Laporan Tugas Kecil 1

IF2211 Strategi Algoritma

Penyelesaian Permainan Queens Linkedin

Semester II / 2025–2026

Disusun oleh:

Nicholas Wise Saragih Sumbayak

Teknik Informatika

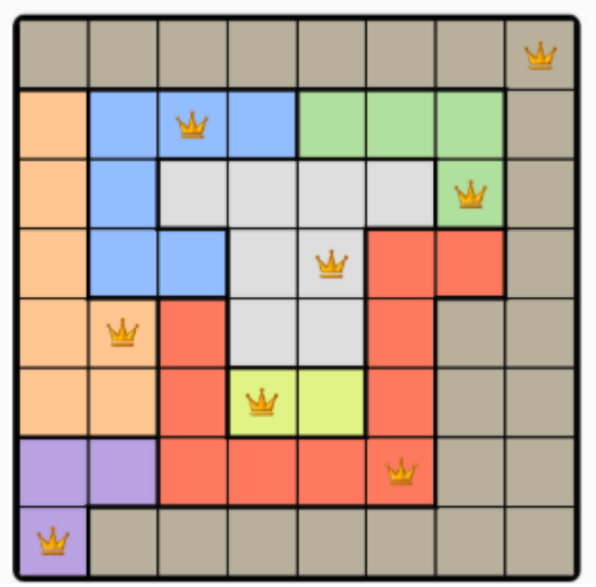
13524037

Laboratorium Ilmu dan Rekayasa Komputasi
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung

Daftar Isi

1 Pendahuluan

Queens adalah gim logika yang tersedia pada situs jejaring profesional LinkedIn. Tujuan dari gim ini adalah menempatkan queen pada sebuah papan persegi berwarna 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.



Gambar 1: Contoh papan permainan LinkedIn Queens

Tujuan dari Tugas Kecil 1 ini adalah untuk 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*.

2 Dasar Teori

2.1 Algoritma *Brute Force*

2.1.1 Definisi *Brute Force*

Algoritma *brute force* adalah algoritma yang menyelesaikan sebuah masalah secara lempang (*straightforward*). Pada dasarnya , algoritma *brute-force* menghasilkan sebuah solusi melalui langkah-langkah yang sederhana, langung, dan jelas. Algoritma ini menjamin menemukan solusi jika solusi tersebut ada, namun memiliki kompleksitas waktu yang tinggi karena harus mengeksplorasi seluruh ruang pencarian.

Karakteristik algoritma *brute force* antara lain:

- **Tidak efisien/sangkil** karena membutuhkan tenaga komputasi yang besar dan waktu yang lama.
- **Deterministik**, yang berarti selalu menghasilkan hasil yang sama untuk input yang sama.

- **Implementasi yang sederhana**, sehingga mudah dipahami dan diimplementasikan.

2.1.2 Algoritma *Heuristic Search*

Algoritma *Heuristic Search* termasuk dalam kategori algoritma *brute-force*. Cara kerja *heuristic search* sangat lempang (seperti *brute-force* pada umumnya): Mencoba semua kemungkinan solusi hingga menemukan solusi yang benar. Umumnya, algoritma ini digunakan untuk menyelesaikan permasalahan dalam lingkup kombinatorika, yang melibatkan solusi berbentuk sebuah permutasi, kombinasi, atau himpunan.

Inti dari *exhaustic search* dapat dibagi menjadi tiga langkah, yaitu:

1. **Enumerasi** *list* setiap kemungkinan solusi,
2. **Evaluasi** semua kemungkinan solusi masing-masing,
3. **Simpan solusi terbaik** selama keberjalanan *search*, dan
4. Tentukan solusi terbaik (***the winner***).

2.1.3 Teknik Heuristik

Algoritma *brute-force* seperti *heuristic search* dapat ditingkatkan efisiensinya dengan mengeliminasi calon-calon solusi yang belum dihasilkan sepenuhnya namun dapat dipastikan salah. Salah satu metode dalam menentukan calon solusi non-valid tersebut adalah menggunakan fungsi heuristik.

Heuristik merupakan teknik yang dirancang untuk memecahkan persoalan dengan mengabaikan apakah teknik tersebut terbukti benar secara matematis. Eliminasi kasus oleh heuristik menggunakan 2 dasar utama, yaitu **intuisi** yang didasari akal sehat/*common sense* dan pembelajaran pada **pengalaman** dari solusi-solusi sebelumnya.

Perlu ditekankan bahwa heuristik berbeda dengan algoritma. Algoritma berbentuk langkah-langkah konkrit yang digunakan untuk menyelesaikan sebuah permasalahan, sedangkan heuristik adalah sebuah pedoman/*rule of thumb* yang diikuti untuk menyederhanakan pembuatan keputusan.

2.2 Permainan LinkedIn Queens

2.2.1 Perumusan Permasalahan

LinkedIn Queens adalah variasi dari permasalahan klasik *N-Queens* dengan tambahan constraint berupa wilayah berwarna (*colored regions*). Pada awal permainan, pemain diberikan papan persegi dengan ukuran $n \times n$, n buah daerah yang diberikan warna masing-masing, dan beberapa ratu yang telah diletakkan sebelumnya.

