Laporan Tugas Kecil 1 IF2211 Strategi Algoritma
Queens LinkedIn Solver

Muhammad Haris Putra Sulastianto
13524053

13524053@std.stei.itb.ac.id
mharisputras.work@gmail.com

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

2026

Daftar Isi

1 Permasalahan

Permainan Queens pada platform LinkedIn merupakan permainan logika berbasis papan berwarna. Diberikan sebuah papan berukuran $N \times N$ yang terbagi ke dalam N wilayah warna (region), tujuan permainan adalah menempatkan tepat N buah ratu sehingga seluruh aturan berikut terpenuhi:

- Setiap baris hanya boleh berisi tepat satu ratu.
- Setiap kolom hanya boleh berisi tepat satu ratu.
- Setiap wilayah warna (region) hanya boleh berisi tepat satu ratu.
- Tidak ada dua ratu yang boleh saling bertetangga (1 petak), termasuk secara diagonal.

Jumlah ratu yang ditempatkan sama dengan jumlah baris, kolom, dan wilayah warna (N).

2 Pendekatan Solusi

Pendekatan paling sederhana untuk menyelesaikan permainan Queens adalah menggunakan metode brute force, yaitu mencoba semua kemungkinan penempatan ratu hingga ditemukan konfigurasi yang memenuhi seluruh aturan atau tidak ditemukan solusi sama sekali. Pendekatan ini dapat direpresentasikan sebagai proses pencarian dalam ruang solusi dengan mengeksplorasi setiap kemungkinan peletakan ratu pada papan berwarna.

3 Implementasi

Program Queens LinkedIn Solver bertugas memproses masukan berupa papan permainan berwarna untuk menentukan konfigurasi penempatan ratu yang valid. Jika solusi ditemukan, program akan menampilkannya, sebaliknya, program akan menyatakan bahwa tidak terdapat solusi yang memenuhi aturan.

Pencarian solusi dilakukan menggunakan pendekatan algoritma brute force. Namun, untuk mempercepat proses, sistem menyediakan fitur 'mode efisien' yang mampu mengeliminasi ruang pencarian, sehingga mekanisme pencarian tidak bergantung sepenuhnya pada brute force murni. Implementasi perangkat lunak dilakukan dalam bentuk aplikasi desktop menggunakan bahasa C++ dan framework Qt, dengan dukungan masukan dan keluaran berformat berkas teks (.txt) serta gambar (.jpg .png).

3.1 Struktur Data dan Arsitektur

Struktur program dibangun di atas dua kelas utama, yaitu Board sebagai model data dan Solver sebagai pengelola logika pencarian.

3.1.1 Kelas Board

Kelas ini bertugas menyimpan representasi memori dari papan permainan dan aturan validasinya. Implementasi struktur data ini dapat dilihat pada Kode 1.

```
#pragma once
#include <vector>
#include <set>
using namespace std;

class Board
{
private:
    int n;
    vector<vector<int>> grid;
    vector<vector<int>> color;

public:
    Board(int n, const vector<vector<int>> &color);

    int countColorId() const;

    void placeQueen(int row, int col);
    void removeQueen(int row, int col);
    bool isValidWholeBoard() const;

    const vector<vector<int>> &getGrid() const;
    const vector<vector<int>> &getColor() const;
    int getSize() const;
};
```

Listing 1: Header file Board.h

Atribut

Data papan permainan disimpan dalam dua variabel utama. Variabel `grid` digunakan untuk mencatat kotak mana yang sedang terisi ratu dan mana yang kosong. Variabel `color` menyimpan informasi warna untuk setiap kotak tersebut. Selain itu, ukuran lebar dan tinggi papan disimpan dalam variabel `n`.

Metode

Untuk mengubah isi papan, program menggunakan fungsi `placeQueen` (untuk menaruh ratu) dan `removeQueen` (untuk menghapus ratu). Pengecekan apakah posisi ratu sudah aman dan sesuai aturan dilakukan oleh fungsi `isValidWholeBoard`. Terakhir, fungsi `countColorId` digunakan hanya untuk menghitung ada berapa banyak warna di papan tersebut.

3.1.2 Kelas Solver

Kelas ini mewarisi `QObject` (dari framework Qt), yang berfungsi sebagai mesin pencari solusi dan penghubung ke antarmuka pengguna. Implementasi struktur data ini dapat dilihat pada Kode 2.

```
#pragma once
#include <algorithm>
#include <QElapsedTimer>
#include <QObject>
#include "Board.h"

class Solver : public QObject
{
    Q_OBJECT
private:
    Board board;
    QElapsedTimer frameTimer;

    long long iterationCount = 0;

    std::atomic<bool> stopRequested{false};

    bool solutionFound = false;
    bool efficientMode = false;

public:
    explicit Solver(const Board &board, bool efficientMode);

    const Board &getBoard() const;
    long long getIterationCount() const;
    bool getSolFound() const;
    void recordIteration();

    void requestStop() { stopRequested = true; }

public slots:
    void solve();

signals:
    void boardUpdated(const Board &snapshot, long long iteration);
    void finished(const Board result, long long iterationCount,
    bool solutionFound, qint64 time);
    void progress(long long iteration);
};
```

Listing 2: Header file Solver.h

Manajemen Status dan Waktu

Manajemen status internal difokuskan pada pengelolaan objek `board` sebagai data utama, serta penggunaan variabel kontrol `efficientMode` dan `stopRequested` untuk keamanan. Sementara itu, pengukuran kinerja algoritma dilakukan menggunakan `frameTimer` dengan penghitung langkah `iterationCount`.

