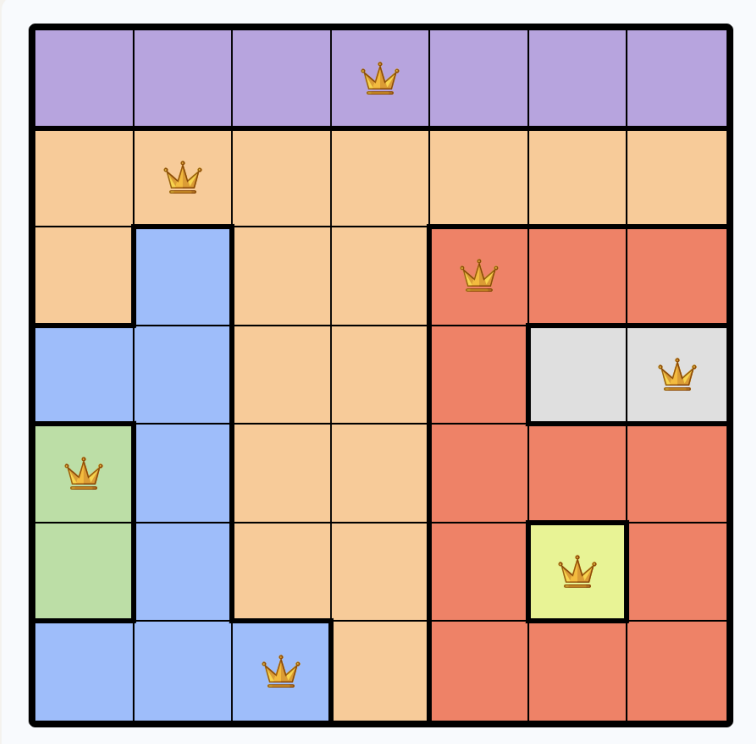
Laporan Tugas Kecil 1

IF2211 Strategi Algoritma

Penyelesaian Permainan **Queens** Linkedin Dengan

Semester II Tahun 2025/2026



Disusun oleh:

Philipp Hamara - 13524101

Laboratorium Ilmu dan Rekayasa Komputasi

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

Daftar Isi

1 Pendahuluan

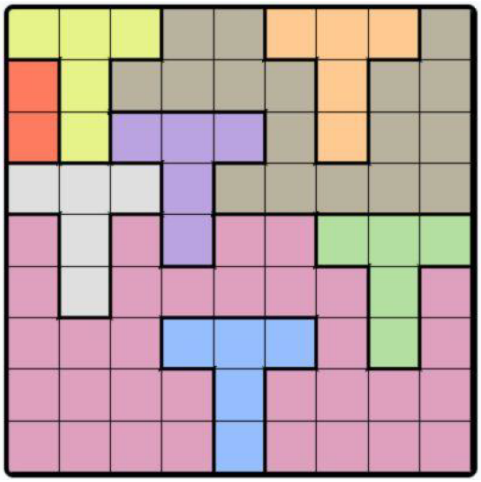
1.1 Deskripsi Tugas

Queens adalah gim logika yang tersedia pada situs jejaring profesional LinkedIn. Tujuan dari gim ini adalah menempatkan *queen* pada sebuah papan persegi berwarna sehingga terdapat tepat satu *queen* pada setiap baris, setiap kolom, dan setiap daerah warna. Selain itu, satu *queen* tidak dapat ditempatkan bersebelahan dengan *queen* lainnya, termasuk secara diagonal.

Tugas pada program ini adalah membuat algoritma yang dapat menemukan satu solusi penempatan *queen* pada papan berwarna yang diberikan, atau menampilkan bahwa tidak ada solusi yang valid. Program melakukan pencarian solusi menggunakan algoritma *brute force*.

