Laporan Tugas Kecil 1

IQ Puzzler Pro Solver

Disusun untuk memenuhi tugas mata kuliah IF2211 Strategi Algoritma pada Semester 2 (Genap) Tahun Akademik 2024/2025

Disusun oleh:

Razi Rachman Widyadhana 13523004

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
JL. GANESA 10, BANDUNG 40132
2025

Daftar Isi

Bab I
Deskripsi Masalah

IQ Puzzler Pro adalah permainan papan yang diproduksi oleh perusahaan Smart Games. Tujuan dari permainan ini adalah pemain harus dapat mengisi seluruh papan dengan piece (blok puzzle) yang telah tersedia.

Komponen penting dari permainan IQ Puzzler Pro terdiri dari:

1. **Board (Papan):** komponen utama yang menjadi tujuan permainan dimana pemain harus mampu mengisi seluruh area papan menggunakan blok-blok yang telah disediakan.
2. **Piece (Blok):** komponen yang digunakan pemain untuk mengisi papan kosong hingga terisi penuh. Setiap blok memiliki bentuk yang unik dan semua blok harus digunakan untuk menyelesaikan puzzle.

Gambar 1: Permainan IQ Puzzler Pro (Sumber: Smart Game USA)

Permainan dimulai dengan papan yang kosong. Pemain dapat meletakkan blok puzzle sedemikian sehingga tidak ada blok yang bertumpang tindih (kecuali dalam kasus 3D). Setiap blok puzzle dapat dirotasikan maupun dicerminkan. Puzzle dinyatakan selesai jika dan hanya jika papan terisi penuh dan seluruh blok puzzle berhasil diletakkan.

Tujuan program yang akan dibuat adalah menemukan **cukup satu solusi** dari permainan **IQ Puzzler Pro** dengan menggunakan **algoritma Brute Force**, atau menampilkan bahwa solusi tidak ditemukan jika tidak ada solusi yang mungkin dari puzzle.

Bab II

Algoritma Bruteforce dalam Penyelesaian IQ Puzzler Pro

2.1. Algoritma *Brute force*

Algoritma *brute force* adalah strategi pencarian simpel dan komprehensif yang secara sistematis mengeksplorasi setiap opsi hingga jawaban suatu masalah ditemukan. Algoritma ini memecahkan persoalan secara sederhana, langsung (*straightforward*), serta dengan cara yang jelas (*obvious*) dan mudah dipahami. Algoritma ini juga sering disebut algoritma naif (*naïve algorithm*).

Mayoritas algoritma *brute force* bukanlah algoritma yang "cerdas" dan tidak sangkil. Hal ini disebabkan algoritma *brute force* membutuhkan biaya komputasi yang besar dan waktu yang lama dalam penyelesaiannya (Rinaldi, 2025). Oleh karena itu, algoritma *brute force* lebih cocok digunakan untuk persoalan yang memiliki ukuran masukannya (n) kecil, memanfaatkan implementasinya yang juga relatif lebih mudah dan sederhana.

Algoritma *brute force* sering digunakan sebagai basis pembanding dengan algoritma lain yang lebih mangkus. Meskipun bukan merupakan pendekatan yang sangkil, hampir seluruh persoalan dapat dipecahkan dengan algoritma brute force, bahkan ada persoalan yang hanya dapat diselesaikan dengan *brute force*.

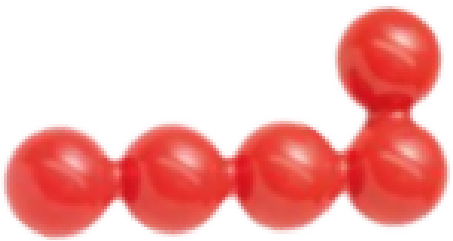
2.2. Abstraksi Permasalahan

Seperti yang telah didefinisikan sebelumnya, algoritma *brute force* akan mencoba seluruh opsi/kemungkinan hingga menemukan sebuah solusi.

Ingat kembali bahwa terdapat dua komponen penting dalam permainan IQ Puzzler Pro, yaitu *Board*/Papan dan *Piece*/Blok. Dengan meninjau masing-masing komponen terlebih dahulu, akan memudahkan proses abstraksi untuk algoritma *brute force* ini.

2.2.1 *Piece*

Sebuah *piece* terdiri atas satu atau lebih unit sel (*parts*). Untuk lebih jelasnya, satu unit sel mewakili satu slot pada *board*.



Gambar 2: Sebuah *piece* dengan 5 unit sel/*parts*.

Dari Gambar 2. Apabila *piece* tersebut dirotasi sebesar 90° tentu akan menghasilkan suatu "*piece*" yang baru. Alhasil, sebanyak

$$360°/90° = 4$$

kemungkinan yang akan terbuat. Akan tetapi, tidak sampai di situ. Apabila *piece* tersebut dibalik (*flipped*) maka secara umum terdapat

$$4 \times 2 = 8$$

total kemungkinan tiap *piece*.

Pendekatan ini masih dapat dioptimalisasi lagi. Hal ini akan dibahas setelah ini, sekarang alihkan fokus terhadap bagaimana caranya untuk merepresentasikan *piece-piece* ini agar dapat diletakkan ke *Board* dengan metode semudah dan sesimpel mungkin.

Secara naif, pertama kali yang terpikir adalah merepresentasikannya sesuai dengan aslinya, yaitu dengan mencatat unit sel/*parts* dari *piece* tersebut. Sebagai gambaran, untuk Gambar 2., dengan memisalkan unit sel/*parts* terkiri dan terbawah adalah $(0, 0)$, maka

$$U = \{(0, 0), (1, 0), (2, 0), (3, 0), (3, 1)\}$$

Akan tetapi, representasi ini akan menyulitkan proses pemasangan nantinya. Pendekatan lainnya adalah menggunakan representasi **Matriks** dengan panjang dan lebar maksimal dari suatu *piece* tersebut, yang nantinya akan ditandai seluruh unit sel/*parts* yang ada. Kembali lagi pada Gambar 2., maka menjadi

$$U = \begin{bmatrix} . & . & . & M \\ M & M & M & M \end{bmatrix}$$

Tentunya hal ini akan memudahkan dalam proses pemasangan dibanding representasi titik-titik. Mengapa? karena hanya perlu dicek ketika isi dari matriks tersebut tertanda, apakah valid untuk diisi atau tidak. Apabila tidak tertanda, tidak perlu dicek.

Baik, sekarang kembali kepada kemungkinan/permutasi dari suatu *piece*. Ketika terdapat *piece* yang simetris baik secara horizontal, vertikal, ataupun keduanya, maka akan berkurang total permutasinya disebabkan sifat kesimetriannya tersebut. Berikut contohnya:

$$U = \begin{bmatrix} M & M \\ M & M \end{bmatrix} \quad \text{dan} \quad U = \begin{bmatrix} M & M & M \end{bmatrix}$$

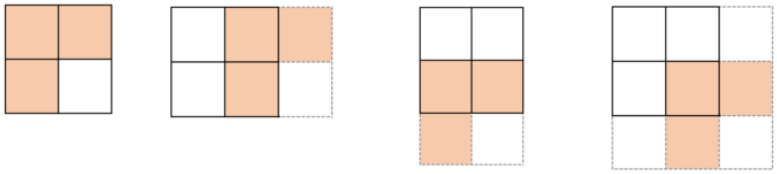
Apabila dibiarkan, tentunya akan dilakukan komputasi yang *redundant*/mubazir. Untuk mencegah hal ini, pendekatannya cukup sederhana dengan menggunakan *Hashmap* untuk hanya menyimpan permutasi-permutasi yang unik.

