

**LAPORAN TUGAS BESAR II**  
**IF2211 STRATEGI ALGORITMA**  
**Pengaplikasian Algoritma BFS dan DFS dalam Menyelesaikan**  
**Persoalan *Maze Treasure Hunt***



Disusun Oleh :

Farhan Nabil Suryono	13521114
Febryan Arota Hia	13521120
Made Debby Almadea P.	13521153

# **DAFTAR ISI**

<b>DAFTAR ISI</b>	<b>1</b>
<b>BAB I</b>	
<b>DESKRIPSI MASALAH</b>	<b>2</b>
<b>BAB II</b>	
<b>LANDASAN TEORI</b>	<b>7</b>
2.1 Dasar Teori Graph Traversal, BFS, DFS	7
2.1.1 Graph Traversal	7
2.1.2 Breadth-First Traversal (BFS)	7
2.1.3 DFS	8
2.2 C# Desktop Application Development	9
<b>BAB III</b>	
<b>APLIKASI ALGORITMA BFS DAN DFS</b>	<b>10</b>
3.1 Langkah-langkah Pemecahan Masalah	10
3.2 Proses Mapping Persoalan Menjadi Elemen BFS dan DFS	10
3.3 Ilustrasi Kasus	10
<b>BAB IV</b>	
<b>ANALISIS PEMECAHAN MASALAH</b>	<b>11</b>
4.1 Implementasi Program (Pseudocode Program Utama)	11
4.2 Penjelasan Struktur Data	16
4.2.1 Queue	16
4.2.2 Stack	16
4.2.3 List	16
4.2.4 HashSet	16
4.2.5 Dictionary	17
4.3 Tata Cara Penggunaan Program	17
4.4 Hasil Pengujian	23
4.5 Analisis Desain Solusi	23
<b>BAB V</b>	
<b>KESIMPULAN DAN SARAN</b>	<b>24</b>
5.1 Kesimpulan	24
5.2 Saran	24
5.3 Refleksi	24
<b>DAFTAR PUSTAKA</b>	<b>25</b>
<b>LAMPIRAN</b>	<b>26</b>

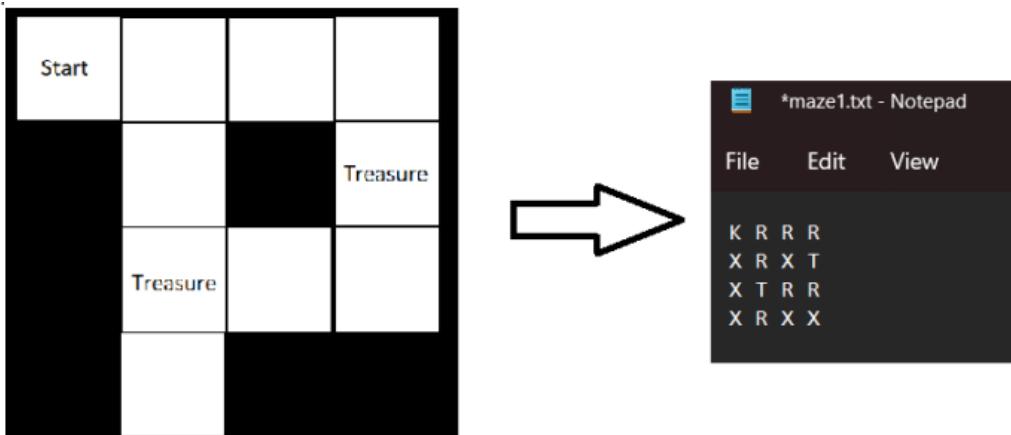
# BAB I

## DESKRIPSI MASALAH

Dalam tugas besar ini, Anda akan diminta untuk membangun sebuah aplikasi dengan GUI sederhana yang dapat mengimplementasikan BFS dan DFS untuk mendapatkan rute memperoleh seluruh treasure atau harta karun yang ada. Program dapat menerima dan membaca input sebuah file txt yang berisi maze yang akan ditemukan solusi rute mendapatkan treasure-nya. Untuk mempermudah, batasan dari input maze cukup berbentuk segi-empat dengan spesifikasi simbol sebagai berikut :

- K : Krusty Krab (Titik awal)
- T : Treasure
- R : Grid yang mungkin diakses / sebuah lintasan
- X : Grid halangan yang tidak dapat diakses

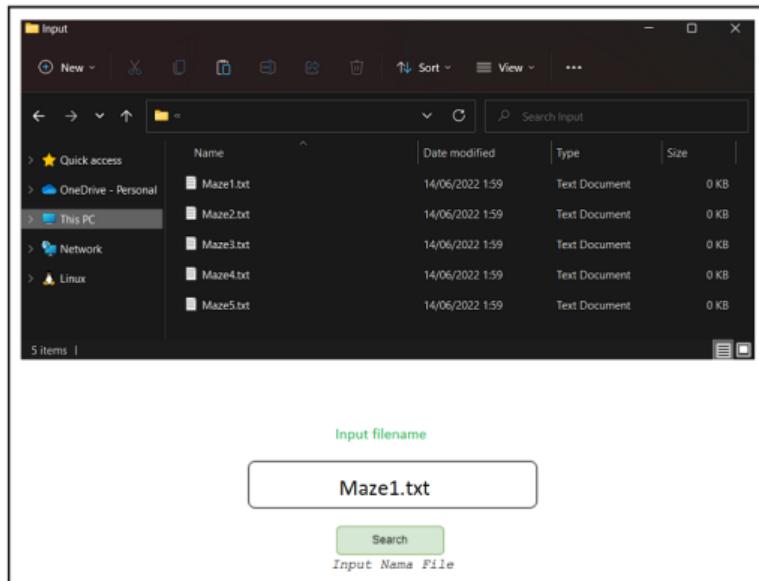
Contoh file input :



Gambar 1.1 Ilustrasi Input File Maze

Dengan memanfaatkan algoritma Breadth First Search (BFS) dan Depth First Search (DFS), anda dapat menelusuri grid (simpul) yang mungkin dikunjungi hingga ditemukan rute solusi, baik secara melebar ataupun mendalam bergantung alternatif algoritma yang dipilih. Rute solusi adalah rute yang memperoleh seluruh treasure pada maze. Perhatikan bahwa rute yang diperoleh dengan algoritma BFS dan DFS dapat berbeda, dan banyak langkah yang dibutuhkan pun menjadi berbeda. Prioritas arah simpul yang dibangkitkan dibebaskan asalkan ditulis di laporan ataupun readme, semisal LRUD (left right up down). Tidak ada pergerakan secara diagonal. Anda juga diminta untuk memvisualisasikan input txt tersebut menjadi suatu grid maze serta hasil pencarian rute solusinya. Cara visualisasi grid dibebaskan, sebagai contoh dalam bentuk matriks yang ditampilkan dalam GUI dengan keterangan berupa teks atau warna. Pemilihan warna dan maknanya dibebaskan ke masing - masing kelompok, asalkan dijelaskan di readme / laporan

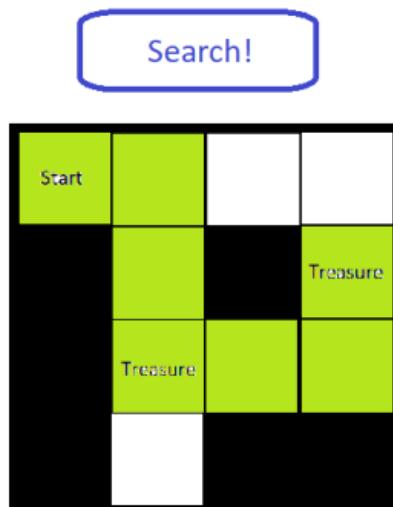
Contoh input aplikasi :



Gambar 1.2 Contoh input program

Daftar input maze akan dikemas dalam sebuah folder yang dinamakan test dan terkandung dalam repository program. Folder tersebut akan setara kedudukannya dengan folder src dan doc (struktur folder repository akan dijelaskan lebih lanjut di bagian bawah spesifikasi tubes). Cara input maze boleh langsung input file atau dengan textfield sehingga pengguna dapat mengetik nama maze yang diinginkan. Apabila dengan textfield, harus menghandle kasus apabila tidak ditemukan dengan nama file tersebut.

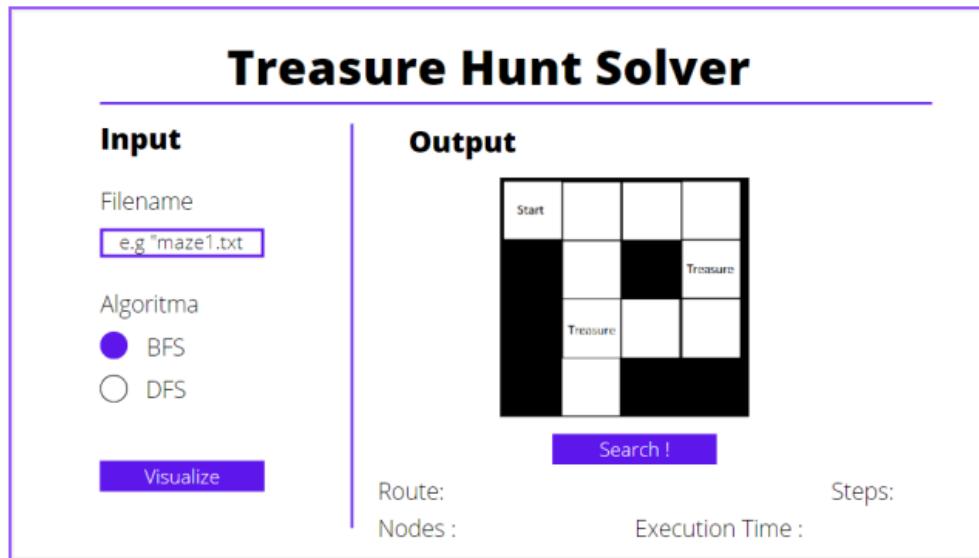
Contoh output Aplikasi :



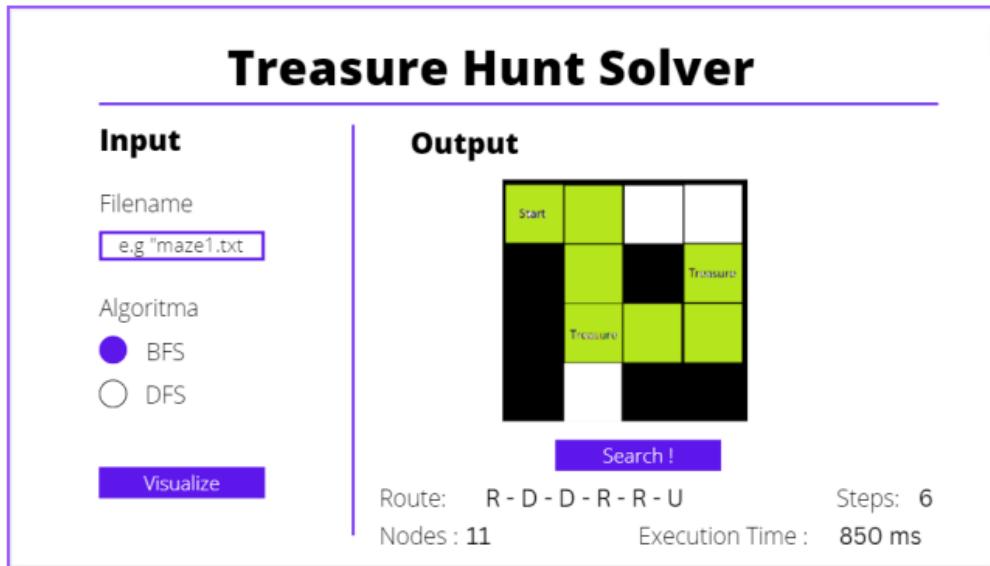
Gambar 3 . Contoh output program untuk gambar 1

Setelah program melakukan pembacaan input, program akan memvisualisasikan gridnya terlebih dahulu tanpa pemberian rute solusi. Hal tersebut dilakukan agar pengguna dapat mengerjakan terlebih dahulu treasure hunt secara manual jika diinginkan. Kemudian, program menyediakan tombol solve untuk mengeksekusi algoritma DFS dan BFS. Setelah tombol diklik, program akan melakukan pemberian warna pada rute solusi.

Spesifikasi Program: Aplikasi yang akan dibangun dibuat berbasis GUI. Berikut ini adalah contoh tampilan dari aplikasi GUI yang akan dibangun



Gambar 1.4 Tampilan Program Sebelum dicari solusinya



Gambar 1.5 Tampilan Program setelah dicari solusinya

Catatan: Tampilan diatas hanya berupa contoh layout dari aplikasi saja, untuk design layout aplikasi dibebaskan dengan syarat mengandung seluruh input dan output yang terdapat pada spesifikasi.

Spesifikasi GUI:

1. Masukan program adalah file maze treasure hunt tersebut atau nama filenya.
2. Program dapat menampilkan visualisasi dari input file maze dalam bentuk grid dan pewarnaan sesuai deskripsi tugas.
3. Program memiliki toggle untuk menggunakan alternatif algoritma BFS ataupun DFS.
4. Program memiliki tombol search yang dapat mengeksekusi pencarian rute dengan algoritma yang bersesuaian, kemudian memberikan warna kepada rute solusi output.
5. Luaran program adalah banyaknya node (grid) yang diperiksa, banyaknya langkah, rute solusinya, dan waktu eksekusi algoritma.
6. (Bonus) Program dapat menampilkan progress pencarian grid dengan algoritma yang bersesuaian. Hal tersebut dilakukan dengan memberikan slider / input box untuk menerima durasi jeda tiap step, kemudian memberikan warna kuning untuk tiap grid yang sudah diperiksa dan biru untuk grid yang sedang diperiksa.
7. (Bonus) Program membuat toggle tambahan untuk persoalan TSP. Jadi apabila toggle dinyalakan, rute solusi yang diperoleh juga harus kembali ke titik awal setelah menemukan segala harta karunnya (Tetap dengan algoritma BFS atau DFS).
8. GUI dapat dibuat sekreatif mungkin asalkan memuat 5 (7 jika mengerjakan bonus) spesifikasi di atas.

Program yang dibuat harus memenuhi spesifikasi wajib sebagai berikut:

1. Buatlah program dalam bahasa C# untuk mengimplementasi Treasure Hunt Solver sehingga diperoleh output yang diinginkan. Penelusuran harus memanfaatkan algoritma BFS dan DFS.
2. Awalnya program menerima file atau nama file maze treasure hunt.
3. Apabila filename tersebut ada, Program akan melakukan validasi dari file input tersebut. Validasi dilakukan dengan memeriksa apakah tiap komponen input hanya berupa K, T, R, X. Apabila validasi gagal, program akan memunculkan pesan bahwa file tidak valid. Apabila validasi berhasil, program akan menampilkan visualisasi awal dari maze treasure hunt.
4. Pengguna memilih algoritma yang digunakan menggunakan toggle yang tersedia.
5. Program kemudian dapat menampilkan visualisasi akhir dari maze (dengan pewarnaan rute solusi).
6. Program menampilkan luaran berupa durasi eksekusi, rute solusi, banyaknya langkah, serta banyaknya node yang diperiksa. Proses visualisasi ini boleh memanfaatkan pustaka atau kakas yang tersedia.

7. Mahasiswa tidak diperkenankan untuk melihat atau menyalin library lain yang mungkin tersedia bebas terkait dengan pemanfaatan BFS dan DFS. Akan tetapi, untuk algoritma lain diperbolehkan menggunakan library jika ada.

## BAB II

# LANDASAN TEORI

### 2.1 Dasar Teori Graph Traversal, BFS, DFS

#### 2.1.1 Graph Traversal

Graph traversal adalah proses mengunjungi setiap simpul (node) atau sisi (edge) dalam sebuah graf (graph). Tujuan dari graf traversal adalah untuk menemukan atau memeriksa semua simpul atau sisi dalam graf. Ada dua jenis graf traversal, yaitu depth-first traversal (DFS) dan breadth-first traversal (BFS).

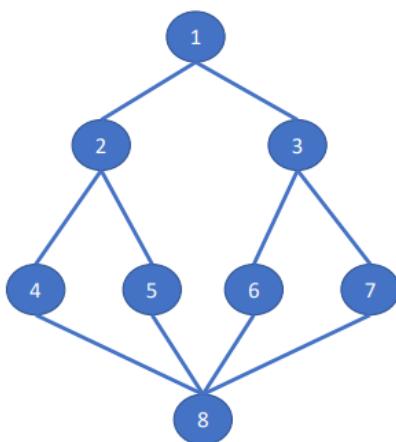
#### 2.1.2 Breadth-First Traversal (BFS)

Breadth-first traversal atau pencarian melebar adalah algoritma pencarian pada graf dimana pencarian dilakukan dengan mengunjungi simpul tetangga terlebih dahulu sebelum simpul anaknya. Pencarian BFS ini menggunakan struktur data *queue* dengan prinsip FIFO (*First In First Out*).

Algoritma:

1. Kunjungi simpul  $v$  (simpul awal)
2. Kunjungi semua simpul yang bertetangga dengan simpul  $v$
3. Kunjungi simpul yang belum dikunjungi dan bertetangga dengan simpul-simpul yang tadi dikunjungi, demikian seterusnya

Ilustrasi:



Iterasi	v	Q	dikunjungi							
			1	2	3	4	5	6	7	8
Inisialisasi	1	{1}	T	F	F	F	F	F	F	F
Iterasi 1	1	{2,3}	T	T	T	F	F	F	F	F
Iterasi 2	2	{3,4,5}	T	T	T	T	T	F	F	F
Iterasi 3	3	{4,5,6,7}	T	T	T	T	T	T	T	F
Iterasi 4	4	{5,6,7,8}	T	T	T	T	T	T	T	T
Iterasi 5	5	{6,7,8}	T	T	T	T	T	T	T	T
Iterasi 6	6	{7,8}	T	T	T	T	T	T	T	T
Iterasi 7	7	{8}	T	T	T	T	T	T	T	T
Iterasi 8	8	{}	T	T	T	T	T	T	T	T

Urutan simpul yang dikunjungi: 1, 2, 3, 4, 5, 6, 7, 8

Gambar 2.1 Ilustrasi BFS

Sumber (<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf>)

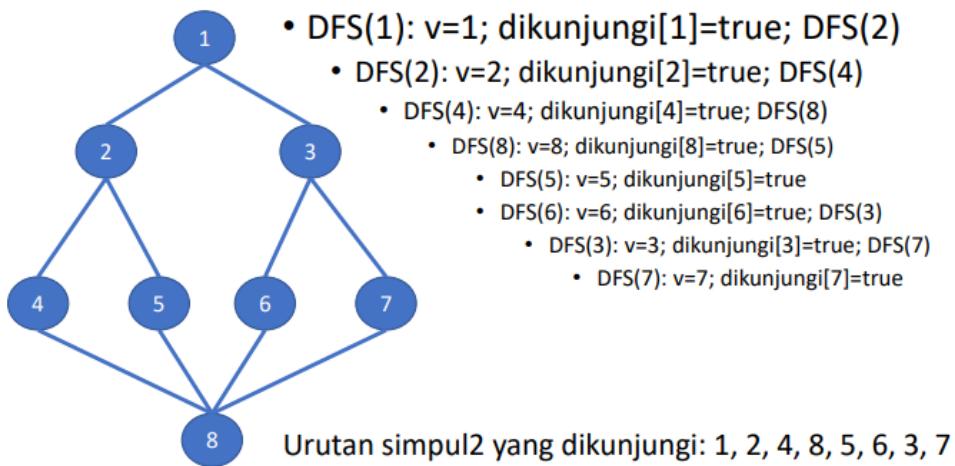
### 2.1.3 DFS

Depth-first search atau pencarian mendalam adalah algoritma pencarian pada graf dimana pencarian dilakukan dengan mengunjungi semua simpul anak hingga level terdalamnya terlebih dahulu baru mengunjungi simpul di sampingnya. Pencarian DFS ini menggunakan struktur data *stack* dengan prinsip LIFO (*Last In First Out*).

Algoritma:

1. Kunjungi simpul  $v$
2. Kunjungi simpul  $w$  yang bertetangga dengan simpul  $v$ .
3. Ulangi DFS mulai dari simpul  $w$
4. Ketika mencapai simpul  $u$  sedemikian sehingga semua simpul yang bertetangga dengannya telah dikunjungi, pencarian diruntut-balik (*backtrack*) ke simpul terakhir yang dikunjungi sebelumnya dan mempunyai simpul  $w$  yang belum dikunjungi.
5. Pencarian berakhir bila tidak ada lagi simpul yang belum dikunjungi yang dapat dicapai dari simpul yang telah dikunjungi.

Ilustrasi:



Gambar 2.2 Ilustrasi DFS

Sumber (<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf>)

## **2.2 C# Desktop Application Development**

C# Desktop Application Development adalah proses pembuatan aplikasi desktop yang menggunakan bahasa pemrograman C#. C# adalah bahasa pemrograman yang dikembangkan oleh Microsoft dan dirilis pada tahun 2000. Bahasa pemrograman ini sangat populer digunakan untuk pengembangan aplikasi desktop karena kemampuannya dalam memanipulasi data dan kinerja yang cepat. Dalam pengembangan aplikasi desktop menggunakan C#, terdapat beberapa tool yang dapat digunakan seperti Visual Studio, Xamarin, dan Windows Forms.