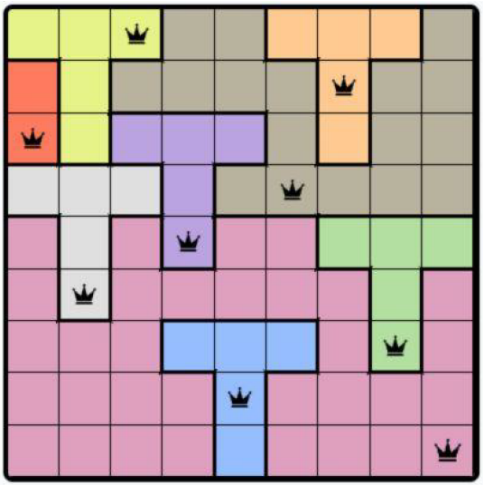
1.2 Ilustrasi Kasus

Sebagai ilustrasi, diberikan sebuah papan berwarna yang pada awalnya kosong.



Gambar 1.2.1: Papan awal dalam keadaan kosong.

Tujuannya adalah menemukan konfigurasi penempatan *queen* yang memenuhi seluruh aturan yang telah ditentukan, sehingga setiap baris, kolom, dan daerah warna masing-masing ditempati tepat satu *queen* tanpa ada yang saling bersebelahan.



Gambar 1.2.2: Salah satu solusi valid penempatan *queen*.

2 Implementasi Algoritma Brute Force

Dalam permainan "Queens", algoritma *brute force* diterapkan untuk mencari solusi dengan cara mengiterasi seluruh kemungkinan kombinasi posisi ratu pada papan berukuran $N \times N$. Pendekatan ini dilakukan dengan mencoba menempatkan satu ratu pada setiap baris secara rekursif, lalu melakukan validasi pada papan setelah seluruh ratu ditempatkan. Syarat-syarat yang harus dipenuhi adalah: tepat satu ratu pada setiap kolom, tidak boleh ada dua ratu yang saling bersinggungan secara ortogonal maupun diagonal (dalam radius satu petak), serta setiap ratu harus menempati wilayah warna yang berbeda.

2.1 Langkah-Langkah Algoritma

Berikut adalah langkah-langkah detail dari algoritma *brute force* yang diimplementasikan pada fungsi `bruteForce` dan `isValidBoard`:

1. Program menerima masukan berupa *grid* berukuran $N \times N$ yang dibaca dari file teks. Setiap karakter pada *grid* merepresentasikan suatu wilayah warna. Selain itu, diinisialisasi sebuah vektor `queenCol` berukuran N dengan seluruh elemen bernilai -1, yang merepresentasikan kolom tempat ratu ditempatkan pada setiap baris.

2. Fungsi `bruteForce` bekerja secara rekursif. Pada setiap pemanggilan, fungsi ini menerima parameter baris saat ini (`row`). Untuk baris tersebut, fungsi mencoba menempatkan ratu pada setiap kolom dari 0 hingga $N - 1$ dengan cara mengisi `queenCol[row] = col`, kemudian memanggil dirinya sendiri untuk baris berikutnya (`row + 1`).

3. Proses rekursif ini berlanjut hingga seluruh N baris telah terisi (`row == size`). Pada titik ini, tercapai sebuah konfigurasi lengkap di mana setiap baris memiliki tepat satu ratu. Konfigurasi ini kemudian divalidasi menggunakan fungsi `isValidBoard`.

4. Fungsi `isValidBoard` memeriksa tiga syarat secara berurutan untuk setiap ratu pada baris r dengan kolom c:

   - **Syarat kolom unik:** Untuk setiap ratu, diperiksa apakah ada ratu lain pada baris berbeda yang menempati kolom yang sama. Jika ditemukan `queenCol[r2] == c` untuk suatu $r2 \neq r$, maka konfigurasi tidak valid.
   - **Syarat *non-adjacency*:** Untuk setiap ratu pada posisi (r, c), diperiksa delapan tetangga di sekelilingnya (3×3 minus posisi dirinya sendiri). Jika terdapat ratu lain pada posisi tetangga tersebut, yakni `queenCol[nr] == nc` untuk suatu tetangga (nr, nc), maka konfigurasi tidak valid.
   - **Syarat wilayah unik:** Karakter wilayah warna pada posisi (r, c) diambil dari `grid[r][c]`. Jika karakter tersebut sudah pernah digunakan oleh ratu sebelumnya (dicek melalui `QSet<QChar>`), maka konfigurasi tidak valid. Jika belum, karakter tersebut ditambahkan ke himpunan wilayah yang telah digunakan.

