

LAPORAN TUGAS BESAR II

Pengaplikasian Algoritma BFS dan DFS dalam Menyelesaikan Persoalan Maze Treasure Hunt

**Laporan Ini Dibuat Untuk Memenuhi Tugas Perkuliahan Mata Kuliah
Strategi Algoritma (IF2211)**



**Disusun oleh:
Kelompok 44
“grape shake”**

Anggota:
Muhammad Equilibrie Fajria (13521047)
Muhammad Farrel Danendra Rachim (13521048)
Tobias Natalio Sianipar (13521090)

**SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
SEMESTER II TAHUN 2022/2023**

Daftar Isi

Daftar Isi	2
BAB 1	
Deskripsi Tugas	3
BAB 2	
Landasan Teori	6
2.1. Dasar Teori	6
2.1.1. Traversal Graf	6
2.1.2. DFS	6
2.1.3. BFS	7
BAB 3	
Aplikasi Algoritma BFS/DFS	8
3.1. Langkah-langkah Pemecahan Masalah	8
3.2. Mapping	8
3.3. Ilustrasi Kasus	9
BAB 4	
Analisis Pemecahan Masalah	13
4.1. Implementasi Program	13
4.2. Penjelasan Struktur Data	23
4.3. Tata Cara Penggunaan Program	26
4.4. Hasil Pengujian	27
4.5. Analisis Desain Solusi Algoritma BFS/DFS	35
BAB 5	
Kesimpulan dan Saran	36
5.1. Kesimpulan	36
5.2. Saran	37
5.3. Refleksi	37
5.4. Tanggapan	37
Daftar Pustaka	37
Lampiran	38

BAB 1

Deskripsi Tugas

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 :

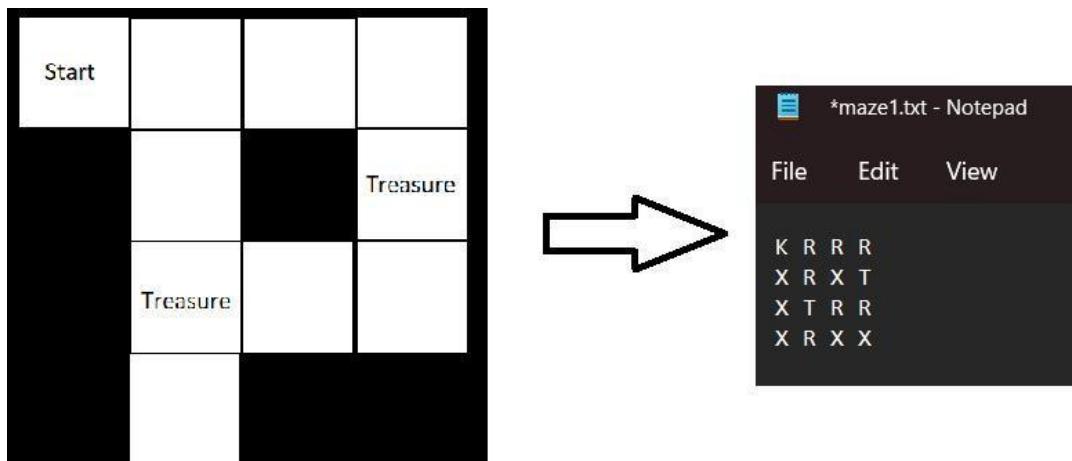
K : Krusty Krab (Titik awal)

T : Treasure

R : Grid yang mungkin diakses / sebuah lintasan

X : Grid halangan yang tidak dapat diakses

Contoh file input :

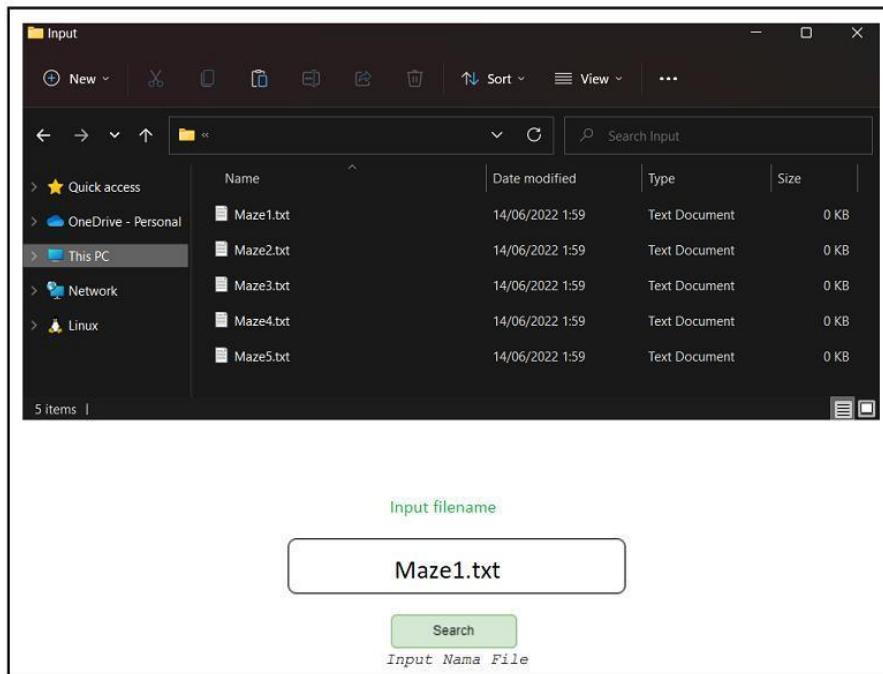


Gambar 1. Ilustrasi input file maze

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.

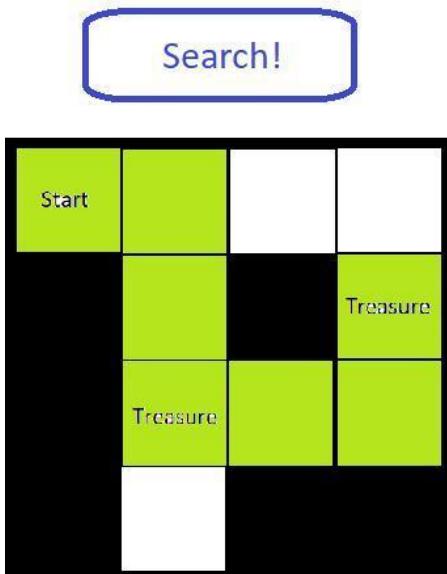
Contoh input aplikasi :



Gambar 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.

BAB 2

Landasan Teori

2.1. Dasar Teori

2.1.1. Traversal Graf

Graph traversal adalah sebuah algoritma yang mengunjungi simpul dalam sebuah graf dengan cara yang sistematik untuk mencari solusi dari persoalan dengan representasi graf. Untuk mencari solusi permasalahan graf, terdapat dua jenis metode yang dapat dilakukan: menelusuri graf tanpa informasi atau menelusuri graf dengan informasi. Jika graf ditelusuri tanpa informasi, biasanya digunakan algoritma BFS, DFS, Depth Limited Search, Iterative Deepening Search, dan Uniform Cost Search. Jika graf ditelusuri dengan informasi, algoritma yang paling efektif digunakan adalah Best First Search dan A*.

Terdapat dua jenis graf permasalahan. Graf pertama adalah graf statis, yaitu graf yang sudah terbentuk sebelum proses pencarian dilakukan dan tidak akan berubah setelah terbentuk, Graf statis direpresentasikan dalam bentuk struktur data dan sangat cocok digunakan jika lingkup masalahnya terbatas, misalnya *citation map* dan *web spider*. Graf yang lain adalah graf dinamis, yaitu graf yang belum terbentuk sebelum proses pencarian, namun terbentuk dan akan terus berubah saat proses pencarian. Graf dinamis sangat cocok digunakan jika lingkup masalah atau pergerakan menyelesaikan permasalahan tersebut mencapai tidak terbatas, misalnya *8-puzzle*, *crossword puzzle*, dan catur.

2.1.2. DFS

Depth-First Search (DFS) adalah algoritma traversal graf yang fokus menelusuri graf secara mendalam. Dengan kata lain, DFS akan menelusuri sebuah graf sampai ujung sebuah cabang graf atau seluruh anak node sudah dikunjungi, lalu melakukan *backtrack* ke node sebelumnya. DFS dapat dilakukan secara rekursif atau menggunakan *stack* untuk mencatat lokasi berikutnya yang akan dikunjungi. Dengan algoritma DFS, simpul tujuan dapat diraih lebih cepat, khususnya jika simpul tersebut memiliki jarak yang jauh dari titik mulai. Namun, algoritma ini tidak menjamin jalur solusi optimal.

Secara umum, pencarian DFS dilakukan sebagai berikut:

1. Buat simpul a sebagai simpul yang mengawali traversal.
2. Kunjungi simpul a.
3. Kunjungi simpul b yang bertetangga dengan simpul a.
4. Ulangi langkah 2 dan 3 dari simpul b.
5. Setelah mencapai suatu simpul sehingga semua simpul yang bertetangga dengannya telah dikunjungi, dilakukan *backtrack* ke simpul terakhir yang

- dikunjungi sebelumnya dan memiliki sebuah simpul tetangga yang belum dikunjungi.
6. Jika tidak ada lagi simpul yang belum dikunjungi yang dapat dicapai dari simpul yang telah dikunjungi, pencarian berakhir.

2.1.3. BFS

Breadth-First Search (BFS) adalah algoritma traversal graf yang fokus menelusuri graf secara melebar. Dengan kata lain, BFS akan menelusuri semua node dalam sebuah level yang belum dikunjungi sebelum pindah ke level graf berikutnya. BFS menggunakan antrian (*queue*) untuk mencatat lokasi berikutnya yang akan dikunjungi. Algoritma BFS menjamin solusi yang optimal, namun juga memakan lebih banyak memori dibandingkan DFS.

Secara umum, pencarian BFS dilakukan sebagai berikut:

1. Buat simpul a sebagai simpul yang mengawali traversal.
2. Buat antrian kosong.
3. Kunjungi simpul a.
4. Masukkan simpul a ke dalam antrian (*enqueue*) dan tandai bahwa simpul a telah dikunjungi.
5. Hapus simpul pertama a pada antrian (*dequeue*) dan cek semua simpul yang bertetangga dengan a, apakah elemen tersebut telah dikunjungi. Jika iya, ulangi langkah lima, sedangkan jika tidak, ulangi langkah 3-4 dengan simpul bertetangga dengan a menggantikan simpul a.
6. Jika antrian kosong, dapat disimpulkan bahwa simpul yang dicari tidak dapat ditemukan dalam graf tersebut.

