Laporan Tugas Kecil 1

IF2211 Strategi Algoritma

Penyelesaian Permainan Queens *Linkedin*

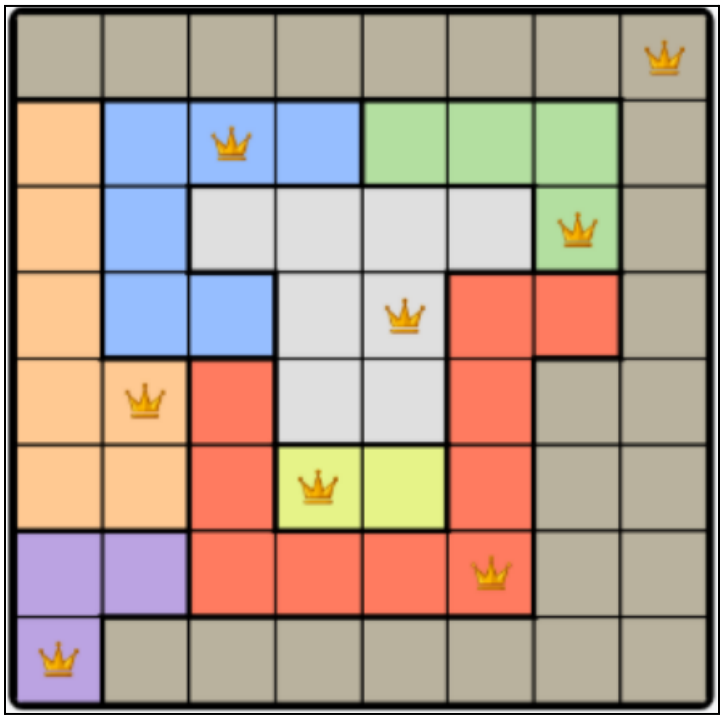
Semester II Tahun Ajaran 2025/2026

Disusun oleh:

IF 2024 - Jatinangor

Nama : **Dzakwan** Muhammad Khairan Putra Purnama

Kelas/NIM: K03/13524**145**



Laboratorium Ilmu dan Rekayasa Komputasi

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

Daftar Isi

1. Pendahuluan

Permainan logika dan teka-teki visual kini tidak hanya menjadi sarana hiburan, tetapi juga menjadi instrumen untuk mengasah kemampuan pemecahan masalah. Salah satu contoh yang populer saat ini adalah permainan "*Queens*" yang terdapat pada platform LinkedIn. Permainan ini menantang pemain untuk menempatkan sejumlah ratu (*queens*) pada papan sedemikian rupa sehingga tidak ada ratu yang saling menyerang, mirip dengan persoalan klasik *N-Queens*, namun dengan penambahan batasan wilayah warna (*color regions*) dan aturan ketetanggaan (*adjacency*) yang berbeda.

Gambar 1. Penyelesaian suatu teka-teki oleh sekumpulan orang.

Permasalahan utama dalam permainan Queens terletak pada pemenuhan berbagai kendala (*constraints*) secara bersamaan. Pemain harus memastikan bahwa dalam satu baris, satu kolom, dan satu wilayah warna hanya terdapat tepat satu ratu. Selain itu, aturan unik yang diterapkan adalah tidak boleh ada dua ratu yang bersentuhan, baik secara ortogonal maupun diagonal. Untuk menyelesaikan permasalahan kombinatorial seperti ini, diperlukan pendekatan algoritmik yang sistematis guna memeriksa kemungkinan solusi yang ada di ruang pencarian.

Dalam rangka memenuhi Tugas Kecil 1 mata kuliah IF2211 Strategi Algoritma, penulis mengembangkan sebuah perangkat lunak berbasis *Python* untuk menyelesaikan permainan ini secara otomatis. Program ini mengimplementasikan algoritma ***Brute Force*** yang bekerja secara sistematis dengan memeriksa setiap kemungkinan konfigurasi penempatan ratu hingga ditemukan solusi yang valid. Untuk memudahkan interaksi dan pemahaman, program ini dilengkapi dengan antarmuka grafis (GUI) menggunakan pustaka *Tkinter* yang mampu memvisualisasikan proses pencarian solusi, serta fitur pengelolaan berkas untuk membaca konfigurasi papan dan menyimpan hasil solusi.

Tujuan yang ingin dicapai adalah mengimplementasikan algoritma *Brute Force* secara mandiri untuk memecahkan persoalan kombinatorial, memvalidasi pemahaman terhadap struktur data dan aturan permainan yang kompleks, serta menyediakan alat bantu visual yang efektif. Dengan demikian, laporan dan program ini diharapkan dapat mendemonstrasikan bagaimana sebuah strategi algoritma sederhana dapat menyelesaikan teka-teki logika yang cukup rumit secara akurat dan efisien.

2. Dasar Teori

2.1. Permainan Queens dan Variasi Masalah N-Queens

Permainan *Queens* yang menjadi objek studi dalam tugas ini merupakan variasi modern dari persoalan klasik *N-Queens Problem*. Pada masalah klasik, tujuannya adalah menempatkan *N* buah ratu pada papan catur berukuran *N x N* tanpa ada dua ratu yang saling menyerang. Masalah ini tergolong dalam kelas *Constraint Satisfaction Problem* (CSP), di mana solusi didapatkan dengan menemukan keadaan (*state*) yang memenuhi sekumpulan variabel dan kendala yang didefinisikan secara spesifik (Russell & Norvig, 2020). Berbeda dengan versi klasik, permainan Queens pada LinkedIn memiliki kendala (*constraints*) tambahan yang lebih ketat, yaitu:

- Kendala Baris dan Kolom: Setiap baris dan setiap kolom harus berisi tepat satu ratu.
- Kendala Wilayah Warna (*Color Regions*): Papan permainan dibagi menjadi *N* wilayah warna yang berbeda. Setiap wilayah warna harus berisi tepat satu ratu.
- Kendala Ketetanggaan (*Adjacency*): Tidak ada dua ratu yang boleh ditempatkan pada petak yang bersebelahan, baik secara ortogonal (atas-bawah, kiri-kanan) maupun diagonal. Aturan ini sering disebut sebagai aturan "8 tetangga" atau *King's move constraint* dalam variasi catur.