Pengembangan aplikasi desktop menggunakan C# memiliki kelebihan dalam hal kemampuan untuk memanipulasi data dan kinerja yang cepat. Selain itu, aplikasi desktop yang dibuat menggunakan C# juga dapat diintegrasikan dengan aplikasi lain seperti database dan web services. Hal ini memungkinkan pengembang untuk membuat aplikasi yang lebih kompleks dan fungsional. Selain itu, karena C# adalah bahasa pemrograman yang terus berkembang dan diperbarui oleh Microsoft, maka aplikasi desktop yang dibuat menggunakan C# juga memiliki keamanan yang lebih baik dan lebih mudah untuk diperbarui.

## BAB III

# APLIKASI ALGORITMA BFS DAN DFS

### 3.1 Langkah-langkah Pemecahan Masalah

Pemecahan permasalahan tugas besar kali ini dilakukan dengan menggunakan algoritma BFS dan DFS. Langkah-langkah yang dilakukan adalah sebagai berikut.

1. Membuat class Maze dan class Tile dengan atribut yang diperlukan seperti id, kategori, dsb.
2. Membuat fungsi untuk membaca *maze* dari input file yang dibentuk dari petak-petak (*tiles*) dengan id dan kategori yang sesuai.
3. Membuat fungsi pencarian yang menerapkan algoritma BFS dan DFS. Kedua fungsi tersebut akan digunakan untuk menelusuri *path* pada *maze* untuk menemukan *treasures*.
4. Membuat fungsi untuk menangani visualisasi dalam proses pencarian baik menggunakan algoritma BFS maupun algoritma DFS.
5. Membuat GUI untuk menampilkan visualisasi.

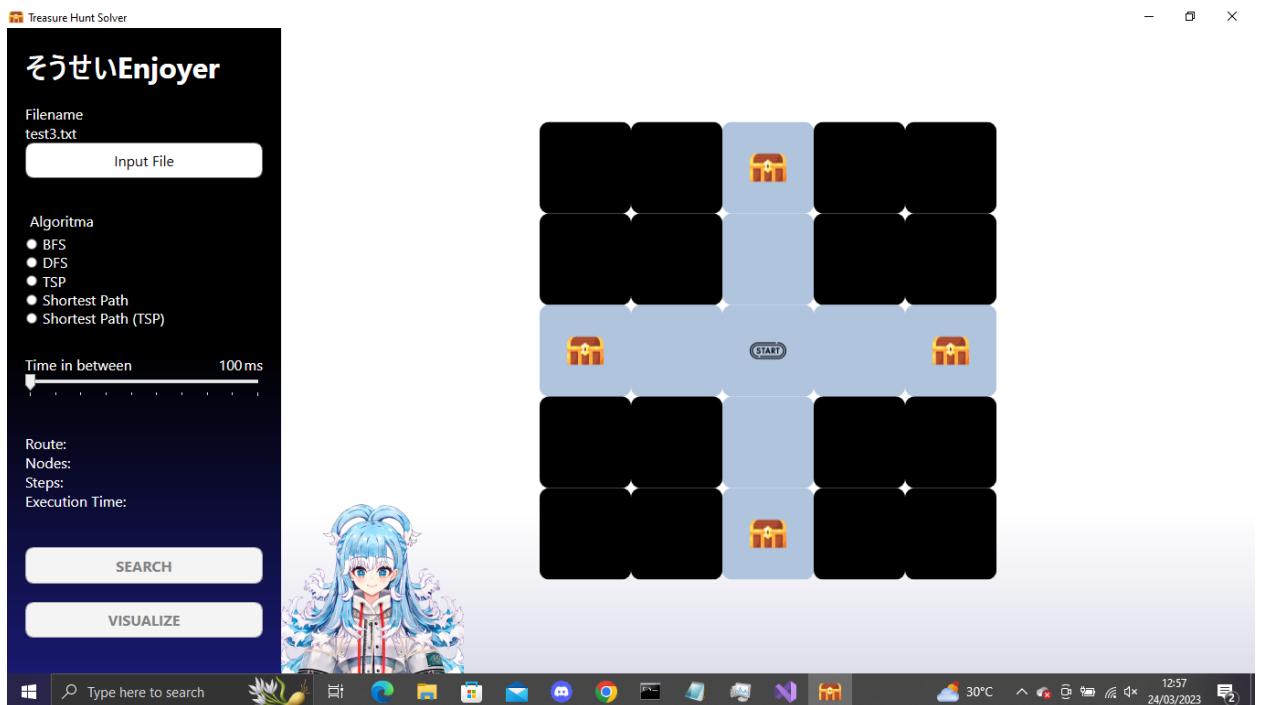
### 3.2 Proses *Mapping* Persoalan Menjadi Elemen BFS dan DFS

Dalam pencarian BFS program akan memulai pencarian dari *starting tile* lalu memeriksa *tile* tetangganya (yang dapat dilalui) dengan prioritas *up-left-right-down*. Proses pencarian dilakukan sesuai dengan teori algoritma BFS yang dijelaskan pada landasan teori. Lalu apabila di tengah pencarian ditemukan *tile* yang merupakan *treasure*, program akan mengulangi pencarian kembali dengan menggunakan *tile* tersebut sebagai *starting tile* barunya. Proses ini dilakukan terus menerus hingga tidak ada lagi *treasure* yang ditemukan.

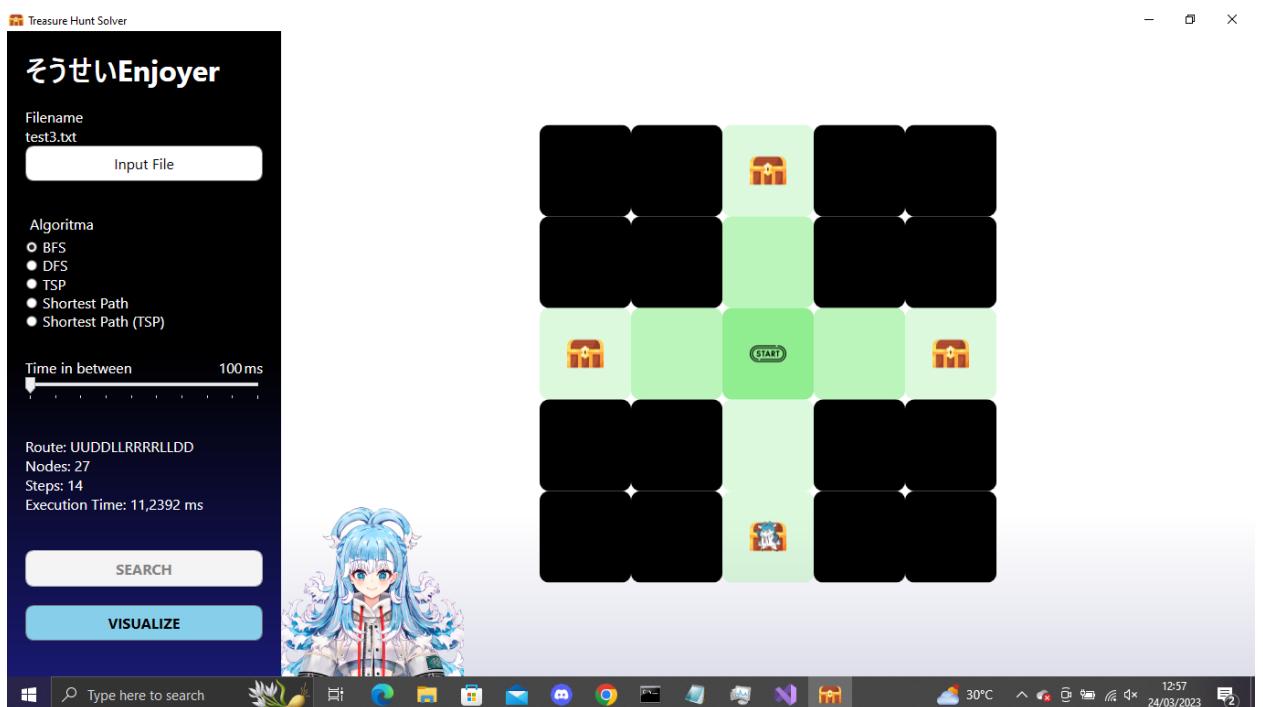
Sedangkan dalam pencarian DFS, program akan melakukan pencarian ke salah satu pilihan jalur hingga jalur tersebut tidak dapat dilanjutkan lagi, baru kemudian program akan melakukan pencarian ke jalur lain. Dalam DFS, program juga memiliki prioritas yang sama yaitu *up-left-right-down*. Program akan langsung berhenti apabila semua *treasure* telah dikunjungi.

### 3.3 Ilustrasi Kasus

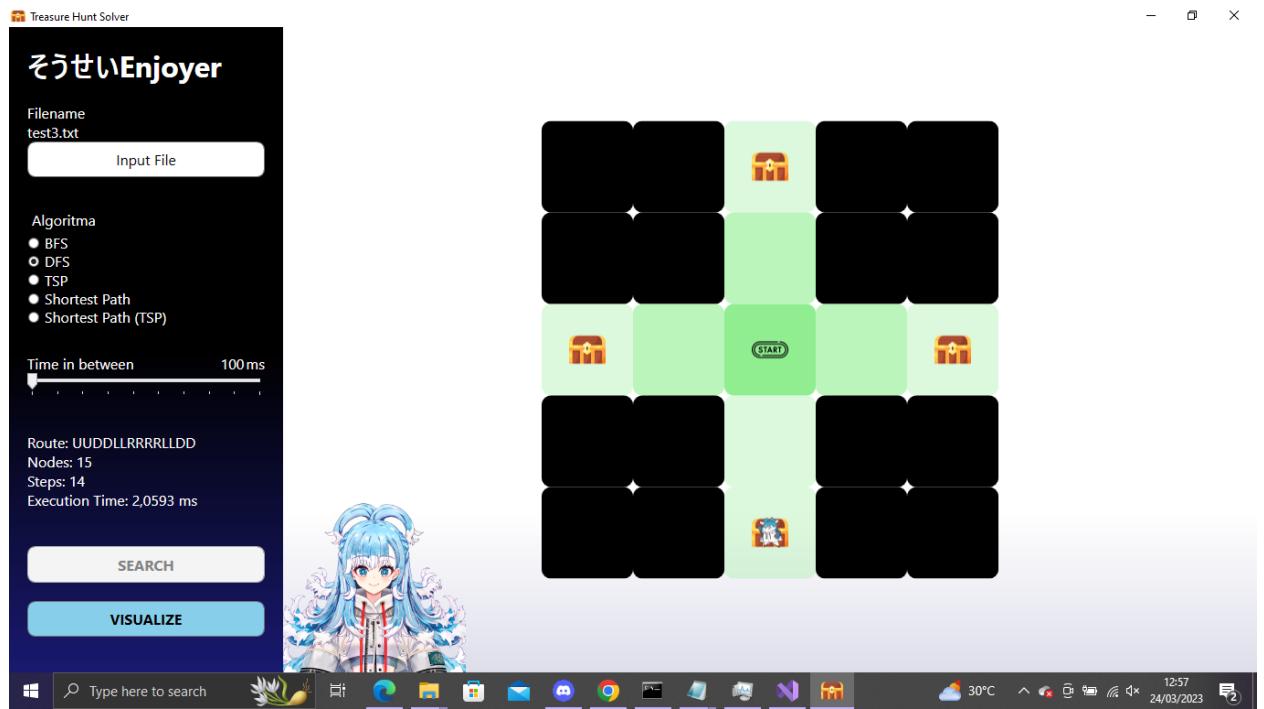
#### 3.3.1. Ilustrasi kasus 1: *treasure* berada di ujung



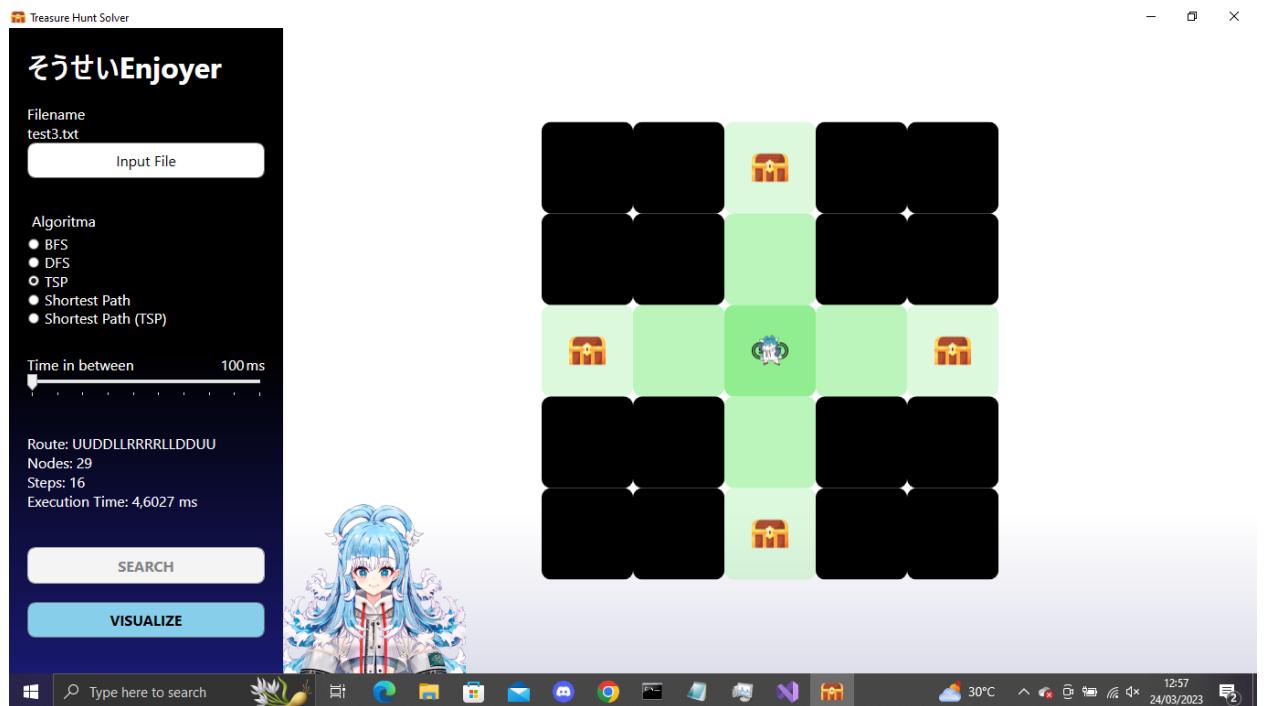
Gambar 3.3.1.1 Ilustrasi Kasus 1



Gambar 3.3.1.2 Ilustrasi Kasus 1 BFS

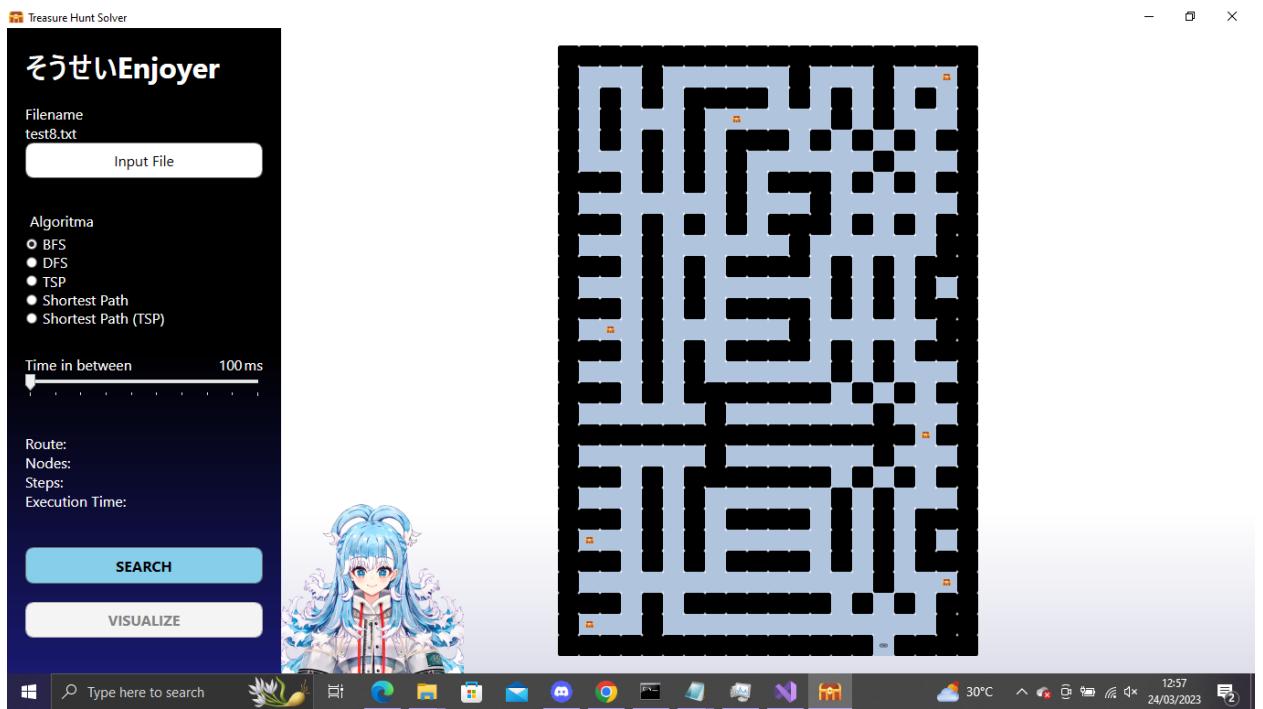


Gambar 3.3.1.3 Ilustrasi Kasus 1 DFS

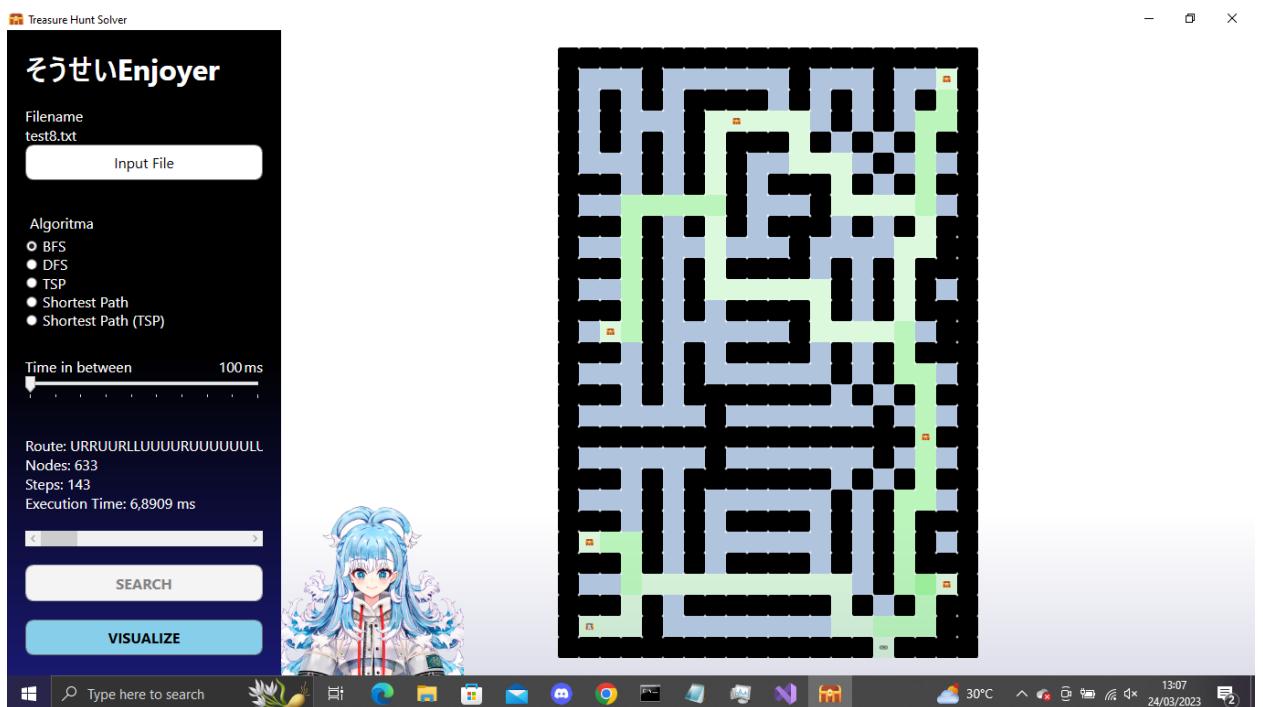


Gambar 3.3.1.4 Ilustrasi Kasus 1 TSP

### 3.3.2. Ilustrasi kasus 2: *maze* berukuran besar dan *treasure* tersebar



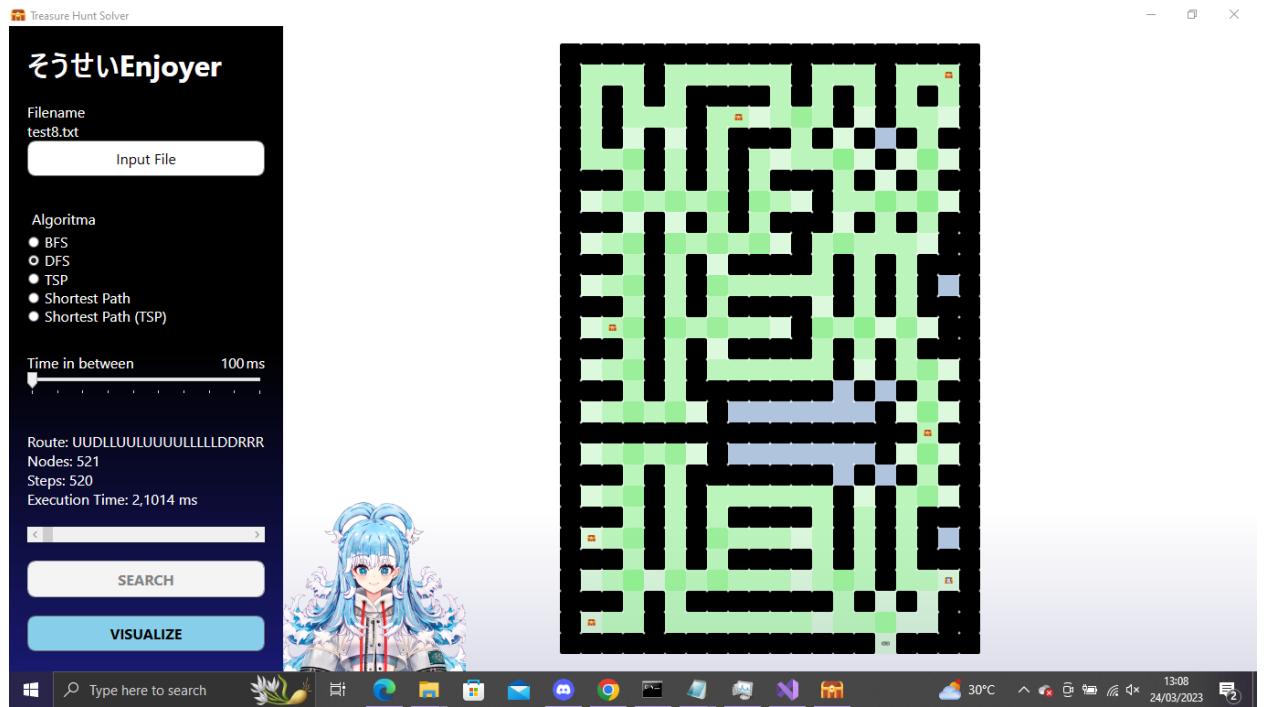
Gambar 3.3.2.1 Ilustrasi Kasus 2



Gambar 3.3.2.2 Ilustrasi Kasus 2 BFS

Rute hasil pencarian BFS:

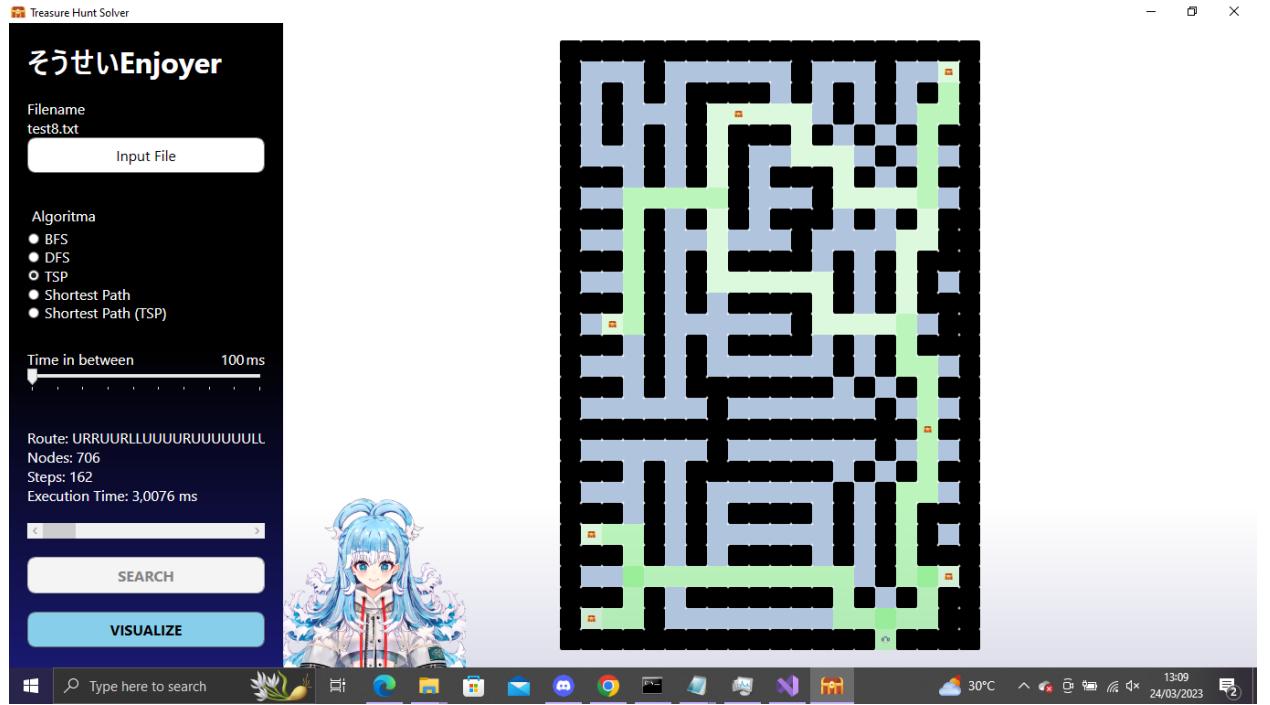
URRUURLLUUUURUUUUULUUUUURUUUUURUUDLDDDLLLUULL  
UULLLLDDDDLLLDLDDDLRUUUUUURRRRDDDRRRRDRDDDRDD  
DDDDLDDDDRDLLLUULLLLLLLUULLRRDDDLL



Gambar 3.3.2.3 Ilustrasi Kasus 2 DFS

Rute hasil pencarian DFS:

UUDLLUULUUULLLDDRRRLLLDDLLUUUUUULLLRRDDLLRRDDLL  
RRDDLLRRRLDDLLRRUUUUUUURRLDDDDDDDRRRRRRLLLLNUUR  
RRRRRLLLUUUUURRRRDDDRRUUUUDDDDLLDRRRRUULUUURUUUU  
ULUUUUUULUUUDLLUUUDLLUUULUULLLDDLUULLDDDRRUUDDDLLR  
RRRUUUDDDRRUUUURRLDDDDLLUDDDDRRUUUDRRRRRUURUDRD  
DDDLUDDDLNUUDRRRRRUURRLDDUUUUUULLDDLLNNNDDRRRLLL  
LDDDDLLUUUUUUUUUDLLRRDDLLRRDDLLRRDDLLRRRRRLUUUU  
UUUURRRRUUUURLDDRRLDDRLLUULLLUUULLUUURRDRDUUURRRR  
RDDRRUURRDDUULLDDDRRLDDRRRUUUULUURRDDUULLDDRDDL  
RLDDRLDDUULLDDRDDDRRLDDRLRRDDRLDDRLDDDRRLDDDRRL



Gambar 3.3.2.4 Ilustrasi Kasus 2 TSP

Rute hasil pencarian TSP:

URRUURLLUUUURUUUUULUUUUUURUUUUUURUUDLDDDDLLLULUULLLULDDDDDRDDRRRRDDRRRDRDD  
UULLLLDDDDLLLULDDDDDLRUUUUUURRRRDRDDDRRRRRDDRRRDRDD  
DDDDLDDDDDRDDLLLULUULLLLLLLUULLRRDDDLRRUURRRRRRRR  
RDDRRD

# BAB IV

## ANALISIS PEMECAHAN MASALAH

### 4.1 Implementasi Program (Pseudocode Program Utama)

```
Function BFS (startingTile : Tile) -> lastTreasure : Tile
{ Pencarian seluruh treasure dengan algoritma BFS, mengembalikan Tile
treasure terakhir }

Algoritma
{ Inisialisasi seluruh atribut }
{ BFSRoute berisi string pergerakan akhir }
{ processRoute berisi urutan list of tile yang diperiksa }
{ visitedTreasure berisi Tile of treasure yang telah ditemukan }
{ temp sebagai temporary value dari BFSfind }
{ finish berisi boolean penanda ada/tidak ada lagi treasure yang belum
ditemukan }

BFSRoute <- ""
processRoute <- new List
visitedTreasure <- new HashSet
temp <- new Tuple
finish <- false

{ Inisialisasi Tile awal }
start <- GetStartingTile()

{ Mencari semua treasure }
while (not finish) do
    { Mencari treasure dari titik start menggunakan metode BFS }
    temp <- BFSfind(start)

    if (temp[1] = "") then { tidak ada lagi treasure yang ditemukan }
        finish <- true
    else
        { Cari treasure selanjutnya menggunakan BFS dengan titik treasure
        sebelumnya sebagai startingTile baru }
        start <- temp[0]
endwhile

-> start

Function BFSfind(startingTile : Tile) -> Tuple<Tile, string>
{ Mengembalikan tuple berisi Tile treasure yang ditemukan beserta string
route hasil pencarian menggunakan BFS }
```

**Algoritma**

```
{ Inisialisasi }
path <- new List
processRoute.Add(startingTile)
```

```

{ Queue untuk pemrosesan menggunakan konsep FIFO }
q <- new Queue
{ Enqueue tuple starting tile }
q.Enqueue(createTuple(startingTile, ""))

{ Inisiasi semua Tile dengan boolean false sebagai penanda Tile belum
dikunjungi }
visited <- new Dictionary<Tile, bool>
for each rowTile in MazeLayout
    for each tile in row Tile
        visited.Add(tile, false)

visited[startingTile] = true
while (GetSize(q) > 0) do
    Tuple <Tile, string> currentTile = Dequeue(q)
    Int x, int y = GetTileCoordinate(currentTile[0])
    path.Add(currentTile[0])

    while (q.Count > 0)
        currentTile <- Dequeue(q)
        int x, int y = GetTileCoordinate(currentTile[0])
        path.Add(currentTile[0])

        if (currentTile[0].IsTreasure() and not
            visitedTreasure.Contains(currentTile [0])) then
            visitedTreasure.Add(currentTile[0])
            path.RemoveAt(path.Count -1)
            path.RemoveAt(0)
            processRoute = processRoute.Concat(path)
            BFSRoute = BFSRoute + currentTile[1]
            -> currentTile

        if (IsIndexValid(x-1, y) and isWalkable() and !visited then
            newTile <- MazeLayout[x-1][y]
            visited[newTile] <- true
            q.Enqueue(CreateTuple(newTile, currentTile[1]+"U"))

        if (IsIndexValid(x, y-1) and isWalkable() and !visited then
            newTile <- MazeLayout[x][y-1]
            visited[newTile] <- true
            q.Enqueue(CreateTuple(newTile, currentTile[1]+"L"))

        if (IsIndexValid(x, y+1) and isWalkable() and !visited then
            newTile <- MazeLayout[x][y+1]
            visited[newTile] <- true
            q.Enqueue(CreateTuple(newTile, currentTile[1]+"R"))

        if (IsIndexValid(x+1, y) and isWalkable() and !visited then
            newTile <- MazeLayout[x+1][y]
            visited[newTile] <- true
            q.Enqueue(CreateTuple(newTile, currentTile[1]+"D"))

    endwhile
endwhile

Temp <- CreateTuple(new Tile, "")
-> temp

```

```

function DFS() -> List<Tile>
{ Kamus }
dfsStack : Stack of Tile
prevTile : Dictionary of Tile to Tile
visited : Dictionary of Tile to bool
visitedTreasure, row, col, cnt : integer
MazeLayout : List of List of Tile
retNode : List of Tile
currentTile : Tile

{ Algoritma }
dfsStack <- new Stack<Tile>
prevTile <- new Dictionary<Tile, Tile>
visited <- new Dictionary<Tile, bool>
visitedTreasure <- 0

retNode <- new List<Tile>()

for each (tileList in MazeLayout)
  for each (tile in tileList)
    prevTile[tile] <- null
    visited[tile] <- false
  end for
end for

dfsStack.Push(GetStartingTile())

while (dfsStack.Count > 0) do
  currentTile <- dfsStack.Pop()
  if (visited[currentTile]) then continue

  visited[currentTile] <- true
  retNode.Add(currentTile)
  if (currentTile.IsTreasure()) then
    visitedTreasure <- visitedTreasure + 1
  endif
  (row, col) <- GetTileCoordinate(currentTile)

  cnt <- 0
  if (IsIndexValid(row + 1, col) and MazeLayout[row + 1][col].IsWalkable())
  then
    if not (visited[MazeLayout[row + 1][col]]) then
      dfsStack.Push(MazeLayout[row + 1][col])
      prevTile[MazeLayout[row + 1][col]] <- currentTile
      cnt <- cnt + 1
    endif
  endif

  if (IsIndexValid(row, col + 1) and MazeLayout[row][col + 1].IsWalkable())
  then
    if not (visited[MazeLayout[row][col + 1]]) then
      dfsStack.Push(MazeLayout[row][col + 1])
      prevTile[MazeLayout[row][col + 1]] <- currentTile
      cnt <- cnt + 1
    endif
  endif

```

```

        endif
    endif

    if (IsIndexValid(row, col - 1) and MazeLayout[row][col - 1].IsWalkable())
then
    if not (visited[MazeLayout[row][col - 1]]) then
        dfsStack.Push(MazeLayout[row][col - 1])
        prevTile[MazeLayout[row][col - 1]] <- currentTile
        cnt <- cnt + 1
    endif
endif

    if (IsIndexValid(row - 1, col) and MazeLayout[row - 1][col].IsWalkable())
then
    if not (visited[MazeLayout[row - 1][col]]) then
        dfsStack.Push(MazeLayout[row - 1][col])
        prevTile[MazeLayout[row - 1][col]] <- currentTile
        cnt <- cnt + 1
    endif
endif

    if (treasureCount = visitedTreasure) then break

    if (cnt = 0 and dfsStack.Count > 0) then
        foundTreasure = false
        while not (currentTile = prevTile[dfsStack.Peek()]) do
            currentTile <- prevTile[currentTile]
            retNode.Add(currentTile)
        endwhile
    endif
endwhile

dfsStack.Clear()

->retNode

procedure TSP (input startingTile : Tile)
{ Algoritma TSP menggunakan BFS }

Algoritma
lastTreasure <- BFS(startingTile)
TSPRoute <- BFSRoute
TSPprocessRoute <- processRoute
TSPfind(lastTreasure)

Function BFSfind(startingTile : Tile) -> Tuple<Tile, string>
{ Mengembalikan tuple berisi Tile treasure yang ditemukan beserta string
route hasil pencarian TSP }

Algoritma
{ Inisialisasi }
path <- new List
processRoute.Add(startingTile)

{ Queue untuk pemrosesan menggunakan konsep FIFO }
q <- new Queue

```

```

{ Enqueue tuple starting tile }
q.Enqueue(createTuple(startingTile, ""))

{ Inisiasi semua Tile dengan boolean false sebagai penanda Tile belum
dikunjungi }
visited <- new Dictionary<Tile, bool>
for each rowTile in MazeLayout
    for each tile in row Tile
        visited.Add(tile, false)

visited[startingTile] = true
while (GetSize(q) > 0) do
    Tuple <Tile, string> currentTile = Dequeue(q)
    Int x, int y = GetTileCoordinate(currentTile[0])
    path.Add(currentTile[0])

    while (q.Count > 0)
        currentTile <- Dequeue(q)
        int x, int y = GetTileCoordinate(currentTile[0])
        path.Add(currentTile[0])

        if (currentTile[0].IsStartingPoint()) then
            path.RemoveAt(path.Count -1)
            path.RemoveAt(0)
            TSPprocessRoute = TSPprocessRoute.Concat (path)
            TSPRoute = TSPRoute + currentTile[1]
            -> currentTile

        if (IsIndexValid(x-1, y) and isWalkable() and !visited then
            newTile <- MazeLayout[x-1][y]
            visited[newTile] <- true
            q.Enqueue(CreateTuple(newTile, currentTile[1]+"U"))

        if (IsIndexValid(x, y-1) and isWalkable() and !visited then
            newTile <- MazeLayout[x][y-1]
            visited[newTile] <- true
            q.Enqueue(CreateTuple(newTile, currentTile[1]+"L"))

        if (IsIndexValid(x, y+1) and isWalkable() and !visited then
            newTile <- MazeLayout[x][y+1]
            visited[newTile] <- true
            q.Enqueue(CreateTuple(newTile, currentTile[1]+"R"))

        if (IsIndexValid(x+1, y) and isWalkable() and !visited then
            newTile <- MazeLayout[x+1][y]
            visited[newTile] <- true
            q.Enqueue(CreateTuple(newTile, currentTile[1]+"D"))

    endwhile
endwhile

Temp <- CreateTuple(new Tile, "")
-> temp

```

## 4.2 Penjelasan Struktur Data

### 4.2.1 Queue

Implementasi struktur data Queue menggunakan *collection* pada kelas System.Collections.Generic. Queue digunakan untuk menyimpan antrian *tile* yang akan diperiksa oleh program saat melakukan traversal menggunakan metode BFS. Berikut merupakan *method* dari Queue yang digunakan oleh program.

- a. Enqueue, berfungsi untuk memasukkan elemen ke dalam *queue* dengan aturan FIFO
- b. Dequeue, berfungsi untuk mengeluarkan elemen dari *queue* dengan aturan FIFO

### 4.2.2 Stack

Implementasi struktur data Stack menggunakan *collection* pada kelas System.Collections.Generic. Stack digunakan untuk menyimpan tumpukan *tile* yang akan diperiksa oleh program saat melakukan traversal menggunakan metode DFS. Berikut merupakan *method* dari Stack yang digunakan oleh program.

- a. Push, berfungsi untuk memasukkan elemen ke dalam tumpukan dengan aturan LIFO
- b. Pop, berfungsi untuk mengeluarkan elemen dari tumpukan dengan aturan LIFO

### 4.2.3 List

Implementasi struktur data List menggunakan *collection* pada kelas System.Collections.Generic. Struktur data List banyak digunakan oleh program. Salah satunya digunakan dalam menyimpan urutan *tile* dalam proses pencarian untuk digunakan sebagai urutan dalam visualisasi. Berikut merupakan *method* dari List yang digunakan oleh program.

- a. Add, berfungsi untuk menambahkan elemen pada list pada urutan paling belakang
- b. RemoveAt, berfungsi untuk menghapus elemen pada list di index tertentu
- c. Foreach, iterasi semua elemen list

### 4.2.4 HashSet

Implementasi struktur data HashSet menggunakan *collection* pada kelas System.Collections.Generic. HashSet menyimpan kumpulan elemen yang unik. Dalam program, HashSet digunakan untuk menyimpan *treasure* yang sudah ditemukan sebagai informasi pada iterasi selanjutnya apakah *treasure* yang ditemukan sudah pernah ditemukan sebelumnya atau belum.

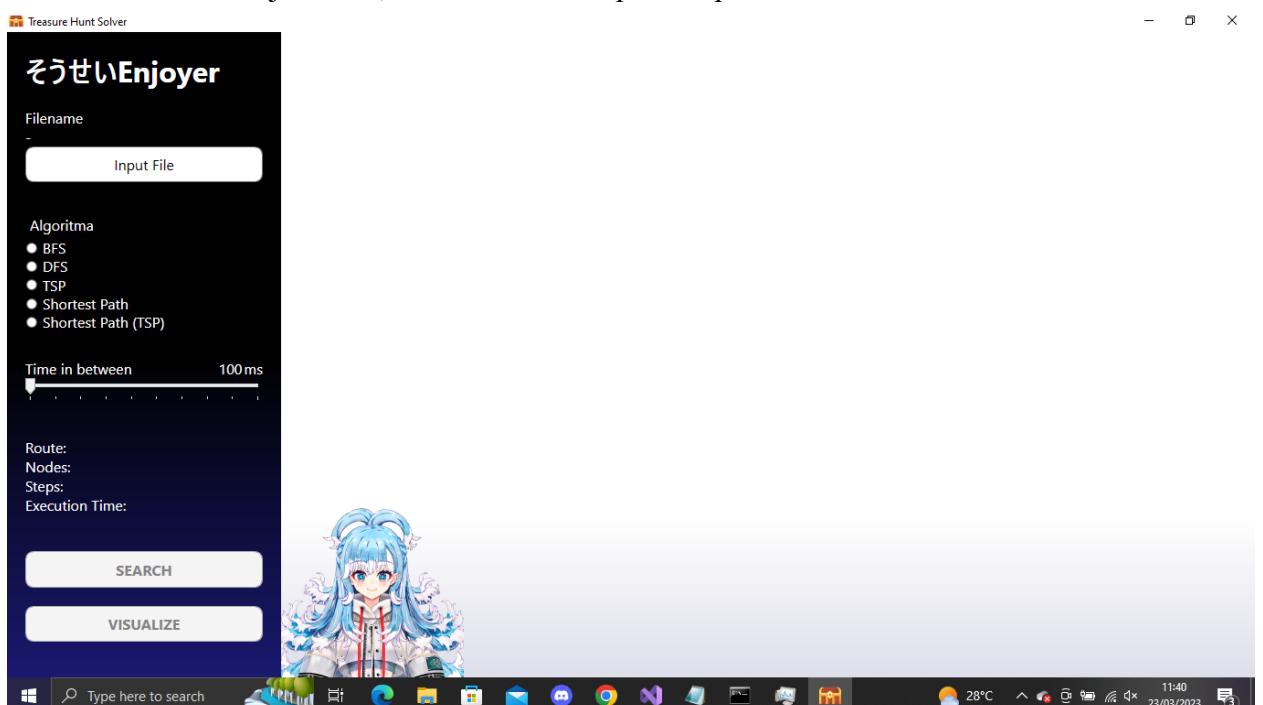
### 4.2.5 Dictionary

Dictionary merupakan koleksi dari beberapa nilai yang bersifat *un-unique* atau *unique* dimana nilai tersebut digunakan untuk mengakses value yang sesuai. Setiap value memiliki

key yang berbeda. Struktur data ini digunakan di dalam program sebagai penanda *visited tile* dalam proses pencarian.