Seperti permasalahan *N-Queens*, tujuan akhir dari LinkedIn Queens adalah untuk menemukan susunan peletakan n buah *Queen* yang tepat sehingga tidak ada yang saling menyerang. Perbedaannya terletak pada *constraint* yang menyatakan 2 atau lebih buah *Queen* tidak saling menyerang:

1. Tepat satu queen pada setiap **baris**.
2. Tepat satu queen pada setiap **kolom**.
3. Tepat satu queen pada setiap **warna**.
4. Queen tidak boleh **bersebelahan secara diagonal**.

2.2.2 Perumusan Solusi

Suatu konfigurasi penempatan queen dianggap valid jika memenuhi kondisi berikut:

1. **Tidak ada serangan antar queen**
   - Tidak ada dua queen pada kolom atau baris yang sama
   - Tidak ada dua queen yang bersebelahan secara diagonal (Jarak Manhattan = 1)
2. **Satu queen per wilayah**

   Setiap region warna memiliki tepat satu queen. Jika terdapat wilayah tanpa queen atau wilayah dengan lebih dari satu queen, solusi tidak valid.

Berdasarkan kedua kondisi di atas, fungsi validasi dapat dinyatakan sebagai: [isValid(S) = ¬hasAttacks(S)∧onePerRegion(S)(0) dimana S adalah himpunan posisi queen, `hasAttacks`(S) mengecek adanya serangan antar queen, dan `onePerRegion`(S) memverifikasi distribusi queen pada region.

3 Desain dan Implementasi

3.1 Arsitektur dan Struktur Proyek

3.1.1 Penerapan Pola MVC

Program ini menerapkan arsitektur **Model-View-Controller (MVC)** untuk memisahkan tanggung jawab setiap komponen. Struktur package dalam proyek mencerminkan pemisahan ini dengan jelas: package `model/` berisi representasi data dan logika domain, package `view/` menangani presentasi GUI, dan package `controller/` mengatur alur aplikasi dan koordinasi antar komponeb. Selain tiga komponen MVC utama, terdapat juga package `service/` yang mengenkapsulasi logika bisnis kompleks seperti algoritma solver dan operasi file I/O, serta package `util/` untuk fungsi-fungsi pembantu.

3.1.2 Build System dengan Gradle

Proyek ini menggunakan **Gradle** sebagai build automation tool dan dependency manager. Gradle dipilih karena kemampuannya mengelola dependensi secara deklaratif dan efisiensi dalam menjalankan perintah *build*. File `build.gradle` mendefinisikan konfigurasi proyek termasuk versi Java yang digunakan (Java 21), dependensi eksternal seperti JavaFX, dan task untuk kompilasi serta distribusi. Gradle Wrapper (`gradlew` dan `gradlew.bat`) disertakan dalam repository sehingga pengguna tidak perlu menginstal Gradle secara manual.

3.1.3 Struktur Direktori

Repository diorganisir mengikuti konvensi standar Maven/Gradle untuk proyek Java. Direktori `src/main/java/` berisi seluruh source code aplikasi yang dikelompokkan dalam package `queens` dengan sub-package sesuai pola MVC. Direktori `src/main/resources/` menyimpan resource non-code seperti file FXML untuk layout GUI. Direktori `test/` berisi file-file test case dengan subdirektori `inputs/` untuk file input papan dan `outputs/` untuk

menyimpan hasil solusi. Direktori `build/` (di-generate oleh Gradle) menyimpan hasil kompilasi.

3.2 Komponen Model

3.2.1 Class Board

Class `Board` merepresentasikan papan permainan dengan region warna yang menjadi fondasi dari permasalahan Queens. Struktur data utama berupa matriks dua dimensi bertipe `char[][]` menyimpan karakter yang merepresentasikan region warna di setiap sel papan. Untuk efisiensi akses, class ini juga memelihara `Map<Character, List<Position>>` yang memetakan setiap region ke daftar posisi-posisinya.

Board dapat dikonstruksi dari `List<String>` yang dibaca dari file input, dimana setiap string merepresentasikan satu baris papan. Class ini menyediakan method untuk mengakses wilayah pada posisi tertentu, mendapatkan daftar semua region unik, dan mengakses grid region secara keseluruhan, sehingga enkapsulasi representasi internal papan terjaga dengan baik.

Listing 1: Board.java

```java
public class Board {
  private final char[][] regions; // Grid region warna
  private final int size; // Ukuran papan (n x n)
  private final Map<Character, List<Position>> regionMap;

  public Board(List<String> lines) {
  }

  public char getRegion(Position pos) {
    return regions[pos.getRow()][pos.getCol()];
  }

  public List<Character> getRegions() {
    return new ArrayList<>(regionMap.keySet());
  }
}
```

3.2.2 Class Position

Class `Position` adalah value object immutable yang merepresentasikan koordinat pada papan dengan atribut `row` dan `col`. Selain sebagai container data, class ini juga mengenkapsulasi logika untuk mendeteksi serangan antar queen melalui method `attacks()`. Method ini mengimplementasikan aturan permainan dengan memeriksa apakah dua posisi berada pada kolom yang sama atau bersebelahan secara diagonal (jarak Manhattan = 1).

