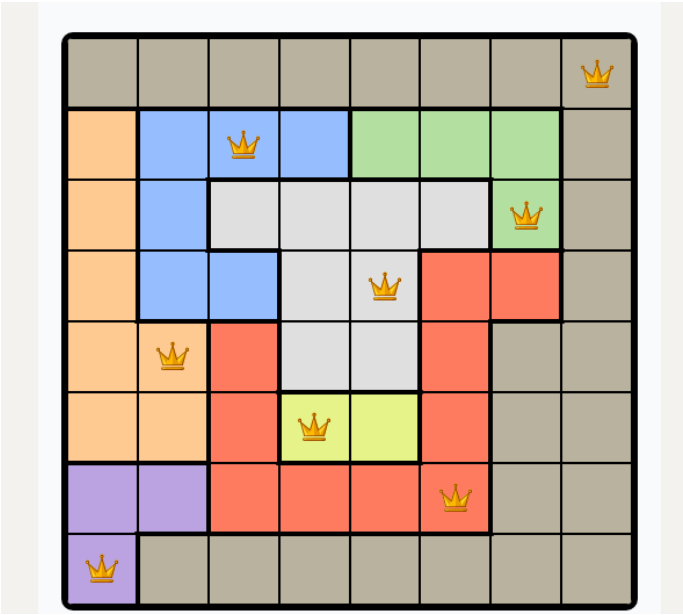
Laporan Tugas Kecil 1 IF2211 Strategi Algoritma
Penyelesaian Permainan Queens LinkedIn



Rava Khoman Tuah Saragih
13524107
13524107@std.stei.itb.ac.id

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

Semester II Tahun 2025/2026

Deskripsi Permasalahan

LinkedIn Queens Game adalah permainan logika berbasis grid yang dimainkan di atas papan berukuran NxN. Papan permainan ini terbagi menjadi N wilayah warna yang bentuknya tidak beraturan namun saling terhubung.

Tujuan utama dari permainan ini adalah menempatkan N buah "Queen" ke dalam papan sedemikian rupa sehingga memenuhi serangkaian batasan yang ketat. Permainan ini menuntut pemain untuk berpikir logis dalam mengeliminasi posisi-posisi yang tidak valid berdasarkan aturan yang ada.

Agar sebuah konfigurasi penempatan Queen dianggap valid dan merupakan solusi, setiap Queen harus ditempatkan dengan mematuhi aturan berikut:

- Satu per Baris: Setiap baris pada papan NxN harus memiliki tepat satu Queen.
- Satu per Kolom: Setiap kolom pada papan harus memiliki tepat satu Queen.
- Satu per Warna: Setiap wilayah warna yang berbeda pada papan harus memiliki tepat satu Queen.
- Batasan Adjacency: Tidak boleh ada dua Queen yang saling bersentuhan, baik secara horizontal, vertikal, maupun diagonal. Ini berarti setiap Queen harus dikelilingi oleh setidaknya satu sel kosong di kedelapan arah mata angin.

Pendekatan Algoritma Solusi

Algoritma Brute Froce

Algoritma Brute Force yang diimplementasikan bekerja dengan cara mengeksplorasi setiap kemungkinan penempatan Queen secara sistematis.

Langkah-lngkah:

1. Inisialisasi sebuah array ukuran N dimana array dengan indeks *i* merepresentasikan kolom Queen pada baris ke-i.
2. Mulai iterasi dalam infinite loop untuk mencoba setiap kombinasi posisi kolom yang mungkin.
3. Setiap kali sebuah konfigurasi lengkap terbentuk, fungsi validasi dipanggil.
4. Validasi Global: Fungsi is_valid memeriksa seluruh konstrain secara bersamaan:
   - Apakah ada kolom yang sama (ditempati >1 Queen)?
   - Apakah ada region warna yang sama?
   - Apakah ada dua Queen yang bertetangga?
5. Jika valid, simpan sebagai solusi. Jika tidak, lanjutkan ke konfigurasi berikutnya.

Pendekatan ini sangat lambat untuk $N > 8$ karena kompleksitas waktunya mendekati $O(N^N)$, yaitu mencoba semua kombinasi tanpa mempedulikan pelanggaran dini.

Algoritma Optimized (Backtracking)

Pendekatan ini merupakan perbaikan signifikan dari algoritma Brute Force dengan menerapkan teknik pruning (pemangkasan) yang agresif. Prinsip kerja algoritma ini memanfaatkan sifat rekursif dari masalah ini untuk membangun solusi secara bertahap, namun dengan perbedaan kunci pada validasi langkah:

1. Validasi Parsial (Incremental Validation): Setiap kali sebelum menempatkan Queen pada posisi (r, c), algoritma memeriksa apakah langkah tersebut aman terhadap Queen yang sudah ditempatkan sebelumnya.

2. Pemangkasan (Pruning): Jika posisi (r, c) ditemukan melanggar salah satu constraint (misalnya, kolom c sudah terisi atau warna wilayah tersebut sudah memiliki Queen), maka algoritma tidak akan menempatkan Queen di sana dan tidak akan melanjutkan pencarian ke baris berikutnya untuk cabang tersebut.

3. Backtracking: Jika pada suatu baris tidak ditemukan kolom yang aman, algoritma akan kembali ke baris sebelumnya untuk mencoba posisi kolom yang berbeda.

Dengan melakukan validasi di setiap langkah, algoritma ini secara efektif memotong sebagian besar cabang dari pohon pencarian yang tidak potensial. Hal ini mengurangi ruang pencarian dari N^N (total kemungkinan penempatan sembarang) menjadi jauh lebih kecil, memungkinkan program menyelesaikan puzzle dengan $N = 20$ dalam waktu yang sangat singkat dibandingkan Brute Force yang mungkin tidak selesai dalam waktu wajar.