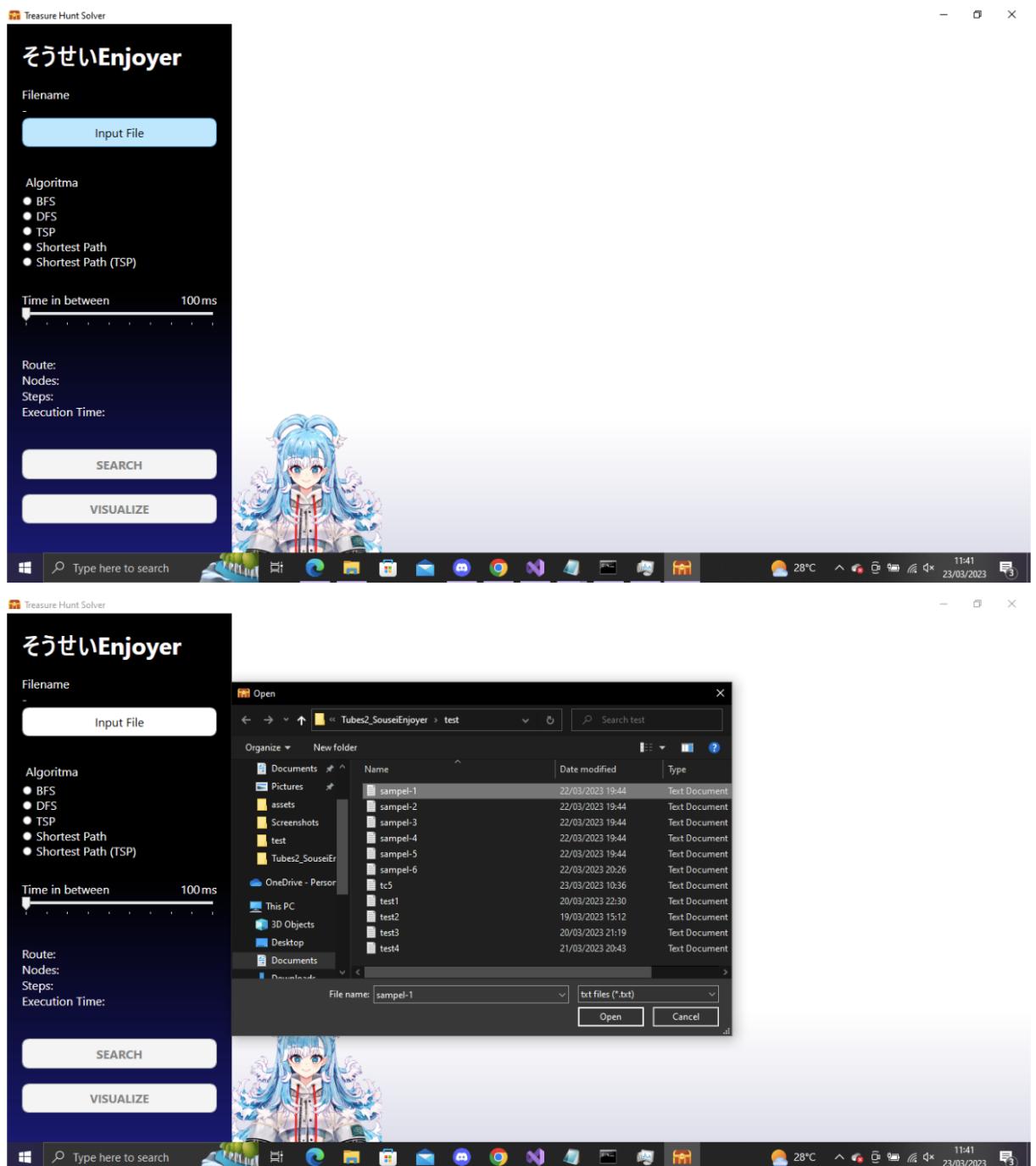
### 4.3 Tata Cara Penggunaan Program

1. Sebelum dapat membangun dan menjalankan program ini, diperlukan beberapa alat berikut:
  - Windows 10 atau yang lebih baru
  - .NET 6.0 SDK
  - Visual Studio atau IDE lain yang mendukung pengembangan .NET.
2. Setelah mendownload program atau melakukan *build*, jalankan *executable file* dalam folder *bin*. Setelah dijalankan, akan muncul tampilan seperti berikut



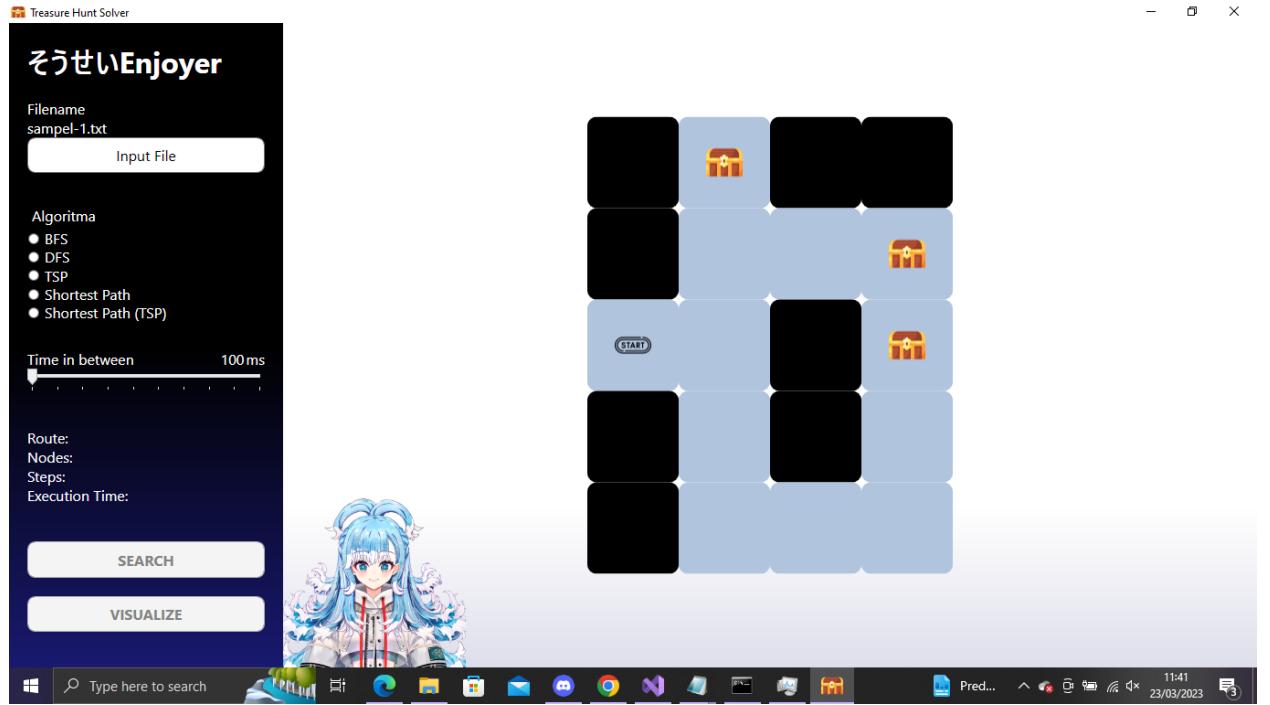
Gambar 4.3.1 Tampilan awal program

3. Klik tombol "Input File" untuk memilih file .txt dari maze dengan format seperti pada deskripsi masalah



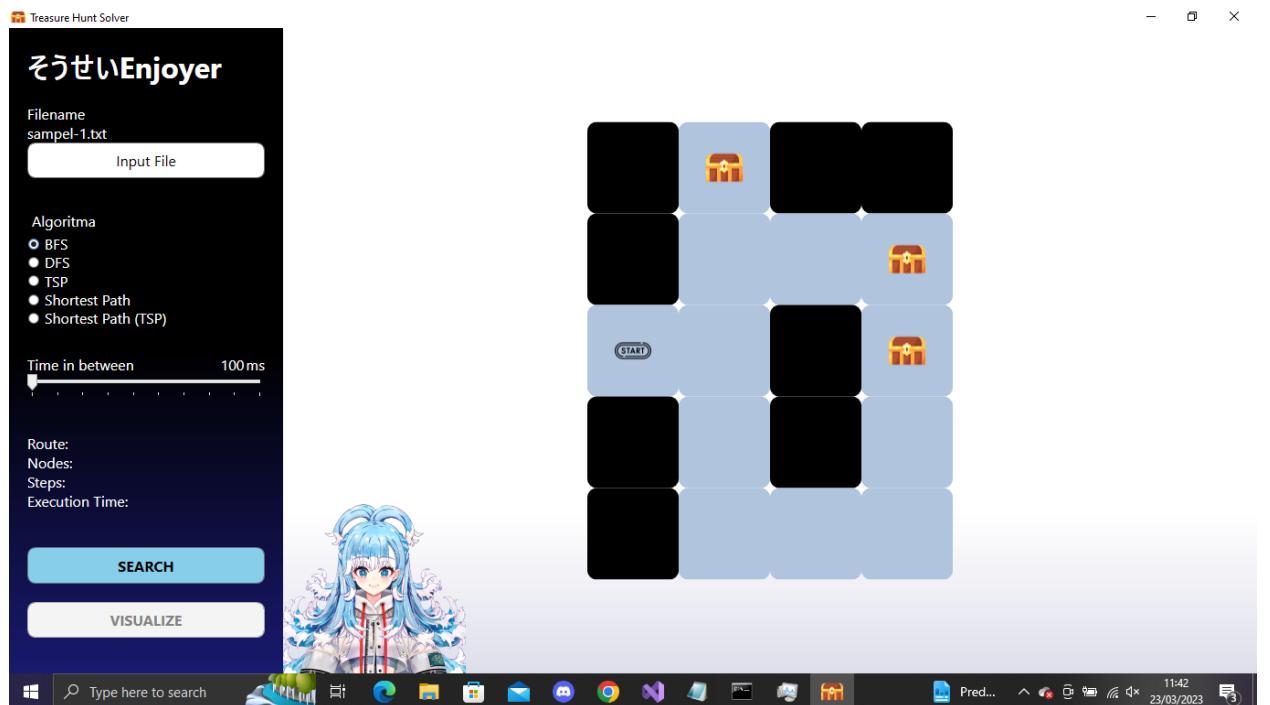
Gambar 4.3.2 Input file maze

4. Jika file yang di-*input* valid, akan muncul nama file yang berhasil di-*input* dan tampilan awal maze seperti berikut



Gambar 4.3.3 Tampilan awal maze

5. Pilih algoritma yang akan digunakan dengan menekan tombol radio button di dalam section Algoritma. Berikut algoritma yang dapat dipilih

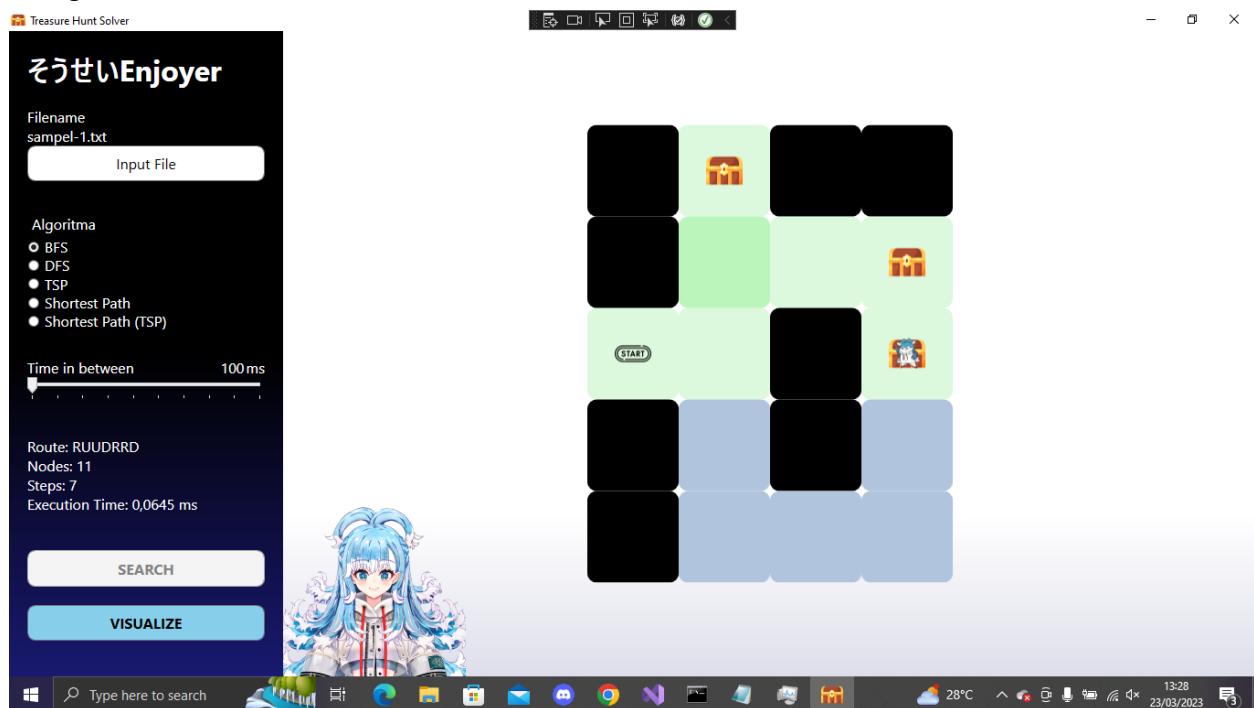


Gambar 4.3.4 Algoritma yang dapat dipilih

- a. BFS: Algoritma pencarian *treasure* dengan *breadth-first search*
- b. DFS: Algoritma pencarian *treasure* dengan *depth-first search*
- c. TSP: Algoritma pencarian *treasure* dengan *breadth-first search* dan rute solusi yang diperoleh juga kembali ke titik awal setelah menemukan segala harta karunnya (*Travelling Salesman Problem*)
- d. Shortest Path: Sebuah eksperimen dengan perpaduan algoritma DFS dan *Dijkstra* untuk mencari *shortest-path* dari *treasure hunt*
- e. Shortest Path (TSP): Sebuah eksperimen dengan perpaduan algoritma DFS dan *Dijkstra* untuk mencari *shortest-path* dari *treasure hunt* dan rute solusi yang diperoleh juga kembali ke titik awal

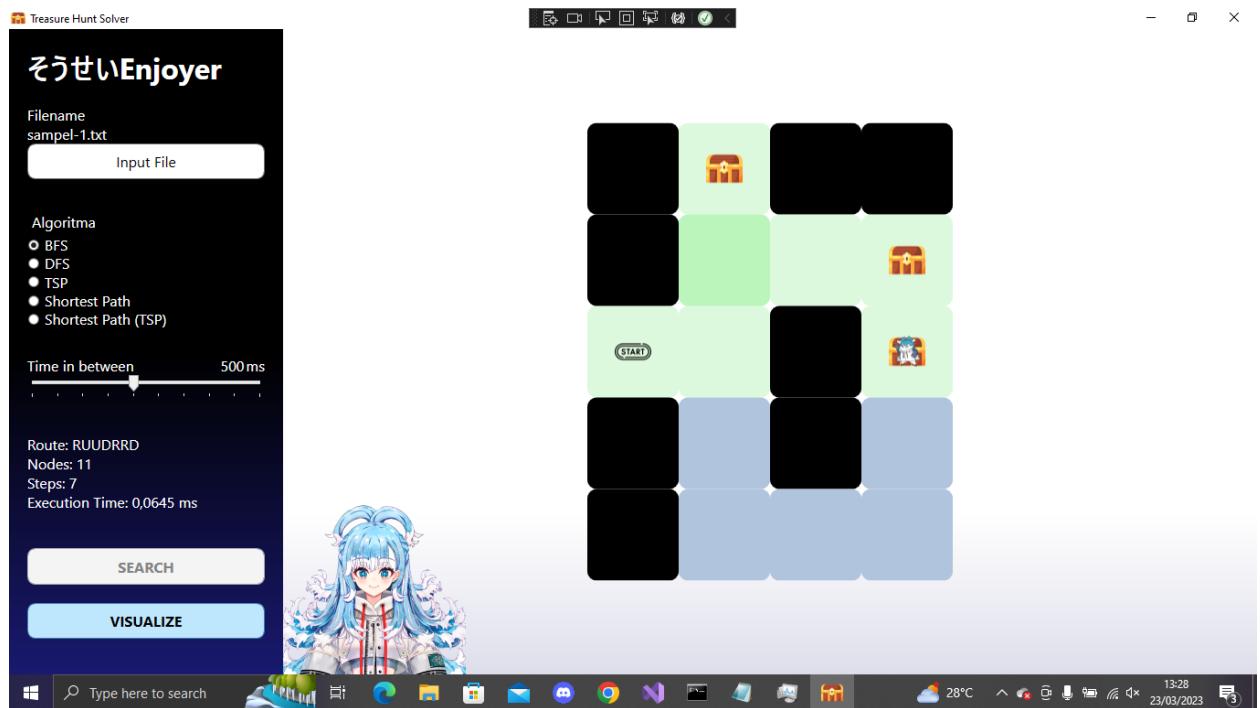
Catatan untuk algoritma (d) dan (e) hanya dapat dilakukan untuk jumlah *treasure* yang kurang dari sama dengan 10

6. Setelah memilih algoritma yang akan digunakan, klik tombol “Search” untuk memulai kalkulasi pencarian solusi. Setelah itu, akan muncul rute, jumlah node, jumlah step, dan waktu eksekusi dari algoritma tersebut. Pada maze juga terlihat rute dari hasil pencarian yang ditandai dengan *tile* berwarna hijau. Semakin tebal warna hijau, maka semakin sering *tile* tersebut dilewati



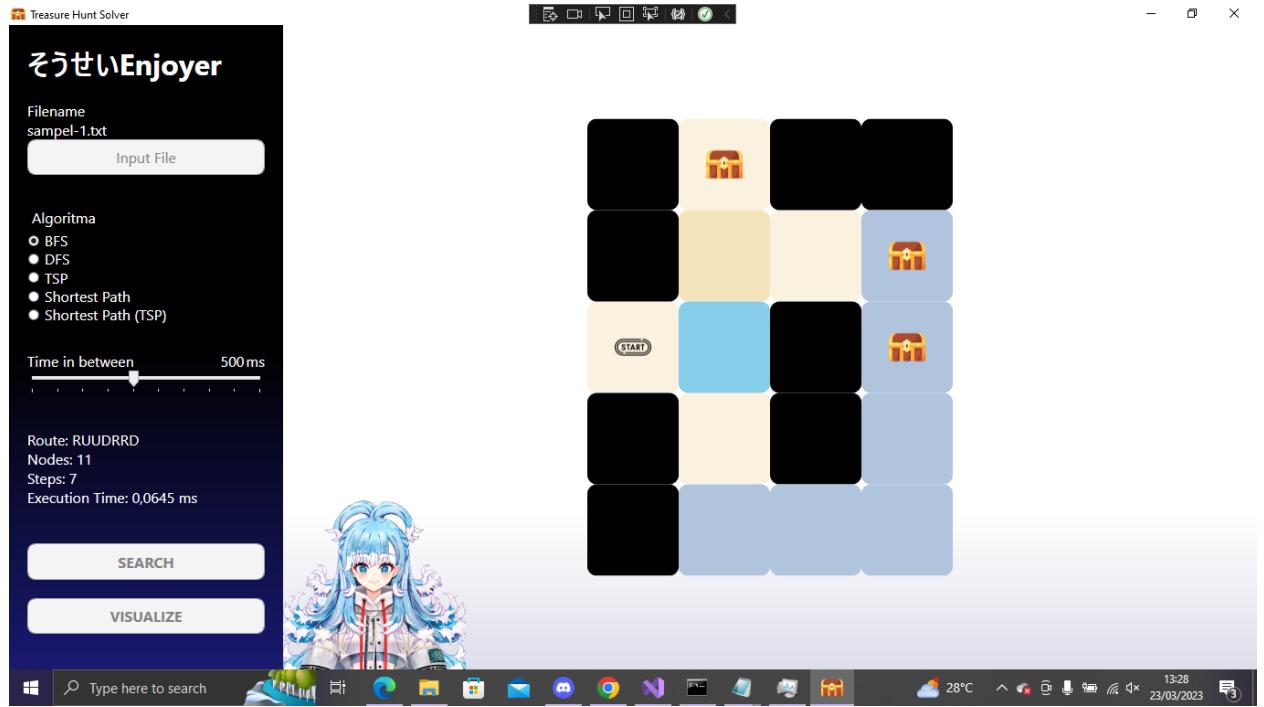
Gambar 4.3.5 Hasil pencarian dari algoritma

