IF2211 Strategi Algoritma: Tugas Kecil 1: Penyelesaian Permainan Queens

Program Studi Teknik Informatika
Institut Teknologi Bandung

Tahun Ajaran 2025/2026

Yavie Azka Putra Araly
13524077

Bagian 1

Deskripsi Algoritma Brute Force

Algoritma Brute Force adalah algoritma yang menyelesaikan permasalahan secara / *straightforward*, tanpa pendekatan heuristik. Pada tugas ini, problem *Linkedin N-Quenss* akan diselesaikan dengan algoritma tersebut. Berikut langkah-langkah penyelesaiannya:

1. **Struktur Data:** Papan direpresentasikan melalui `struct Board` yang menyimpan matriks dua dimensi `vector<vector<Tile>>`. Setiap elemen `Tile` menyimpan informasi ID wilayah warna (`region`) dan status penempatan Queen (`isQueenPlaced`).
2. **Brute-Force:** Melalui fungsi `solveNoHeuristics()`, program mencoba menempatkan tepat satu Queen pada setiap baris secara rekursif. Untuk baris ke-i, program akan menempatkan Queen di kolom ke-j (dari 0 hingga $N-1$), menandai `isQueenPlaced = true`, menambahkan iterasi, lalu memanggil fungsi rekursif untuk baris selanjutnya. Program akan terus meletakkan Queen hingga jumlah Queen sama dengan jumlah region.
3. **Pengujian:** Fungsi `checkConfig()` akan dipanggil setelah jumlah Queen yang berada pada papan sudah sesuai. Jika konfigurasi benar, maka program akan mengembalikan nilai `True` beserta papan yang telah diisi. Jika tidak, program akan mengembalikan nilai false dan tidak memindahkan queen ke kolom selanjutnya.
4. **Kriteria Validasi Konfigurasi:** Fungsi `checkConfig()` memeriksa empat aturan permainan secara sekuensial:
   - Memeriksa agar tidak ada lebih dari 1 Queen di setiap baris.
   - Memeriksa agar tidak ada lebih dari 1 Queen di setiap kolom.
   - Memeriksa agar tidak ada lebih dari 1 Queen pada `region` warna yang sama.
   - Memeriksa 8 sel tetangga (*adjacent*) di sekitar setiap Queen untuk memastikan tidak ada Queen yang saling bersentuhan (termasuk secara diagonal).

Pseudocode Algoritma

```
1:  function SolveNoHeuristics(board, currentRow, iterations)
2:      if currentRow == board.size then
3:          return CheckConfig(board)                          ▷ Test keseluruhan konfigurasi
4:      end if
5:      for i ← 0 to board.size − 1 do
6:          board.tile[currentRow][i].isQueenPlaced ← true
7:          iterations ← iterations + 1
8:          if SolveNoHeuristics(board, currentRow + 1, iterations) then
9:              return true                                    ▷ Solusi valid ditemukan
10:         end if
11:         board.tile[currentRow][i].isQueenPlaced ← false    ▷ Reset status
12:     end for
13:     return false
14: end function
```

Algorithm 1: Algoritma Brute Force Murni (Generate and Test)

```
1: function CheckConfig(board)
2:     if Terdapat lebih dari 1 Queen pada baris mana pun then return false
3:     end if
4:     if Terdapat lebih dari 1 Queen pada kolom mana pun then return false
5:     end if
6:     if Terdapat lebih dari 1 Queen pada region warna yang sama then return false
7:     end if
8:     if Terdapat Queen yang bersebelahan (horizontal, vertikal, diagonal) then return false
9:     end if
10:    return true
11: end function
```

Algorithm 2: Fungsi Validasi Konfigurasi Papan

Namun, algoritma diatas akan memiliki kompleksitas yang sangat tinggi:

1. **solveNoHeuristics():** Fungsi solveNoHeuristics() mencoba untuk meletakkan satu Queen pada setiap baris. Dengan papan berukuran $N \times N$, banyaknya konfigurasi Queen yang bisa diletakkan di setiap barisnya adalah $N \times N \times \cdots \times N$, sehingga kompleksitasnya menjadi $\mathcal{O}(N^N)$

