IF2211 - K01

TUGAS KECIL 1 STRATEGI ALGORITMA

Penyelesaian IQ Puzzler Pro dengan Algoritma Brute Force

Disusun oleh:

Rafen Max Alessandro 13523031

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2024

Daftar Pustaka

Daftar Gambar

Daftar Tabel

Bab I

Deskripsi Masalah

Algoritma *Brute Force* merupakan algoritma berbasis pendekatan yang lurus atau lempang (*straightforward*) dalam memecahkan suatu persoalan. Algoritma ini memecahkan persoalan dengan sangat sederhana, langsung, dan menggunakan cara yang paling kentara. Algoritma *brute force* digunakan sebagai dasar dalam penyelesaian tugas kecil 1 IF2211 Strategi Algoritma, sebagai sarana dalam menyelesaikan permainan pengasah otak dan ketangkasan IQ Puzzle Pro.

Gambar 1. IQ Puzzler Pro

Permainan IQ Puzzler Pro dimulai dengan papan kosong yang diisi oleh pemain menggunakan potongan puzzle yang tersedia untuk mengisi penuh papan tanpa adanya potongan puzzle yang tersisa. Dengan tujuan permainan yang definit tanpa adanya variabel acak di luar permainan, permainan IQ Puzzler Pro dapat diselesaikan menggunakan algoritma *brute force*.

Tahapan penyelesaian permainan IQ Puzzler Pro menggunakan algoritma *brute force* adalah sebagai berikut:

1. Untuk setiap potongan puzzle yang tersedia, iterasikan setiap kemungkinan posisi yang dapat diletakkan pada papan. Iterasi dilakukan dengan melakukan rotasi 90 derajat kepada potongan puzzle sebanyak 4 kali, refleksi potongan puzzle secara horizontal, lalu

rotasi 90 derajat lagi sebanyak 4 kali untuk menghasilkan 8 posisi puzzle yang dapat diletakkan pada papan.

2. Pilih salah satu potongan puzzle dan salah satu kemungkinan posisi yang dimilikinya. Letakkan potongan puzzle pada slot paling kiri atas pada papan. Langkah selanjutnya ditentukan berdasarkan apakah potongan puzzle dapat diletakkan atau tidak.

3. Jika potongan puzzle dapat diletakkan, lanjutkan proses peletakkan pada potongan puzzle selanjutnya hingga seluruh potongan puzzle berhasil diletakkan.

4. Jika potongan puzzle tidak dapat diletakkan, coba kemungkinan posisi lain yang dimiliki hingga ditemukan potongan puzzle yang dapat diletakkan atau hingga semua kemungkinan posisi telah dicoba.

5. Jika hingga kemungkinan posisi terakhir potongan puzzle tidak dapat diletakkan, pindah posisi peletakan potongan puzzle pada kolom slot selanjutnya, lalu ulangi langkah 2. Jika seluruh kolom telah dicoba, pindah posisi peletakan pada baris slot selanjutnya dan kolom paling kiri sebelum mengulangi langkah 2.

6. Jika suatu saat terdapat potongan puzzle yang sudah dicoba seluruh kemungkinan posisi dan seluruh slot papan, lakukan *backtrack* dengan mencoba posisi atau letak baru pada potongan puzzle yang sebelumnya telah diletakkan sesuai prosedur algoritma, lalu ulangi percobaan peletakan puzzle terhadap potongan puzzle yang sebelumnya tidak dapat diletakkan.

7. Kondisi berhasil didapat setelah semua potongan puzzle digunakan untuk menutupi seluruh slot papan, sedangkan kondisi gagal ditentukan oleh salah satu kondisi berikut:

    - Semua kemungkinan kombinasi peletakan puzzle telah dicoba dan papan tidak dapat dipenuhi sepenuhnya;

    - Seluruh potongan puzzle telah diletakkan, tetapi terdapat slot papan yang tidak terisi.

Untuk kondisi kedua dimana kegagalan terjadi, yaitu terdapat slot papan yang tidak terisi meskipun seluruh potongan puzzle telah diletakkan, maka permainan langsung dinyatakan gagal meskipun algoritma *brute force* belum selesai. Untuk kasus tersebut, kegagalan baru dinyatakan setelah semua kemungkinan kombinasi peletakan puzzle dicoba, maka program akan melakukan banyak sekali iterasi yang tidak menghasilkan solusi menggunakan tingkatan sumber daya komputasi yang tinggi. Oleh karena itu, untuk meningkatkan efisiensi dan kinerja program, jika seluruh potongan puzzle telah berhasil diletakkan mematuhi aturan permainan, tetapi tidak menutupi papan sepenuhnya, algoritma langsung dihentikan dan dinyatakan gagal. Namun, perlu disadari bahwa sudut pandang yang digunakan untuk mengambil keputusan ini memiliki campur

tangan heuristik. Pendekatan yang absolut berbasis algoritma *brute force* akan terus melakukan iterasi hingga kondisi gagal hanya dapat dinyatakan setelah seluruh kombinasi peletakan puzzle telah dicoba dan tidak memenuhi syarat keberhasilan.

Sesuai dengan spesifikasi tugas kecil, setelah algoritma berhasil menemukan satu solusi yang berhasil menyelesaikan permainan IQ Puzzler Pro, permainan dikatakan selesai dan program akan menunjukkan hasil akhir papan dengan huruf berwarna sebagai deskripsi dari posisi peletakan potongan puzzle pada papan. Program juga menampilkan waktu yang diperlukan untuk menemukan solusi serta jumlah iterasi yang dilakukan, serta memberikan opsi bagi pengguna untuk menyimpan solusi permainan ke dalam file .txt.

Sesuai dengan eksistensi papan non-kotak pada permainan IQ Puzzler Pro di dunia nyata, program juga dapat menerima konfigurasi CUSTOM yang menerima masukan bentuk papan untuk diisi oleh potongan puzzle. Lalu, untuk membentuk interaksi yang menarik dan terhubung dengan pengguna, program dijalankan menggunakan *graphical user interface* (GUI) berbasis JavaFX yang meningkatkan kemudahan dan operasi program bagi pengguna.

