

LAPORAN
TUGAS BESAR 2 IF2211 STRATEGI ALGORITMA
PENGAPLIKASIAN ALGORITMA BFS DAN DFS
DALAM MENYELESAIKAN PERSOALAN MAZE
TREASURE HUNT



oleh

Fakhri Muhammad Mahendra	13521045
Fatih Nararya Rashadyfa I.	13521060
Michael Jonathan Halim	13521124

Kelompok	SiPalingKeliling
-----------------	-------------------------

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
BANDUNG
2023

DAFTAR ISI

DAFTAR ISI	1
BAB 1	1
DESKRIPSI TUGAS	1
1.1 Spesifikasi Tugas	1
BAB II	7
LANDASAN TEORI	7
2.1 Dasar Teori	7
2.1.1 Graf Traversal	7
2.1.2 Breadth First Search	7
2.1.3 Depth First Search	8
2.2 C# Desktop Application Development	9
BAB III	11
PENJELASAN ALGORITMA	11
3.1 Langkah-Langkah Pemecahan Masalah	11
3.2 Proses Mapping Persoalan untuk BFS dan DFS	13
3.2.1 Breadth First Search	14
3.3 Ilustrasi Kasus	14
3.3.1 Kasus BFS	14
3.3.2 Kasus DFS	18
BAB IV	24
ANALISIS PEMECAHAN MASALAH	24
4.1 Implementasi Program	24
4.1.1 BFS	24
4.1.2 tspBFS	26
4.1.3 DFS	28
4.1.4 tspDFS	30
4.1.5 Helper	32
4.1.6 ReadFile	34
4.2 Penjelasan Struktur Data	35
4.2.1 List	35
4.2.2 Tuple	36
4.2.3 PrioQueue	36
4.2.4 Graf	38
4.2.5 Solution	39
4.3 Penjelasan Tata Cara Penggunaan Program	41
4.4 Hasil Pengujian	42
4.4.1 Pengujian 1	42
4.4.2 Pengujian 2	43
4.4.3 Pengujian 3	44

4.4.4 Pengujian 4	46
4.4.5 Pengujian 5	47
4.4.6 Pengujian 6	49
4.4.7 Pengujian 7	50
4.4.8 Pengujian 8	51
4.4.9 Pengujian 9	52
4.4.10 Pengujian 10	53
4.5 Analisis dari Desain Solusi Algoritma	54
4.5.1 BFS	54
4.5.2 DFS	55
4.6 Bonus TSP	55
BAB V	57
KESIMPULAN, SARAN, DAN REFLEKSI	57
5.1 Kesimpulan	57
5.2 Saran	57
5.3 Refleksi	58
5.4 Tanggapan	58
DAFTAR PUSTAKA	59
LAMPIRAN	60

BAB 1

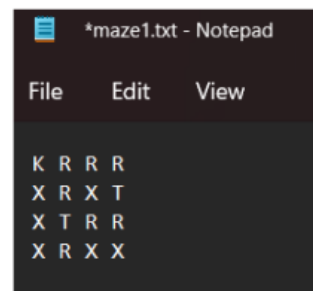
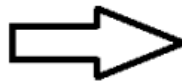
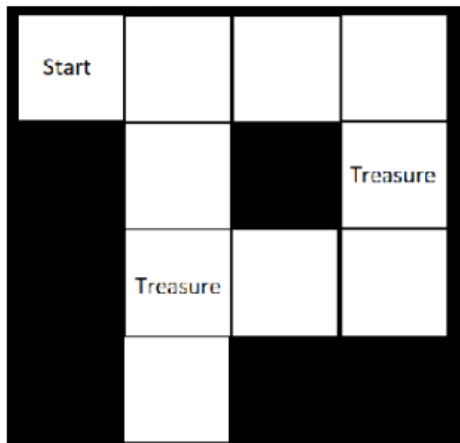
DESKRIPSI TUGAS

1.1 Spesifikasi 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 :

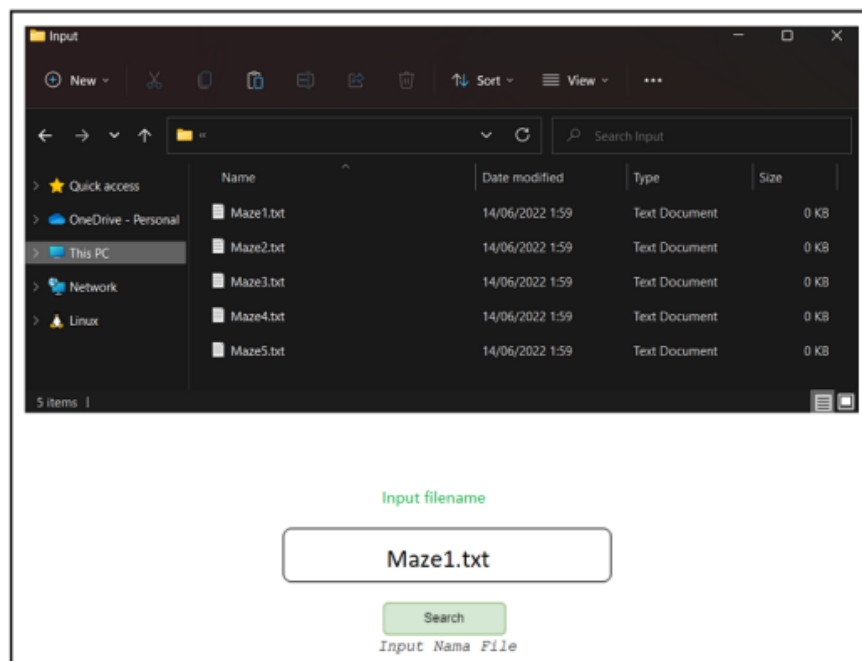


Dengan memanfaatkan algoritma Breadth First Search (BFS) dan Depth First Search (DFS), anda dapat menelusuri grid (simpul) yang mungkin dikunjungi hingga ditemukan rute solusi, baik secara melebar ataupun mendalam bergantung alternatif algoritma yang dipilih. Rute solusi adalah rute yang memperoleh seluruh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

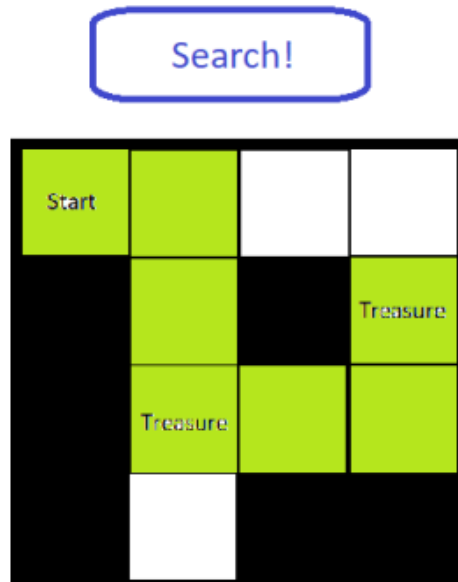
Contoh input aplikasi :



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.

Contoh output Aplikasi :



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

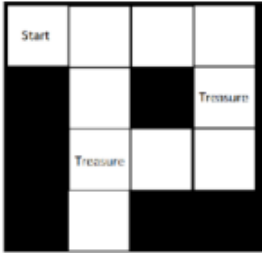
Treasure Hunt Solver

Input

Filename

Algoritma
☒ BFS
☐ DFS

Output



Route:

Steps:

Nodes :

Execution Time :

sebelum dicari rute solusinya dan

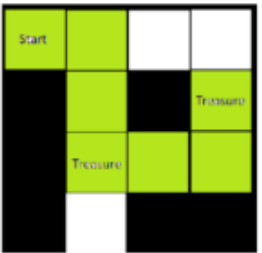
Treasure Hunt Solver

Input

Filename

Algoritma
☒ BFS
☐ DFS

Output



Route: R - D - D - R - R - U

Steps: 6

Nodes : 11

Execution Time : 850 ms

setelah dicari rute 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.

BAB II

LANDASAN TEORI

2.1 Dasar Teori

2.1.1 Graf Traversal

Algoritma traversal dalam graf merupakan suatu algoritma yang mengunjungi simpul dengan cara yang sistematis. Algoritma ini terbagi menjadi dua jenis, yaitu pencarian secara melebar (Breadth First Search/BFS) dan pencarian secara mendalam (Depth First Search/DFS) dengan asumsi bahwa graf merupakan graf terhubung.