2. **checkConfig():** Fungsi checkConfig() akan dipanggil setiap kali fungsi solveNoHeuristics() mencapai basecase, atau ketika seluruh Queens sudah ditempatkan pada papan. Fungsi ini akan melakukan pengecekan seluruh baris dan kolom berukuran N, sehingga kompleksitasnya menjadi $\mathcal{O}(N^2)$. Selain itu, fungsi ini juga akan melakukan pengecekan pada setiap region dengan kompleksitas $\mathcal{O}(N^4)$ dan pengecekan ketetanggan untuk setiap Queens dengan kompleksitas $\mathcal{O}(N^2)$. Sehingga secara keseluruhan, kompleksitasnya menjadi $\mathcal{O}(N^2) + \mathcal{O}(N^4) + \mathcal{O}(N^2) = \mathcal{O}(N^4)$.

Sehingga, secara keseluruhan kompleksitas algoritma brute force murni ini adalah $\mathcal{O}(N^N) \times \mathcal{O}(N^4) = \mathcal{O}(N^{N+4})$. Perhitungan untuk contoh kasus di spesifikasi tugas tidak akan bisa dikomputasi, karena ukuran papan 9×9 akan membutuhkan komputasi sebanyak $9^{13} = 2541865828329$ atau sekitar 2,5 triliun kali.

Oleh karena itu, penulis menambahkan sedikit optimasi untuk memungkinkan komputasi dengan ukuran papan yang lebih besar. Disini, penulis menambahkan fungsi `solveOptimized()`, dimana validasi dilakukan lebih awal pada setiap langkah penempatan Queen menggunakan pendekatan dengan pemangkasan (*pruning*) ruang pencarian.

1. **Pengecekan Validitas Awal:** Berbeda dengan `solveNoHeuristics` yang menunggu seluruh N Queen diletakkan, `solveOptimized` menguji validitas sel (r, c) *sebelum* memanggil fungsi rekursif untuk baris selanjutnya menggunakan fungsi bantuan `checkCurrentQueenPos()`. Fungsi ini akan memeriksa apakah sel (r, c) akan menghasilkan konfigurasi yang valid jika sebuah Queen diletakkan pada sel tersebut.

2. **Pemangkasan Jalur Buntu:** Jika peletakan Queen di (r, c) terbukti melanggar aturan (misal, satu kolom dengan Queen sebelumnya atau berada di *region* yang sama), algoritma tidak akan menelusuri cabang tersebut lebih dalam. Ini langsung memangkas ribuan kombinasi tak berguna di bawahnya.

3. **Kompleksitas Evaluasi Lebih Ringan:** Fungsi `checkCurrentQueenPos()` memakan biaya komputasi yang jauh lebih ringan karena hanya mengevaluasi bentrokan yang beririsan langsung dengan koordinat (r, c) yang sedang diuji, bukan menyisir ulang seluruh papan.

Pseudocode Algoritma Optimasi (Brute force dengan Pruning)

```
1: function SolveOptimized(board, currentRow, iterations)
2:     if iterations > 0 and iterations (mod 10) == 0 then
3:         LogIteration(board, iterations)
4:     end if
5:     if currentRow == board.size then
6:         return true                          ▷ Semua Queen berhasil ditempatkan dengan valid
7:     end if
8:     for i ← 0 to board.size − 1 do
9:         iterations ← iterations + 1
10:        if CheckCurrentQueenPos(board, currentRow, i) then
11:            board.tile[currentRow][i].isQueenPlaced ← true
12:            if SolveOptimized(board, currentRow + 1, iterations) then
13:                return true
14:            end if
15:            board.tile[currentRow][i].isQueenPlaced ← false          ▷ Cabut Queen
16:        end if
17:    end for
18:    return false
19: end function
```

Algorithm 3: Algoritma Backtracking Termodifikasi

Bagian 2

Source Program

Program ini ditulis menggunakan bahasa **C++** untuk komputasi utama algoritma brute force, dan **Python** untuk GUI.