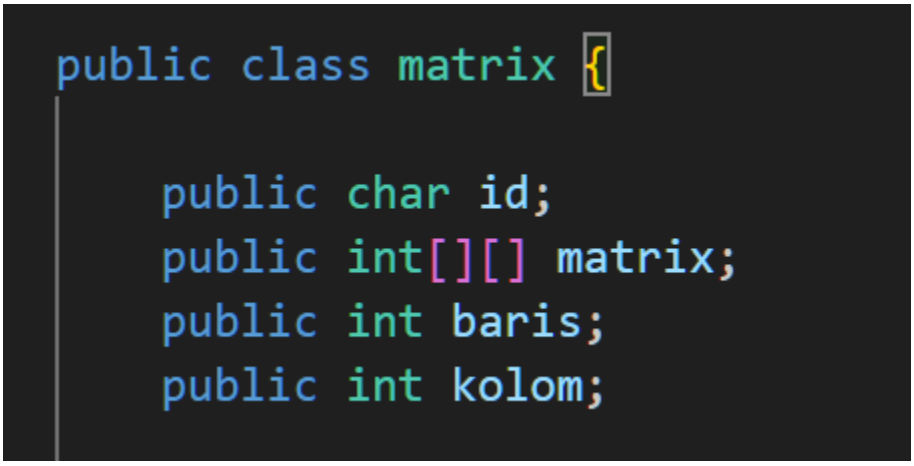
Bab II

Implementasi *Source Code* Algoritma

Program dibentuk dalam bahasa pemrograman Java dengan pendekatan berbasis objek. Direktori file utama program terdiri atas beberapa file Java sebagai berikut:

1. **matrix.java**

Matrix dipilih sebagai kelas yang merepresentasikan potongan puzzle dalam permainan. Pemilihan ini didasarkan terhadap atribut-atribut matrix yang memberikan tingkatan abstraksi yang baik terhadap panjang, lebar, dan bentuk dari potongan puzzle.



Gambar 2. Atribut kelas matrix

Atribut *id* yang dimiliki oleh matrix berperan sebagai penyimpan terhadap huruf yang digunakan untuk mendeskripsikan potongan puzzle tersebut pada papan. Penyimpanan terhadap huruf ini juga membuat komponen matrix untuk diimplementasikan menggunakan *integer* 0 dan 1, 0 untuk menandakan bahwa tidak terdapat potongan puzzle pada bagian matrix tersebut, dan sebaliknya.

2. **solver.java**

Kelas solver bertanggung jawab dalam menyelesaikan permainan menggunakan algoritma *brute force*.

```java
public class solver extends mainSolver {

    public boolean gagal = false;
    public int barisPapan;
    public int kolomPapan;
    public char[][] papan;
```

Gambar 3. Atribut kelas solver

Atribut *papan, barisPapan,* dan *kolomPapan* menjelaskan karakteristik dari papan permainan yang digunakan, sedangkan *gagal* digunakan sebagai semacam variabel global yang menyatakan bahwa permainan telah gagal dan tidak dapat diselesaikan, untuk kemudian menghentikan algoritma berlangsung. Konstruktor pembentuk kelas solver dibagi menjadi dua berdasarkan konfigurasi yang dimasukkan pengguna, DEFAULT atau CUSTOM, untuk kemudian membentuk data ketersediaan slot untuk diisi pada papan.

```java
/* konstruktor solver */
public solver(int barisPapan, int kolomPapan, boolean custom, List<List<Character>> papanCustom) {
    this.barisPapan = barisPapan;
    this.kolomPapan = kolomPapan;
    this.papan = new char[barisPapan][kolomPapan];

    if (!custom) {
        for (int i = 0; i < barisPapan; i++) {
            for (int j = 0; j < kolomPapan; j++) {
                papan[i][j] = ' ';
            }
        }

    } else {
        for (int i = 0; i < barisPapan; i++) {
            for (int j = 0; j < kolomPapan; j++) {
                if (papanCustom.get(i).get(j) == 'X') {
                    papan[i][j] = ' ';
                } else {
                    papan[i][j] = '.';
                }
            }
        }
    }
}
```

Gambar 4. Konstruktor kelas solver

Sesuai dengan penjelasan algoritma *brute force* pada permainan ini, metode *taruhPuzzle* mencoba untuk meletakkan salah satu kemungkinan posisi potongan puzzle pada slot berkoordinat *currentBaris* dan *currentKolom*. Apabila dapat diletakkan, potongan puzzle akan

diletakkan pada slot tersebut dengan menambahkan ID dari potongan puzzle pada slot-slot tersebut.

```java
/* meletakkan puzzle pada papan */
public boolean taruhPuzzle(matrix currentPuzzleOrientation, int currentBaris, int currentKolom, boolean custom) {
    for (int i = 0; i < currentPuzzleOrientation.getBaris(); i++) {
        for (int j = 0; j < currentPuzzleOrientation.getKolom(); j++) {
            if (currentPuzzleOrientation.matrix[i][j] == 1) {
                int newBaris = currentBaris + i;
                int newKolom = currentKolom + j;

                if (!custom) {
                    if (newBaris >= barisPapan || newKolom >= kolomPapan || papan[newBaris][newKolom] != ' ') {
                        return false;
                    }

                } else {
                    if (newBaris >= barisPapan || newKolom >= kolomPapan || papan[newBaris][newKolom] != ' ' || papan
                        return false;
                    }
                }
            }
        }
    }

    for (int i = 0; i < currentPuzzleOrientation.getBaris(); i++) {
        for (int j = 0; j < currentPuzzleOrientation.getKolom(); j++) {
            if (currentPuzzleOrientation.matrix[i][j] == 1) {
                papan[currentBaris + i][currentKolom + j] = currentPuzzleOrientation.getID();
            }
        }
    }

    return true;
}
```

Gambar 5. Metode taruhPuzzle()

Kelas solver juga memiliki metode untuk melakukan *backtrack* bernama *buangPuzzle*, metode tersebut menghapus ID yang telah dimasukkan sebelumnya ke dalam papan.

```java
/* melakukan backtrack dengan menghapus puzzle yang telah diletakkan pada papan */
public void buangPuzzle(matrix currentPuzzleOrientation, int currentBaris, int currentKolom) {
    for (int i = 0; i < currentPuzzleOrientation.getBaris(); i++) {
        for (int j = 0; j < currentPuzzleOrientation.getKolom(); j++) {
            if (currentPuzzleOrientation.matrix[i][j] == 1) {
                papan[currentBaris + i][currentKolom + j] = ' ';
            }
        }
    }
}
```

Gambar 6. Metode buangPuzzle()

Kedua metode tersebut digunakan untuk membentuk metode utama dalam kelas ini, yaitu metode *solve*. Menerima hasil iterasi semua kemungkinan posisi yang mungkin dari setiap

potongan puzzle yang tersedia, metode *solve* melakukan iterasi terhadap letak baris, letak kolom, dan kemungkinan posisi yang mungkin.

```
/* metode untuk menyelesaikan puzzle */
public int[] solve(matrix[] puzzles, boolean custom) {
    int iterations = 0;
    int index = 0;

    // menyimpan baris, kolom, dan orientasi puzzle yang sedang diletakkan
    int[][] kondisiPuzzle = new int[puzzles.length][3];
    for (int i = 0; i < puzzles.length; i++) {
        kondisiPuzzle[i] = new int[]{-1, -1, -1};
    }

    while (0 <= index  && index < puzzles.length) {
        preprocessing preprocessing = new preprocessing();
        matrix currentPuzzle = puzzles[index];
        List<matrix> currentPuzzleAllOrientations = preprocessing.otakAtik(currentPuzzle);

        boolean placed = false;

        int barisAwal, kolomAwal, orientasiAwal;
        if (kondisiPuzzle[index][0] == -1) {
            barisAwal = 0;
        } else {
            barisAwal = kondisiPuzzle[index][0];
        }

        if (kondisiPuzzle[index][1] == -1) {
            kolomAwal = 0;
        } else {
            kolomAwal = kondisiPuzzle[index][1];
        }

        if (kondisiPuzzle[index][2] == -1) {
            orientasiAwal = 0;
        } else {
            orientasiAwal = kondisiPuzzle[index][2];
        }
```