Algoritma pencarian solusi ini terbagi menjadi dua persoalan juga, yaitu tanpa informasi (blind search) dan dengan informasi (informed search). Informed search merupakan pencarian berbasis heuristik karena kita sudah mengetahui non-goal state “lebih menjanjikan” daripada yang lain. Namun, untuk pencarian solusi pada permasalahan tugas ini adalah blind search dengan DFS dan BFS.

Representasi graf dalam proses pencarian terdapat dua jenis, yaitu graf statis dan graf dinamis. Graf statis merupakan graf yang sudah terbentuk sebelum proses pencarian dilakukan. Graf dinamis merupakan graf yang terbentuk saat proses pencarian dilakukan sehingga graf dibangun selama pencarian solusi. Untuk kasus persoalan tugas ini, graf yang dipakai adalah graf statis karena peta harta karun sudah diberikan.

2.1.2 Breadth First Search

Breadth First Search atau BFS merupakan salah satu algoritma traversal dalam graf secara melebar. Secara sederhana, dalam suatu pencarian dalam graf tentu harus ditentukan simpul sebagai titik awal pencarian. Misalkan, penelusuran dimulai dari simpul v . Simpul v akan diperiksa apakah simpul tersebut merupakan tujuan akhir atau tidak. Jika tidak, akan dilakukan penelusuran lebih lanjut kepada seluruh simpul yang bertetangga dengan simpul v . Penelusuran dilakukan secara iteratif sehingga seluruh simpul yang bertetangga akan

diperiksa satu per satu. Lalu, akan dilakukan penelusuran lebih lanjut pada simpul-simpul lain yang belum dikunjungi dan bertetangga dengan simpul-simpul yang telah dikunjungi. Semua langkah di atas dilakukan hingga simpul tujuan didapati. Karena penelusuran dilakukan berurutan dan melebar, digunakan struktur data *Queue* untuk menentukan urutan simpul yang diperiksa terlebih dahulu. Setelah itu, agar simpul yang sama tidak diperiksa lagi, digunakan struktur data *array* yang bertipe *boolean* untuk menandakan apakah suatu simpul telah dikunjungi atau tidak.

Berikut adalah *pseudocode* untuk algoritma BFS.

```

procedure BFS(input v:integer)
{ Traversal graf dengan algoritma pencarian BFS.

Masukan: v adalah simpul awal kunjungan
Keluaran: semua simpul yang dikunjungi dicetak ke layar
}
Deklarasi
w : integer
q : antrian;

procedure BuatAntrian(input/output q : antrian)
{ membuat antrian kosong, kepala(q) diisi 0 }

procedure MasukAntrian(input/output q:antrian, input v:integer)
{ memasukkan v ke dalam antrian q pada posisi belakang }

procedure HapusAntrian(input/output q:antrian, output v:integer)
{ menghapus v dari kepala antrian q }

function AntrianKosong(input q:antrian) → boolean
{ true jika antrian q kosong, false jika sebaliknya }

Algoritma:
BuatAntrian(q)          { buat antrian kosong }

write(v)                { cetak simpul awal yang dikunjungi }
dikunjungi[v]←true      { simpul v telah dikunjungi, tandai dengan
                        true}
MasukAntrian(q,v)       { masukkan simpul awal kunjungan ke dalam
                        antrian}

{ kunjungi semua simpul graf selama antrian belum kosong }
while not AntrianKosong(q) do
    HapusAntrian(q,v)    { simpul v telah dikunjungi, hapus dari
                        antrian }
    for tiap simpul w yang bertetangga dengan simpul v do
        if not dikunjungi[w] then
            write(w)      {cetak simpul yang dikunjungi}
            MasukAntrian(q,w)
            dikunjungi[w]←true
        endif
    endfor
endwhile

```

2.1.3 Depth First Search

DFS atau Depth First Search merupakan algoritma traversal dalam graf secara mendalam. Seperti namanya “Depth”, proses penelusuran diutamakan dengan dilakukan secara

mendalam ketika memungkinkan. Sama seperti BFS, perlu ditentukan suatu simpul sebagai titik awal pencarian. Misalkan, penelusuran dimulai dari simpul v . Setelah penelusuran pada simpul v selesai dan belum ditemukan simpul tujuan, akan dilanjutkan penelusuran ke salah satu simpul yang bertetangga dengan simpul v . Lalu, akan dilanjutkan lagi penelusuran pada salah satu simpul yang bertetangga dengan simpul sebelumnya. Langkah ini akan dilakukan terus menerus hingga tidak ada simpul tetangga yang bisa dikunjungi lagi. Maka dari itu, ada proses backtrack untuk kembali ke simpul sebelumnya untuk mencari simpul tetangga lain yang belum dikunjungi. Pencarian akan dilanjutkan terus hingga ditemukan simpul tujuan. Sama seperti BFS, digunakan struktur data *array* yang bertipe *boolean* untuk menandakan apakah suatu simpul telah dikunjungi atau tidak. Namun, struktur data yang berbeda adalah pada DFS digunakan Stack.

Berikut adalah *pseudocode* untuk algoritma DFS.

```

procedure DFS(input v:integer)
  {Mengunjungi seluruh simpul graf dengan algoritma pencarian DFS

  Masukan: v adalah simpul awal kunjungan
  Keluaran: semua simpul yang dikunjungi ditulis ke layar
  }
  Deklarasi
    w : integer

  Algoritma:
    write(v)
    dikunjungi[v] ← true
    for w ← 1 to n do
      if A[v,w]=1 then (simpul v dan simpul w bertetangga )
        if not dikunjungi[w] then
          DFS(w)
        endif
      endif
    endfor

```

2.2 C# Desktop Application Development

C# Desktop Application Development merupakan sebuah tool untuk men-develop desktop application. Pengembangan desktop application ini memerlukan sebuah Graphical User Interface yang dapat menampilkan grafik juga tidak hanya teks saja sehingga digunakan

IF2211

sebuah framework yaitu Window Forms/WPF sebagai pondasi dari aplikasi ini. Untuk mempermudah development, digunakan IDE yaitu Visual Studio yang tersinkronisasi dengan WinForms sehingga pengimplementasian aplikasi semakin lebih mudah. Aplikasi yang dibuat dengan WinForms ini hanya bisa dijalankan di sistem operasi Windows saja. Pengembangan aplikasi ini menggunakan bahasa pemrograman C#.

BAB III

PENJELASAN ALGORITMA

3.1 Langkah-Langkah Pemecahan Masalah

Pada tugas ini, persoalan dapat direpresentasikan sebagai suatu graf. Struktur data yang digunakan untuk membuat graf dalam permasalahan ini adalah matriks. Matriks akan merepresentasikan peta dari *maze treasure hunt*. Terdapat empat jenis simpul pada permasalahan ini. Pertama, simpul titik awal pencarian. Simpul ini adalah simpul dimana Mr. Krab mulai mencari semua harta karun. Kedua, simpul *treasure*. Simpul ini adalah simpul yang memiliki harta karun. Ketiga, simpul lintasan. Simpul ini adalah simpul yang dapat dilalui oleh Mr. Krab. Terakhir, simpul yang tidak dapat diakses oleh Mr. Krab. Masalah ini dipecahkan dengan mencari rute yang memperoleh seluruh treasure pada *maze*. Untuk prioritas arah simpul yang dibangkitkan, digunakan prioritas LURD (Left-Up-Right-Down).