Potongan kode di bawah menampilkan struktur data utama serta algoritma Brute Force (Murni dan Teroptimasi). Kode lengkap dan file GUI dapat diakses melalui repositori GitHub yang dilampirkan pada bagian selanjutnya.

```
struct Tile{
    char region;
    bool isQueenPlaced;
};

struct Board{
    vector<vector<Tile>> tile;
};

bool solveNoHeuristics(Board& board, int currentRow, long long& iterations){
    if(currentRow == board.tile.size()){
        return checkConfig(board);
    }

    for(int i = 0; i < board.tile.size(); i++){
        board.tile[currentRow][i].isQueenPlaced = true;
```

```
        iterations++;

        if(iterations % 10 == 0){
            logIteration(board, iterations);
        }

        if(solveNoHeuristics(board, currentRow + 1, iterations))
            return true;

        board.tile[currentRow][i].isQueenPlaced = false;
    }
    return false;
}

bool solveOptimized(Board& board, int currentRow, long long& iterations){
    if(iterations > 0 && iterations % 10 == 0){
        logIteration(board, iterations);
    }

    if(currentRow == board.tile.size()){
        return true;
    }

    for(int i = 0; i < board.tile.size(); i++){
        iterations++;

        if(checkCurrentQueenPos(board, currentRow, i)){
            board.tile[currentRow][i].isQueenPlaced = true;

            if(solveOptimized(board, currentRow + 1, iterations)){
                return true;
            }

            board.tile[currentRow][i].isQueenPlaced = false;
        }
    }
    return false;
}
```

Bagian 3

Tangkapan Layar Input dan Output

Berikut adalah dokumentasi hasil eksekusi program. (*Catatan: Untuk kasus uji papan dengan ukuran diatas* 8×8*, optimasi akan dinyalakan agar solusi dapat ditemukan*).

Kasus Uji 1: Contoh kasus pada spesifikasi tugas

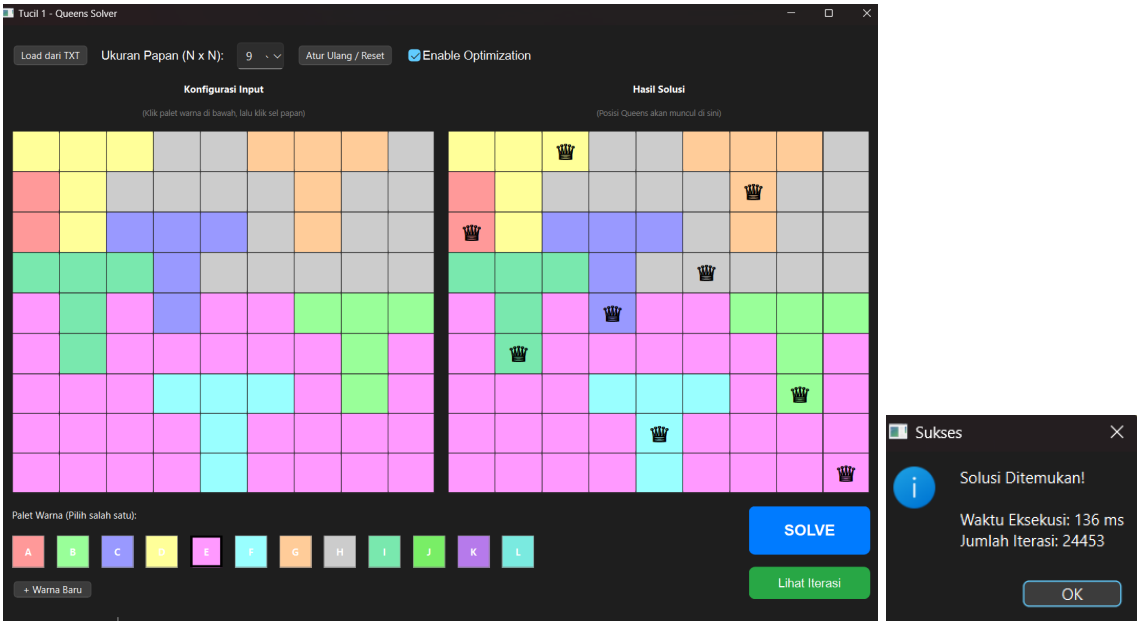


Figure 1: Visualisasi pencarian dan solusi akhir untuk Test Case 1.