Mekanisme Sinyal dan Slot (Qt)

Interaksi dengan antarmuka pengguna ditangani melalui mekanisme sinyal dan slot Qt. Fungsi slot `solve()` bertugas menjalankan logika utama, sedangkan komunikasi asinkron melalui sinyal `boardUpdated` dan `finished` memastikan pembaruan visual berjalan lancar tanpa membekukan aplikasi utama.

3.2 Algoritma

3.2.1 Algoritma Validator Board

Implementasi algoritma ini dapat dilihat pada Kode 3.

```
bool Board::isValidWholeBoard() const
{
    // queen
    int qCount = 0;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (grid[i][j] == 1)
            {
                qCount++;
            }
        }
    }

    if (qCount != n)
    {
        return false;
    }

    // row
    for (int i = 0; i < n; i++)
    {
        int count = 0;
        for (int j = 0; j < n; j++)
        {
            if (grid[i][j] == 1)
            {
                count++;
                if (count > 1)
                {
                    return false;
                }
            }
        }

        if (count != 1)
        {
            return false;
        }
    }
```

```
42
43      // column
44      for (int j = 0; j < n; j++)
45      {
46          int count = 0;
47          for (int i = 0; i < n; i++)
48          {
49              if (grid[i][j] == 1)
50              {
51                  count++;
52                  if (count > 1)
53                  {
54                      return false;
55                  }
56              }
57          }
58
59          if (count != 1)
60          {
61              return false;
62          }
63      }
64
65      // adjacent diagonally
66      for (int i = 0; i < n; i++)
67      {
68          for (int j = 0; j < n; j++)
69          {
70              if (grid[i][j] == 1)
71              {
72                  if (i < n - 1 && j < n - 1 && grid[i + 1][j + 1] ==
    1)
73                  {
74                      return false;
75                  }
76                  if (i < n - 1 && j > 0 && grid[i + 1][j - 1] == 1)
77                  {
78                      return false;
79                  }
80              }
81          }
82      }
83
84      // color
85      int maxColorId = countColorId();
86      vector<int> countColor(maxColorId, 0);
87
88      for (int i = 0; i < n; i++)
89      {
90          for (int j = 0; j < n; j++)
91          {
92              if (grid[i][j] == 1)
93              {
94                  int c = color[i][j];
95                  countColor[c]++;
96
```

```
                if (countColor[c] != 1)
                {
                    return false;
                }
            }
        }
    }

    for (int c = 0; c < maxColorId; c++)
    {
        if (countColor[c] != 1)
        {
            return false;
        }
    }

    return true;
}
```

Listing 3: Validator papan

Kompleksitas Algoritma :

$$O(N^2)$$

Kompleksitas fungsi validasi adalah $O(N^2)$ karena seluruh papan dipindai beberapa kali. Walaupun ada beberapa loop terpisah (untuk baris, kolom, dan warna), semuanya tetap berbasis ukuran papan $N \times N$.

Langkah - Langkah

Langkah pertama adalah menghitung total ratu yang ada di atas papan. Program akan memeriksa seluruh kotak untuk memastikan jumlah ratu sama persis dengan ukuran papan (N). Jika jumlah ratu kurang atau lebih dari yang seharusnya, maka susunan papan langsung dianggap salah. Ini adalah cara cepat untuk membuang kemungkinan jawaban yang sudah pasti keliru sebelum mengecek aturan lainnya.

Selanjutnya, program memeriksa posisi ratu di setiap baris dan kolom. Program memastikan bahwa setiap baris dan setiap kolom hanya berisi tepat satu ratu. Jika ditemukan ada baris atau kolom yang kosong, maka susunan papan dianggap tidak sah. Aturan ini memastikan ratu tersebar merata secara horizontal maupun vertikal.

Setelah itu, program mengecek apakah ada ratu yang saling bersentuhan. Program akan melihat sekeliling posisi setiap ratu untuk memastikan tidak ada ratu lain di dekatnya. Jika ada dua ratu yang posisinya saling berdempetan bahkan secara diagonal, maka aturan dilanggar.