7. Atur durasi jeda setiap step dengan menggeser *slider* “Time in between”. Durasi ini diperlukan dalam menunjukkan progress pencarian solusi dengan algoritma yang bersesuaian



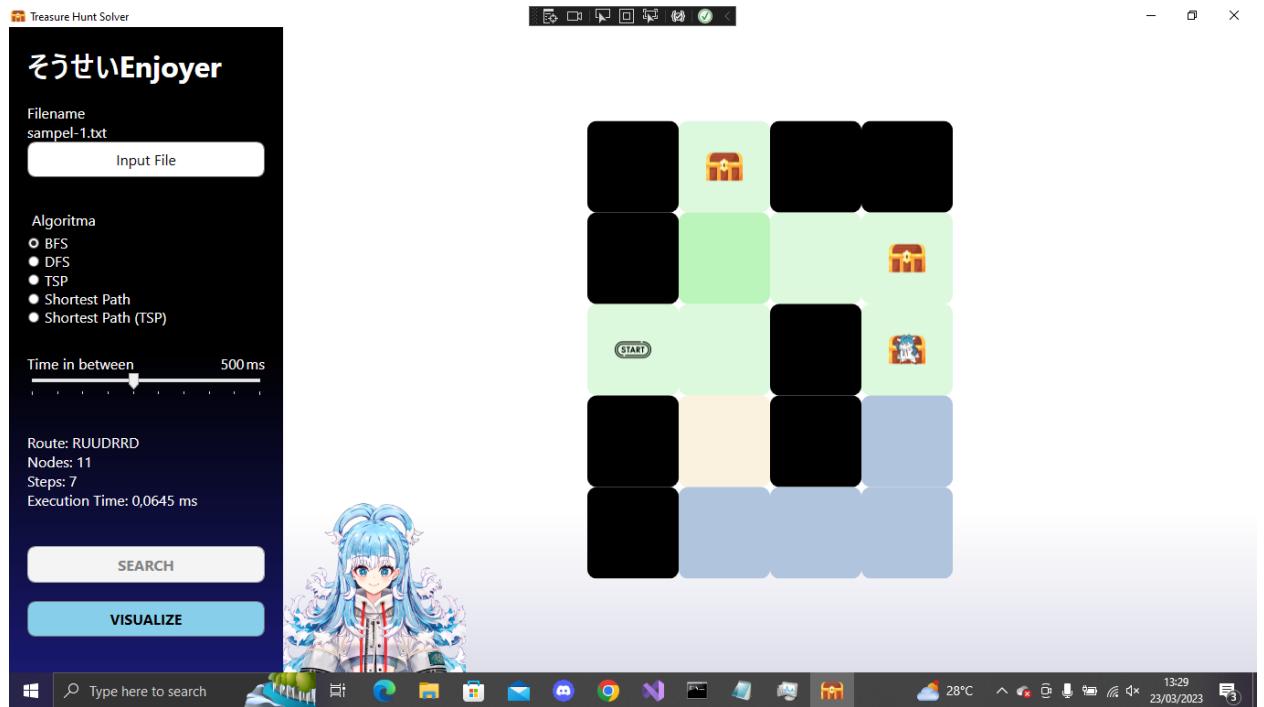
Gambar 4.3.6 *Slider* durasi jeda setiap step

8. Klik tombol “Visualize” untuk melihat progress pencarian solusi dengan algoritma yang bersesuaian. Warna kuning berarti *tile* tersebut sudah diperiksa dan biru berarti *tile* tersebut sedang diperiksa. Semakin tebal warna kuning, maka semakin sering *tile* tersebut diperiksa



Gambar 4.3.7 Progress pencarian solusi

Di akhir progress pencarian solusi, ditampilkan rute akhir/solusi akhir dengan algoritma yang bersesuaian (warna hijau) seperti poin ketujuh



Gambar 4.3.8 Rute akhir/Solusi akhir

#### 4.4 Hasil Pengujian

Program ini diuji menggunakan sistem dengan spesifikasi

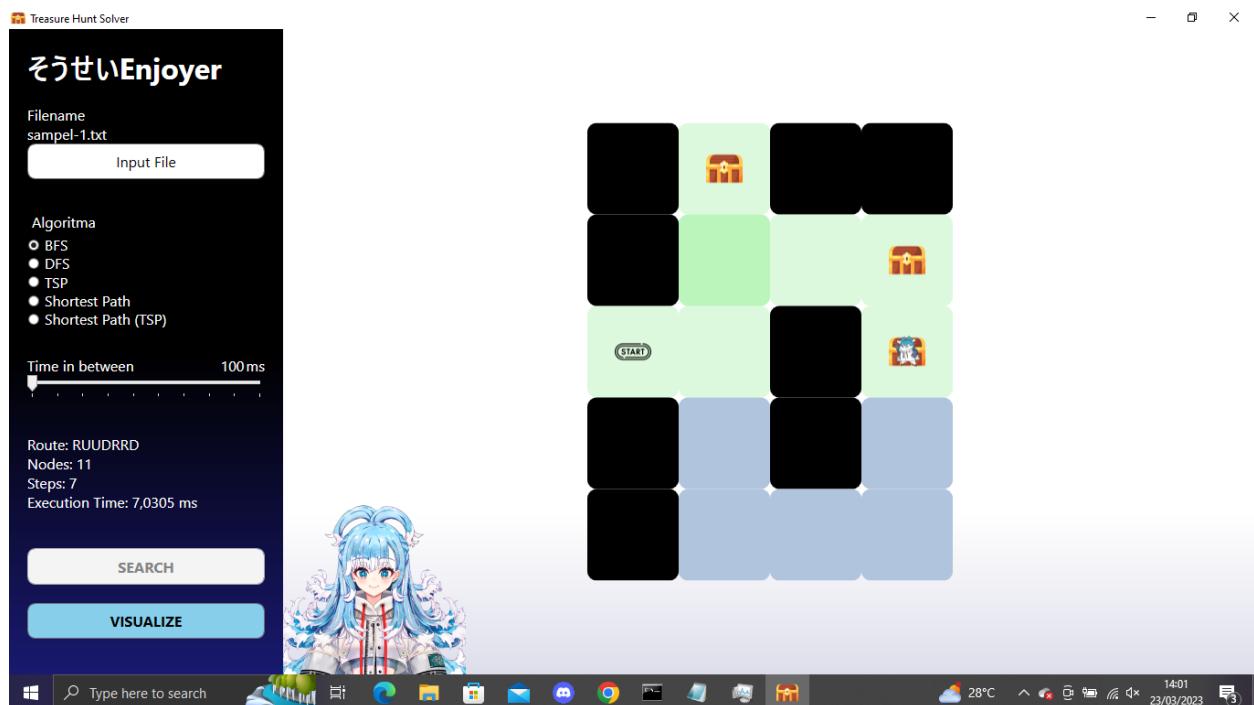
Item	Value
OS	Microsoft Windows 10 Pro
System Type	x64-based PC
Processor	Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz, 2601 Mhz, 2 Core(s), 4 Logical Processor(s)
RAM	8 GB

##### 4.4.1 Test Case 1

```
X T X X  
X R R T  
K R X T  
X R X R  
X R R R
```

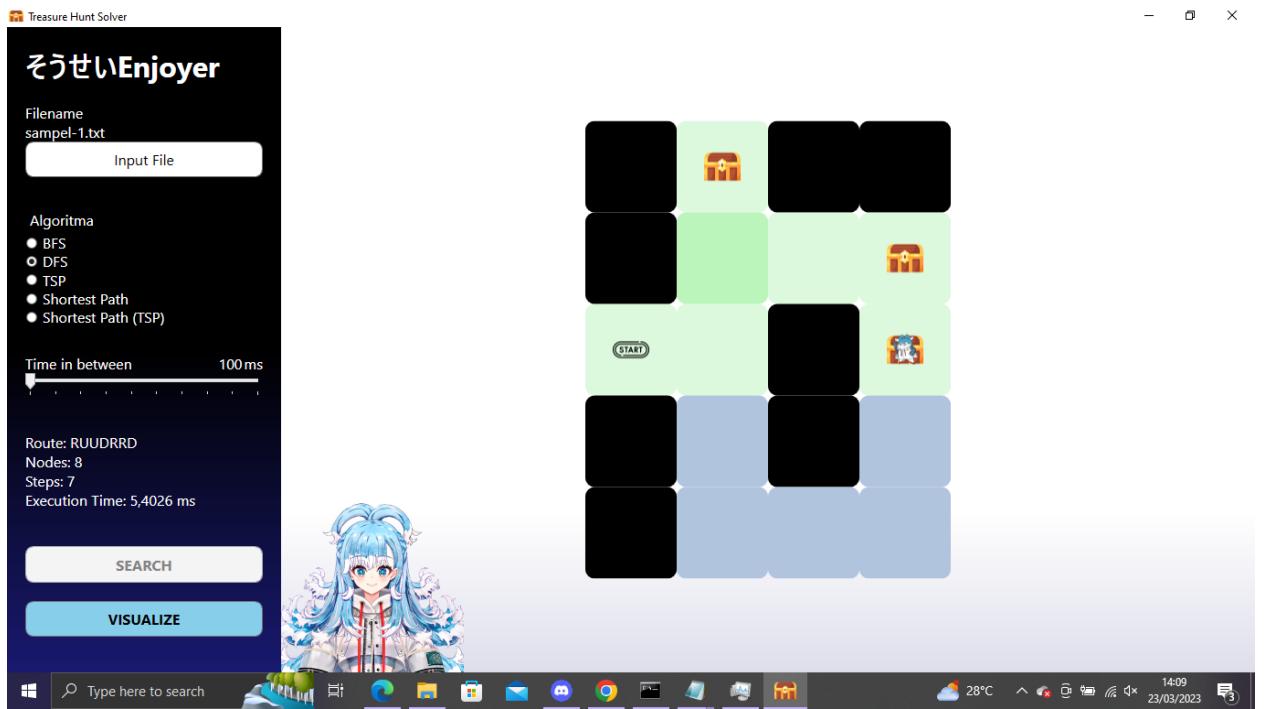
Gambar 4.4.1.1 Test Case 1

###### a. BFS



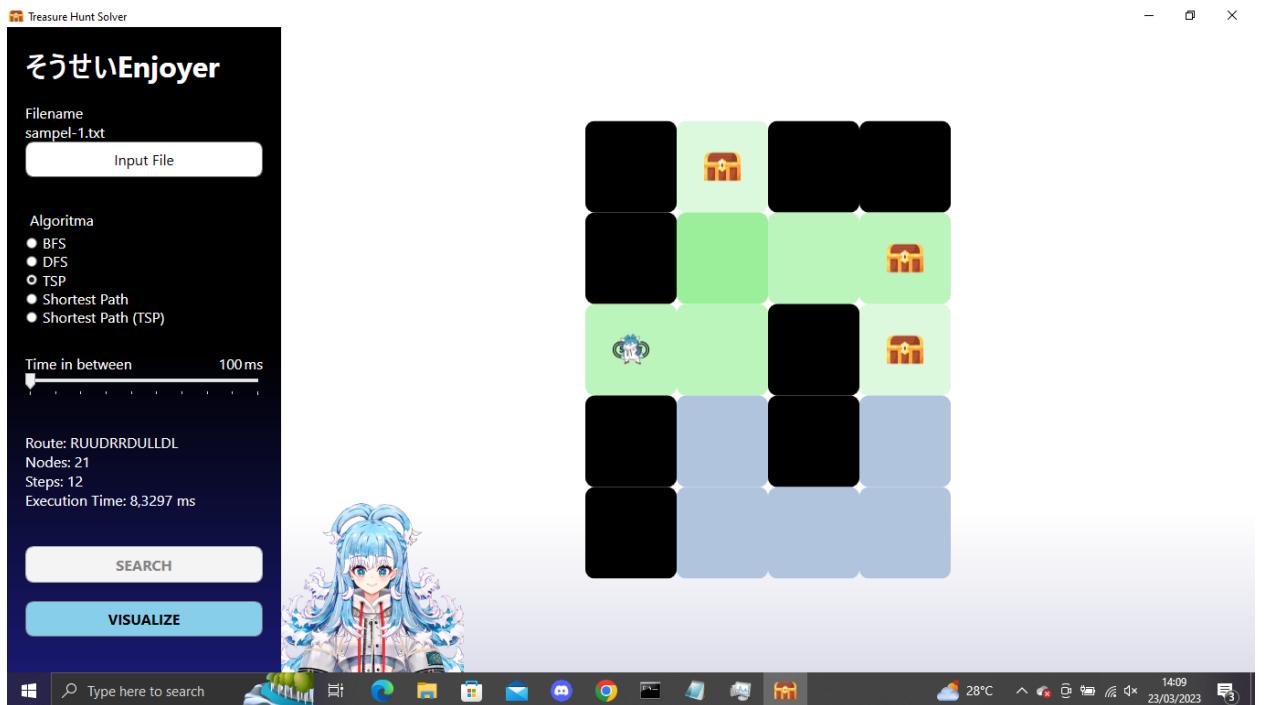
Gambar 4.4.1.2 Tampilan pencarian BFS untuk Test Case 1

### b. DFS



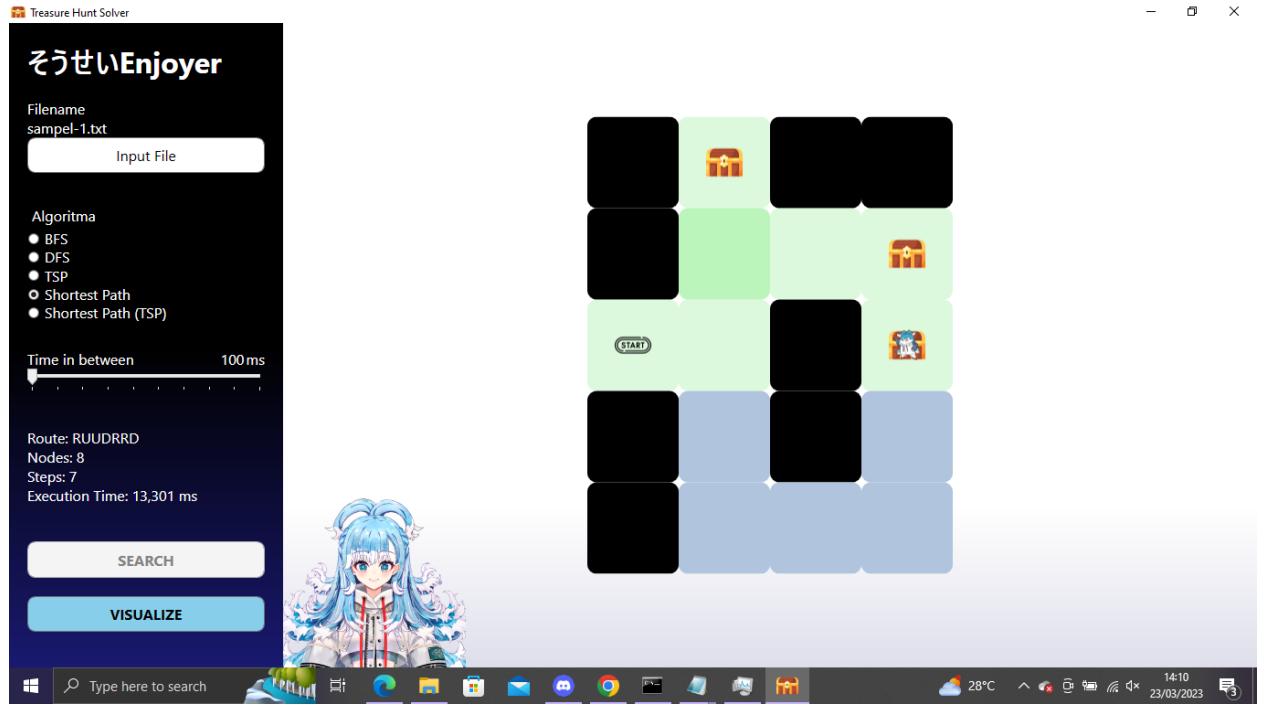
Gambar 4.4.1.3 Tampilan pencarian DFS untuk Test Case 1

### c. TSP



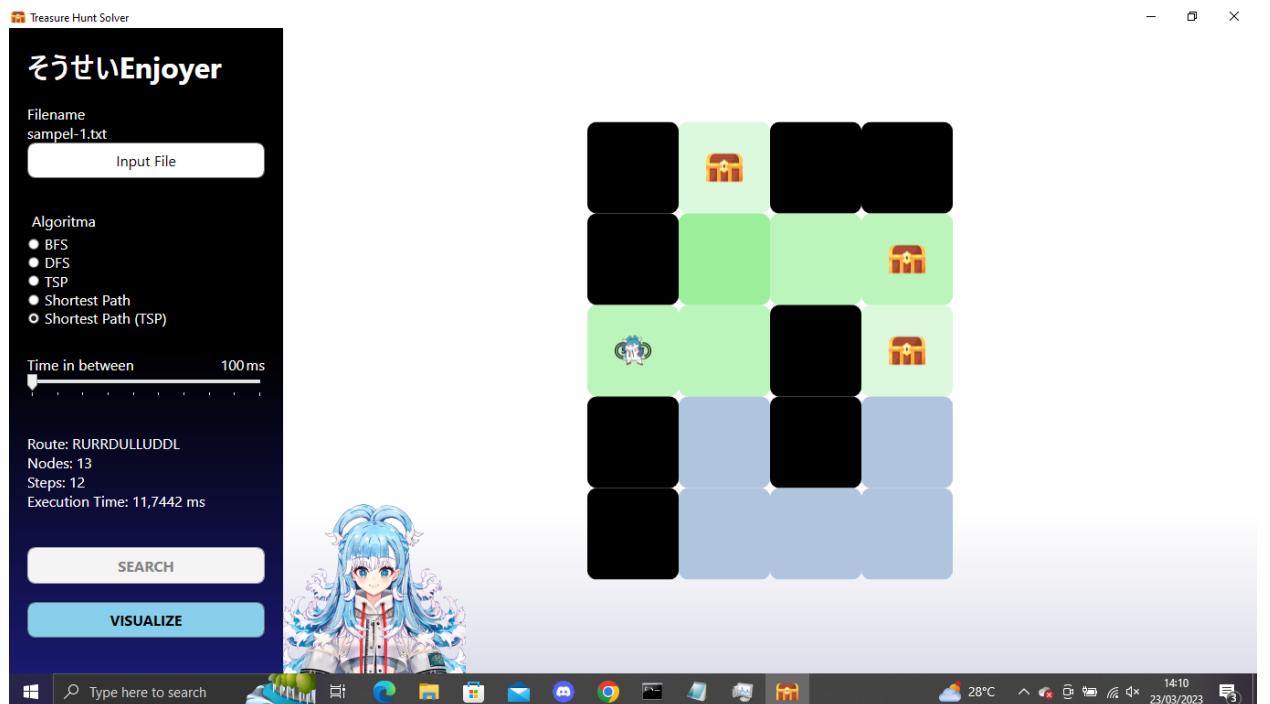
Gambar 4.4.1.4 Tampilan pencarian TSP untuk Test Case 1

