LAPORAN

TUGAS KECIL 1

IF2211 Strategi Algoritma

Semester II 2025/2026

Algoritma Brute Force pada

Penyelesaian Permainan Queens LinkedIn

Oleh:

Niko Samuel Simanjuntak

13524029

K-01

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

BANDUNG

2026

1. PENDAHULUAN

1.1. Latar Belakang

Queens adalah gim logika yang tersedia pada situs jejaring profesional LinkedIn. Tujuan dari gim ini adalah menempatkan *N queen* pada sebuah papan persegi $N \times N$ yang terbagi dalam beberapa daerah warna sehingga terdapat hanya satu *queen* pada tiap baris, kolom, dan daerah warna. Selain itu, satu *queen* tidak dapat ditempatkan bersebelahan dengan *queen* lainnya, termasuk secara diagonal.

1.2. Tujuan

Pada tugas kecil 1 ini, saya akan membuat program yang dapat menemukan satu solusi penempatan *queen* pada suatu papan berwarna yang diberikan atau menampilkan bahwa tidak ada solusi yang valid. Program melakukan pencarian solusi menggunakan algoritma *brute force* murni, tanpa menggunakan heuristik apapun.

2. DASAR TEORI

2.1. Algoritma *Brute Force*

Algoritma *brute force* didefinisikan sebagai pendekatan penyelesaian masalah yang dilakukan secara lempang (*straightforward*), yaitu memecahkan persoalan dengan cara yang sederhana, langsung, dan jelas (*obvious way*). Langkah-langkah dalam algoritma ini biasanya disusun berdasarkan pernyataan masalah (*problem statement*) serta definisi konsep yang terlibat di dalamnya. Secara umum, metode *brute force* bekerja dengan cara membangkitkan kandidat solusi dan memeriksa masing-masing kandidat hingga ditemukan solusi yang memenuhi seluruh kriteria. Kesederhanaan menjadi keunggulan utama pendekatan ini karena mudah dipahami dan diimplementasikan. Selain itu, *brute force* menjamin ditemukannya solusi yang benar apabila solusi tersebut memang ada, karena seluruh ruang solusi diperiksa secara menyeluruh. Namun, konsekuensi dari sifat menyeluruh tersebut adalah tingginya kebutuhan waktu komputasi, terutama ketika ukuran masukan bertambah besar, sehingga metode ini sering kali hanya efektif untuk persoalan berskala kecil atau sebagai dasar pembanding bagi algoritma yang lebih efisien.

2.2. Pencarian *Exhaustive*

Pencarian *exhaustive* merupakan bagian dari *brute force* yang diterapkan pada persoalan yang solusinya berupa kombinasi elemen-elemen, khususnya pada masalah kombinatorial. Teknik ini umumnya menggunakan mekanisme *generate and test* (bangkitkan dan uji), yaitu dengan cara menyenaraikan seluruh kemungkinan kandidat

solusi terlebih dahulu, kemudian menguji setiap kandidat tersebut apakah memenuhi kriteria solusi atau tidak.

Ruang pencarian pada metode ini biasanya berupa seluruh himpunan bagian (*subset*) atau seluruh permutasi dari elemen-elemen masalah. Contoh penerapannya dapat ditemukan pada *Traveling Salesperson Problem* (TSP) maupun *Knapsack Problem*, dengan semua kemungkinan susunan atau kombinasi perlu diperiksa untuk memastikan solusi optimal ditemukan. Meskipun pendekatan ini menjamin ditemukannya solusi global yang benar, kompleksitas waktunya sering kali tidak mangkus karena berada pada orde eksponensial, seperti $O(2^n)$, atau bahkan faktorial, seperti $O(n!)$. Oleh sebab itu, pencarian *exhaustive* umumnya hanya praktis digunakan untuk ukuran masukan yang relatif kecil.

2.3. Teknik Heuristik

Teknik heuristik merupakan pendekatan yang dikembangkan untuk mengatasi kelemahan utama dari pencarian *exhaustive*, yaitu kompleksitas waktu yang sangat besar pada persoalan kombinatorial. Jika pencarian *exhaustive* harus memproses seluruh kemungkinan permutasi atau himpunan bagian yang jumlahnya dapat mencapai 2^n atau $n!$, maka heuristik bertujuan untuk memangkas ruang pencarian tersebut dengan memanfaatkan informasi tambahan atau aturan praktis tertentu.

Heuristik tidak menjamin solusi optimal secara matematis, tetapi mampu menghasilkan solusi yang baik dalam waktu yang lebih singkat. Teknik ini menggunakan pendekatan yang tidak formal, misalnya berdasarkan penilaian intuitif, terkaan, atau akal sehat. Dengan memanfaatkan batasan dari permasalahan, algoritma tidak perlu membangkitkan seluruh kemungkinan secara buta, melainkan hanya menelusuri konfigurasi yang berpotensi memenuhi syarat. Akibatnya, waktu komputasi dapat dikurangi dibandingkan pencarian *exhaustive* murni.

3. IMPLEMENTASI ALGORITMA

3.1. Algoritma *Brute Force* Kombinasi Petak

Algoritma ini diimplementasikan dalam kelas `BruteForce.java`. Algoritma ini bekerja dengan prinsip *brute force* murni menggunakan mekanisme *Generate and Test*. Algoritma membangkitkan seluruh kemungkinan kombinasi penempatan *N queen* pada *N* × *N* petak, lalu menguji validitasnya di akhir.