Terakhir, program memeriksa aturan wilayah warna. Setiap warna atau area di papan harus memiliki tepat satu ratu. Program akan mendata warna apa saja yang sudah terisi oleh ratu. Jika ada warna yang terlewat (kosong) maka papan dinyatakan salah. Jika semua syarat dari awal sampai akhir ini terpenuhi, barulah solusi dianggap benar.

3.2.2 Algoritma Brute Force Murni

Pendekatan ini menganggap papan permainan sebagai sekumpulan sel biner (kosong atau berisi ratu) dan mencoba setiap kemungkinan kombinasi penempatan ratu di seluruh sel papan (NxN) tanpa mempedulikan aturan baris atau kolom sejak awal. Implementasi algoritma ini dapat dilihat pada Kode 4.

```
int totalSize = n * n;
while (true)
{
    if (stopRequested)
        break;

    recordIteration();

    if (board.isValidWholeBoard())
    {
        solutionFound = true;
        break;
    }

    bool keepGoing = true;

    for (int k = totalSize - 1; k >= 0 && keepGoing; k--)
    {
        int row = k / n;
        int col = k % n;

        if (board.getGrid()[row][col] == 0)
        {
            board.placeQueen(row, col);
            keepGoing = false;
        }
        else
        {
            board.removeQueen(row, col);
        }
    }

    if (keepGoing)
    {
        break;
    }
}
```

Listing 4: Brute force murni

Kompleksitas Algoritma :

$$O(2^{N^2})$$

Algoritma ini memiliki kompleksitas eksponensial yang sangat tinggi karena memperlakukan papan permainan sebagai sebuah bilangan biner raksasa.

Langkah - Langkah

Algoritma dimulai dari papan keadaan kosong dan sudah memiliki region warna.

Selanjutnya menghitung total sel yaitu `totalSize = n*n`.

Program masuk ke loop yang merepresentasikan satu siklus pengecekan papan dengan jumlah iterasi pengecekan di hitung di variabel `iterationCount`.

Pada setiap iterasi, program memanggil `board.isValidWholeBoard()`. Jika konfigurasi papan saat ini valid (memenuhi aturan warna, baris, kolom, dan tidak bersinggungan), maka solusi dianggap ditemukan (solutionFound = true), dan proses berhenti.

Jika ternyata konfigurasi papan belum valid, algoritma akan membuat konfigurasi papan baru. Caranya mirip dengan cara kerja penambahan angka biner. Program melakukan mengecek sel dari posisi paling akhir (pojok kanan bawah) mundur ke posisi awal (pojok kiri atas): Jika sel kosong (0) maka tempatkan ratu di sana (`placeQueen`), lalu hentikan pemindaian untuk iterasi ini. Jika sel terisi ratu (1) hapus ratu di sana (`removeQueen`) dan lanjutkan pemindaian ke sel sebelumnya.

Mekanisme ini memastikan bahwa algoritma menelusuri setiap kombinasi penempatan ratu yang mungkin ada, mulai dari 0 ratu hingga papan penuh.

3.2.3 Algoritma Brute Force Berbasis Permutasi

Pendekatan ini jauh lebih teroptimasi dibandingkan metode pertama. Algoritma ini membatasi ruang pencarian dengan asumsi dasar: Setiap baris pasti memiliki tepat satu ratu. Algoritma ini digunakan di mode efisien.

```
vector<int> permutation(n);
for (int i = 0; i < n; i++)
{
    permutation[i] = i;
}

do
{
    if (stopRequested)
        break;

    recordIteration();

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            board.removeQueen(i, j);
        }
    }

    for (int row = 0; row < n; row++)
    {
        board.placeQueen(row, permutation[row]);
    }

    if (board.isValidWholeBoard())
    {
```

```
29          solutionFound = true;
30          break;
31      }
32  } while (next_permutation(permutation.begin(), permutation.end()));
```

Listing 5: Brute force permutasi

Kompleksitas Algoritma :

$$O(N! \times N^2)$$

Algoritma ini memiliki kompleksitas faktorial ($N!$). Kompleksitas total menjadi

$$O(N! \times N^2)$$

berasal dari jumlah permutasi, sedangkan N^2 berasal dari proses validasi yang dilakukan di tiap iterasi.

Langkah - Langkah

Algoritma memulai dengan membuat sebuah array berisi urutan kolom: $[0, 1, 2, ..., N-1]$. Array ini merepresentasikan posisi kolom ratu untuk setiap baris. Contoh: Jika indeks ke-0 bernilai 2, berarti pada Baris 0, ratu diletakkan di Kolom 2.