2.2.2 *Board*

Sebuah *board*/papan (tipe default) dengan panjang/tinggi N dan lebar M akan memiliki unit sel sebanyak

$$N \times M$$

Artinya, suatu *piece* dapat diiterasi sebanyak N tempat pada panjang/tinggi *board* tersebut dan sebanyak M tempat pada lebar *board* tersebut. Untuk memudahkan pemahaman, berikut visualisasi proses iterasi suatu *piece* pada papan berukuran 2×2



Gambar 3: Proses pemasangan *piece* pada *board*.

Dari ilutrasi di atas, digunakan unit sel teratas dan terkiri *piece* sebagai patokan pemasangan sehingga meskipun unit sel patokan masih di dalam *board*, unit sel lainnya sudah tidak valid. Hal ini hanya sebagai visualisasi dan tentunya akan divalidasi sebelum *piece* terpasang pada *board*.

2.2.3 *Backtracking*

Ada suatu hal lagi yang perlu menjadi pertimbangan dalam optimalisasi algoritma *brute force* yang akan digunakan. Sebagai visualisasi, terdapat *piece-piece* sebagai berikut yang akan ditempatkan pada *board* (tipe default) 3×4

Gambar 4: *pieces* terurut pertama dari kiri.

Secara urutan, *piece* krem dan biru akan diproses terlebih dahulu. Apabila telah ditaruh keduanya pada posisi *board* terkiri dan teratas secara berurutan. Kemudian, akan dipasang *piece* hijau. Apabila jika tidak ada satupun posisi penempatan *piece* hijau yang valid, maka *piece* hijau tersebut dihapus dan kembali ke *piece* sebelumnya, yaitu *piece* biru. Hal ini terus berulang hingga kedua *piece* krem dan biru berada tepat keduanya di tengah-tengah *board*.

Meskipun *best case* kasus tersebut terjadi apabila *piece* hijau mendapat urutan pertama sehinga tidak akan terjadi penghapusan *piece*. Akan tetapi, program akan mengolah *piece* sesuai urutan pada *input*/masukan yang diberikan

Salah satu teknik optimalisasi ini cukup berguna untuk diimplementasikan. Dibanding dengan pendekatan yang ketika dirasa mustahil suatu kemungkinan menjadi solusi, program akan "mereset" *board* dan kembali dari *piece* pertama.

2.2.4 *Brute Force*

Dari proses abstraksi yang telah dipaparkan, diperolehlah metode untuk menemukan solusi dari permainan IQ Puzzler Pro secara algoritmik. Adapun algoritma *brute force* yang digunakan adalah sebagai berikut.

1. Proses terlebih dahulu seluruh *input*/masukan yang ada menjadi *board* dan *piece-piece* yang direpresentasikan sebagai matriks 2D
2. Secara rekursif untuk setiap *piece*, program akan melakukan iterasi untuk sebanyak N tempat pada panjang/tinggi *board* tersebut dan sebanyak M tempat pada lebar *board* dengan posisi pertama menempatkan *piece* pada (terkiri dan teratas) pada *board*.
3. Validasi apakah *piece* dapat diletakkan, jika dapat diletakkan maka program akan lanjut ke *piece* berikutnya.
4. Jika poin [3] tidak dapat diletakkan, maka program akan melakukan iterasi untuk setiap permutasi unik pada *piece* tersebut (*rotation* dan *flip*)
4. Jika poin [4] tetap tidak dapat diletakkan, maka *piece* tersebut dihapus sementara dan kembali berfokus pada penempatan *piece* tepat sebelumnya.
5. Proses akan berulang hingga bertemu kasus basis, yaitu seluruh *piece* berhasil diletakkan pada *board* (sukses menemukan solusi) atau mustahil seluruh *piece* diletakkan pada *board* (tidak ditemukan solusi).

Bab III
Implementasi Program dengan Java

Algoritma yang telah dijelaskan pada Bab 2 diimplementasikan dengan bahasa pemrograman `Java`. Untuk menyesuaikan proses perancangan program yang *intended* pada Java, digunakan pendekatan *object-oriented programming*, di mana setiap komponen utama program akan dienkapulasi oleh *Object*. Untuk program ini, sebenarnya terdapat beberapa kelas. Akan tetapi, tiga kelas utama yang berperan sebagai *core logic* program ini adalah `Bruteforce`, `Board`, dan `Piece`.

3.1. Kelas `Bruteforce`

```java
package src;

public class Bruteforce
{
    private int attempts;
    private Board board;
    private Piece[] pieces;
    private Piece[][] permutations;\

    public int getAttempts() {return this.attempts;}

    public Bruteforce(Board board, Piece[] pieces)
    {
        this.attempts = 0;
        this.board = board;
        this.pieces = pieces;
        this.permutations = Piece.uniquePermutations(pieces);
    }

    public boolean search(int pieceIndex)
    {
        attempts++;

        // Base case: If all pieces have been placed, return true.
        if (pieceIndex == pieces.length)
        {
            return true;
        }

        // Try to place the current piece at every board cell.
        for (int i = 0; i < board.getHeight(); i++)
        {
            for (int j = 0; j < board.getWidth(); j++)
            {
                // Get all unique transformations (permutations) of the current piece.
                Piece[] permutations = this.permutations[pieceIndex];
                for (Piece permuation : permutations)
                {
                    // Attempt to place the piece at (i, j)
                    if (board.fitPiece(i, j, permuation))
                    {
                        if (search(pieceIndex + 1))
                        {
                            return true;
                        }
                        board.removePiece(i, j, permuation);
                    }
                }
            }

        }

        // Base case: If no piece can be placed, return false.
        return false;
    }

}
```

Gambar 5: Implementasi Java dari Kelas `Bruteforce`

Dalam Kelas `Bruteforce`, terdapat konstruktor untuk dirinya, yaitu *method* `Bruteforce(Board board, Piece[] pieces)` yang menyimpan *board* dan *piece* yang diberikan kemudian menghasilkan permutasi unik untuk setiap *piece* yang disimpan pada `permutations` dan melakukan *counting* untuk banyaknya iterasi yang disimpan pada `attempt`.

Kemudian, terdapat *method* `int search(int pieceIndex)` yang digunakan dengan memanggil `search(0)` atau indeks pertama dari *list of piece*. Sesuai abstraksi sebelumnya, di dalam *method* tersebut terdapat iterasi validasi pemasangan *piece* sebanyak M baris dan N kolom yang di dalamnya lagi dilakukan *loop* permutasi unik untuk setiap *pieces*. Apabila tidak dapat dipasang, akan dilakukan *backtracking* (dihapus sementara *piece* sekarang dan kembali ke *piece* sebelumnya) hingga bertemu kasus basis, mengembalikan `true` jika ditemukan solusi atau `false` jika tidak.