5. Jika `isValidBoard` mengembalikan `true`, maka solusi ditemukan dan fungsi `bruteForce` langsung mengembalikan `true` secara berantai ke seluruh pemanggilan rekursif di atasnya. Jika tidak valid, fungsi mengembalikan ratu pada baris tersebut ke -1 (*reset*) dan mencoba kolom berikutnya.

6. Jika seluruh kolom pada suatu baris telah dicoba tanpa menghasilkan solusi, fungsi mengembalikan `false`, yang menyebabkan baris sebelumnya mencoba kolom berikutnya. Proses ini berlanjut hingga solusi ditemukan atau seluruh kemungkinan telah habis.

2.2 Analisis Kompleksitas

Algoritma *brute force* ini membangkitkan seluruh kemungkinan penempatan melalui rekursi, di mana untuk setiap baris terdapat N pilihan kolom. Oleh karena itu, jumlah total konfigurasi yang mungkin dibangkitkan adalah N^N. Pada setiap konfigurasi lengkap, fungsi `isValidBoard` melakukan pengecekan dengan kompleksitas $O(N^2)$ pada kasus terburuk. Dengan demikian, kompleksitas waktu keseluruhan adalah $O(N^N \cdot N^2)$, yang dapat disederhanakan menjadi $O(N^{N+2})$.

Sebagai perbandingan, program juga menyediakan mode *backtracking* yang melakukan *pruning* lebih awal. Pada mode ini, validasi dilakukan di setiap baris sebelum melanjutkan ke baris berikutnya melalui fungsi `isSafe`, sehingga cabang pencarian yang tidak mungkin menghasilkan solusi dapat dipangkas lebih dini. Hal ini secara signifikan mengurangi jumlah konfigurasi yang perlu diperiksa dibandingkan *brute force* murni.

2.3 Pseudocode

Secara formal, algoritma *brute force* yang diimplementasikan dapat direpresentasikan dalam *pseudocode* berikut.

Algorithm 1 Algoritma Brute Force Queens

```
procedure BruteForce(queenCol, row, size)
    if row = size then
        return IsValidBoard(queenCol, size)
    end if
    for col ← 0 to size − 1 do
        queenCol[row] ← col
        if BruteForce(queenCol, row + 1, size) then
            return true
        end if
        queenCol[row] ← −1
    end for
    return false
end procedure
```

Algorithm 2 Fungsi Validasi Papan

```
function IsValidBoard(queenCol, size)
    usedColors ← ∅
    for r ← 0 to size − 1 do
        c ← queenCol[r]
        if c < 0 then
            return false
        end if
        for r2 ← 0 to size − 1 do
            if r2 ≠ r and queenCol[r2] = c then
                return false                          ▷ Kolom tidak unik
            end if
        end for
        for di ← −1 to 1 do
            for dj ← −1 to 1 do
                if di = 0 and dj = 0 then continue
                end if
                nr ← r + di, nc ← c + dj
                if 0 ≤ nr < size and 0 ≤ nc < size then
                    if queenCol[nr] = nc then
                        return false                  ▷ Ratu bersentuhan
                    end if
                end if
            end for
        end for
        color ← grid[r][c]
        if color ∈ usedColors then
            return false                              ▷ Wilayah tidak unik
        end if
        usedColors ← usedColors ∪ {color}
    end for
    return true
end function
```

Algorithm 3 Algoritma Backtracking Queens

```
1: procedure BACKTRACK(queenCol, row, size, usedColors)
2:     if row = size then
3:         return true
4:     end if
5:     for col ← 0 to size − 1 do
6:         if ISSAFE(queenCol, row, col, size, usedColors) then
7:             queenCol[row] ← col
8:             usedColors ← usedColors ∪ {grid[row][col]}
9:             if BACKTRACK(queenCol, row + 1, size, usedColors) then
10:                return true
11:            end if
12:            queenCol[row] ← −1
13:            usedColors ← usedColors \ {grid[row][col]}
14:        end if
15:    end for
16:    return false
17: end procedure
```