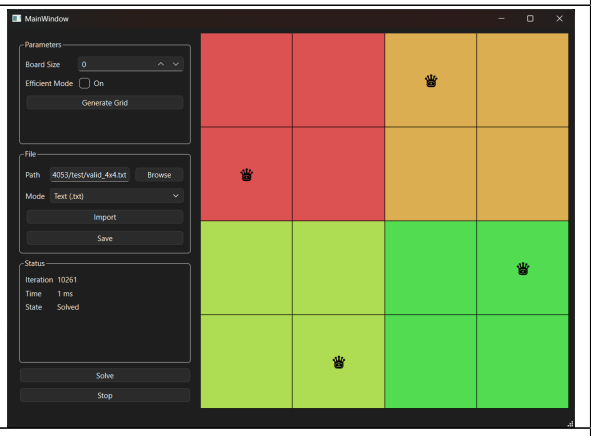
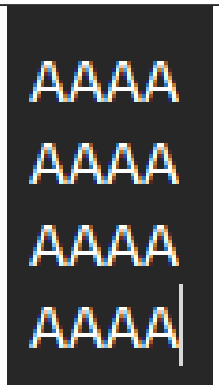
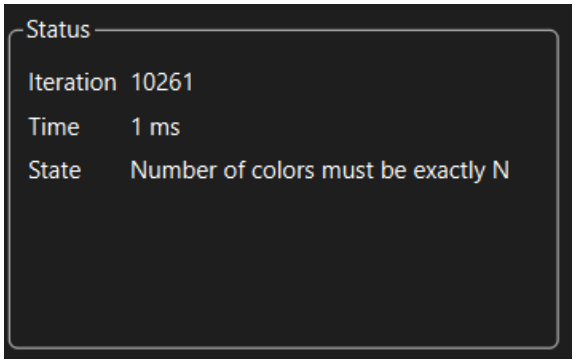
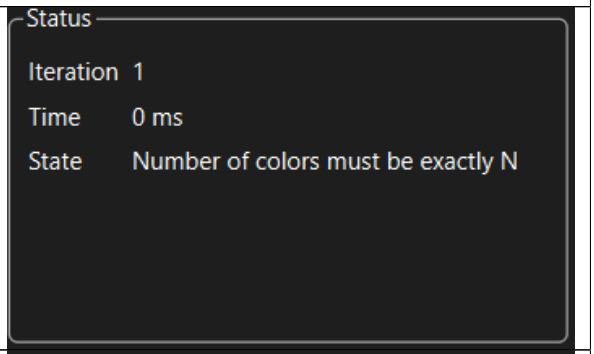
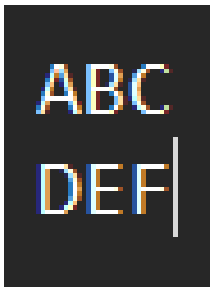
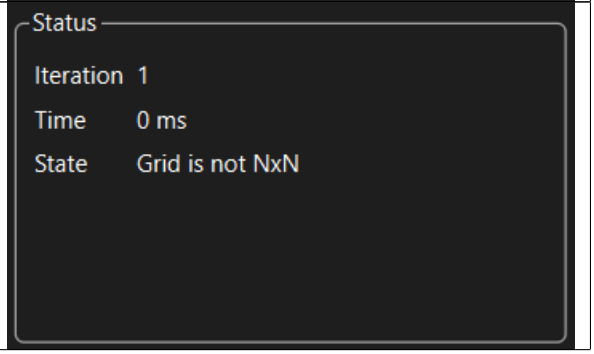
Selanjutnya, program memasuki siklus pencarian utama menggunakan struktur perulangan `do...while` yang memanfaatkan fungsi `std::next_permutation`. Fungsi pustaka standar ini bertugas untuk menghasilkan kombinasi urutan unik berikutnya secara leksikografis dari array tersebut, misalnya mengubah urutan dari $[0, 1, 2]$ menjadi $[0, 2, 1]$, lalu $[1, 0, 2]$, dan seterusnya.

Pada setiap iterasi baru, sistem melakukan rekonstruksi papan dengan terlebih dahulu membersihkan seluruh penempatan ratu sebelumnya. Ratu kemudian ditempatkan kembali sesuai dengan konfigurasi array permutasi yang sedang aktif, di mana baris ke-i akan diisi ratu pada kolom yang ditunjukkan oleh nilai elemen ke-i dari array. Pendekatan ini secara otomatis menjamin bahwa tidak ada dua ratu dalam satu baris maupun satu kolom yang sama, karena setiap angka dalam array bersifat unik.

Setelah penempatan selesai, program memvalidasi konfigurasi dengan memanggil fungsi `board.isValidWholeBoard()`. Mengingat aturan baris dan kolom telah terpenuhi secara implisit oleh logika permutasi, fungsi validasi ini secara efektif hanya memeriksa pelanggaran terhadap aturan wilayah warna dan persinggungan diagonal antar ratu. Jika konfigurasi terbukti valid, status solusi ditemukan akan diaktifkan dan proses pencarian dihentikan.