Serangan baris tidak perlu dicek karena algoritma brute force yang digunakan menjamin hanya satu queen per baris. Class ini mengoverride method `equals()` dan `hashCode()` untuk memastikan dua posisi dengan koordinat sama dianggap identik, yang penting untuk operasi collections seperti pengecekan duplikasi.

Listing 2: Position.java

```java
public class Position {
  private final int row;
  private final int col;

  // Memeriksa penyerangan antar Queen
  public boolean attacks(Position other) {
    if (this.equals(other)) return false;

    if (this.col == other.col) return true;

    if (Math.abs(this.row - other.row) == 1 &&
        Math.abs(this.col - other.col) == 1) {
      return true;
    }
    return false;
  }
}
```

3.2.3 Class Solution dan SolutionStats

Class `Solution` mengenkapsulasi hasil pencarian dengan menyimpan list posisi-posisi queen yang membentuk solusi valid beserta statistik pencariannya dalam bentuk objek `SolutionStats`. Pemisahan ini memungkinkan solusi dapat di-pass antar komponen tanpa kehilangan informasi penting tentang proses pencariannya.

Class `SolutionStats` berfungsi sebagai data transfer object yang menyimpan metadata pencarian: waktu yang diperlukan dalam milidetik (`elapsedTime`), jumlah konfigurasi yang telah dievaluasi (`casesReviewed`), dan flag boolean yang mengindikasikan apakah solusi ditemukan atau tidak (`found`). Informasi statistik ini berguna untuk analisis performa algoritma dan memberikan feedback kepada pengguna tentang kompleksitas pencarian yang dilakukan.

Listing 3: Class Solution dan SolutionStats

```java
public class Solution {
  private final List<Position> queenPositions;
  private final SolutionStats stats;
}

public class SolutionStats {
  private final long elapsedTime;      // Waktu pencarian (ms)
  private final int casesReviewed;     // Jumlah kasus dicek
  private final boolean found;         // Solusi ditemukan?
}
```

3.3 Komponen Service

3.3.1 SolverService

Class `SolverService` mengimplementasikan algoritma *brute force* untuk mencari solusi permasalahan Queens. Algoritma ini bekerja dengan melakukan enumerasi sistematis terhadap semua kemungkinan penempatan n queen pada papan berukuran $n \times n$. Untuk efisiensi enumerasi, algoritma membatasi pencarian dengan menempatkan tepat satu queen pada setiap baris, sehingga ruang pencarian berkurang dari n^{2n} menjadi n^n. Array `columns[]` digunakan untuk merepresentasikan konfigurasi, dimana `columns[i]` menyimpan nomor kolom tempat queen pada baris i berada.

Class ini menyediakan dua overload method `solve()`: satu tanpa parameter tambahan untuk mode CLI, dan satu dengan parameter `ProgressCallback` untuk mendukung update GUI secara real-time:

Listing 4: Method solve() di SolverService

```
public Solution solve(Board board) {
  return solve(board, null);
}

public Solution solve(Board board, ProgressCallback
    progressCallback) {
  int size = board.getSize();
  int cases = 0;
  long startTime = System.currentTimeMillis();

  // Generate all n^n cases, store column number for queen in
      row i in column[i]
  int[] columns = new int[size];

  while (true) {
    // Start with 1 Queen in each row
    List<Position> solution = new ArrayList<>();
    for (int row = 0; row < size; row++) {
      solution.add(new Position(row, columns[row]));
    }

    cases++;

    // Progress callback for GUI updates (every 10,000
       iterations)
    if (progressCallback != null && cases % 10000 == 0) {
      progressCallback.onProgress(cases, solution);
    }

    if (isValidSolution(solution, board)) {
      long elapsedTime = System.currentTimeMillis() -
         startTime;
      SolutionStats stats = new SolutionStats(elapsedTime,
         cases, true);
      return new Solution(solution, stats);
    }
```

```
32
33      // Generate new solutions
34      int row = size - 1;
35      while (row >= 0 && columns[row] == size - 1) {
36        columns[row] = 0;
37        row--;
38      }
39
40      if (row < 0) {
41        break;
42      }
43
44      columns[row]++;
45    }
46
47    // No solution found
48    return null;
49  }
```

3.3.2 FileService

Class `FileService` menangani semua operasi input/output file dalam aplikasi. Method `readBoardFromFile()` bertanggung jawab melakukan parsing file teks yang berisi representasi papan menjadi objek `Board`, dengan validasi format dan error handling yang sesuai. Method `saveSolutionToFile()` menyimpan solusi yang ditemukan ke file output dalam format yang mudah dibaca, mencakup visualisasi papan dengan penempatan queen dan informasi statistik pencarian.