Implementasi Kode

queens_game_solver.py

```
import time
import os
import math

class Board:
    def __init__(self):
        self.n = 0
        self.grid = []
        self.regions = {}
        self.queens = []
        self.region_colors = {}

    def load_from_file(self, filepath):
        # tambahkan data ke grid
        try:
            with open(filepath) as file:
                for line in file:
                    line = line.strip()
                    if line:
                        row = list(line)
                        self.grid.append(row)
                        # kaya gini tuh bisa juga:
                        # self.grid.append(list(line.strip()))
        except FileNotFoundError:
            print("File tidak ditemukan.")
            return False
```

```
        # hitung jumlah baris (n)
        self.n = len(self.grid)

        # tambahkan data daerah warna
        for row in range(self.n):
            for col in range(len(self.grid[row])):
                colour = self.grid[row][col]
                if colour not in self.regions:
                    self.regions[colour] = []
                self.regions[colour].append((row,col))

        if not self.validate():
            return False

    def display(self):
        for row in range(self.n):
            for col in range(self.n):
                if (row,col) in self.queens:
                    print("#",end="")
                else:
                    print(self.grid[row][col],end="")
            print()

    def validate(self):
        if self.n == 0:
            print("Papan kosong.")
            return False

        for row in range(self.n):
            if len(self.grid[row]) != self.n:
                print("Papan tidak persegi")
                return False

        if len(self.regions) != self.n:
            print("Jumlah warna tidak sama dengan jumlah sisi papan.")
            return False

        return True

    def load_from_image(self, filepath):
        try:
            from PIL import Image
        except ImportError:
            print("Library Pillow belum terinstall. Jalankan: pip install
Pillow")
            return False

        try:
            img = Image.open(filepath).convert("RGB")
        except Exception:
            print("Gagal membuka file gambar.")
            return False

        width, height = img.size
        pixels = img.load()

        col_positions = self._find_cell_positions(pixels, width, height,
```

```
axis="x")
        if len(col_positions) < 2:
            print("Tidak bisa mendeteksi kolom dari gambar.")
            return False

        first_col_center = (col_positions[0][0] + col_positions[0][1]) // 2
        row_positions = self._find_cell_positions(pixels, width, height,
axis="y", scan_at=first_col_center)

        n_cols = len(col_positions)
        n_rows = len(row_positions)

        if n_cols != n_rows or n_cols < 2:
            print(f"Grid tidak valid: {n_rows} baris, {n_cols} kolom
terdeteksi.")
            return False

        self.n = n_cols

        raw_colors = []
        for row_start, row_end in row_positions:
            row_colors = []
            for col_start, col_end in col_positions:
                cx = (col_start + col_end) // 2
                cy = (row_start + row_end) // 2
                color = pixels[cx, cy]
                row_colors.append(color)
            raw_colors.append(row_colors)

        color_groups = []
        letter_index = 0
        letters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

        self.grid = []
        for row in range(self.n):
            grid_row = []
            for col in range(self.n):
                color = raw_colors[row][col]
                assigned = False
                for i, (rep_color, letter) in enumerate(color_groups):
                    if self._color_distance(color, rep_color) < 35:
                        grid_row.append(letter)
                        assigned = True
                        break
                if not assigned:
                    letter = letters[letter_index % len(letters)]
                    letter_index += 1
                    color_groups.append((color, letter))
                    grid_row.append(letter)
            self.grid.append(grid_row)

        # Simpan mapping warna asli untuk GUI
        self.region_colors = {}
        for rep_color, letter in color_groups:
            r, g, b = rep_color
            self.region_colors[letter] = f"#{r:02x}{g:02x}{b:02x}"

        self.regions = {}
```

```
        for row in range(self.n):
            for col in range(self.n):
                colour = self.grid[row][col]
                if colour not in self.regions:
                    self.regions[colour] = []
                self.regions[colour].append((row, col))

        if not self.validate():
            return False

        return True

    def _find_cell_positions(self, pixels, width, height, axis="x",
scan_at=None):
        if axis == "x":
            length = width
            scan_pos = scan_at if scan_at else height // 4
            get_pixel = lambda i: pixels[i, scan_pos]
        else:
            length = height
            scan_pos = scan_at if scan_at else width // 4
            get_pixel = lambda i: pixels[scan_pos, i]

        is_dark = []
        for i in range(length):
            c = get_pixel(i)
            is_dark.append(c[0] < 100 and c[1] < 100 and c[2] < 100)

        cells = []
        in_cell = False
        start = 0
        for i in range(length):
            if not is_dark[i] and not in_cell:
                start = i
                in_cell = True
            elif is_dark[i] and in_cell:
                cells.append((start, i - 1))
                in_cell = False
        if in_cell:
            cells.append((start, length - 1))

        min_size = 10
        cells = [(s, e) for s, e in cells if (e - s + 1) >= min_size]

        return cells

    def _color_distance(self, c1, c2):
        # hitung jarak euclidean buat dua warna rgb
        return math.sqrt((c1[0]-c2[0])**2 + (c1[1]-c2[1])**2 + (c1[2]-c2[2])**2)

    def save_solution(self, filepath):
        with open(filepath, "w") as file:
            for row in range(self.n):
                for col in range(self.n):
                    if (row, col) in self.queens:
                        file.write("#")
                    else:
                        file.write(self.grid[row][col])
```

```
                file.write("\n")

class BruteForceSolver:
    def __init__(self, board):
        self.iteration_count = 0
        self.board = board

    def is_valid(self, config):
        for i in range(self.board.n):
            for j in range(i+1,self.board.n):
                if config[i] == config[j]:
                    return False
                if self.board.grid[i][config[i]] ==
self.board.grid[j][config[j]]:
                    return False
                if abs(i - j) <= 1 and abs(config[i] - config[j]) <= 1:
                    return False
        return True

    def solve(self, on_update=None, get_interval=None):
        n = self.board.n
        config = [0 for i in range(n)]
        while True:
            self.iteration_count += 1

            if on_update and get_interval:
                interval = get_interval()
                if self.iteration_count % interval == 0:
                    self.board.queens = [(row, config[row]) for row in range(n)]
                    on_update()

            if self.is_valid(config):
                self.board.queens = [(row, config[row]) for row in range(n)]
                return True
            else:
                i = n - 1
                config[i] += 1

                while config[i] >= n:
                    config[i] = 0
                    i -= 1
                    if i < 0:
                        return False
                    config[i] += 1

class OptimizedSolver:
    def __init__(self, board):
        self.iteration_count = 0
        self.board = board
        self.cols_used = set()
        self.regions_used = set()

    def is_valid(self, row, col):
        if col in self.cols_used:
            return False

        if self.board.grid[row][col] in self.regions_used:
            return False
```

```
        for (queen_row, queen_col) in self.board.queens:
            if abs(queen_row - row) <= 1 and abs(queen_col - col) <= 1:
                return False

        return True

    def solve(self, row, on_update=None, get_interval=None):
        if row == self.board.n:
            return True

        for col in range(self.board.n):
            self.iteration_count += 1

            if on_update and get_interval:
                interval = get_interval()
                if self.iteration_count % interval == 0:
                    on_update()

            if self.is_valid(row,col):
                self.board.queens.append((row,col))
                self.cols_used.add(col)
                self.regions_used.add(self.board.grid[row][col])
                if self.solve(row+1, on_update, get_interval):
                    return True
                self.board.queens.pop()
                self.cols_used.remove(col)
                self.regions_used.remove(self.board.grid[row][col])
        return False

#main
if __name__ == "__main__":
    filepath = input("Masukkan path file test case (.txt): ")
    board = Board()
    result = board.load_from_file(filepath)
    if result != False:
        solver = OptimizedSolver(board)
        start = time.time()
        found = solver.solve(0)
        end = time.time()
        elapsed_time = (end - start) * 1000
        if found:
            print("Solusi ditemukan!")
            board.display()
            print(f"Waktu pencarian: {elapsed_time:.2f} ms")
            print(f"Banyak kasus yang ditinjau: {solver.iteration_count}")
            simpan = input("Apakah Anda ingin menyimpan solusi? (Ya/Tidak): ")
            if simpan == "Ya":
                nama_file = input("Masukkan nama file output: ")
                board.save_solution(nama_file)
                print(f"Solusi disimpan ke {nama_file}")
        else:
            print("Tidak ada solusi.")
```