Gambar 7. Metode solve() (1)

Pada gambar 7, setelah *while loop* dimulai, dilakukan perhitungan terhadap peletakan potongan puzzle. Jika potongan puzzle belum pernah diletakkan, maka kemungkinan posisi pertama potongan puzzle dicoba untuk diletakkan pada slot pertama papan, sesuai dengan inisialisasi variabel *barisAwal*, *kolomAwal*, dan *orientasiAwal* dengan nilai nol. Namun, jika dilakukan *backtrack* pada potongan puzzle yang telah diletakkan, maka nilai dari variabel-variabel tersebut merupakan nilai yang telah disimpan dalam *int*[][] *kondisiPuzzle*.

```
for (int currentBaris = barisAwal; currentBaris < barisPapan; currentBaris++) {
    int temp;
    if (currentBaris == barisAwal) {
        // ketika pengecekan dimulai di baris yang sama, kolom dimulai dari kolom terakhir + 1
        temp = kolomAwal;
    } else {
        // ketika pengecekan dilanjutkan ke baris yang baru, kolom dimulai dari 0
        temp = 0;
    }

    for (int currentKolom = temp; currentKolom < kolomPapan; currentKolom++) {
        for (int currentOrientasi = orientasiAwal; currentOrientasi < currentPuzzleAllOrientations.size(); currentOrientasi++) {
            matrix currentPuzzleOrientation = currentPuzzleAllOrientations.get(currentOrientasi);

            if (taruhPuzzle(currentPuzzleOrientation, currentBaris, currentKolom, custom)) {
                iterations++;
                kondisiPuzzle[index] = new int[]{currentBaris, currentKolom, currentOrientasi};
                placed = true;
                break;
            }

        }
        if (placed) {
            break;
        }
    }
    if (placed) {
        break;
    }
}

if (placed) {
    // puzzle berhasil ditempatkan, lanjut ke puzzle selanjutnya
    index++;
```

Gambar 8. Metode solve() (2)

```
        } else {
            // puzzle gagal ditempatkan, coba kemungkinan lain puzzle sebelumnya
            kondisiPuzzle[index] = new int[]{-1, -1, -1};
            index--;

            if (index >= 0) {
                matrix puzzleUntukDihapus = preprocessing.otakAtik(puzzles[index]).get(kondisiPuzzle[index][2]);
                buangPuzzle(puzzleUntukDihapus, kondisiPuzzle[index][0], kondisiPuzzle[index][1]);

                kondisiPuzzle[index][2]++;

                if (kondisiPuzzle[index][2] >= preprocessing.otakAtik(puzzles[index]).size()) {
                    kondisiPuzzle[index][2] = 0;
                    kondisiPuzzle[index][1]++;

                    if (kondisiPuzzle[index][1] >= kolomPapan) {
                        kondisiPuzzle[index][1] = 0;
                        kondisiPuzzle[index][0]++;
                    }
                }
            }
        }
    }

    if (index == -1) {
        return new int[]{0, iterations};
    }

    if (cekPapan()) {
        return new int[]{1, iterations};
    } else {
        gagal = true;
        return new int[]{0, iterations};
    }
}
```

Gambar 9. Metode solve() (3)

Lalu dilakukan iterasi peletakan dan penghapusan puzzle pada papan menggunakan metode *taruhPuzzle* dan *buangPuzzle*. Jika potongan puzzle berhasil diletakkan, akan dilakukan iterasi peletakan terhadap potongan puzzle selanjutnya. Namun, jika potongan puzzle tidak berhasil diletakkan, sesuai dengan gambar 9, maka dilakukan *backtrack* dengan mengubah posisi potongan puzzle sebelumnya, atau slot kolom jika seluruh kemungkinan posisi telah dicoba, atau slot baris jika seluruh slot kolom telah dicoba.

Jika seluruh kemungkinan telah dicoba dan gagal, maka *index* akan bernilai -1 dinyatakan gagal. Di sisi lain, jika terdapat kondisi dimana seluruh potongan puzzle berhasil diletakkan, maka dilakukan pengecekan melalui metode *cekPapan* apakah seluruh papan tertutupi puzzle atau tidak. Jika iya, maka kombinasi peletakan tersebut berhasil, sedangkan jika terdapat bagian yang tidak tertutupi puzzle, maka permainan dikatakan gagal dan langsung dihentikan karena tidak mungkin ada kombinasi yang mungkin menutupi seluruh bagian papan. Metode mengembalikan *list of integers* dengan *integer* pertama menyatakan berhasil atau gagal, dan *integer* kedua menyatakan jumlah iterasi yang dilakukan, yaitu nilai berapa kali suatu potongan puzzle dapat berhasil diletakkan pada papan.

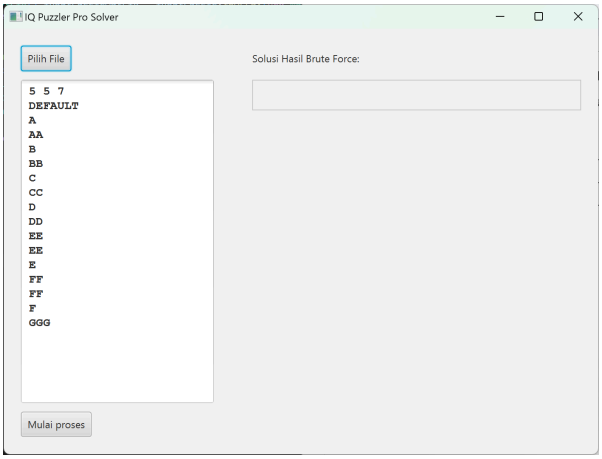
3. **mainSolver.java dan Main.java**

Setelah didapatkan hasil iterasi, baik berhasil maupun tidak, data hasil iterasi diberikan kepada *parent class* dari kelas solver, yaitu kelas mainSolver. Kelas ini menjembatani pengolahan data dengan menyesuaikan tipe-tipe data menjadi kompatibel untuk digunakan dalam *graphical user interface* (GUI), yaitu metode-metode yang dimiliki oleh kelas Main pada file Main.java. Kelas mainSolver juga bertanggung jawab untuk membaca masukan file .txt dari pengguna dan membentuknya menjadi tipe data yang dapat diterima oleh kelas solver.