2.2. C# Desktop Application Development

C# adalah bahasa pemrograman yang dikembangkan oleh Microsoft dan berjalan di framework .NET yang terdiri dari *common language runtime* (CLR) berupa mesin eksekusi dan .NET Framework Class Library yang menyediakan berbagai macam API. C# digunakan untuk mengembangkan aplikasi untuk *web*, *desktop*, *mobile*, aplikasi *game*, dan lain-lain. Pemrograman C# biasanya menggunakan *integrated development environment* (IDE) Microsoft Visual Studio yang mengandung berbagai macam alat dan fitur untuk menyempurnakan setiap tahap pengembangan aplikasi. IDE Visual Studio telah mencakup *compiler*, alat pelengkap kode, perancang grafis, dan fitur lainnya.

BAB 3

Aplikasi Algoritma BFS/DFS

3.1. Langkah-langkah Pemecahan Masalah

Berikut adalah langkah-langkah pemecahan masalah Treasure Hunt dengan algoritma DFS.

1. Menentukan titik awal maze, yaitu di simbol “K”.
2. Membuat *stack* kosong yang berisi daftar node tetangga yang bukan merupakan “X”.
3. *Push* titik awal ke dalam *stack*.
4. Lakukan *pop* dari *stack* selama tidak kosong. Dari titik yang di-*pop* tersebut, jika titik belum dikunjungi, tandai titik tersebut menjadi dikunjungi dan tambahkan titik tersebut ke larik proses titik yang dikunjungi.
5. Pindah ke titik berikutnya yang bertetangga dengan titik tersebut, dan *push* titik-titik tersebut ke dalam *stack*.

Berikut adalah langkah-langkah pemecahan masalah Treasure Hunt dengan algoritma BFS.

1. Menentukan titik awal maze, yaitu di simbol “K”.
2. Membuat antrian kosong yang berisi daftar node tetangga yang bukan merupakan “X”.
3. *Enqueue* titik awal ke dalam antrian.
4. Lakukan *dequeue* dari antrian selama tidak kosong. Dari titik yang di-*dequeue* tersebut, jika titik belum dikunjungi, tandai titik tersebut menjadi dikunjungi dan tambahkan titik tersebut ke larik proses titik yang dikunjungi.
5. Pindah ke titik berikutnya yang bertetangga dengan titik tersebut, dan *enqueue* titik-titik tersebut ke dalam antrian.

3.2. Mapping

Berikut adalah pemetaan masalah dalam algoritma DFS:

1. *Array boolean* “dikunjungi” adalah *array* yang bernilai *true* untuk semua titik “R”, “K”, dan “T” pada mainMatrix yang telah dikunjungi serta *false* untuk

semua titik “R”, “K”, dan “T” pada mainMatrix yang belum dikunjungi, dan semua titik “X” pada mainMatrix.

2. Matriks ketetanggaan A diganti dengan blok percabangan. Sebuah titik (misalkan A) yang bertetangga dengan titik B akan di-*push* dalam *stack* jika titik A bukan merupakan “X”. Titik A akan tetap di-*push* ke dalam *stack* jika telah dikunjungi, dan tidak akan diproses setelah di-*pop* dari stack.

Berikut adalah pemetaan masalah dalam algoritma DFS:

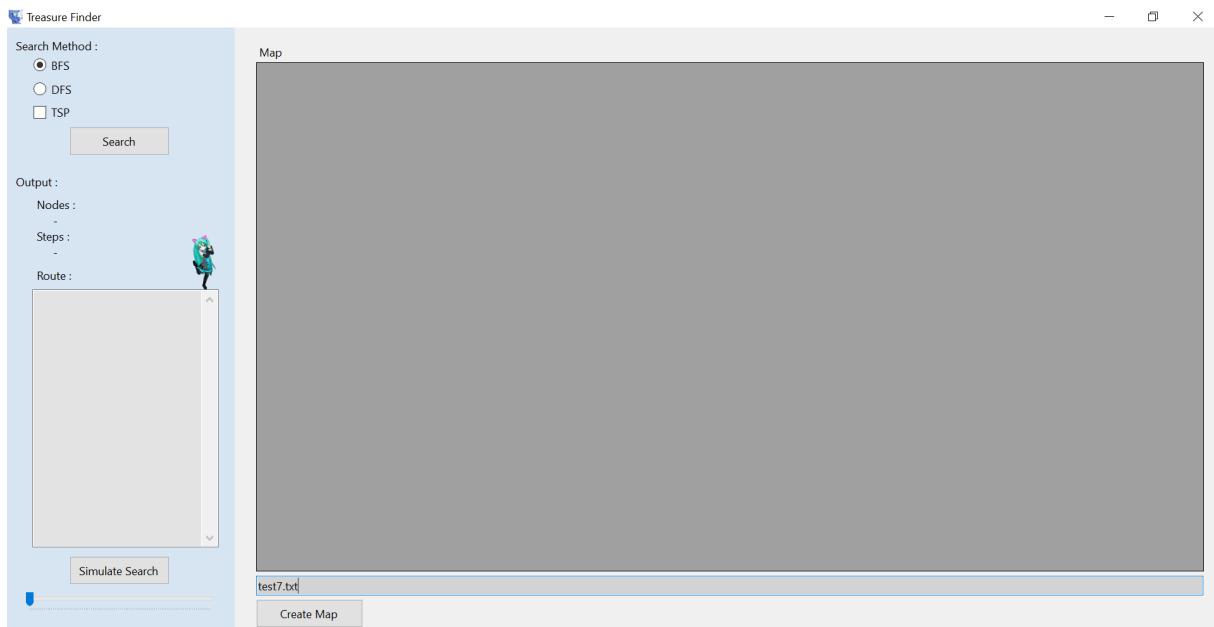
1. *Array boolean* “dikunjungi” adalah *array* yang bernilai *true* untuk semua titik “R”, “K”, dan “T” pada mainMatrix yang telah dikunjungi serta *false* untuk semua titik “R”, “K”, dan “T” pada mainMatrix yang belum dikunjungi, dan semua titik “X” pada mainMatrix.
2. Matriks ketetanggaan A diganti dengan blok percabangan. Sebuah titik (misalkan A) yang bertetangga dengan titik B akan di-*enqueue* dalam antrian jika titik A bukan merupakan “X”. Titik A tidak akan di-*enqueue* ke dalam antrian jika telah dikunjungi.
3. Antrian “q” berguna untuk menyimpan titik-titik valid yang telah dikunjungi dan belum ditelusuri. Jika titik tepat akan ditelusuri, titik akan di-*dequeue* dari antrian.

3.3. Ilustrasi Kasus

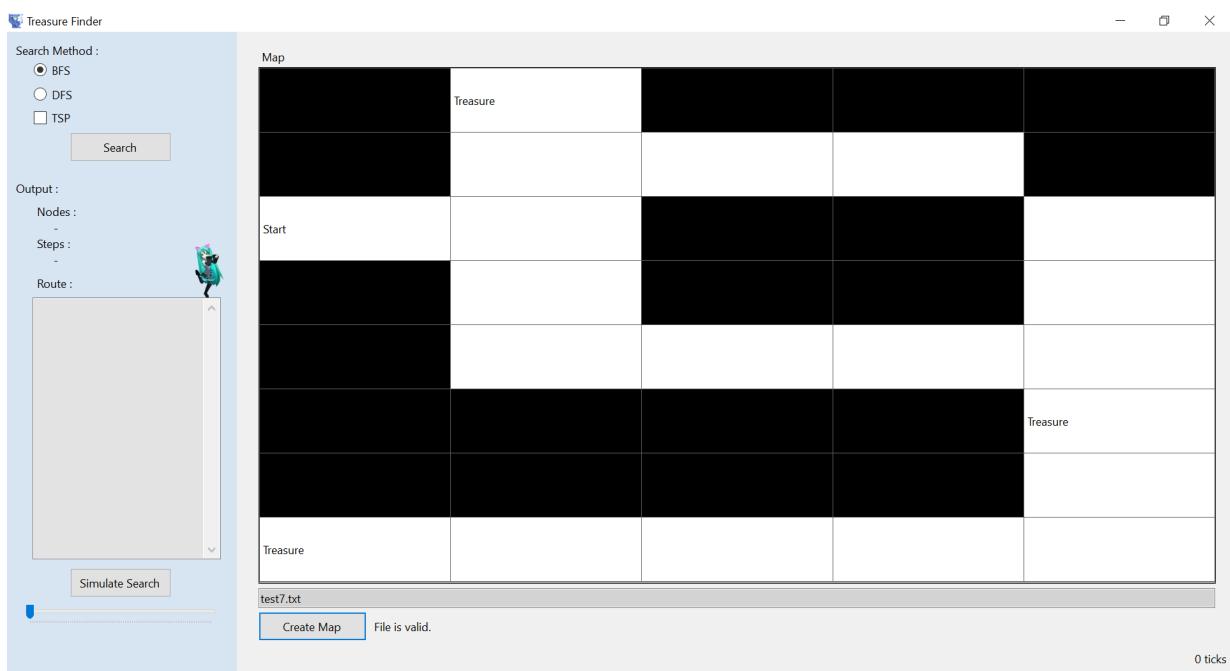
Contoh kasus 1:

- Masukan (test7.txt):

```
test > test7.txt
 1   X T X X X
 2   X R R R X
 3   K R X X R
 4   X R X X R
 5   X R R R R
 6   X X X X T
 7   X X X X R
 8   T R R R R
```



- Keluaran:



Pencarian solusi dengan metode BFS:



Pada kasus ini, pengguna ingin mencari jalur solusi yang tepat untuk memperoleh semua harta karun berdasarkan algoritma BFS. Secara total terdapat 32 node yang dikunjungi oleh program selama penelusuran ini, dan diperoleh 18 langkah yang harus diambil pengguna sesuai algoritma ini. Jalur yang dimaksud adalah R -> U -> U -> D -> D -> D -> D -> R -> R -> R -> D -> D -> D -> D -> L -> L -> L -> L (R: Right, L: Left, U: Up, D: Down) dan ditandai dengan grid berwarna hijau.