Persoalan dapat dipecahkan dengan dua algoritma berbeda yaitu BFS dan DFS. Untuk pencarian dengan BFS, terdapat beberapa informasi yang perlu diatasi. Pertama, karena kita ketahui berapa banyak treasures dalam maze, kita dapat memberhentikan pencarian jika semua treasure sudah ditemukan. Kedua, karena goal di dalam pencarian ini bisa lebih dari satu, maka sebuah simpul bisa saja dikunjungi lebih dari satu kali. Oleh karena itu, perlu dikembangkan algoritma dari BFS ini. Beberapa perkembangan yang bisa diimplementasikan adalah penggunaan struktur data Priority Queue dan pemberhentian pencarian ketika semua goal telah tercapai. Priority dari priority queue disini merepresentasikan berapa banyak simpul tersebut telah dikunjungi. Agar BFS bisa lebih *optimize*, maka kita perlu memprioritaskan terlebih dahulu simpul yang belum dikunjungi agar tidak membangkitkan simpul yang telah dikunjungi jika memang tidak diperlukan. Dari informasi-informasi tersebut, terdapat berbagai langkah sebagai berikut

1. Tentukan simpul pertama sebagai starting point dari pencarian dengan BFS.
2. Masukkan simpul pertama ke dalam sebuah priority queue sebagai urutan pencarian dengan priority 0 (belum pernah dikunjungi).
3. Mulai iterasi pemeriksaan dari simpul pertama dari queue (paling depan), periksalah apakah simpul tersebut sudah dikunjungi atau tidak. Jika tidak, tandai bahwa simpul saat ini sudah pernah dikunjungi. Jika sudah, tidak perlu diperiksa lagi.
4. Lakukan pemeriksaan apakah memiliki treasure atau tidak. Jika tidak terdapat treasure, lakukan pencarian BFS kepada seluruh simpul yang bertetangga dengan simpul saat ini.
5. Jika terdapat treasure, berikan tanda bahwa telah ditemukan sebuah treasure. Lakukan ulang BFS dari awal lagi dengan me-reset semua penanda telah dikunjungi, menghapus seluruh isi dari priority queue, dan simpul treasure saat ini sebagai *starting point*.
6. Masukkan seluruh simpul yang bertetangga tersebut ke dalam priority queue dengan prioritas pemasukkan secara Left-Up-Right-Down (Karena graf berbentuk segi empat, maka left disini berarti simpul tetangga yang berada di sebelah kirinya) jika ada. Jangan lupa bahwa priority dari setiap simpul diisi dengan banyaknya simpul tersebut telah dikunjungi.
7. Lakukan langkah 3-6 hingga semua treasure ditemukan.

Dengan langkah-langkah diatas, dapat dipastikan bahwa simpul yang dikunjungi terlebih dahulu adalah simpul yang dikunjunginya lebih sedikit sehingga BFS dapat teroptimisasi.

Untuk algoritma traversal dengan DFS, juga digunakan optimisasi dengan pemberhentian pencarian jika semua treasure telah ditemukan. Langkah-langkahnya adalah sebagai berikut

1. Tentukan simpul pertama sebagai starting point dari pencarian dengan DFS.

2. Mulai iterasi pemeriksaan dari simpul pertama, periksalah apakah simpul tersebut sudah dikunjungi atau belum. Jika belum, tandai bahwa simpul tersebut telah dikunjungi. Jika sudah, tidak perlu diperiksa lagi.
3. Lakukan pemeriksaan apakah simpul tersebut merupakan simpul treasure atau bukan. Jika iya, tandai bahwa telah ditemukan sebuah treasure.
4. Lakukan DFS terhadap seluruh simpul yang bertetangga dengan simpul sekarang. Prioritas dari pencarian adalah Left-Up-Right-Down. Lakukan pencarian secara **rekursif** dengan arti bahwa pemanggilan fungsi DFS terjadi secara berurutan berdasarkan prioritas.
5. Jika semua tetangga dari simpul sudah diperiksa, akan terjadi backtrack ke simpul-simpul sebelumnya untuk memeriksa simpul tetangga lainnya. Backtrack ini akan terjadi secara otomatis karena pemanggilan fungsi dilakukan secara rekursif.
6. Ulangi langkah 2-5 hingga semua treasure telah ditemukan.

Pencarian treasure secara DFS tidak perlu menggunakan prioritas karena DFS dilakukan secara rekursif sehingga selalu dilakukan backtrack jika sudah diperiksa seluruh simpul tetangganya.

3.2 Proses Mapping Persoalan untuk BFS dan DFS

Persoalan yang diberikan pada aplikasi adalah sebuah konfigurasi peta labirin. Peta labirin tersebut akan dikonversikan menjadi sebuah graf statis yang direpresentasikan dalam sebuah matriks (array dua dimensi) yang setiap elemennya merupakan sebuah character yang menandakan apakah simpul tersebut adalah jalur, halangan, titik mulai, atau harta karun. Untuk setiap simpulnya, diperlukan suatu array dua dimensi juga untuk penanda apakah simpul tersebut telah dikunjungi atau belum.

3.2.1 Breadth First Search

Untuk algoritma BFS, diperlukan struktur data Priority Queue untuk menentukan urutan proses dari pengunjungan simpul. Oleh karena itu, selama pemrosesan BFS berlangsung, diperlukan enqueue suatu elemen priority queue yang berisi suatu simpul dan prioritas berupa suatu integer. Untuk menyimpan lintasan menuju suatu simpul, digunakan juga priority queue dengan elemennya adalah lintasan tersebut (lintasan mengandung urutan dari simpul mulai hingga simpul tersebut). Diperlukan juga suatu array dua dimensi untuk menghitung berapa kali suatu simpul telah dikunjungi.

3.2.2 Depth First Search

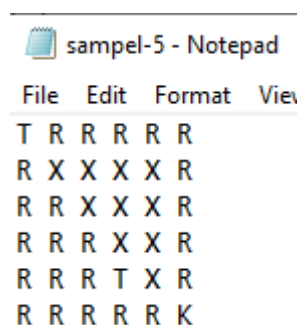
Untuk algoritma DFS, tidak diperlukan struktur data yang spesifik karena proses dilakukan secara rekursif sehingga digunakan array yang merepresentasikan lintasan ke simpul saat itu dari simpul mulai. Pemeriksaan tidak dilakukan secara “teleport” sehingga tidak diperlukan juga array untuk menghitung berapa kali suatu simpul telah dikunjungi.

3.3 Ilustrasi Kasus

Dalam aplikasi ini, Fatih akan berperan sebagai Mr. Krab dan Onodera akan berperan sebagai harta karun.

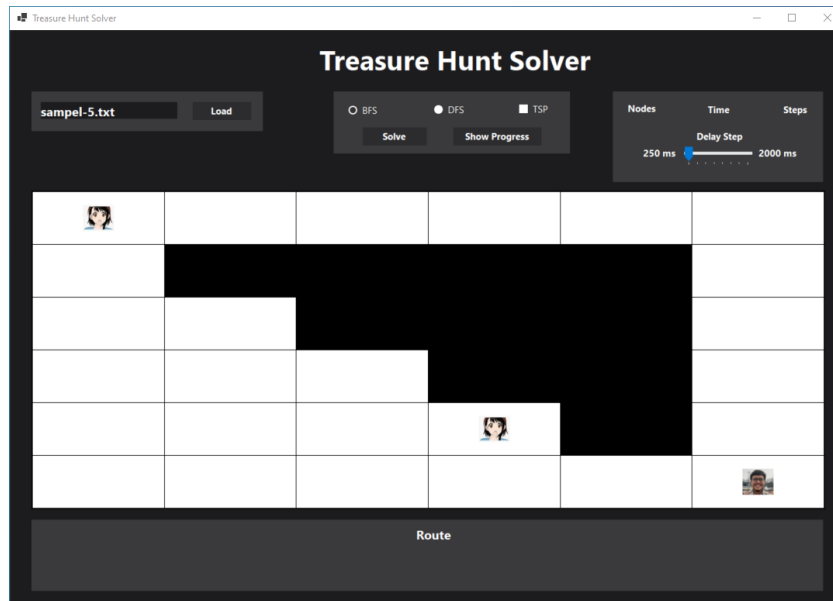
3.3.1 Kasus BFS

Berikut adalah konfigurasi peta labirin untuk suatu kasus dalam file .txt



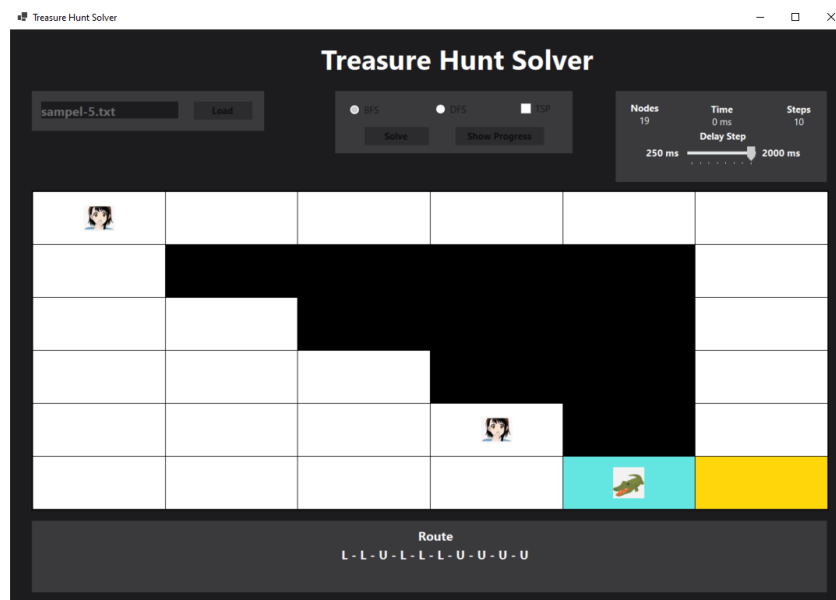
```
sampel-5 - Notepad
File Edit Format View
T R R R R R
R X X X X R
R R X X X R
R R R X X R
R R R T X R
R R R R R K
```

