

LAPORAN TUGAS BESAR 2
IF2211 STRATEGI ALGORITMA
Pengaplikasian Algoritma BFS dan DFS dalam
Menyelesaikan Persoalan Maze Treasure Hunt



Kelompok karepmu:

Salomo Reinhart Gregory Manalu (13521063)

Margaretha Olivia Haryono (13521071)

Muchammad Dimas Sakti Widyatmaja (13521160)

Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
2023

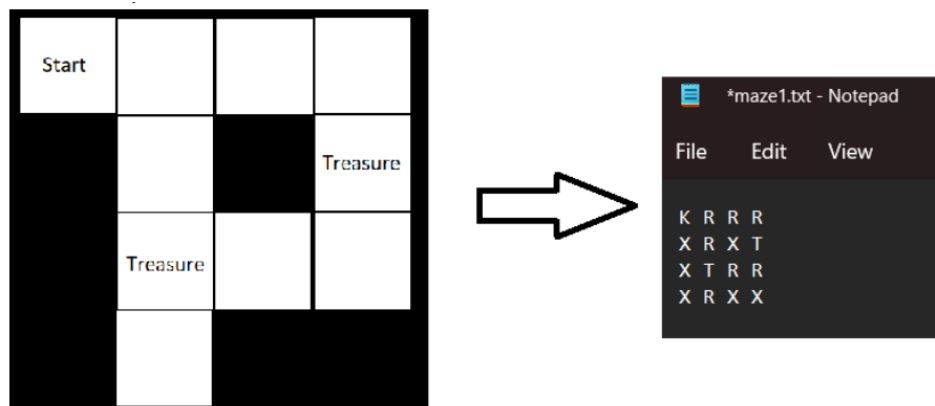
BAB I

DESKRIPSI TUGAS

Dalam tugas besar ini akan dibangun 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

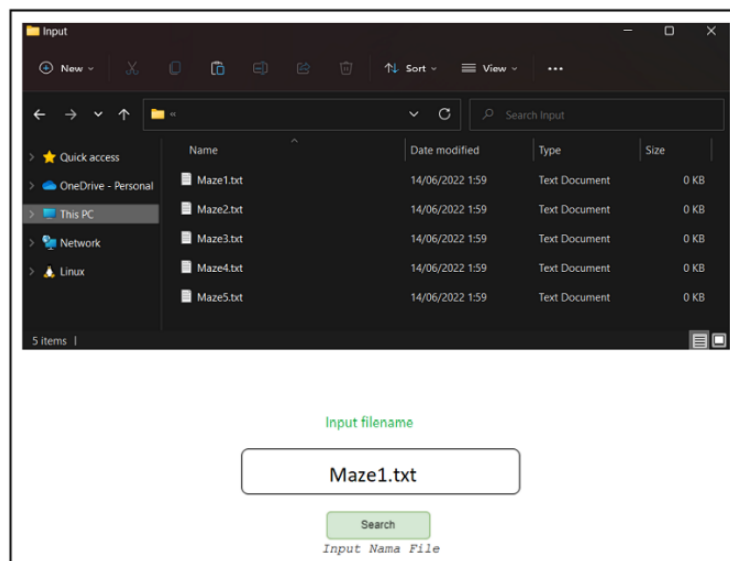
Contoh file input:



Gambar 1.1 Ilustrasi input maze

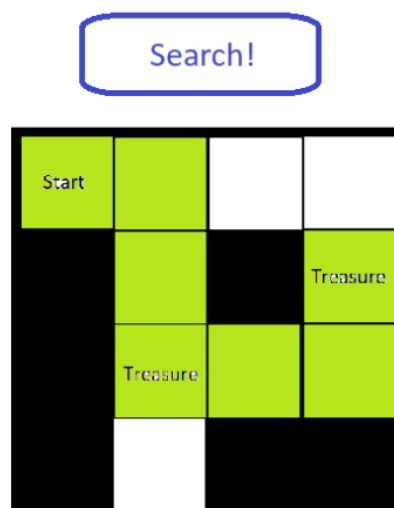
Dengan memanfaatkan algoritma Breadth First Search (BFS) dan Depth First Search (DFS) untuk menelusuri grid (simpul) yang mungkin dikunjungi hingga ditemukan rute solusi, baik secara melebar ataupun mendalam bergantung alternatif algoritma yang dipilih. Rute solusi adalah rute yang memperoleh seluruh 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. Selain itu, terdapat visualisasi dari 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.



Gambar 1.2 Contoh input program

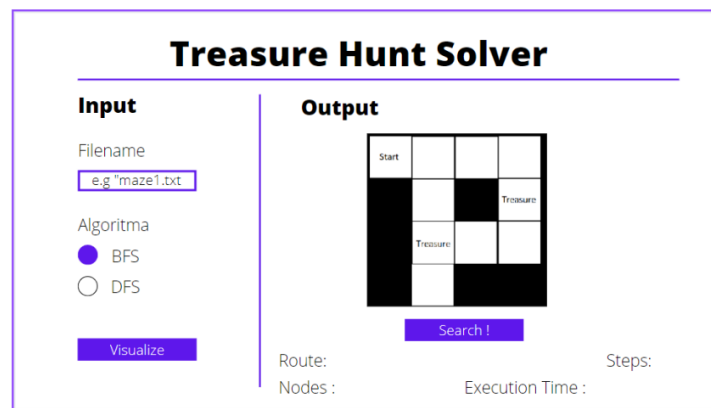
Daftar input maze akan dikemas dalam sebuah folder yang dinamakan test dan terkandung dalam repository program. Folder tersebut akan setara kedudukannya dengan folder src dan doc (struktur folder repository akan dijelaskan lebih lanjut di bagian bawah spesifikasi tubes). Cara input maze boleh langsung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



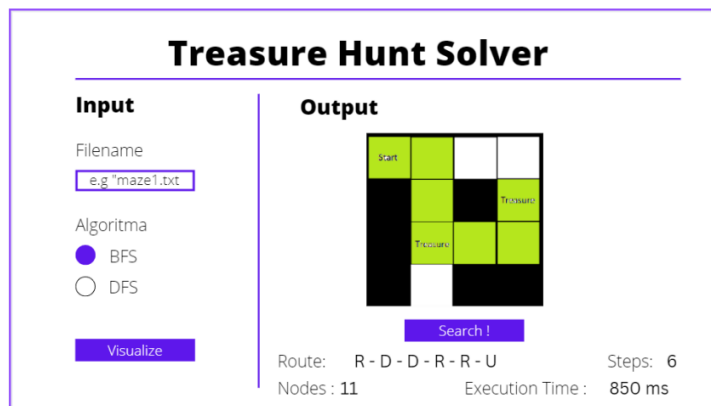
Gambar 1.3 Contoh output program untuk gambar 1.2

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.

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

BAB II

LANDASAN TEORI

A. Dasar Teori

1. Graph Traversal

Graph traversal atau dikenal juga dengan penjelajahan graf mengacu pada proses mengunjungi setiap simpul dalam graf. Algoritma ini mengunjungi simpul dengan cara yang sistematis. Algoritma traversal dalam graf terdiri dari dua, yaitu pencarian melebar (Breadth First Search / BFS) serta pencarian mendalam (Depth First Search / DFS). Graf merepresentasikan persoalan, sedangkan traversal graf merupakan pencarian solusinya, dengan asumsi graf terhubung.

Traversal graf ini merupakan algoritma pencarian solusi dan terbagi menjadi dua jenis, yaitu:

a. Tanpa informasi (*uninformed / blind search*)

Pada jenis ini tidak ada informasi tambahan. Contohnya yaitu DFS, BFS, Depth Limited Search, Iterative Deepening Search, dan Uniform Cost Search.

b. Dengan informasi (*informed Search*)

Pencarian pada jenis ini berbasis heuristik dengan mengetahui non-goal state yang “lebih menjanjikan” daripada yang lain. Contohnya Best First Search dan A*.

Dalam proses pencarian solusi, terdapat dua pendekatan, yaitu:

a. Graf statis, yaitu graf yang sudah terbentuk sebelum proses pencarian dilakukan.

Pada pendekatan ini, graf direpresentasikan sebagai struktur data.

b. Graf dinamis, yaitu graf yang terbentuk saat proses pencarian dilakukan. Pada pendekatan ini, graf tidak tersedia sebelum pencarian, melainkan graf dibangun selama pencarian solusi.

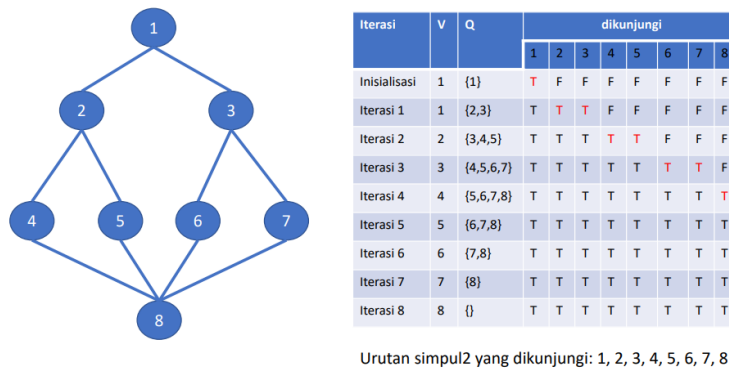
2. Breadth First Search (BFS)

BFS (Breadth-First Search) adalah salah satu algoritma traversal pada grafik atau jaringan yang digunakan untuk mengunjungi setiap simpul dalam grafik secara merata, mulai dari simpul awal dan kemudian menyebar ke simpul-simpul yang bertetangga

dengan simpul tersebut, kemudian ke simpul-simpul yang bertetangga dengan simpul-simpul tersebut, dan seterusnya.