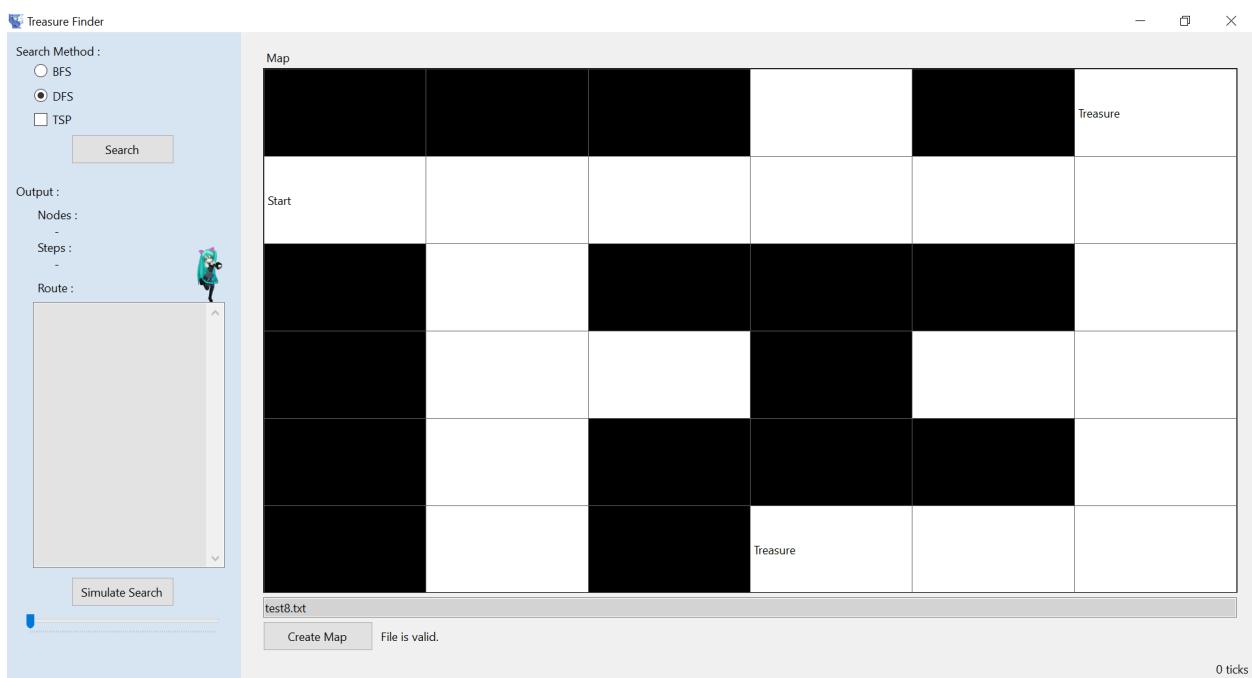
Contoh kasus 2:

- Masukan (test8.txt)

```
test > ≡ test8.txt
      1   X X X R X T
      2   K R R R R R
      3   X R X X X R
      4   X R R X R R
      5   X R X X X R
      6   X R X T R R
```



- Keluaran



Pencarian solusi dengan metode DFS:



Pada kasus ini, pengguna ingin mencari jalur solusi yang tepat untuk memperoleh semua harta karun berdasarkan algoritma DFS. Secara total terdapat 27 node yang dikunjungi oleh program selama penelusuran ini, dan diperoleh 24 langkah yang harus diambil pengguna sesuai algoritma ini. Jalur yang dimaksud adalah R -> R -> R -> R -> R -> U -> D -> D -> D -> D -> D -> L -> L (R: Right, L: Left, U: Up, D: Down) dan ditandai dengan grid berwarna hijau.

BAB 4

Analisis Pemecahan Masalah

4.1. Implementasi Program

Berikut adalah implementasi dari algoritma BFS dan DFS yang digunakan.

BFS:

class BFS:

```
findBFS(MatrixElement[][] treasureMap, string[][] jag, integer x, integer y) ->
Tuple<List<Tuple<integer, integer, integer>>, List<Tuple<integer, integer>>>
```

SPESIFIKASI

{Menghasilkan list process BFS secara keseluruhan dan list path dari titik start sampai ke treasure terakhir}

KAMUS LOKAL

```
bfsProcess : List<Tuple<integer, integer, integer, integer>>
sub_solution : Tuple<integer, integer, List<Tuple<integer, integer, integer, integer>>>
startX : integer
startY : integer
treasureFound : integer
treasureAmount : integer
solutionPath : List<Tuple<integer, integer>>
bfsList : Tuple<List<Tuple<integer, integer, integer, integer>>, List<Tuple<integer, integer>>>
```

ALGORITMA

```
startX <- x
startY <- y
treasureFound <- 0
treasureAmount <- ElementCount(jag, "T")
```

{Menggunakan BFS untuk mencari semua treasure dari start. Pencarian BFS dilakukan dari start ke treasure 1, treasure 1 ke treasure 2, dst sampai ke treasure terakhir}

```
while (treasureFound < treasureAmount) do
```

```

sub_solution <- findSubBFS(treasureMap, jag, x, y)
    bfsProcess.AddRange(sub_solution.Item3)
x <- sub_solution.Item1
y <- sub_solution.Item2
treasureFound++

```

{Mendapatkan list path dari list process BFS}

```

solutionPath <- findPath(bfsProcess, startX, startY, x, y)
bfsList <- bfsProcess, solutionPath
resetMainMatrix(treasureMap)
->bfsList

```

findSubBFS(MatrixElement[][] treasureMap, string[][] jag, integer x, integer y) -> Tuple<integer, integer, List<Tuple<integer, integer, integer>>>

SPESIFIKASI

{Fungsi BFS dari satu treasure (x, y) ke treasure berikutnya. Menghasilkan koordinat treasure berikutnya dan list process untuk koordinat yang diperiksa menggunakan BFS}

KAMUS LOKAL

```

found : boolean
isVisited : boolean[][]
bfsProcess : List <Tuple<integer, integer, integer, integer>>
bfsQueue : Queue<Tuple<integer, integer, integer, integer>>
bfsQueue.Enqueue(new Tuple<integer, integer, integer, integer>(x, y, x, y))
currentX : integer
currentY : integer
prevX : integer
prevY : integer
retVal : Tuple<integer, integer, List<Tuple<integer, integer, integer>>>

```

ALGORITMA

```

found <- false
isVisited <- InitBoolMatrix(jag)

```

{Melakukan dequeue elemen bfsQueue dan visit ke elemen tersebut Untuk setiap jalan yang dapat ditempuh dari elemen tersebut, maka akan dienqueue ke bfsQueue. Prioritas enqueue adalah down > up > right > left}

```

while (not found) do
    currentX <- bfsQueue.Peek().Item1
    currentY <- bfsQueue.Peek().Item2
    prevX <- bfsQueue.Peek().Item3
    prevY <- bfsQueue.Peek().Item4
    bfsQueue.Dequeue()

{Jika elemen yang didequeue belum dikunjungi, maka dimasukkan ke dalam list bfsProcess}
if (not isVisited[currentX, currentY]) then
    isVisited[currentX, currentY] <- true
    bfsProcess.Add(currentX, currentY, prevX, prevY)

{Jika treasure ditemukan, maka proses BFS berhenti}
if      (treasureMap[currentX][currentY].symbol      =      "T"      and
treasureMap[currentX][currentY].numberOfVisits < 1) then
    found <- true

else

{Mencari jalan selanjutnya yang akan dienqueue}
    if  (canMoveDown(treasureMap,  currentX,  currentY)  and  not
isVisited[currentX + 1, currentY]) then
        bfsQueue.Enqueue(currentX + 1, currentY, currentX, currentY)

    if  (canMoveUp(treasureMap,  currentX,  currentY)  and  not
isVisited[currentX - 1, currentY]) then
        bfsQueue.Enqueue(currentX - 1, currentY, currentX, currentY)

    if  (canMoveRight(treasureMap,  currentX,  currentY)  and  not
isVisited[currentX, currentY + 1]) then
        bfsQueue.Enqueue(currentX, currentY + 1, currentX, currentY)

    if  (canMoveLeft(treasureMap,  currentX,  currentY)  and  not
isVisited[currentX, currentY - 1]) then
        bfsQueue.Enqueue(currentX, currentY - 1, currentX, currentY)

    treasureMap[currentX][currentY].numberOfVisits++

retVal <- (currentX, currentY, bfsProcess)
-> retVal

```

findBFSTSP(MatrixElement[][] treasureMap, string[][] jag, integer x, integer y) ->
 Tuple<List<Tuple<integer, integer, integer>>, List<Tuple<integer, integer>>>

SPESIFIKASI

{Menghasilkan list process BFS secara keseluruhan dan list path dari titik start sampai ke titik start kembali dengan semua treasure telah didapatkan}

KAMUS LOKAL

```
bfsProcess : List<Tuple<integer, integer, integer, integer>>
sub_solution : Tuple<integer, integer, List<Tuple<integer, integer, integer, integer>>>
startX : integer
startY : integer
treasureFound : integer
treasureAmount : integer
solutionPath : List<Tuple<integer, integer>>
bfsList : Tuple<List<Tuple<integer, integer, integer, integer>>, List<Tuple<integer, integer>>>
integer>>>
tspPart : List<Tuple<int, int, int, int>>
```

ALGORITMA

```
startX <- x
startY <- y
treasureFound <- 0
treasureAmount <- ElementCount(jag, "T")
```

{Menggunakan BFS untuk mencari semua treasure dari start. Pencarian BFS dilakukan dari start ke treasure 1, treasure 1 ke treasure 2, dst sampai ke treasure terakhir}

```
while (treasureFound < treasureAmount) do
    sub_solution <- findSubBFS(treasureMap, jag, x, y)
    bfsProcess.AddRange(sub_solution.Item3)
    x <- sub_solution.Item1
    y <- sub_solution.Item2
    treasureFound++
```

{Menggunakan BFS untuk mencapai titik start dari titik treasure terakhir (x, y)}

```
tspPart <- findTreasureToStartBFS(treasureMap, jag, x, y)
bfsProcess.AddRange(tspPart)
```

{Mendapatkan list path dari list process BFS}

```
solutionPath <- findPath(bfsProcess, startX, startY, x, y)
bfsList <- bfsProcess, solutionPath
```

```
resetMainMatrix(treasureMap)
-> bfsList
```

```
findTreasureToStartBFS(MatrixElement[][] treasureMap, string[][] jag, integer x, integer y) ->
List<Tuple<integer, integer, integer, integer>>
```

SPESIFIKASI

{Fungsi BFS dari treasure terakhir ke titik start. Menghasilkan koordinat dan list process untuk koordinat yang dicek menggunakan BFS}

KAMUS LOKAL

```
found : boolean
isVisited : boolean[][]
bfsProcess : List <Tuple<integer, integer, integer, integer>>
bfsQueue : Queue<Tuple<integer, integer, integer, integer>>
bfsQueue.Enqueue(new Tuple<integer, integer, integer, integer>(x, y, x, y))
currentX : integer
currentY : integer
prevX : integer
prevY : integer
retVal : Tuple<integer, integer, List<Tuple<integer, integer, integer>>>
```