3.2. Kelas `Board`

```java
public boolean isPartFit(int i, int j)
{
    return i >= 0 && i < getHeight() && j >= 0 && j < getWidth() && getElement(i, j) == '*';
}

public boolean fitPiece(int boardX, int boardY, Piece piece)
{
    for (int i = 0; i < piece.getHeight(); i++)
    {
        for (int j = 0; j < piece.getWidth(); j++)
        {
            if (piece.getPart(i, j) != '#' && !isPartFit(boardX + i, boardY + j))
            {
                return false;
            }
        }
    }

    for (int i = 0; i < piece.getHeight(); i++)
    {
        for (int j = 0; j < piece.getWidth(); j++)
        {
            char letter = piece.getPart(i, j);
            if (letter != '#')
            {
                setElement(boardX + i, boardY + j, letter);
            }
        }
    }
    return true;
}

public void removePiece(int boardX, int boardY, Piece piece)
{
    for (int i = 0; i < piece.getHeight(); i++)
    {
        for (int j = 0; j < piece.getWidth(); j++)
        {
            if (piece.getPart(i, j) != '#')
            {
                setElement(boardX + i, boardY + j, '*');
            }
        }
    }
}
```

Gambar 6: Implementasi Java dari Kelas `Board`

Gambar 6 hanya memperlihatkan beberapa *method* utama dari Kelas `Board` yang digunakan dalam *core logic* algoritma *Brute force*.

Kemudian, `int fitPiece(int boardX, int boardY, Piece piece` akan melakukan validasi apakah untuk tiap unit sel/*parts* pada suatu *piece* dapat diletakkan dengan unit sel patokan yang diletakkan pada unit sel dengan N adalah `boardX` dan M adalah `boardY`.

Terakhir, terdapat `void removePiece(int boardX, int boardY, Piece piece)` akan menghapus *piece* dengan unit sel patokan yang diletakkan pada unit sel dengan N adalah `boardX` dan M adalah `boardY` dalam implementasi proses *backtracking*.

3.3. Kelas `Piece`

```java
public Piece rotate()
{
    char[][] rotated = new char[getWidth()][getHeight()];

    for (int i = 0; i < getHeight(); i++)
    {
        for (int j = 0; j < getWidth(); j++)
        {
            rotated[j][getHeight() - i - 1] = getPart(i, j);
        }
    }

    return new Piece(getWidth(), getHeight(), rotated, null);
}

public Piece flip()
{
    char[][] flipped = new char[getHeight()][getWidth()];

    for (int i = 0; i < getHeight(); i++)
    {
        for (int j = 0; j < getWidth(); j++)
        {
            flipped[i][j] = getPart(i, getWidth() - j - 1);
        }
    }

    return new Piece(this.N, this.M, flipped, null);
}

public static Piece[][] uniquePermutations(Piece[] Pieces)
{
    Piece[][] result = new Piece[Pieces.length][];

    for (int i = 0; i < Pieces.length; i++)
    {
        Set<Piece> uniqueSet = new HashSet<>();
        Piece current = Pieces[i];

        for (int j = 0; j < 4; j++)
        {
            uniqueSet.add(current);
            uniqueSet.add(current.flip());
            current = current.rotate();
        }

        result[i] = uniqueSet.toArray(new Piece[0]);
    }
    return result;
}
```

Gambar 7: Implementasi Java dari Kelas `Piece`

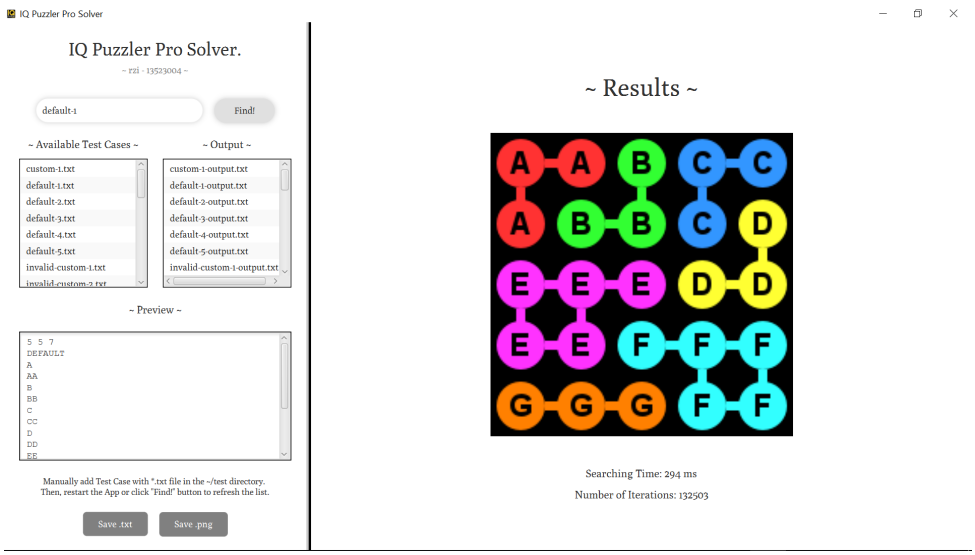
Gambar 7 hanya memperlihatkan beberapa *method* utama dari Kelas `Piece` yang digunakan dalam *core logic* algoritma *Brute force*.

terdapat `Piece[][] uniquePermutations(Piece[] Pieces` akan menghasilkan *list of list* yang berisi permutasi unik dari setiap *piece*. Sesuai abstraksi, hal ini dilakukan dengan melakukan *rotation* dan *flip* hingga total iterasi 8 kali dan hanya menyimpan permutasi uniknya.