Algoritma pada BFS adalah sebagai berikut:

- Traversal dimulai dengan mengunjungi simpul v.
- Kunjungi semua simpul yang bertetangga dengan simpul v terlebih dahulu.
- Kunjungi simpul yang belum dikunjungi dan bertetangga dengan simpul-simpul yang tadi dikunjungi, demikian seterusnya.



Gambar 2.1 Ilustrasi pencarian dengan BFS

Keuntungan dari BFS adalah setiap simpul akan dikunjungi secara merata, sehingga algoritma ini dapat digunakan untuk memecahkan masalah yang melibatkan pencarian rute atau jalur terpendek. Namun, kerugian dari BFS adalah waktu eksekusi yang mungkin cukup lama jika grafik memiliki banyak simpul atau edge, serta ruang memori yang dibutuhkan untuk menyimpan antrian dapat menjadi masalah jika grafik sangat besar.

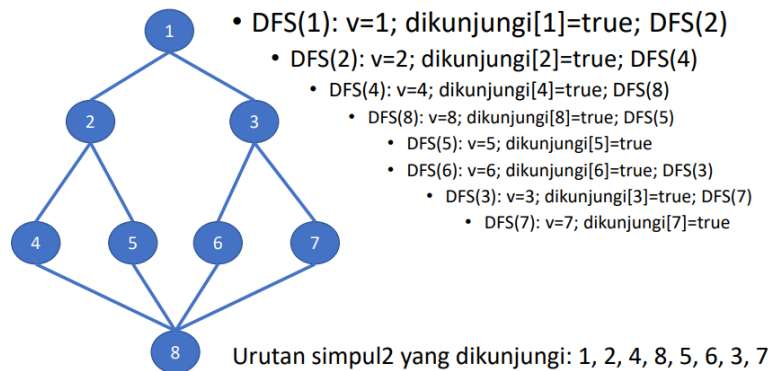
3. Depth First Search (DFS)

DFS (Depth-First Search) adalah salah satu algoritma traversal pada grafik atau jaringan yang digunakan untuk mengunjungi setiap simpul dalam grafik secara terurut dengan cara menjelajahi setiap cabang dari simpul awal terlebih dahulu sebelum kembali ke simpul yang belum terjelajah dan menjelajahi cabang lainnya.

Algoritma pada DFS adalah sebagai berikut:

- Traversal dimulai dengan mengunjungi simpul v.
- Kunjungi simpul w yang bertetangga dengan simpul v.
- Ulangi DFS mulai dari simpul w.

- d. Ketika mencapai simpul u sedemikian sehingga semua simpul yang bertetangga dengannya telah dikunjungi, pencarian dirunut-balik (*backtrack*) ke simpul terakhir yang dikunjungi sebelumnya dan mempunyai simpul w yang belum dikunjungi.
- e. Pencarian berakhir bila tidak ada lagi simpul yang belum dikunjungi yang dapat dicapai dari simpul yang telah dikunjungi.



Gambar 2.2 Ilustrasi pencarian dengan DFS

DFS memiliki kelebihan yaitu dapat bekerja dengan cepat untuk graf yang besar dan kompleks, karena DFS hanya membutuhkan sedikit memori untuk menyimpan *stack* dan tidak memerlukan tambahan antrian seperti algoritma BFS. Namun, kelemahan dari DFS adalah jika digunakan pada grafik yang sangat besar dan kompleks, maka dapat terjebak dalam *loop* atau siklus dan menghabiskan waktu yang lama.

B. C# Desktop Application Development

Pengembangan Aplikasi Desktop C# adalah jenis pengembangan perangkat lunak yang melibatkan pembuatan aplikasi desktop menggunakan bahasa pemrograman C#. Aplikasi desktop merupakan program yang diinstal di komputer pengguna dan dijalankan secara lokal di komputer. Jenis aplikasi ini memberi pengguna antarmuka grafis yang memungkinkan mereka untuk berinteraksi dengan program dan melakukan berbagai tugas.

C# adalah bahasa pemrograman yang populer untuk pengembangan aplikasi desktop karena berorientasi objek, efisien, dan relatif mudah dipelajari. Bahasa ini dikembangkan oleh Microsoft sebagai bagian dari kerangka kerja .NET, yang menyediakan sekumpulan pustaka dan alat untuk membangun aplikasi untuk desktop Windows. Aplikasi desktop C# dapat digunakan

untuk berbagai tujuan, mulai dari aplikasi sederhana yang menjalankan fungsi dasar, seperti kalkulator atau editor teks, hingga aplikasi kompleks seperti sistem perencanaan sumber daya perusahaan, sistem manajemen perawatan kesehatan, atau bahkan permainan video.

Pengembangan Aplikasi Desktop C# adalah cara populer untuk membuat aplikasi perangkat lunak untuk desktop Windows. Dengan desainnya yang efisien dan ramah pengguna, C# memungkinkan developer untuk membuat aplikasi yang kuat dan kaya fitur yang dapat digunakan untuk berbagai tujuan. Dengan mengikuti proses pengembangan terstruktur, developer dapat membangun aplikasi yang dapat memenuhi kebutuhan penggunanya.

Berikut adalah langkah-langkah untuk membuat mengembangkan *desktop application* menggunakan Windows Form App dengan IDE Microsoft Visual Studio.

a. Membuat *project* baru

1. Buka Visual Studio.
2. Pada jendela awal, pilih *Create a new project*.
3. Pilih *Windows Forms App (.Net Framework)*.
4. Pada jendela *Configure your new project*, isi nama *project* pada *Project name*, lalu tekan tombol *Create*.

b. Membuat aplikasi

1. Pilih menu *Toolbox* untuk membuka jendela *Toolbox fly-out*.
2. Pada jendela *Toolbox fly-out* dapat dipilih komponen-komponen yang ingin digunakan pada aplikasi desktop yang sedang dibangun. Komponen juga dapat dimodifikasi dengan mengubah properti pada jendela *properties*.

c. Menambahkan kode ke *form*

1. Pada jendela *Form1.cs [Design]*, tekan dua kali untuk membuka jendela *Form1.cs*.
2. Kode dapat ditulis pada file *Form1.cs*.

d. Menjalankan aplikasi

1. Tekan tombol *Start* pada menu.

BAB III

ANALISIS PEMECAHAN MASALAH

A. Langkah-langkah Pemecahan Masalah

Terdapat dua jenis solusi utama dalam pemecahan masalah yang tersedia yaitu penyelesaian dengan algoritma *breadth-first search* dan penyelesaian dengan algoritma *depth-first search*. Pada algoritma *depth-first search* juga terdapat pilihan untuk mengaktifkan pencarian yang bisa melakukan *multivisit* pada simpul-simpul pada peta.

Dalam menyelesaikan persoalan bonus, pada pilihan penggunaan algoritma *breadth-first search* akan terdapat pilihan untuk menggunakan solusi dari TSP (*Traveling Salesman Problem*) supaya program juga mencari rute untuk kembali ke rute awal.

B. Elemen-elemen Algoritma BFS dan DFS

1. *Breadth-First Search*

Algoritma *breadth-first search* yang kami pakai memanfaatkan struktur data List untuk menyimpan simpul hidup. List ini berperan sebagai *queue* yang menampung simpul hidup dengan mekanisme FIFO (*First In First Out*).

2. *Depth-First Search*

Algoritma *depth-first search* pada penyelesaian permasalahan ini menggunakan pendekatan rekursif, dimana pada setiap langkah yang valid akan memanggil kembali algoritma ini. Mekanisme algoritma DFS memanfaatkan prinsip LIFO (*Last In First Out*). Prioritas arah gerak pada algoritma ini adalah L-R-U-D (*left-right-up-down*). Implementasi algoritma DFS pada program ini terbagi menjadi dua, yaitu:

1. DFS multivisit

Pada algoritma DFS multivisit, setiap kotak dapat dilalui lebih dari satu kali, sehingga proses pencarian secara rekursif akan terus dilakukan hingga ditemukan semua *treasure* yang ada. Pada algoritma ini, tidak ada proses *backtrack* karena tidak ada batasan maksimum berapa kali sebuah kotak dapat dikunjungi, sehingga tidak ada jalan yang buntu.

2. DFS normal

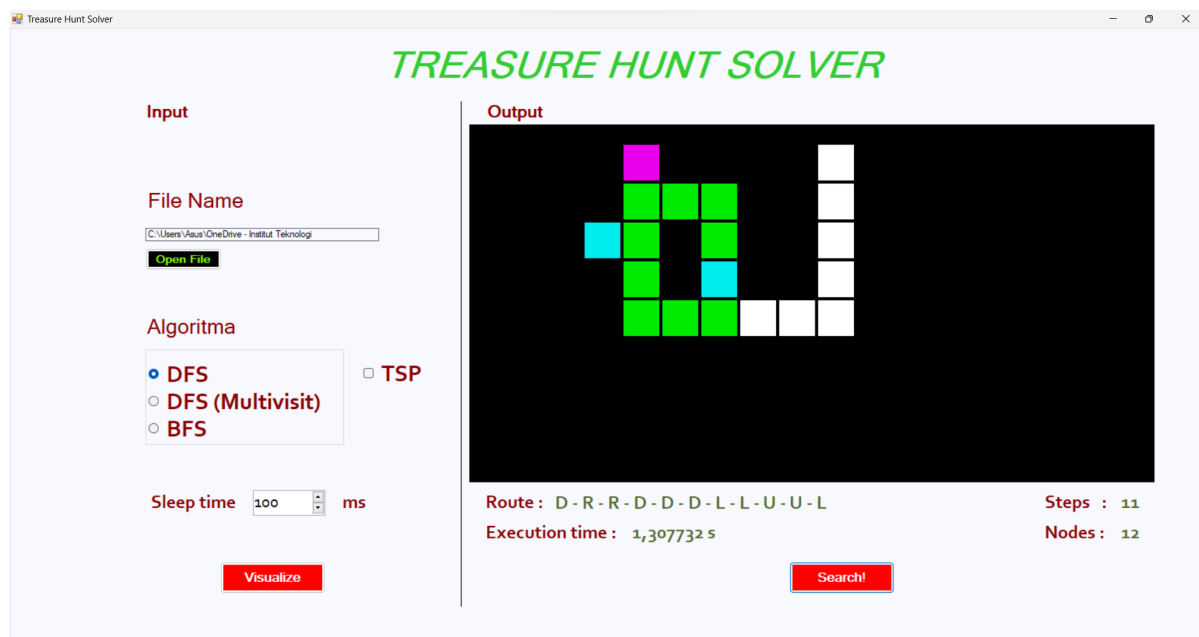
Pada algoritma DFS normal, setiap kotak hanya dapat dikunjungi satu kali, sehingga proses pencarian mendalam hingga mendapati jalan yang buntu. Ketika mencapai kondisi ini, maka dilakukan *backtrack* hingga persimpangan sebelumnya atau mendapatkan alternatif jalur yang lain.

