prefWidth="800"
    stylesheets="@style.css">

    <top>
        <VBox spacing="10">
            <padding>
                <Insets top="10" right="10" bottom="10" left="10" />
            </padding>
            <HBox spacing="10" alignment="CENTER_LEFT">
                <Button fx:id="loadButton" text="Load Configuration" onAction="#handleLoadFile" />
                <Label fx:id="filePathLabel" text="No file loaded" />
            </HBox>

            <HBox spacing="10" alignment="CENTER_LEFT">
                <Label text="Algorithm:" />
                <ComboBox fx:id="algorithmComboBox" prefWidth="150" />

                <Label text="Heuristic:" fx:id="heuristicLabel" visible="false" managed="false" />
                <ComboBox fx:id="heuristicComboBox" prefWidth="150" visible="false" managed="false" />

                <Button fx:id="solveButton" text="Solve" onAction="#handleSolve" disable="true" />
            </HBox>
        </VBox>
    </top>

    <center>
        <StackPane fx:id="boardContainer" alignment="CENTER">
            <padding>
                <Insets top="10" right="10" bottom="10" left="10" />
            </padding>
            <Pane fx:id="boardPane" />
        </StackPane>
    </center>

    <right>
        <VBox fx:id="sidePanel" spacing="10" prefWidth="200">
            <padding>
                <Insets top="10" right="10" bottom="10" left="10" />
            </padding>

            <Button fx:id="saveSolutionButton" text="💾 Save Solution"
                onAction="#handleSaveSolution" disable="true" />
```

```
            <Label text="Solution Information" styleClass="title" />
            <Separator />

            <GridPane hgap="10" vgap="5">
                <columnConstraints>
                    <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
                    <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
                </columnConstraints>

                <Label text="Status:" styleClass="grid-label" GridPane.rowIndex="0"
                    GridPane.columnIndex="0" />
                <Label fx:id="statusLabel" styleClass="grid-label" GridPane.rowIndex="0"
                    GridPane.columnIndex="1" />

                <Label text="Nodes Visited:" styleClass="grid-label" GridPane.rowIndex="1"
                    GridPane.columnIndex="0" />
                <Label fx:id="nodesVisitedLabel" styleClass="grid-label" GridPane.rowIndex="1"
                    GridPane.columnIndex="1" />

                <Label text="Steps:" styleClass="grid-label" GridPane.rowIndex="2"
                    GridPane.columnIndex="0" />
                <Label fx:id="stepsLabel" styleClass="grid-label" GridPane.rowIndex="2"
                    GridPane.columnIndex="1" />

                <Label text="Execution Time:" styleClass="grid-label" GridPane.rowIndex="3"
                    GridPane.columnIndex="0" />
                <Label fx:id="executionTimeLabel" styleClass="grid-label" GridPane.rowIndex="3"
                    GridPane.columnIndex="1" />
            </GridPane>

            <Separator />

            <Label text="Animation Controls" styleClass="title" />
            <HBox spacing="10" alignment="CENTER">
                <Button fx:id="playButton" text="▶" onAction="#handlePlay" disable="true" />
                <Button fx:id="stepBackButton" text="⏮" onAction="#handleStepBack" disable="true" />
                <Button fx:id="stepForwardButton" text="⏭" onAction="#handleStepForward"
                    disable="true" />
                <Button fx:id="resetButton" text="🔁" onAction="#handleReset" disable="true" />
            </HBox>

            <Separator />

            <Label text="Animation Speed" />
            <Slider fx:id="speedSlider" min="1" max="10" value="5" />

            <Separator />
```

```
            <Label text="Move History" styleClass="title" />
            <ListView fx:id="moveHistoryListView" VBox.vgrow="ALWAYS" />
        </VBox>
    </right>

    <bottom>
        <HBox spacing="10" alignment="CENTER_RIGHT">
            <padding>
                <Insets top="10" right="10" bottom="10" left="10" />
            </padding>
            <Label fx:id="currentMoveLabel" text="No move" HBox.hgrow="ALWAYS" />
            <ProgressBar fx:id="solutionProgress" prefWidth="200" progress="0.0" />
        </HBox>
    </bottom>
</BorderPane>
```

LAMPIRAN

Tautan Repository Github

Berikut tautan repository github kelompok kami untuk Tugas Kecil 3 IF2211 Strategi Algoritma :
https://github.com/carllix/Tucil3_13523091_13523107

Tabel Kelengkapan Spesifikasi

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

DAFTAR PUSTAKA

Maulidevi, Nur Ulfa. 2025. *Penentuan Rute (Route/Path Planning) (Bagian 1)*. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-(2025)-Bagian1.pdf (Diakses pada 18 Mei 2025).

Maulidevi, Nur Ulfa. 2025. *Penentuan Rute (Route/Path Planning) (Bagian 2)*. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf (Diakses pada 18 Mei 2025).

Belwariar, R. (2024, July 30). *A* Search Algorithm*. GeeksforGeeks. Retrieved May 18, 2025, from https://www.geeksforgeeks.org/a-search-algorithm/

*Introduction to A**. (2025, April 19). Stanford CS Theory. Retrieved May 18, 2025, from https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

Jain, S. (2024, January 18). *Greedy Best first search algorithm*. GeeksforGeeks. Retrieved May 18, 2025, from https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/

Jain, S. (2024, August 23). *Uniform Cost Search (UCS) in AI*. GeeksforGeeks. Retrieved May 18, 2025, from https://www.geeksforgeeks.org/uniform-cost-search-ucs-in-ai/