tucil1.py

```
import tkinter as tk
from tkinter import filedialog, messagebox
import time
import threading
import multiprocessing
import math
from queens_game_solver import Board, OptimizedSolver, BruteForceSolver

try:
    from PIL import Image, ImageDraw, ImageFont
except ImportError:
    Image = None

def load_board_from_image(board, filepath):
    if Image is None:
        print("Library Pillow belum terinstall. Jalankan: pip install Pillow")
        return False

    try:
        img = Image.open(filepath).convert("RGB")
    except Exception:
        print("Gagal membuka file gambar.")
        return False

    width, height = img.size
    pixels = img.load()

    prof_x = _get_projection_profile(pixels, width, height, axis="x")
    col_ranges = _detect_grid_cells(prof_x, width)
    col_ranges = _refine_cells(col_ranges)

    if len(col_ranges) < 2:
        print("Gagal mendeteksi kolom (grid tidak ditemukan).")
        return False

    prof_y = _get_projection_profile(pixels, width, height, axis="y")
    row_ranges = _detect_grid_cells(prof_y, height)

    row_ranges = _refine_cells(row_ranges)

    n_cols = len(col_ranges)
    n_rows = len(row_ranges)

    if n_cols != n_rows or n_cols < 2:
        print(f"Grid tidak valid: {n_rows} baris, {n_cols} kolom terdeteksi.")
        return False

    board.n = n_cols

    raw_colors = []
    for row_start, row_end in row_ranges:
        row_colors = []
        for col_start, col_end in col_ranges:
            cw = col_end - col_start
            ch = row_end - row_start
```

```
            check_x = col_start + cw // 2
            check_y = row_start + ch // 2

            r_sum, g_sum, b_sum = 0, 0, 0
            count = 0
            for dx in range(-1, 2):
                for dy in range(-1, 2):
                    px = check_x + dx
                    py = check_y + dy
                    if 0 <= px < width and 0 <= py < height:
                        r, g, b = pixels[px, py]
                        r_sum += r
                        g_sum += g
                        b_sum += b
                        count += 1

            avg_color = (r_sum//count, g_sum//count, b_sum//count)
            row_colors.append(avg_color)
        raw_colors.append(row_colors)

    color_groups = []
    letter_index = 0
    letters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

    board.grid = []
    for row in range(board.n):
        grid_row = []
        for col in range(board.n):
            color = raw_colors[row][col]
            assigned = False
            for i, (rep_color, letter) in enumerate(color_groups):
                if _color_distance(color, rep_color) < 40:
                    grid_row.append(letter)
                    assigned = True
                    break
            if not assigned:
                letter = letters[letter_index % len(letters)]
                letter_index += 1
                color_groups.append((color, letter))
                grid_row.append(letter)
        board.grid.append(grid_row)

    board.region_colors = {}
    for rep_color, letter in color_groups:
        r, g, b = rep_color
        board.region_colors[letter] = f"#{r:02x}{g:02x}{b:02x}"

    board.regions = {}
    for row in range(board.n):
        for col in range(board.n):
            colour = board.grid[row][col]
            if colour not in board.regions:
                board.regions[colour] = []
            board.regions[colour].append((row, col))

    if not board.validate():
        return False
```

```
    return True

def _get_projection_profile(pixels, width, height, axis="x"):
    profile = []
    threshold = 100

    if axis == "x":
        for x in range(width):
            score = 0
            for y in range(0, height, 5):
                r, g, b = pixels[x, y]
                if r < threshold and g < threshold and b < threshold:
                    score += 1
            profile.append(score)
    else:
        for y in range(height):
            score = 0
            for x in range(0, width, 5):
                r, g, b = pixels[x, y]
                if r < threshold and g < threshold and b < threshold:
                    score += 1
            profile.append(score)

    return profile

def _detect_grid_cells(profile, length):
    if not profile: return []

    max_score = max(profile)
    border_threshold = max_score * 0.2

    is_border = [score > border_threshold for score in profile]

    cells = []
    in_cell = False
    start = 0

    for i in range(length):
        if not is_border[i] and not in_cell:
            start = i
            in_cell = True
        elif is_border[i] and in_cell:
            cells.append((start, i - 1))
            in_cell = False

    if in_cell:
        cells.append((start, length - 1))

    return cells

def _refine_cells(cells):
    if not cells: return []

    sizes = [e - s + 1 for s, e in cells]
    sizes.sort()
    mid_idx = len(sizes) // 2
```

```
    median = sizes[mid_idx]

    processed = []
    for s, e in cells:
        size = e - s + 1

        if size < 0.4 * median:
            continue

        if size > 1.5 * median:
            ratio = size / median
            num_splits = round(ratio)
            if num_splits < 2: num_splits = 2

            cell_w = size / num_splits
            for i in range(num_splits):
                start = s + int(i * cell_w)
                end = s + int((i + 1) * cell_w) - 1
                if start <= end:
                    processed.append((start, end))
            continue

        processed.append((s, e))

    return processed

def _color_distance(c1, c2):
    return math.sqrt((c1[0]-c2[0])**2 + (c1[1]-c2[1])**2 + (c1[2]-c2[2])**2)

# GUI
def _silent_solve(grid, n, regions, mode, result_queue):
    """Module-level function untuk multiprocessing (Windows butuh ini)."""
    board = Board()
    board.n = n
    board.grid = grid
    board.regions = regions
    board.queens = []

    start = time.perf_counter()
    if mode == "brute_force":
        solver = BruteForceSolver(board)
        found = solver.solve()
    else:
        solver = OptimizedSolver(board)
        found = solver.solve(0)
    end = time.perf_counter()

    result_queue.put({
        'found': found,
        'elapsed': (end - start) * 1000,
        'iterations': solver.iteration_count
    })

COLOR_PALETTE = [
    "#E74C3C","#27AE60","#8E44AD","#D4AC0D","#0984E3",
    "#E17055","#6C5CE7","#00CEC9","#FDCB6E","#2D3436",
    "#A29BFE","#55EFC4","#FF7675","#74B9FF","#FAB1A0",
```

```
    "#636E72","#FD79A8","#B2BEC3","#C0392B","#FFEAA7",
]

class QueensGUI:
    def __init__(self, root):
        self.root = root
        self.root.title("Queens LinkedIn Solver")
        self.root.configure(bg="#2C3E50")
        self.root.resizable(False, False)

        self.board = None
        self.solver = None
        self.cell_size = 60
        self.color_map = {}
        self.solve_mode = tk.StringVar(value="brute_force")
        self.update_interval = 1
        self.solving = False
        self.stop_solving = False
        self.solve_start_time = 0

        self._build_ui()

    def _build_ui(self):
        title_frame = tk.Frame(self.root, bg="#2C3E50", pady=10)
        title_frame.pack(fill=tk.X)

        tk.Label(
            title_frame, text="Queens LinkedIn Solver",