Algorithm 4 Fungsi IsSafe (Validasi Per Langkah)

```
1: function ISSAFE(queenCol, row, col, size, usedColors)
2:     for r ← 0 to row − 1 do
3:         if queenCol[r] = col then
4:             return false                          ▷ Kolom sudah digunakan
5:         end if
6:     end for
7:     for dr ← −1 to 1 do
8:         for dc ← −1 to 1 do
9:             if dr = 0 and dc = 0 then continue
10:            end if
11:            nr ← row + dr, nc ← col + dc
12:            if 0 ≤ nr < size and 0 ≤ nc < size then
13:                for r ← 0 to row − 1 do
14:                    if r = nr and queenCol[r] = nc then
15:                        return false              ▷ Bersentuhan
16:                    end if
17:                end for
18:            end if
19:        end for
20:    end for
21:    if grid[row][col] ∈ usedColors then
22:        return false                              ▷ Warna sudah digunakan
23:    end if
24:    return true
25: end function
```

3 Source Code

Built with C++ and Qt. Dikarenakan ukuran *source code* yang cukup besar, pada bagian ini hanya ditampilkan struktur proyek serta potongan kelas QueensSolver yang mengandung algoritma brute-force yang berupa fokus dari tugas kecil ini. Implementasi lengkap dapat diakses melalui repositori GitHub yang tautannya tercantum pada lampiran laporan ini.

3.1 Struktur Proyek

```
Tucil1_13524101/
 bin/
 doc/
 src/
    CMakeLists.txt
    Main.qml
    main.cpp
    InputScreen.qml
    InputScreenForm.ui.qml
    SolverScreen.qml
    SolverScreenForm.ui.qml
    CustomButton.qml
    CustomButtonForm.ui.qml
    CustomToolBar.qml
    CustomToolBarForm.ui.qml
    queenssolver.h
    queenssolver.cpp
    filehandler.h
    filehandler.cpp
 test/
 README.md
```