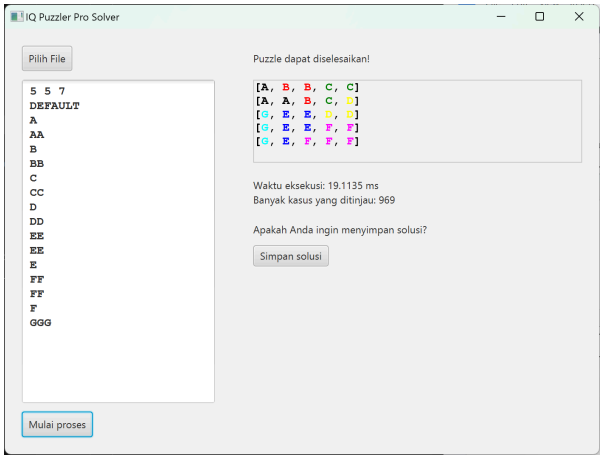
Bab III

Eksperimen

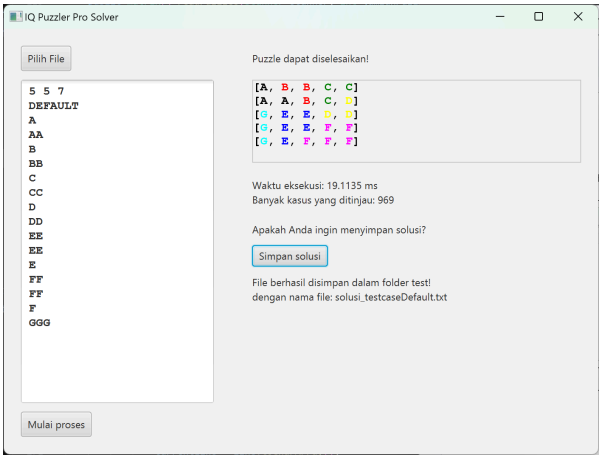
1. Konfigurasi default 1



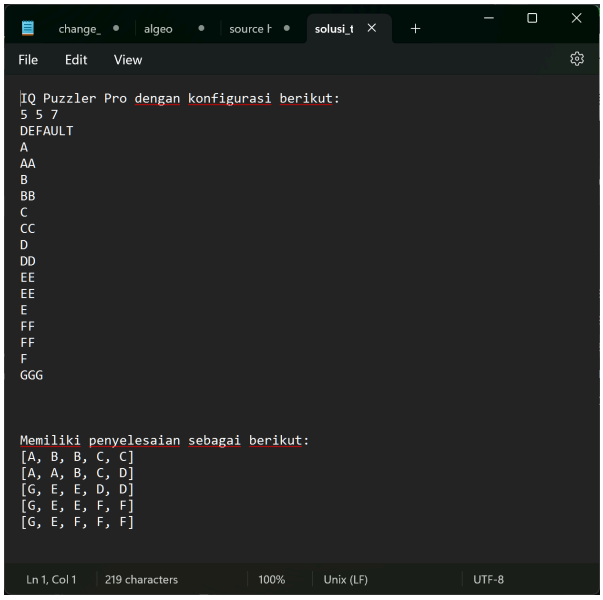
(a) Input dari file .txt



(b) Hasil algoritma *brute force*



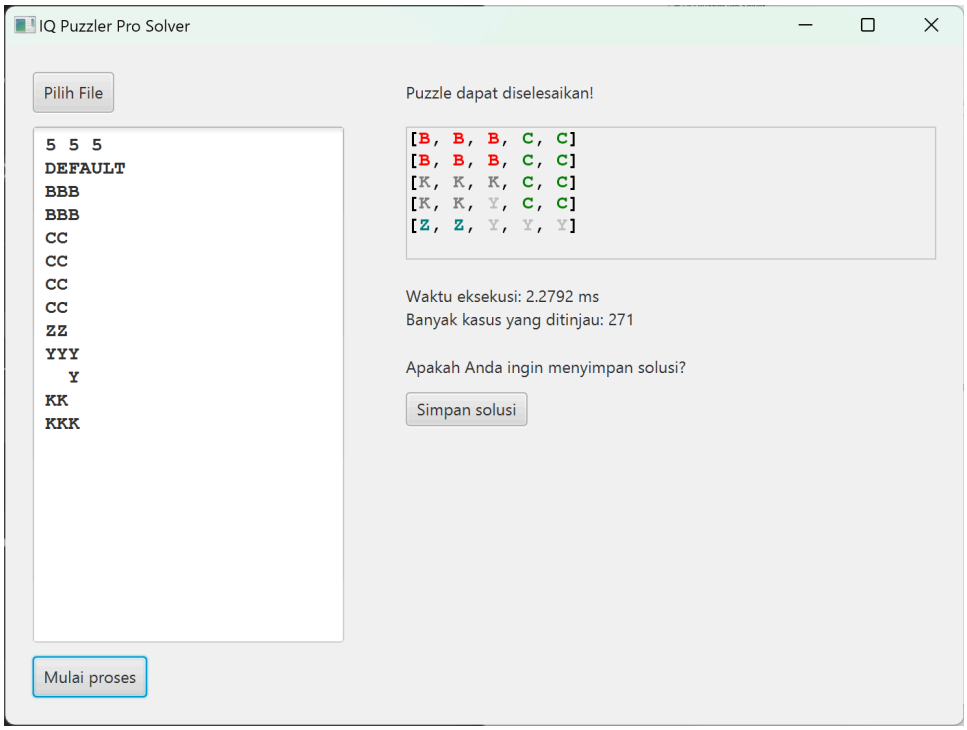
(c) Penyimpanan hasil ke dalam file .txt