            font=("Arial", 20, "bold"), bg="#2C3E50", fg="#ECF0F1"
        ).pack()

        tk.Label(
            title_frame, text="Rava Khoman Tuah Saragih - 13524107",
            font=("Arial", 11), bg="#2C3E50", fg="#BDC3C7"
        ).pack()
        top_frame = tk.Frame(self.root, bg="#2C3E50", pady=5)
        top_frame.pack(fill=tk.X, padx=10)
        self.btn_load = tk.Button(
            top_frame, text="Load .txt", command=self.load_file,
            font=("Arial", 11, "bold"), bg="white", fg="black",
            relief=tk.FLAT, padx=15, pady=5, cursor="hand2"
        )
        self.btn_load.pack(side=tk.LEFT, padx=5)

        self.btn_load_img = tk.Button(
            top_frame, text="Load Image", command=self.load_image,
            font=("Arial", 11, "bold"), bg="white", fg="black",
            relief=tk.FLAT, padx=15, pady=5, cursor="hand2"
        )
        self.btn_load_img.pack(side=tk.LEFT, padx=5)
        self.btn_solve = tk.Button(
            top_frame, text="Solve", command=self._on_solve_click,
            font=("Arial", 11, "bold"), bg="white", fg="black",
            relief=tk.FLAT, padx=15, pady=5, cursor="hand2",
            state=tk.DISABLED
        )
        self.btn_solve.pack(side=tk.LEFT, padx=5)
```

```
        self.btn_stop = tk.Button(
            top_frame, text="Stop", command=self.stop_solve,
            font=("Arial", 11, "bold"), bg="white", fg="black",
            relief=tk.FLAT, padx=15, pady=5, cursor="hand2",
            state=tk.DISABLED
        )
        self.btn_stop.pack(side=tk.LEFT, padx=5)
        self.btn_save = tk.Button(
            top_frame, text="Save", command=self.save_solution,
            font=("Arial", 11, "bold"), bg="white", fg="black",
            relief=tk.FLAT, padx=15, pady=5, cursor="hand2",
            state=tk.DISABLED
        )
        self.btn_save.pack(side=tk.LEFT, padx=5)
        self.canvas_frame = tk.Frame(self.root, bg="#2C3E50")
        self.canvas_frame.pack(padx=10, pady=5)
        self.canvas = tk.Canvas(
            self.canvas_frame, width=400, height=400,
            bg="#34495E", highlightthickness=0
        )
        self.canvas.pack()
        bottom_frame = tk.Frame(self.root, bg="#2C3E50", pady=10)
        bottom_frame.pack(fill=tk.X, padx=10)
        self.label_status = tk.Label(
            bottom_frame, text="Silahkan load file terlebih dahulu.",
            font=("Arial", 11), bg="#2C3E50", fg="#ECF0F1"
        )
        self.label_status.pack()
        self.label_time = tk.Label(
            bottom_frame, text="",
            font=("Arial", 10), bg="#2C3E50", fg="#BDC3C7"
        )
        self.label_time.pack()
        self.label_iter = tk.Label(
            bottom_frame, text="",
            font=("Arial", 10), bg="#2C3E50", fg="#BDC3C7"
        )
        self.label_iter.pack()
        self.label_real_time = tk.Label(
            bottom_frame, text="",
            font=("Arial", 10, "bold"), bg="#2C3E50", fg="#F39C12"
        )
        self.label_real_time.pack()
        algo_frame = tk.Frame(self.root, bg="#34495E", pady=10, padx=15)
        algo_frame.pack(fill=tk.X, padx=10, pady=(5, 0))

        tk.Label(
            algo_frame, text="Pilih Algoritma:",
            font=("Arial", 11, "bold"), bg="#34495E", fg="#ECF0F1"
        ).pack(anchor=tk.W)
        bf_frame = tk.Frame(algo_frame, bg="#34495E")
        bf_frame.pack(fill=tk.X, pady=(5, 2))
        tk.Radiobutton(
            bf_frame, text="Brute Force",
            variable=self.solve_mode, value="brute_force",
            font=("Arial", 11, "bold"), bg="#34495E", fg="#ECF0F1",
            selectcolor="#2C3E50", activebackground="#34495E",
```

```
            activeforeground="#ECF0F1"
        ).pack(anchor=tk.W)
        tk.Label(
            bf_frame, text="Mencoba semua kemungkinan penempatan queen secara
exhaustive.",
            font=("Arial",  9),  bg="#34495E",  fg="#95A5A6",  wraplength=500,
justify=tk.LEFT
        ).pack(anchor=tk.W, padx=20)
        opt_frame = tk.Frame(algo_frame, bg="#34495E")
        opt_frame.pack(fill=tk.X, pady=(2, 0))
        tk.Radiobutton(
            opt_frame, text="Optimized",
            variable=self.solve_mode, value="optimized",
            font=("Arial", 11, "bold"), bg="#34495E", fg="#ECF0F1",
            selectcolor="#2C3E50", activebackground="#34495E",
            activeforeground="#ECF0F1"
        ).pack(anchor=tk.W)
        tk.Label(
            opt_frame,
            text="Backtracking dengan validasi per baris. Memangkas cabang yang
tidak valid lebih awal untuk efisiensi.",
            font=("Arial",  9),  bg="#34495E",  fg="#95A5A6",  wraplength=500,
justify=tk.LEFT
        ).pack(anchor=tk.W, padx=20)
        slider_frame = tk.Frame(self.root, bg="#2C3E50", pady=5)
        slider_frame.pack(fill=tk.X, padx=10)
        tk.Label(
            slider_frame, text="Update setiap:",
            font=("Arial", 10), bg="#2C3E50", fg="#BDC3C7"
        ).pack(side=tk.LEFT, padx=5)
        self.speed_slider = tk.Scale(
            slider_frame, from_=0, to=1000, orient=tk.HORIZONTAL,
            bg="#34495E", fg="#ECF0F1", troughcolor="#2C3E50",
            highlightthickness=0, length=300, showvalue=0,
            command=self._on_slider_change
        )
        self.speed_slider.set(0)
        self.speed_slider.pack(side=tk.LEFT, padx=5)

        self.iter_entry = tk.Entry(
            slider_frame, width=8, font=("Arial", 10),
            bg="#34495E", fg="#ECF0F1", insertbackground="#ECF0F1",
            justify=tk.CENTER
        )
        self.iter_entry.insert(0, "1")
        self.iter_entry.pack(side=tk.LEFT, padx=5)
        self.iter_entry.bind("<Return>", self._on_entry_change)
        self.iter_entry.bind("<FocusOut>", self._on_entry_change)
        tk.Label(
            slider_frame, text="iterasi",
            font=("Arial", 10), bg="#2C3E50", fg="#BDC3C7"
        ).pack(side=tk.LEFT, padx=2)

    def _on_solve_click(self):
        self.start_solve(self.solve_mode.get())

    def stop_solve(self):
```

```
        self.stop_solving = True
        self.label_status.config(text="□ Solving dihentikan.", fg="#E74C3C")

    def _assign_colors(self):
        self.color_map = {}
        if self.board.region_colors:
            self.color_map = dict(self.board.region_colors)
        else:
            regions = sorted(self.board.regions.keys())
            for i, region in enumerate(regions):
                self.color_map[region] = COLOR_PALETTE[i % len(COLOR_PALETTE)]

    def draw_board(self):
        self.canvas.delete("all")
        if self.board is None or self.board.n == 0:
            return
        n = self.board.n
        self.cell_size = min(400 // n, 80)
        canvas_size = self.cell_size * n
        self.canvas.config(width=canvas_size, height=canvas_size)
        for row in range(n):
            for col in range(n):
                x1 = col * self.cell_size
                y1 = row * self.cell_size
                x2 = x1 + self.cell_size
                y2 = y1 + self.cell_size

                region = self.board.grid[row][col]
                color = self.color_map.get(region, "#CCCCCC")

                self.canvas.create_rectangle(
                    x1, y1, x2, y2,
                    fill=color, outline="#2C3E50", width=2