dan berikut adalah tampilan peta labirin dalam aplikasi

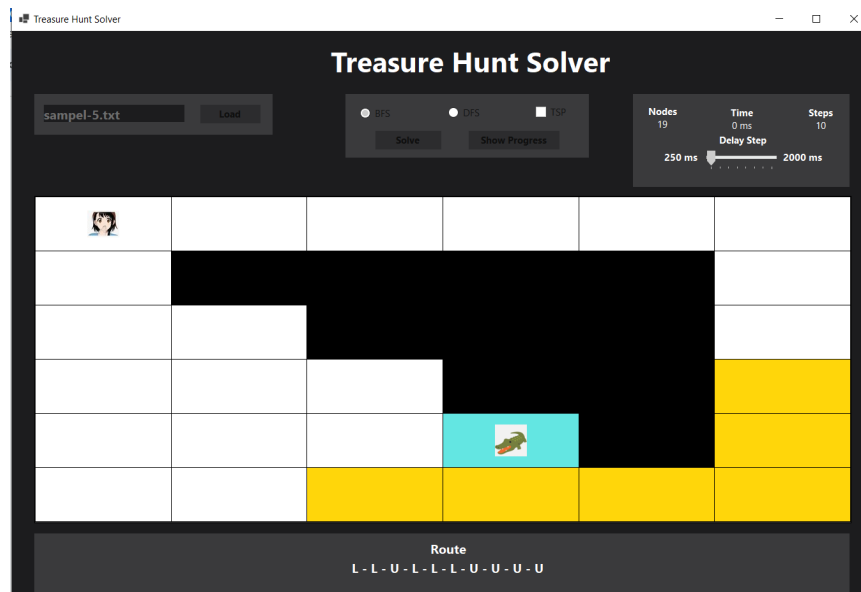
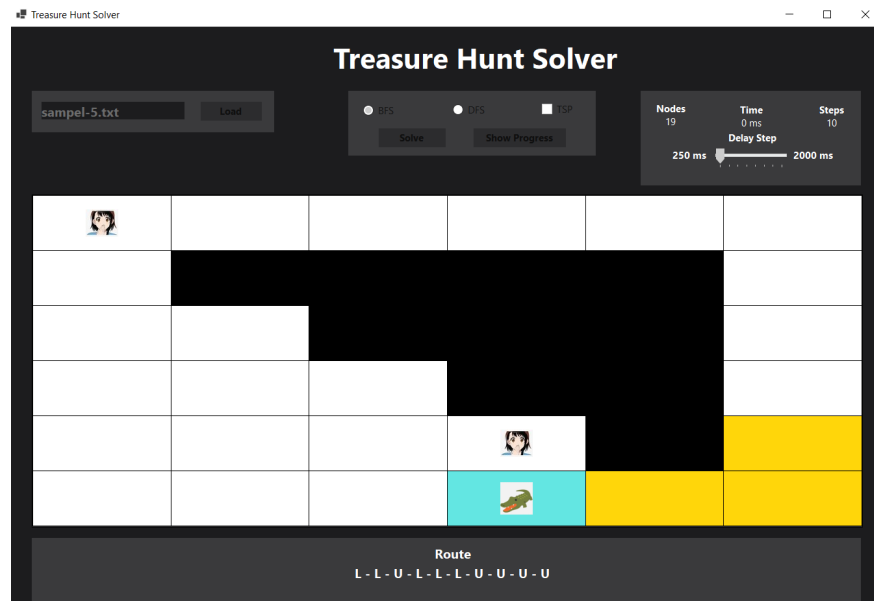


Dari peta tersebut, terlihat bahwa terdapat dua harta karun yang perlu dicari. Berikut adalah langkah-langkah yang akan dilalui dalam penelusuran menggunakan BFS.

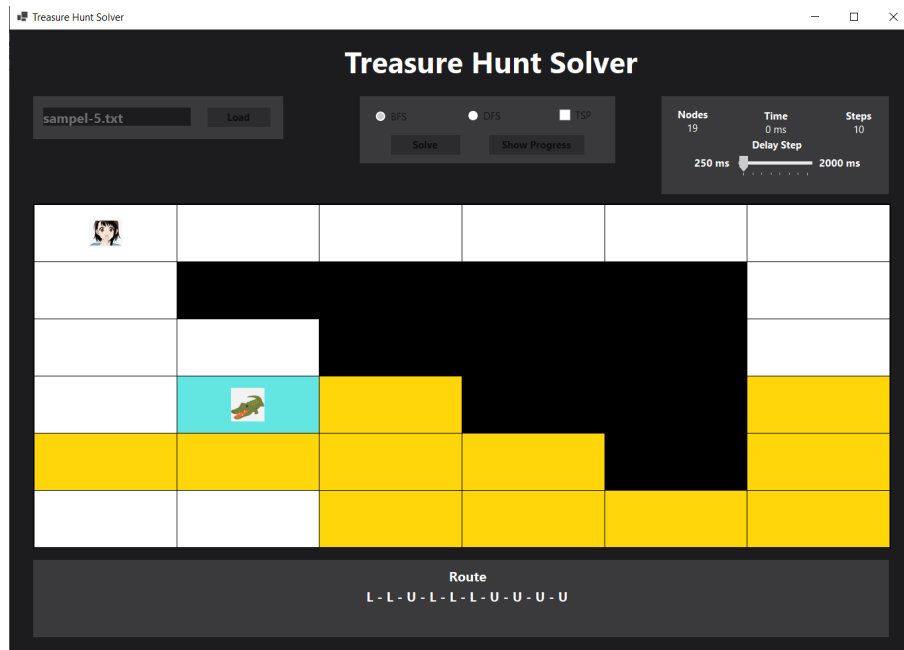
1. Prioritas pergerakan BFS adalah L-U-R-D, maka Fatih akan melakukan searching mulai dari petak di kirinya terlebih dahulu setelah itu di atasnya.



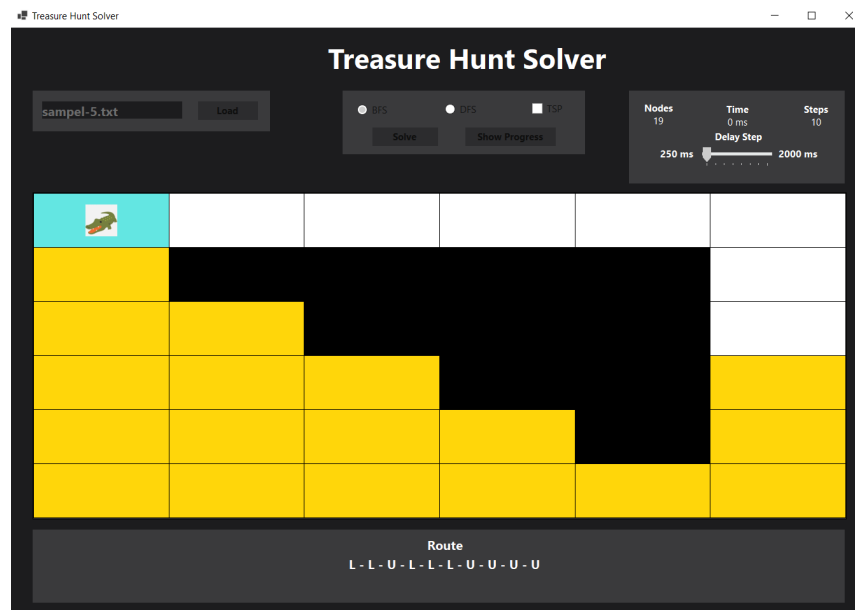
2. Karena BFS merupakan algoritma traversal graf yang menyebar, maka pencarian akan dilakukan secara menyebar juga pada labirin sehingga Fatih akan memeriksa setiap tetangga dari node yang terakhir ia kunjungi lalu berpindah ke setiap tetangga dari node berdasarkan antrian dari Priority Queue.