(d) File .txt hasil algoritma

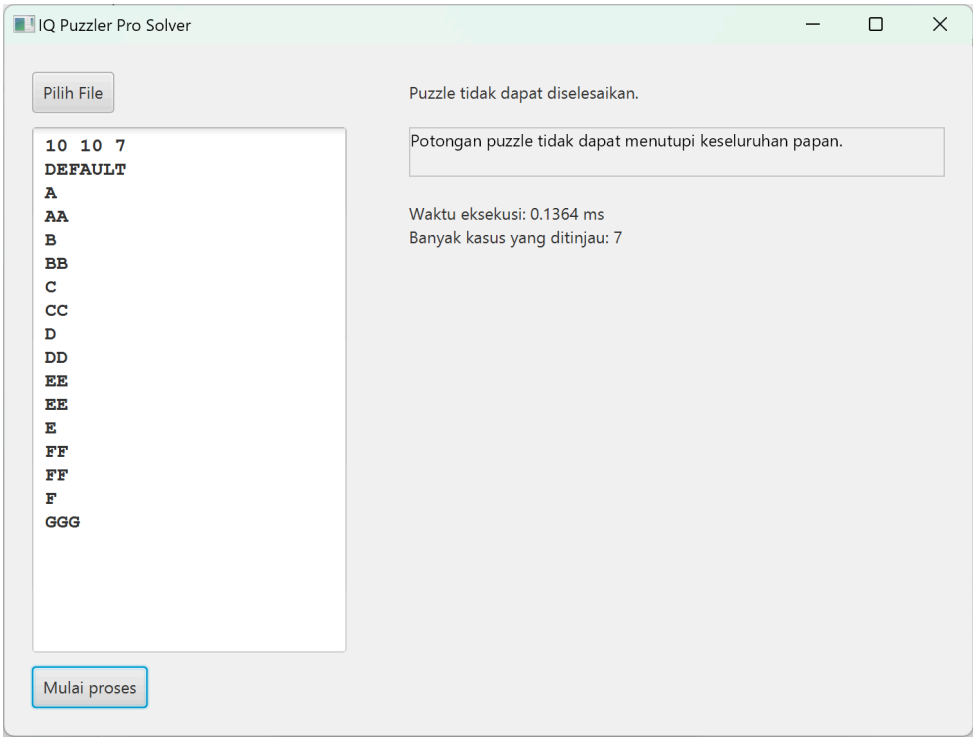
Gambar 10. Tangkapan layar konfigurasi default 1

2. Konfigurasi default 2



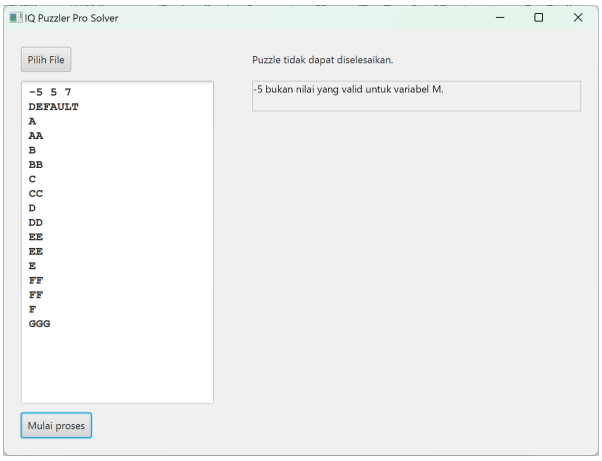
Gambar 11. Tangkapan layar konfigurasi default 2

3. Potongan puzzle tidak menutupi papan

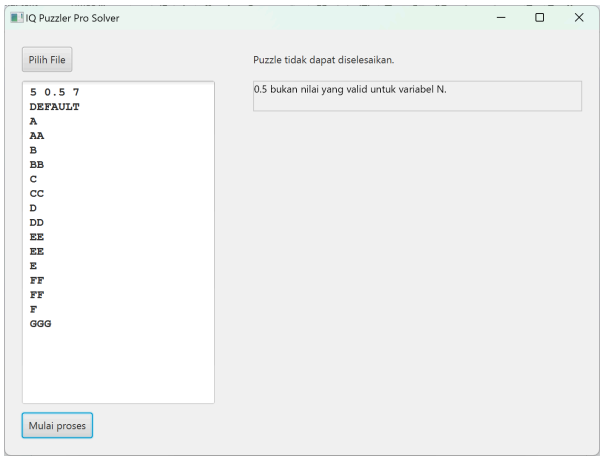


Gambar 12. Tangkapan layar kondisi potongan puzzle tidak menutupi papan

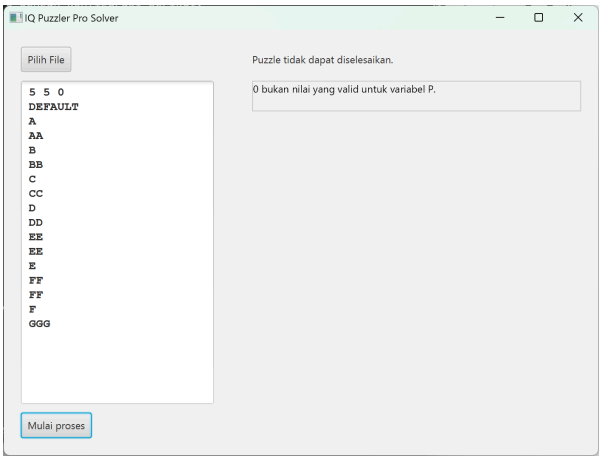
4. Nilai variabel tidak valid



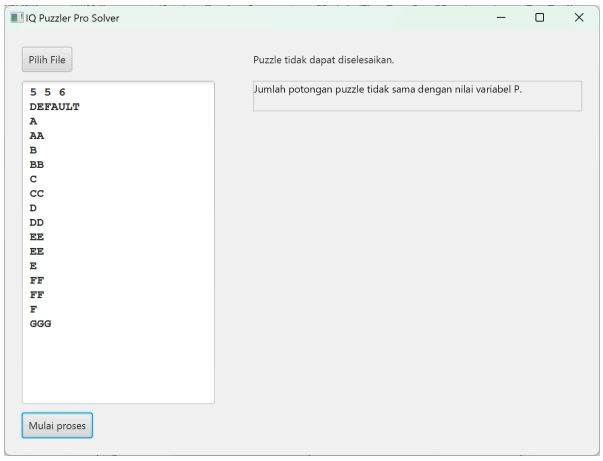
(a) variabel M, N, P tidak dapat bernilai negatif



(b) variabel M, N, P tidak dapat tidak bilangan bulat



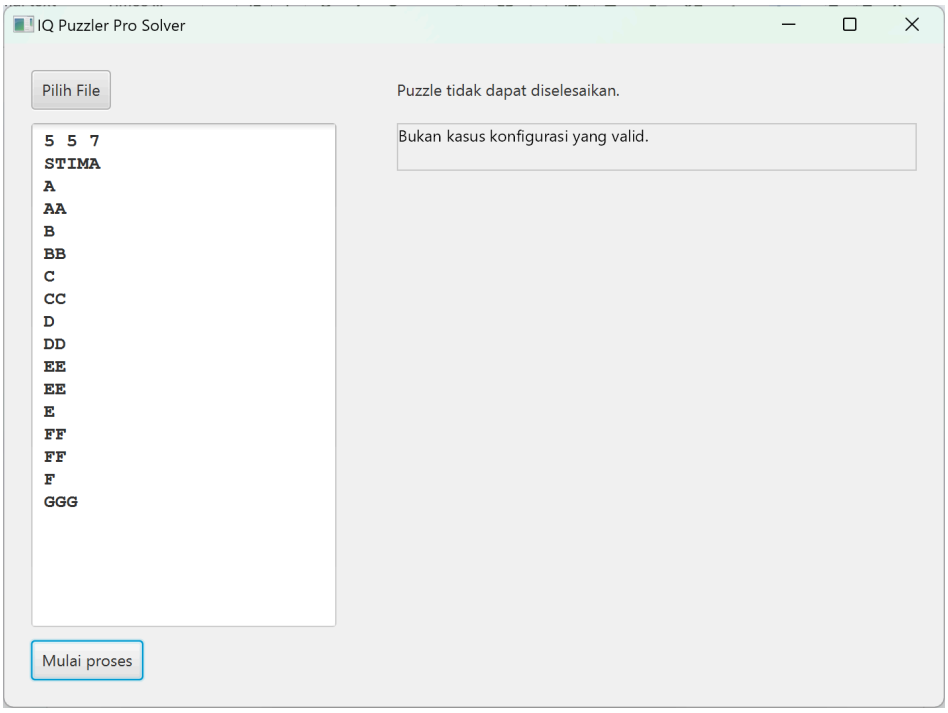
(c) variabel M, N, P tidak dapat bernilai nol