3.2 Kelas QueensSolver

```cpp
#include "queenssolver.h"
#include <QThread>
#include <QtMath>

QueensSolver::QueensSolver(QObject *parent) : QObject(parent) {}

void QueensSolver::setColors(const QVector<QVector<QChar>> &colors) {
    m_colors = colors;
}

void QueensSolver::setColorMap(const QMap<QChar, QColor> &colorMap) {
    m_colorMap = colorMap;
}

void QueensSolver::setUpdateInterval(int n) {
    m_updateEveryN.store(qMax(1, n));
}

void QueensSolver::setDelayMs(int ms) {
    m_delayMs.store(qMax(0, ms));
}

int QueensSolver::iteration() const {
    return m_iteration;
}

void QueensSolver::setUseBacktracking(bool bt) {
    m_useBacktracking = bt;
}

void QueensSolver::solve() {
    int size = m_colors.size();
    QVector<int> queenCol(size, -1);
    m_iteration = 0;
```

```
35
36      bool result;
37      if (m_useBacktracking) {
38          QSet<QChar> usedColors;
39          result = backtrack(queenCol, 0, size, usedColors);
40      } else {
41          result = bruteForce(queenCol, 0, size);
42      }
43
44      emitCurrentState(queenCol, size);
45      emit finished(result, m_iteration);
46  }
47
48  void QueensSolver::tickIteration(const QVector<int> &queenCol, int size) {
49      m_iteration++;
50      int interval = m_updateEveryN.load();
51      if (interval > 0 && m_iteration % interval == 0) {
52          emitCurrentState(queenCol, size);
53          emit iterationUpdated(m_iteration);
54          int delay = m_delayMs.load();
55          if (delay > 0)
56              QThread::msleep(delay);
57      }
58  }
59
60  // BACKTRACKING
61
62  bool QueensSolver::isSafe(const QVector<int> &queenCol, int row, int col, int size, const
        QSet<QChar> &usedColors) {
63      // column check
64      for (int r = 0; r < row; r++) {
65          if (queenCol[r] == col) return false;
66      }
67
68      // 3x3 adjacency check
69      for (int dr = -1; dr <= 1; dr++) {
70          for (int dc = -1; dc <= 1; dc++) {
71              if (dr == 0 && dc == 0) continue;
72              int nr = row + dr, nc = col + dc;
73              if (nr >= 0 && nr < size && nc >= 0 && nc < size) {
74                  for (int r = 0; r < row; r++) {
75                      if (r == nr && queenCol[r] == nc) return false;
76                  }
77              }
78          }
79      }
80
81      // color check
82      if (usedColors.contains(m_colors[row][col])) return false;
83
84      return true;
85  }
86
87  bool QueensSolver::backtrack(QVector<int> &queenCol, int row, int size, QSet<QChar> &
        usedColors) {
88      tickIteration(queenCol, size);
89
90      if (row == size)
91          return true;
92
93      for (int col = 0; col < size; col++) {
94          if (isSafe(queenCol, row, col, size, usedColors)) {
95              queenCol[row] = col;
96              usedColors.insert(m_colors[row][col]);
97
98              if (backtrack(queenCol, row + 1, size, usedColors))
99                  return true;
100
101             queenCol[row] = -1;
102             usedColors.remove(m_colors[row][col]);
103         }
104     }
105     return false;
106 }
107
108 // PURE BRUTE FORCE
```

```
109
110 bool QueensSolver::isValidBoard(const QVector<int> &queenCol, int size) {
111     QSet<QChar> usedColors;
112
113     for (int r = 0; r < size; r++) {
114         int c = queenCol[r];
115         if (c < 0) return false;
116
117         // column check
118         for (int r2 = 0; r2 < size; r2++) {
119             if (r2 != r && queenCol[r2] == c) return false;
120         }
121
122         // 3x3 adjacency check
123         for (int di = -1; di <= 1; di++) {
124             for (int dj = -1; dj <= 1; dj++) {
125                 if (di == 0 && dj == 0) continue;
126                 int nr = r + di, nc = c + dj;
127                 if (nr >= 0 && nr < size && nc >= 0 && nc < size) {
128                     if (queenCol[nr] == nc) return false;
129                 }
130             }
131         }
132
133         // color uniqueness
134         QChar color = m_colors[r][c];
135         if (usedColors.contains(color)) return false;
136         usedColors.insert(color);
137     }
138     return true;
139 }
140
141 bool QueensSolver::bruteForce(QVector<int> &queenCol, int row, int size) {
142     tickIteration(queenCol, size);
143
144     if (row == size) {
145         return isValidBoard(queenCol, size);
146     }
147
148     for (int col = 0; col < size; col++) {
149         queenCol[row] = col;
150         if (bruteForce(queenCol, row + 1, size))
151             return true;
152         queenCol[row] = -1;
153     }
154     return false;
155 }
156
157 void QueensSolver::emitCurrentState(const QVector<int> &queenCol, int size) {
158     QVector<Cell> cells;
159     cells.reserve(size * size);
160
161     for (int r = 0; r < size; r++) {
162         for (int c = 0; c < size; c++) {
163             Cell cell;
164             cell.color = m_colors[r][c];
165             cell.hasQueen = (queenCol[r] == c);
166             cell.cellColor = m_colorMap.value(cell.color, QColor(0x88, 0x88, 0x88));
167             cells.append(cell);
168         }
169     }
170
171     emit boardUpdated(cells, size);
172 }
```

Listing 3.1: Kelas QueensSolver Yang mengandung Algoritma Brute Force Penyelesaian "Queens" Linkedin