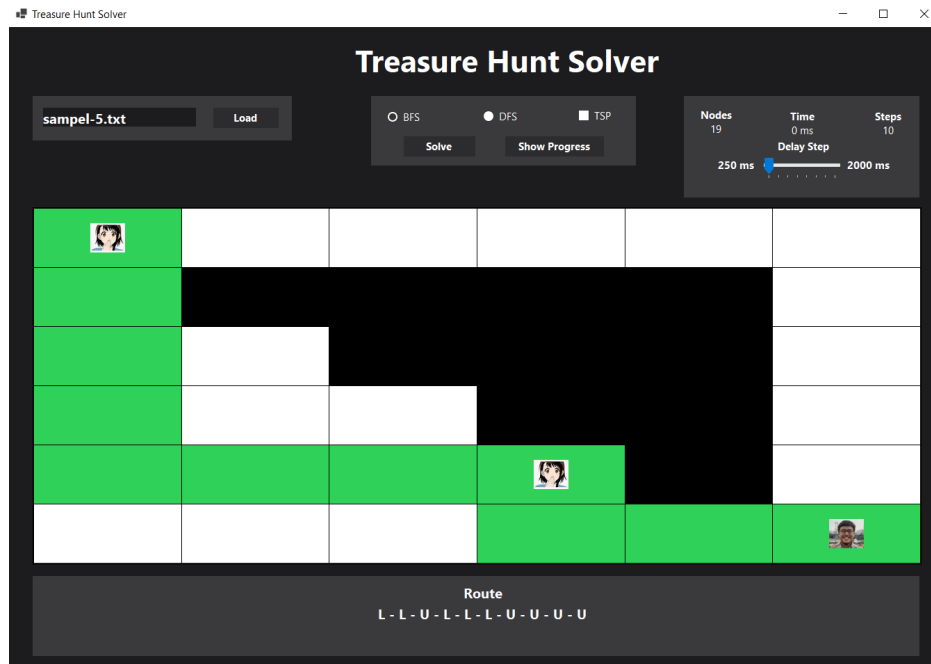
- Setelah Fatih mendapatkan harta karunnya yang pertama, maka Fatih akan melakukan algoritma BFS secara ulang, mulai dari petak harta karun tersebut. Namun, karena petak tersebut memiliki tetangga yang pernah ia kunjungi, maka berdasarkan aturan priority, petak yang belum pernah dikunjungi lebih diprioritaskan sehingga Fatih akan lanjut menelusuri petak-petak yang belum dikunjungi sampai ia mentok.



4. Pencarian tetap dilakukan secara menyebar kepada petak-petak yang belum pernah ia kunjungi hingga akhirnya Fatih menemukan harta karun terakhir.



Sehingga untuk kasus peta labirin di atas dengan BFS, ditemukan rute solusi sebagai berikut.



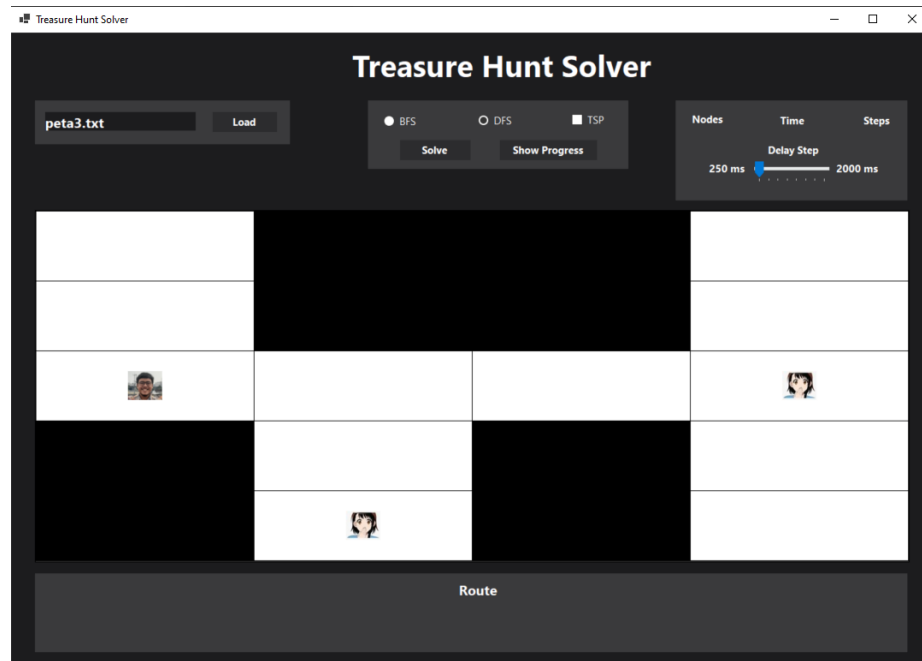
3.3.2 Kasus DFS

Berikut adalah konfigurasi peta labirin untuk suatu kasus dalam file .txt

peta3 - Notepad

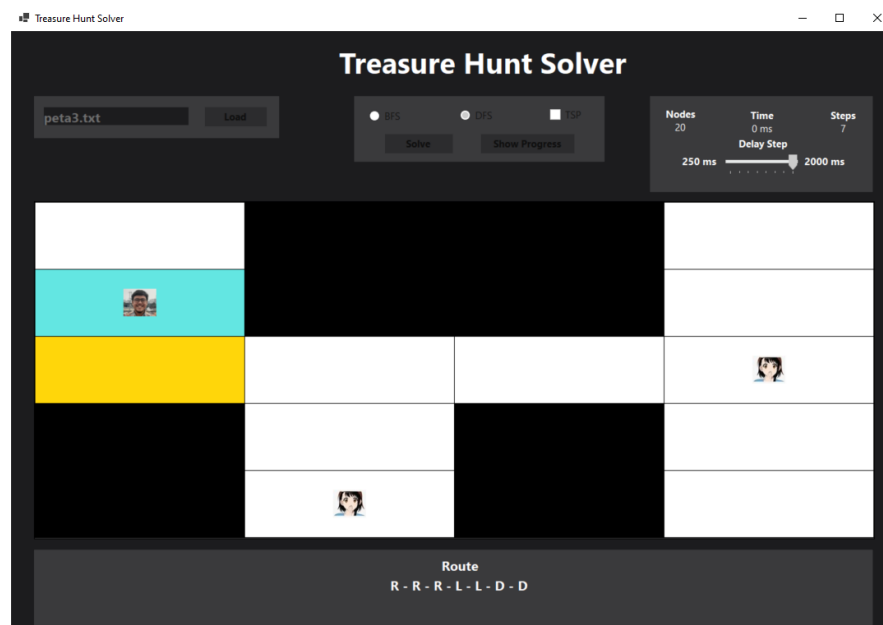
File	Edit	Format
R	X	X R
R	X	X R
K	R	R T
X	R	X R
X	T	X R

dan berikut adalah tampilan peta labirin dalam aplikasi

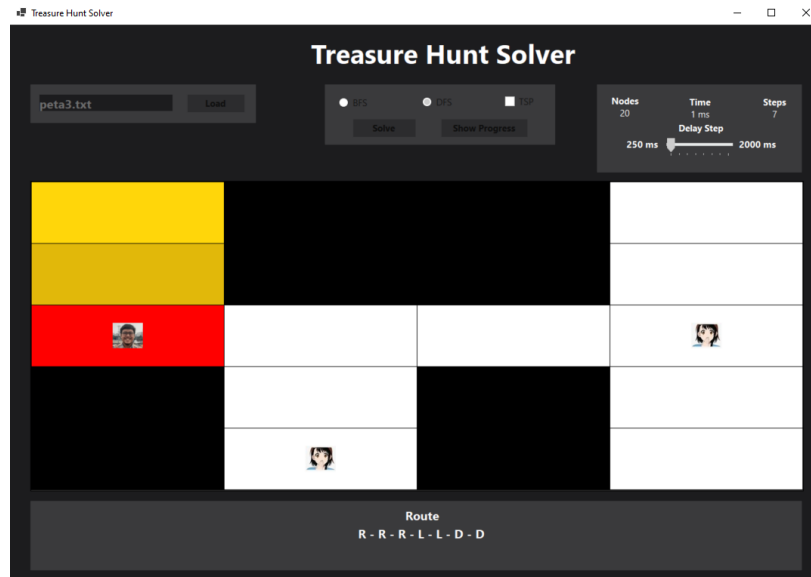


Dari peta tersebut, terlihat bahwa terdapat dua harta karun yang perlu dicari. Berikut adalah langkah-langkah yang akan dilalui dalam penelusuran menggunakan DFS.