3.1.1. Pseudocode

```
procedure solveCombination(start_cell_idx, queens_placed)
    // Basis: Jika jumlah ratu yang diletakkan sudah mencapai total wilayah (N)
    if (queens_placed == total_regions) then
        // Validasi Akhir: Cek seluruh aturan (Baris, Kolom, Wilayah, Tetangga)
        if (isValidConfiguration()) then
            solution_found <- true
        return
    // Rekursi: Iterasi sel linear dari indeks saat ini hingga sel terakhir
    for i <- start_cell_idx to total_cells - 1 do
        // Konversi indeks linear ke koordinat matriks
        r <- i div board_size
        c <- i mod board_size
        // Coba letakkan ratu
        board.setQueen(r, c, true)
        // Panggil rekursif untuk menaruh ratu berikutnya
        solveCombination(i + 1, queens_placed + 1)
        // Jika solusi sudah ditemukan, hentikan loop
        if (solution_found or not running) then
            return
        // Angkat kembali ratu (batalkan langkah)
        board.setQueen(row, col, false)
end procedure
```

```
function isValidConfiguration() -> boolean
    // Langkah 1: Kumpulkan lokasi seluruh ratu yang ada di papan
    queens <- []
    for r <- 0 to board_size - 1 do
        for c <- 0 to board_size - 1 do
            if board.hasQueen(r, c) then
                add {r, c} to queens
    // Langkah 2: Bandingkan setiap pasangan ratu
    for i <- 0 to size(queens) - 1 do
        for j <- i + 1 to size(queens) - 1 do
            q1 <- queens[i]
            q2 <- queens[j]
            // Cek 1: Apakah satu baris?
            if (q1.row == q2.row) then return false
            // Cek 2: Apakah satu kolom?
            if (q1.col == q2.col) then return false
            // Cek 3: Apakah satu wilayah warna?
            if board.getRegion(q1.row, q1.col) == board.getRegion(q2.row, q2.col) then
                return false
            // Cek 4: Apakah bersentuhan?
            if (abs(q1.row - q2.row) <= 1 and abs(q1.col - q2.col) <= 1) then
                return false
    // Jika semua pasangan aman
    return true
end function
```

3.1.2. Penjelasan Langkah

1. Papan $N \times N$ dianggap sebagai senarai sel linear dengan panjang N^2.

2. Algoritma mencoba menempatkan *queen* pada setiap sel, lalu melanjutkan secara rekursif ke sel-sel berikutnya untuk menempatkan sisa ratu. Ini dilakukan tanpa mempedulikan aturan baris/kolom/warna pada saat penempatan.
3. Setelah *N* buah *queen* berhasil diletakkan di atas papan, fungsi `isValidConfiguration()` dipanggil. Fungsi ini memeriksa apakah konfigurasi tersebut melanggar aturan (ada *queen* sebaris, sekolom, sewarna, atau bersentuhan).
4. Jika tes berhasil, status solusi disimpan dan pencarian dihentikan. Jika gagal, algoritma mencoba kombinasi sel berikutnya.

3.1.3. Analisis Kompleksitas

- **Kompleksitas Waktu**

  Masalah ini ekuivalen dengan memilih N posisi dari N^2 petak yang tersedia. Jumlah kemungkinan konfigurasi adalah:

  $$C(N^2, N) = \frac{(N^2)!}{N!\,(N^2 - N)!}$$

  Untuk setiap konfigurasi *N queen*, dilakukan validasi `isValidConfiguration()` yang membandingkan setiap pasangan ratu, dengan kompleksitas $O(N^2)$. Kompleksitas waktu total terburuk (*Worst Case*) adalah:

  $$T(n) = O(N^2 . \binom{N^2}{N})$$

- **Kompleksitas Ruang**

  Algoritma menggunakan rekursi `solveCombination()` dengan kedalaman maksimal *N* (jumlah *queen*) sehingga *auxiliary space* untuk tumpukan rekursi adalah $O(N)$. Pada `isValidConfiguration()`, juga dibuat struktur data sementara, `List<int[]> queens`, dengan ukuran maksimal *N* sehingga tambahan ruang sementara itu juga $O(N)$. Dengan demikian, ruang tambahan (*auxiliary*) yang digunakan algoritma ini adalah $O(N)$. Namun, algoritma tetap menyimpan *state* papan permainan berupa matriks $N \times N$. Jika menghitung seluruh memori yang digunakan program (bukan hanya *auxiliary*), kebutuhan ruang menjadi $O(N^2)$ untuk menyimpan papan ditambah O(N) untuk rekursi yang secara asimtotik tetap $O(N^2)$.

3.2. Algoritma *Brute Force* Per Wilayah

Algoritma ini diimplementasikan dalam kelas `OptimationBruteForce.java` dan menggunakan metode validasi dari kelas `Board.java`. Algoritma ini dirancang untuk menangani kelemahan algoritma pertama pada masukan berukuran besar dengan cara memangkas ruang pencarianmenggunakan teknik *backtracking*. *Backtracking* adalah perbaikan dari *brute force* yang secara sistematis mencari solusi persoalan di antara semua kemungkinan solusi yang ada. Perbedaannya dengan *brute force* murni adalah bahwa *backtracking* tidak menelusuri semua kemungkinan jalur sampai tuntas. Algoritma ini menerapkan prinsip *pruning* (pemangkasan). Algoritma ini menempatkan satu *queen* secara bertahap pada setiap wilayah warna, lalu menguji validitasnya segera di setiap langkah penempatan. Jika pada suatu langkah penempatan ratu diketahui melanggar aturan (fungsi `isSafe()` bernilai *false*), algoritma langsung mematikan simpul pencarian tersebut dan mundur (*backtrack*) ke langkah sebelumnya. Hal ini menjamin kebenaran solusi, namun dengan waktu eksekusi yang lebih efisien.