4 Pengujian

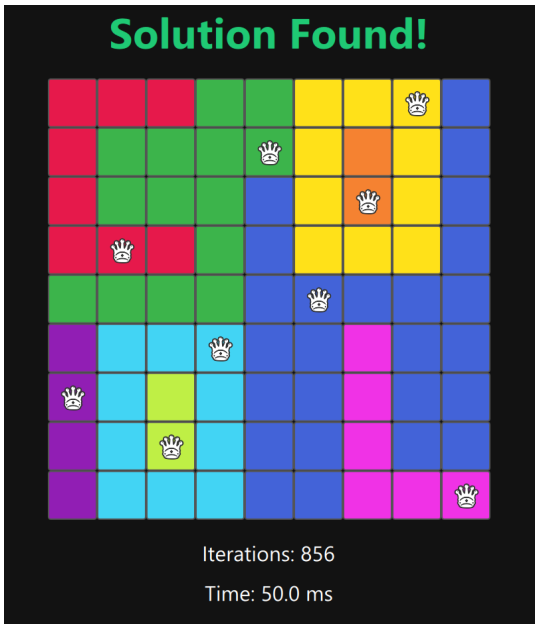
Untuk section ini, setiap tes dilakukan dengan optimized algoritma dengan backtracking dan ketentuan Live Update set to every 500 iterations dan delay to 30ms.