Bab IV
Eksperimen

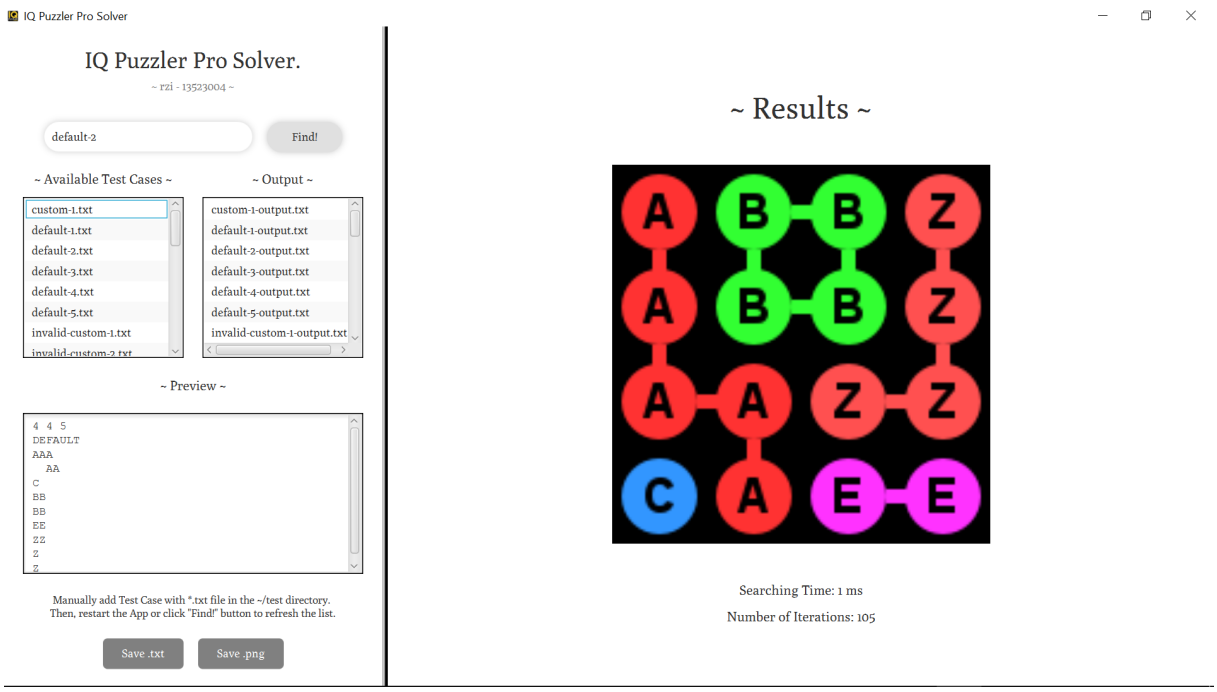
4.1. Default Type

```
Input:
5 5 7
DEFAULT
A
AA
B
BB
C
CC
D
DD
EE
EE
E
FF
FF
F
GGG

Solution:
AABCC
ABBCD
EEEDD
EEFFF
GGGFF

Searching Time: 294ms

Number of Iterations: 132503
```



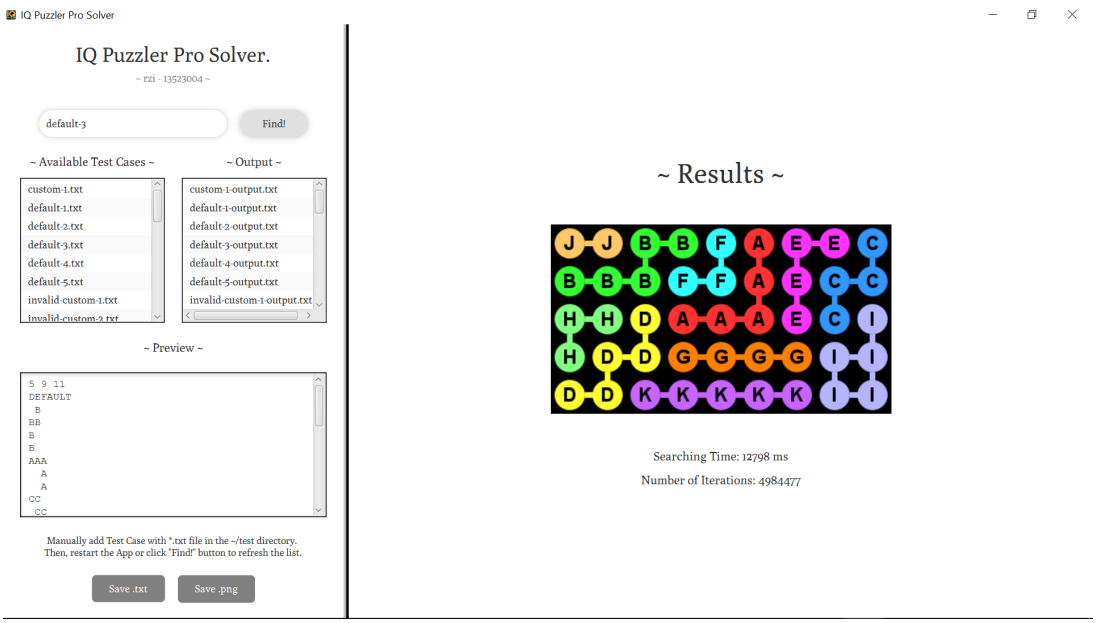
Gambar 8: Kasus Uji `default-1`

```
Input:
4 4 5
DEFAULT
AAA
  AA
C
BB
BB
EE
ZZ
Z
Z

Solution:
ABBZ
ABBZ
AAZZ
CAEE

Searching Time: 1ms

Number of Cases: 105
```



Gambar 9: Kasus Uji `default-2`

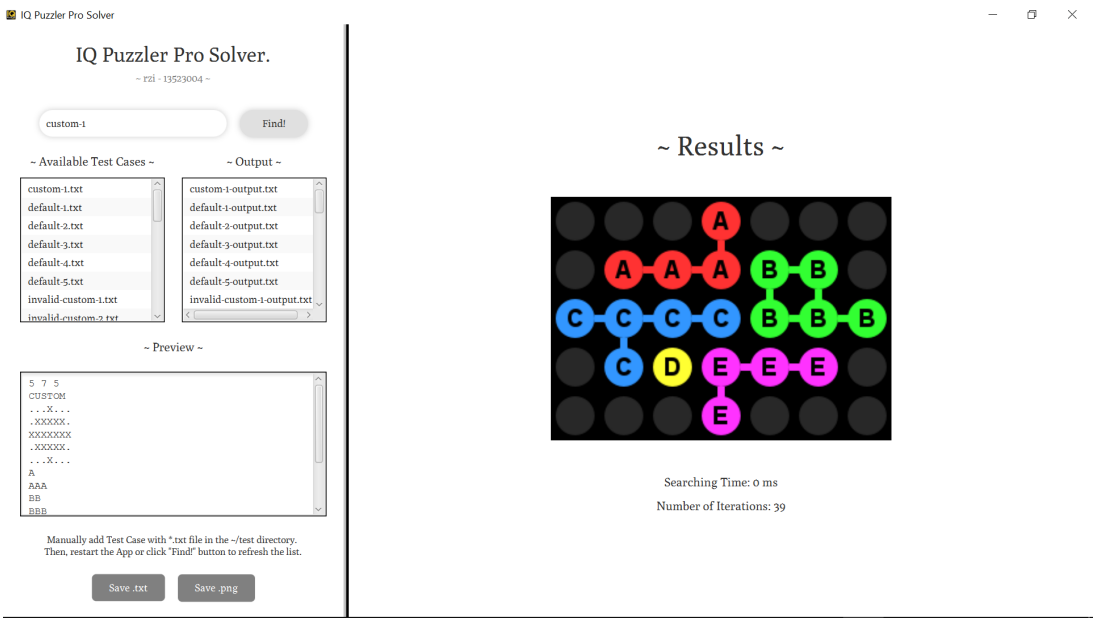
```
Input:
5 9 11
DEFAULT
 B
BB
B
B
AAA
  A
  A
CC
 CC
DD
 DD
  D
EE
E
E
FF
F
G
G
G
G
HH
H
I
II
II
JJ
KKKKK

Solution:
JJBBFAEEC
BBBFFAECC
HHDAAAECI
HDDGGGGII
DDKKKKKII

Searching Time: 12798ms

Number of Iterations: 4984477
```