Aturan ketetanggaan ini mempersempit ruang pencarian solusi dibandingkan catur konvensional, karena ratu tidak hanya menyerang sejauh langkahnya, tetapi juga memblokir petak di sekelilingnya secara langsung.

2.2. Algoritma Brute Force

Algoritma *Brute Force* adalah pendekatan pemecahan masalah yang bersifat langsung (*straightforward*) dan menyeluruh. Menurut Levitin (2012), algoritma ini sering disebut sebagai pendekatan *naive* atau *generate and test*, di mana sistem akan membangkitkan setiap kemungkinan kandidat solusi dan mengujinya apakah memenuhi pernyataan masalah atau tidak. Dalam konteks penyelesaian permainan *Queens*, algoritma *Brute Force* bekerja secara sistematis:

- Enumerasi: Membangkitkan setiap kemungkinan konfigurasi penempatan ratu pada papan. Karena setiap baris pasti berisi satu ratu, ruang pencarian dapat disederhanakan dengan merepresentasikan posisi ratu sebagai vektor $C = [c_1, c_2, ..., c_N]$, di mana c_i adalah posisi kolom ratu pada baris ke-*i*.
- Verifikasi: Setiap konfigurasi yang dibangkitkan diperiksa validitasnya terhadap seluruh aturan permainan (baris, kolom, warna, dan tetangga).

- Terminasi: Algoritma berhenti ketika solusi pertama yang valid ditemukan, atau ketika seluruh kemungkinan konfigurasi dalam ruang solusi telah diperiksa namun tidak ada yang memenuhi syarat.

Meskipun algoritma ini menjamin ditemukannya solusi jika solusi tersebut ada (*complete*), kelemahan utamanya adalah kompleksitas waktu yang tumbuh secara eksponensial seiring bertambahnya ukuran masukan *N*. Oleh karena itu, *Brute Force* murni umumnya hanya efektif untuk ukuran masalah yang kecil hingga menengah (Levitin, 2012) (M. Rinaldi 2026).

2.3. Struktur Data dan Antarmuka Grafis

Implementasi program membutuhkan struktur data yang efisien untuk merepresentasikan papan permainan. Papan *N x N* umumnya direpresentasikan sebagai matriks atau larik dua dimensi. Namun, untuk menyimpan posisi ratu, cukup digunakan larik satu dimensi di mana indeks merepresentasikan baris dan nilai merepresentasikan kolom, yang secara implisit memenuhi kendala "satu ratu per baris" (Russell & Norvig, 2020).

Untuk keperluan visualisasi, program menggunakan pustaka Tkinter. Tkinter adalah antarmuka standar Python untuk perangkat toolkit GUI Tcl/Tk, yang menyediakan mekanisme untuk menggambar elemen grafis primitif seperti persegi dan teks pada sebuah kanvas digital berdasarkan koordinat piksel (Python Software Foundation, 2026). Penggunaan GUI ini bertujuan untuk mempermudah pengguna dalam memvalidasi solusi yang dihasilkan oleh algoritma secara visual.

3. Implementasi

Pada bab ini dijelaskan detail implementasi algoritma yang digunakan dalam program. Program dibangun menggunakan bahasa *Python* dengan struktur modular yang terdiri dari validasi **board**, pengecekan validitas konfigurasi, dan algoritma pencarian solusi Brute Force.

3.1. Struktur Data

Pemilihan struktur data yang tepat sangat penting untuk menunjang efisiensi representasi ruang keadaan (*state space*). Dalam implementasi ini, digunakan dua struktur data utama:

- **Representasi Papan (*Board Matrix*)**
  Papan permainan direpresentasikan sebagai matriks dua dimensi (senarai bersarang atau *list of lists*) berukuran *N x N*. Setiap elemen matriks B[i][j] menyimpan karakter (tipe data *string*) yang merepresentasikan identitas wilayah warna (*color region*) pada sel tersebut (contoh: 'A', 'B', 'C').
- **Representasi Konfigurasi Ratu (*Configuration Vector*)**
  Alih-alih menggunakan matriks penuh untuk menyimpan posisi ratu, konfigurasi solusi direpresentasikan secara efisien menggunakan larik satu dimensi (array) C dengan panjang N. Nilai pada indeks ke-*i*, yakni C[i], menyatakan posisi kolom tempat ratu diletakkan pada baris ke-*i*. Pendekatan ini memberikan keuntungan implisit, yaitu menjamin bahwa tidak akan ada dua ratu yang menempati baris yang sama, sehingga algoritma tidak perlu memeriksa kendala baris secara eksplisit.