(d) Nilai variabel P tidak sama dengan jumlah puzzle

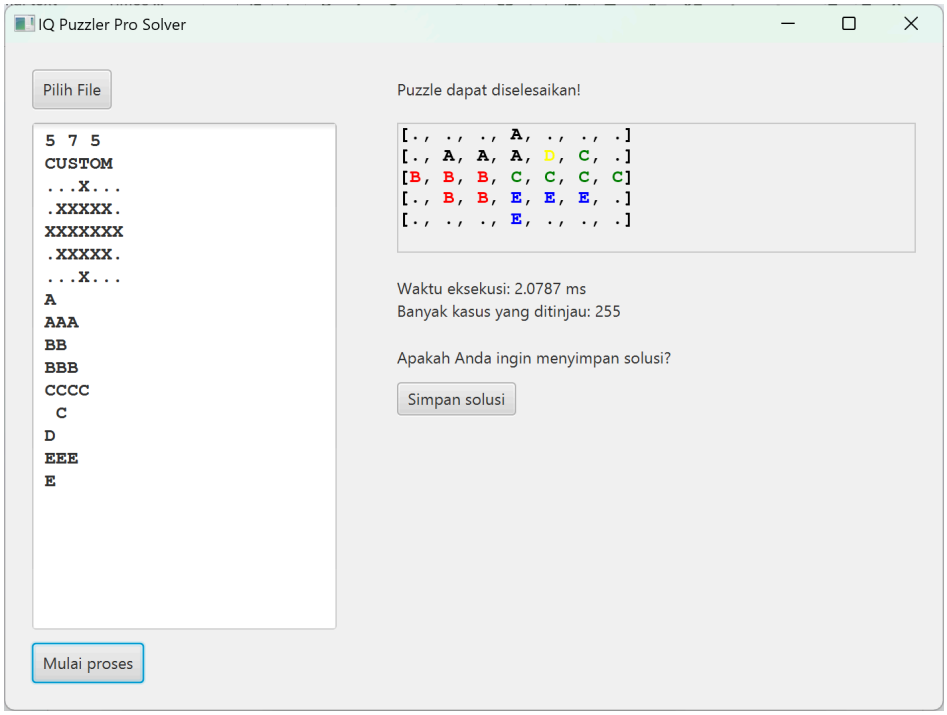
Gambar 13. Tangkapan layar kondisi nilai variabel tidak valid

5. Kasus konfigurasi selain “DEFAULT” atau “CUSTOM”



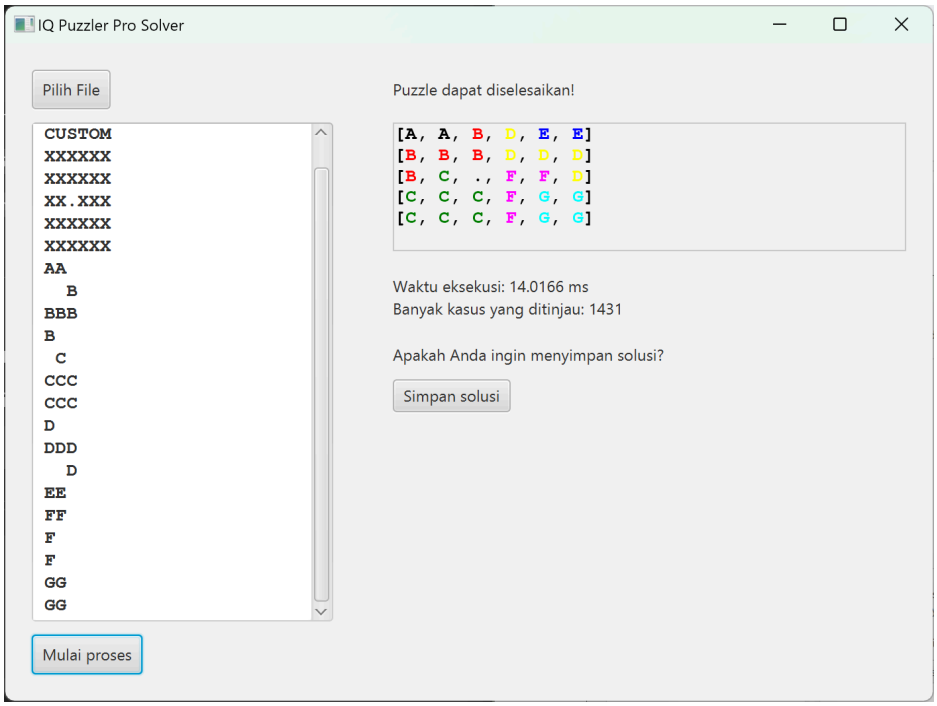
Gambar 14. Tangkapan layar kasus konfigurasi invalid

6. Konfigurasi custom 1



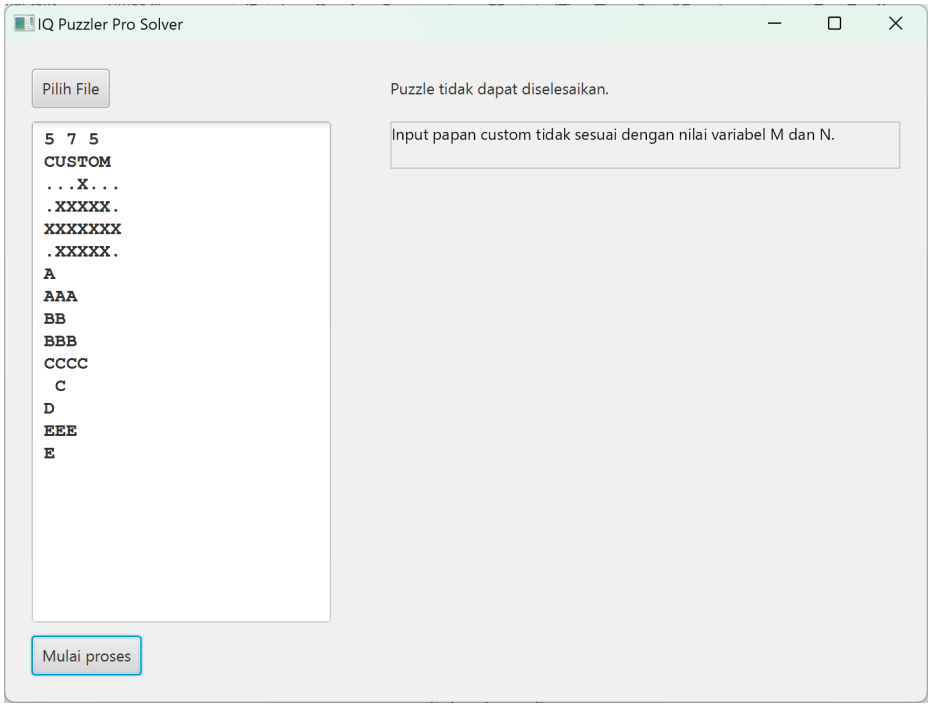
Gambar 15. Tangkapan layar konfigurasi custom 1