### d. Shortest Path - Eksperimen



Gambar 4.4.1.5 Tampilan pencarian Shortest Path untuk Test Case 1

#### e. Shortest Path (TSP) - Eksperimen



Gambar 4.4.1.6 Tampilan pencarian Shortest Path (TSP) untuk Test Case 1

#### 4.4.2 Test Case 2

```

X X X X X X X X X
X X X X X X X X X
X X T K R T X X
X X X X X X X X X
X X X X X X X X X

```

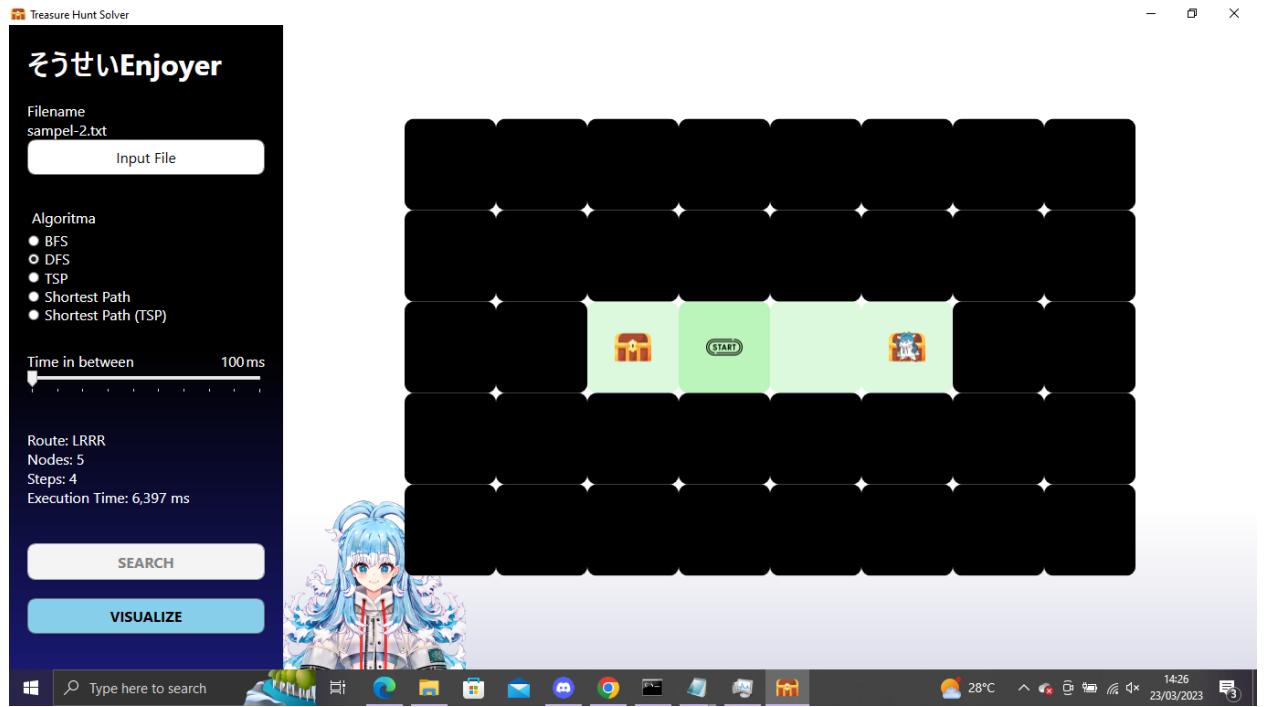
Gambar 4.4.2.1 Test Case 2

#### a. BFS



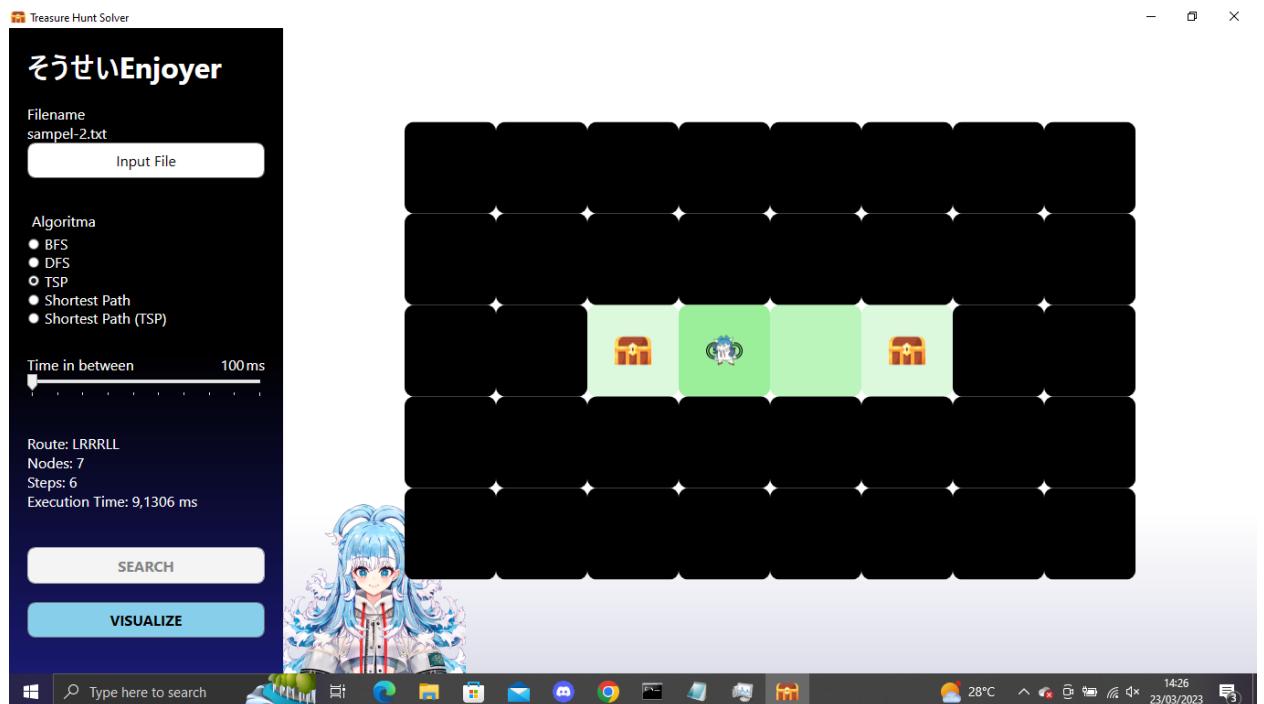
Gambar 4.4.2.2 Tampilan pencarian BFS untuk Test Case 2

#### b. DFS



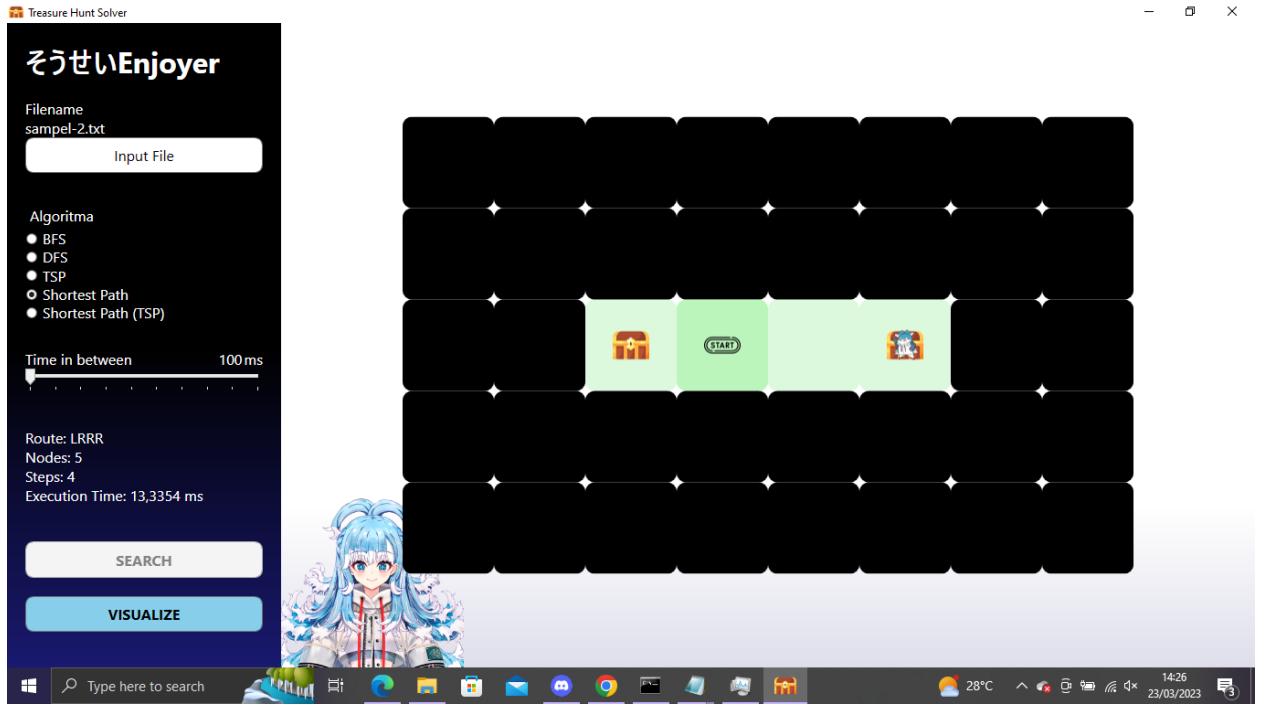
Gambar 4.4.2.3 Tampilan pencarian DFS untuk Test Case 2

### c. TSP



Gambar 4.4.2.4 Tampilan pencarian TSP untuk Test Case 2

### d. Shortest Path - Eksperimen



Gambar 4.4.2.5 Tampilan pencarian Shortest Path untuk Test Case 2

#### e. Shortest Path (TSP) - Eksperimen

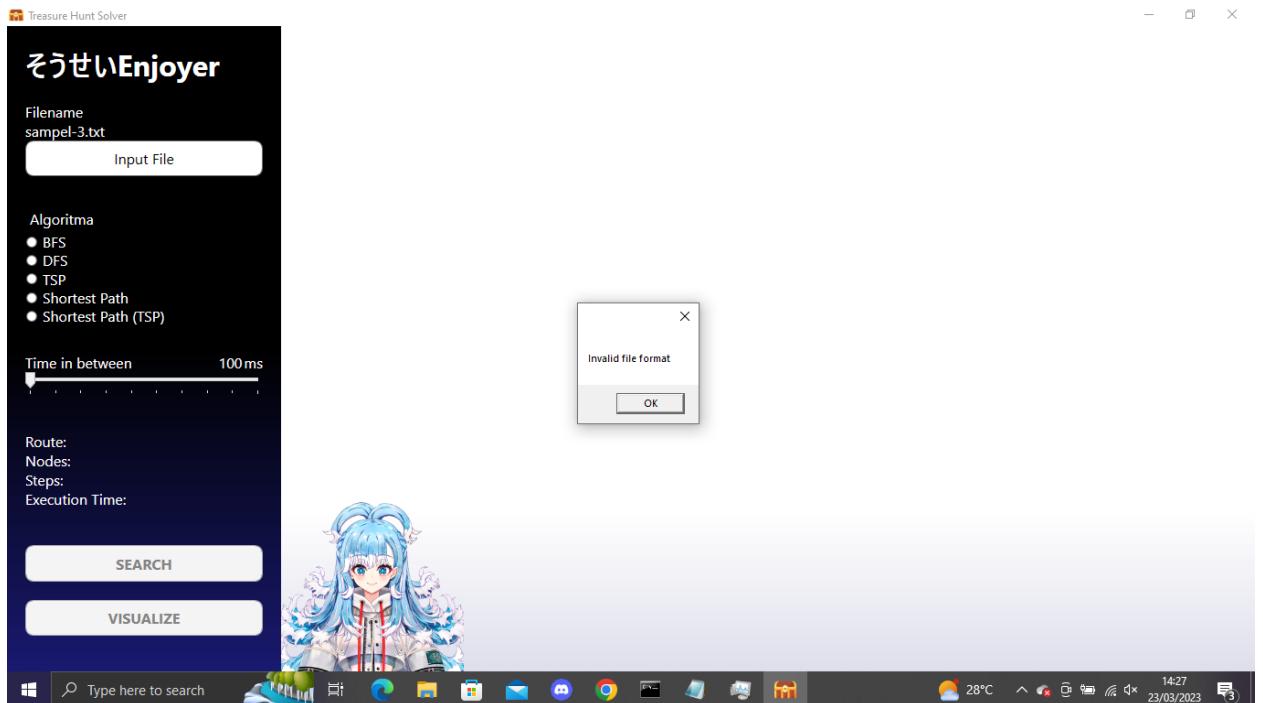


Gambar 4.4.2.6 Tampilan pencarian Shortest Path (TSP) untuk Test Case 2

#### 4.4.3 Test Case 3

J A N G A N  
L U P A C E  
K Y A N G B  
E G I N I Y

Gambar 4.4.3.1 Test Case 3



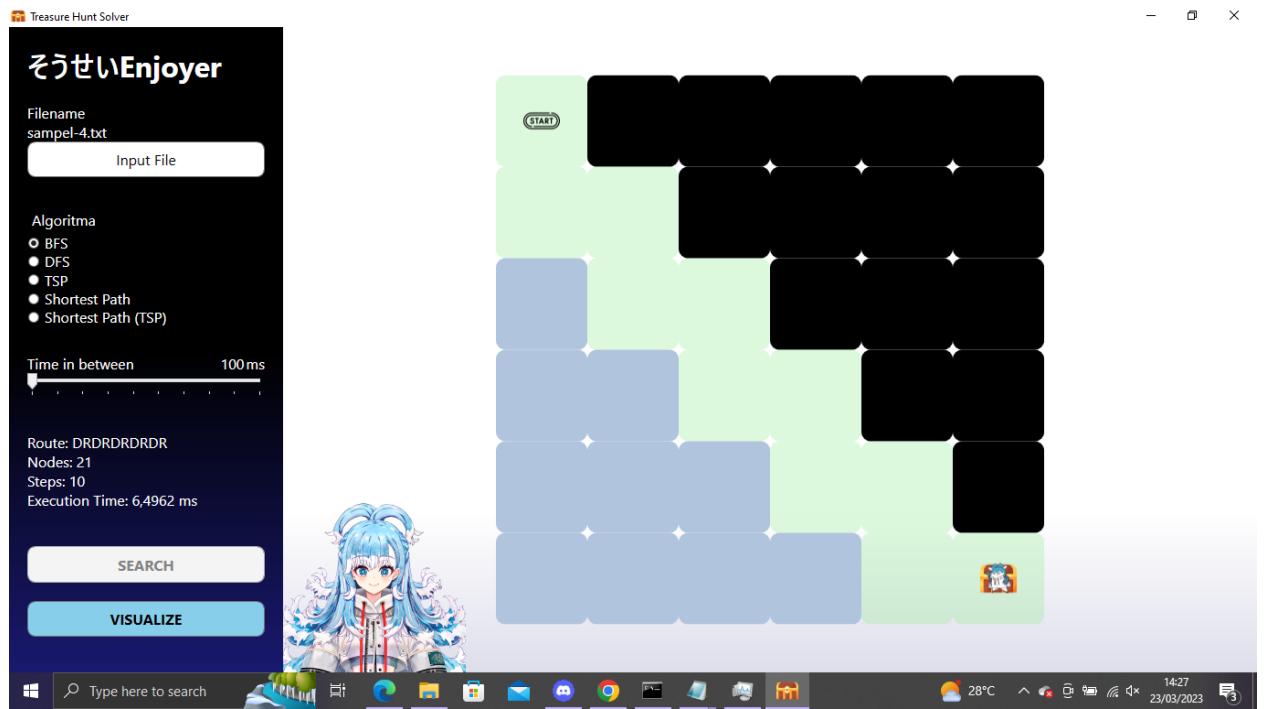
Gambar 4.4.3.2 Tampilan hasil Test Case 3

#### 4.4.4 Test Case 4

K X X X X X X  
R R X X X X  
R R R X X X  
R R R R X X  
R R R R R X  
R R R R R T

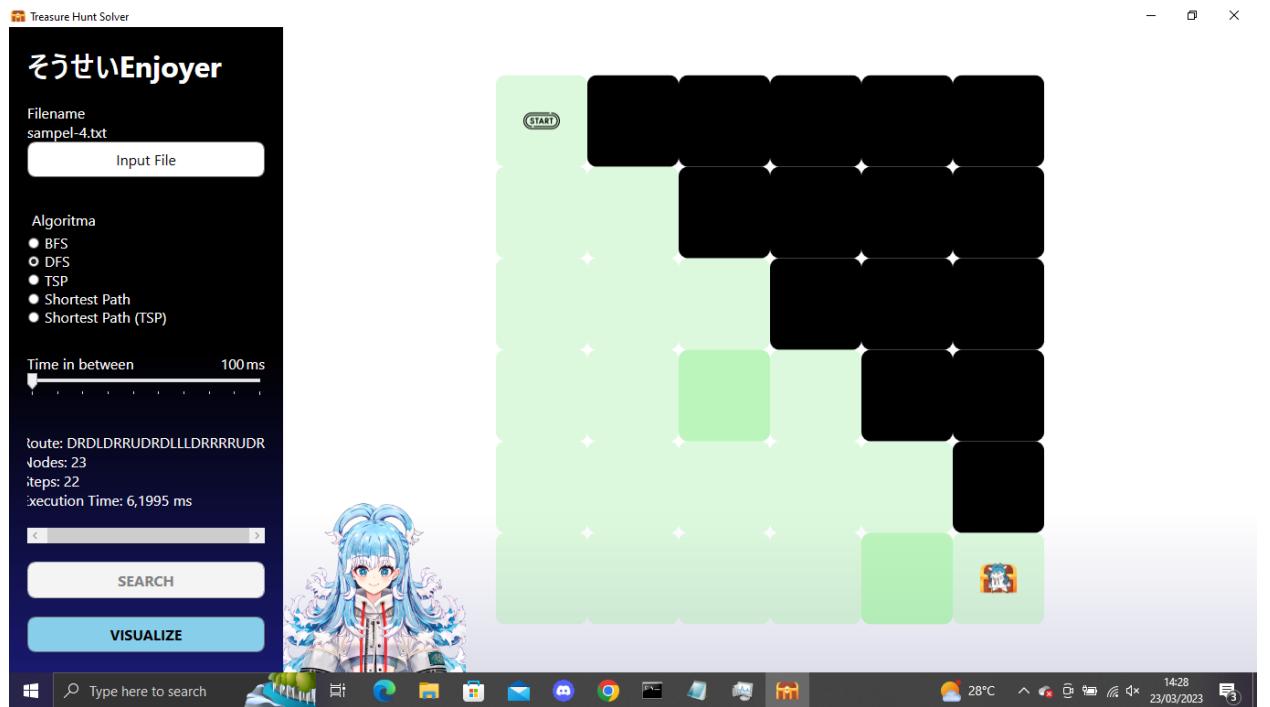
Gambar 4.4.4.1 Test Case 4

### a. BFS



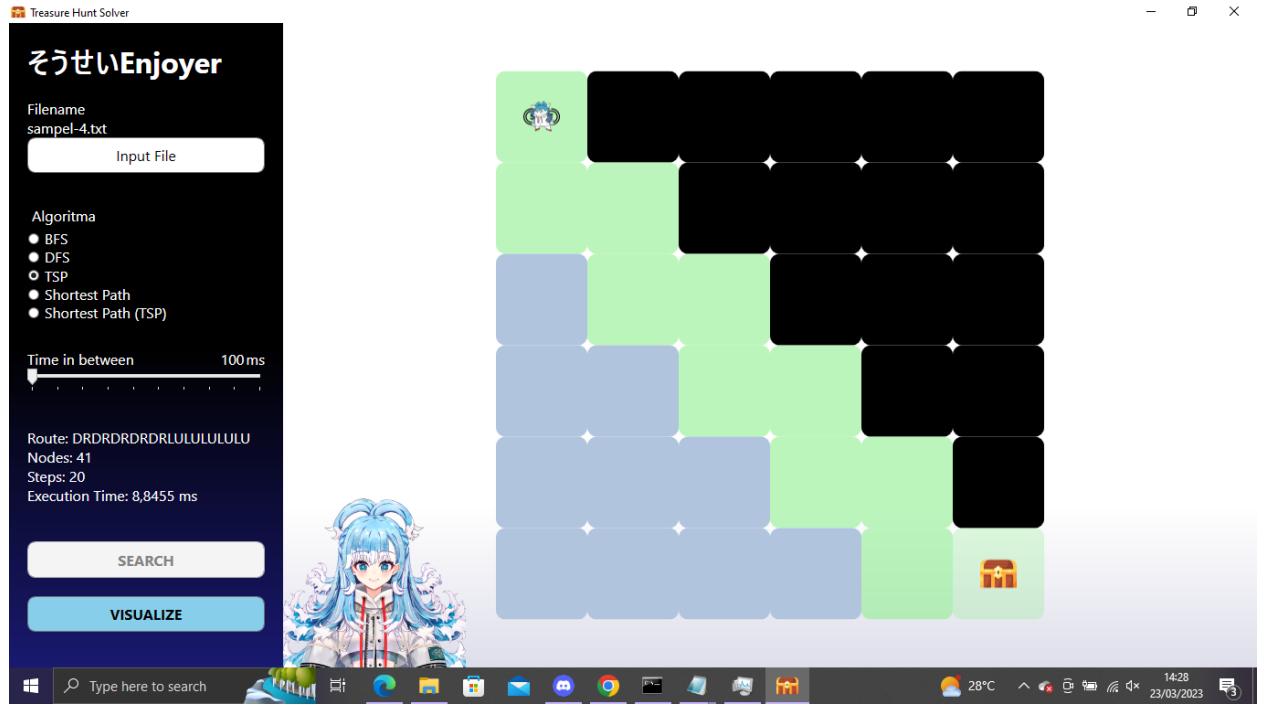
Gambar 4.4.4.2 Tampilan pencarian BFS untuk Test Case 4