3.2. Algoritma Validasi Papan

Fungsi *validate_board* bertanggung jawab untuk memverifikasi integritas dan format data masukan sebelum proses pencarian solusi dijalankan. Validasi ini krusial untuk mencegah *runtime error* atau hasil yang tidak valid akibat masukan yang cacat. Proses validasi meliputi pemeriksaan dimensi matriks (harus persegi *N x N*), validitas karakter penyusun (huruf kapital 'A'-'Z'), serta memastikan jumlah wilayah warna unik (*distinct regions*) tepat berjumlah *N*. Berikut adalah notasi algoritma (*pseudocode*) untuk proses validasi papan:

```
function validate_board(board) -> boolean:
    n <- length(board)

    // Validasi 1: Board tidak boleh kosong
    if n == 0 then
        return False

    // Validasi 2: Setiap baris harus memiliki panjang N (matriks persegi)
    for row in board:
```

```
        if length(row) != n then
            return False

    regions <- empty Set

    // Validasi 3: Input hanya boleh karakter uppercase 'A'-'Z'
    for row in board:
        for cell in row:
            if cell < 'A' or cell > 'Z' then
                return False
            add cell to regions

    // Validasi 4: Jumlah region warna unik harus sama dengan N
    if length(regions) != n then
        return False

    return True
```

3.3. Algoritma Validasi Konfigurasi

Fungsi *is_valid_config* memegang peranan krusial dalam menjamin kebenaran solusi. Fungsi ini memeriksa apakah penempatan ratu pada array konfigurasi saat ini mematuhi seluruh aturan permainan Queens. Validasi dilakukan secara bertahap meliputi kendala kolom unik, kendala wilayah warna, dan kendala ketetanggaan (tidak boleh bersentuhan). Berikut adalah implementasi logika validasi dalam bentuk pseudocode:

```
function is_valid_config(board, queen_positions) -> boolean:
    n <- length(board)

    // 1. Cek Constraint Kolom
    // Menggunakan Set untuk memeriksa duplikasi nilai kolom
    if length(set(queen_positions)) != n then
        return False  // Terdapat kolom yang digunakan > 1 kali

    // 2. Cek Constraint Region Warna & Tetangga
    used_regions <- empty Set

    for r1 from 0 to n-1:
        c1 <- queen_positions[r1]

        // Ambil warna region pada posisi ratu saat ini
        region_color <- board[r1][c1]

        // Jika region warna sudah ditempati ratu lain, konfigurasi invalid
        if region_color in used_regions then
            return False

        add region_color to used_regions

        // Cek interaksi dengan seluruh ratu pada baris setelahnya (r2 >
r1)
        for r2 from r1+1 to n-1:
            c2 <- queen_positions[r2]
```

```
            // Hitung jarak vertikal dan horizontal
            delta_row <- abs(r1 - r2)
            delta_col <- abs(c1 - c2)

            // Constraint Tetangga: Ratu tidak boleh bersentuhan
            // (jarak baris <= 1 AND jarak kolom <= 1)
            if delta_row <= 1 and delta_col <= 1 then
                return False

    // Jika lolos semua pengecekan, konfigurasi valid
    return True
```

3.4. Algoritma Pencarian Solusi (*Brute Force*)

Fungsi *solve_queens* merupakan inti (core) dari mekanisme pemecahan masalah. Algoritma ini menerapkan pendekatan Iterative Brute Force. Berbeda dengan pendekatan rekursif yang memanfaatkan tumpukan pemanggilan fungsi (call stack), pendekatan ini membangkitkan konfigurasi secara leksikografis menyerupai prinsip kerja odometer atau penghitung basis-*N*. Array config diinisialisasi dengan [0, 0, ..., 0] dan terus di increment hingga mencapai [N-1, N-1, ..., N-1] atau sampai solusi ditemukan.

```
function solve_queens(board) -> (SolutionMatrix, CheckedCount,
ElapsedTime):
    n <- length(board)
    checked_count <- 0
    start_time <- current_time()

    // Inisialisasi konfigurasi awal: semua ratu di kolom 0
    config <- array of size N initialized with 0
    finished <- False

    while not finished:
        checked_count <- checked_count + 1

        // Mengirim update progress jika ada callback (untuk visualisasi
GUI)
        if callback_exists and (checked_count mod update_interval == 0)
then
            callback(config, checked_count)

        // Periksa apakah konfigurasi saat ini adalah solusi
        if is_valid_config(board, config) then
            elapsed_time <- current_time() - start_time
            solution <- construct_solution_matrix(board, config)
            return solution, checked_count, elapsed_time

        // Pembangkitan Konfigurasi Berikutnya (Next Configuration)
        // Algoritma penambahan bilangan basis-N:
        idx <- n - 1
        while idx >= 0:
            // Tambahkan 1 pada posisi digit paling kanan
            config[idx] <- config[idx] + 1
```

```
            // Jika nilai masih < N, berarti valid, hentikan carry-over
            if config[idx] < n then
                break

            // Jika nilai mencapai N, reset jadi 0 dan lanjut ke digit kiri
(carry)
            config[idx] <- 0
            idx <- idx - 1

        // Jika indeks menjadi -1, berarti telah terjadi overflow pada
digit paling kiri
        // Artinya seluruh ruang pencarian telah habis diperiksa
        if idx < 0 then
            finished <- True

    elapsed_time <- current_time() - start_time
    return None, checked_count, elapsed_time
```

3.1. Analisis Kompleksitas

Berdasarkan algoritma yang diimplementasikan sebelumnya, berikut adalah analisis kompleksitas waktu dan ruang:

1. **Kompleksitas Waktu** (*Time Complexity*)
   - **Ruang Pencarian:** Algoritma membangkitkan permutasi dengan pengulangan (*permutations with repetition*), di mana setiap baris dari N baris memiliki N kemungkinan posisi kolom. Total konfigurasi maksimal adalah $N \times N \times \ldots \times N = N^N$.
   - **Biaya Validasi:** Untuk setiap konfigurasi, fungsi *IsValidConfig* melakukan iterasi bersarang (*nested loop*) untuk membandingkan setiap pasangan ratu guna mengecek ketetanggaan. Jumlah operasi perbandingan adalah $\frac{N(N-1)}{2}$, sehingga kompleksitas validasi adalah $O(N^2)$.
   - **Total:** Dalam kasus terburuk (*worst case*) di mana solusi berada di akhir pencarian atau tidak ada solusi, algoritma harus memvalidasi seluruh kemungkinan.