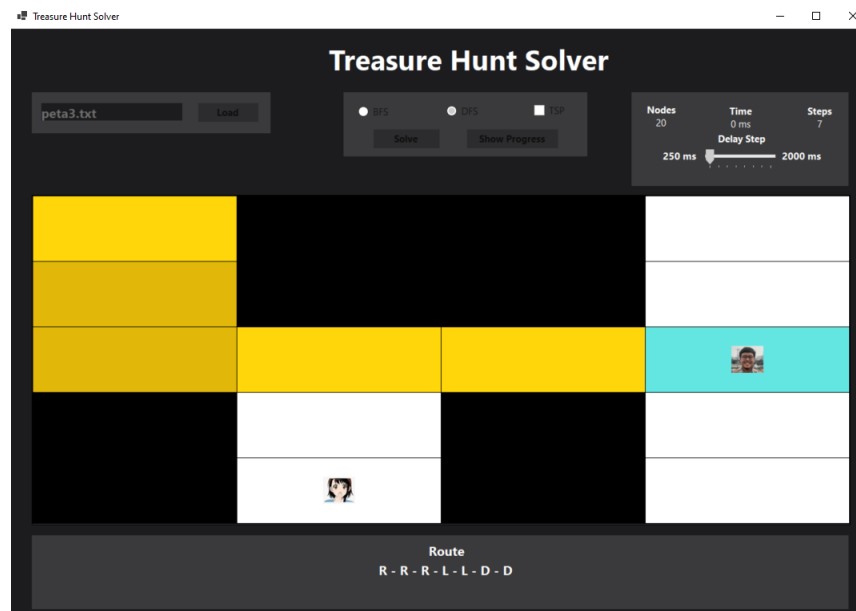
1. Karena prioritas pergerakan DFS adalah L-U-R-D, maka Fatih akan pindah ke petak atas terlebih dahulu.



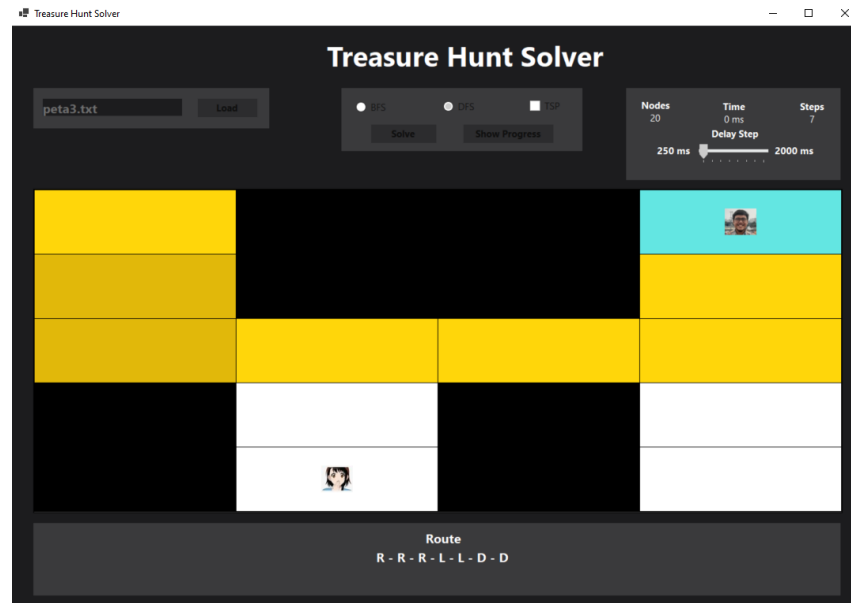
2. Fatih akan pindah ke petak paling atas hingga tidak bisa pindah petak lagi sehingga Fatih akan backtrack hingga ke petak yang memiliki tetangga yang bisa ditelusuri yaitu pada starting point.



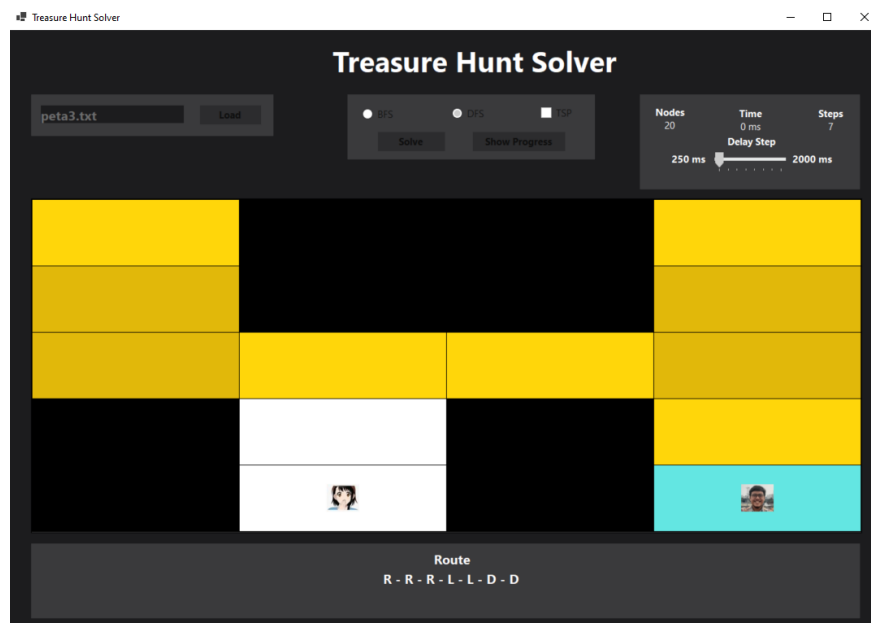
3. Kemudian, Fatih akan bergerak ke kanan terus hingga menemukan harta karun pertamanya. (Karena tidak bisa naik ke petak atas selama bergerak ke kanan)



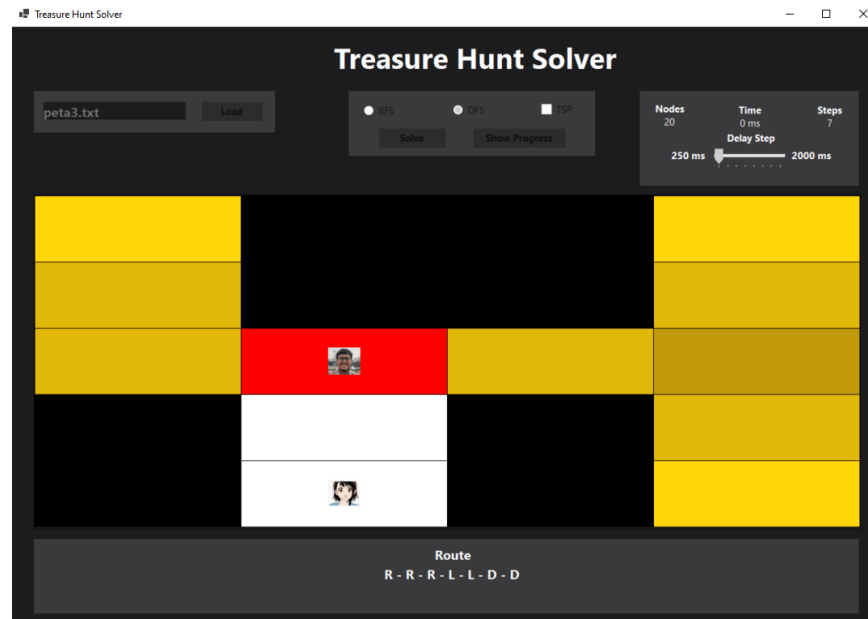
4. Setelah mendapatkan harta karun pertamanya, Fatih akan naik ke petak atas dan backtrack hingga ke posisi harta karun pertama karena sudah tidak bisa ke petak lain lagi dan tidak terdapat harta karun.