ALGORITMA

```
found <- false
isVisited <- InitBoolMatrix(jag)
```

{Melakukan dequeue elemen bfsQueue dan visit ke elemen tersebut Untuk setiap jalan yang dapat ditempuh dari elemen tersebut, maka akan dienqueue ke bfsQueue. Prioritas enqueue adalah down > up > right > left}

```
while (not found) do
    currentX <- bfsQueue.Peek().Item1
    currentY <- bfsQueue.Peek().Item2
    prevX <- bfsQueue.Peek().Item3
    prevY <- bfsQueue.Peek().Item4
    bfsQueue.Dequeue()
```

```
{Jika elemen yang didequeue belum dikunjungi, maka dimasukkan ke dalam list bfsProcess}
    if (not isVisited[currentX, currentY]) then
        isVisited[currentX, currentY] <- true
```

```

        bfsProcess.Add(currentX, currentY, prevX, prevY)

{Jika titik start ditemukan, maka proses BFS berhenti}
    if (treasureMap[currentX][currentY].symbol = "K) then
        found <- true

    else

{Mencari jalan selanjutnya yang akan dienqueue}
    if (canMoveDown(treasureMap, currentX, currentY) and not
isVisited[currentX + 1, currentY]) then
        bfsQueue.Enqueue(currentX + 1, currentY, currentX, currentY)

    if (canMoveUp(treasureMap, currentX, currentY) and not
isVisited[currentX - 1, currentY]) then
        bfsQueue.Enqueue(currentX - 1, currentY, currentX, currentY)

    if (canMoveRight(treasureMap, currentX, currentY) and not
isVisited[currentX, currentY + 1]) then
        bfsQueue.Enqueue(currentX, currentY + 1, currentX, currentY)

    if (canMoveLeft(treasureMap, currentX, currentY) and not
isVisited[currentX, currentY - 1]) then
        bfsQueue.Enqueue(currentX, currentY - 1, currentX, currentY)

    treasureMap[currentX][currentY].numberOfVisits++

-> bfsProcess

```

DFS:

class DFS:

findDFS(MatrixElement[][] treasureMap, string[][] jag, integer x, integer y) ->
Tuple<List<Tuple<integer, integer, integer>>, List<Tuple<integer, integer>>>

SPESIFIKASI

{Menghasilkan list process DFS secara keseluruhan dan list path dari titik start sampai ke treasure terakhir}

KAMUS LOKAL

```
dfsProcess : List<Tuple<integer, integer, integer, integer>>
sub_solution : Tuple<integer, integer, List<Tuple<integer, integer, integer, integer>>>
startX : integer
startY : integer
treasureFound : integer
treasureAmount : integer
solutionPath : List<Tuple<integer, integer>>
dfsList : Tuple<List<Tuple<integer, integer, integer, integer>>, List<Tuple<integer, integer>>>
```

ALGORITMA

```
startX <- x
startY <- y
treasureFound <- 0
treasureAmount <- ElementCount(jag, "T")
```

{Menggunakan DFS untuk mencari semua treasure dari start. Pencarian DFS dilakukan dari start ke treasure 1, treasure 1 ke treasure 2, dst sampai ke treasure terakhir}

```
while (treasureFound < treasureAmount) do
    sub_solution <- findSubDFS(treasureMap, jag, x, y)
    dfsProcess.AddRange(sub_solution.Item3)
    x <- sub_solution.Item1
    y <- sub_solution.Item2
    treasureFound++
```

{Mendapatkan list path dari list process dfs}

```
solutionPath <- findPath(dfsProcess, startX, startY, x, y)
dfsList <- dfsProcess, solutionPath
```

```
resetMainMatrix(treasureMap)
```

```
-> dfsList
```

```
findSubDFS(MatrixElement[][] treasureMap, string[][] jag, integer x, integer y) ->  
Tuple<integer, integer, List<Tuple<integer, integer, integer>>>
```

SPESIFIKASI

{Fungsi DFS dari satu treasure (x, y) ke treasure berikutnya. Menghasilkan koordinat treasure berikutnya dan list process untuk koordinat yang diperiksa menggunakan DFS }

KAMUS

```
found : boolean
```

```
isVisited : boolean[][]
```

```
dfsProcess : List <Tuple<integer, integer, integer, integer>>
```

```
dfsStack : Stack<Tuple<integer, integer, integer, integer>>
```

```
dfsStack.push(new Tuple<integer, integer, integer, integer>(x, y, x, y))
```

```
currentX : integer
```

```
currentY : integer
```

```
prevX : integer
```

```
prevY : integer
```

```
retVal : Tuple<integer, integer, List<Tuple<integer, integer, integer>>>
```

ALGORITMA

```
found <- false
```

```
isVisited <- InitBoolMatrix(jag)
```

{Melakukan pop elemen dfsStack dan visit ke elemen tersebut Untuk setiap jalan yang dapat ditempuh dari elemen tersebut, maka akan dipush ke dfsStack. Prioritas push adalah down > up > right > left}

```
while (not found) do
    currentX <- dfsStack.Peek().Item1
    currentY <- dfsStack.Peek().Item2

    prevX <- dfsStack.Peek().Item3

    prevY <- dfsStack.Peek().Item4

    dfsStack.pop()
```

{Jika elemen yang dipop belum dikunjungi, maka dimasukkan ke dalam list dfsProcess}

```
if (not isVisited[currentX, currentY]) then
    isVisited[currentX, currentY] <- true
    dfsProcess.Add(currentX, currentY, prevX, prevY)
```

{Jika treasure ditemukan, maka proses dfs berhenti}

```
if      (treasureMap[currentX][currentY].symbol      =      "T"      and
treasureMap[currentX][currentY].numberOfVisits < 1) then
    found <- true
```

```
else
```

{Mencari jalan selanjutnya yang akan dipush}

```
if      (canMoveDown(treasureMap,      currentX,      currentY)      and      not
isVisited[currentX + 1, currentY]) then
    dfsStack.push(currentX + 1, currentY, currentX, currentY)
```

```

if (canMoveUp(treasureMap, currentX, currentY) and not
isVisited[currentX - 1, currentY]) then

    dfsStack.push(currentX - 1, currentY, currentX, currentY)

if (canMoveRight(treasureMap, currentX, currentY) and not
isVisited[currentX, currentY + 1]) then

    dfsStack.push(currentX, currentY + 1, currentX, currentY)

if (canMoveLeft(treasureMap, currentX, currentY) and not
isVisited[currentX, currentY - 1]) then

    dfsStack.Push(currentX, currentY - 1, currentX, currentY)

treasureMap[currentX][currentY].numberOfVisits++

RetVal <- (currentX, currentY, dfsProcess)

-> RetVal

```

4.2. Penjelasan Struktur Data

a. Partial class Form1

Struktur Data dan Kelas	Penjelasan Singkat
trackBar1_ValueChanged	Untuk mengubah kecepatan simulasi pencarian
button1_Click	Untuk membuat map dari file pada folder test
drawMap	Untuk menggambar map atau mereset map
button2_Click	Untuk mencari solusi jalur BFS/DFS/TSP
button3_Click	Untuk mensimulasi pencarian

Form1_Load	Untuk menginisiasi variabel lokal
Form1_Resize	Untuk melakukan perubahan bentuk kontroler pada form saat ukuran window diubah
moveOnResize	Untuk merubah posisi kontroler
AdjustRowHeight	Untuk merubah ukuran row pada dataGridView sesuai ukuran window
searchDFS	Aplikasi penggunaan class DFS untuk mencari solusi jalur
searchBFS	Aplikasi penggunaan class BFS untuk mencari solusi jalur
sSolution	Untuk menampilkan solusi jalur
simulate	Untuk mensimulasi pencarian solusi jalur
wait	Untuk mengatur kecepatan simulasi
translateSteps	Untuk mendapatkan Step dari solusi jalur (U,D,R,L)
backgroundWorker1_DoWork	Untuk memungkinkan program menerima input saat simulasi/pencarian solusi berlangsung

b. Kelas BFS

Struktur Data dan Kelas	Penjelasan Singkat
findBFS	Fungsi untuk mencari semua treasure dengan BFS. Menghasilkan list process BFS secara keseluruhan dan list path dari titik start sampai ke treasure terakhir
findSubBFS	Merupakan sub solusi dari findBFS. Menghasilkan titik treasure berikutnya dan list proses dari titik (x,y) ke titik treasure berikutnya
findBFSTSP	Fungsi untuk mencari semua treasure dan kembali ke titik start. Bedanya dengan findBFS yaitu setelah treasure terakhir didapatkan, ada pencarian path ke titik start
findTreasureToStartBFS	Fungsi untuk mencari path dari titik treasure terakhir ke titik start dengan proses BFS

c. Kelas DFS

Struktur Data dan Kelas	Penjelasan Singkat
findDFS	Fungsi untuk mencari semua treasure dengan DFS. Menghasilkan list process DFS secara keseluruhan dan list path dari titik start sampai ke treasure terakhir
findSubDFS	Merupakan sub solusi dari findDFS. Menghasilkan titik treasure berikutnya dan list proses dari titik (x,y) ke titik treasure berikutnya
findDFSTSP	Fungsi untuk mencari semua treasure dan kembali ke titik start. Bedanya dengan findDFS yaitu setelah treasure terakhir didapatkan, ada pencarian path ke titik start
findTreasureToStartDFS	Fungsi untuk mencari path dari titik treasure terakhir ke titik start dengan proses DFS

d. Kelas Utils

Struktur Data dan Kelas	Penjelasan Singkat
ReadFile	Menerima dan membaca file .txt yang akan menjadi input peta <i>maze treasure hunt</i> .
isSymbolValid	Mengecek apakah sebuah string berupa “K”, “R”, “T”, atau “X”.
isLineHaveEqualElement	Mengecek apakah tiap baris dari sebuah <i>array</i> 2D memiliki jumlah elemen yang sama
InitMatrix	Menginisialisasi matriks peta dengan simbol sesuai input file dan <i>numberOfVisits</i> = 0
InitBoolMatrix	Menginisialisasi <i>array</i> 2D boolean sesuai dimensi pada matriks peta dengan nilai semua elemen = <i>false</i>
ElementCount	Menghitung jumlah sebuah elemen tertentu dalam sebuah matriks
canMoveRight	Mengecek apakah program bisa berpindah ke kanan (syarat: indeks ke kanan tidak <i>out of bounds</i> dan titik

	sebelah kanan bukan “X”)
canMoveLeft	Mengecek apakah program bisa berpindah ke kiri (syarat: indeks ke kiri tidak <i>out of bounds</i> dan titik sebelah kiri bukan “X”)
canMoveUp	Mengecek apakah program bisa berpindah ke atas (syarat: indeks ke atas tidak <i>out of bounds</i> dan titik di atas bukan “X”)
canMoveDown	Mengecek apakah program bisa berpindah ke bawah (syarat: indeks ke bawah tidak <i>out of bounds</i> dan titik di bawah bukan “X”)
ResetMatrix	Membuat nilai <code>numberOfVisits</code> pada seluruh titik di peta menjadi nol dan membuat nilai elemen pada matriks boolean menjadi <code>false</code>
resetMainMatrix	Membuat nilai <code>numberOfVisits</code> pada seluruh titik di peta menjadi nol
ResetBoolMatrix	Membuat nilai elemen pada matriks boolean menjadi <code>false</code>
findPath	Fungsi untuk mencari path dari list proses BFS maupun DFS

e. *Struct MatrixElement*

Atribut	Penjelasan Singkat
<code>symbol</code>	Simbol berupa string yang direpresentasikan pada peta
<code>numberOfVisits</code>	Jumlah kunjungan pada titik tersebut

f. *Program.cs* dan *ProgramTest.cs*

GUI utama dijalankan di *Program.cs*. Sebagai alternatif, dapat dijalankan *ProgramTest.cs* yang menjalankan fungsionalitas yang sama namun menggunakan *Command Line Interface* (CLI).