4.1 Test 1: Papan berukuran 9×9

Tabel 4.1: Input Test 1

AAABBCCCD ABBBBCECD ABBBDCECD AAABDCCCD BBBBDDDDD FGGGDDHDD FGIGDDHDD FGIGDDHDD FGGGDDHHH

Gambar 4.1.1: Output Test 1

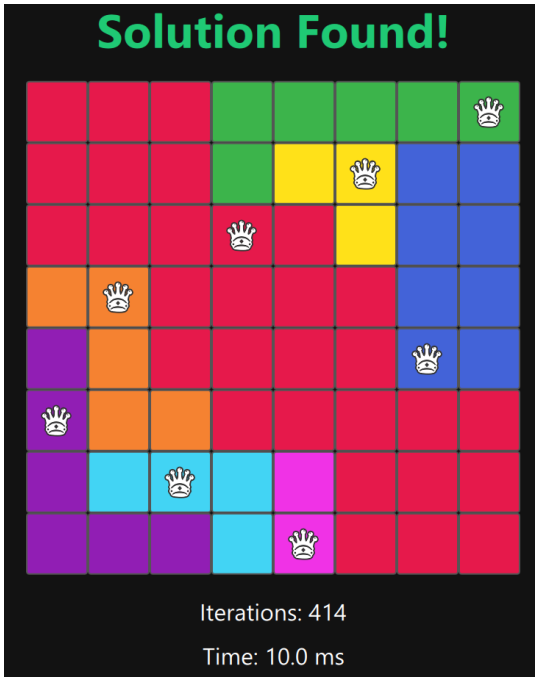


4.2 Test 2: Papan berukuran 8×8

Tabel 4.2: Input Test 2

AAABBBBB AAABCCDD AAAAACDD EEAAAADD FEAAAADD FEEAAAAA FGGGHAAA FFFGHAAA

Gambar 4.2.1: Output Test 2

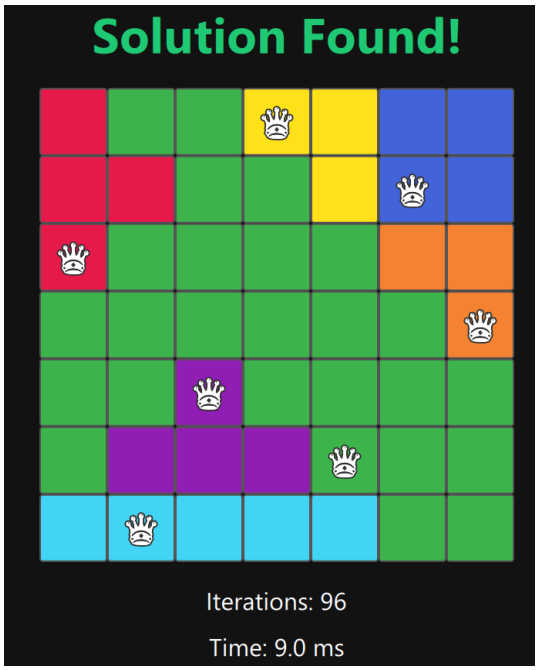


4.3 Test 3: Papan berukuran 7×7

Tabel 4.3: Input Test 3

ABBCCDD AABBCDD ABBBBEE BBBBBBE BBFBBBB BFFFBBB GGGGGBB

Gambar 4.3.1: Output Test 3

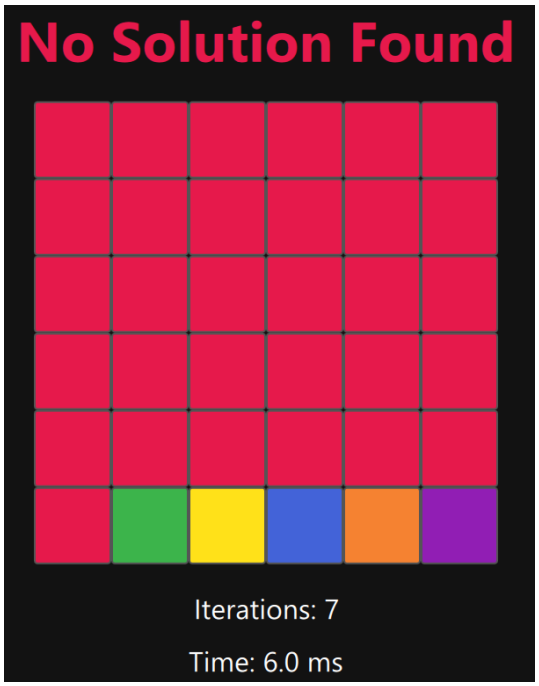


4.4 Test 4: Papan berukuran 6×6

Tabel 4.4: Input Test 4

AAAAAA AAAAAA AAAAAA AAAAAA AAAAAA ABCDEF

Gambar 4.4.1: Output Test 4

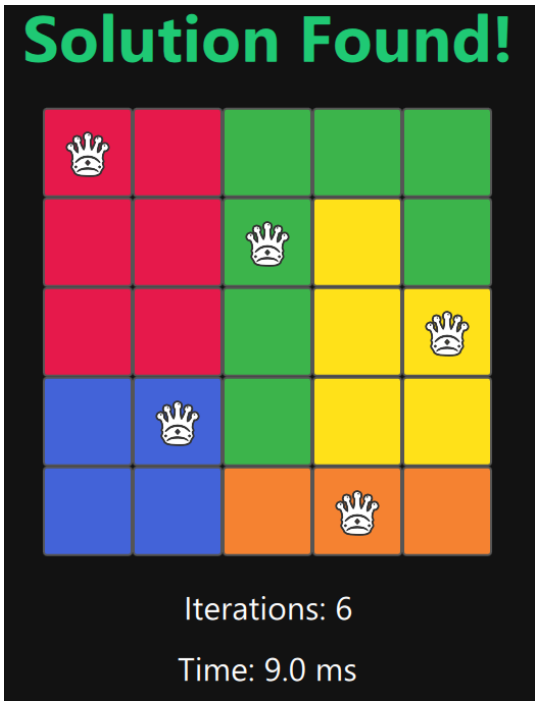


4.5 Test 5: Papan berukuran 5×5

Tabel 4.5: Input Test 5

AABBB AABCB AABCC DDBCC DDEEE

Gambar 4.5.1: Output Test 5



4.6 Test 6: Papan berukuran 4×4

Tabel 4.6: Input Test 6

AABB ABBB CCDB CCDD

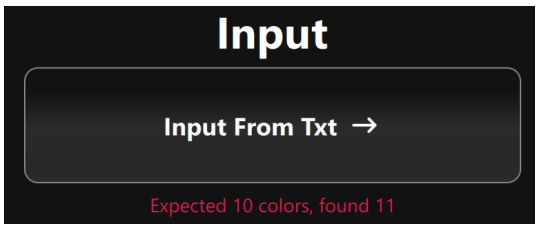
Gambar 4.6.1: Output Test 6

4.7 Test 7: Papan berukuran 10×10

Tabel 4.7: Input Test 7

```
AABBCCDDEE
AABBCCDDEE
FFBBCCDDGG
FFHHCCIIGG
FFHHJJIIGG
FFHHJJIIGG
FFHHJJIIGG
KKHHJJIIGG
KKHHJJIIGG
KKHHJJIIGG
```