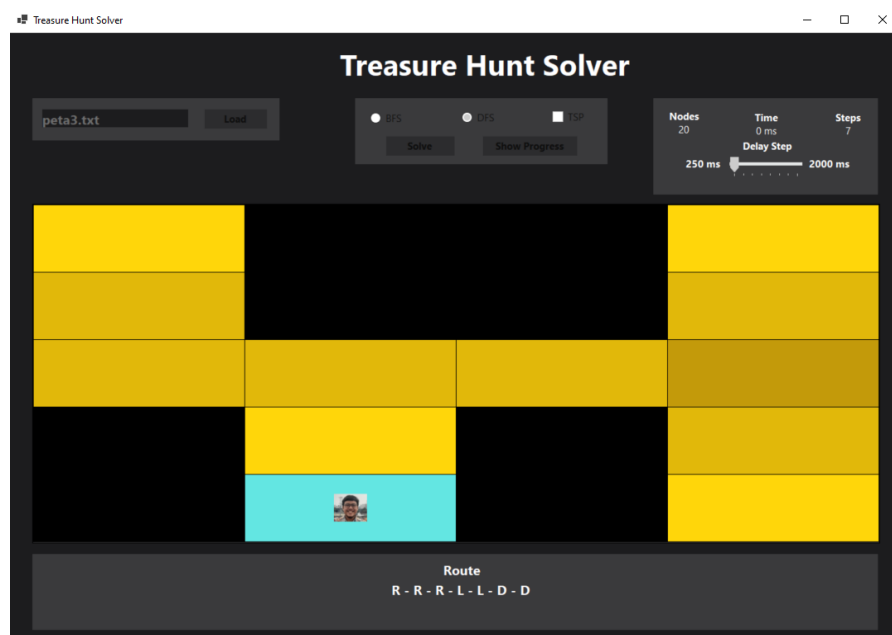
5. Fatih juga akan turun ke petak bawah dan backtrack hingga ke posisi harta karun pertama lagi karena tidak ditemukan treasure.



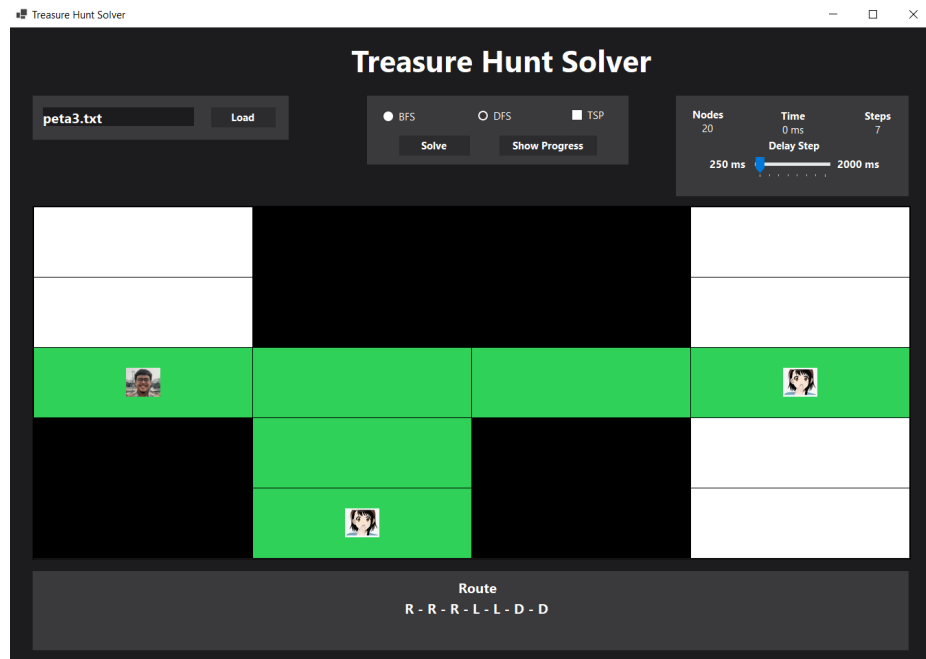
6. Karena sudah tidak bisa ke petak tetangga lainnya, Fatih backtrack hingga ke petak yang memiliki tetangga yang belum dikunjungi.



7. Kemudian Fatih turun ke petak bawah untuk mendapatkan harta karun terakhir.



Sehingga dengan kasus peta labirin di atas dengan DFS, didapatkan rute solusi sebagai berikut



BAB IV

ANALISIS PEMECAHAN MASALAH

4.1 Implementasi Program

Untuk melihat implementasi User Interface dan program lainnya bisa dilihat pada tautan repository Github di bagian lampiran.

4.1.1 BFS

```
procedure bfs(input map, input current, input isVisited, input/output count,
input/output path, input track, input/output solution, input/output progress,
input/output nodes, input M, input N, input treasures, input/output countVisit)
```

KAMUS LOKAL

```
map : array of array of char
current : Priority Queue of Tuple(integer, integer)
isVisited : array of array of boolean
count, M, N, treasures, nodes, x, y, countVisitQueue : integer
path, solution, newTrack, newTrack2, newTrack3, newTrack4 : array of
Tuple(integer, integer)
track : array of array of Tuple(integer, integer)
progress : array of Tuple(integer, integer, string)
countVisit : array of array of integer
```

ALGORITMA

```
while(current.getNeff()  $\neq$  0) do
{ If the queue is empty, no more nodes can be visited OR all treasures have been
found }
  if (current.getNeff() = 0 or count = treasures) then
    break
  { Getting the current coordinate }
  coor  $\leftarrow$  current.Dequeue()
  x  $\leftarrow$  coor.Item1
  y  $\leftarrow$  coor.Item2

  { Getting the current track from starting point }
  currentTrack  $\leftarrow$  track.Dequeue()

  { If map is not out of bounds }
  if (x  $\geq$  0 and x < M and y  $\geq$  0 and y < N) then
    { If map is already visited or blocked }
    if (isVisited[x][y] or map[x][y] = 'X') then
      { Do nothing }
    else
      { Visit map }
      nodes  $\leftarrow$  nodes + 1
      isVisited[x][y]  $\leftarrow$  True
      countVisit[x][y]  $\leftarrow$  countVisit[x][y] + 1
      progress  $\leftarrow$  progress + (x, y, "GREEN")

      { If it contains treasure }
```

```

if (map[x][y] = 'T') then
  { Add the amount of collected treasures }
  count ← count + 1
  { Add current coordinate to solutions }
  solution ← solution + (x, y)
  { Save path }
  path ← path + currentTrack

  { Reset settings (Restart the search process from the current treasure) }
  i traversal [0..M-1]
    j traversal [0..N-1]
      isVisited[i][j] ← False

  { Collect Treasure }
  map[x][y] ← 'R'

  { Reset lists and queue }
  track.Clear()
  current.Clear()
  currentTrack.Clear()

  { Visit current node }
  isVisited[x][y] ← True

  { BFS on L-U-R-D pattern }
  { Enqueue Left Node }
  if (y - 1 ≥ 0) then
    countVisitQueue ← countVisit[x][y - 1]
  else
    countVisitQueue ← 0

  current.Enqueue((x, y - 1), countVisitQueue)

  { Create a new track to left node from starting point }
  newTrack ← currentTrack
  newTrack ← newTrack + (x, y - 1)

  { Enqueue the new track }
  track.Enqueue(newTrack, countVisitQueue)

  { Enqueue Upper Node }
  if (x - 1 ≥ 0) then
    countVisitQueue ← countVisit[x - 1][y]
  else
    countVisitQueue ← 0

  current.Enqueue((x - 1, y), countVisitQueue)

  { Create a new track to upper node from starting point }
  newTrack2 ← currentTrack
  newTrack2 ← newTrack2 + (x - 1, y)

  { Enqueue the new track }
  track.Enqueue(newTrack2, countVisitQueue)

  { Enqueue Right Node }
  if (y + 1 ≥ 0) then

```

```

    countVisitQueue ← countVisit[x][y + 1]
else
    countVisitQueue ← 0

current.Enqueue((x, y + 1), countVisitQueue)

{ Create a new track to right node from starting point }
newTrack3 ← currentTrack
newTrack3 ← newTrack3 + (x, y + 1)

{ Enqueue the new track }
track.Enqueue(newTrack3, countVisitQueue)

{ Enqueue Lower Node }
if (x + 1 ≥ 0) then
    countVisitQueue ← countVisit[x + 1][y]
else
    countVisitQueue ← 0

current.Enqueue((x + 1, y), countVisitQueue)

{ Create a new track to lower node from starting point }
newTrack4 ← currentTrack
newTrack4 ← newTrack4 + (x + 1, y)

{ Enqueue the new track }
track.Enqueue(newTrack4, countVisitQueue)