Gambar 10: Kasus Uji `default-3`

4.2. Custom Type

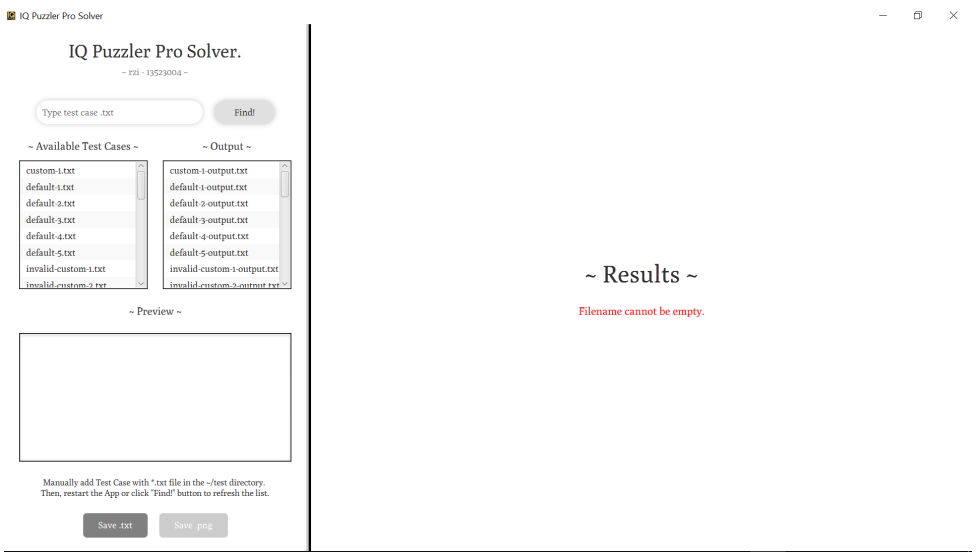
```
Input:
5 7 5
CUSTOM
...X...
.XXXXX.
XXXXXXX
.XXXXX.
...X...
A
AAA
BB
BBB
CCCC
 C
D
EEE
E

Solution:
   A
 AAABB
CCCCBBB
 CDEEE
   E

Searching Time: 0ms

Number of Iterations: 39
```



Gambar 11: Kasus Uji `custom-1`

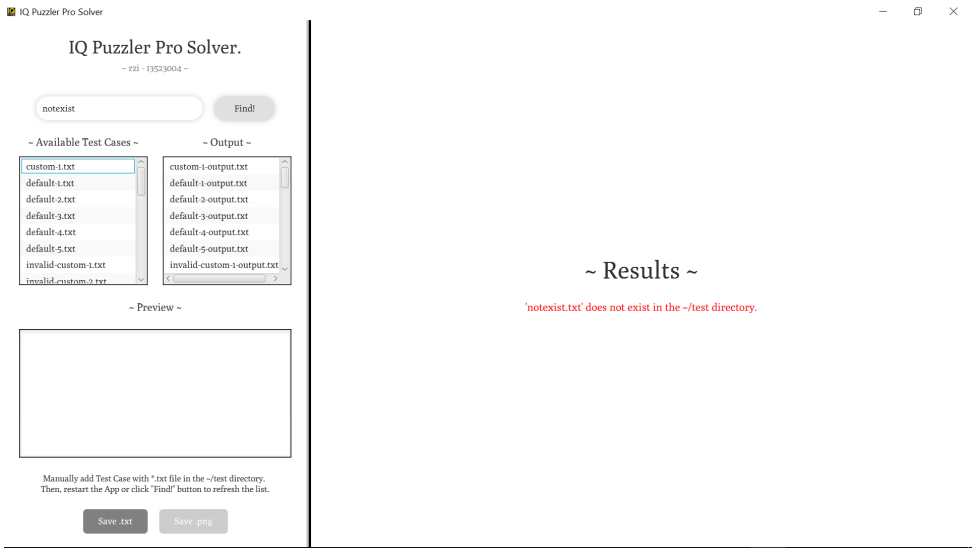
4.3. Invalid

```
Input:
Empty .txt name

Output:
Filename cannot be empty.
```



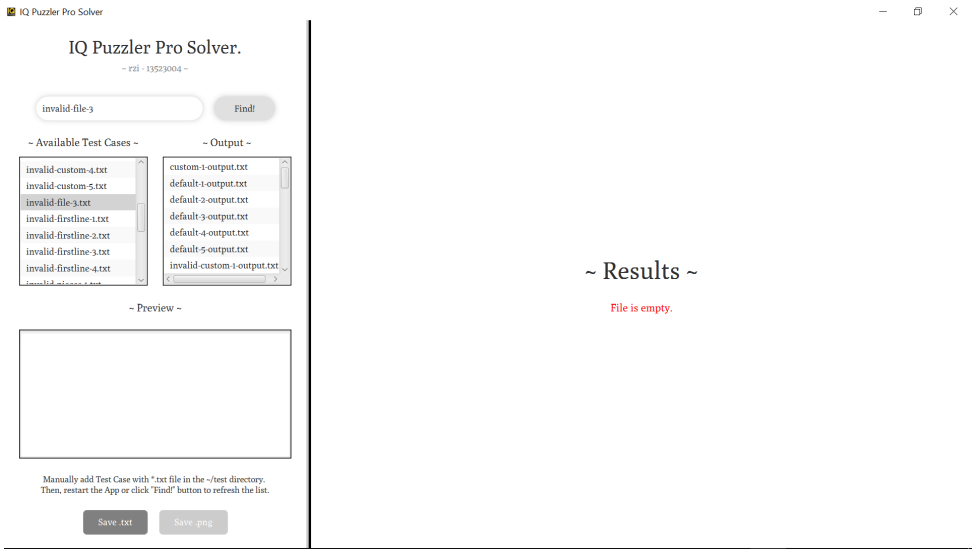
Gambar 12: Kasus Uji `invalid-file-1`

```
Input:
Non-existing filename

Output:
'notexist.txt' does not exist in the ~/test directory.
```



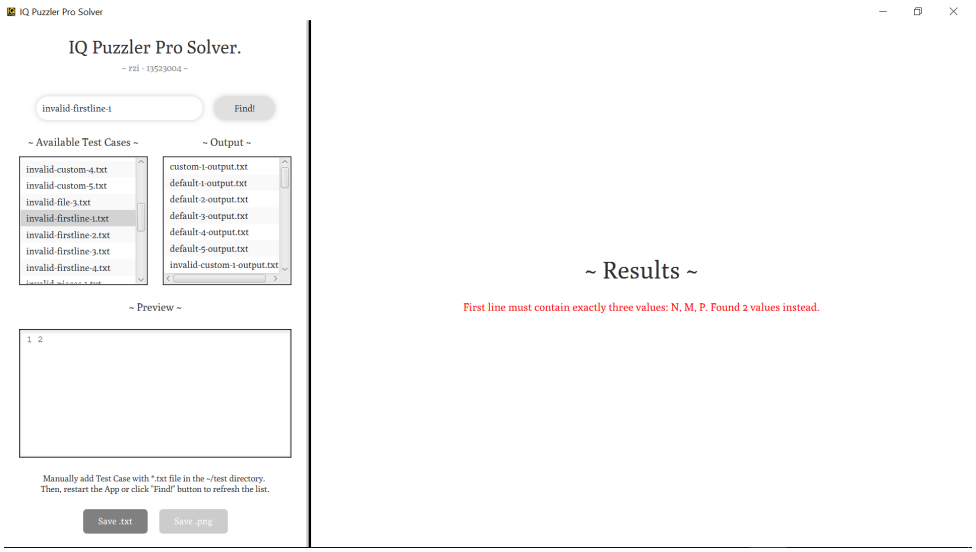
Gambar 13: Kasus Uji `invalid-file-2`

```
Input: (Empty File)


Output:
File is empty.
```