C. Ilustrasi Kasus Lain

Berikut adalah ilustrasi program pada contoh salah satu contoh kasus. Peta yang menjadi masukan pada ilustrasi di bawah ini adalah sebagai berikut.

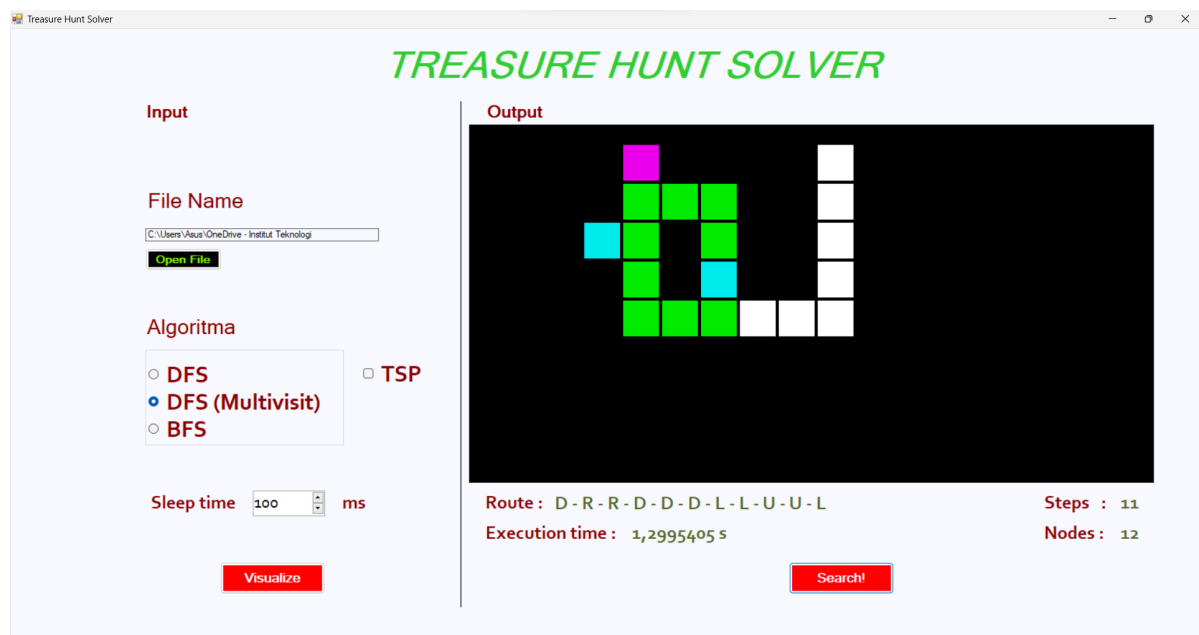
```
X K X X X X R
X R R R X X R
T R X R X X R
X R X T R X R
X R R X R R R
```

Kemudian, masukan tersebut akan di-*parsing* untuk visualisasi menjadi *grid* pada GUI program. Selanjutnya, program melakukan proses pencarian rute solusi sesuai dengan algoritma yang dipilih. Setelah proses pencarian selesai, program menampilkan rute yang ditemukan, waktu eksekusi, jumlah langkah, dan jumlah *nodes* yang dikunjungi. Berikut adalah tampilan dari hasil akhir pencarian pada setiap algoritma yang terdapat pada program.



Gambar 3.1 Ilustrasi pencarian rute dengan DFS (normal)

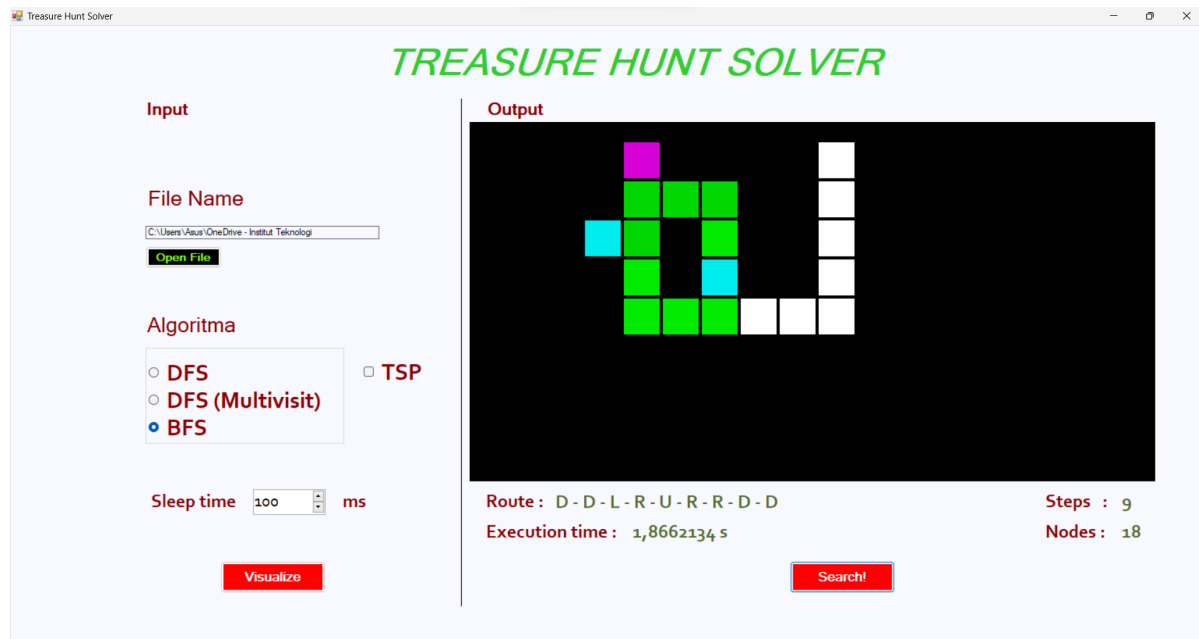
Pada kasus ini, algoritma DFS yang normal tidak akan mengunjungi *grid* lebih dari satu kali. Hal ini memungkinkan tidak ditemukannya rute yang mengarah kepada semua *treasure*. Rute pencarian DFS normal yang dihasilkan berdasarkan peta tersebut adalah sebagai berikut: D-R-R-D-D-D-L-L-U-U-L. Rute ini didapat dari *start grid* yang berada pada koordinat (2, 1). Dari posisi awal tersebut, *solver* akan bergerak sesuai dengan prioritas gerakan yang dipilih oleh kami yaitu LRUD (*Left, Right, Up, Down*). Berdasarkan prioritas tersebut, *solver* bergerak menuju *treasure* yang berada di sebelah kanan terlebih dahulu dengan langkah D-R-R-D-D-D. Setelah itu, *solver* akan bergerak ke *treasure* di sebelah kiri dengan langkah L-L-U-U-L. Rute tersebut ditempuh tanpa mengunjungi *grid* yang pernah dikunjungi sebelumnya. *Solver* baru akan mengunjungi *grid* yang telah dikunjungi hanya apabila sudah tidak ada langkah lain yang bisa diambil oleh *solver*.



Gambar 3.2 Ilustrasi pencarian rute dengan DFS (*multivisit*)