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

2. **Kompleksitas Ruang** (*Space Complexity*)
   - Penyimpanan matriks papan membutuhkan memori sebesar $O(N^2)$.
   - Penyimpanan array konfigurasi *config* membutuhkan memori sebesar $O(N)$
   - Karena algoritma bersifat iteratif dan tidak menggunakan rekursi, tidak ada *overhead* memori untuk *stack*.
   - **Total**: $S(N) = O(N^2)$.

4. Pengujian Dan Analisis

Bab ini membahas pengujian fungsionalitas dan kinerja program dalam menyelesaikan persoalan *Queens Puzzle*. Pengujian dilakukan untuk memverifikasi kebenaran algoritma *Brute Force* dalam menemukan solusi yang valid serta mengukur efisiensi waktu eksekusi pada berbagai ukuran papan *N x N*.

4.1. Lingkungan Pengujian dan Mekanisme Eksekusi

Program menyediakan dua mode antarmuka untuk menjalankan pengujian, yaitu *Command Line Interface* (CLI) dan *Graphical User Interface* (GUI). Keduanya mendukung fitur visualisasi proses pencarian (*live animation*).

A. Mode Command Line Interface (CLI)

Mode ini dijalankan melalui terminal dengan perintah *python main.py* pada direktori src. Pengguna diberikan opsi untuk melihat animasi langkah per langkah atau langsung melompat ke hasil akhir.

Contoh Interaksi CLI:

```
Masukkan nama file: 1.txt

Mode pencarian:
1. Langsung solusi (tanpa animasi)
2. Live condition (dengan animasi)
Pilih mode (1/2): 1

Board input:
BABC
ACDB
CDAA
BCDA

Solusi ditemukan:
B#BC
ACD#
#DAA
BC#A

Waktu pencarian: 0.00 ms
Banyak kasus yang ditinjau: 115 kasus
Apakah Anda ingin menyimpan solusi? (Ya/Tidak):
```

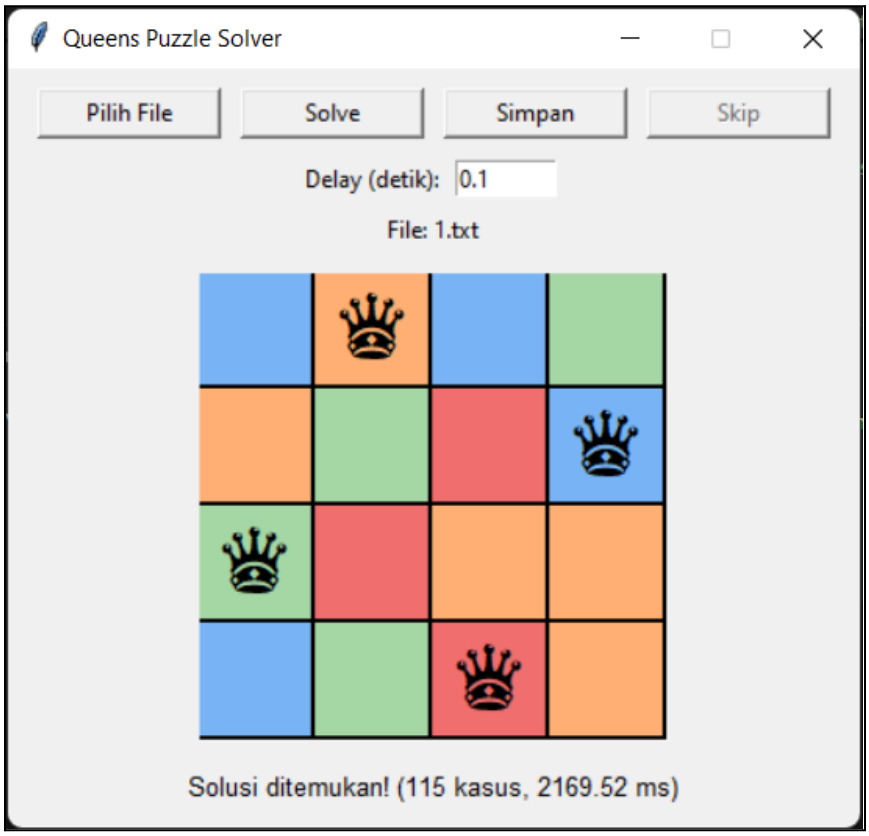
B. Mode Graphical User Interface (GUI)

Mode ini dijalankan dengan perintah *python gui.py pada lokasi src*. Antarmuka grafis mempermudah pengguna dalam memuat berkas dan mengamati proses algoritma secara visual.

Fitur utama pada GUI meliputi:

- Visualisasi Papan: Warna region dirender secara visual sesuai input.
- Kontrol Animasi: Tombol **Solve** memulai pencarian dengan animasi. Pengguna dapat mengatur kecepatan animasi menggunakan kolom *Delay*.
- Fitur Skip: Jika animasi berjalan terlalu lama, pengguna dapat menekan tombol **Skip** untuk segera menyelesaikan algoritma di latar belakang dan menampilkan hasil akhir seketika.

Berikut adalah tampilan antarmuka saat solusi berhasil ditemukan:



Gambar 2. Tampilan GUI Program Queens Solver.