4 Uji Coba Program

Table 1: Tabel Hasil Uji Coba Import .txt

Deskripsi Kasus	**Masukan (Input)**	**Keluaran (Output)**
Valid 4x4, Warna = N	 valid_4x4.txt	
Valid NxN, Warna < N	 warna_kurang.txt	
Valid NxN, Warna > N	 warna_lebih.txt	
Tidak NxN	 not_nxn.txt	

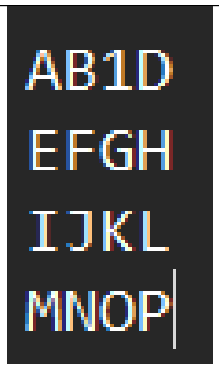
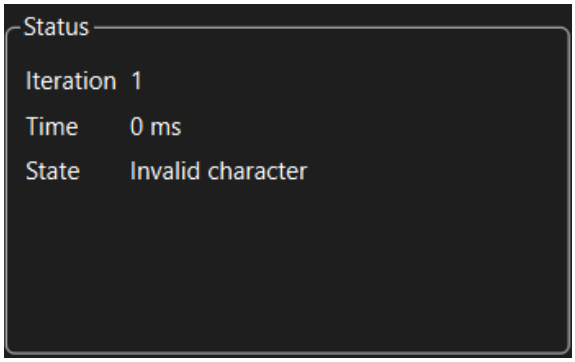
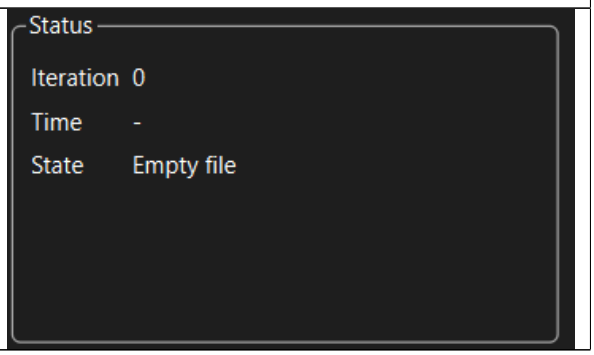
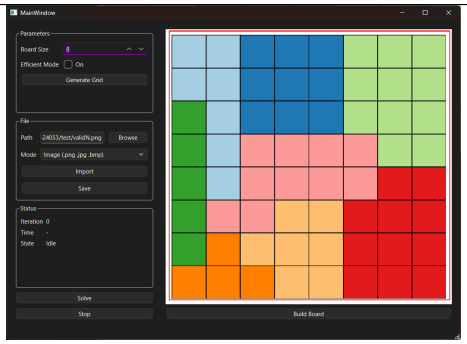
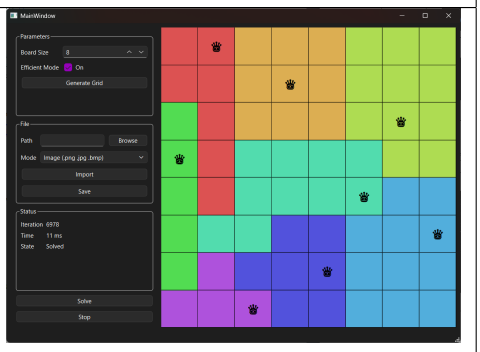
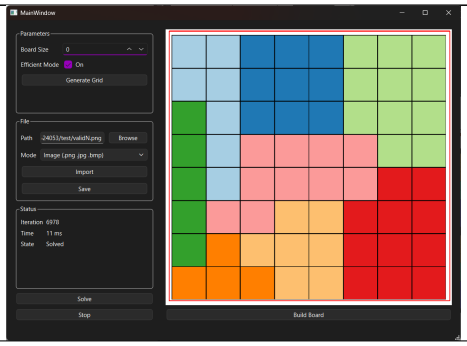
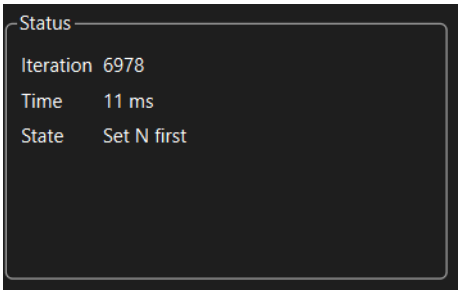
Deskripsi Kasus	Masukan (Input)	Keluaran (Output)
Karakter Invalid	 invalid_char.txt	
File Kosong	 kosong.txt	

Table 2: Tabel Hasil Uji Coba Import Image

Deskripsi Kasus	Masukan (Input)	Keluaran (Output)
Valid Image, Board Size = N, file validN.png semua		
Valid Image, Board Size = 0		