Berikut adalah spesifikasi umum dari program ini:

1. Program dapat menerima input file .txt dari folder test yang telah divalidasi
2. Program dapat menerima pilihan algoritma apa yang ingin digunakan: BFS atau DFS

3. Program dapat menerima pilihan solusi menggunakan *traveling salesperson problem* (TSP) agar rute solusi yang diperoleh juga harus kembali ke titik awal setelah menemukan segala harta karunnya
4. Program dapat menampilkan sebuah grid *maze* peta sesuai dengan simbol-simbol pada file input file .txt
5. Program dapat menampilkan rute solusi pada grid dengan kota yang diberi warna hijau dan memberikan keterangan berupa waktu eksekusi algoritma (dalam satuan *tick*), banyak node yang ditelusuri, banyak langkah pada rute solusi, serta keterangan arah untuk rute solusinya.
6. Program dapat menampilkan progress pencarian grid dengan algoritma yang bersesuaian dengan memberikan slider untuk menerima durasi jeda tiap langkah, kemudian memberikan warna kuning untuk tiap grid yang sudah diperiksa dan biru untuk grid yang sedang diperiksa.

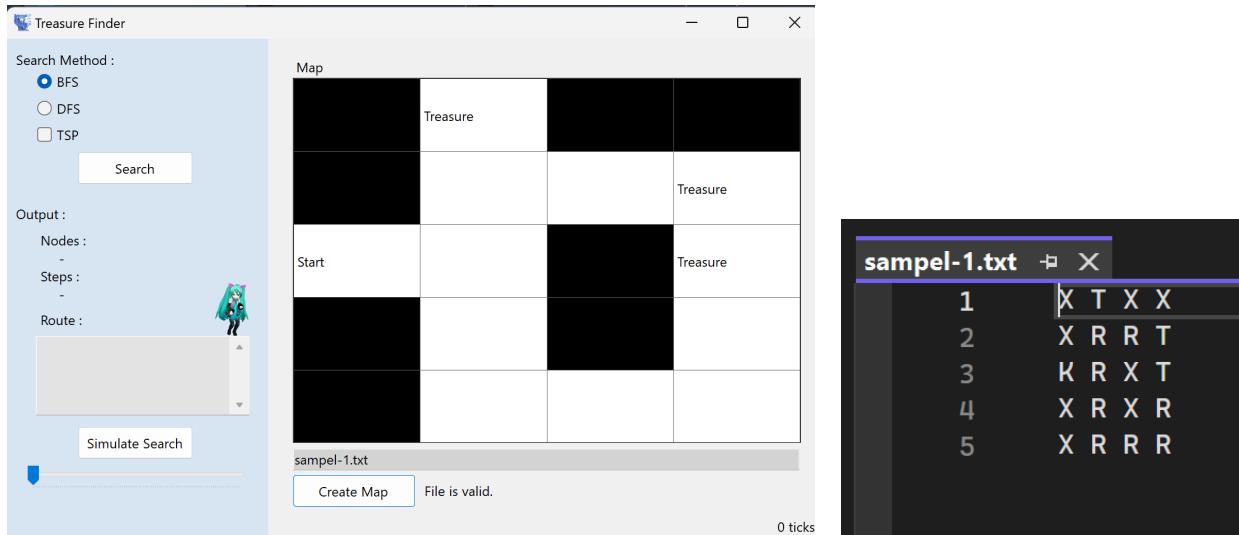
4.3. Tata Cara Penggunaan Program

Program dibangun dengan bahasa C# dan menggunakan kakas Visual Studio .NET untuk mempermudah pembuatan GUI. Berikut langkah-langkah yang harus dilakukan untuk menggunakan program ini:

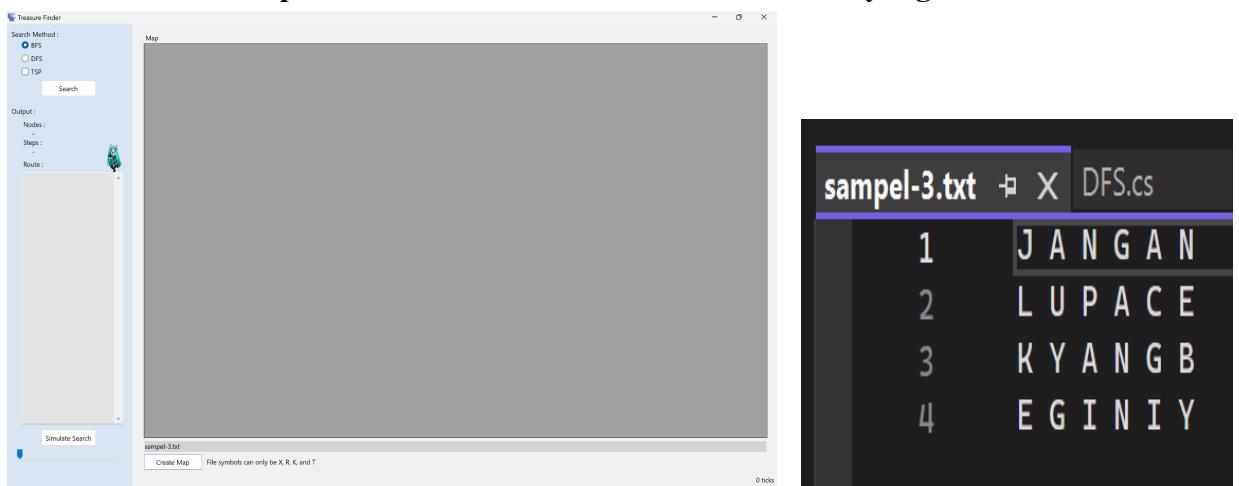
1. Run program dengan mengubah direktori ke src pada terminal, kemudian mengetik ‘dotnet run’, pastikan dotnet versi 7.-- telah terinstall pada komputer.
2. Ketiklah nama file pada folder test yang akan dianalisa lalu klik ‘Create Map’.
3. Pilihlah cara pencarian dan opsi penambahan TSP, kemudian cari solusi jalur dengan menekan ‘Search’, kemudian program akan menampilkan Nodes, Steps, serta Route, dan lama eksekusi algoritma pada bagian kanan bawah window (1 tick setara 1 nano second).
4. Jika ingin melihat simulasi pencarian, tekanlah ‘Simulate Search’ setelah melakukan proses pada tahap 3.
5. Aturlah kecepatan simulasi menggunakan track bar di bawah tombol ‘Simulate Search’.

4.4. Hasil Pengujian

Test Case 1: Menampilkan visualisasi awal maze treasure hunt yang valid



Test Case 2: Menampilkan visualisasi awal maze treasure hunt yang tidak valid



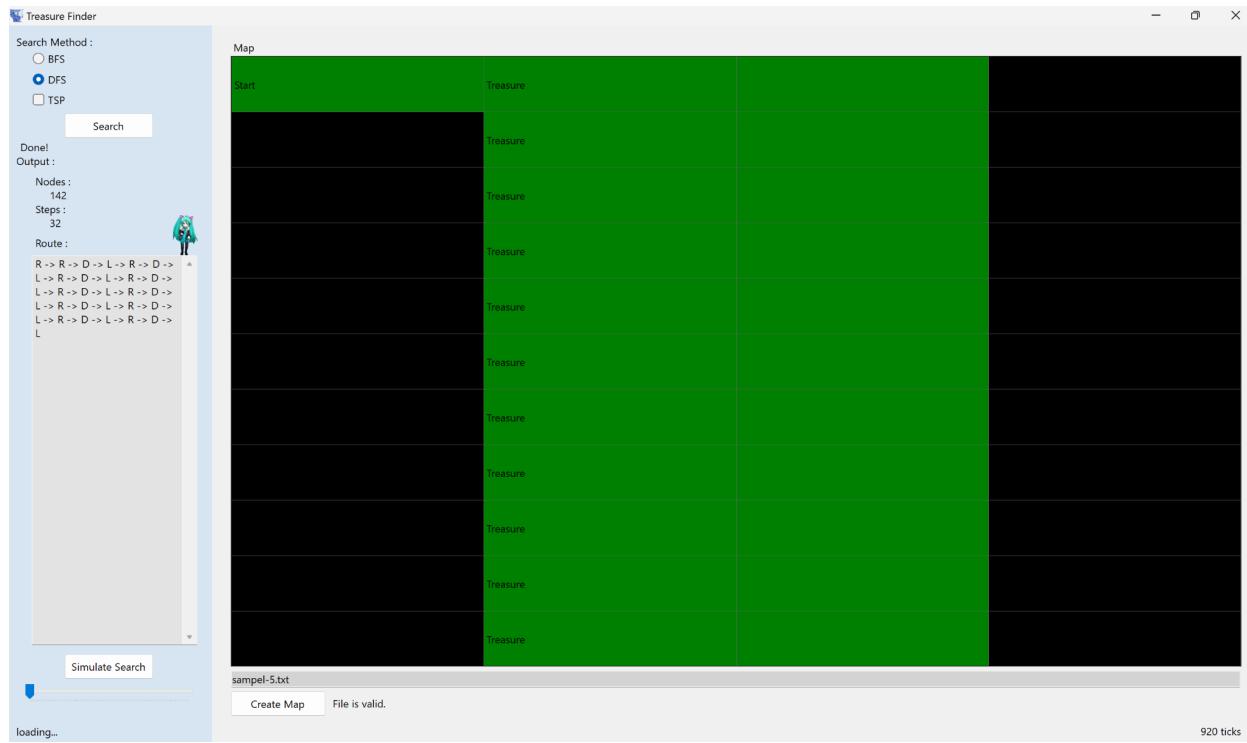
Test Case 3: Menampilkan visualisasi awal maze treasure hunt dengan nama file salah