4.2. Skenario Uji Eksperimental

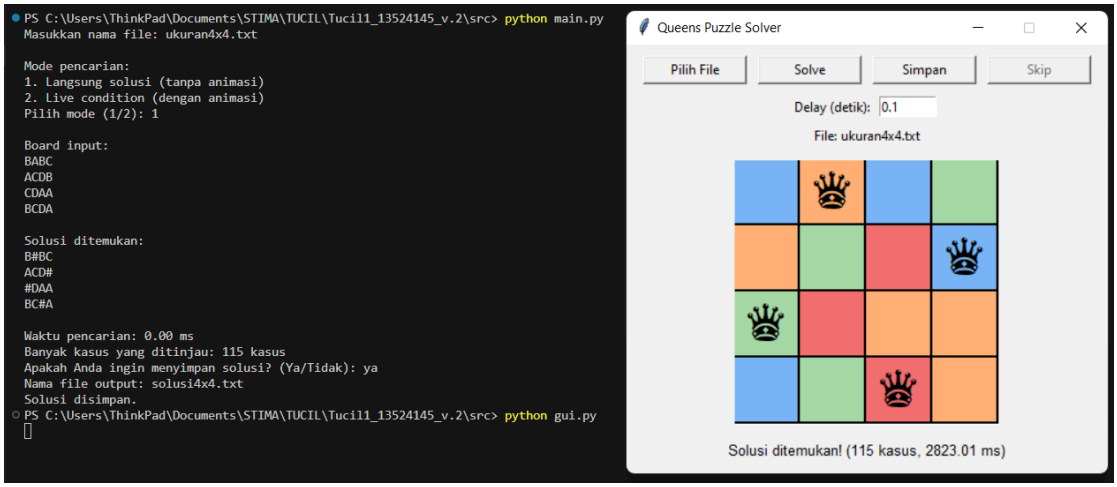
Pengujian dilakukan terhadap lima kasus uji dengan ukuran papan yang berbeda, mulai dari ukuran 4 x 4 hingga di ukuran 9 x 9, untuk menguji skalabilitas algoritma.

Kasus 1. Ukuran papan *4 x 4*

INPUT:

```
BABC
ACDB
CDAA
BCDA
```

OUTPUT:



Gambar 3. Tampilan output CLI & GUI pada kasus 01 (ukuran *4x4*).

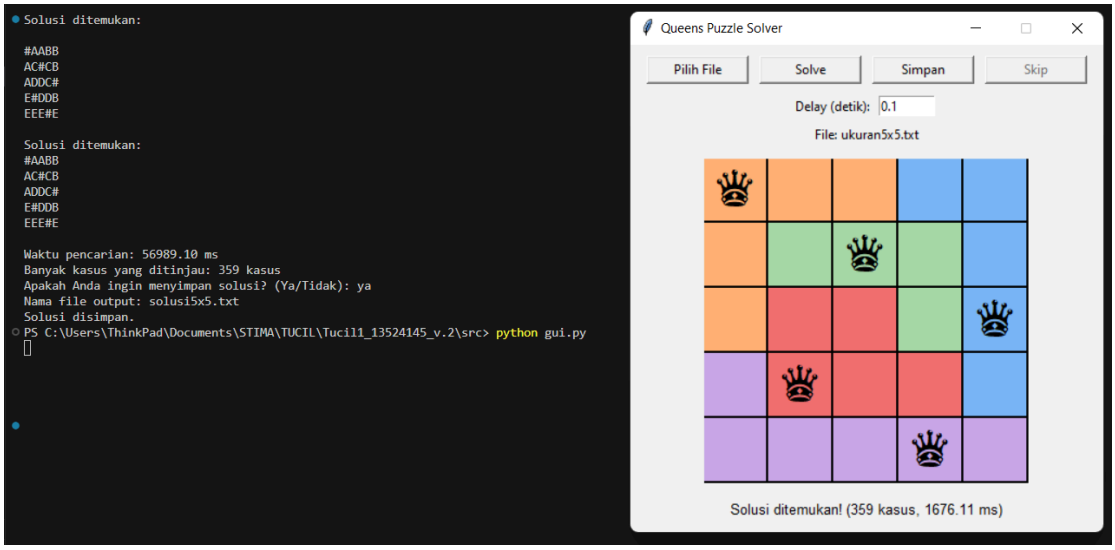
Berdasarkan Gambar 3. pengujian pada papan berukuran *4 x 4*, program **berhasil** menemukan solusi yang valid dan identik baik pada antarmuka CLI maupun GUI, di mana posisi ratu tidak saling menyerang sesuai aturan permainan. Konsistensi logika algoritma *Brute Force* terbukti dari jumlah iterasi yang diperiksa, yaitu tepat **115 kasus** pada kedua mode tersebut. Perbedaan mencolok hanya terletak pada waktu eksekusi; mode CLI mampu menyelesaikan pencarian secara instan (0.00 ms) karena prosesnya murni komputasional, sedangkan mode GUI mencatat waktu yang lebih lama (~2823 ms) bukan karena inefisiensi algoritma, melainkan akibat adanya *overhead* visualisasi grafis dan penambahan jeda waktu (*delay*) untuk keperluan animasi yang diatur oleh pengguna.

Kasus 2. Ukuran papan *5 x 5*

INPUT:

```
AAABB
ACCCB
ADDCB
EDDDB
EEEEE
```

OUTPUT:



Gambar 4. Tampilan output CLI & GUI pada kasus 02 (ukuran *5x5*).