Untuk kemudahan penggunaan, class ini juga mengimplementasikan path resolution otomatis: jika pengguna hanya memberikan nama file tanpa path lengkap, file input akan dicari di direktori `test/inputs/` dan output akan disimpan ke `test/outputs/`. Pendekatan ini mengurangi kebutuhan pengguna untuk mengetik path lengkap sambil tetap mendukung absolute path untuk fleksibilitas.

3.4 Komponen View

3.4.1 QueensGUI

Class `QueensGUI` berfungsi sebagai entry point JavaFX Application untuk mode GUI. `QueensGUI` bertanggung jawab atas:

- Inisialisasi aplikasi GUI dengan memuat layout FXML
- Membuat `Scene` dari `FXMLLoader`
- Konfigurasi dan menampilkan `Stage` utama aplikasi
- Menerima dan memproses command-line arguments jika ada
- Memisahkan definisi UI (FXML) dari inisialisasi runtime

3.4.2 BoardCanvas

`BoardCanvas` adaklah component View untuk rendering papan dan queen secara programatik dan dinamis. Tugasnya, antara lain:

- Rendering region warna dengan `GraphicsContext`
- Menghitung dan menggambar grid lines sebagai pemisah sel
- Visualisasi papan dengan warna berbeda per region
- Rendering simbol queen pada posisi solusi
- Adaptasi ukuran sel berdasarkan dimensi canvas
- Custom drawing dengan kontrol penuh atas rendering

3.5 Komponen Controller

3.5.1 CLIController

`CLIController` berfungsi sebagai controller untuk mode prototype *command-line interface*, yang kemudian disesuaikan untuk logging selama keberjalanan aplikasi dalam bentuk GUI. Tugasnya, antara lain:

- Parsing argumen command-line atau prompt input dari pengguna
- Koordinasi dengan `FileService` untuk membaca file input
- Memanggil `SolverService` untuk eksekusi pencarian solusi
- Menampilkan hasil ke console dengan format user-friendly
- Menawarkan opsi penyimpanan hasil ke file output

3.5.2 MainController

`MainController` adalah controller utama untuk GUI JavaFX yang dilengkapi dengan FXML binding. MainController bertanggung jawab atas:

- Binding dengan komponen UI melalui annotation `@FXML`
- Event handling untuk pemilihan file dengan `FileChooser`
- Implementasi multithreading untuk solver menggunakan `Task`
- Progress update real-time dengan `Platform.runLater()`
- Koordinasi rendering hasil solusi pada canvas
- Menampilkan statistik pencarian di GUI

3.6 Analisis Algoritma

3.6.1 Cara Kerja Algoritma

Algoritma bekerja menggunakan prinsip *generate-and-test* dengan strategi enumerasi berbasis counting. Proses dimulai dengan inisialisasi array `columns[]` dengan nilai 0, merepresentasikan semua queen pada kolom paling kiri. Pada setiap iterasi, algoritma melakukan tiga langkah utama:

1. **Penghasilan kandidat solusi**

   Algoritma membentuk list `Position` dari representasi array, dimana posisi queen pada baris i adalah $(i, \text{columns}[i])$. Proses ini dilakukan dalam waktu $O(n)$.

2. **Uji validitas solusi**

   Setiap kandidat divalidasi dengan memeriksa:

   - **Tidak ada serangan**: Dilakukan pengecekan pairwise antara semua queen untuk memastikan tidak ada yang berada pada kolom sama atau diagonal bersebelahan. Validasi ini memerlukan $O(n^2)$ perbandingan.
   - **Satu queen per region**: Menggunakan `HashMap` untuk menghitung jumlah queen pada setiap region dalam waktu $O(n)$, kemudian memverifikasi bahwa semua region memiliki tepat satu queen.

3. **Inkrementasi ke kandidat berikutnya**

   Jika kandidat current tidak valid, algoritma melakukan inkrementasi pada array `columns[]`, mengikuti prinsip counting dalam basis n. Dimulai dari indeks paling kanan, algoritma mencari posisi pertama yang nilainya belum mencapai $n-1$, mengembalikan semua posisi di sebelah kanannya ke 0, lalu melakukan *increment* pada posisi tersebut. Jika tidak ditemukan posisi yang dapat diinkrementasi (semua bernilai $n-1$), maka seluruh ruang pencarian telah dieksplorasi.

   Proses enumerasi ini menjamin completeness karena counting sistematis menghasilkan semua permutasi yang mungkin tanpa duplikasi atau missing configuration.

3.6.2 Fungsi Validasi

Fungsi validasi memeriksa dua kondisi utama yang harus dipenuhi oleh setiap kandidat solusi, yaitu tidak ada *Queen* yang saling menyerang (yang dipenuhi oleh method `isValidSolution`) dan hanya terdapata 1 *Queen* pada tiap wilayah/region (yang dipenuhi oleh `hasOnePerRegion`).

