Michael James Liman
13524106
K-02

A. Algoritma Brute Force

```
function checkOutputValid():
    if(isEmpty(coordiantes)) → False
    if(len(coordinates) != self.num_regions → false
    i traversal [0..len(coordinates)] @1
        j traversal[i+1..len(coordinates] @1
            if(coordinates[i][0] == coordinates[j][0])→false @1a
            if(coordinates[i][1] = coordinates[j][1])→false @1b
            if(abs(coordinates[i][0] - coordinates[j][0]) = 1 and
abs(coordinates[i][1] - coordinates[j][1]) = 1)→false @1c

    i traversal [0..len(coordinates] @2
        if self.arr[coordinates[i][0]][coordinates[i][1]] !=
    self.regions_list[i] → False

    →true

function solve(region_idx):
    if(region_idx >= num_regions): #1
        self.gui.queens ← self.coordinate
        → checkOutputValid()
    curr_region ← region_list[region_idx] #2
    i traversal [0..rows]
        j traversal [i..cols]:
            original_value ← ans[i][j]
            ans[i][j] ← 'X' #3
            coordinates.append([i,j])

            if (solve(region_idx+1)) → true #4
            coordinates.pop()
            ans[i][j] ← original_value
```

Algoritma *brute force* pada program dilakukan dengan dua fungsi utama, yaitu solve untuk mengiterasi dan melakukan backtrack pada board dan checkOutputValid untuk melakukan validasi apakah kombinasi sesuai. Iterasi brute force yang menghasilkan semua kombinasi mungkin pada *search space* dilakukan dengan 3 buah iterasi, yaitu iterasi terhadap setiap regions pada input, yang dilakukan dengan *recursive*; kemudian iterasi per baris; dan terakhir per kolom. Program menawarkan dua opsi, yaitu full search dan optimized search. Full search tidak peduli peletakan queens pada sel manapun, meskipun cell tersebut bukan merupakan cell regionnya. Sebaliknya, optimized search melakukan optimasi minim pada iterasi sehingga hanya meletakkan queens pada region sesuai iterasi.

Algoritma yang dibahas lebih lanjut, termasuk yang dipaparkan di atas, adalah versi full search. Iterasi bersarang paling pertama menggunakan rekursif dengan base case pada #1, sehingga pengecekan dipanggil ketika queens untuk semua region telah diletakkan dan pencarian selesai ketika pengecekan benar (#4). Untuk menangani input dengan region yang tidak berurutan, maka terdapat sebuah list of region yang diakses dengan indexnya untuk mendapatkan nilai region yang asli pada #2. Algoritma ini juga menggunakan backtracking untuk mengembalikkan ke state semula ketika terdapat langkah yang salah/tidak valid.

Selanjutnya, pengecekan validitas kombinasi queens dilakukan oleh fungsi checkOutputValid. Fungsi tersebut melakukan searching terhadap kondisi-kondisi yang salah pada kombinasi yang valid. Kombinasi tersebut adalah terdapat queen lain pada row(@1a) atau kolom(@1b) yang sama, serta terdapat queen lain yang bersebelahan secara diagonal(@1c). Kemudian, algoritma juga melakukan validasi apakah penempatan queen telah sesuai dengan regionnya(@2).