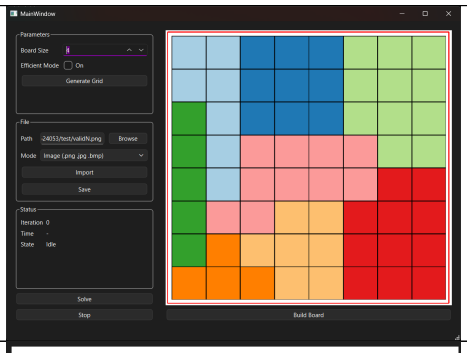
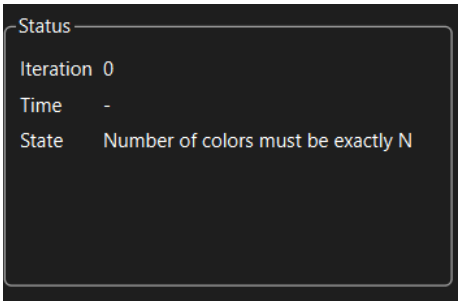
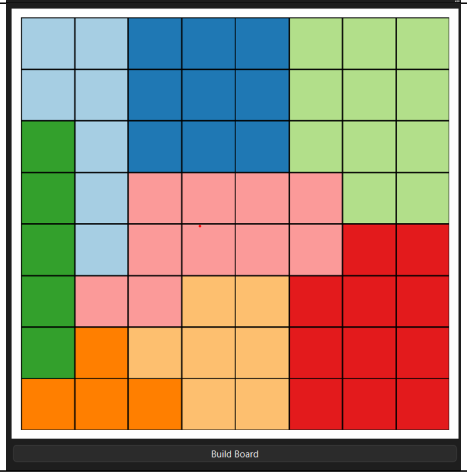
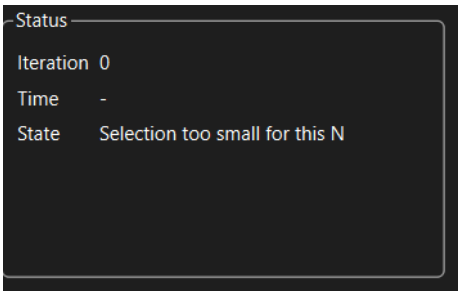
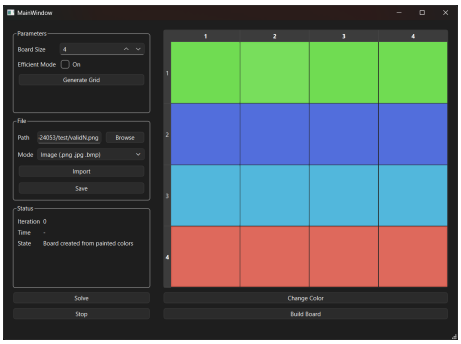
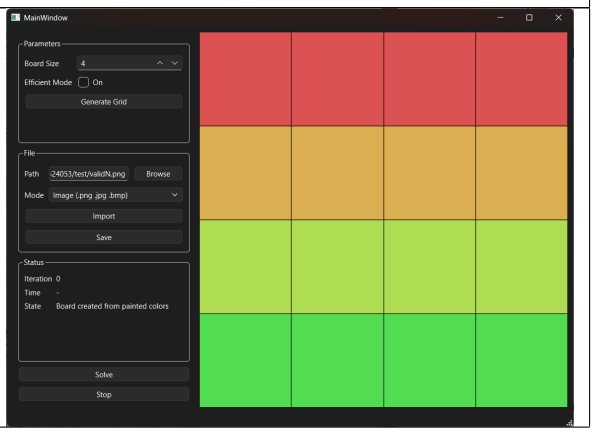
Deskripsi Kasus	Masukan (Input)	Keluaran (Output)
Valid Image, Board Size ≠ N		
Selection terlalu kecil		

Table 3: Tabel Hasil Uji Coba Input Manual

Deskripsi Kasus	Masukan (Input)	Keluaran (Output)
Semua sel diwarnai, input manual sesuai foto semua		