                )
                if (row, col) in self.board.queens:
                    cx = (x1 + x2) // 2
                    cy = (y1 + y2) // 2
                    size = self.cell_size // 3
                    self.canvas.create_text(
                        cx, cy, text="♛", font=("Arial", size, "bold"),
                        fill="#2C3E50"
                    )

    def load_file(self):
        filepath = filedialog.askopenfilename(
            title="Pilih file test case",
            filetypes=[("Text files", "*.txt"), ("All files", "*.*")]
        )
        if not filepath:
            return

        self.board = Board()
        result = self.board.load_from_file(filepath)
        if result == False:
            messagebox.showerror("Error", "File tidak valid!")
            self.board = None
            return
```

```
        self._assign_colors()
        self.draw_board()
        self.btn_solve.config(state=tk.NORMAL)
        self.btn_save.config(state=tk.DISABLED)
        self.label_status.config(text=f"Board       {self.board.n}x{self.board.n}
loaded. Pilih metode solve!")
        self.label_time.config(text="")
        self.label_iter.config(text="")

    def load_image(self):
        filepath = filedialog.askopenfilename(
            title="Pilih gambar papan Queens",
            filetypes=[
                ("Image files", "*.png *.jpg *.jpeg *.bmp *.gif"),
                ("All files", "*.*")
            ]
        )
        if not filepath:
            return

        self.board = Board()

        # Call the internal robust image loader
        result = load_board_from_image(self.board, filepath)

        if result == False:
            messagebox.showerror("Error",  "Gagal  membaca  gambar  atau  format
tidak valid!")
            self.board = None
            return

        self._assign_colors()
        self.draw_board()
        self.btn_solve.config(state=tk.NORMAL)
        self.btn_save.config(state=tk.DISABLED)
        self.label_status.config(
            text=f"Board {self.board.n}x{self.board.n} dari gambar. Pilih metode
solve!"
        )
        self.label_time.config(text="")
        self.label_iter.config(text="")

    def _slider_to_value(self, slider_pos):
        if slider_pos <= 0:
            return 1
        return  max(1,  min(500000,  int(round(math.exp(slider_pos  /  1000  *
math.log(500000))))))

    def _value_to_slider(self, value):
        if value <= 1:
            return 0
        return   max(0,   min(1000,   int(round(1000   *   math.log(value)   /
math.log(500000)))))

    def _on_slider_change(self, val):
        self.update_interval = self._slider_to_value(int(val))
```

```
        self.iter_entry.delete(0, tk.END)
        self.iter_entry.insert(0, str(self.update_interval))

    def _on_entry_change(self, event=None):
        try:
            val = int(self.iter_entry.get())
            val = max(1, min(500000, val))
        except ValueError:
            val = 1
        self.update_interval = val
        self.iter_entry.delete(0, tk.END)
        self.iter_entry.insert(0, str(val))
        self.speed_slider.set(self._value_to_slider(val))

    def start_solve(self, mode):
        self._current_mode = mode
        self.btn_solve.config(state=tk.DISABLED)
        self.btn_stop.config(state=tk.NORMAL)
        self.btn_load.config(state=tk.DISABLED)
        self.btn_load_img.config(state=tk.DISABLED)
        self.btn_save.config(state=tk.DISABLED)
        mode_label = "Brute Force" if mode == "brute_force" else "Optimized"
        self.label_status.config(text=f"Sedang           mencari           solusi
({mode_label})...", fg="#ECF0F1")
        self.label_time.config(text="Waktu berjalan: 0 ms")
        self.label_iter.config(text="")
        self.label_real_time.config(text="")
        self.board.queens = []
        self.solving = True
        self.stop_solving = False
        self.solve_start_time = time.time()
        self._real_time_shown = False
        self._result_queue = multiprocessing.Queue()
        self._silent_process = multiprocessing.Process(
            target=_silent_solve,
            args=(self.board.grid, self.board.n, self.board.regions,
                  mode, self._result_queue),
            daemon=True
        )
        self._silent_process.start()
        self._anim_thread    =    threading.Thread(target=self._animated_solver,
daemon=True)
        self._anim_thread.start()
        self._poll_board_state()

    def _poll_board_state(self):
        if not self._real_time_shown:
            try:
                result = self._result_queue.get_nowait()
                self.label_real_time.config(
                    text=f"Waktu eksekusi algoritma: {result['elapsed']:.4f} ms"
                )
                self.label_iter.config(
                    text=f"Banyak kasus yang ditinjau: {result['iterations']}"
                )
                self._real_time_shown = True
                self._real_found = result['found']
```

```
            except Exception:
                pass

        # If solver is still running, update time and schedule next poll
        if self.solving:
            self.draw_board()
            elapsed = (time.time() - self.solve_start_time) * 1000
            self.label_time.config(text=f"Waktu berjalan: {elapsed:.0f} ms")
            self.root.after(50, self._poll_board_state)
        elif not self._real_time_shown:
            # Even if solving is False, if results haven't appeared, keep
polling for a bit
            # This handles cases where the solver is extremely fast (< 50ms)
            self.root.after(50, self._poll_board_state)

    def _animated_solver(self):
        def on_update():
            if self.stop_solving:
                raise StopIteration("Solving stopped")
            time.sleep(0.02)

        def get_interval():
            return self.update_interval

        try:
            if self._current_mode == "brute_force":
                self.solver = BruteForceSolver(self.board)
                found           =           self.solver.solve(on_update=on_update,
get_interval=get_interval)
            else:
                self.solver = OptimizedSolver(self.board)
                found      =      self.solver.solve(0,      on_update=on_update,
get_interval=get_interval)
        except StopIteration:
            found = False
            self.board.queens = []

        self.solving = False
        elapsed = (time.time() - self.solve_start_time) * 1000

        if self.stop_solving:
            if self._silent_process.is_alive():
                self._silent_process.terminate()
            self.root.after(0, self.draw_board)
            self.root.after(0, self._enable_buttons, False)
        else:
            self.root.after(0, self.draw_board)
            self.root.after(0, self._show_final_results, found, elapsed)
            self.root.after(0, self._enable_buttons, found)

    def _show_final_results(self, found, total_elapsed):
        # Update metrics one last time if they haven't been captured yet
        if not self._real_time_shown:
            try:
                # Use a small timeout to let the silent process finish if it's
right behind
                result = self._result_queue.get(timeout=0.05)
```