B. Source Code

```
src/main.py
# ------------------------------ Initialization ------------------------------ #
import copy
import pprint
import time
import threading
from gui import GUI

# Global variables
arr = None
solver = None
solver_thread = None
on_exit = threading.Event()

def load_file(file_path):
    global arr
    try:
        with open(file_path, "r") as f:
            text = f.read()
        lines = text.split('\n')
        arr = [list(x.strip()) for x in lines if x.strip()]
        print(f"Loaded file: {file_path}")
        print(arr)
        return arr
    except Exception as e:
        print(f"Error loading file: {e}")
        return None

def save_file(file_path, ans):
    try:
        with open(file_path, "w") as f:
            for row in ans:
                f.write(''.join(row) + '\n')
        print(f"Saved file: {file_path}")
    except Exception as e:
        print(f"Error saving file: {e}")

class Algo:
    def __init__(self, arr, gui=None):
        self.arr = [[ord(item) - ord('A') for item in rows] for rows in arr]
        self.ans = arr
        self.coordinate = list()
        unique_regions = set()
        for row in self.arr:
            for cell in row:
                unique_regions.add(cell)

        self.regions_list = sorted(list(unique_regions))
        self.num_regions = len(self.regions_list)
        self.region = {region: False for region in self.regions_list}

        self.rows = len(arr)
        self.cols = len(arr[0])
        self.gui = gui

        self.optimizer = False

    def checkInputValid(self):
        if(self.num_regions >= self.rows or self.num_regions >= self.cols): return False

    # --------------------------------- Main Algo -------------------------------- #
    def checkOutputValid(self):
        coordinates = self.coordinate
        if coordinates == []: return False
        if len(coordinates) != self.num_regions: return False
        for i in range(len(coordinates)):
            for j in range(i + 1, len(coordinates)):
                if(coordinates[i][0] == coordinates[j][0]): return False
                if(coordinates[i][1] == coordinates[j][1]): return False
                if(abs(coordinates[i][0] - coordinates[j][0]) == 1 and abs(coordinates[i][1] -
coordinates[j][1]) == 1): return False

        if not self.optimizer:
            for i in range(len(coordinates)):
                if self.arr[coordinates[i][0]][coordinates[i][1]] != self.regions_list[i]: return
False

        return True

    def solve(self, region_idx):
        if on_exit.is_set():
            return False

        if(region_idx >= self.num_regions):
            if self.gui:
                self.gui.queens = self.coordinate
            return self.checkOutputValid()

        curr_region = self.regions_list[region_idx]

        ans = self.ans
        region = self.region
        coordinates = self.coordinate
        for i in range(self.rows):
            for j in range(self.cols):
                if(not self.optimizer or self.arr[i][j] == curr_region):
                    ans[i][j] = "X"
                    region[curr_region] = True
                    coordinates.append([i, j])

                    if self.solve(region_idx+1): return True

                    coordinates.pop()
                    region[curr_region] = False
                    ans[i][j] = chr(self.arr[i][j] + ord('A'))
        return False
```

```
def run_solver(solver):
    start_time = time.perf_counter()
    result = solver.solve(0)
    end_time = time.perf_counter()
    if(not on_exit.is_set()):
        print(f"Solution found: {result}")
        print("Time Elapsed:", f"{((end_time-start_time)*1000):.4f}", "ms")
        print("Coordinates:", solver.coordinate)
        save_file("test/output.txt", solver.ans)

def on_import_callback(file_path):
    global arr, solver, on_exit
    arr = load_file(file_path)
    if arr:
        solver = Algo(arr, gui=gui)
        on_exit.set()
        gui.queens = []
        gui.rows = len(arr)
        gui.cols = len(arr[0])
        gui.arr = solver.arr
        gui.regions = solver.regions_list
        gui.generate_colors()


def on_solve_callback(optimized=False):
    global solver, solver_thread, arr

    if arr is None:
        print("Please import a file first")
        return

    if solver_thread and solver_thread.is_alive():
        print("Solver already running")
        return

    solver.optimizer = optimized

    on_exit.clear()
    solver_thread = threading.Thread(target=run_solver, args=(solver,), daemon=True)
    solver_thread.start()

gui = GUI(5, 5)
gui.on_import = on_import_callback
gui.on_solve = on_solve_callback

gui.run()
```

```
src/gui.py
import pygame
import tkinter as tk
from tkinter import filedialog
import random

class GUI:
    def __init__(self, rows, cols):
        pygame.init()

        self.WIDTH = 1000
        self.HEIGHT = 600
        self.rows = rows
        self.cols = cols

        self.screen = pygame.display.set_mode([self.WIDTH, self.HEIGHT])