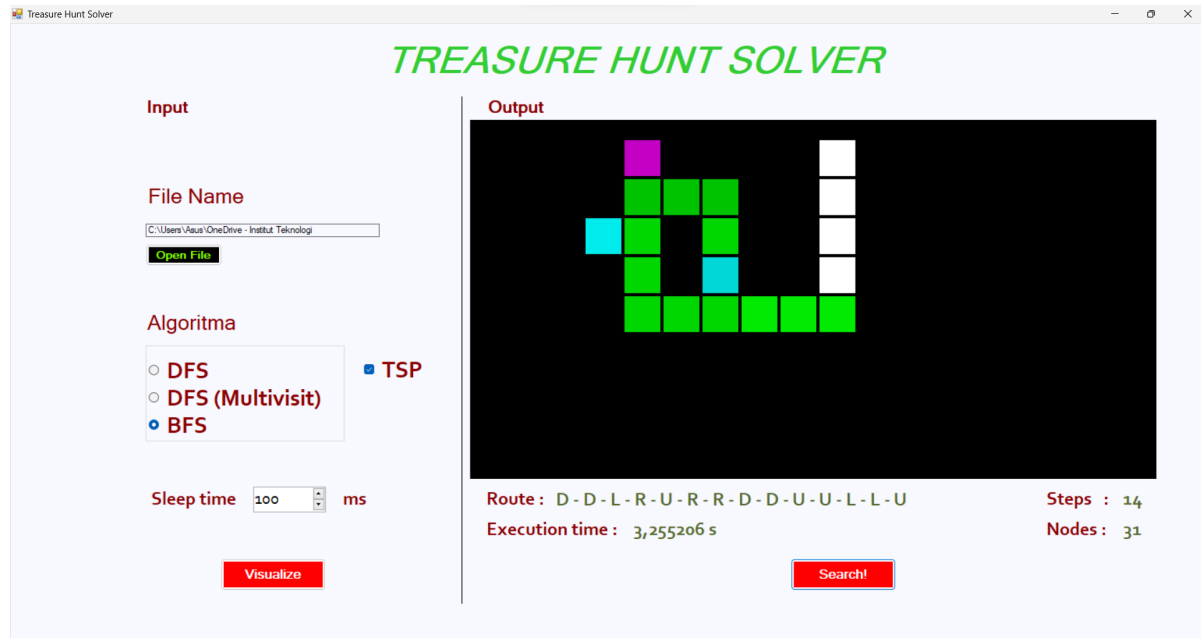
Pada algoritma DFS *multivisit*, secara kebetulan solusi yang dihasilkan sama dengan solusi dari DFS normal, yaitu: D-R-R-D-D-D-L-L-U-U-L. Hal ini dapat dikarenakan prioritas pergerakan yaitu LRUD. Perbedaan mendasar yang terdapat pada DFS *multivisit* dengan DFS normal terletak pada kebolehan untuk mengunjungi suatu *grid* lebih dari sekali. Pada DFS normal mengunjungi suatu *grid* lebih dari sekali tidak diperbolehkan, sehingga

terdapat fitur *backtrack* pada DFS normal. Sebaliknya, pada DFS multivisit tidak terdapat *backtrack* karena setiap *grid* dapat dikunjungi lebih dari sekali.



Gambar 3.3 Ilustrasi pencarian rute dengan BFS

Sedangkan pada algoritma BFS, pencarian dilakukan secara melebar. Akan tetapi, berdasarkan heuristik kami, *solver* akan memulai ulang rute pencarian dimulai dari *treasure* yang ditemukan. Pada kasus tersebut, setelah *solver* berada pada *treasure* yang berada di sebelah kiri, *queue* rute yang telah direncanakan sebelumnya dikosongkan dan BFS akan dijalankan dengan *start grid* diubah menjadi posisi *treasure* sekarang. Setelah melakukan pencarian, rute akhir dari BFS yang mengunjungi semua *treasure* dan tersambung dari titik awal yaitu sebagai berikut: D-D-L-R-U-R-R-D-D. Rute ini cukup pendek apabila dibandingkan dengan rute yang diambil oleh DFS normal maupun *multivisit*. Akan tetapi, jumlah *node* yang dikunjungi lebih banyak, sehingga proses pencarian *treasure* akan memakan banyak memori, namun akan mendapatkan rute yang paling efisien.



Gambar 3.4 Ilustrasi pencarian rute dengan BFS dan kembali ke titik awal (TSP)

Algoritma BFS juga memiliki fitur tambahan yakni TSP. Setelah *solver* mengunjungi semua *treasure*, Ia akan kembali ke posisi awal dengan algoritma BFS juga. Jadi, setelah *solver* mengunjungi *treasure* ke-dua di sebelah kanan dengan rute D-D-L-R-U-R-R-D-D, *solver* akan mencari jalan untuk kembali ke posisi awal yakni (2, 1). Rute kembali yang didapatkan dengan BFS yakni U-U-L-L-U. Jadi rute penuh dari BFS ditambah TSP untuk kembali ke posisi awal dari *treasure* terakhir adalah D-D-L-R-U-R-R-D-D-U-U-L-L-U.

BAB IV

IMPLEMENTASI DAN PENGUJIAN

A. Pseudocode Program

1. Breadth-First Search

```
procedure Solve(Tuple<int, int> startNode, ref string[] treasurePath, ref List<Tuple<int, int>> simpulHidup, bool
tsp, int sleepTime)
    int startRow ← simpulHidup[0].Item1
    int startCol ← simpulHidup[0].Item2

    simpulHidup.RemoveAt(0)

    if (pathCount == 1) then
        return

    nodeCount++

    { Check if current cell is a goal state }
    if (IsGoalState(startRow, startCol)) then
        treasurePath[numGoalsVisited] ← nodePath[startRow, startCol]
        { Increment number of goals visited and add path to current goal state to paths list }
        numGoalsVisited++

        goalStates.Remove(new Tuple<int,int> (startRow, startCol))

        { Mengubah semua node menjadi belum dikunjungi }
        for i = 1 To numRows
            for j = 1 To numCols
                visited[i, j] ← false
                nodePath[i, j] ← ""
            Endfor
        Endfor

        pathCount ← 0
        simpulHidup.Clear()
        simpulHidup.Add(new Tuple<int, int>(startRow, startCol))
        Solve(startNode, ref treasurePath, ref simpulHidup, tsp, sleepTime)

        { Mark current cell as visited }
        visited[startRow, startCol] ← true
        visitCount[startRow, startCol]++
        Form1.outputRoute(startRow, startCol, visitCount[startRow, startCol], sleepTime)
        Console.WriteLine("currentpath " + path)

        { Check if all goal states have been visited }
        if (goalStates.Count == 0) then
            path ← ""
```

```

for i = 1 To numGoalsVisited
    path ← path + treasurePath[i]
Endfor
Console.WriteLine("Path that visited all goal states: " + path)

if (tsp) then
    goalStates.Add(startNode)

    { Mengubah semua node menjadi belum dikunjungi }
    for i = 1 To numRows
        for j = 1 To numCols
            visited[i, j] ← false
            nodePath[i, j] ← ""
        Endfor
    Endfor

    pathCount ← 0
    numGoalsVisited ← 0
    simpulHidup.Clear()
    simpulHidup.Add(new Tuple<int, int>(startRow, startCol))
    Solve(startNode, ref treasurePath, ref simpulHidup, tsp, sleepTime)

else
    pathCount ← pathCount + 1
    return

    { Check all possible moves from current cell }
    if (CanMove(simpulHidup, startRow, startCol - 1)) then { move left }
        Console.WriteLine(startRow)
        Console.WriteLine(" ")
        Console.WriteLine(startCol - 1)
        nodePath[startRow, startCol - 1] ← nodePath[startRow, startCol] + "L"
        simpulHidup.Add(new Tuple<int, int>(startRow, startCol - 1))

    if (CanMove(simpulHidup, startRow, startCol + 1)) then { move right }
        Console.WriteLine(startRow)
        Console.WriteLine(" ")
        Console.WriteLine(startCol + 1)
        nodePath[startRow, startCol + 1] ← nodePath[startRow, startCol] + "R"
        simpulHidup.Add(new Tuple<int, int>(startRow, startCol + 1))

    if (CanMove(simpulHidup, startRow - 1, startCol)) then { move up }
        Console.WriteLine(startRow - 1)
        Console.WriteLine(" ")
        Console.WriteLine(startCol)
        nodePath[startRow - 1, startCol] ← nodePath[startRow, startCol] + "U"
        simpulHidup.Add(new Tuple<int, int>(startRow - 1, startCol))

    if (CanMove(simpulHidup, startRow + 1, startCol)) then { move down }
        Console.WriteLine(startRow + 1)
        Console.WriteLine(" ")
        Console.WriteLine(startCol)
        nodePath[startRow + 1, startCol] ← nodePath[startRow, startCol] + "D"
        simpulHidup.Add(new Tuple<int, int>(startRow + 1, startCol))
    Solve(startNode, treasurePath, simpulHidup, tsp, sleepTime)

```

```

function IsGoalState(int row, int col) -> boolean
    foreach (Tuple<int, int> goalState in goalStates)
        if (row == goalState.Item1 && col == goalState.Item2) then
            return true
    return false

function CanMove(List<Tuple<int,int>> simpulHidup,int row, int col) -> boolean
    { Check if cell is within maze boundaries or have been added to simpulHidup }
    if (row < 0 || row >= numRows || col < 0 || col >= numCols || simpulHidup.Any(s => s.Item1 == row &&
s.Item2 == col)) then
        return false

    { Check if cell is not a wall and has not been visited }
    if (maze[row, col] == 1 || visited[row, col]) then
        return false
    return true

```

2. Depth-First Search

a. DFS multivisit

```

procedure SolveMultivisit(int startRow, int startCol, string path, int sleepTime)
    { if already found the path, stop the searching }
    if (isFound) then
        return

    { increment visitCount for current cell }
    visitCount[startRow, startCol]++
    nodeCount++

    { Check if current cell is a goal state }
    if (IsGoalState(startRow, startCol)) then
        if (not goalsVisited.Contains(new Tuple<int, int>(startRow, startCol))) then
            { Increment number of goals visited and add path to current goal state to paths list }
            numGoalsVisited++
            goalsVisited.Add(new Tuple<int, int>(startRow, startCol))

    { Check if all goal states have been visited }
    if (numGoalsVisited == goalStates.Count) then
        this.path <- path
        isFound <- true { path is found }
        return { stop the searching }

    { heuristic method: compare visitCount for all valid adjacent cells }
    List<Tuple<char, int, int, int>> precedence -> new List<Tuple<char, int, int, int>> { }

    if (isValidCell(startRow + 1, startCol)) then
        precedence.Add(new Tuple('D', visitCount[startRow + 1, startCol], startRow + 1, startCol))
    if (isValidCell(startRow - 1, startCol)) then
        precedence.Add(new Tuple('U', visitCount[startRow - 1, startCol], startRow - 1, startCol))
    if (isValidCell(startRow, startCol + 1)) then