### b. DFS



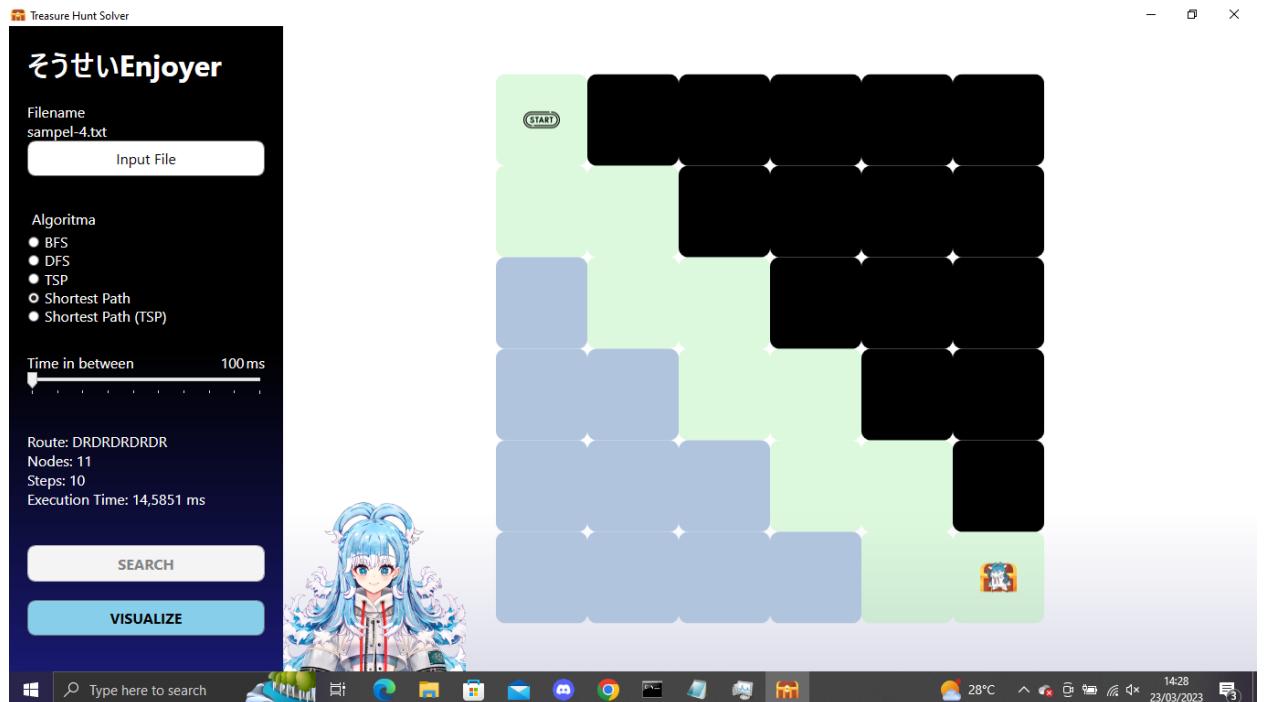
Gambar 4.4.4.3 Tampilan pencarian DFS untuk Test Case 4

### c. TSP



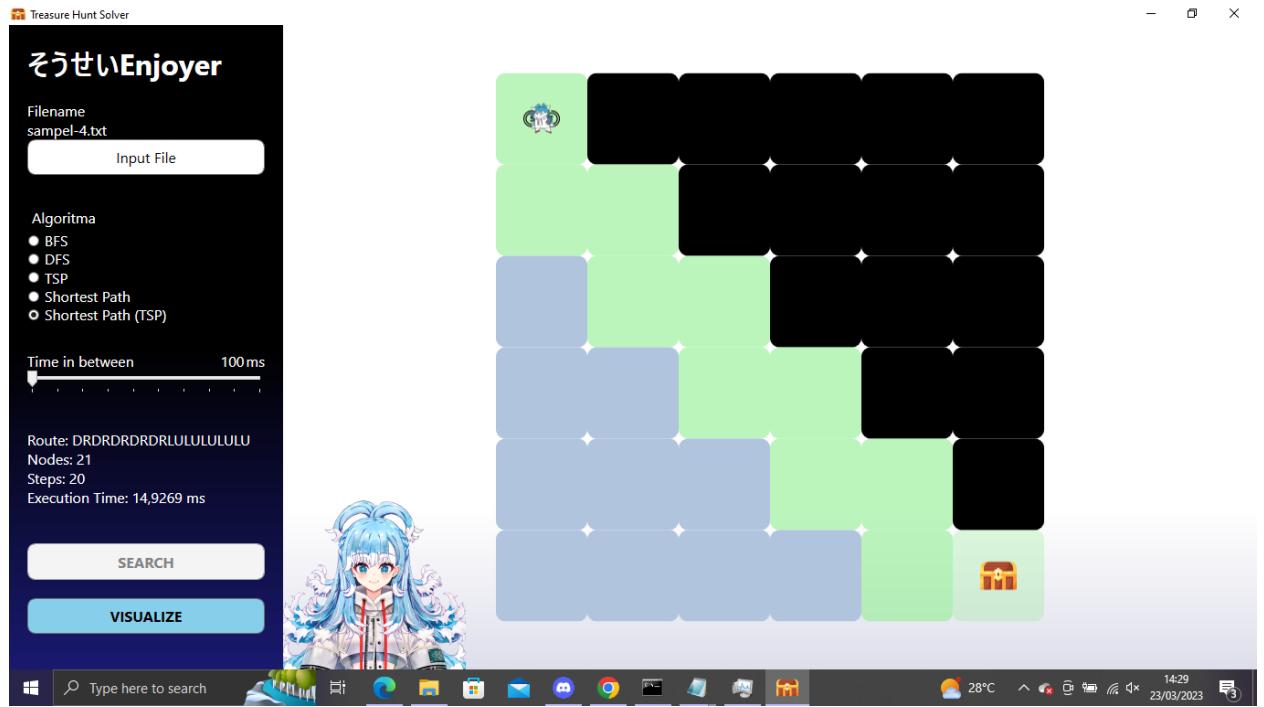
Gambar 4.4.4.4 Tampilan pencarian TSP untuk Test Case 4

#### d. Shortest Path - Eksperimen



Gambar 4.4.4.5 Tampilan pencarian Shortest Path untuk Test Case 4

#### e. Shortest Path (TSP) - Eksperimen



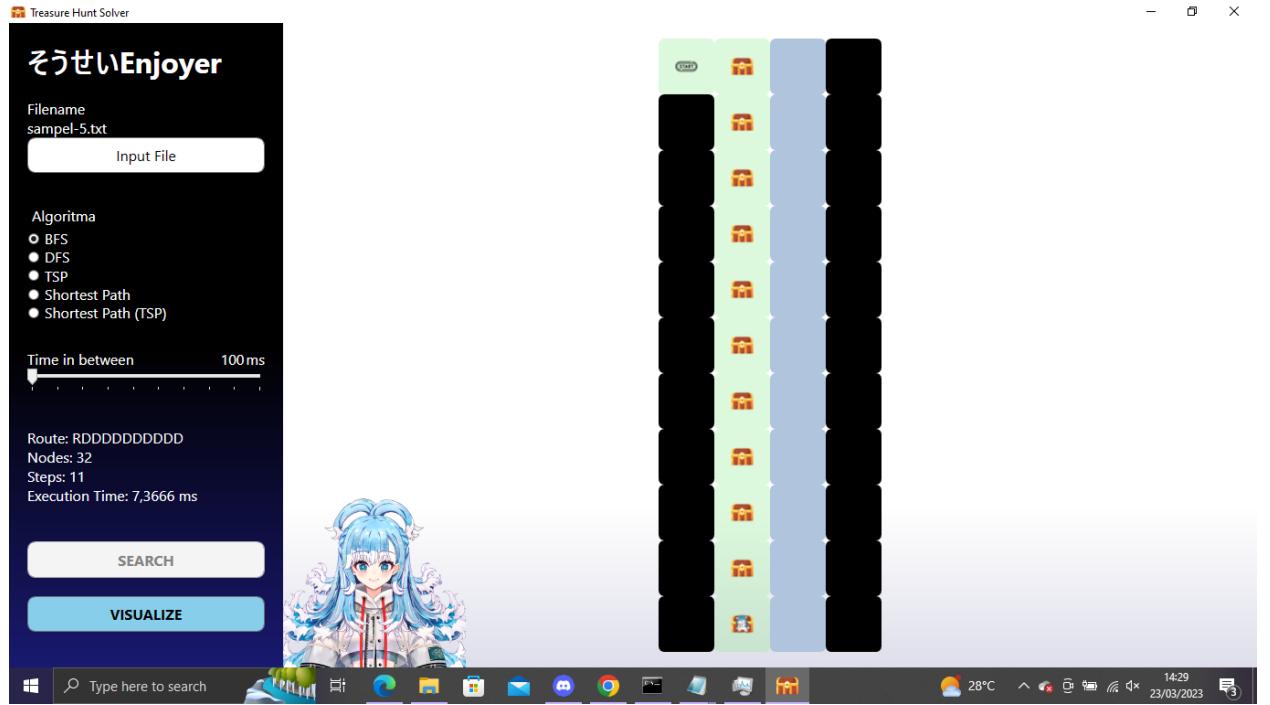
Gambar 4.4.4.6 Tampilan pencarian Shortest Path (TSP) untuk Test Case 4

#### 4.4.5 Test Case 5

K T R X  
X T R X  
X T R X  
X T R X  
X T R X  
X T R X  
X T R X  
X T R X  
X T R X  
X T R X

Gambar 4.4.5.1 Test Case 5

a. BFS



Gambar 4.4.5.2 Tampilan pencarian BFS untuk Test Case 5

b. DFS



Gambar 4.4.5.3 Tampilan pencarian DFS untuk Test Case 5

c. TSP



Gambar 4.4.5.4 Tampilan pencarian TSP untuk Test Case 5

- d. Shortest Path - Eksperimen  
Waktu eksekusi terlalu lama
- e. Shortest Path (TSP) - Eksperimen  
Waktu eksekusi terlalu lama

#### 4.4.6 Test Case 6

K	R	R	R	R	T	R	R	R
X	R	X	X	R	X	X	X	R
T	R	X	X	R	R	X	X	R
R	X	X	X	R	R	X	X	R
R	R	R	X	X	X	X	X	R
R	X	R	R	R	R	X	X	R
R	X	X	X	X	X	X	X	R
T	X	X	T	R	R	R	R	R
R	R	X	X	R	X	X	X	X
X	R	R	R	R	T	X	X	X

Gambar 4.4.4.1 Test Case 6

- a. BFS  
Rute yang dihasilkan:

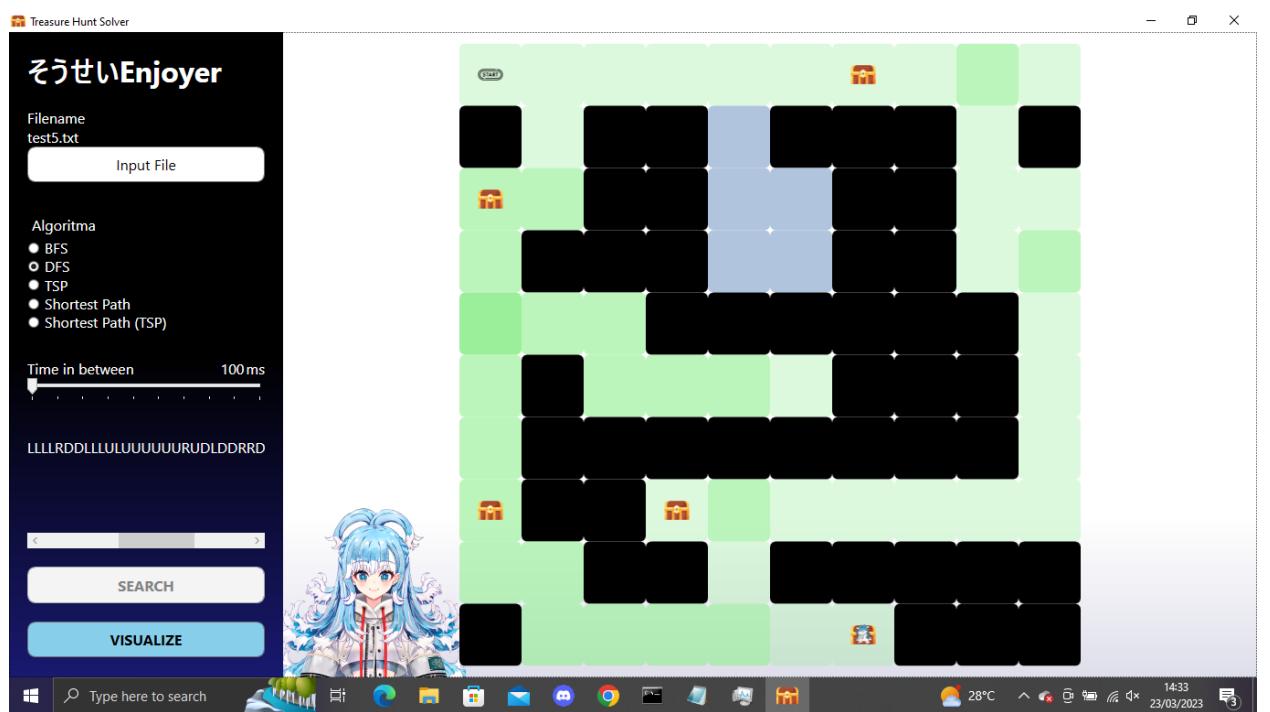
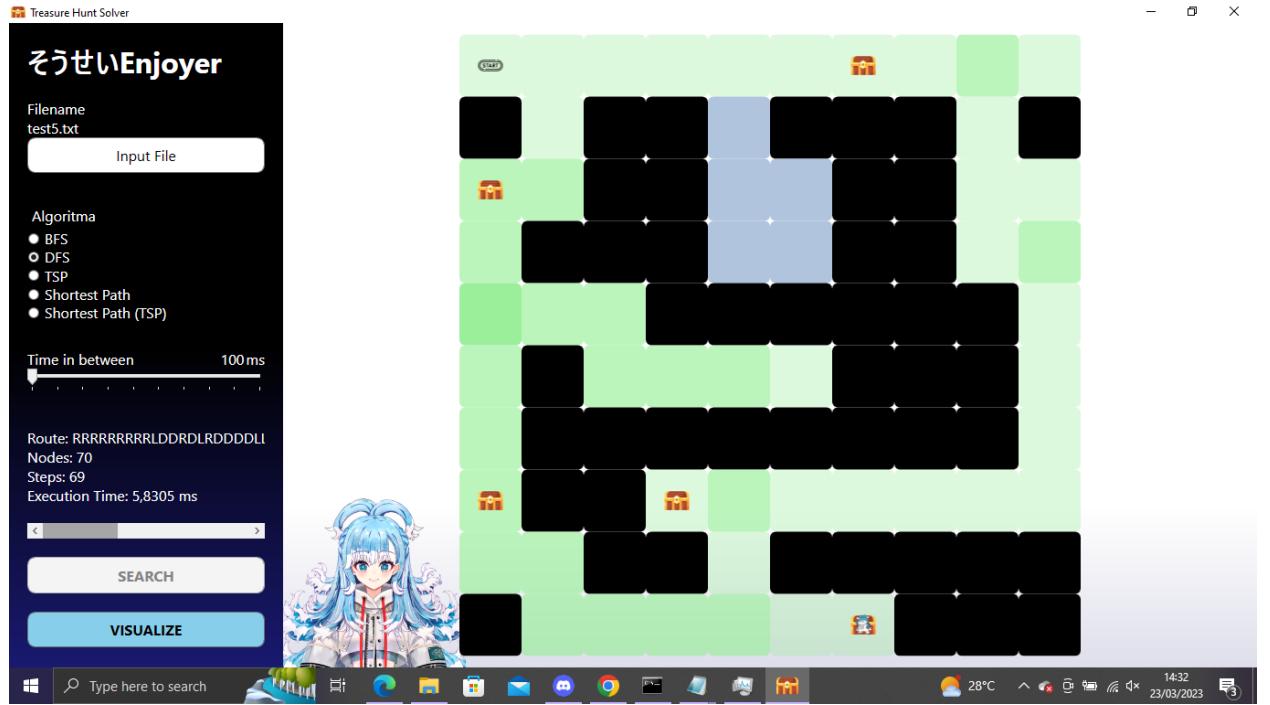


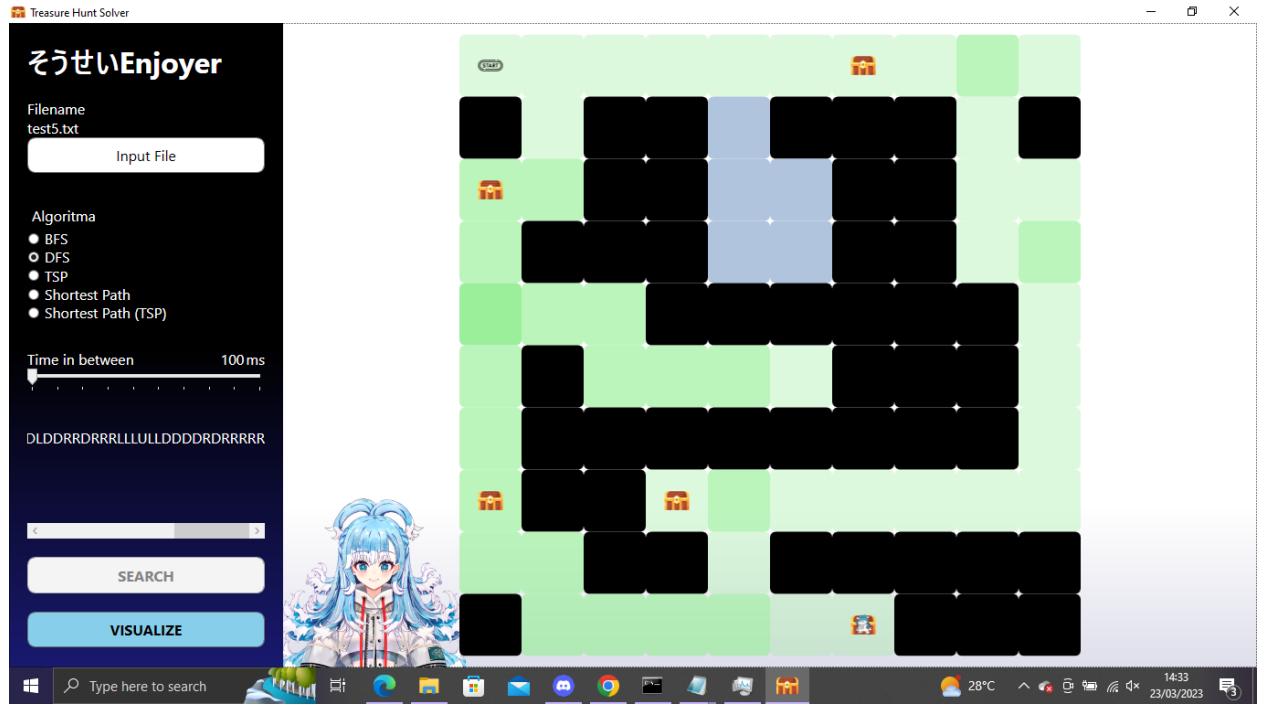
Gambar 4.4.6.2 Tampilan pencarian BFS untuk Test Case 6

b. DFS

Rute yang dihasilkan:

RRRRRRRRRLDDRDLRDDDLUULLLRDDLLLULUUUUURUDLDDRR  
DRRRLLLULLDLDRRRRRR



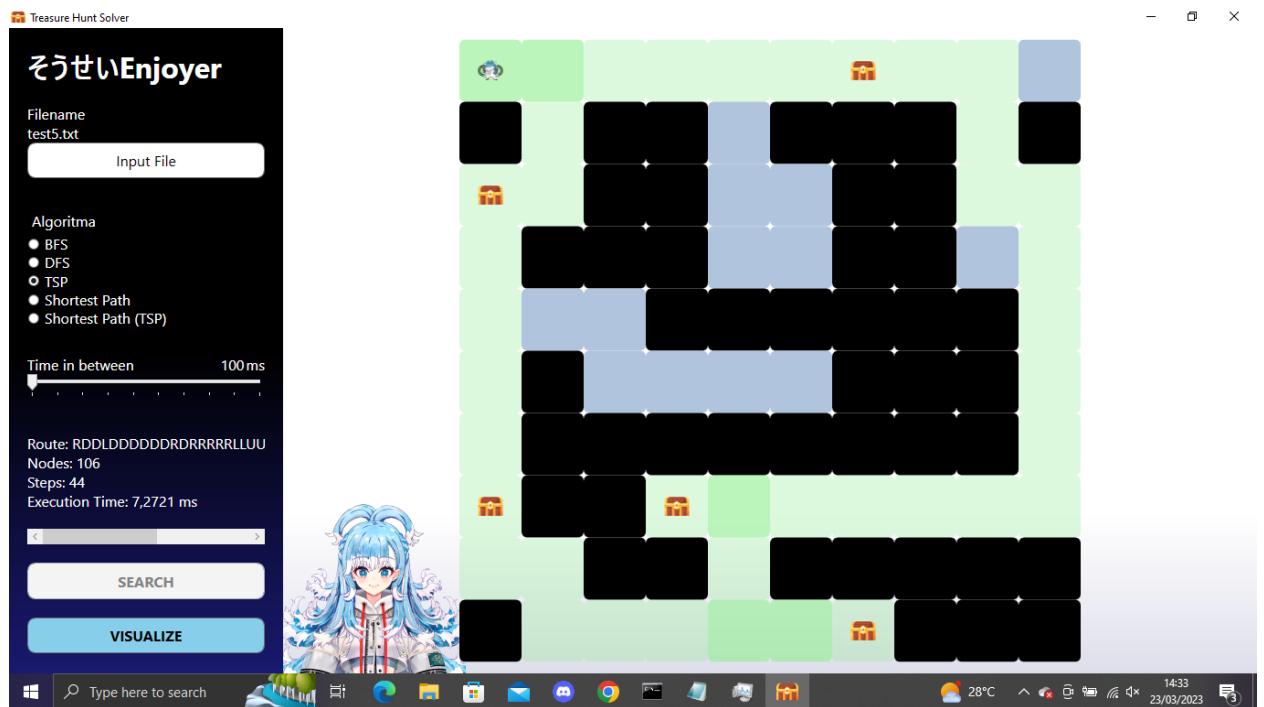


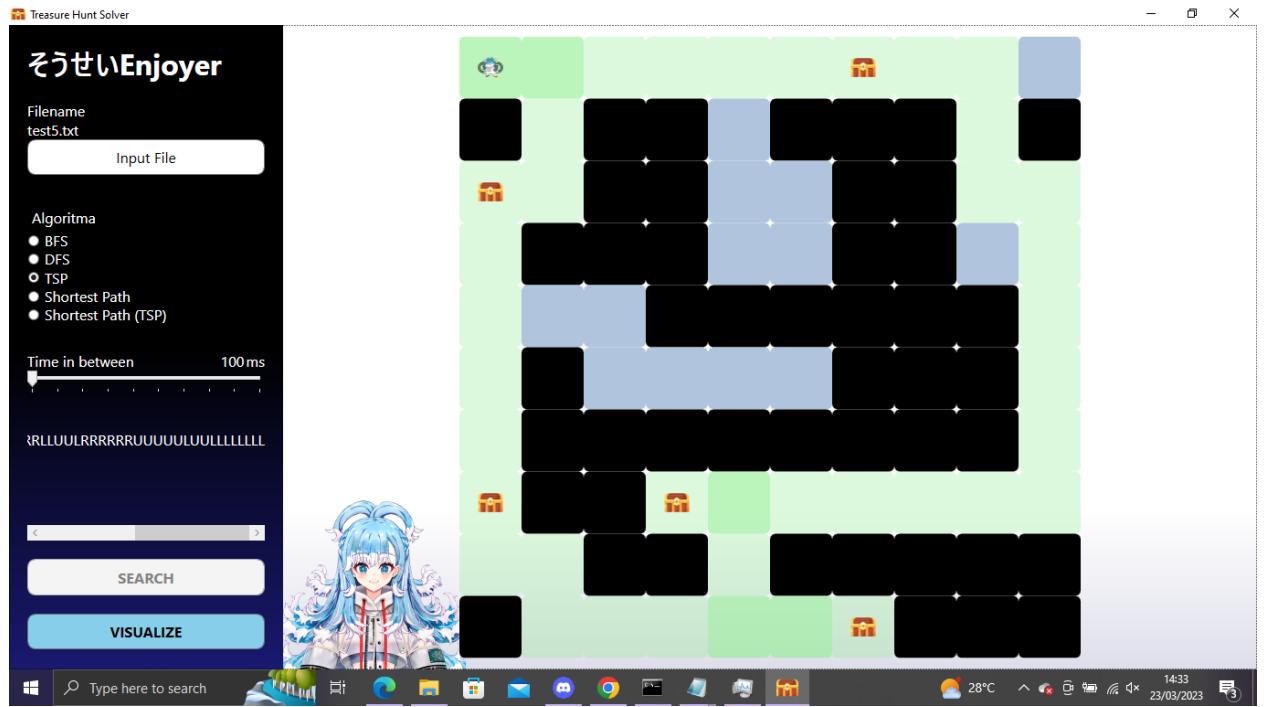
Gambar 4.4.6.3 Tampilan pencarian DFS untuk Test Case 6

### c. TSP

Rute yang dihasilkan:

RDDLLLLDDDRDRRRRRLUULRRRRRUUUULUULLLLLLL



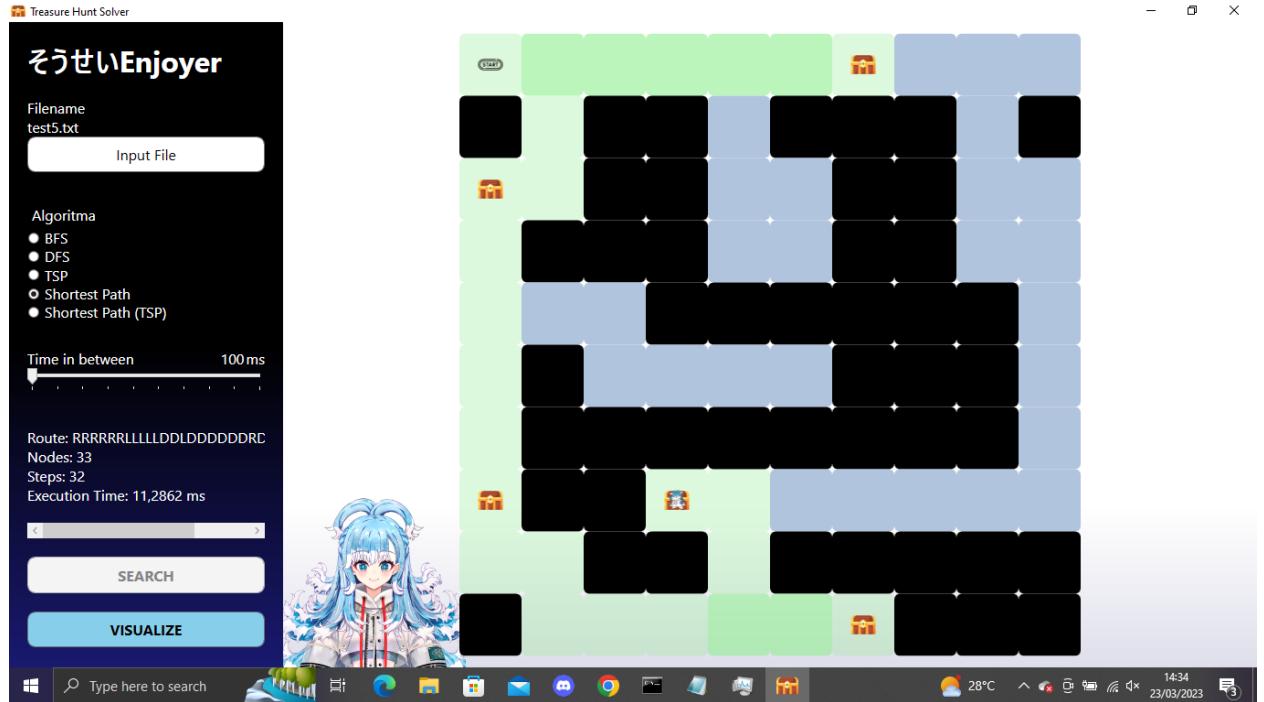


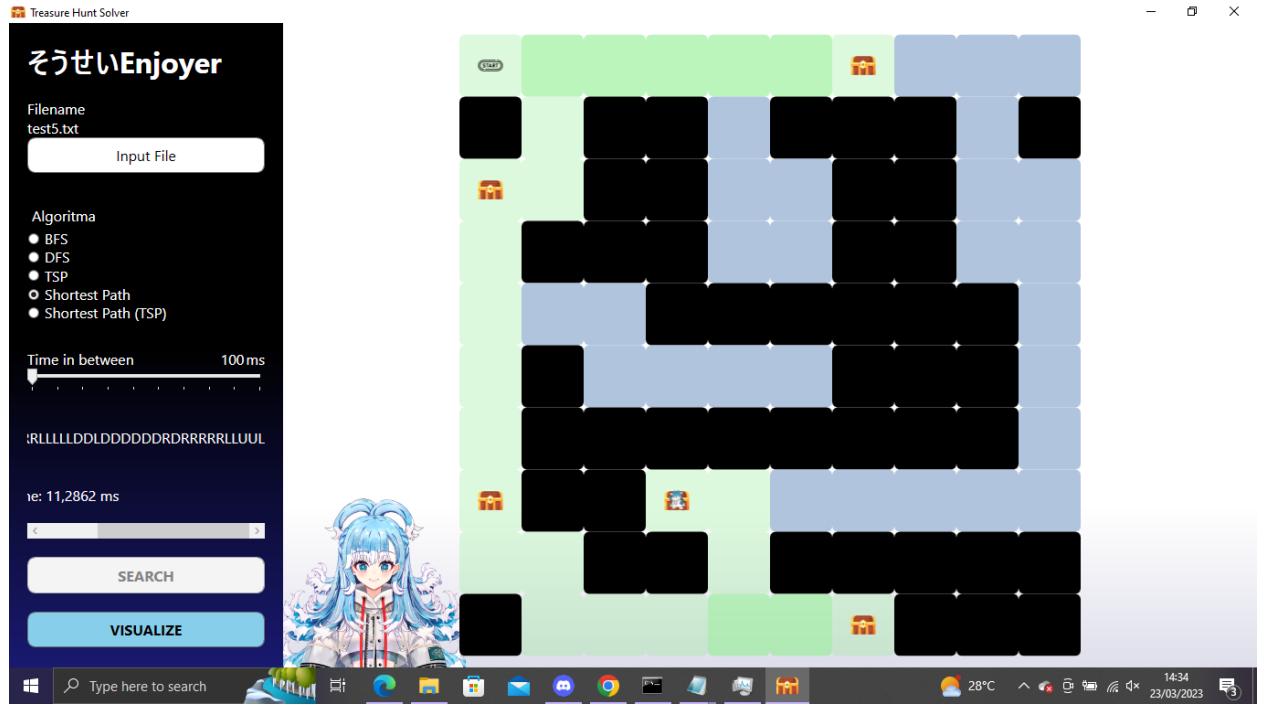
Gambar 4.4.6.4 Tampilan pencarian TSP untuk Test Case 6

d. Shortest Path - Eksperimen

Rute yang dihasilkan:

RRRRRRRLLLDDDDDDDRDRRRRRLLUUL



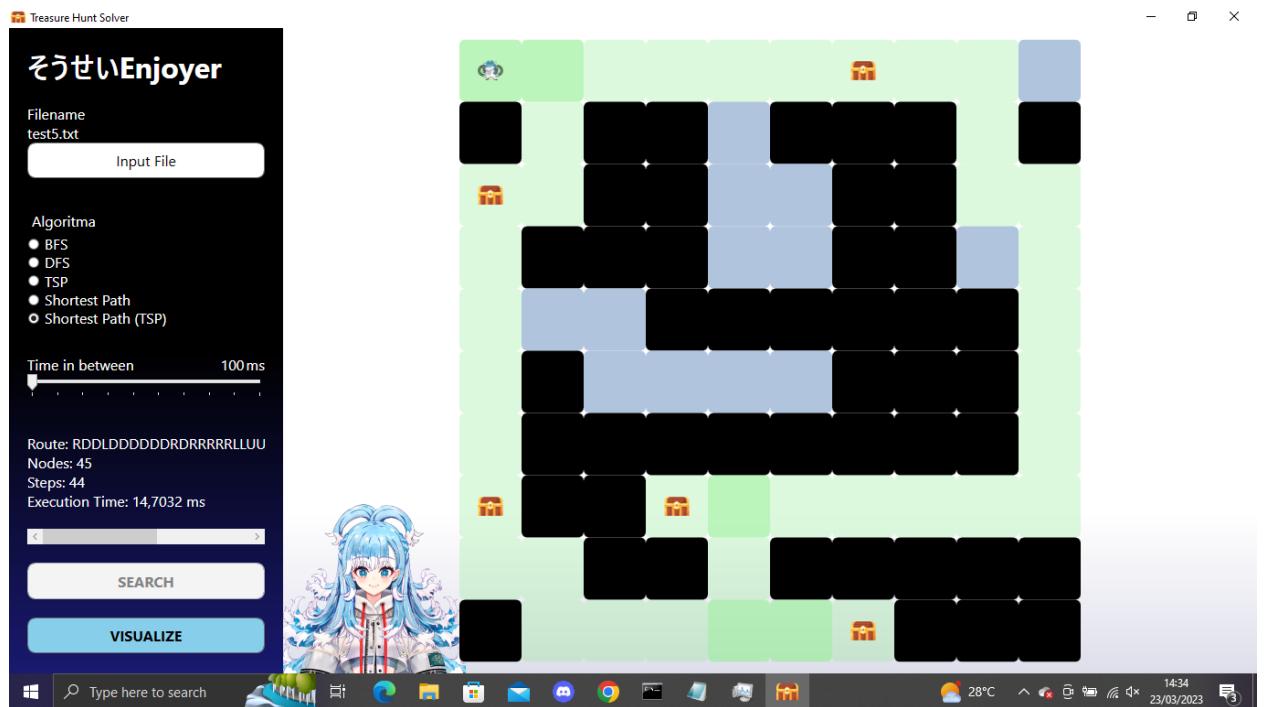


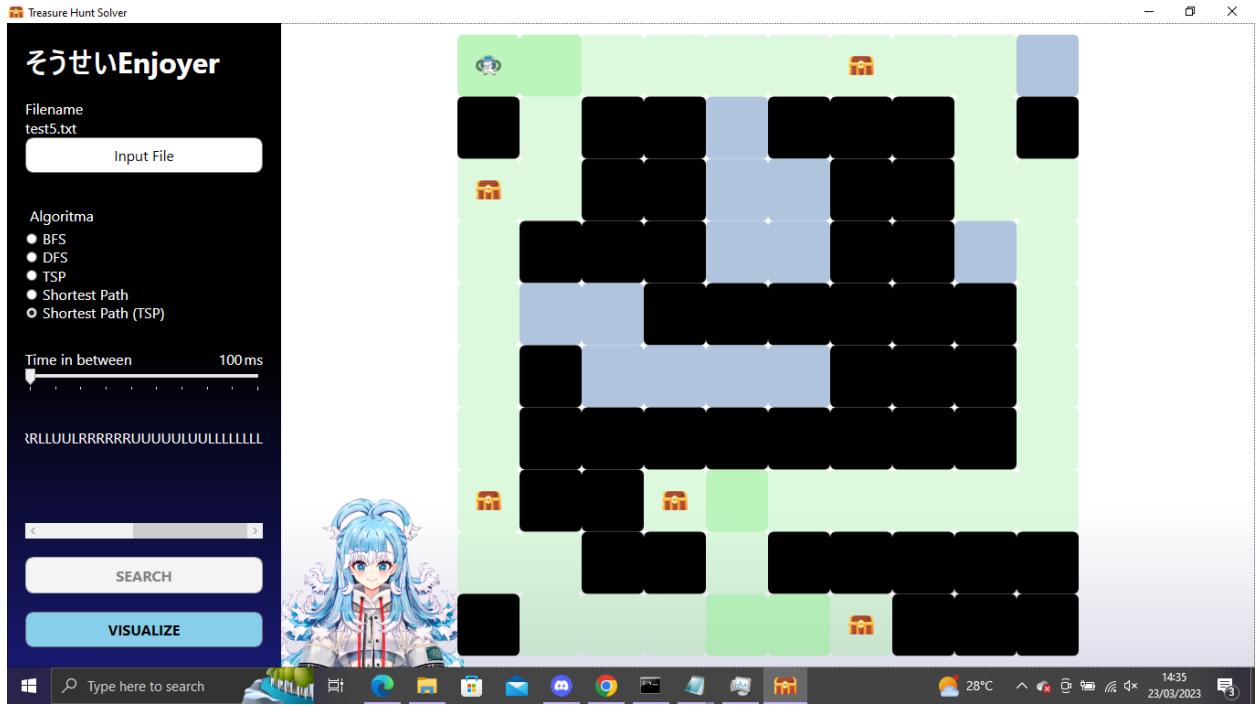
Gambar 4.4.6.5 Tampilan pencarian Shortest Path untuk Test Case 6

e. Shortest Path (TSP) - Eksperimen

Rute yang dihasilkan:

RDDLLLLDDDDDRDRRRRRLUULRRRRRUUUULUULLLLLLL





Gambar 4.4.6.6 Tampilan pencarian Shortest Path (TSP) untuk Test Case 6

#### 4.5 Analisis Desain Solusi

Tabel 4.5.1 Perbandingan Algoritma BFS dan DFS

Test Case (TC)	Elemen	BFS	DFS
1	Rute	RUUDRRD	RUUDRRD
	Jumlah node	11	8
	Jumlah step	7	7
	Waktu eksekusi (ms)	7.0305	5.4026
2	Rute	LRRR	LRRR
	Jumlah node	5	5
	Jumlah step	4	4
	Waktu eksekusi (ms)	6.4401	6.397
4	Rute	DRDRDRDRDR	DRDLDRRUDRDLL LDRRRRUDR
	Jumlah node	21	23

	Jumlah step	10	22
	Waktu eksekusi (ms)	6.4962	6.1995
5	Rute	RDDDDDDDDDDDD	RRDLDRDLDRDLD RDLDRDLD
	Jumlah node	32	22
	Jumlah step	11	21
	Waktu eksekusi (ms)	7.3666	7.1545
6	Rute	RDDLD-DDDDDRD RRRRRLLUULRRR RRRUUUUULUUL L	RRRRRRRRRLDDR DLRDDDDLLLLL RDDLLLULUUUU UURUDLDDRRDR RRLLLULLDDDDR DRRRRR
	Jumlah node	89	70
	Jumlah step	38	69
	Waktu eksekusi (ms)	7.4516	5.8305

Berdasarkan hasil pengujian TC1 dan TC2, algoritma BFS dan DFS menghasilkan rute yang sama. Akan tetapi, dalam kasus TC1, algoritma DFS lebih efisien karena hanya memeriksa 8 node dibandingkan algoritma BFS yang memeriksa 11 node. Hal ini karena dalam algoritma BFS untuk setiap *tile* yang diperiksa, diperiksa pula *tile* tetangganya (yang dapat dilalui) dengan prioritas *up-left-right-down*.

Berdasarkan hasil pengujian TC4, TC5, dan TC6, diperoleh algoritma BFS menghasilkan rute yang lebih pendek dibandingkan algoritma DFS. Hal ini karena algoritma BFS mencari semua *tile* yang dapat diakses dari *tile* awal terlebih dahulu. Algoritma BFS menjamin bahwa jika ada jalur yang lebih pendek menuju *tile* tujuan, maka jalur tersebut akan ditemukan terlebih dahulu. Sedangkan jalur yang dihasilkan algoritma DFS sangat bergantung pada prioritas yang digunakan oleh algoritma tersebut.

Dalam kasus ini, *treasure* yang dicari tidak hanya satu, sehingga algoritma BFS tidak menjamin menghasilkan jalur terpendek untuk mencari seluruh *treasure*. Dapat terlihat pada hasil pengujian TC6, algoritma BFS bukanlah jalur terpendek karena ada jalur yang lebih pendek yang diperoleh menggunakan algoritma *Shortest Path*. Hal ini karena algoritma BFS hanya mencari jalur terpendek dari satu *treasure* ke *treasure* lain.

Berdasarkan seluruh hasil pengujian, dapat disimpulkan bahwa algoritma BFS menghasilkan jalur yang lebih pendek daripada algoritma DFS, walaupun bukan merupakan jalur

yang paling optimal. Akan tetapi, algoritma DFS cenderung lebih efisien dalam memori, terlihat dari jumlah node yang dikunjungi cenderung lebih sedikit tetapi tidak selalu lebih sedikit dibandingkan algoritma BFS.

## **BAB V**

# **KESIMPULAN DAN SARAN**

### **5.1 Kesimpulan**

Baik algoritma BFS maupun DFS dapat digunakan untuk mencari suatu *path* yang melalui semua *treasure* pada permainan *maze* ini. Namun, kedua algoritma tersebut masing-masing memiliki kelebihan dan kekurangan. DFS menggunakan memori yang lebih sedikit dan *node* yang dicek umumnya lebih sedikit namun *path* yang dihasilkan bisa sangat panjang akibat prioritas yang tidak sesuai, sedangkan BFS menggunakan memori yang lebih banyak karena semua kemungkinan *path* dicek setiap langkahnya namun *path* yang dihasilkan umumnya lebih pendek dibandingkan dengan DFS. Namun, BFS sendiri belum tentu selalu menghasilkan *shortest path*, dapat terlihat pada beberapa tes dimana BFS masih menghasilkan *path* yang lebih panjang dibanding *shortest path*.

### **5.2 Saran**

Apabila ada hal penting dalam jawaban QNA, alangkah lebih baik bila hal tersebut ditambahkan dalam spesifikasi karena terkadang ada sesuatu yang terlewat karena hanya dibahas di QNA.

### **5.3 Refleksi**

Algoritma memiliki kelebihan dan kekurangannya sehingga penggunaannya harus disesuaikan dengan masalah yang ingin dipecahkan. Oleh karena itu, kita sebagai *programmer* sebaiknya bisa menentukan apa algoritma yang paling sesuai digunakan ketika kita akan memecahkan masalah.

## **DAFTAR PUSTAKA**

<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf>  
<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf>

## **LAMPIRAN**

Link Github: [https://github.com/Altair1618/Tubes2\\_SouseiEnjoyer](https://github.com/Altair1618/Tubes2_SouseiEnjoyer)

Link Video: <https://youtu.be/y9ON4NPMFBE>