        pygame.display.set_caption("LinkedIn's Queens Game Solver")

        self.queens = []
        self.on_import = None
        self.on_solve = None

        self.font = pygame.font.Font('freesansbold.ttf', 20)
        self.medium_font = pygame.font.Font('freesansbold.ttf', 40)
        self.big_font = pygame.font.Font('freesansbold.ttf', 50)

        self.timer = pygame.time.Clock()
        self.fps = 60

        self.import_button = pygame.Rect(self.WIDTH - 180, 20, 160, 50)
        self.solve_full_button = pygame.Rect(self.WIDTH - 180, 90, 160, 50)
        self.solve_opt_button = pygame.Rect(self.WIDTH - 180, 160, 160, 100)

        self.regions = None
        self.colors = None
        self.arr = None

    def generate_colors(self):
        if not self.regions:
            return

        self.colors = {}
        for region in self.regions:
            self.colors[region] = (random.randint(0, 255), random.randint(0, 255), random.randint(0,
255))

    def draw_board(self):
        cell_size = min(self.WIDTH // self.cols, self.HEIGHT // self.rows)
        # Draw checkerboard pattern
        for row in range(self.rows):
            for col in range(self.cols):
                x = col * cell_size
                y = row * cell_size
                if self.colors and self.regions is not None:
                    region = self.arr[row][col]
                    color = self.colors.get(region, (200, 200, 200))
                    pygame.draw.rect(self.screen, color, [x, y, cell_size, cell_size])
                else :
                    if (row + col) % 2 == 0:
                        pygame.draw.rect(self.screen, 'light gray', [x, y, cell_size, cell_size])

        # Draw grid lines
        for i in range(self.rows + 1):
            pygame.draw.line(self.screen, 'black', (0, cell_size * i), (self.cols * cell_size,
cell_size * i), 2)
        for i in range(self.cols + 1):
            pygame.draw.line(self.screen, 'black', (cell_size * i, 0), (cell_size * i, self.rows *
cell_size), 2)

    def draw_queens(self):
        queens = self.queens
        cell_size = min(self.WIDTH // self.cols, self.HEIGHT // self.rows)
        for queen in queens:
            row, col = queen
            x = col * cell_size + cell_size // 2
            y = row * cell_size + cell_size // 2
            pygame.draw.circle(self.screen, 'red', (x, y), cell_size // 3)

    def draw_buttons(self):
        # Draw import button
        pygame.draw.rect(self.screen, 'light blue', self.import_button)
        import_text = self.font.render('Import TXT', True, 'black')
        text_rect = import_text.get_rect(center=self.import_button.center)
        self.screen.blit(import_text, text_rect)

        # Draw solve full button
        pygame.draw.rect(self.screen, 'light green', self.solve_full_button)
        solve_text = self.font.render('Solve Full', True, 'black')
        text_rect = solve_text.get_rect(center=self.solve_full_button.center)
        self.screen.blit(solve_text, text_rect)

        # Draw solve optimized button
        pygame.draw.rect(self.screen, 'light green', self.solve_opt_button)
        solve_text1 = self.font.render('Solve', True, 'black')
        solve_text2 = self.font.render('Optimized', True, 'black')
        text_rect1 = solve_text1.get_rect(center=(self.solve_opt_button.centerx,
self.solve_opt_button.centery - 15))
        text_rect2 = solve_text2.get_rect(center=(self.solve_opt_button.centerx,
self.solve_opt_button.centery + 15))
        self.screen.blit(solve_text1, text_rect1)
        self.screen.blit(solve_text2, text_rect2)

    def handle_import(self):
        root = tk.Tk()
        root.withdraw()

        file_path = filedialog.askopenfilename(
            title="Select a text file",
            filetypes=[("Text files", "*.txt"), ("All files", "*.*")]
        )

        root.destroy()

        if file_path and self.on_import:
            self.on_import(file_path)

    def handle_solve(self, optimized=False):
        if self.on_solve:
            self.on_solve(optimized=optimized)

    def run(self):
        run = True
        while run:
            self.timer.tick(self.fps)
            self.screen.fill('white')

            # Event Handler
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    run = False
                elif event.type == pygame.MOUSEBUTTONDOWN:
                    if event.button == 1:
                        if self.import_button.collidepoint(event.pos):
                            self.handle_import()
                        elif self.solve_full_button.collidepoint(event.pos):
                            self.handle_solve(optimized=False)
                        elif self.solve_opt_button.collidepoint(event.pos):
                            self.handle_solve(optimized=True)

            self.draw_board()
            self.draw_queens()
            self.draw_buttons()

            pygame.display.flip()

        pygame.quit()
```