7. Konfigurasi custom 2



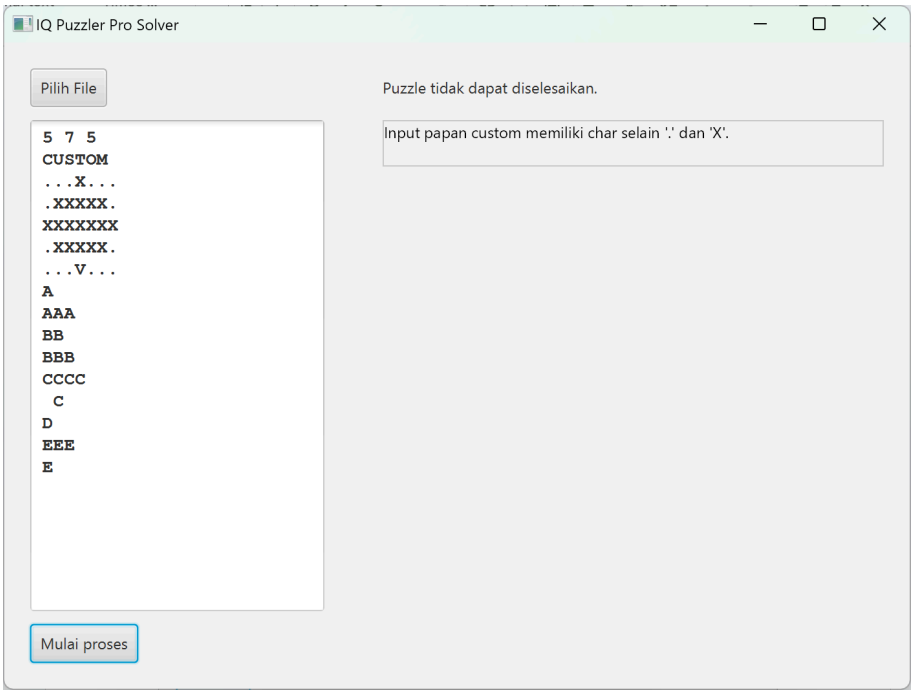
Gambar 16. Tangkapan layar konfigurasi custom 2

8. Papan custom tidak sesuai variabel M dan N



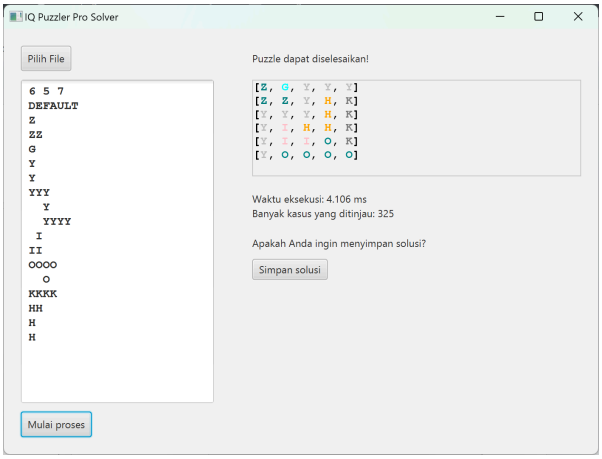
Gambar 17. Tangkapan layar papan custom tidak sesuai ukuran