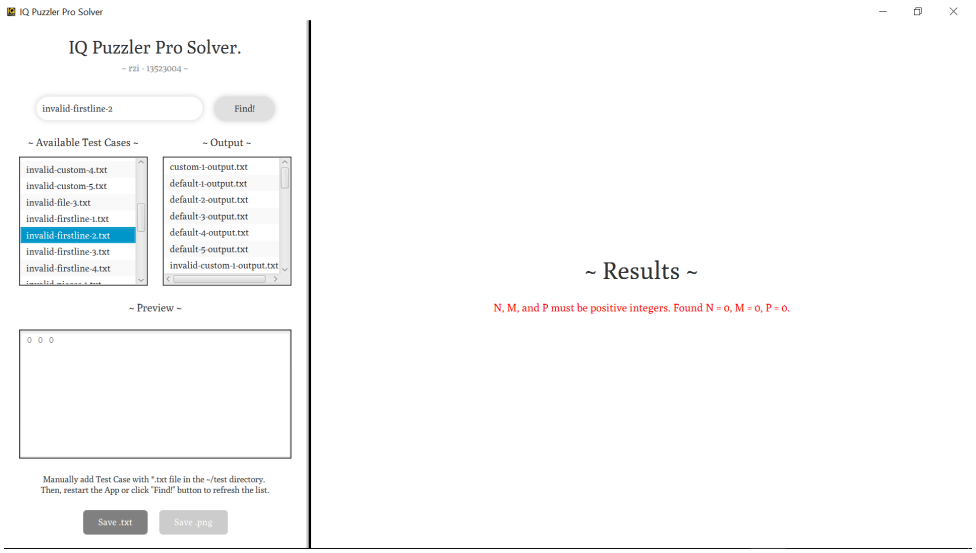
Gambar 14: Kasus Uji `invalid-file-3`

```
Input:
1 2

Output:
First line must contain exactly three values: N, M, P.
Found 2 values instead.
```



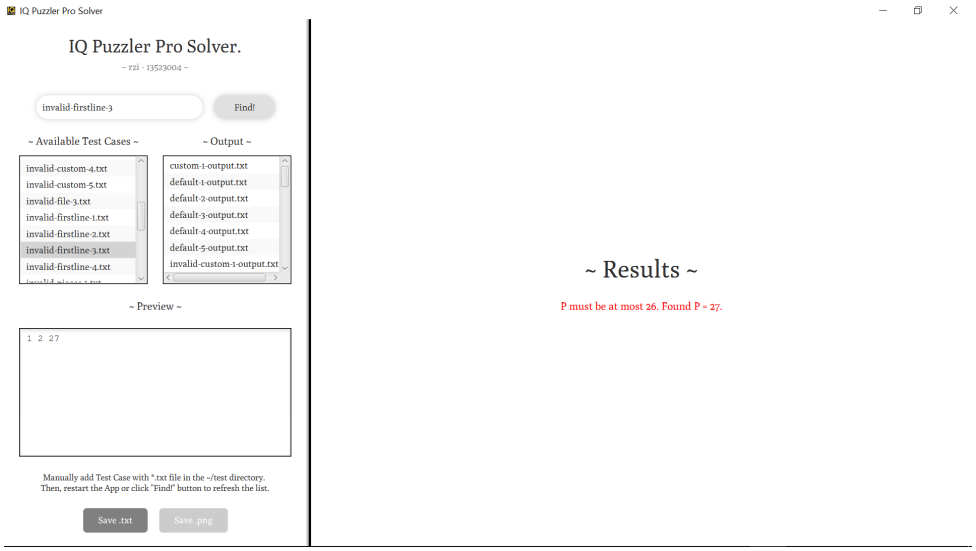
Gambar 15: Kasus Uji `invalid-firstline-1`

```
Input:
0 0 0

Output:
N, M, and P must be positive integers. Found N = 0, M = 0, P = 0.
```



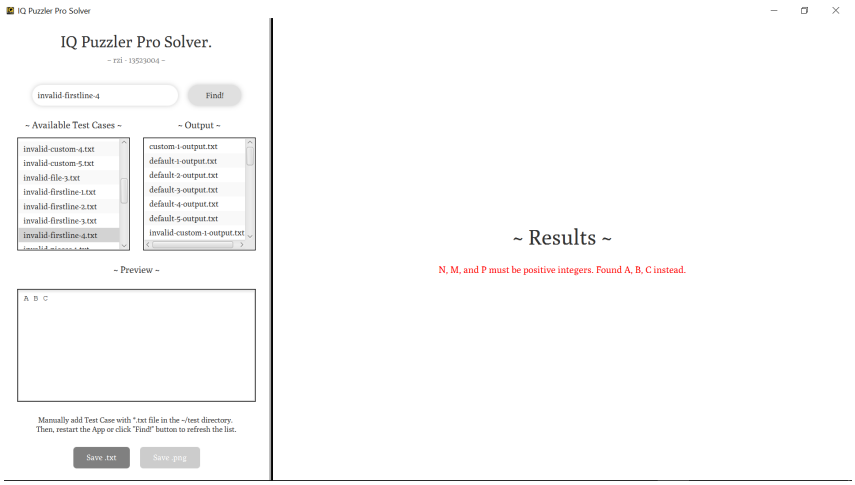
Gambar 16: Kasus Uji `invalid-firstline-2`

```
Input:
1 2 27

Output:
P must be at most 26. Found P = 27.
```