Listing 5: Fungsi validasi constraint

```
1 private boolean isValidSolution(List<Position> queenPositions
     , Board board) {
2   if (hasAttacks(queenPositions)) {
3     return false;
4   }
5   return hasOnePerRegion(queenPositions, board);
6 }
7
8 private boolean hasAttacks(List<Position> queenPositions) {
```

```
    for (int i = 0; i < queenPositions.size(); i++) {
      for (int j = i + 1; j < queenPositions.size(); j++) {
        if (queenPositions.get(i).attacks(queenPositions.get(j)
            )) {
          return true;
        }
      }
    }
    return false;
}

private boolean hasOnePerRegion(List<Position> queenPositions
    , Board board) {
    Map<Character, Integer> regionCounts = new HashMap<>();

    for (Position queen : queenPositions) {
      char region = board.getRegion(queen);
      regionCounts.put(region, regionCounts.getOrDefault(region
          , 0) + 1);
    }

    // Check for only 1 queen per region
    List<Character> regions = board.getRegions();
    if (regionCounts.size() != regions.size()) {
      return false;
    }

    for (int count : regionCounts.values()) {
      if (count != 1) {
        return false;
      }
    }

    return true;
}
```

3.6.3 Kompleksitas Waktu Algoritma

Algoritma mengeksplorasi hingga n^n konfigurasi berbeda. Nilai ini berasal dari fakta bahwa terdapat n pilihan kolom untuk setiap n baris, dan dengan menempatkan tepat satu queen per baris, total kombinasi yang mungkin adalah:

$$\underbrace{n \times n \times n \times \cdots \times n}_{n \text{ faktor}} = n^n$$

Sebagai contoh, untuk papan 5×5, ruang pencarian memiliki $5^5 = 3,125$ konfigurasi. Untuk papan 10×10, terdapat $10^{10} \approx 10$ miliar konfigurasi.

Setiap kali program dijalankan, algoritma melaksanakan:

- **Penghasilan calon solusi** dengan kompleksitas $O(n)$ untuk membuat list posisi.
- **Validasi serangan** dengan kompleksitas $O(n^2)$ untuk pengecekan pairwise semua queen.
- **Validasi *Queen* pada setiap wilayah** dengan kompleksitas $O(n)$ untuk counting dengan HashMap.
- **Inkrementasi array konfigurasi** dengan kompleksitas $O(n)$ (*worst case*) untuk reset dan increment.

Computational Cost per iterasi adalah $O(n^2)$ dari validasi serangan, sehingga kompleksitas waktu total adalah:

$$T(n) = O(n^n \cdot n^2)$$

Kompleksitas eksponensial ini membatasi ukuran papan yang dapat diselesaikan dalam waktu wajar. Tabel berikut menunjukkan estimasi jumlah operasi:

n	**Konfigurasi (n^n)**	**Estimasi Operasi**
4	256	$\sim 4 \times 10^3$
5	$3,125$	$\sim 8 \times 10^4$
6	$46,656$	$\sim 2 \times 10^6$
7	$823,543$	$\sim 4 \times 10^7$
8	$16,777,216$	$\sim 10^9$
10	10^{10}	$\sim 10^{12}$

Tabel 1: Pertumbuhan ruang pencarian algoritma brute force

3.6.4 Kompleksitas Ruang Algoritma

Kompleksitas ruang adalah $O(n^2)$, dengan rincian alokasi memori utama sebagai berikut:

- **Matriks *board***: $O(n^2)$ untuk menyimpan grid region warna
- **Struktur data map untuk region warna**: $O(n^2)$ worst case jika setiap sel adalah region berbeda
- **Array `columns[]`**: $O(n)$ untuk representasi konfigurasi current
- **List kandidat sementara**: $O(n)$ untuk list posisi temporary per iterasi
- **HashMap validasi**: $O(k)$ dimana $k \leq n$ adalah jumlah region unik

Dominasi penggunaan memori adalah struktur data `board` yang memerlukan penyimpanan proporsional dengan luas papan, sehingga kompleksitas ruang algoritma ini adalah:

$$S(n) = O(n^2)$$

4 Eksperimen dan Analisis

4.1 Test Case 1: Papan 2×2

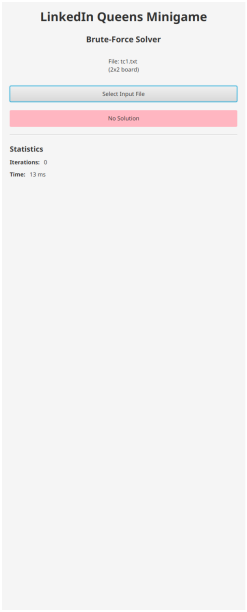
4.1.1 Input

File `tc1.txt` berisi papan berukuran 2×2 dengan 2 wilayah warna (`A` dan `B`):

Listing 6: tc1.txt

```
AB
BA
```

4.1.2 Output



Gambar 2: Output program untuk test case 1 (papan 2×2)

4.1.3 Analisis

Pada papan 2×2 dengan 2 wilayah, tidak ada konfigurasi penempatan 2 queen yang memenuhi seluruh constraint. Setiap penempatan queen pada baris pertama akan menyebabkan queen pada baris kedua berada pada kolom yang sama atau bersebelahan secara diagonal. Program dengan benar menampilkan pesan *"No Solution"*.