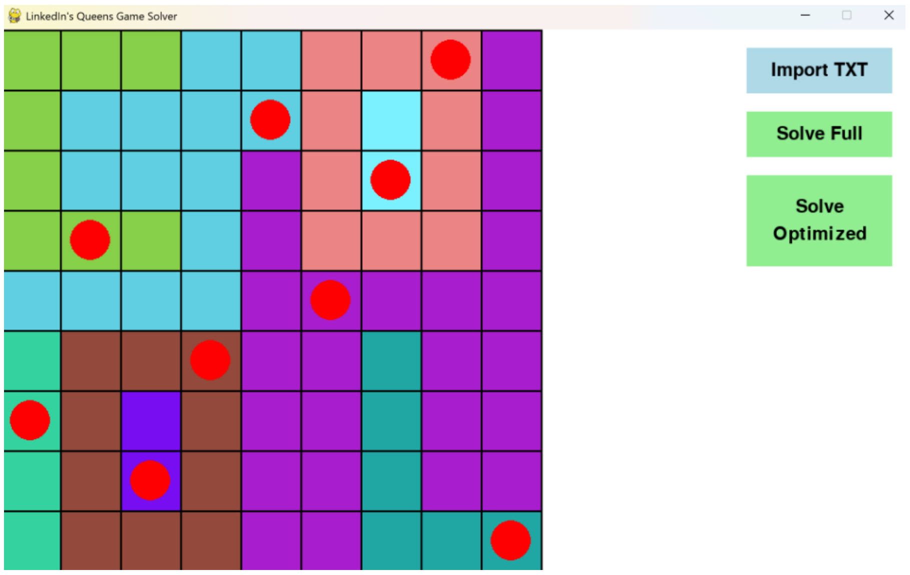
C. Tangkapan Layar

a. Test Case 1

Input :

AAABBCCCD
ABBBBCECD
ABBBDCECD
AAABDCCCD
BBBBDDDDD
FGGGDDHDD
FGIGDDHDD
FGIGDDHDD
FGGGDDHHH

Output :