Kasus Uji 2: Tanpa solusi

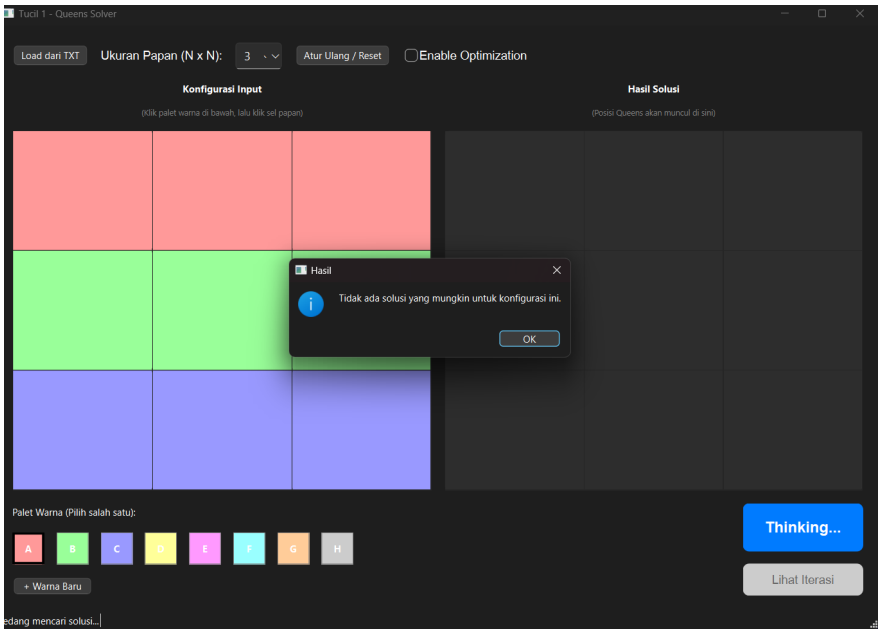


Figure 2: Visualisasi pencarian dan solusi akhir untuk Test Case 2.