```



```

    precedence.Add(new Tuple('R', visitCount[startRow, startCol + 1], startRow, startCol + 1))
if (isValidCell(startRow, startCol - 1)) then
    precedence.Add(new Tuple('L', visitCount[startRow, startCol - 1], startRow, startCol - 1))

{ order by visitCount (ascending) }
precedence <- precedence.OrderByDescending(t => t.Item2).ToList()

{ solve for every valid adjacent cell (order by ascending visitCount) }
for i = precedence.Count To 1
    SolveMultivisit(precedence[i].Item3, precedence[i].Item4, path + precedence[i].Item1, sleepTime)
    { if already found the path, stop the searching }
    if (isFound) then
        return
return

{ check if a cell is a valid cell (for multivisit dfs) }
function isValidCell(int row, int col) -> boolean
    { Check if cell is within maze boundaries }
    if (row < 0 or row >= numRows or col < 0 or col >= numCols) then
        <- false
    { Check if cell is not a wall and not a start state }
    if (maze[row, col] = 1 or IsStartState(row, col)) then
        return false
    return true

```

b. DFS normal

```

procedure Solve(int startRow, int startCol, string path, int sleepTime)
    { if already found the path, stop the searching }
    if (isFound) then
        return

    { Mark current cell as visited }
    visited[startRow, startCol] -> true
    if (IsStartState(startRow, startCol)) then
        startCount++

    nodeCount++

    { Check if current cell is a goal state }
    if (IsGoalState(startRow, startCol)) then
        { Increment number of goals visited and add path to current goal state to paths list }
        numGoalsVisited++

    { Check if all goal states have been visited }
    if (numGoalsVisited = goalStates.Count) then
        this.path <- path
        isFound <- true
        return

    { Check all possible moves from current cell }
    if (CanMove(startRow, startCol - 1)) then

```

```

    Solve(startRow, startCol - 1, path + "L", sleepTime)
    if (isFound) then
        return

    if (CanMove(startRow, startCol + 1)) then
        Solve(startRow, startCol + 1, path + "R", sleepTime)
        if (isFound) then
            return

    if (CanMove(startRow - 1, startCol)) then
        Solve(startRow - 1, startCol, path + "U", sleepTime)
        if (isFound) then
            return

    if (CanMove(startRow + 1, startCol)) then
        Solve(startRow + 1, startCol, path + "D", sleepTime)
        if (isFound) then
            return

    { BACKTRACKING }

    { Mark current cell as unvisited }
    visited[startRow, startCol] -> false
    nodeCount++

    { Decrement number of goals visited (if necessary) }
    if (IsGoalState(startRow, startCol)) then
        numGoalsVisited--
    return

{ check if a cell is a valid cell (for normal dfs) }
function CanMove(int row, int col) -> boolean
    { Check if cell is within maze boundaries }
    if (row < 0 or row >= numRows or col < 0 or col >= numCols) then
        <- false
    { Check if cell is not a wall and has not been visited }
    if (maze[row, col] = 1 or visited[row, col]) then
        if (IsStartState(row, col) and startCount != 2 and numGoalsVisited = goalStates.Count - 1) then
            return true
        return false
    return true

{ check if a cell is a goal state }
function IsGoalState(int row, int col) -> boolean
    foreach (Tuple<int, int> goalState in goalStates)
        if (row = goalState.Item1 and col = goalState.Item2) then
            return true
    return false

{ check if a cell is a start state }
function IsStartState(int row, int col) -> boolean
    return (row = startState[0] and col = startState[1])

```

B. Struktur Data dan Spesifikasi Program

Berikut ini merupakan beberapa struktur data yang digunakan dalam tugas besar ini :

1. Matrix

Implementasi struktur data matriks digunakan untuk menyimpan *maze* atau *map* yang berasal dari input data oleh pengguna. Terdapat dua jenis matriks yang digunakan dalam program ini, yaitu *matrix of char* dan *matrix of int*. Dengan menyimpan *map* ke dalam matriks, program dapat mengakses tiap titik atau node dalam *map/maze*.

Selain untuk menyimpan *map*, struktur data matriks juga digunakan dalam menghitung berapa kali sebuah *grid* pada *map* telah dikunjungi. Hal ini digunakan untuk keperluan visualisasi setiap langkah pada proses pencarian solusi. Semakin sering suatu *grid* dikunjungi, maka warna dari *grid* tersebut akan semakin gelap.

2. List

Implementasi struktur data List digunakan untuk proses pencarian dengan algoritma BFS dan DFS. Kami mengimplementasi struktur data List menggunakan konsep operasi Queue pada BFS dan konsep operasi Stack pada DFS.

a. List dengan operasi Queue

Pada algoritma BFS, operasi pada List dilakukan seperti pada konsep operasi Stack, yaitu FIFO (*First In First Out*). Struktur data ini digunakan untuk menyimpan node yang akan dikunjungi selama program berjalan. List menjadi tipe data yang cocok untuk hal ini karena memiliki banyak method yang sudah tersedia untuk penyuntingan isi List.

b. List dengan operasi Stack

Pada algoritma DFS, operasi pada List dilakukan seperti pada konsep operasi Stack, yaitu LIFO (*Last In First Out*). Struktur data ini digunakan untuk menyimpan tumpukan *grid* yang akan dikunjungi oleh program selama melakukan proses pencarian. Hal ini dilakukan agar program mencari solusi secara mendalam terlebih dahulu, kemudian melakukan *backtrack* ketika tidak ada *grid* yang dapat dilalui lagi.

C. Tata Cara Penggunaan Program

Untuk menjalankan program, pengguna dapat menjalankan file “Tubes2_karepmu.exe”. Kemudian, akan terbuka *window* aplikasi yang menampilkan GUI dari program. Untuk menggunakannya, pengguna dapat memasukkan *file* peta *maze treasure hunt* dengan menekan tombol “Enter File”. Kemudian, pengguna juga harus memilih salah satu *option button* untuk jenis algoritma yang ingin digunakan. Pilihan algoritma yang ada yaitu DFS, DFS (multivisit), dan BFS. Terdapat juga tambahan *checkboxlist* untuk TSP jika ingin solusi yang dihasilkan kembali ke titik awal. Selain itu, terdapat juga masukan untuk durasi jeda setiap langkah pada pencarian dengan durasi minimal 0 ms dan maksimal 5000 ms dengan interval 500 ms.

Untuk menampilkan peta *maze* yang telah dipilih, pengguna dapat menekan tombol “Visualize”. Jika masukan *file maze* tidak valid, maka program akan menampilkan pesan error pada layar. Jika masukan telah sesuai, maka akan terlihat visualisasi dari peta dengan keterangan warna sebagai berikut.

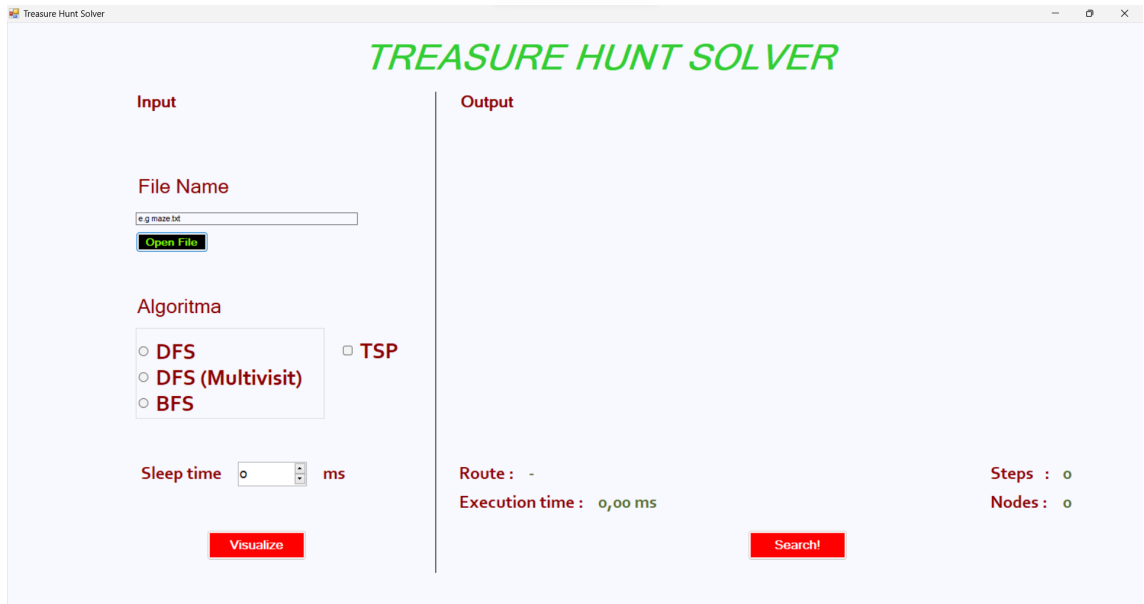
K (titik awal)	: merah
T (treasure)	: ungu
R (lintasan)	: putih
X (halangan)	: hitam

Setelah mengatur semua masukan yang diperlukan, pengguna dapat menekan tombol “Search” untuk memulai proses pencarian pada *maze* sesuai algoritma yang dipilih dengan jeda waktu tiap langkah sesuai masukan sebelumnya. Pada visualisasi langkah, terdapat beberapa warna yang kami gunakan sebagai berikut.