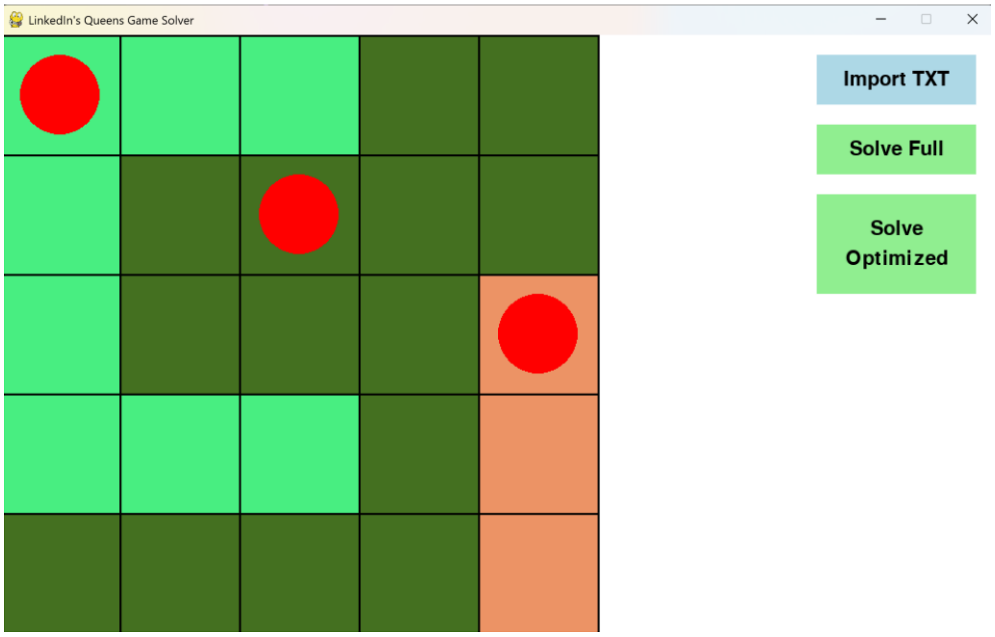


b. Test Case 2

Input :

AAABB
ABBBB
ABBBC
AAABC
BBBBC

Output :



c. Test Case 3

Input :

ZZZZZ

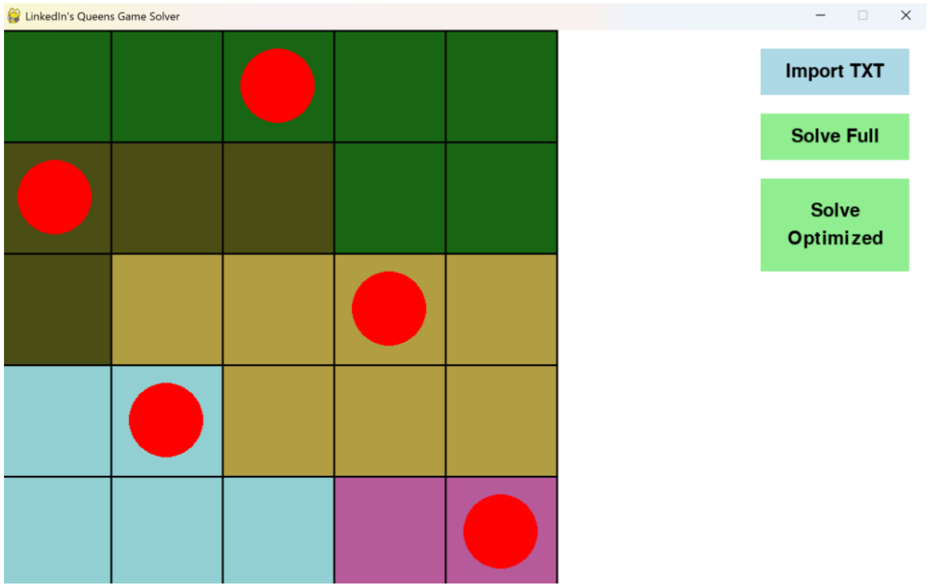
BBBZZ

BCCCC

DDCCC

DDDEE

Output :