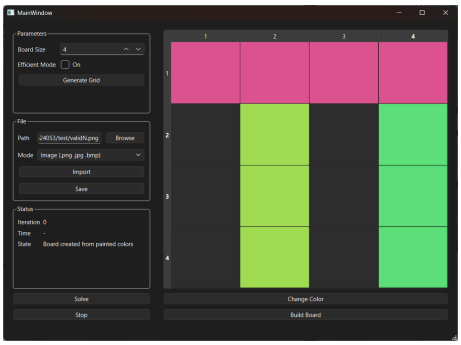
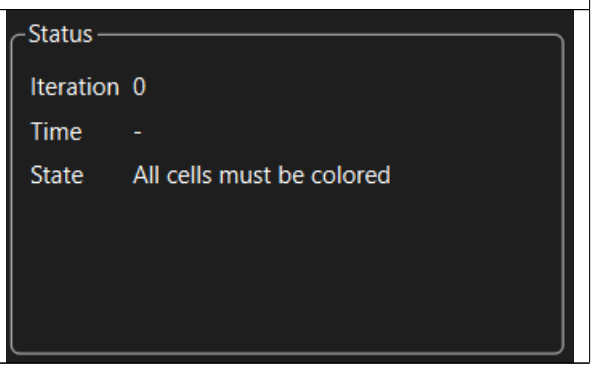
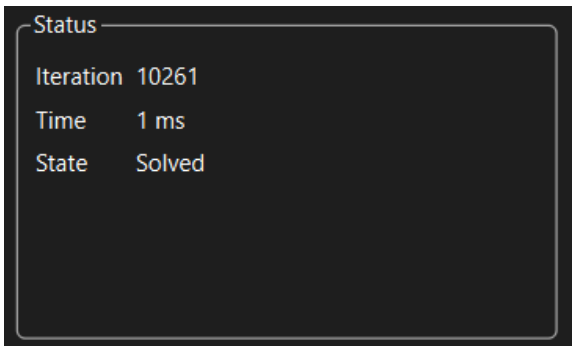
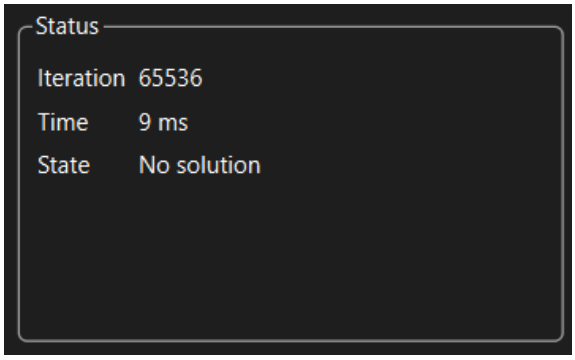
Deskripsi Kasus	Masukan (Input)	Keluaran (Output)
Ada sel kosong		

Table 4: Tabel Hasil Uji Coba Solver

Deskripsi Kasus	Masukan (Input)	Keluaran (Output)
Puzzle Ada Solusi	 valid_4x4.txt	
Puzzle Tidak Ada Solusi	 no_solution.txt	

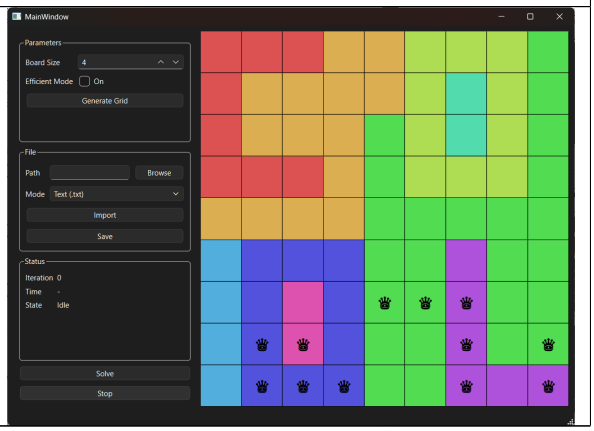
Deskripsi Kasus	Masukan (Input)	Keluaran (Output)
Stop Solver	 8x8.txt	

Table 5: Tabel Hasil Uji Saver .txt

Deskripsi Kasus	Masukan (Input)	Keluaran (Output)
Puzzle Ada Solusi	 valid_4x4.txt	 saver_txt_ada_o.txt

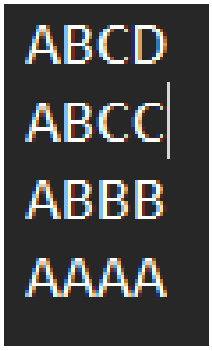
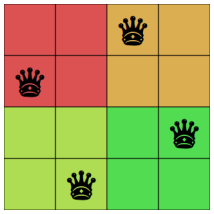
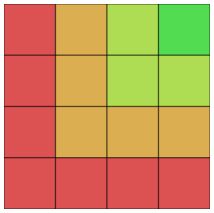
Deskripsi Kasus	Masukan (Input)	Keluaran (Output)
Puzzle Tidak Ada Solusi	 no_solution.txt	 saver_txt_no_o.txt

Table 6: Tabel Hasil Uji Saver Image

Deskripsi Kasus	Masukan (Input)	Keluaran (Output)
Puzzle Ada Solusi	 valid_4x4.txt	 saver_img_ada_o.png
Puzzle Tidak Ada Solusi	 no_solution.txt	 saver_img_no_o.png

A Lampiran

A.1 Pranala Ke Repositori

https://github.com/mhps-null/Tucil1_13524053

A.2 Pernyataan

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

Muhammad Haris Putra Sulastianto

A.3 Tabel Spesifikasi Tugas Kecil

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	✓	

A.4 Lampiran Kode Program dalam C++

A.4.1 main.cpp

```cpp
#include <QApplication>
#include "mainwindow.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    MainWindow w;
    w.show();

    return app.exec();
}
```