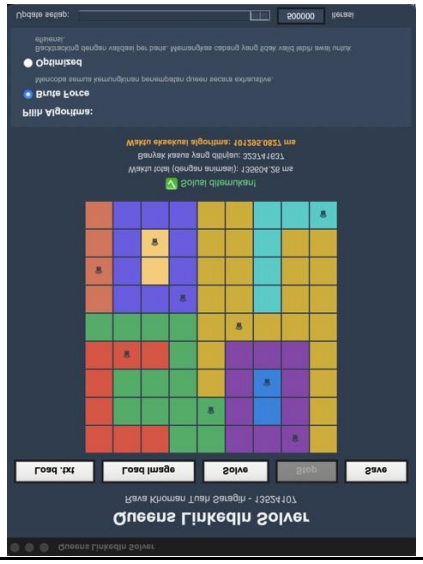
```
                self.label_real_time.config(text=f"Waktu    eksekusi    algoritma:
{result['elapsed']:.4f} ms")
                self.label_iter.config(text=f"Banyak    kasus    yang    ditinjau:
{result['iterations']}")
                self._real_time_shown = True
            except:
                pass

        if found:
            self.label_status.config(text="\u2705       Solusi       ditemukan!",
fg="#2ECC71")
        else:
            self.label_status.config(text="\u274c     Tidak     ada     solusi.",
fg="#E74C3C")
        self.label_time.config(text=f"Waktu      total      (dengan      animasi):
{total_elapsed:.2f} ms")

    def _enable_buttons(self, found):
        self.btn_load.config(state=tk.NORMAL)
        self.btn_load_img.config(state=tk.NORMAL)
        self.btn_solve.config(state=tk.NORMAL)
        self.btn_stop.config(state=tk.DISABLED)
        if found:
            self.btn_save.config(state=tk.NORMAL)



    def save_solution(self):
        filepath = filedialog.asksaveasfilename(
            title="Simpan solusi",
            defaultextension=".txt",
            filetypes=[
                ("Text files", "*.txt"),
                ("PNG Image", "*.png"),
                ("JPEG Image", "*.jpg"),
            ]
        )
        if not filepath:
            return

        if filepath.lower().endswith(('.png', '.jpg', '.jpeg')):
            self._save_as_image(filepath)
        else:
            self.board.save_solution(filepath)

        messagebox.showinfo("Berhasil", f"Solusi disimpan ke {filepath}")

    def _save_as_image(self, filepath):
        from PIL import Image, ImageDraw, ImageFont

        cell_size = 60
        border = 3
        n = self.board.n
        img_size = cell_size * n + border * (n + 1)

        img = Image.new("RGB", (img_size, img_size), "#2C3E50")
        draw = ImageDraw.Draw(img)
```

```
        for row in range(n):
            for col in range(n):
                x1 = border + col * (cell_size + border)
                y1 = border + row * (cell_size + border)
                x2 = x1 + cell_size
                y2 = y1 + cell_size

                region = self.board.grid[row][col]
                color = self.color_map.get(region, "#CCCCCC")

                draw.rectangle([x1, y1, x2, y2], fill=color)
                if (row, col) in self.board.queens:
                    self._draw_queen_icon(draw, x1, y1, cell_size)

        img.save(filepath)

    def _draw_queen_icon(self, draw, x, y, size):
        padding = size * 0.2
        w = size - 2 * padding
        h = size - 2 * padding
        cx = x + size / 2
        cy = y + size / 2
        color = "#2C3E50"
        base_h = h * 0.2
        base_y = cy + h/2 - base_h
        draw.rectangle(
            [cx - w/2, base_y, cx + w/2, cy + h/2],
            fill=color
        )
        draw.polygon([
            (cx - w/2, base_y),
            (cx - w/2 - w*0.1, cy - h/4),
            (cx - w/4, base_y)
        ], fill=color)
        draw.polygon([
            (cx + w/2, base_y),
            (cx + w/2 + w*0.1, cy - h/4),
            (cx + w/4, base_y)
        ], fill=color)
        draw.polygon([
            (cx - w/3, base_y),
            (cx, cy - h/2),
            (cx + w/3, base_y)
        ], fill=color)
        r = w * 0.06
        draw.ellipse([cx - w/2 - w*0.1 - r, cy - h/4 - r, cx - w/2 - w*0.1 + r,
cy - h/4 + r], fill=color)
        draw.ellipse([cx + w/2 + w*0.1 - r, cy - h/4 - r, cx + w/2 + w*0.1 + r,
cy - h/4 + r], fill=color)
        draw.ellipse([cx - r, cy - h/2 - r, cx + r, cy - h/2 + r], fill=color)

# main
if __name__ == "__main__":
    multiprocessing.freeze_support()
    root = tk.Tk()
    app = QueensGUI(root)
```