d. Test Case 4

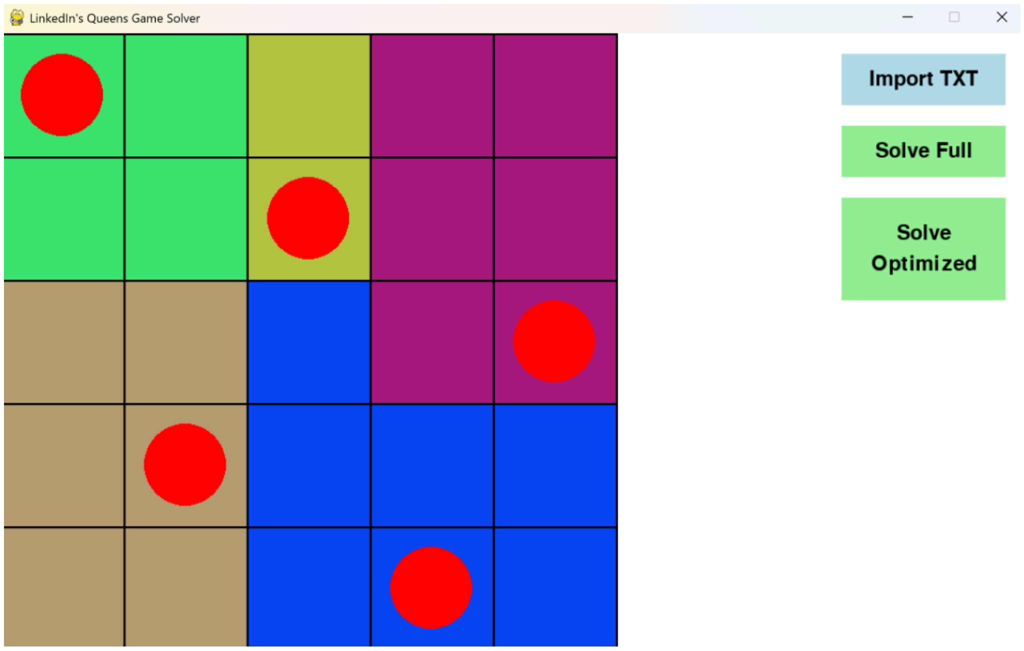
Input :

AABCC

AABCC

DDECC
DDEEE
DDEEE

Output :

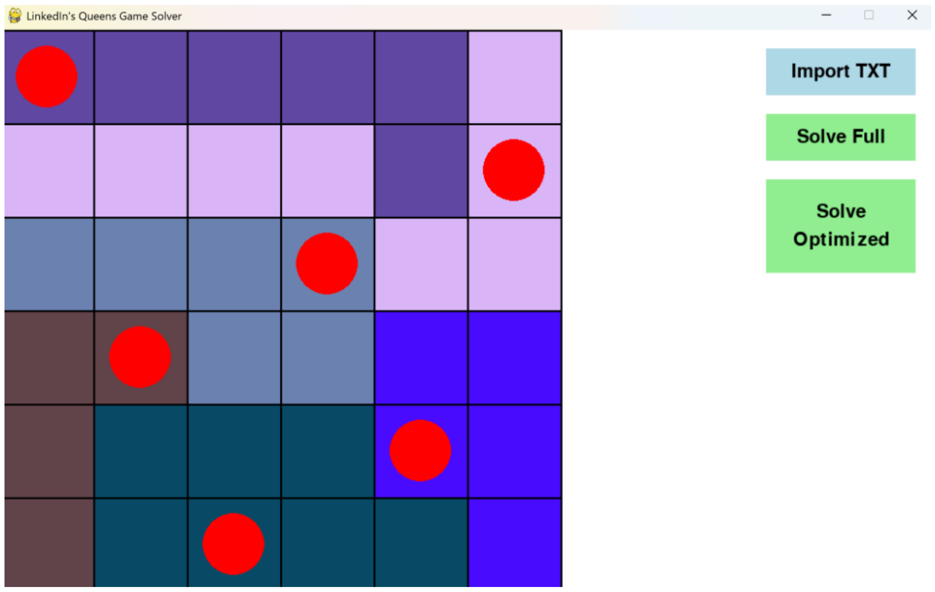


e. Test Case 5

Input :

AAAAAB
BBBBAB
CCCCBB
DDCCEE
DFFFEE
DFFFFE

Output :



D. Lampiran

a. Pernyataan tidak melakukan kecurangan

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

[illegible]

Michael James Liman

b. Link Repository: https://github.com/MichaelJamesL/queens-linkedin-solver.git

c. Tabel Capaian

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		✓