3.2.1. Pseudocode

```
procedure prepareRegions()
    // Inisialisasi struktur data peta (Map) dan senarai (List)
    region_cells <- new Map<Character, List<Point>>()
    region_order <- new List<Character>()
    // Iterasi seluruh sel pada papan
    for r <- 0 to size - 1 do
        for c <- 0 to size - 1 do
            region_char <- board.getRegion(r, c)
            // Jika wilayah ini belum terdaftar, tambahkan ke peta
            if (not region_cells.containsKey(region_char)) then
                region_cells.put(region_char, new List<Point>())
                region_order.add(region_char)
            // Simpan koordinat sel ke dalam daftar wilayahnya
            region_cells.get(region_char).add(new Point(r, c))
end procedure
```

```
function solveByRegion(region_idx) -> boolean
    // Basis: Jika semua wilayah sudah berhasil diisi (Solusi ditemukan)
    if (region_idx == region_order.size()) then
        return true
    // Ambil daftar koordinat sel untuk wilayah saat ini
    current_region_char <- region_order.get(region_idx)
    available_cells <- region_cells.get(current_region_char)
    // Iterasi: Coba letakkan ratu di setiap sel dalam wilayah ini
    for each point in available_cells do
        // Langkah 1: Letakkan ratu sementara
        board.setQueen(point.row, point.col, true)
        // Langkah 2: Pruning
        // Cek apakah posisi ini aman terhadap ratu-ratu sebelumnya
        if (board.isSafe(point.row, point.col)) then
```

```
            // Langkah 3: Rekursif ke wilayah berikutnya
            if (solveByRegion(region_idx + 1) is true) then
                return true
        // Langkah 4: Backtrack
        board.setQueen(point.row, point.col, false)
    // Jika tidak ada sel di wilayah ini yang valid
    return false
end function
```

```
function isSafe(target_row, target_col) -> boolean
    current_region <- region_map[target_row][target_col]
    // Periksa seluruh sel di papan untuk mencari ratu lain
    for r <- 0 to size - 1 do
        for c <- 0 to size - 1 do
            // Jika ditemukan ada ratu di posisi (r, c)
            if (queens[r][c] is true) then
                // Aturan 1 & 2: Cek Baris dan Kolom yang sama
                if (r == target_row or c == target_col) then return false
                // Aturan 3: Cek Wilayah Warna yang sama
                if (region_map[r][c] == current_region) then return false
                // Aturan 4: Cek Tetangga
                if (abs(r - target_row) <= 1 and abs(c - target_col) <= 1) then
                    return false
    // Jika aman
    return true
end function
```

3.2.2. Penjelasan Langkah

1. Algoritma menyalin koordinat seluruh sel (N^2) ke dalam `Map` mengubah cara pandang terhadap papan dari matriks 2D menjadi himpunan wilayah.
2. Pada setiap level, algoritma mencoba memilih satu sel dari sekumpulan sel yang tersedia di wilayah tersebut.
3. Fungsi `isSafe()` dipanggil setiap kali *queen* diletakkan. Jika `isSafe()` mengembalikan *false*, algoritma tidak perlu mengecek cabang di bawahnya (*pruning*).
4. Jika pada suatu wilayah tidak ditemukan posisi yang aman, algoritma kembali ke pemanggil fungsi sebelumnya (*backtrack*) untuk memindahkan ratu di wilayah sebelumnya ke posisi baru.
5. Jika algoritma berhasil mencapai kedalaman rekursi saat indeks wilayah sama dengan jumlah total wilayah, berarti seluruh wilayah telah berhasil diisi oleh ratu yang valid. Algoritma mengembalikan nilai *true* yang akan merambat naik ke tumpukan rekursi teratas untuk menghentikan seluruh proses pencarian dan menyatakan solusi ditemukan.

3.2.3. Analisis Kompleksitas

- **Kompleksitas Waktu**

  Pencarian memiliki kedalaman N (jumlah wilayah) dengan fator percabangan rata-rata adalah N. Dalam kasus terburuk (*worst case*), algoritma harus menelusuri seluruh kemungkinan kombinasi sehingga batas atas penulusuran adalah $O(N^N)$. Sementara itu, pada setiap langkah penempatan ratu, algoritma memanggil fungsi `isSafe()` yang memakan waktu $O(N^2)$. Kompleksitas waktu total terburuk (*Worst Case*) adalah:

  $$T(n) = O(N^{N+2})$$

  Meskipun batas atasnya eksponensial, *pruning* memangkas sebagian besar cabang yang tidak valid di awal.

- **Kompleksitas Ruang**

  Algoritma ini memakai representasi papan $N \times N$ (membutuhkan $O(N^2)$) dan juga mengelompokkan koordinat seluruh sel ke dalam `Map<Character, List<Point>>`. Karena struktur *map* tersebut menyimpan informasi untuk sebagian besar (atau seluruh) sel, penggunaan ruang tambahan dari struktur ini adalah $O(N^2)$. Kedalaman rekursi setara jumlah wilayah ($\approx N$), sehingga tumpukan rekursi menambah $O(N)$. Secara total, kompleksitas ruang asimtotiknya adalah $O(N^2)$.

4. IMPLEMENTASI PROGRAM

4.1. Lingkungan Pengembangan

Program ini dikembangkan dengan spesifikasi teknis sebagai berikut:

- Bahasa Pemrograman: Java (JDK 21)
- Manajemen Proyek & Dependensi: Apache Maven
- Kerangka Kerja GUI: JavaFX (Versi 21)

Untuk menjalankan program, gunakan perintah berikut pada terminal:

```
mvn clean compile
mvn clean javafx:run
```

atau dapat dengan membuka file `.jar` yang tersedia pada `bin/queens-solver-1.0.jar` di repositori yang ditunjukkan pada 4.3.