9. Papan custom memiliki *char* lain selain ‘.’ dan ‘X’

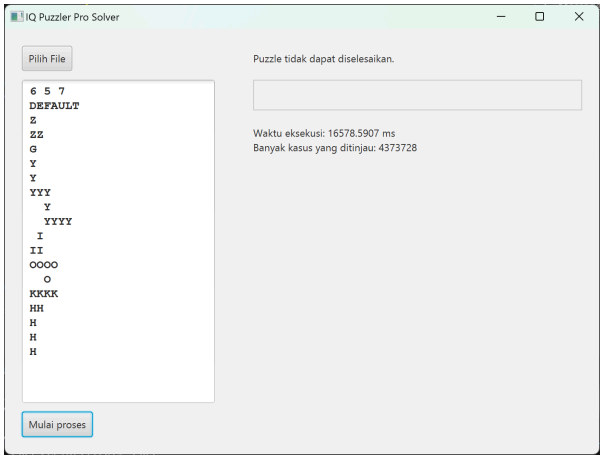


Gambar 18. Tangkapan layar papan custom mengandung *char* lain

10. Konfigurasi default, berhasil dan gagal



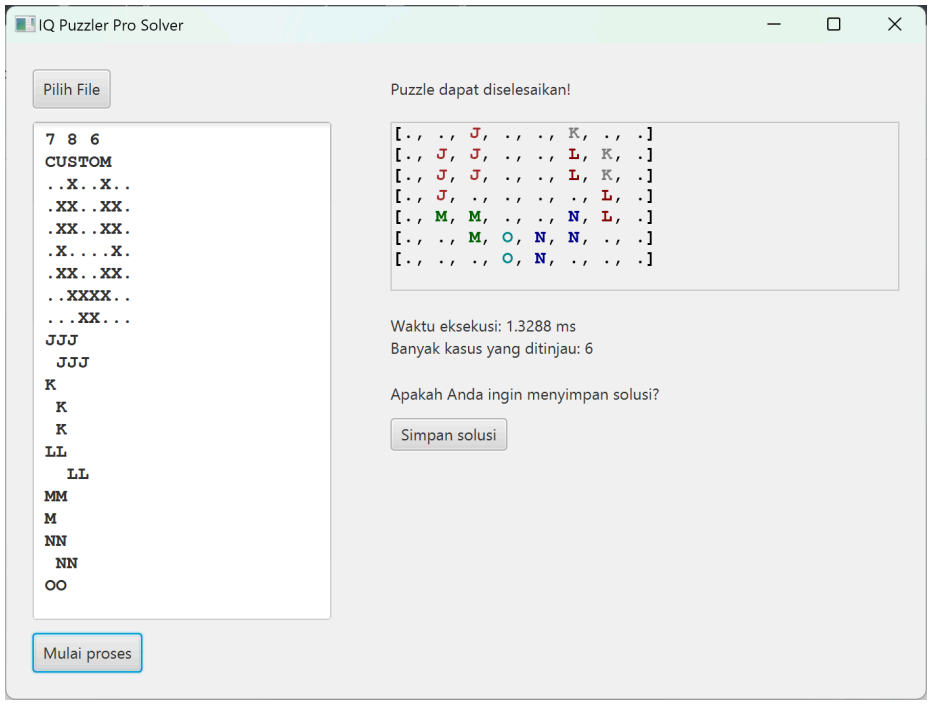
(a) Berhasil untuk konfigurasi default



(b) Gagal untuk konfigurasi default

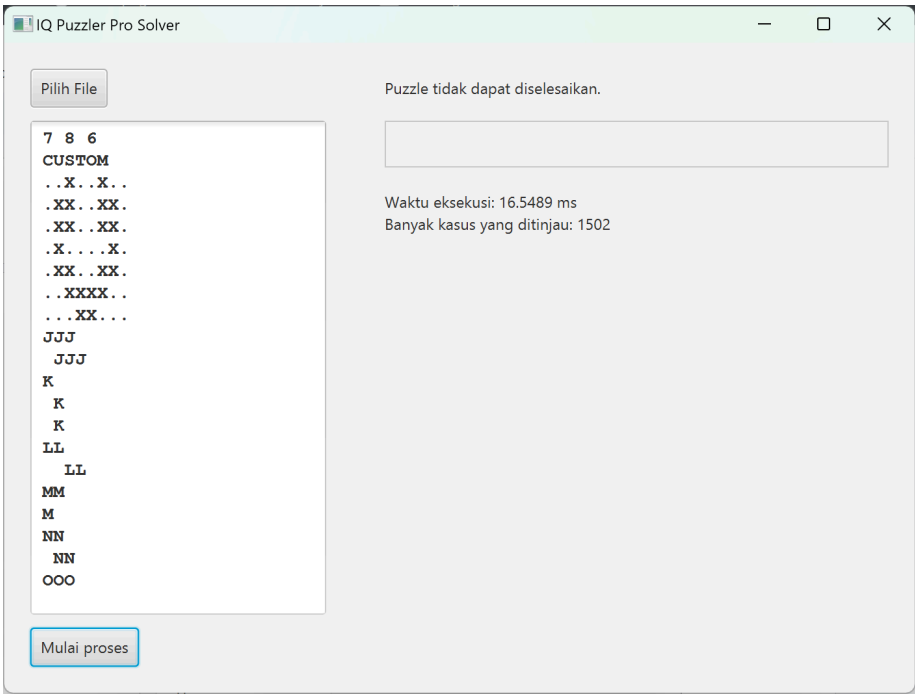
Gambar 19. Tangkapan layar berhasil dan gagal untuk konfigurasi default

11. Konfigurasi custom membaca input puzzle diagonal



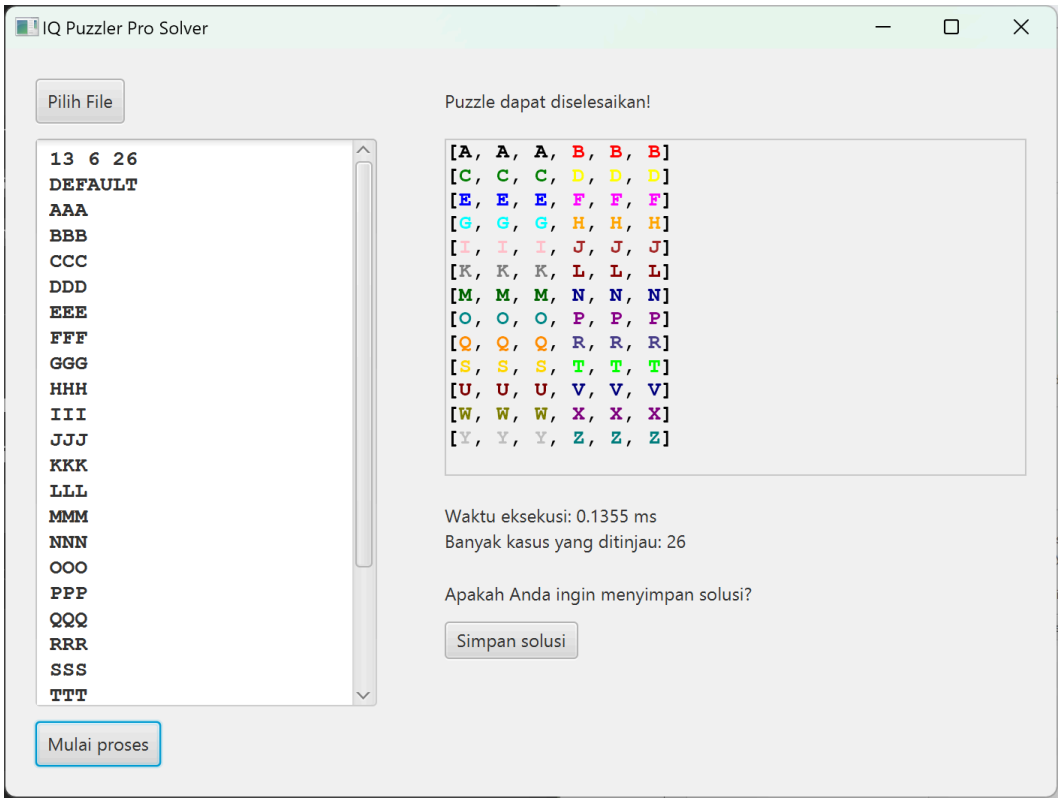
Gambar 20. Tangkapan layar konfigurasi custom dengan input diagonal

12. Kegagalan untuk konfigurasi custom



Gambar 21. Tangkapan layar gagal untuk konfigurasi custom

13. Pengecekan warna



Gambar 22. Pengecekan warna setiap huruf

Bab IV

Lampiran

1. **Referensi**

Informatika.stei.itb.ac.id. (2025). Algoritma Brute Force (Bagian 1). Diakses pada 23 Februari 2024, dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/02-Algoritma-Brute-Force-(2025)-Bag1.pdf

2. **Pranala *repository***

https://github.com/rafenmaxxx/Tucil1_13523031.git

3. **Tabel *Checklist***

Tabel IV.3 Tabel *checklist*

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		✔
7	Program dapat menyelesaikan kasus konfigurasi *custom*	✔	
8	Program dapat menyelesaikan kasus konfigurasi Piramida (3D)		✔
9	Program dibuat oleh saya sendiri	✔	