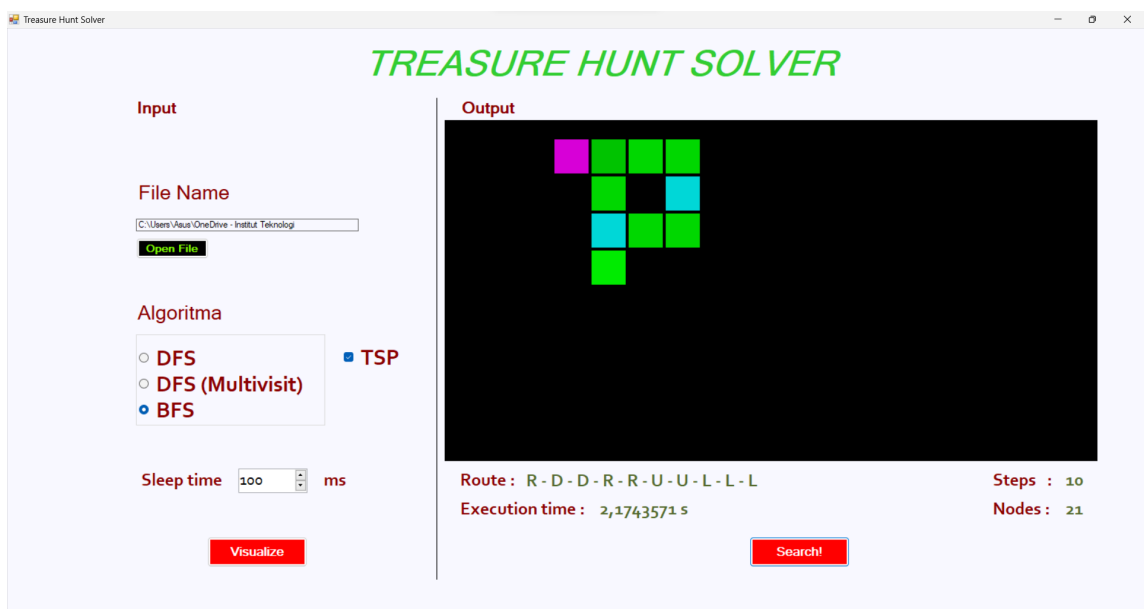
lintasan (R) yang telah dikunjungi	: hijau
<i>treasure</i> (T) yang telah dikunjungi	: biru
titik awal (K) telah dikunjungi	: ungu muda

Pada kasus multivisit, semakin sering suatu *grid* dikunjungi, maka warna dari *grid* tersebut akan semakin gelap.

Setelah proses pencarian selesai, ditampilkan rute pencariannya beserta waktu eksekusi, jumlah langkah, dan jumlah *nodes* yang dikunjungi selama proses pencarian. Berikut adalah tampilan awal serta akhir dari program yang kami buat.



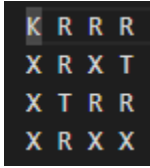
Gambar 4.1 Tampilan awal program



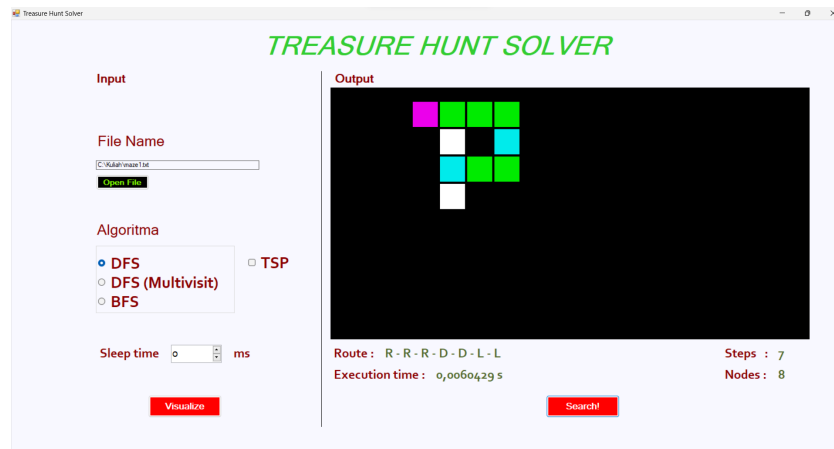
Gambar 4.2 Tampilan akhir program

D. Hasil Pengujian

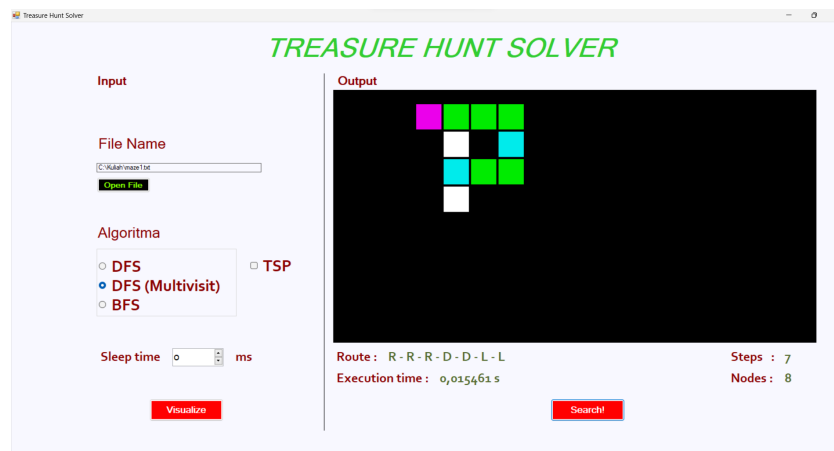
1. Uji Kasus 1



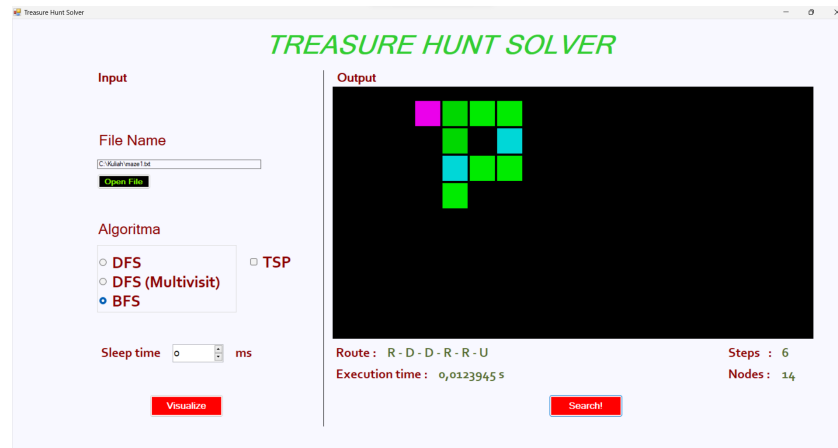
a. DFS



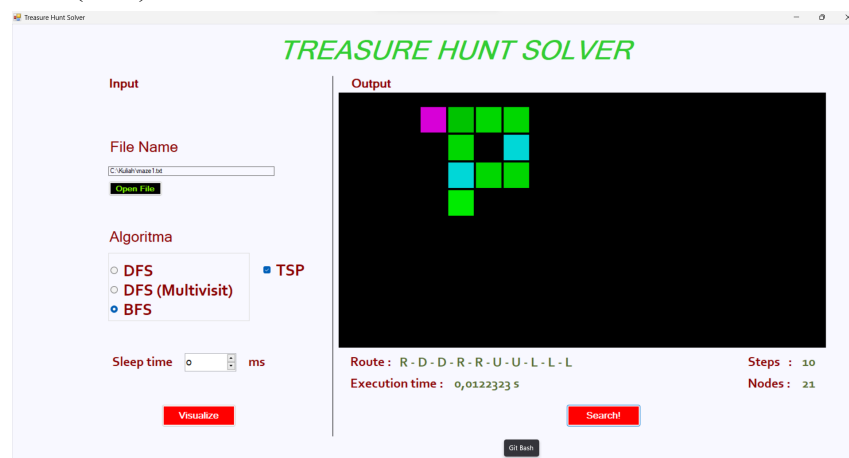
b. DFS Multivisit



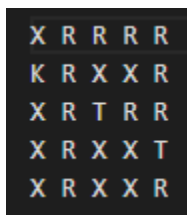
c. BFS



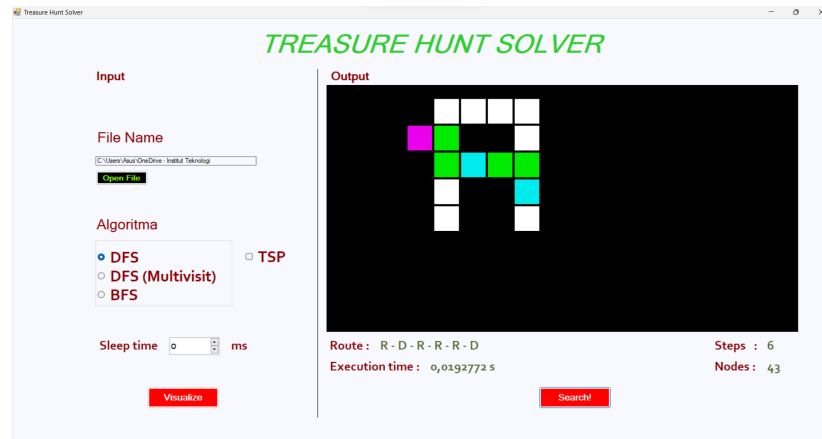
d. BFS (TSP)