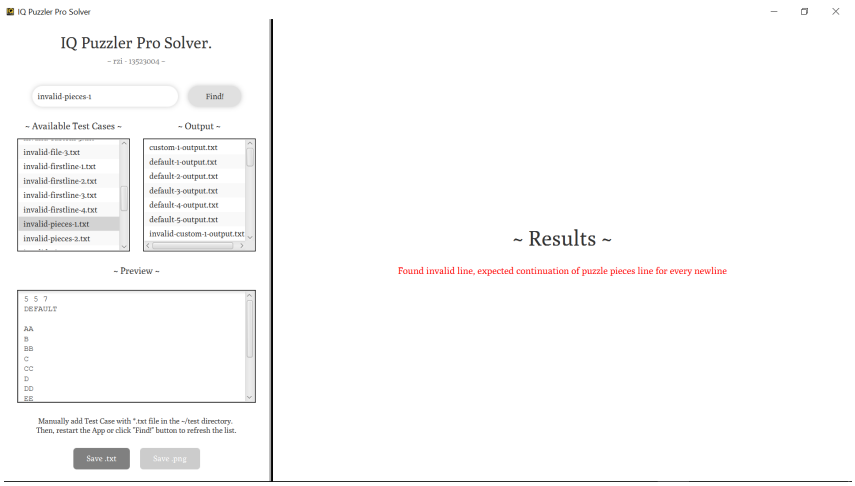
Gambar 17: Kasus Uji `invalid-firstline-3`

```
Input:
A B C

Output:
N, M, and P must be positive integers. Found A, B, C instead.
```



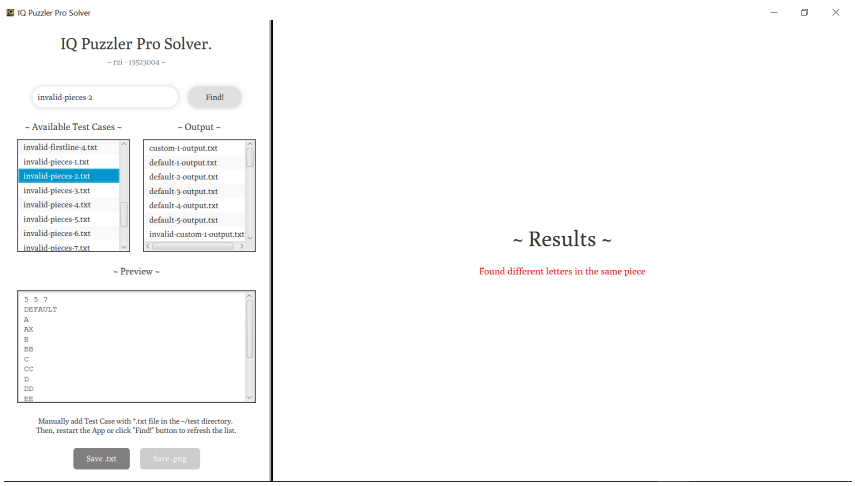
Gambar 18: Kasus Uji `invalid-firstline-4`

```
Input:
5 5 7
DEFAULT

AA

Output:
Found invalid line, expected continuation of puzzle pieces line
for every newline
```



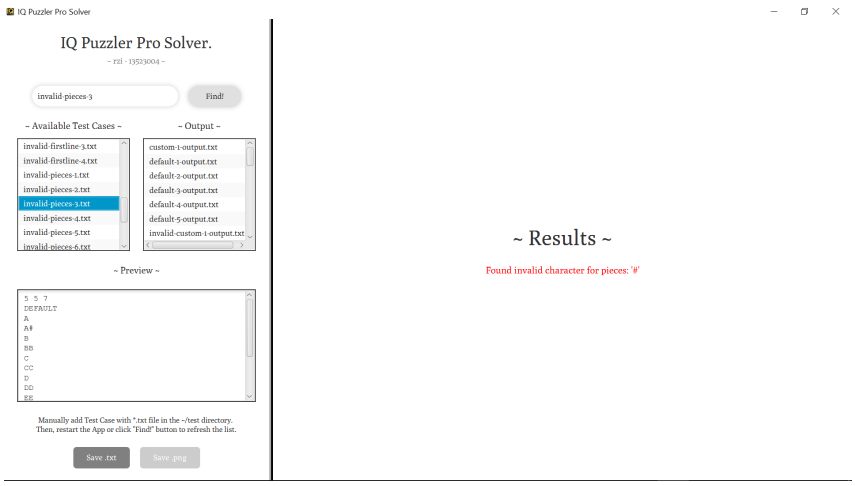
Gambar 19: Kasus Uji `invalid-pieces-1`

```
Input:
5 5 7
DEFAULT
A
AX

Output:
Found different letters in the same piece
```



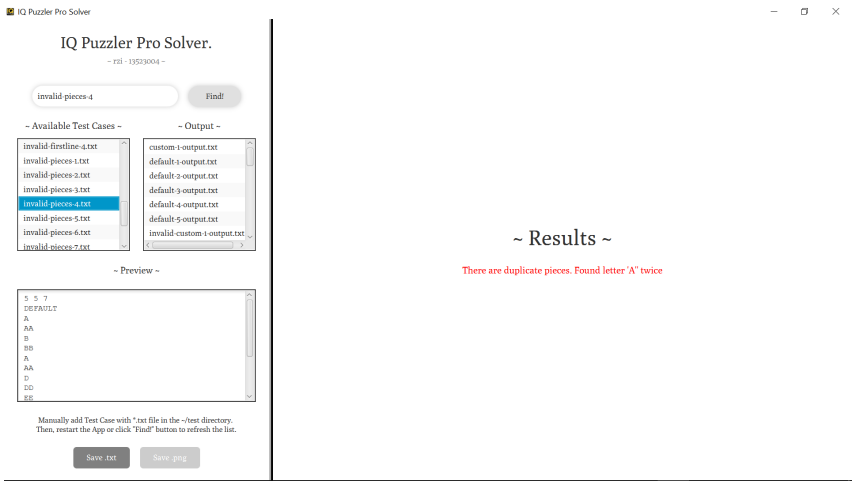
Gambar 20: Kasus Uji `invalid-pieces-2`

```
Input:
5 5 7
DEFAULT
A
A#

Output:
Found invalid character for pieces: '#'
```



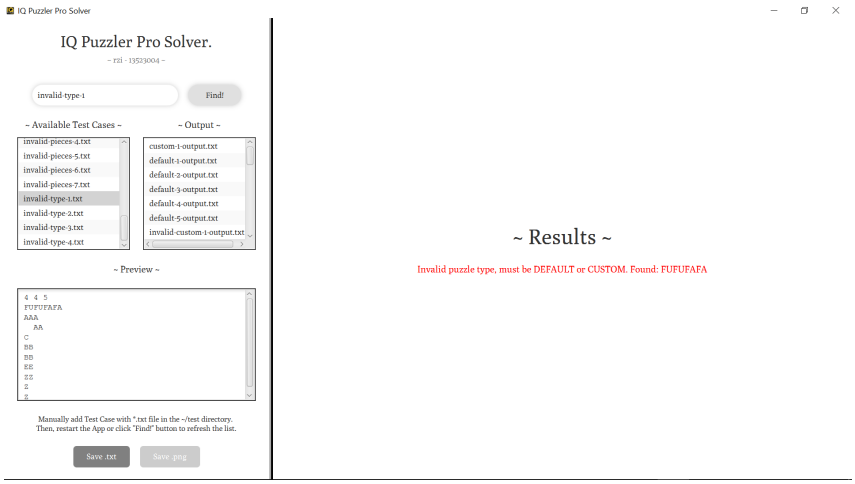
Gambar 21: Kasus Uji `invalid-pieces-3`

```
Input:
5 5 7
DEFAULT
A
AA
B
BB
A
AA

Output:
There are duplicate pieces. Found A twice
```