4.2. Struktur Program

Struktur direktori proyek secara keseluruhan adalah sebagai berikut:

```
.
├── .git/
├── bin/
│   └── queens-solver-1.0.jar
├── doc/
├── src/
│   └── main/
│       ├── java/
│       │   └── stima/
│       │       ├── model/
│       │       │   └── Board.java
│       │       ├── solver/
│       │       │   ├── BruteForce.java
│       │       │   ├── OptimizationBruteForce.java
│       │       │   ├── Solver.java
│       │       │   └── SolverAlgorithm.java
│       │       ├── utils/
│       │       │   └── InputLoader.java
│       │       ├── App.java
│       │       ├── Launcher.java
│       │       └── MainController.java
│       └── resources/
│           └── stima/
│               ├── layout.fxml
│               └── logo.png
├── test/
├── .gitignore
├── LICENSE
├── pom.xml
└── README.md
```

Program dibuat dengan menerapkan prinsip *Object-Oriented Programming* (OOP) dan pola desain MVC (*Model-View-Controller*) untuk memisahkan logika data, tampilan, dan kontrol alur program. Struktur paket (*package*) dalam kode sumber adalah sebagai berikut:

1. `stima.model`
    - `Board.java`: Merepresentasikan model data papan permainan. Kelas ini menyimpan status matriks wilayah warna (`char[][]`) dan posisi ratu (`boolean[][]`), serta menyediakan metode validasi aturan permainan.
2. `stima.solver`
    - `BruteForce.java`: Implementasi algoritma *brute force* standar yang menelusuri kombinasi petak.
    - `OptimationBruteForce.java`: Implementasi algoritma optimasi menggunakan *backtracking* berbasis wilayah.
    - `Solver.java`: Kelas pembantu (*helper*) yang berisi definisi status (`record SolverStatus`) untuk komunikasi antara *thread* algoritma dan UI.
    - `SolverAlgorithm.java`: Sebuah *interface* yang memungkinkan program untuk mengganti algoritma dengan mudah.
3. `stima.utils`
    - `InputLoader.java`: Menangani pembacaan input dari fail teks `.txt` dan melakukan validasi format.
4. `stima` (*Controller* dan *Main*)
    - `App.java` dan `Launcher.java`: Titik masuk utama (*entry point*) aplikasi yang menginisialisasi JavaFX *Stage*.
    - `MainController.java`: Kelas pengendali (*controller*) yang menghubungkan antarmuka pengguna FXML dengan logika model dan solver. Kelas ini juga menangani *multithreading* agar proses pencarian solusi tidak membekukan antarmuka pengguna.

4.3. Repositori Kode Sumber

Kode sumber lengkap disimpan dan dapat diakses melalui repositori publik berikut: https://github.com/NikSamSim/Tucil1_13524029.

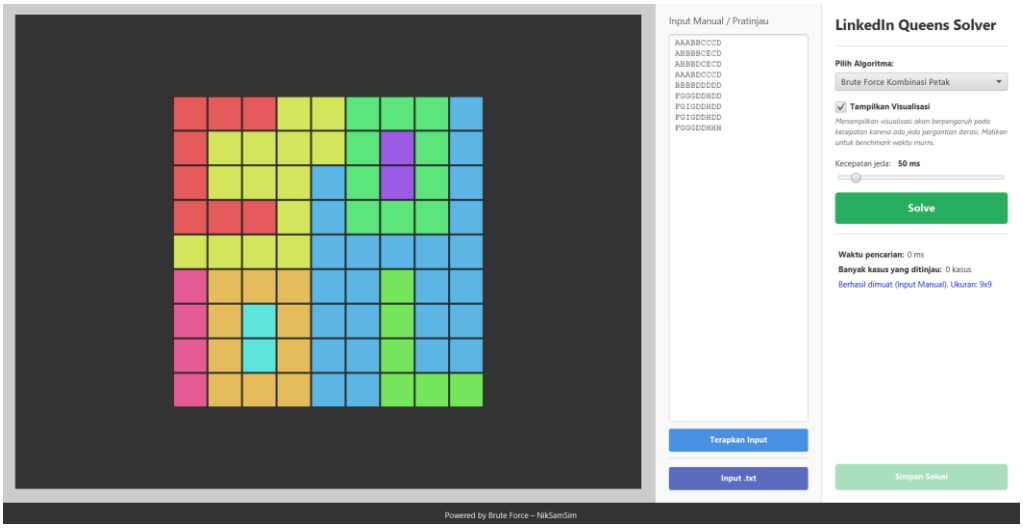
5. PENGUJIAN DAN EKSPERIMEN

Pada bab ini dilakukan pengujian program berdasarkan spesifikasi yang telah ditetapkan. Pengujian dibagi menjadi beberapa studi kasus, sesuai dengan metode input yang digunakan.

5.1. Input Teks Manual

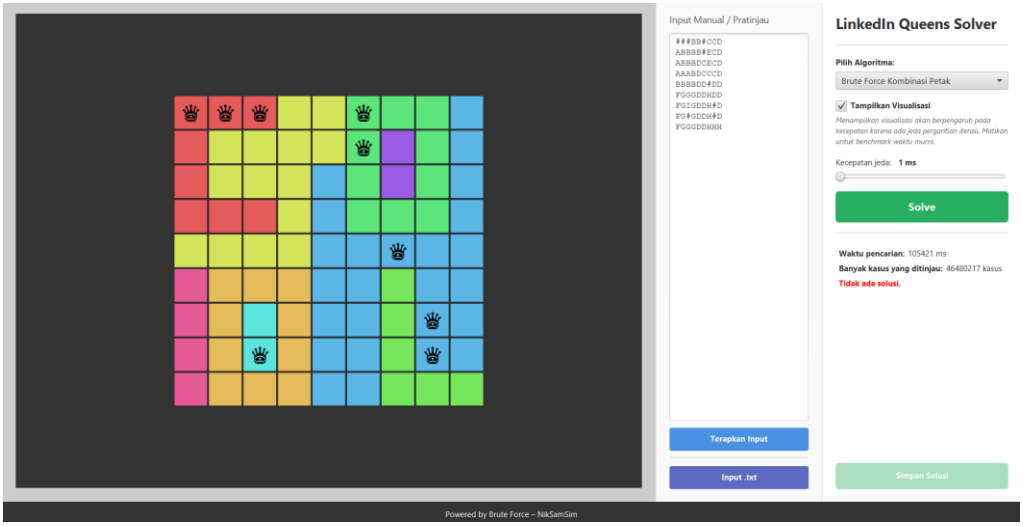
- input1.txt

  Input:



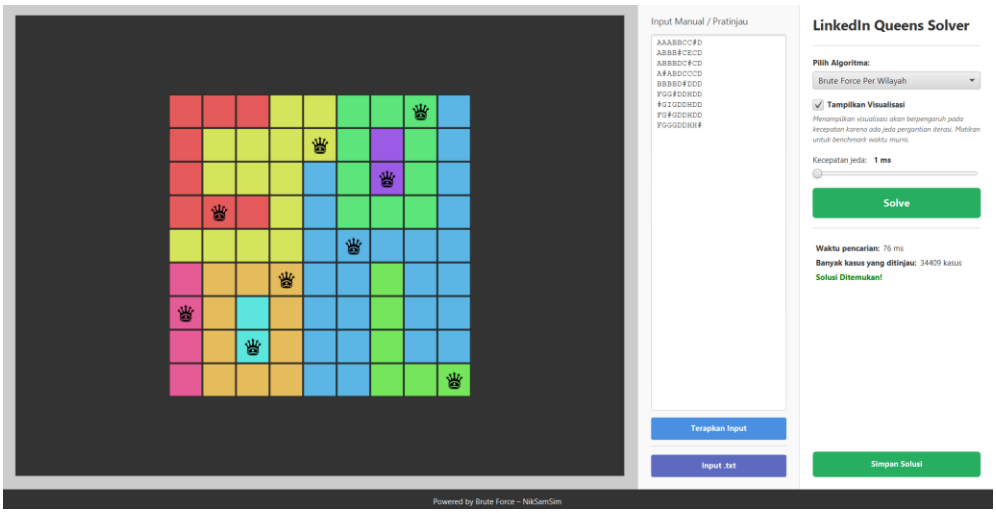
  Hasil:

  Menggunakan *Brute Force* Kombinasi Petak



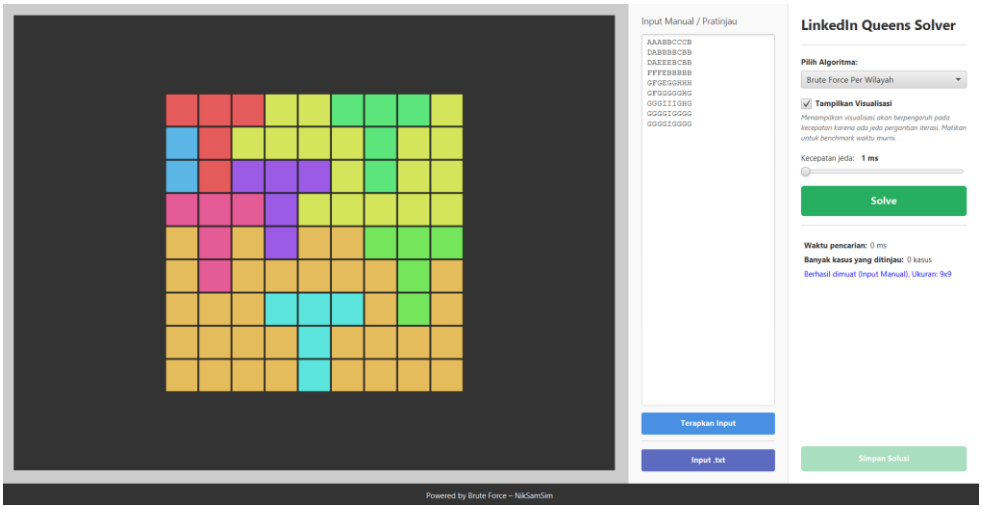
  Keterangan: Sudah ditinjau 46 juta lebih kasus, namun solusi masih belum ditemukan. Oleh karena itu, selanjutnya akan digunakan algoritma *brute force* per wilayah (*backtracking*) pada pengujian untuk laporan ini (kecuali untuk input3.txt karena *N* masih cukup kecil).

Menggunakan *Brute Force* Per Wilayah (*Backtracking*)

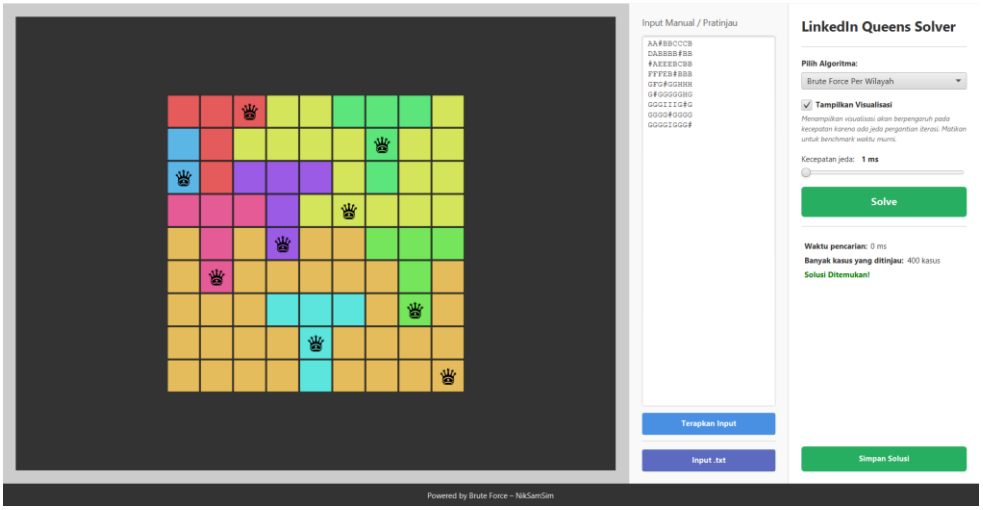


- input2.txt

Input:

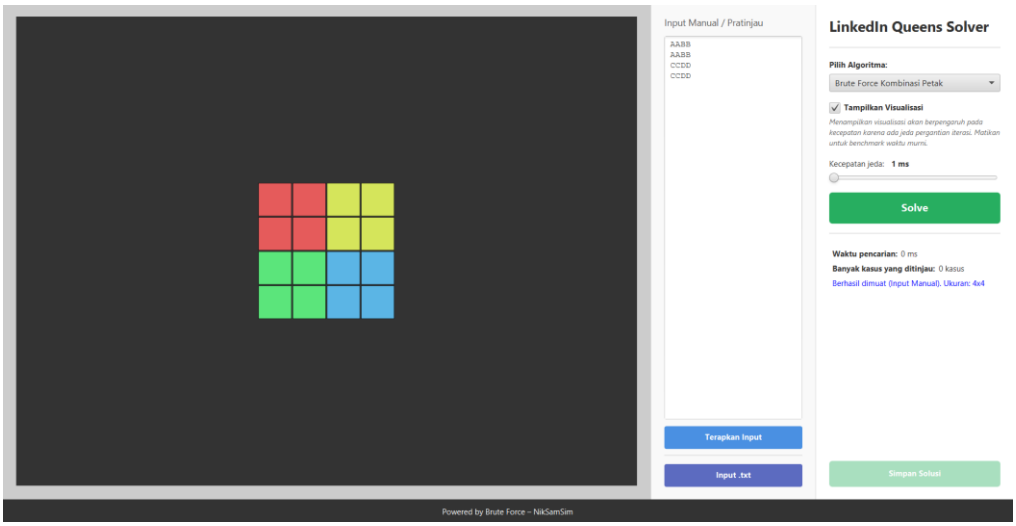


Hasil:



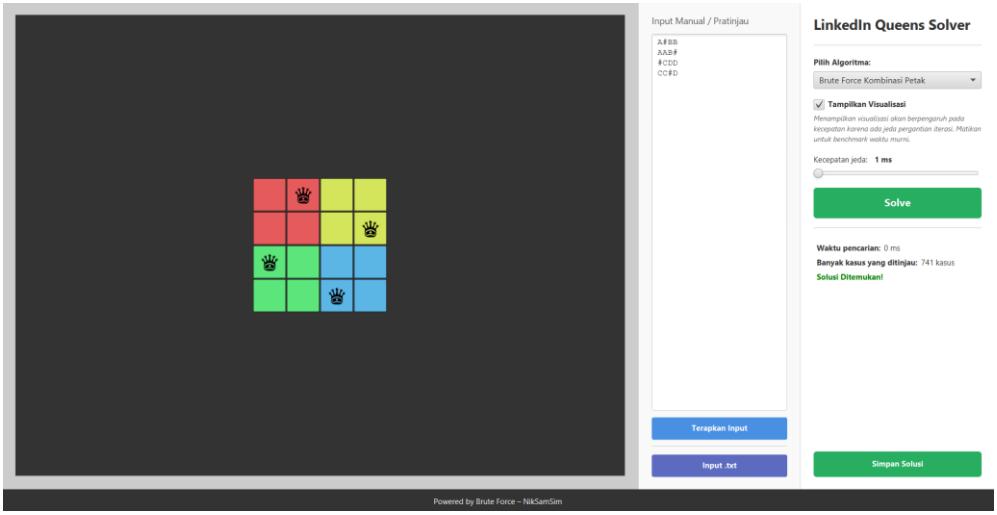
- input3.txt

Input:

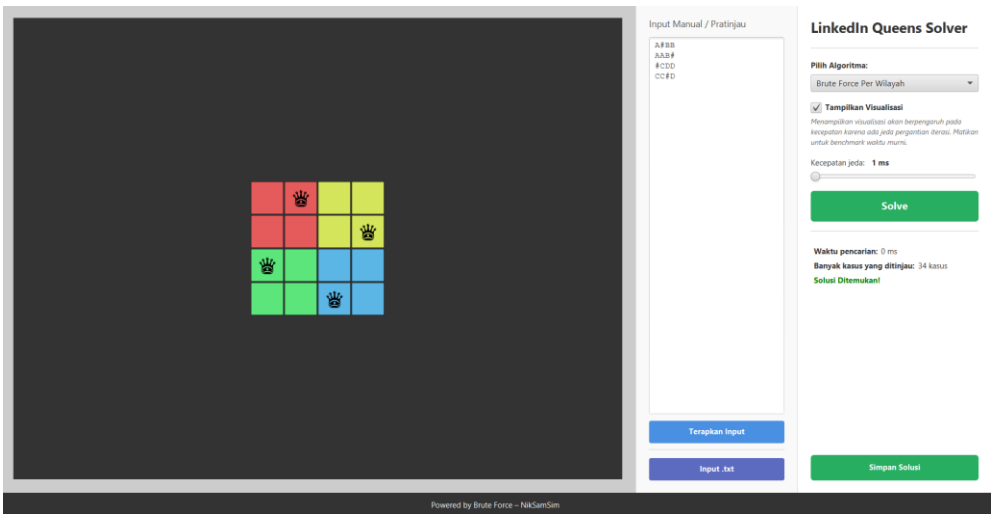


Hasil:

Menggunakan *Brute Force* Kombinasi Petak (741 kasus yang ditinjau)



Menggunakan *Brute Force* Per Wilayah (34 kasus yang ditinjau)