Gambar 4.7.1: Output Test 7

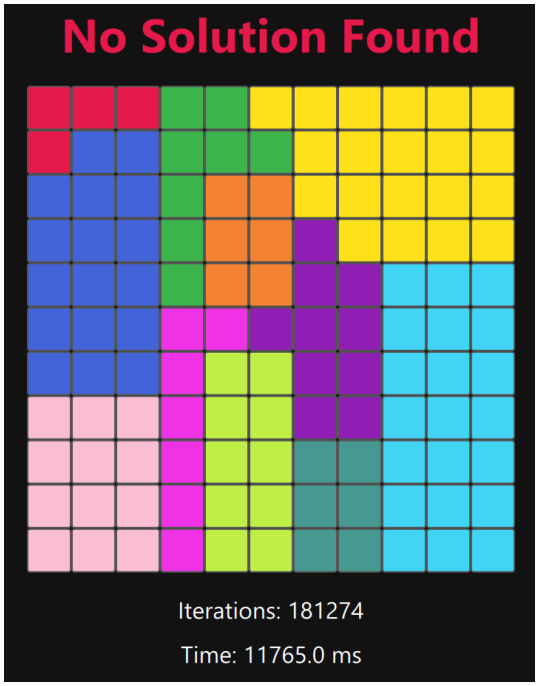


4.8 Test 8: Papan berukuran 11×11

Tabel 4.8: Input Test 8

```
AAABBCCCCCC
ADDBBBCCCCC
DDDBEECCCCC
DDDBEEFCCCC
DDDBEEFFGGG
DDDHHFFFGGG
DDDHIIFFGGG
JJJHIIFFGGG
JJJHIIKKGGG
JJJHIIKKGGG
JJJHIIKKGGG
```

Gambar 4.8.1: Output Test 8

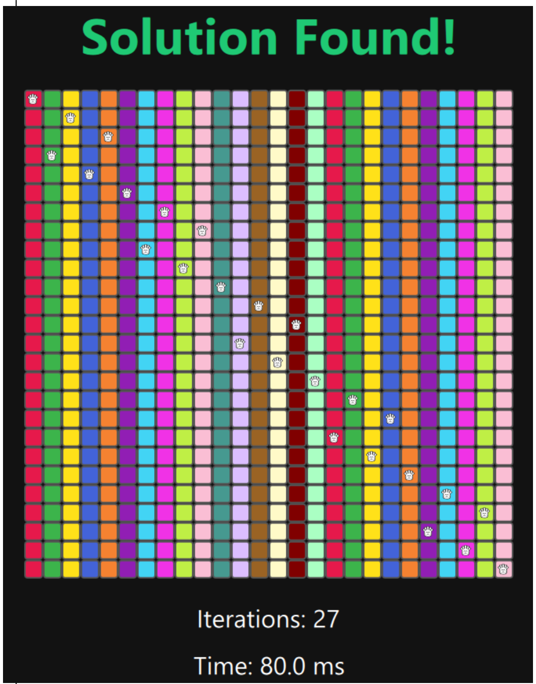


4.9 Test 9: Papan berukuran 26×26

Tabel 4.9: Input Test 9

ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Gambar 4.9.1: Output Test 9

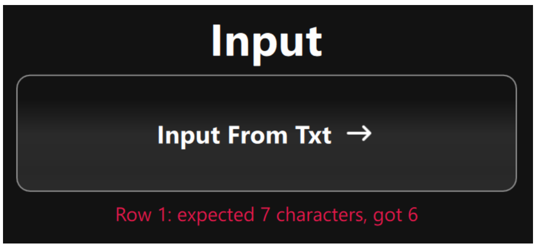


4.10 Test 10: Papan Tidak Persegi (6×7)

Tabel 4.10: Input Test 10

ABCDEF
ABCDEF
ABCDEF
ABCDEF
ABCDEF
ABCDEF
ABCDEF

Gambar 4.10.1: Output Test 10



5 Lampiran

https://github.com/philipphqiwu/Tucil1_13524101

Tabel 5.1: Evaluasi

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	✓	

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

Philipp Hamara [13524101]