4.2 Test Case 2: Papan 5×5

4.2.1 Input

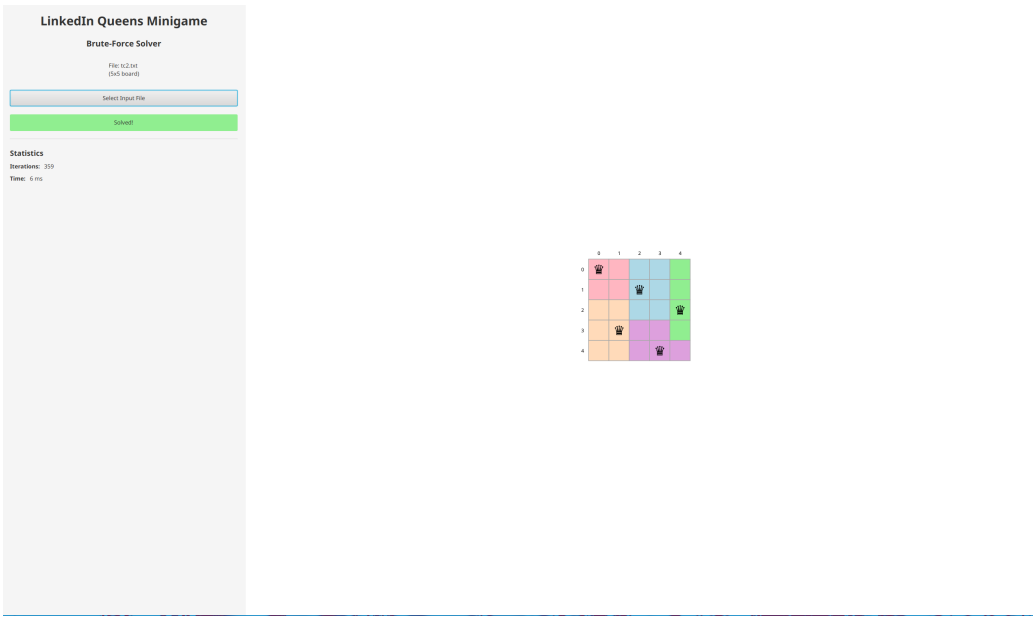
File `tc2.txt` berisi papan berukuran 5×5 dengan 5 wilayah warna (`A`–`E`):

Listing 7: tc2.txt

```
AABBC
AABBC
DDBBC
```

```
DDEEC
DDEEE
```

4.2.2 Output



Gambar 3: Output program untuk test case 2 (papan 5×5)

4.2.3 Analisis

Program berhasil menemukan solusi valid pada papan 5×5. Setiap queen ditempatkan pada baris, kolom, dan wilayah yang berbeda, serta tidak ada dua queen yang bersebelahan secara diagonal. Ruang pencarian sebesar $5^5 = 3.125$ konfigurasi dapat diselesaikan dalam waktu sangat singkat (di bawah 6 ms) dalam 350 iterasi.

4.3 Test Case 3: Papan 7×7

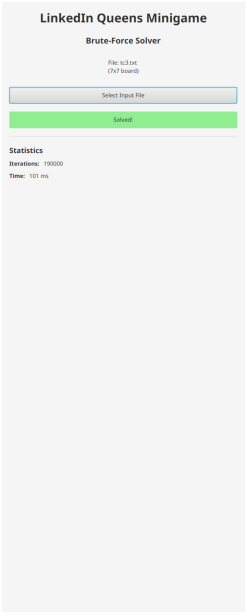
4.3.1 Input

File `tc3.txt` berisi papan berukuran 7×7 dengan 7 wilayah warna (`A`–`G`):

Listing 8: tc3.txt

```
AAABBBC
AACBBBC
AACDDBC
EECDDBC
EECDDFG
EEEDDFF
EEEDDFF
```

4.3.2 Output



Gambar 4: Output program untuk test case 3 (papan 7×7)

4.3.3 Analisis

Program berhasil menemukan solusi valid pada papan 7×7. Ruang pencarian sebesar $7^7 = 823.543$ konfigurasi membutuhkan waktu yang lebih lama dibandingkan TC2, namun masih dalam rentang waktu yang wajar (di sekitar 1 detik) dalam 190.000 iterasi. Hasil ini menunjukkan pertumbuhan waktu eksekusi yang signifikan seiring bertambahnya ukuran papan.