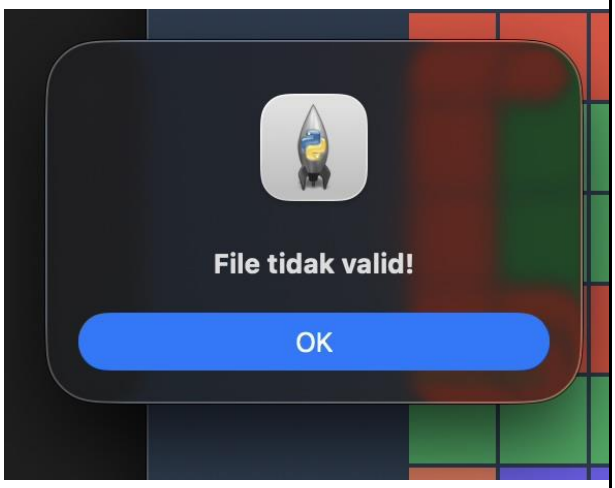
```
    root.mainloop()
```

Pengujian Test Case

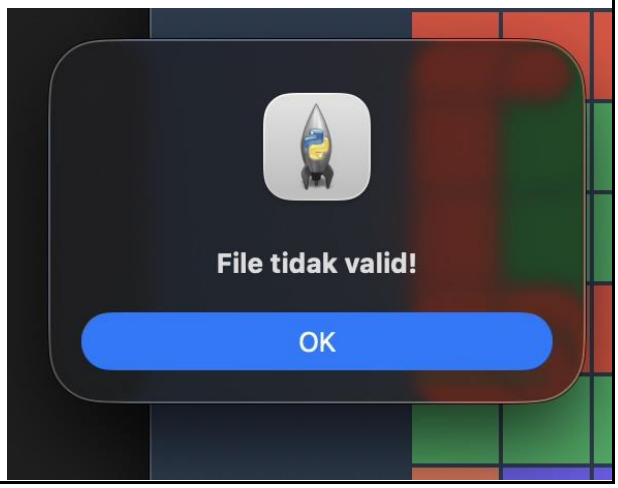
Test Case .txt #1

Input	Output
AAABBCCCD ABBBBCECD ABBBDCECD AAABDCCCD BBBBDDDDD FGGGDDHDD FGIGDDHDD FGIGDDHDD FGGGDDHHH	

Test Case .txt #2 (Satu baris memiliki satu elemen kurang)

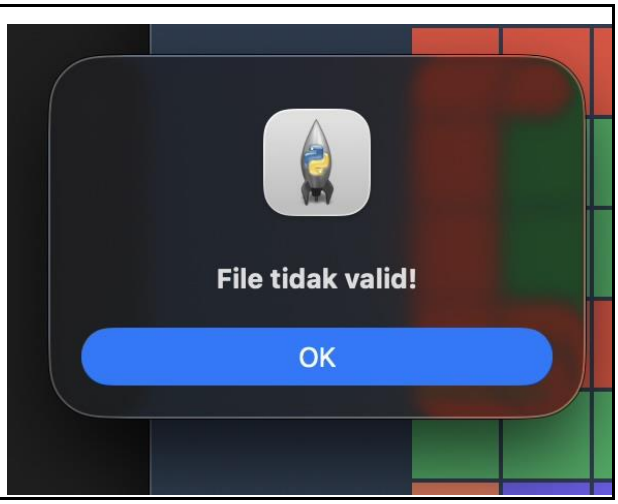
Input	Output
	

Test Case .txt #3 (Tidak persegi)