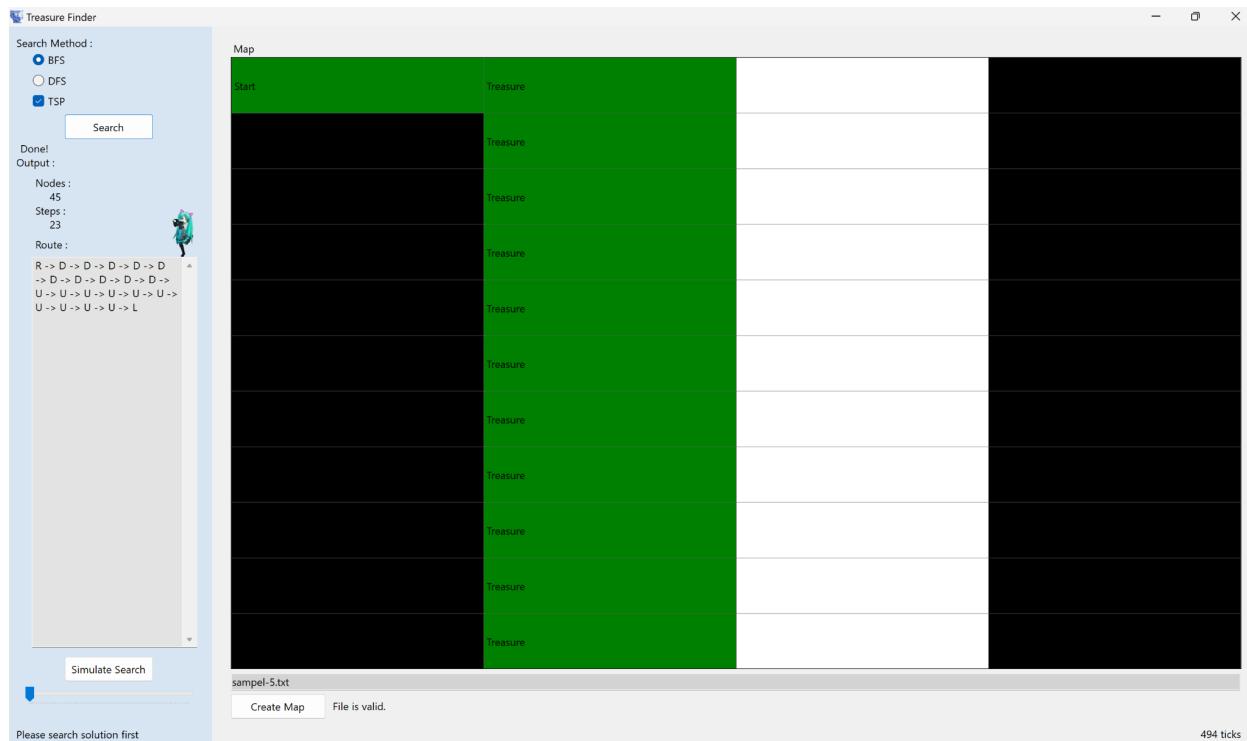
Test Case 4: Menampilkan visualisasi BFS dengan treasure bersebelahan berturut-turut



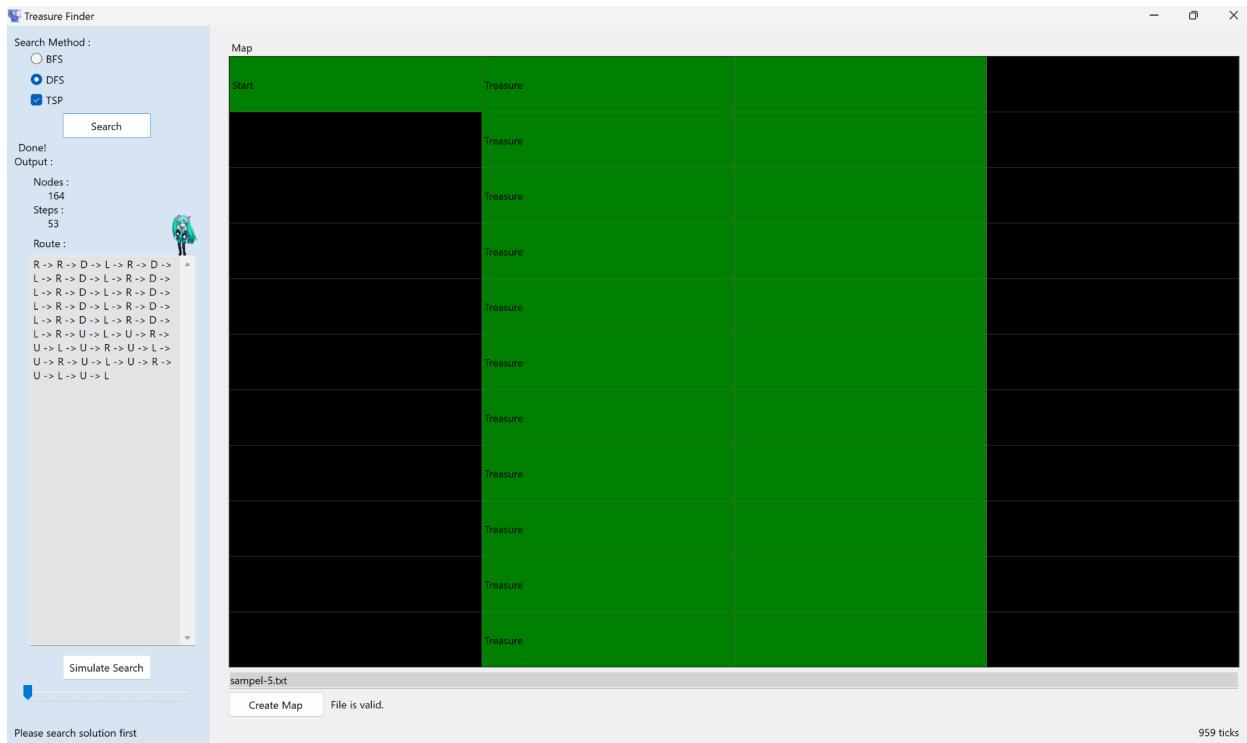
Test Case 5: Menampilkan visualisasi DFS dengan treasure bersebelahan berturut-turut



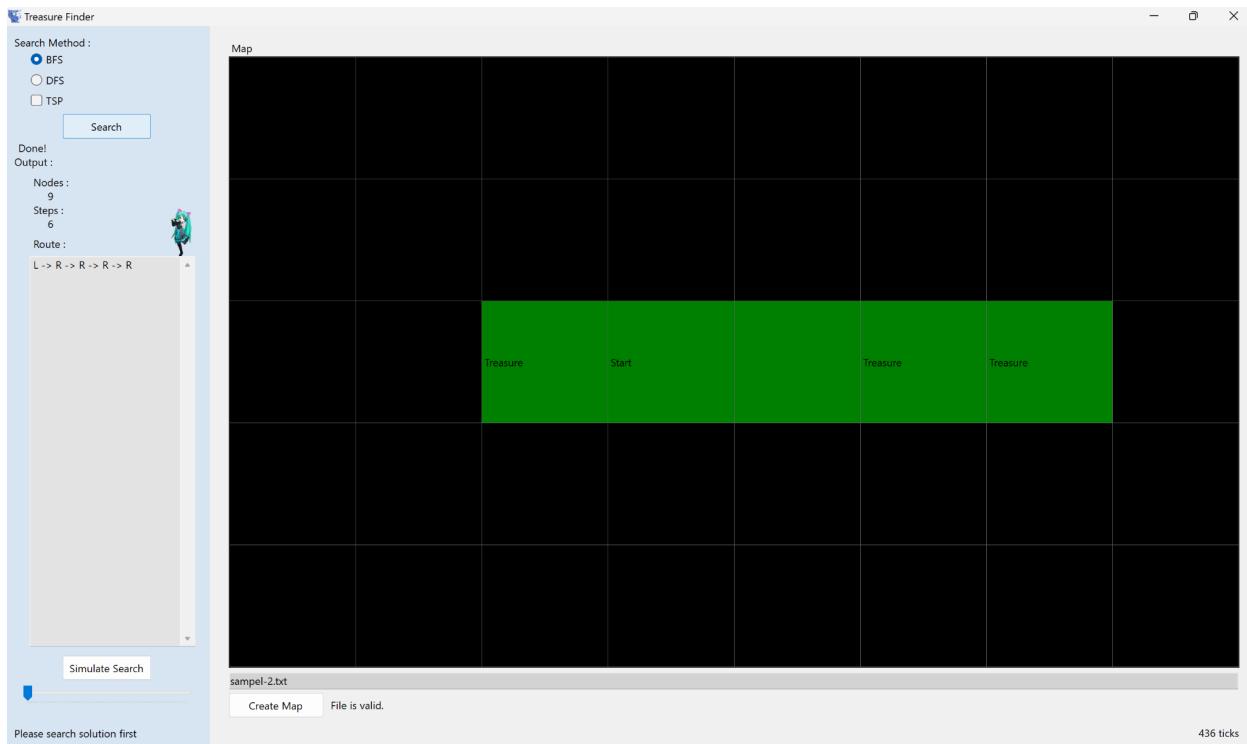
Test Case 6: Menampilkan visualisasi TSP BFS dengan treasure bersebelahan berturut-turut