A.4.2 Board.cpp

```cpp
#include "Board.h"

Board::Board(int n, const vector<vector<int>> &color)
    : n(n), grid(n, vector<int>(n, 0)), color(color) {}

void Board::placeQueen(int row, int col)
{
    grid[row][col] = 1;
}

void Board::removeQueen(int row, int col)
{
    grid[row][col] = 0;
}

bool Board::isValidWholeBoard() const
{
    // queen
    int qCount = 0;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (grid[i][j] == 1)
            {
                qCount++;
            }
        }
    }

    if (qCount != n)
    {
        return false;
    }

    // row
    for (int i = 0; i < n; i++)
    {
        int count = 0;
```

```
        for (int j = 0; j < n; j++)
        {
            if (grid[i][j] == 1)
            {
                count++;
                if (count > 1)
                {
                    return false;
                }
            }
        }

        if (count != 1)
        {
            return false;
        }
    }

    // column
    for (int j = 0; j < n; j++)
    {
        int count = 0;
        for (int i = 0; i < n; i++)
        {
            if (grid[i][j] == 1)
            {
                count++;
                if (count > 1)
                {
                    return false;
                }
            }
        }

        if (count != 1)
        {
            return false;
        }
    }

    // adjacent diagonally
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (grid[i][j] == 1)
            {
                if (i < n - 1 && j < n - 1 && grid[i + 1][j + 1] ==
1)
                {
                    return false;
                }
                if (i < n - 1 && j > 0 && grid[i + 1][j - 1] == 1)
                {
                    return false;
                }
```

```
            }
        }
    }

    // color
    int maxColorId = countColorId();
    vector<int> countColor(maxColorId, 0);

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (grid[i][j] == 1)
            {
                int c = color[i][j];
                countColor[c]++;

                if (countColor[c] != 1)
                {
                    return false;
                }
            }
        }
    }

    for (int c = 0; c < maxColorId; c++)
    {
        if (countColor[c] != 1)
        {
            return false;
        }
    }

    return true;
}

int Board::countColorId() const
{
    set<int> colorId;

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            colorId.insert(color[i][j]);
        }
    }

    return colorId.size();
}

// getter grid
const vector<vector<int>> &Board::getGrid() const
{
    return grid;
}
```

```
151
152 // getter color
153 const vector<vector<int>> &Board::getColor() const
154 {
155     return color;
156 }
157
158 // getter size
159 int Board::getSize() const
160 {
161     return n;
162 }
```

A.4.3 Solver.cpp

```
1  #include "Solver.h"
2
3  Solver::Solver(const Board &board, bool efficientMode)
4      : board(board), efficientMode(efficientMode) {}
5
6  void Solver::solve()
7  {
8      QElapsedTimer timer;
9      timer.start();
10
11     frameTimer.start();
12
13     int n = board.getSize();
14
15     if (!efficientMode)
16     {
17         int totalSize = n * n;
18         while (true)
19         {
20             if (stopRequested)
21                 break;
22
23             recordIteration();
24
25             if (board.isValidWholeBoard())
26             {
27                 solutionFound = true;
28                 break;
29             }
30
31             bool keepGoing = true;
32
33             for (int k = totalSize - 1; k >= 0 && keepGoing; k--)
34             {
35                 int row = k / n;
36                 int col = k % n;
37
38                 if (board.getGrid()[row][col] == 0)
39                 {
```

```
                    board.placeQueen(row, col);
                    keepGoing = false;
                }
                else
                {
                    board.removeQueen(row, col);
                }
            }

            if (keepGoing)
            {
                break;
            }
        }
    }
    else
    {
        vector<int> permutation(n);
        for (int i = 0; i < n; i++)
        {
            permutation[i] = i;
        }

        do
        {
            if (stopRequested)
                break;

            recordIteration();

            for (int i = 0; i < n; i++)
            {
                for (int j = 0; j < n; j++)
                {
                    board.removeQueen(i, j);
                }
            }

            for (int row = 0; row < n; row++)
            {
                board.placeQueen(row, permutation[row]);
            }

            if (board.isValidWholeBoard())
            {
                solutionFound = true;
                break;
            }
        } while (next_permutation(permutation.begin(), permutation.
    end()));
    }
    qint64 time = timer.elapsed();

    if (!solutionFound)
    {
        for (int i = 0; i < n; i++)
```

```
95          {
96              for (int j = 0; j < n; j++)
97              {
98                  board.removeQueen(i, j);
99              }
100         }
101     }
102
103     emit finished(board, iterationCount, solutionFound, time);
104 }
105
106 void Solver::recordIteration()
107 {
108     iterationCount++;
109
110     if (frameTimer.elapsed() >= 30)
111     {
112         emit boardUpdated(board, iterationCount);
113         frameTimer.restart();
114     }
115
116     emit progress(iterationCount);
117 };
118
119 const Board &Solver::getBoard() const
120 {
121     return board;
122 };
123
124 long long Solver::getIterationCount() const
125 {
126     return iterationCount;
127 }
128
129 bool Solver::getSolFound() const
130 {
131     return solutionFound;
132 }
```