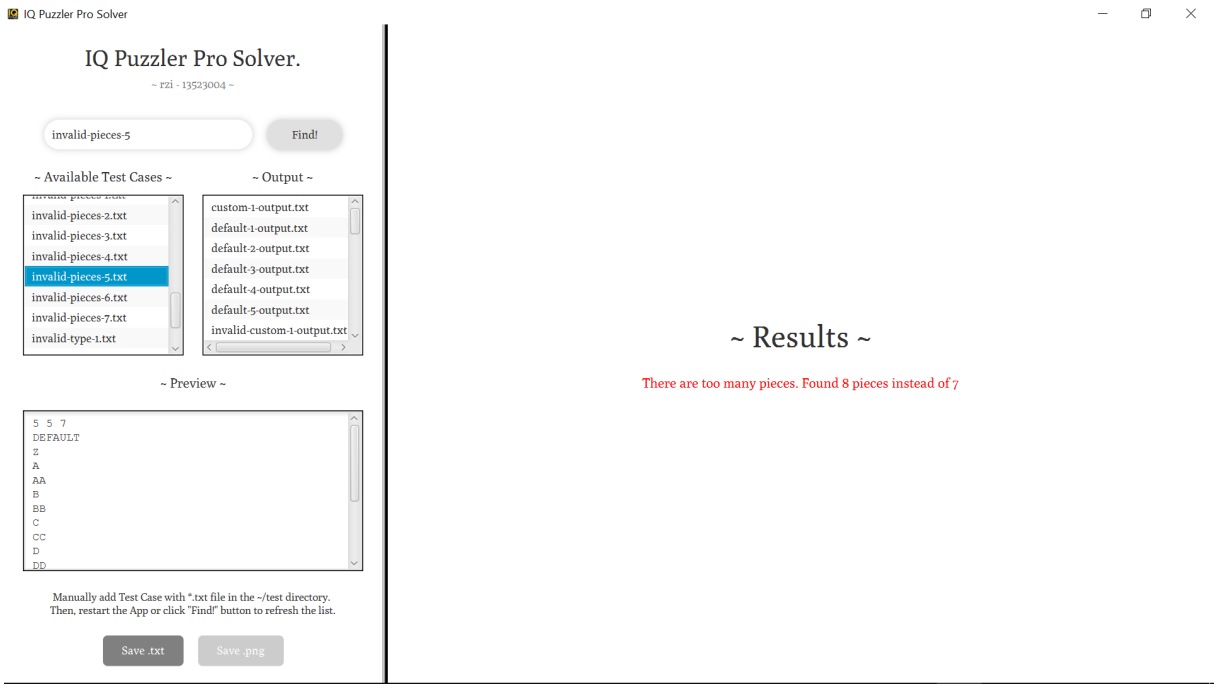
Gambar 22: Kasus Uji `invalid-pieces-4`

```
Input:
4 4 5
FUFUFAFA

Output:
Invalid puzzle type, must be DEFAULT or CUSTOM. Found: FUFUFAFA
```



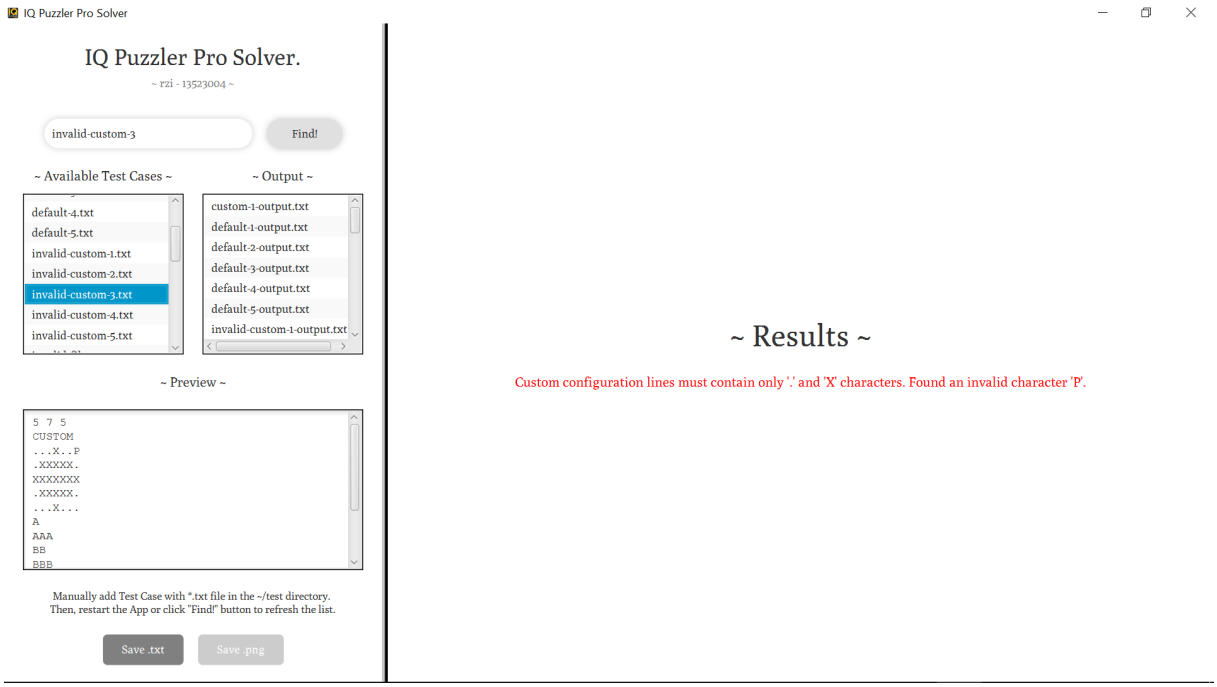
Gambar 23: Kasus Uji `invalid-type-1`

```
Input:
5 5 7
DEFAULT
Z
A
AA
B
BB
C
CC
D
DD
EE
EE
E
FF
FF
F
GGG

Output:
There are too many pieces. Found 8 pieces instead of 7
```



Gambar 24: Kasus Uji `invalid-pieces-5`

```
Input:
5 7 5
CUSTOM
...X..P

Output:
Custom configuration lines must contain only '.' and 'X'
characters. Found an invalid character 'P'.
```



Gambar 25: Kasus Uji `invalid-custom-3`

Bab V
Penutup

5.1. Tautan

Repository program dapat diakses melalui tautan berikut:

`https://github.com/zirachw/Tucil1_13523004`

5.2. Lampiran

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	✔	

Gambar 26: Tabel Spesifikasi Tucil 1

Referensi

Rinaldi Munir. 2025. "Algoritma Brute Force (Bagian 1) (Versi baru 2025)." https://informatika.stei.itb.ac.id/ rinaldi.munir/Stmik/2024-2025/02-Algoritma-Brute-Force-(2025)-Bag1.pdf

Geeks for Geeks. 2024. Brute Force Approach and its pros and cons. Diakses pada 24 Februari 2025. https://www.geeksforgeeks.org/brute-force-approach-and-its-pros-and-cons/