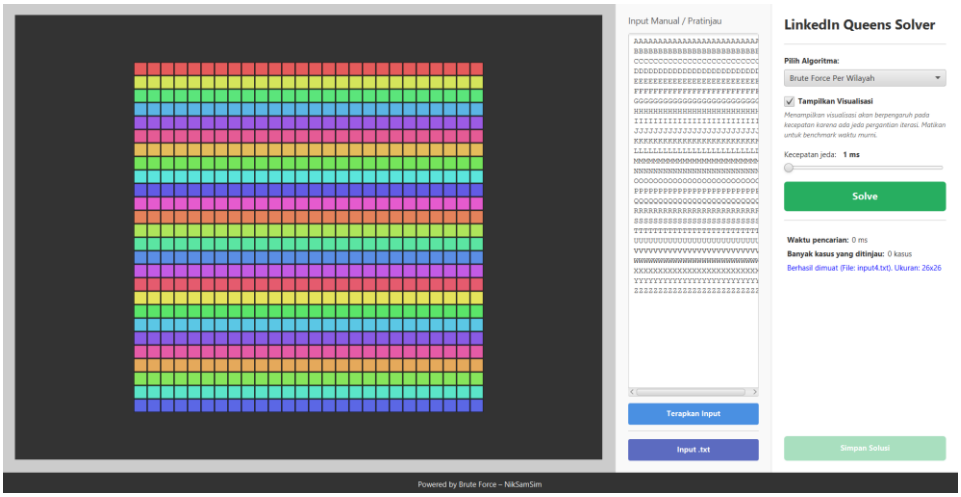


Terlihat perbedaan jumlah kasus yang ditinjau oleh kedua algoritma.

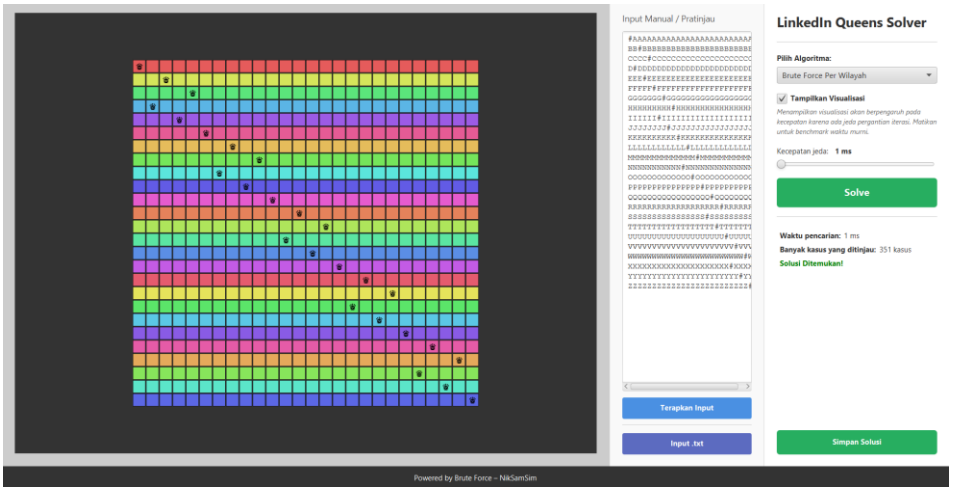
5.2. Input File `.txt`

- input4.txt

  Input:

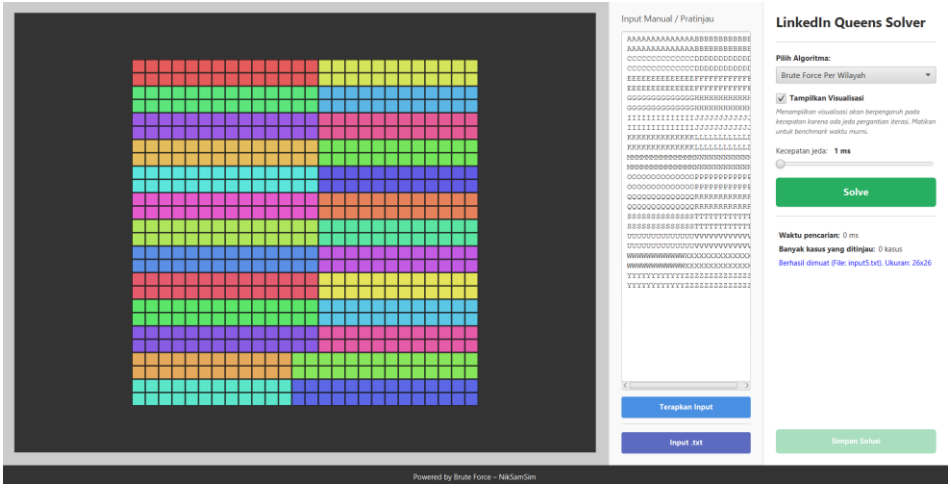


  Hasil:

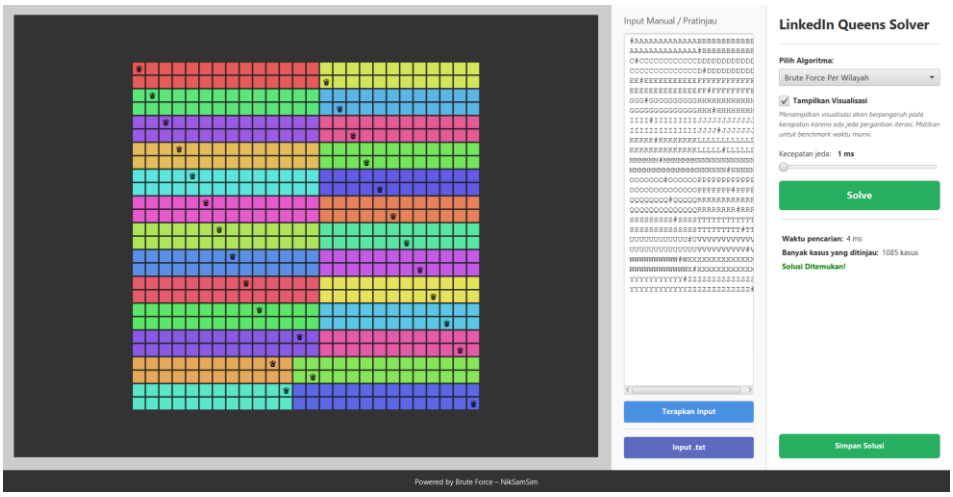


- input5.txt

  Input:

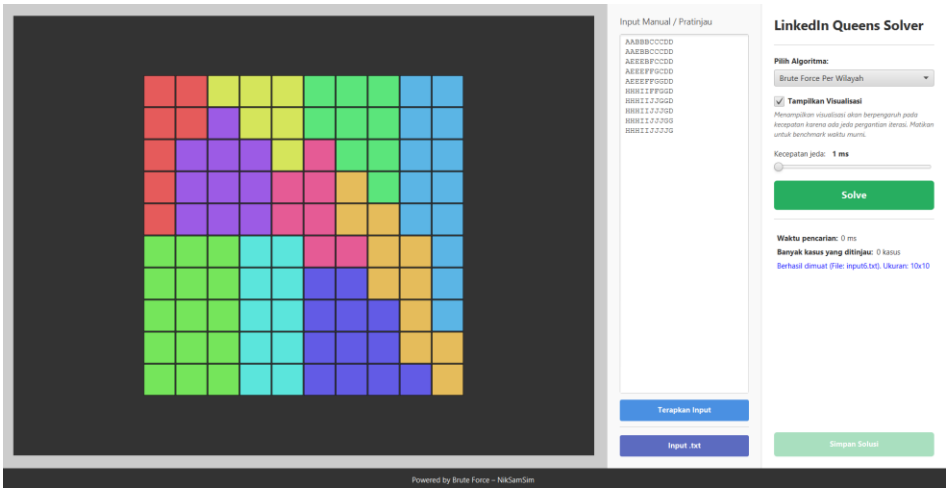


Hasil:

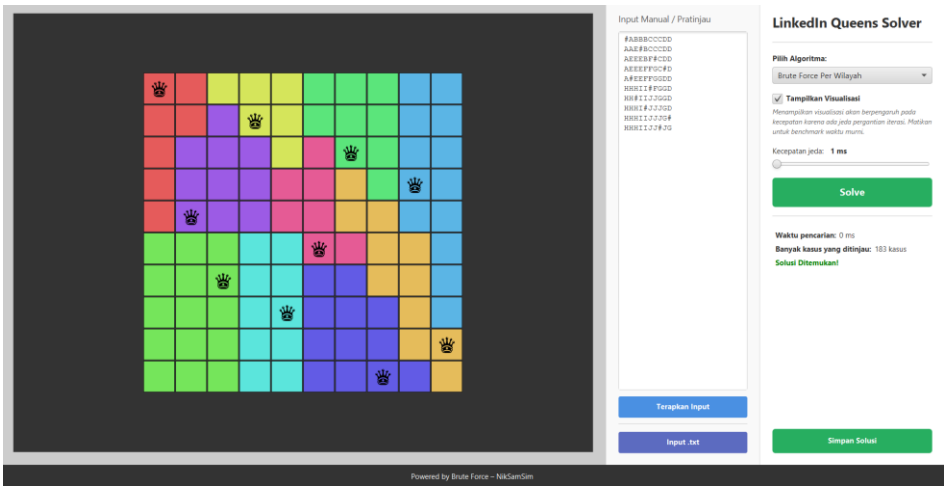


- input6.txt

Input:



Hasil:



6. KESIMPULAN

Implementasi program ini membuktikan bahwa *brute force* bukan sekadar metode teoritis yang lambat, melainkan sebuah solusi deterministik yang menjamin kebenaran. *Brute force* memeriksa opsi secara pasti. Keberhasilan program dalam menyelesaikan papan menunjukkan bahwa *brute force* dapat diandalkan sepenuhnya untuk kasus ini, asalkan aturan permainan diterapkan secara ketat di setiap langkah penelusuran. Implementasi ini juga membuktikan bahwa *brute force* menjamin ditemukannya solusi yang benar apabila solusi tersebut memang ada.

Implementasi program ini juga membuktikan bahwa algoritma *backtracking* mampu memangkas ruang pencarian secara signifikan dengan memanfaatkan batasan satu *queen* per wilayah dan mekanisme *pruning*, sehingga solusi untuk papan $N <= 26$ dapat ditemukan dalam hitungan detik. Sebaliknya, algoritma kombinasi petak memiliki kompleksitas waktu yang tumbuh secara faktorial sehingga tidak cocok untuk ukuran papan yang besar.

7. REFERENSI

- https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/02-Algoritma-Brute-Force-(2026)-Bag1.pdf
- https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/03-Algoritma-Brute-Force-(2026)-Bag2.pdf
- https://www.geeksforgeeks.org/java/javafx-tutorial/
- https://www.youtube.com/watch?v=FLkOX4Eez6o&list=PL6gx4Cwl9DGBzfXLWLSYVy8EbTdpGbUIG
- https://stackoverflow.com/questions/41287372/how-to-take-snapshot-of-selected-area-of-screen-in-javafx
- https://openjfx.io/javadoc/21/index.html
- https://www.youtube.com/watch?v=xnzPxDNbMLo

No	Poin	Ya	Tidak
1	Program berhasil di kompilasi tanpa kesalahan	✓	
2	Program berhasil di jalankan	✓	
3	Solusi yang diberikan program benar dan mematuhi aturan permainan	✓	
4	Program dapat membaca masukan berkas .txt serta menyimpan solusi dalam berkas .txt	✓	
5	Program memiliki Graphical User Interface (GUI)	✓	
6	Program dapat menyimpan solusi dalam bentuk file gambar	✓	

Tugas ini disusun sepenuhnya tanpa bantuan kecerdasan buatan (Generative AI), melainkan hasil pemikiran dan analisis mandiri.

Niko Samuel Simanjuntak