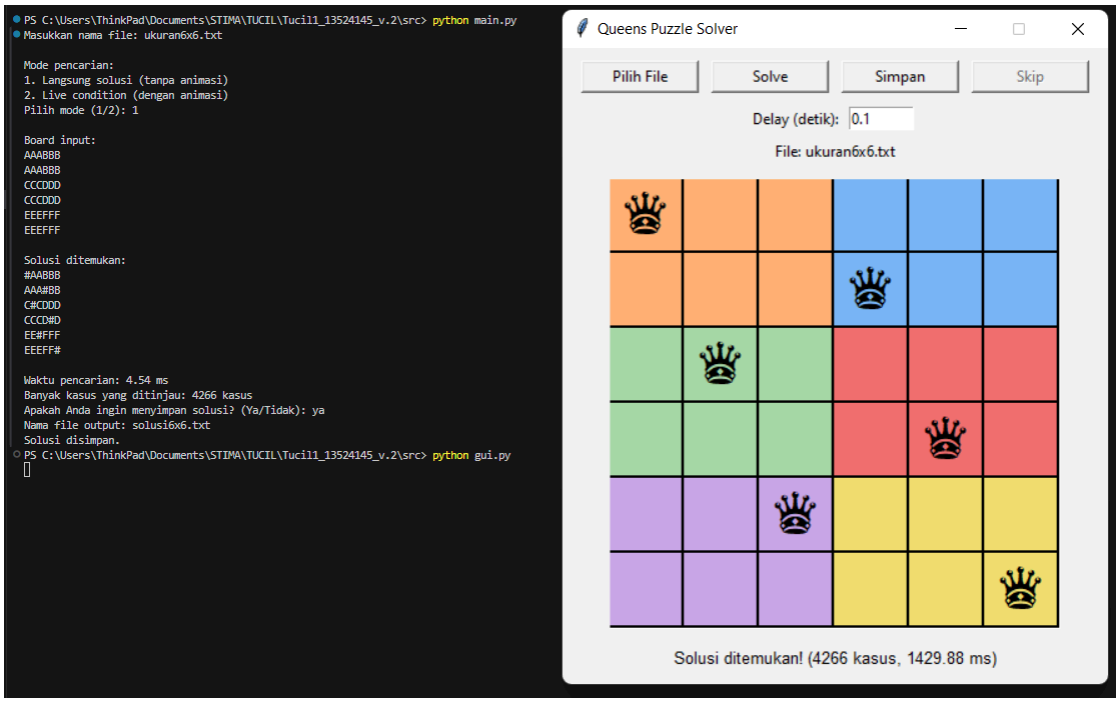
Pada eksperimen dengan papan berukuran *5 x 5*, algoritma berhasil mengidentifikasi solusi **valid** di mana setiap ratu ditempatkan pada posisi aman tanpa melanggar batasan wilayah warna maupun aturan ketetanggaan. Berdasarkan data pengujian, program secara konsisten meninjau tepat 359 konfigurasi sebelum mencapai solusi akhir. Meskipun jumlah iterasinya relatif sedikit, waktu eksekusi yang tercatat pada antarmuka CLI mencapai 56.989,10 ms (sekitar 57 detik), jauh lebih lambat dibandingkan hasil pada GUI yang hanya memerlukan 1.676,11 ms. Disparitas waktu yang signifikan ini terjadi karena pengujian pada CLI dijalankan menggunakan mode *Live Condition* (animasi), di mana beban operasi *input output* (I/O) untuk mencetak ulang papan ke terminal pada setiap langkah iterasi mendominasi total waktu proses, sementara waktu komputasi murni algoritma *Brute Force* itu sendiri sebenarnya berjalan sangat efisien dalam orde milidetik.

Kasus 3. Ukuran papan *6 x 6*

INPUT:

```
AAABBB
AAABBB
CCCDDD
CCCDDD
EEEFFF
EEEFFF
```

OUTPUT:



Gambar 5. Tampilan output CLI & GUI pada kasus 03 (ukuran *6x6*).

Pada pengujian papan berukuran *6 x 6* dengan enam wilayah warna (A-F) yang terdistribusi merata, program berhasil menemukan solusi valid di mana setiap ratu ditempatkan tanpa melanggar aturan baris, kolom, maupun ketetanggaan. Kompleksitas ruang pencarian mulai terlihat meningkat, dibuktikan dengan jumlah iterasi yang menembus angka 4.266 kasus, jauh lebih tinggi dibandingkan kasus *5 x 5*. Meskipun demikian, algoritma *Brute Force* tetap menunjukkan efisiensi yang tinggi pada mode CLI (Mode 1: Langsung Solusi) dengan waktu eksekusi yang sangat singkat, yaitu 4,54 ms. Sebaliknya, pada mode GUI tercatat waktu 1.429,88 ms; perbedaan waktu yang signifikan ini terjadi karena adanya *overhead* visualisasi grafis serta pengaturan *delay* animasi sebesar 0,1 detik, yang mengonfirmasi bahwa algoritma dasar berjalan optimal dan perlambatan hanya terjadi akibat faktor antarmuka pengguna.