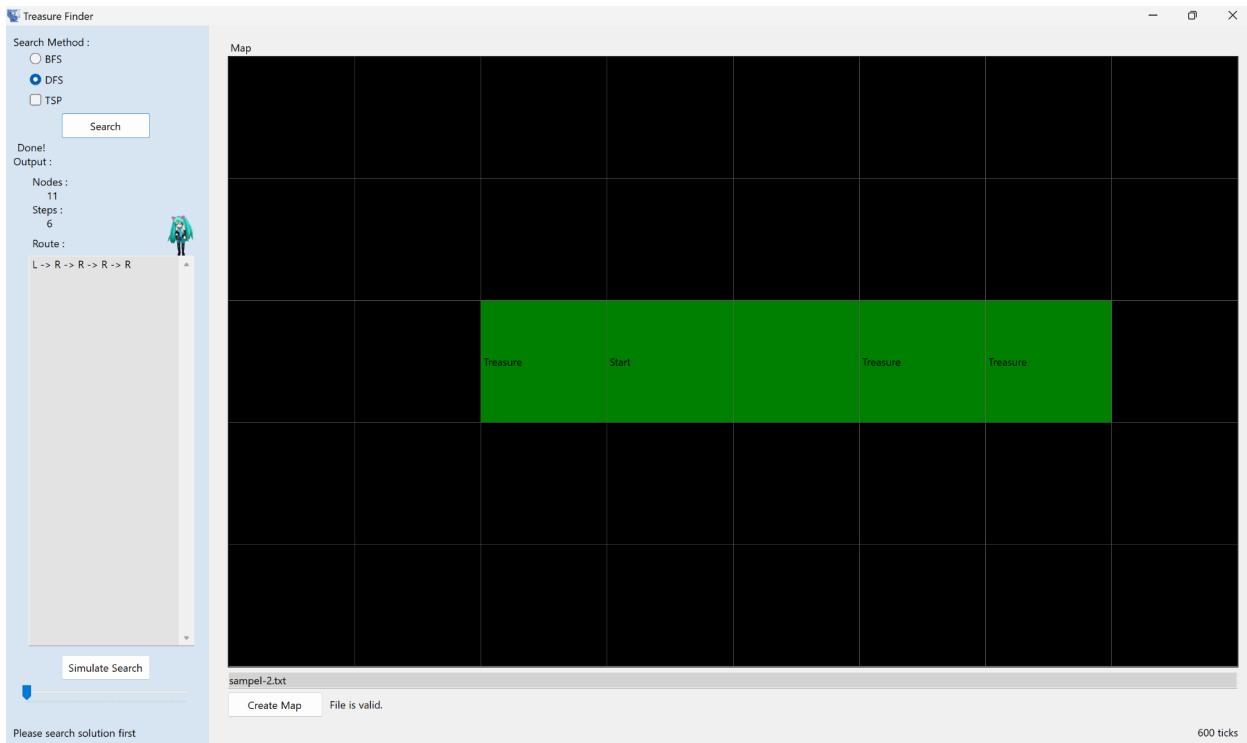
Test Case 7: Menampilkan visualisasi TSP DFS dengan treasure bersebelahan berturut-turut



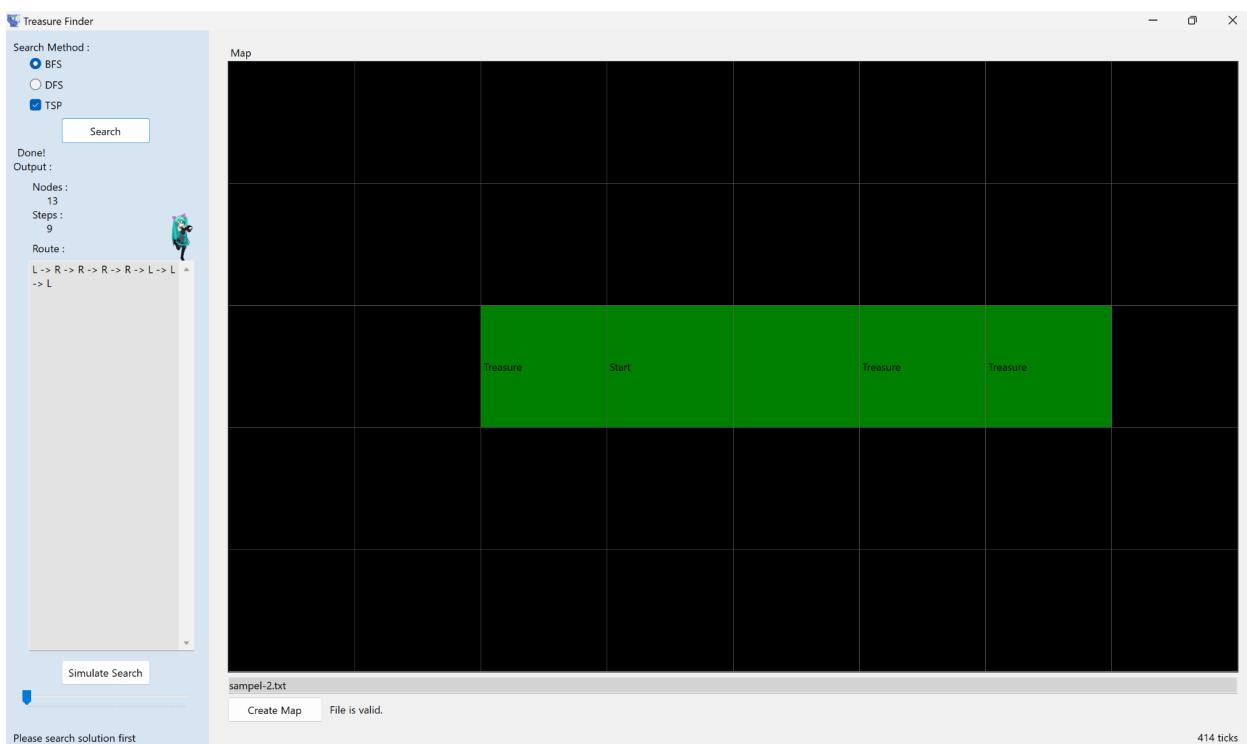
Test Case 8: Menampilkan visualisasi BFS dengan maze treasure hunt satu garis lurus



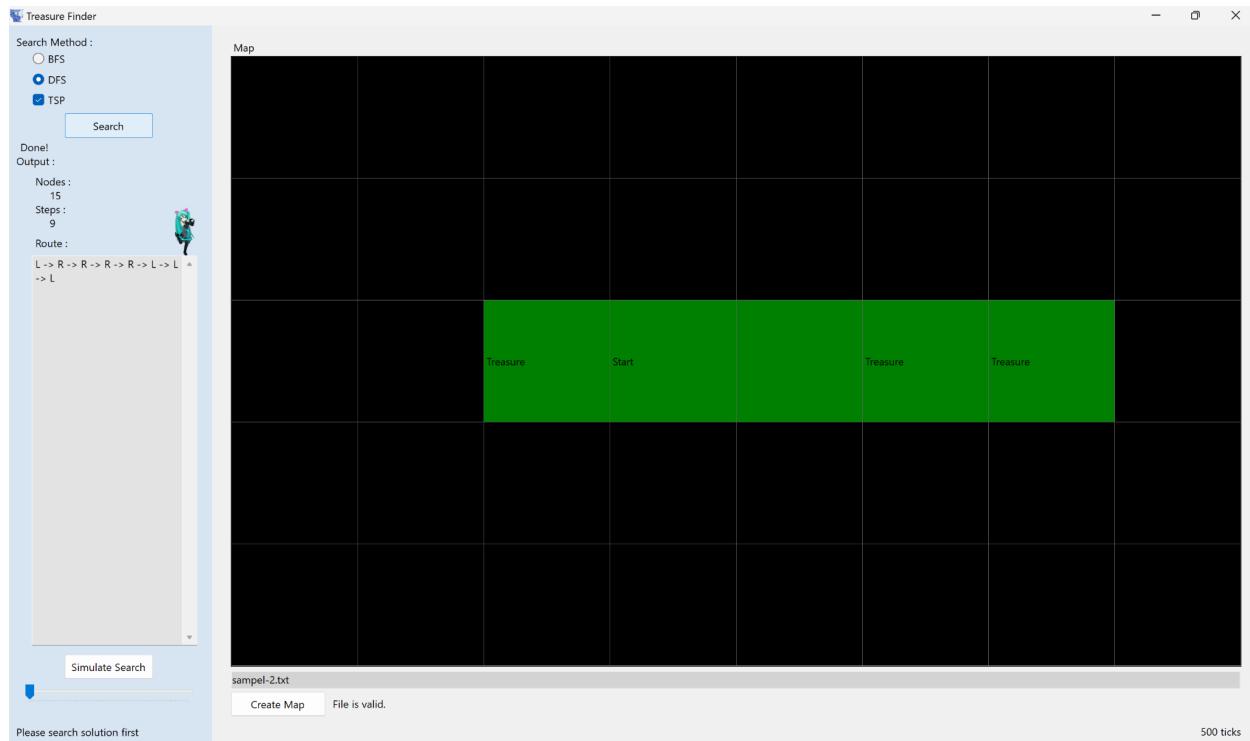
Test Case 9: Menampilkan visualisasi DFS dengan maze treasure hunt satu garis lurus



Test Case 10: Menampilkan visualisasi TSP BFS dengan maze treasure hunt satu garis lurus



Test Case 11: Menampilkan visualisasi TSP DFS dengan maze treasure hunt satu garis lurus



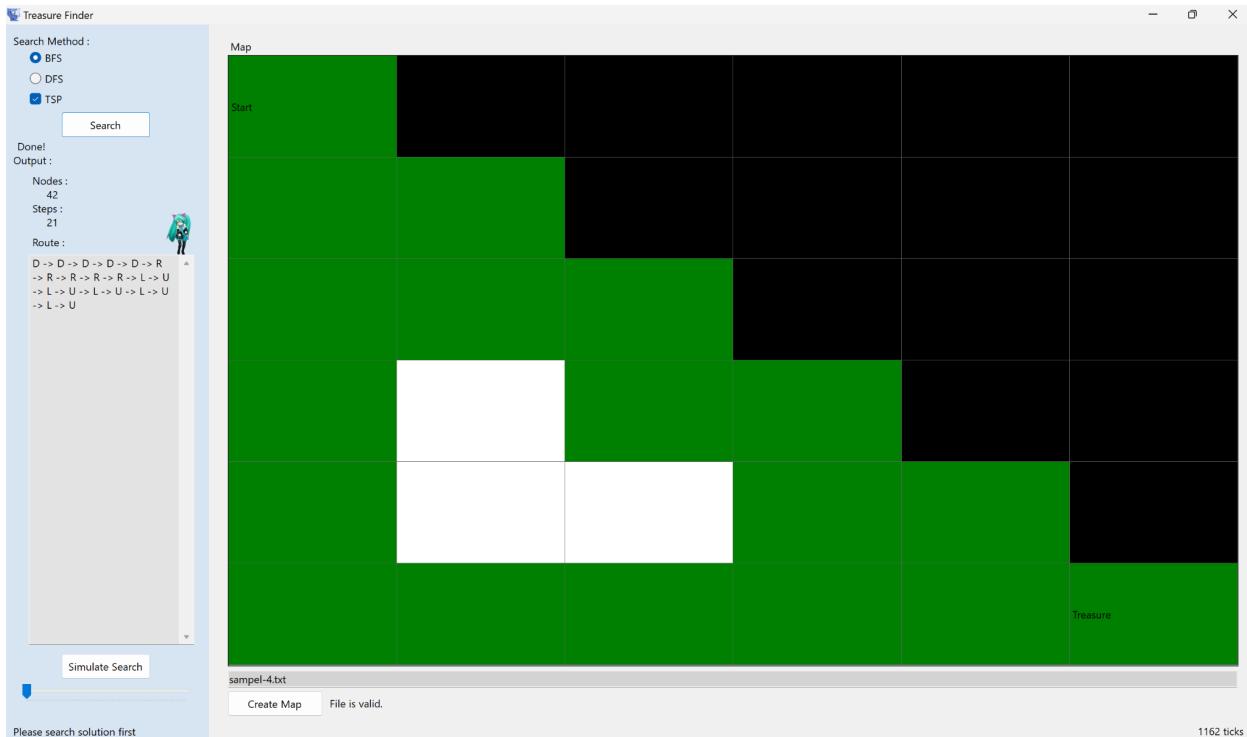
Test Case 12: Menampilkan visualisasi BFS dengan satu treasure