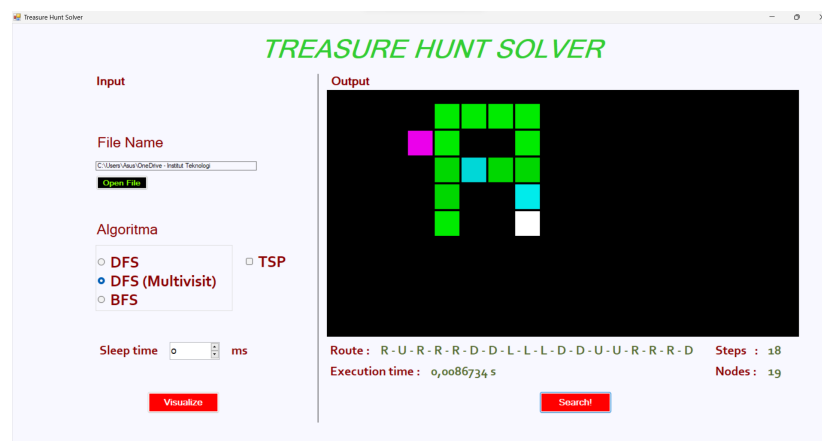
2. Uji Kasus 2



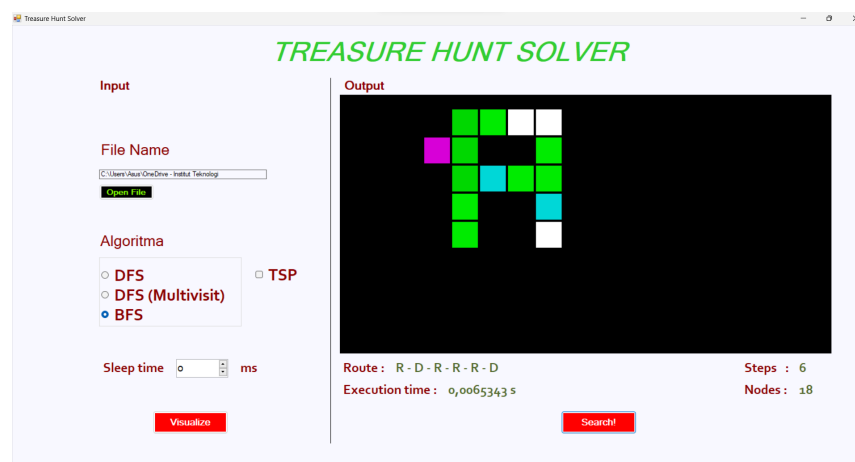
a. DFS



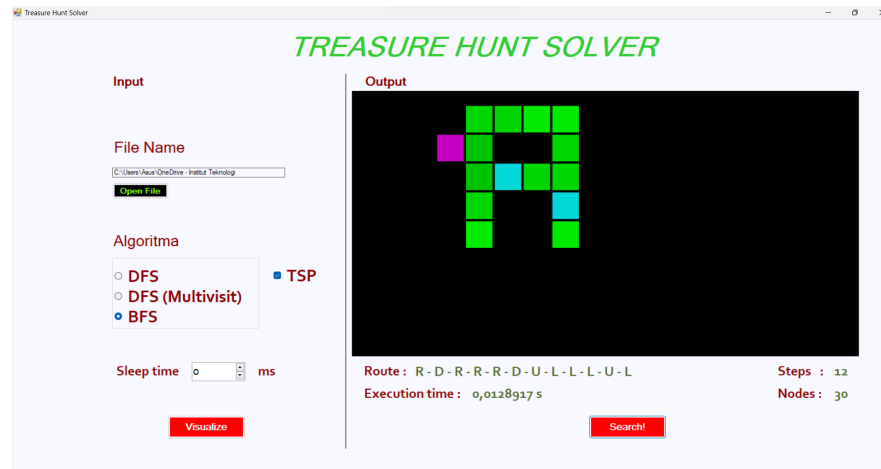
b. DFS Multivisit



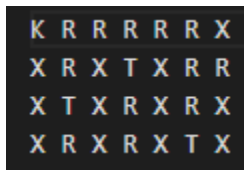
c. BFS



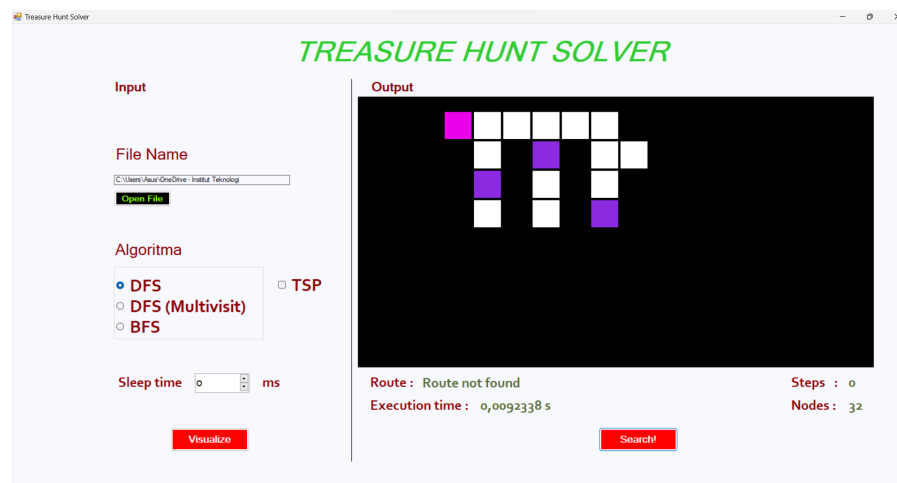
d. BFS (TSP)



3. Uji Kasus 3



a. DFS



b. DFS Multivisit

TREASURE HUNT SOLVER

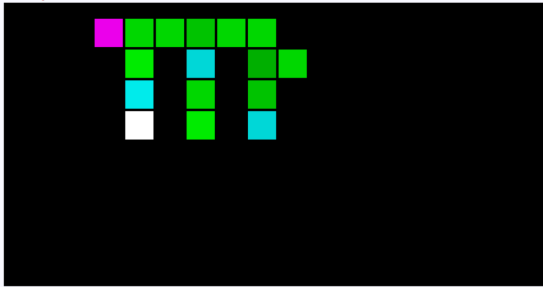
Input

File Name

Algoritma
☐ DFS
☒ DFS (Multivisit)
☐ BFS
☐ TSP

Sleep time ms

Output



Route : R-R-R-R-R-D-D-L-D-D-U-D-U-U-R-L-U-L-L-D-D-D-U-U-U-L-L-D-D
 Execution time : 0,0163095 s
 Steps : 29
 Nodes : 30

c. BFS

TREASURE HUNT SOLVER

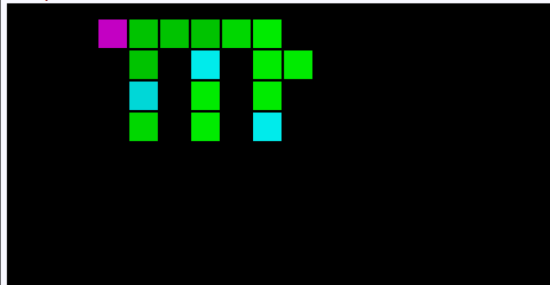
Input

File Name

Algoritma
☐ DFS
☐ DFS (Multivisit)
☒ BFS
☐ TSP

Sleep time ms

Output



Route : R-D-D-U-U-R-R-D-U-R-R-D-D-D
 Execution time : 0,0137884 s
 Steps : 14
 Nodes : 31

d. BFS (TSP)

TREASURE HUNT SOLVER

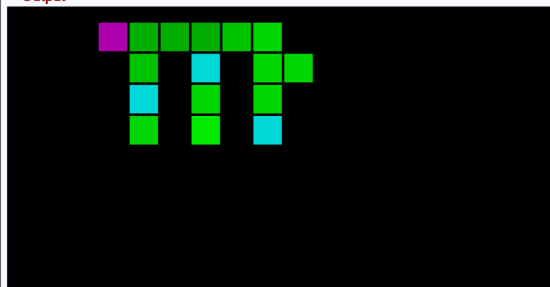
Input

File Name

Algoritma
☐ DFS
☐ DFS (Multivisit)
☒ BFS
☒ TSP

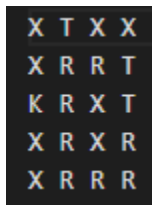
Sleep time ms

Output

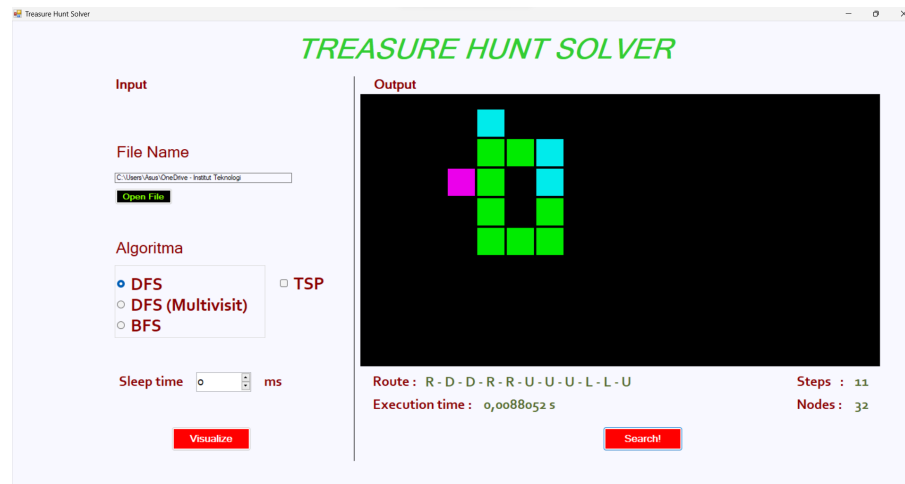


Route : R-D-D-U-U-R-R-D-U-R-R-D-D-D-U-U-U-L-L-L-L-L
 Execution time : 0,0095294 s
 Steps : 22
 Nodes : 43