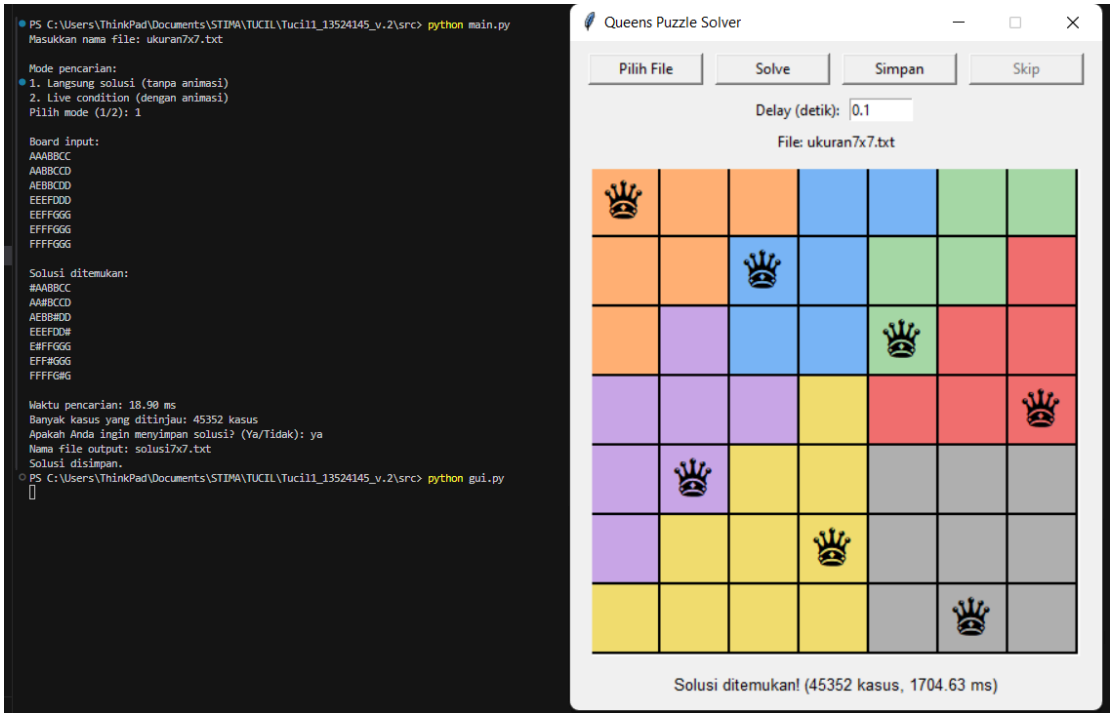
Kasus 4. Ukuran papan *7 x 7*

INPUT:

```
AAABBCC
AABBCCD
AEBBCDD
EEEFDDD
EEFFGGG
EFFFGGG
```

```
FFFFGGG
```

OUTPUT:



Gambar 6. Tampilan output CLI & GUI pada kasus 04 (ukuran *7x7*).

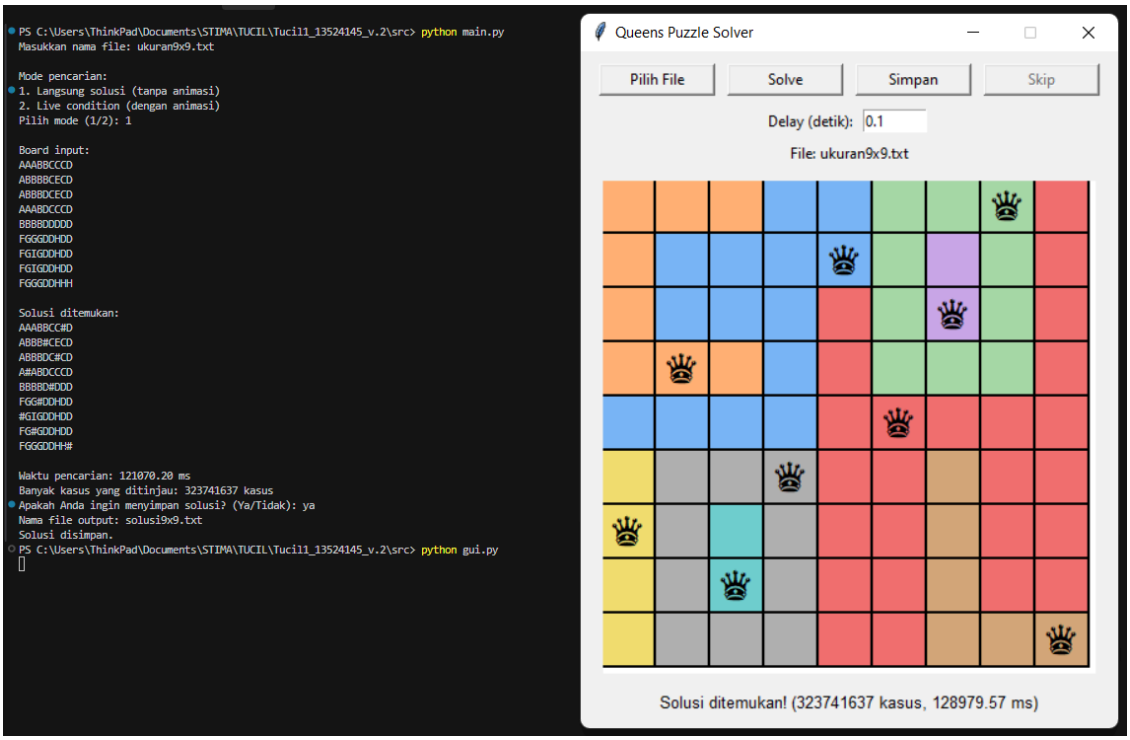
Pada pengujian papan berukuran 7 x 7, algoritma menghadapi peningkatan kompleksitas ruang pencarian yang cukup tajam, di mana program harus meninjau sebanyak 45.352 kasus naik lebih dari sepuluh kali lipat dibandingkan kasus 6 x 6 sebelum menemukan konfigurasi **solusi yang valid**. Solusi yang dihasilkan membentuk pola "langkah kuda" (*knight's move*) yang konsisten dari koordinat (0,0) hingga (6,5), memastikan setiap wilayah warna (A-G) terisi tanpa pelanggaran aturan. Meskipun beban komputasi bertambah, efisiensi algoritma pada mode CLI tetap terjaga dengan waktu eksekusi yang sangat singkat yaitu 18,90 ms, sementara mode GUI mencatat waktu 1.704,63 ms akibat *overhead* visualisasi, membuktikan bahwa algoritma *Brute Force* ini masih sangat andal menangani papan berukuran menengah dengan hasil deterministik.

Kasus 5. Ukuran papan *9 x 9 (dalam file spesifikasi tugas besar)*

INPUT:

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

OUTPUT:



Gambar 7. Tampilan output CLI & GUI pada kasus 05 (ukuran *9x9 - pada spesifikasi tugas kecil 01*).

Eksperimen pada papan 9 x 9 menunjukkan lonjakan kompleksitas yang ekstrem, di mana algoritma harus memvalidasi lebih dari **32 juta kasus** untuk menemukan solusi. Beban komputasi yang masif ini menyebabkan waktu eksekusi meningkat drastis hingga **121.070 ms** pada CLI dan **128.979 ms** pada GUI. Kecilnya selisih waktu antara kedua mode tersebut mengindikasikan bahwa *bottleneck* kinerja kini didominasi sepenuhnya oleh proses komputasi CPU akibat ruang pencarian yang sangat besar, bukan lagi oleh *overhead* visualisasi antarmuka.