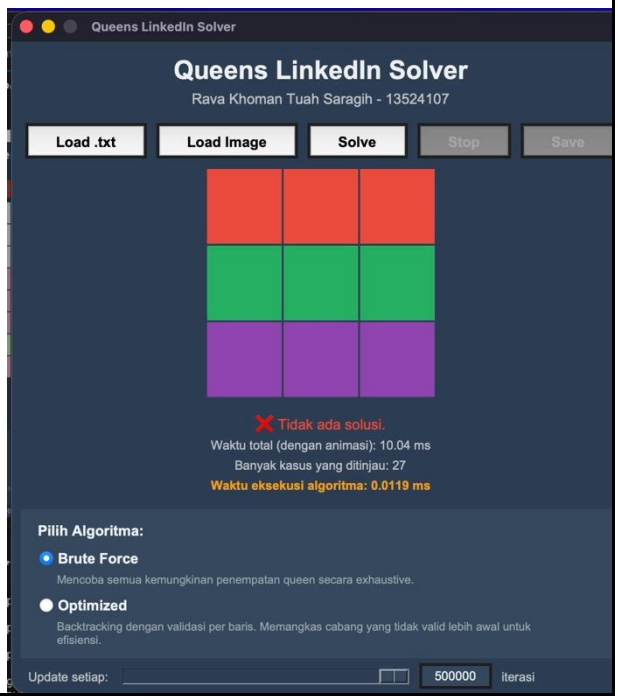
Input	Output
	

Test Case .txt #4 (Warna lebih banyak dari sisi grid)

Input	Output

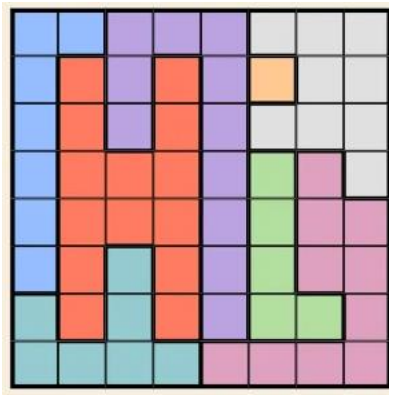
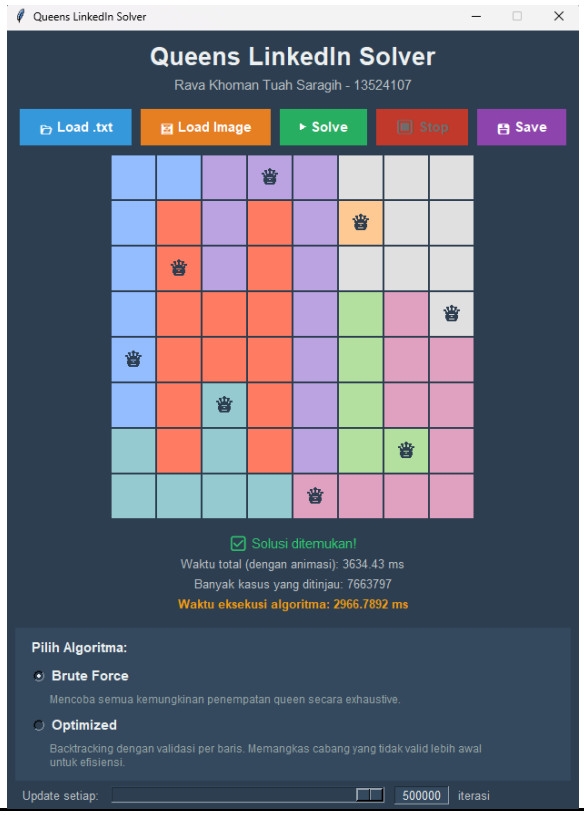
	

Test Case .txt #5 (Tidak ada solusi)

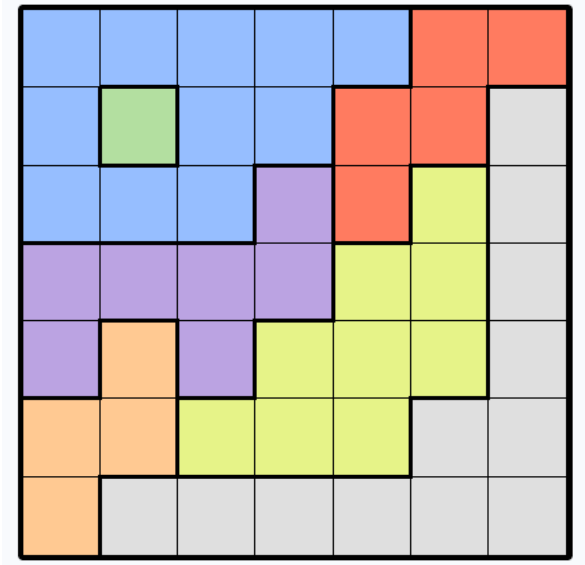
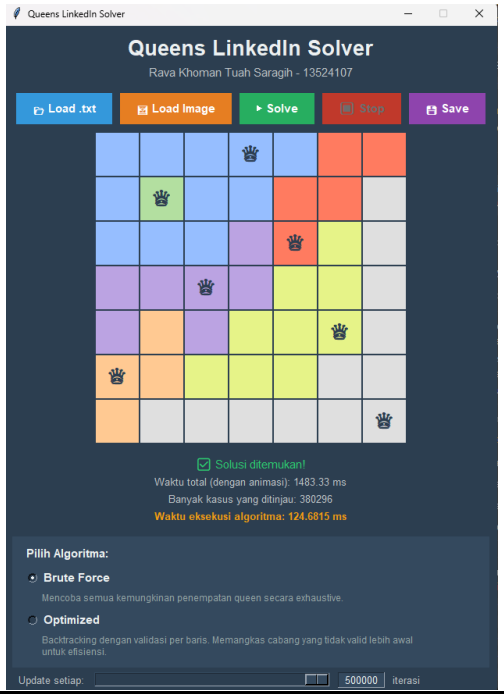
Input	Output
	

Test Case Image #1

Input	Output

Test Case Image #2

Input	Output
	

Lampiran

No	Poin	Ya	Tidak
1	Program berhasil dikompilasi tanpa kesalahan	✓	
2	Program berhasil dijalankan	✓	
3	Solusi yang diberikan program benar dan mematuhi aturan permainan	✓	
4	Program dapat membaca masukan berkas .txt 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.

Rava Khoman Tuah Saragih

Tautan Repositori: https://github.com/ravasrgh/Tucil1_13524107