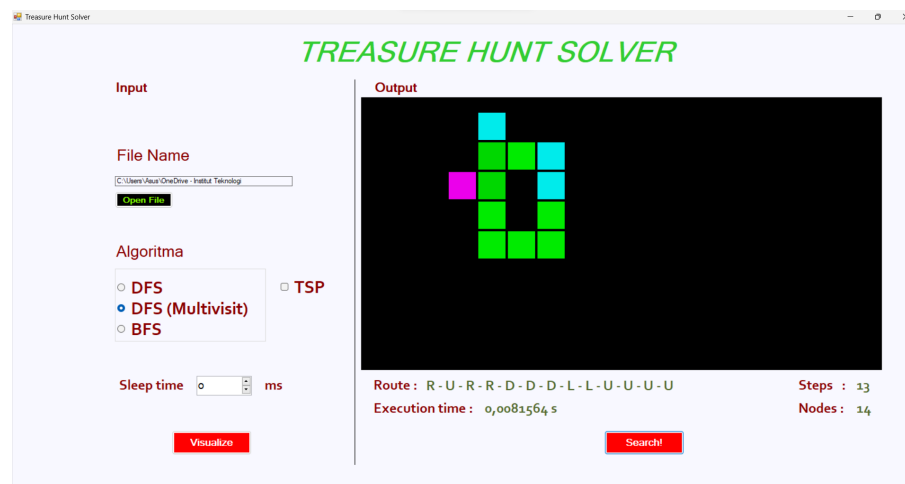
4. Uji Kasus 4



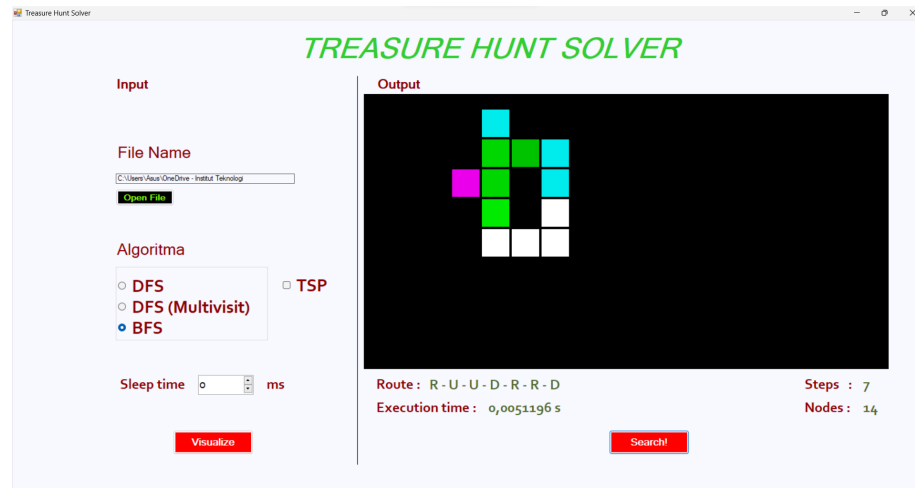
a. DFS



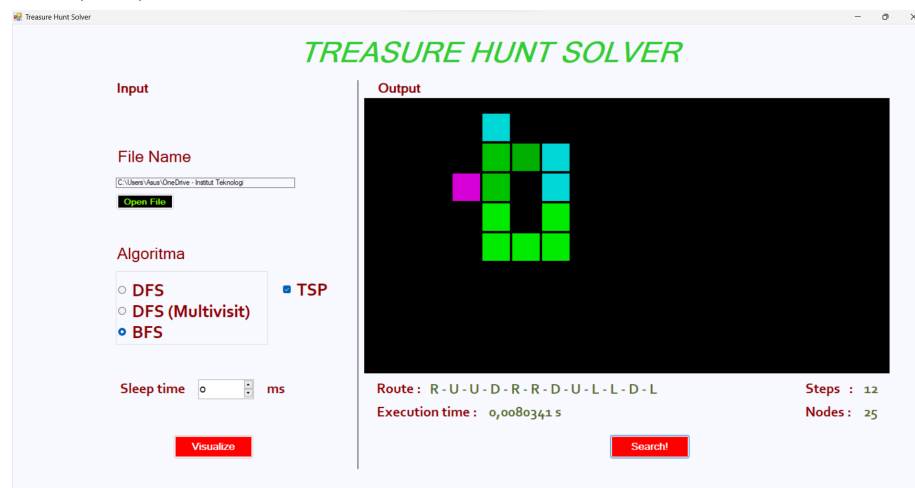
b. DFS Multivisit



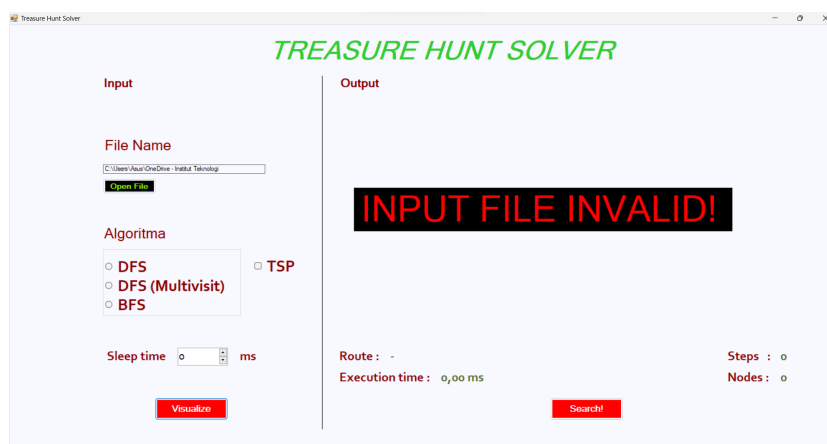
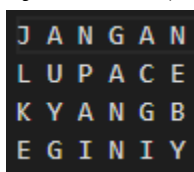
c. BFS



d. BFS (TSP)



5. Uji Kasus 5 (Input Invalid)



E. Analisis Desain Solusi Algoritma BFS dan DFS

Pada *test case 3*, algoritma DFS yang normal tidak memungkinkan sebuah *grid* dikunjungi lebih dari satu kali. Sehingga, tidak ditemukan rute solusi yang dapat mencapai semua *treasure* pada *map* tersebut. Rute pencarian DFS normal pada peta tersebut adalah sebagai berikut: R-R-R-R-R-D-D-D-U-U-R-L-U-L-L-D-D-D-U-U-U-L-L-D-D-D-U-U-U-L

Pada algoritma DFS multivisit, didapati sebuah solusi untuk mencapai semua *treasure*. Rute pencarian DFS multivisit pada peta tersebut adalah sebagai berikut:
R-R-R-R-R-D-R-L-D-D-U-D-U-U-R-L-U-L-L-D-D-D-U-U-U-L-L-D-D

Sedangkan pada algoritma BFS, pencarian dilakukan secara melebar. Setelah melakukan pencarian, rute akhir dari BFS yang mengunjungi semua *treasure* dan tersambung dari titik awal yaitu sebagai berikut: R-D-D-U-U-R-R-D-U-R-R-D-D-D

Dari uji kasus tersebut dan juga uji kasus lainnya pada bagian sebelumnya, dapat dilihat bahwa algoritma BFS selalu mendapati rute dengan jumlah langkah yang paling sedikit. Hal ini menunjukkan bahwa BFS selalu mendapatkan rute yang paling efektif. Tetapi, jumlah *nodes* yang dikunjungi cenderung lebih banyak dibandingkan algoritma DFS yang menunjukkan bahwa BFS menggunakan lebih banyak memori dalam proses pencariannya. Namun demikian, algoritma DFS tidak selalu menghasilkan rute solusi yang paling efektif.

BAB V

KESIMPULAN, DAN SARAN

A. Kesimpulan

Algoritma *breath-first search* dan *depth-first search* dapat digunakan untuk menyelesaikan permasalahan *Maze Treasure Hunt*. Algoritma *breath-first search* melakukan pencarian jalur secara melebar atau tersebar untuk setiap persimpangan yang ada, sedangkan algoritma *depth-first search* melakukan pencarian jalur secara mendalam, yaitu hingga menemukan semua *treasure* yang ada (untuk kasus multivisit) atau hingga tidak ada jalan yang dapat dikunjungi lagi dan kemudian melakukan *backtrack* hingga persimpangan sebelumnya.

Pengembangan *desktop application* menggunakan bahasa pemrograman C# dapat dilakukan menggunakan Visual Studio. Pengembangan GUI dapat dilakukan dengan lebih mudah menggunakan WinForm.

B. Saran

Pengembangan *desktop application* menggunakan WinForm hanya dapat membuat *desktop application* dengan sistem operasi berbasis Windows, sehingga tidak dapat dijalankan pada multiplatform. Saran untuk pengembangan aplikasi selanjutnya yaitu dapat menggunakan teknologi yang berbeda dan yang mendukung pengembangan aplikasi multiplatform sehingga dapat membuat *desktop application* yang lebih dinamik.

C. Refleksi dan Tanggapan

Melalui tugas besar ini, kami menjadi lebih memahami mengenai implementasi algoritma *breath-first search* serta *depth-first search* dalam mencari rute solusi *Maze Treasure Hunt*. Banyak tantangan yang kami hadapi, seperti mempelajari bahasa pemrograman baru (C#) serta perangkat lunak / IDE (Visual Studio) yang belum pernah kami gunakan sebelumnya. Namun dengan adanya tugas ini, kami dapat memperluas wawasan kami serta menambah pengalaman dalam mengembangkan *desktop application* dengan GUI.

BAB VI

DAFTAR PUSTAKA

1. <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf>
2. <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf>
3. <https://learn.microsoft.com/en-us/visualstudio/ide/create-csharp-winform-visual-studio?view=vs-2022>

Link Repository : https://github.com/margarethaolivia/Tubes2_karepmu

Link Video : https://youtu.be/ZZ2SqD_rXF4