Kasus Uji 3: Papan berukuran besar

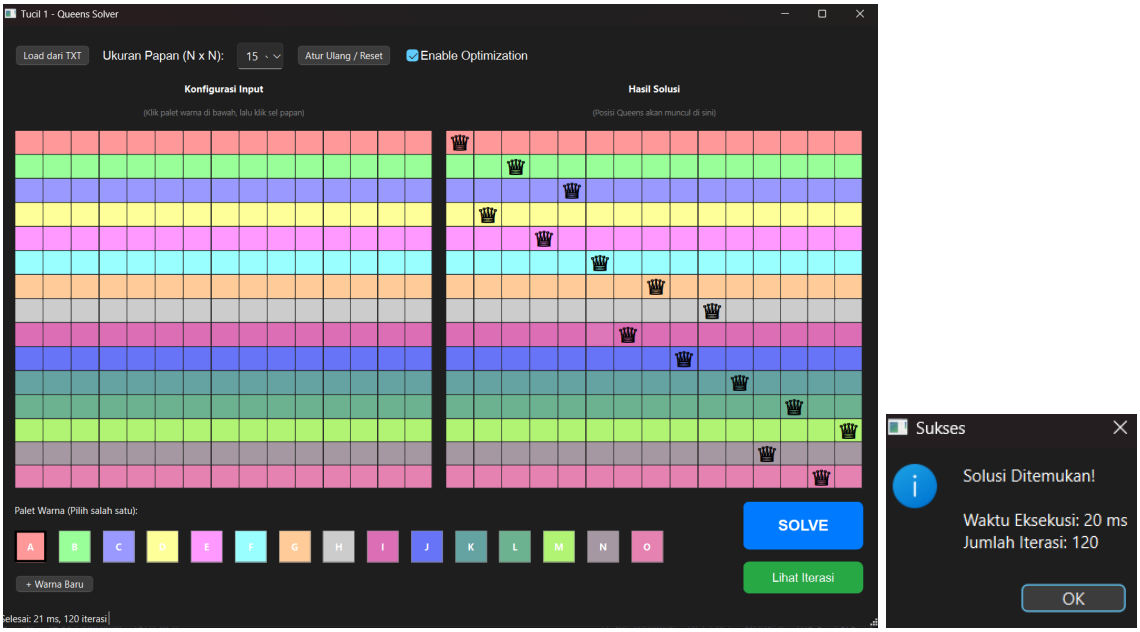


Figure 3: Visualisasi pencarian dan solusi akhir untuk Test Case 3.

Kasus Uji 4: Konfigurasi warna tidak valid

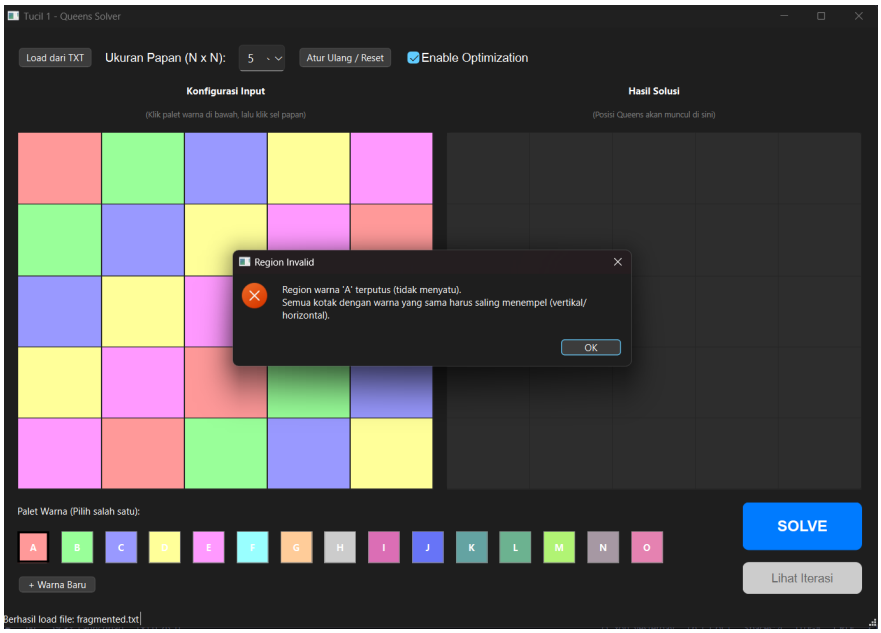
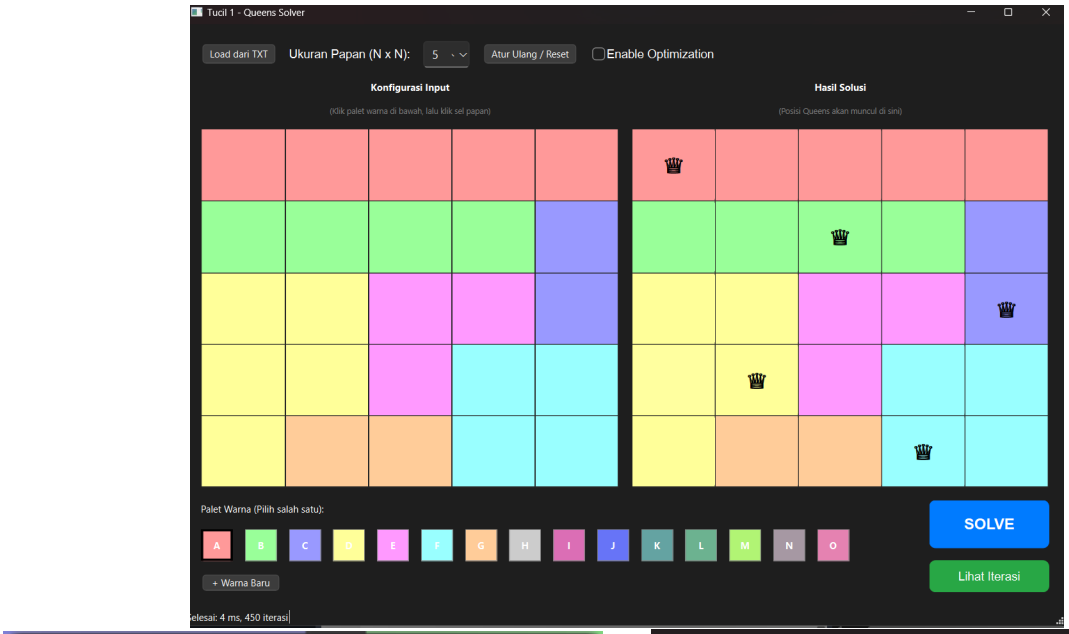
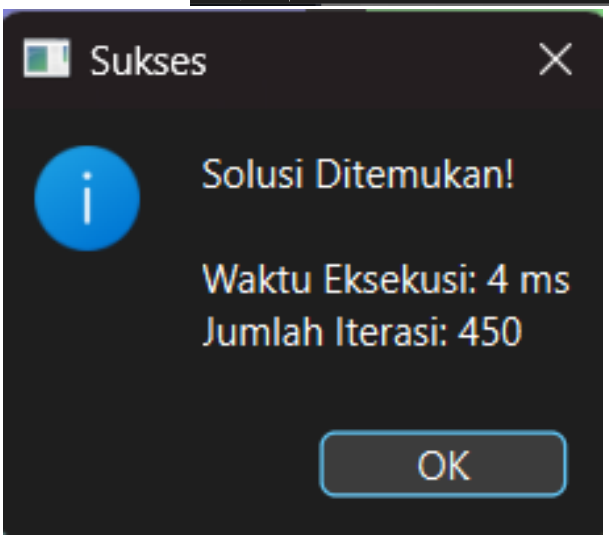