4.4 Test Case 4: Papan 7×7 dengan Input Tidak Valid

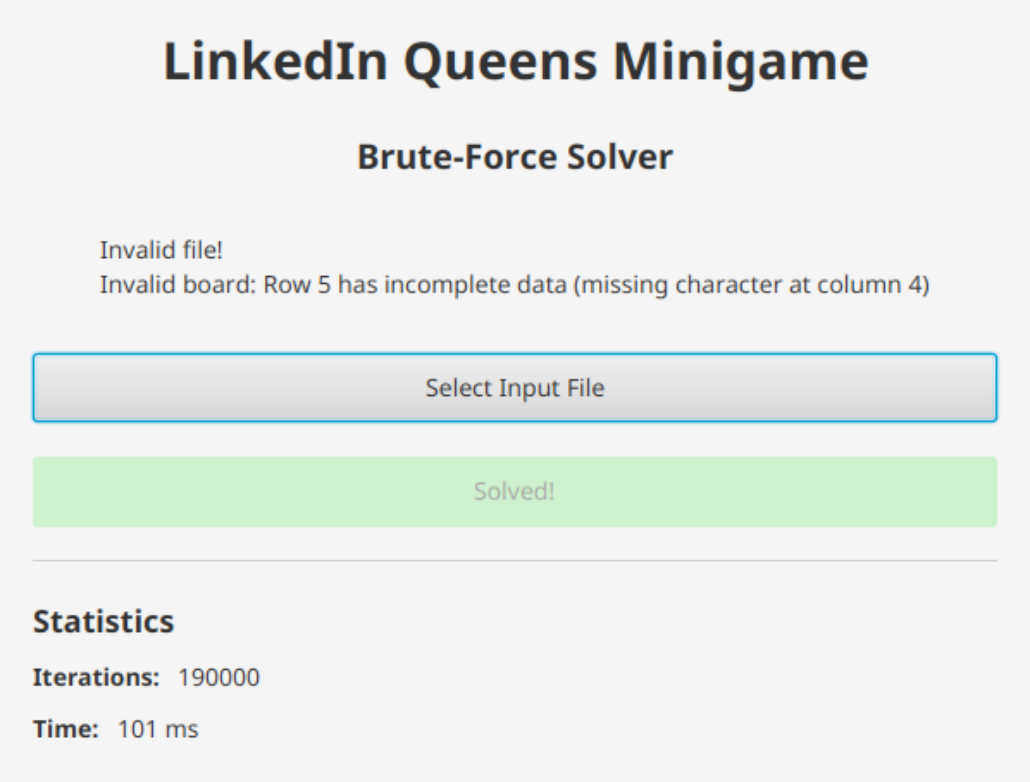
4.4.1 Input

File `tc4.txt` berisi papan berukuran 7×7 namun baris ke-5 hanya memiliki 3 karakter (tidak lengkap):

Listing 9: tc4.txt

```
AAABBBC
AACBBBC
AACDDBC
EECDDBC
EEC
EEEDDFF
EEEDDFF
```

4.4.2 Output



Gambar 5: Output program untuk test case 4 (input tidak valid)

4.4.3 Analisis

Program mendeteksi bahwa baris ke-5 memiliki panjang yang tidak konsisten dengan baris-baris lainnya (3 karakter alih-alih 7). `FileService` memvalidasi format input sebelum memulai pencarian dan menampilkan pesan error yang informatif kepada pengguna.

4.5 Test Case 5: Papan 9×9

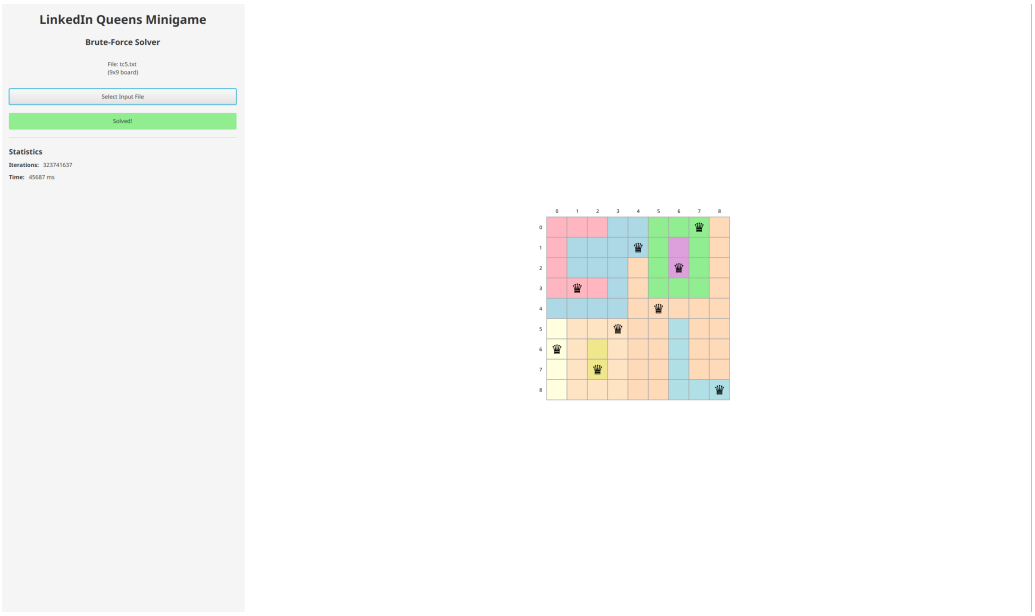
4.5.1 Input

File `tc5.txt` berisi papan berukuran 9×9 dengan 9 wilayah warna (`A`–`I`):

Listing 10: tc5.txt

```
AAABBCCCD
ABBBBCECD
ABBBDCECD
AAABDCCCD
BBBBDDDDD
FGGGDDHDD
FGIGDDHDD
FGIGDDHDD
FGGGDDHHH
```

4.5.2 Output



Gambar 6: Output program untuk test case 5 (papan 9×9)

4.5.3 Analisis

Program berhasil menemukan solusi valid pada papan 9×9. Ruang pencarian sebesar $9^9 \approx 387$ juta konfigurasi membutuhkan waktu yang jauh lebih lama dibandingkan TC3, yaitu 45687 ms dalam 323741637 iterasi. Hasil ini mengkonfirmasi pertumbuhan kompleksitas eksponensial $O(n^n)$ dari algoritma *brute force*.