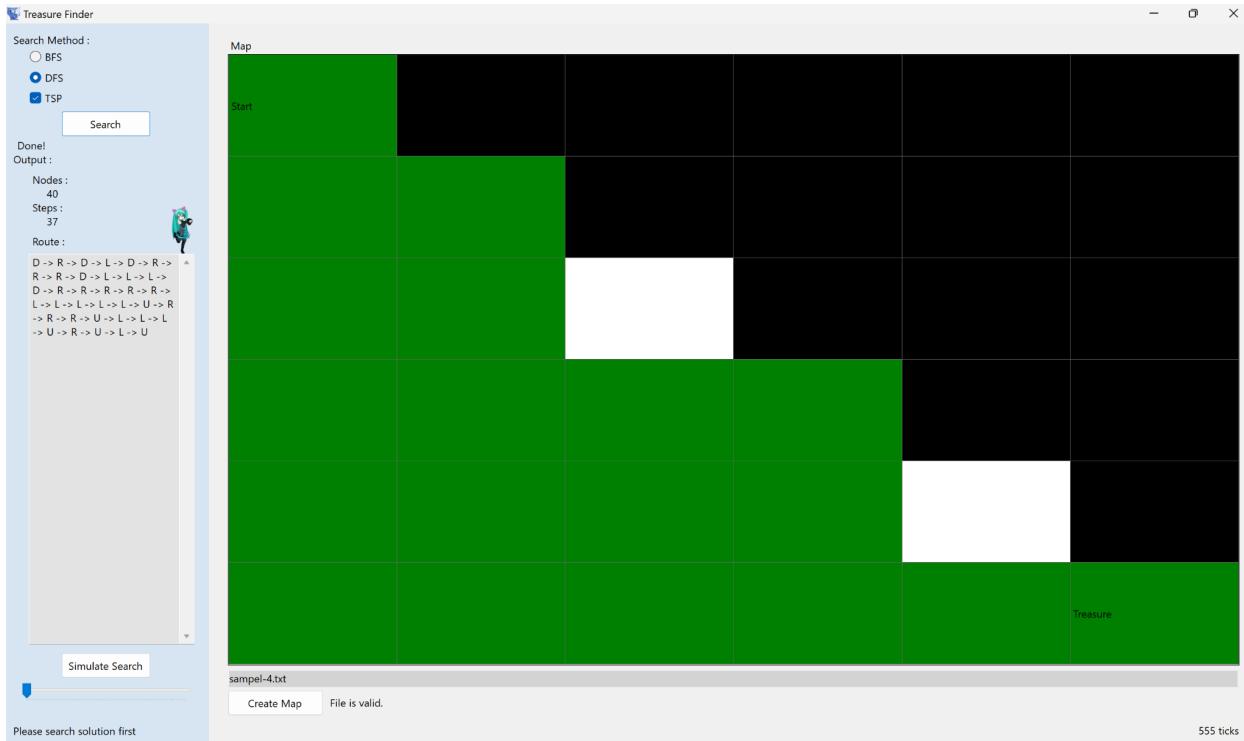
Test Case 13: Menampilkan visualisasi DFS dengan satu treasure



Test Case 14: Menampilkan visualisasi TSP BFS dengan satu treasure



Test Case 15: Menampilkan visualisasi TSP DFS dengan satu treasure



4.5. Analisis Desain Solusi Algoritma BFS/DFS

Secara umum, pencarian menggunakan BFS akan menghasilkan rute terpendek. Namun, node yang dikunjungi biasanya akan lebih banyak daripada pencarian menggunakan DFS. Hal ini dikarenakan pencarian BFS yang bersifat melebar, sehingga cenderung memeriksa seluruh lintasan sampai tujuan ditemukan. Sedangkan untuk pencarian DFS bersifat mendalam hanya memeriksa satu arah lintasan sampai jalan buntu atau tujuan ditemukan. Oleh karena itu, lintasan yang dihasilkan oleh pencarian DFS secara umum lebih panjang, namun kecepatan pencarinya lebih cepat daripada pencarian menggunakan BFS, karena jumlah node yang dikunjungi dengan menggunakan pencarian DFS lebih sedikit.

Skenario 1: Treasure saling bersebelahan

Kasus	Hasil	
	BFS	DFS
Terbaik: Susunan semua treasure merupakan prioritas arah simpul tertinggi/pertama	Treasure langsung ditemukan pada <i>visit</i> node pertama	Treasure langsung ditemukan pada <i>visit</i> node pertama
Rata-rata: Susunan treasure ada yang prioritas arah simpulnya pertama, kedua,	Treasure berikutnya pasti didapatkan pada <i>visit</i> pertama sampai keempat	Treasure berikutnya didapatkan pada <i>visit</i> pertama sampai jumlah lintasan-1

ketiga, atau keempat		
Terburuk: Semua treasure susunannya merupakan prioritas arah simpul terendah/keempat	Treasure berikutnya pasti didapatkan pada <i>visit</i> keempat	Treasure berikutnya didapatkan pada <i>visit</i> kelima sampai lintasan-1

Untuk skenario treasure yang bersebelahan, secara keseluruhan BFS lebih baik karena BFS memeriksa kotak dalam satu area, sehingga bila treasure bersebelahan, maka pasti akan didapatkan dalam 4 visit. Sedangkan DFS, bila posisi treasure merupakan prioritas arah simpul terendah dan lintasan banyak maka DFS akan mengunjungi lintasan lain terlebih dahulu sehingga meenghasilkan rute yang jauh dan node yang dikunjungi lebih banyak.

Skenario 2: Treasure tersebar dalam satu garis lurus

Kasus	Hasil	
	BFS	DFS
Terbaik: Susunan semua treasure merupakan prioritas arah simpul tertinggi/pertama	Treasure akan ditemukan pada <i>visit</i> node sesuai jarak titik awal dengan treasure + jumlah node yang ada di sekitar lintasan lurus tersebut	Treasure akan ditemukan pada <i>visit</i> node sesuai jarak titik awal dengan treasure
Rata-rata: Susunan treasure ada yang prioritas arah simpulnya pertama, kedua, ketiga, atau keempat	Treasure akan ditemukan pada <i>visit</i> node sesuai jarak titik awal dengan treasure + jumlah node yang ada di sekitar lintasan lurus tersebut	Sangat bergantung pada jumlah lintasan. Jika bukan prioritas pertama, maka treasure akan ditemukan pada <i>visit</i> node sesuai panjang lintasan sampai ujung.
Terburuk: Semua treasure susunannya merupakan prioritas arah simpul terendah/keempat	Treasure akan ditemukan pada <i>visit</i> node sesuai jarak titik awal dengan treasure + jumlah node yang ada di sekitar lintasan lurus tersebut	Treasure ditemukan pada iterasi <i>visit</i> terakhir, karena prioritas terendah berarti lintasan akan dikunjungi paling terakhir

Pada skenario ini, pada kasus terbaik DFS akan selalu lebih unggul daripada BFS. Namun untuk skenario rata-rata dan skenario terburuk bergantung pada ukuran lintasan pada maze treasure hunt. Semakin besar ukurannya maka akan semakin unggul DFS, begitu juga sebaliknya semakin kecil lintasan maka BFS yang akan semakin unggul.

BAB 5

Kesimpulan dan Saran

5.1. Kesimpulan

Dari Tugas Besar 2 IF2211 Strategi Algoritma Semester II 2022/2023 ini kami menyimpulkan bahwa kami dapat memanfaatkan algoritma traversal graf seperti DFS dan BFS untuk menyelesaikan permasalahan *maze treasure hunt*. Masing-masing algoritma DFS dan BFS memiliki kelebihan dan kelemahan tersendiri, sehingga untuk mencapai solusi yang paling optimal dan cepat, algoritma yang sesuai bergantung kepada jenis peta *maze*-nya. Algoritma BFS lebih cocok digunakan jika *treasure* terletak relatif dekat dari titik awal karena pencarinya yang melebar, sedangkan algoritma DFS digunakan jika *treasure* terletak relatif jauh dari titik awal karena pencarinya yang mendalam.

5.2. Saran

Kami memiliki beberapa saran untuk disampaikan berkaitan dengan Tugas Besar 2 IF2211 Strategi Algoritma Semester II 2022/2023, yaitu:

- Dapat ditambahkan fitur di mana input file dapat berupa file Excel sehingga pembuatan peta *maze* dapat dilakukan dengan mudah.
- Program dapat diimplementasikan di GUI lain selain di kakas .NET.

5.3. Refleksi

Dalam tugas besar ini kami menghadapi berbagai kendala. Kami perlu beradaptasi dalam menggunakan software Visual Studio dan GUI-nya. Selain itu, kami juga mengalami kesulitan dalam mencari rute solusi dengan algoritma DFS. Namun, kami juga mampu memperluas pengetahuan mengenai algoritma traversal graf dan bahasa C# untuk pertama kalinya.

5.4. Tanggapan

Tugas besar ini sangat bermanfaat bagi kami karena kami dapat mengetahui penerapan algoritma traversal graf seperti BFS dan DFS dalam bentuk program, sambil melakukan eksplorasi terhadap bahasa pemrograman C# serta kakas Visual Studio .NET. Kami juga dapat melatih diri dalam melakukan integrasi bahasa pemrograman dengan GUI.

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>
- <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-backtracking-2021-Bagian1.pdf>

Lampiran

- Link repository: https://github.com/Breezy-DR/Tubes2_grape-shake
- Link YouTube: <https://youtu.be/uF6i-YrxIAE>