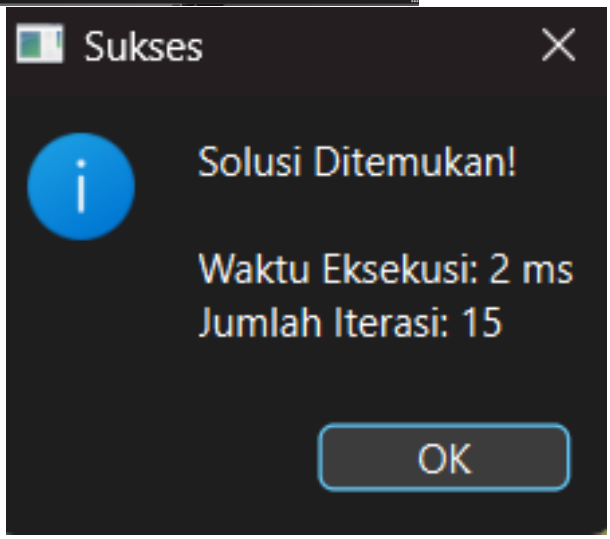
Figure 4: Visualisasi pencarian dan solusi akhir untuk Test Case 4.

Kasus Uji 5: Perbedaan jumlah iterasi dan waktu pencarian solusi dengan dan tanpa optimasi





(a) Tanpa optimasi



(b) Dengan optimasi

Figure 5: Perbandingan hasil pencarian solusi pada Kasus Uji 5.

Bagian 4

Pranala Repository

Seluruh kode program, file executable, dan data uji dapat diakses pada repositori publik berikut:

`https://github.com/YavieAzka/Tucil1_13524077`

Bagian 5

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

Yavie Azka Putra Araly

Lampiran Checklist 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	✓	