4.3. Skenario Uji - Kasus Khusus

Kasus 1. Papan Kosong

```

```

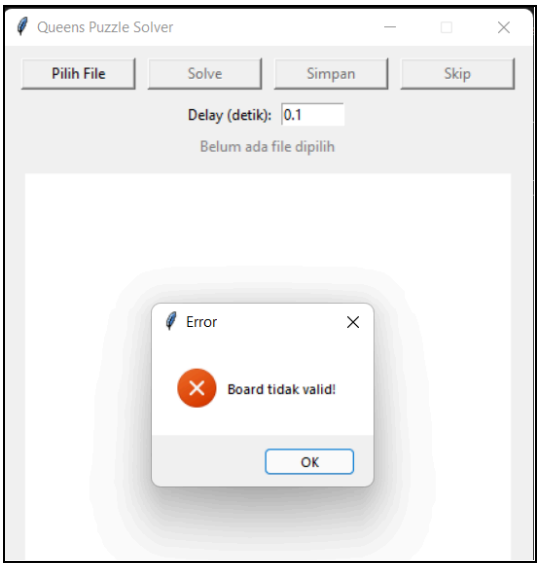
Kasus 2. Papan bukan ukuran *NxN*

```
AAAB
CCCB
CCCB
```

Kasus 3. *Warna region papan acak (tidak urut 'A' - 'Z')*

```
KKLL
KKLL
MMNN
MMNN
```

OUTPUT:



Gambar 8. Tampilan output CLI & GUI pada kasus khusus (Special Case).

Pengujian pada kasus khusus (edge cases) seperti file kosong, dimensi non-persegi, dan karakter acak mengonfirmasi bahwa validasi input berfungsi optimal; program berhasil mendeteksi format data yang salah dan merespons dengan pesan peringatan ***'Board tidak valid!'*** *untuk mencegah kegagalan sistem.*

5. Kesimpulan

Berdasarkan hasil pengujian menyeluruh, implementasi algoritma *Brute Force* dengan pendekatan rekursif *backtracking* terbukti efektif dan akurat dalam menyelesaikan permainan *Queen's Puzzle*. Program mampu menghasilkan solusi deterministik yang valid untuk berbagai variasi papan, mulai dari ukuran kecil (*4 x 4*) hingga besar (*9 x 9*), dengan memastikan setiap penempatan ratu memenuhi seluruh batasan aturan yang ketat, meliputi larangan konflik pada baris, kolom, wilayah warna, serta aturan ketetanggaan (*adjacency constraints*). Kebenaran logika algoritma terkonfirmasi melalui konsistensi solusi yang dihasilkan pada mode CLI maupun GUI.

Analisis performa menunjukkan korelasi langsung antara ukuran papan (N) dengan kompleksitas komputasi yang bersifat eksponensial. Pada kasus berskala kecil hingga menengah ($N \leq 7$), algoritma bekerja sangat efisien dengan waktu eksekusi di bawah 20 ms. Namun, keterbatasan metode *Brute Force* terlihat jelas pada kasus *9 x 9*, di mana ruang pencarian meledak hingga memerlukan pemeriksaan lebih dari 32 juta kasus dengan waktu eksekusi mencapai 121 detik. Hal ini mengonfirmasi bahwa meskipun algoritma ini menjamin kelengkapan pencarian (*completeness*), biaya komputasinya menjadi sangat mahal seiring bertambahnya dimensi masalah, di mana *bottleneck* bergeser dari *overhead* visualisasi antarmuka menjadi beban pemrosesan CPU murni.

Dari sisi kualitas perangkat lunak, aplikasi menunjukkan tingkat ketahanan (*robustness*) yang baik melalui mekanisme validasi input yang responsif terhadap kondisi ekstrim (*edge cases*), seperti format papan yang tidak valid atau file kosong, sehingga mencegah terjadinya kegagalan sistem (*crash*). Selain itu, penyediaan dua moda antarmuka memberikan fleksibilitas fungsional: mode CLI yang mengutamakan kecepatan untuk pengukuran performa murni, dan mode GUI yang memfasilitasi pemahaman visual terhadap langkah-langkah *backtracking* algoritma, meskipun dengan konsekuensi *trade-off* waktu eksekusi yang signifikan akibat proses *rendering* grafis.

6. Daftar Pustaka

- **Levitin, A.** (2012). *Introduction to the Design and Analysis of Algorithms* (3rd ed.). Boston: Pearson Education. *(Buku wajib untuk mata kuliah Strategi Algoritma)*
- **M. Rinaldi**, *Spesifikasi Tugas Kecil 1 - IF2211 - 2025/2026* [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/Tucil1-IF2211-2026.pdf
- **M. Rinaldi**, *02-Algoritma-Brute-Force-(2026)-Bag1.pdf* [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/02-Algoritma-Brute-Force-(2026)-Bag1.pdf
- **M. Rinaldi**, *03-Algoritma-Brute-Force-(2026)-Bag2.pdf* [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/03-Algoritma-Brute-Force-(2026)-Bag2.pdf
- **Python Software Foundation.** (2026). *Tkinter — Python interface to Tcl/Tk*. Tersedia di: https://docs.python.org/3/library/tkinter.html [Diakses 14 Februari 2026]. *(Dokumentasi resmi untuk pustaka GUI yang kamu pakai)*
- **Russell, S. J., & Norvig, P.** (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Hoboken: Pearson. *(Referensi standar dunia untuk teori Constraint Satisfaction Problem)*

7. Lampiran

7.1. Kriteria Program

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		✓

7.2. Pernyataan Tidak Melakukan Kecurangan

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

Dzakwan Muhammad Khairan Putra Purnama

*__Oleh:__ *Dzakwan Muhammad Khairan Putra Purnama - 13524145 (Teknik Informatika 2024).*

Link Source Code Program (github): https://github.com/dzakwanmkpp/Tucil1_13524145

Link Source Emoji Queen: https://emojidb.org/crown-emojis