4.6 Test Case 6: Papan 10×10

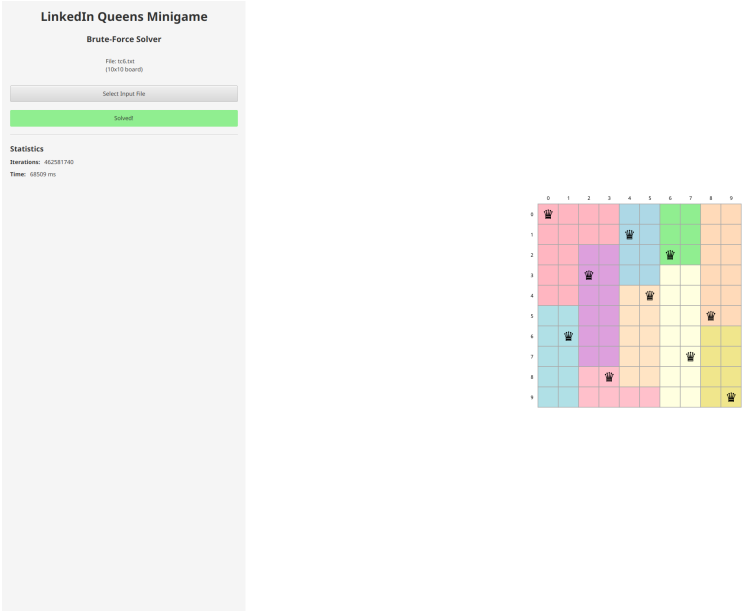
4.6.1 Input

File `tc6.txt` berisi papan berukuran 10×10 dengan 10 wilayah warna (`A`–`J`):

Listing 11: tc6.txt

```
AAAABBCCDD
AAAABBCCDD
AAEEBBCCDD
AAEEBBFFDD
AAEEGGFFDD
HHEEGGFFDD
HHEEGGFFII
HHEEGGFFII
HHJJGGFFII
HHJJJJFFII
```

4.6.2 Output



Gambar 7: Output program untuk test case 6 (papan 10×10)

4.6.3 Analisis

Papan 10×10 memiliki ruang pencarian sebesar $10^{10} = 10$ miliar konfigurasi. Waktu eksekusi yang meningkat drastis dibandingkan TC5 menjadi 68509 ms dalam 462581740 iterasi menggambarkan batas praktis dari algoritma *brute force*. Hasil ini menunjukkan bahwa untuk papan berukuran besar, lebih ideal jika menggunakan teknik heuristik untuk menyederhanakan eliminasi calon solusi atau menggunakan algoritma yang lebih efisien seperti *backtracking*.

5 Kesimpulan

5.1 Ringkasan

1. **Temuan 1** — deskripsi temuan pertama dari eksperimen.
2. **Temuan 2** — deskripsi temuan kedua dari eksperimen.
3. **Temuan 3** — deskripsi temuan ketiga dari eksperimen.

5.2 Saran Pengembangan

1. Implementasi algoritma *backtracking* untuk meningkatkan efisiensi pencarian.
2. Penambahan heuristik seperti *constraint propagation* untuk mengurangi ruang pencarian.
3. Optimisasi multithreading untuk memanfaatkan multiple CPU cores.
4. Implementasi mode batch processing untuk menyelesaikan multiple puzzle secara bersamaan.

5.3 Refleksi dan Saran

Tugas ini keren! Jujur, ga *expect* kalo Tucil 1 ini bakal menggunakan permainan yang masih cukup relevan di zaman sekarang, dan lewat Tucil ini saya belajar banyak. Terima kasih aku sampaikan kepada asisten yang telah memberikan pertimbangan baik dalam menjawab dan merevisi spesifikasi yang tertera pada QnA!

Daftar Pustaka

[1] LinkedIn, "Queens Puzzle," 2024. [Online]. Available: https://www.linkedin.com/games/queens/.

[2] R. Munir, *Diktat Kuliah IF2211 Strategi Algoritma*. Bandung: Program Studi Teknik Informatika ITB, 2024.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA: MIT Press, 2009.

[4] Oracle, "JavaFX Documentation," 2024. [Online]. Available: https://openjfx.io/.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1994.

[6] Gradle Inc., "Gradle User Manual," 2024. [Online]. Available: https://docs.gradle.org/.

Appendix

Resource	Link
GitHub Repository	https://github.com/your-org/your-repo
Live Deployment	https://your-deployment-url.example.com

Tabel 2: Repository and deployment links