```

4.1.2 tspBFS

procedure tspBFS(input map, input current, input isVisited, input/output path, input track, input/output progress, input/output nodes, input M, input N, input/output countVisit)

KAMUS LOKAL

map : array of array of char
current : Priority Queue of Tuple(integer, integer)
isVisited : array of array of boolean
count, M, N, nodes, x, y, countVisitQueue : integer
path, newTrack, newTrack2, newTrack3, newTrack4 : array of Tuple(integer, integer)
track : array of array of Tuple(integer, integer)
progress : array of Tuple(integer, integer, string)
countVisit : array of array of integer

ALGORITMA

```

while(current.getNeff() ≠ 0) do
{ If the queue is empty, no more nodes can be visited }
if (current.getNeff() = 0) then
    break
{ Getting the current coordinate }
coor ← current.Dequeue()
x ← coor.Item1
y ← coor.Item2

```

```

{ Getting the current track from starting point }
currentTrack ← track.Dequeue()

{ If map is not out of bounds }
if (x ≥ 0 and x < M and y ≥ 0 and y < N) then
  { If map is already visited or blocked }
  if (isVisited[x][y] or map[x][y] = 'X') then
    { Do nothing }
  else
    { Skips the first search on last treasure }
    if (x ≠ this.startingX or y ≠ this.startingY) then
      { Visit map }
      nodes ← nodes + 1
      isVisited[x][y] ← True
      countVisit[x][y] ← countVisit[x][y] + 1
      progress ← progress + (x, y, "GREEN")

      { Reached starting point }
      if (map[x][y] = 'K') then
        path ← path + currentTrack
        exit
      else
        isVisited[x][y] ← True

{ TSP BFS on L-U-R-D pattern }
{ Enqueue Left Node }
if (y - 1 ≥ 0) then
  countVisitQueue ← countVisit[x][y - 1]
else
  countVisitQueue ← 0

current.Enqueue((x, y - 1), countVisitQueue)

{ Create a new track to left node from starting point }
newTrack ← currentTrack
newTrack ← newTrack + (x, y - 1)

{ Enqueue the new track }
track.Enqueue(newTrack, countVisitQueue)

{ Enqueue Upper Node }
if (x - 1 ≥ 0) then
  countVisitQueue ← countVisit[x - 1][y]
else
  countVisitQueue ← 0

current.Enqueue((x - 1, y), countVisitQueue)

{ Create a new track to upper node from starting point }
newTrack2 ← currentTrack
newTrack2 ← newTrack2 + (x - 1, y)

{ Enqueue the new track }
track.Enqueue(newTrack2, countVisitQueue)

{ Enqueue Right Node }

```

```

if (y + 1 ≥ 0) then
    countVisitQueue ← countVisit[x][y + 1]
else
    countVisitQueue ← 0

current.Enqueue((x, y + 1), countVisitQueue)

{ Create a new track to right node from starting point }
newTrack3 ← currentTrack
newTrack3 ← newTrack3 + (x, y + 1)

{ Enqueue the new track }
track.Enqueue(newTrack3, countVisitQueue)

{ Enqueue Lower Node }
if (x + 1 ≥ 0) then
    countVisitQueue ← countVisit[x + 1][y]
else
    countVisitQueue ← 0

current.Enqueue((x + 1, y), countVisitQueue)

{ Create a new track to lower node from starting point }
newTrack4 ← currentTrack
newTrack4 ← newTrack4 + (x + 1, y)

{ Enqueue the new track }
track.Enqueue(newTrack4, countVisitQueue)

```

4.1.3 DFS

procedure dfs(input map, input x, input y, input xBefore, input yBefore, input isVisited, input/output count, input/output path, input track, input/output solution, input/output progress, input/output nodes, input M, input N, input treasures)

KAMUS LOKAL

map : array of array of char
isVisited : array of array of boolean
count, M, N, treasures, nodes, x, y, xBefore, yBefore : integer
path, solution : array of Tuple(integer, integer)
track : array of Tuple(integer, integer)
progress : array of Tuple(integer, integer, string)

ALGORITMA

```

{ If map is not out of bounds }
if (x ≥ 0 and x < M and y ≥ 0 and y < N) then
    { If map is already visited or blocked }
    if (isVisited[x][y] or map[x][y] = 'X') then
        { Do nothing }
    else
        { Visit map }
        nodes ← nodes + 1
        isVisited[x][y] ← True

```

```

progress ← progress + (x, y, "GREEN")

{ Add map to track }
track ← track + (x, y)

{ If it contains treasure }
if (map[x, y] = 'T')
    count ← count + 1
    solution ← solution + (x, y)
    path ← path + track
    track.Clear()
    if (count = treasures) then
        exit

{ DFS on L-U-R-D pattern }
dfs(map, x, y - 1, x, y, isVisited, count, path, track, solution, progress,
nodes, M, N, treasures)

{ Stops searching if all treasures have been found }
if (count = treasures) then
    exit

{ Add backtrack to progress }
if (progress.Last()[0] ≠ x or progress.Last()[1] ≠ y) then
    progress ← progress + (x, y, "RED")
    nodes ← nodes + 1

dfs(map, x - 1, y, x, y, isVisited, count, path, track, solution, progress,
nodes, M, N, treasures)

{ Stops searching if all treasures have been found }
if (count = treasures) then
    exit

{ Add backtrack to progress }
if (progress.Last()[0] ≠ x or progress.Last()[1] ≠ y) then
    progress ← progress + (x, y, "RED")
    nodes ← nodes + 1

dfs(map, x, y + 1, x, y, isVisited, count, path, track, solution, progress,
nodes, M, N, treasures)

{ Stops searching if all treasures have been found }
if (count = treasures) then
    exit

{ Add backtrack to progress }
if (progress.Last()[0] ≠ x or progress.Last()[1] ≠ y) then
    progress ← progress + (x, y, "RED")
    nodes ← nodes + 1

dfs(map, x + 1, y, x, y, isVisited, count, path, track, solution, progress,
nodes, M, N, treasures)

{ Stops searching if all treasures have been found }
if (count = treasures) then
    exit

```



```

    { Add backtrack to progress }
    if (progress.Last()[0]  $\neq$  x or progress.Last()[1]  $\neq$  y) then
        progress  $\leftarrow$  progress + (x, y, "RED")
        nodes  $\leftarrow$  nodes + 1
    { If track is blocked and nowhere to continue }
    if (xBefore - 1  $\geq$  0 and xBefore + 1 < M and yBefore - 1  $\geq$  0 and yBefore + 1
    < N and isVisited[xBefore - 1][yBefore] and isVisited[xBefore][yBefore + 1] and
    isVisited[xBefore + 1][yBefore] and isVisited[xBefore][yBefore - 1]) then
        track.Clear()
    else if (track.Count > 0) then
        { Remove current map from track and continue add another map to track }
        track.RemoveAt(track.Count - 1)

    { Pop last backtrack to starting point }
    if (x == this.startingX and y == this.startingY) then
        startingCoor  $\leftarrow$  (solution.Last()[0], solution.Last()[1], "GREEN")
        while (!progress.Last().Equals(startingCoor)) do
            progress.RemoveAt(progress.Count - 1)
            nodes  $\leftarrow$  nodes - 1

```

4.1.4 tspDFS

procedure tspDFS(input map, input x, input y, input xBefore, input yBefore, input isVisited, input/output path, input track, input/output finish, input/output progress, input/output nodes, input M, input N)

KAMUS LOKAL

map : array of array of char
isVisited : array of array of boolean
M, N, nodes, x, y, xBefore, yBefore : integer
path : array of Tuple(integer, integer)
track : array of Tuple(integer, integer)
progress : array of Tuple(integer, integer, string)
finish : boolean

ALGORITMA

```

{ If map is not out of bounds }
if (x  $\geq$  0 and x < M and y  $\geq$  0 and y < N) then
    { If map is already visited or blocked }
    if (isVisited[x][y] or map[x][y] == 'X') then
        { Do nothing }
    else
        { Skips the first search on the last treasure }
        if (x  $\neq$  xBefore or y  $\neq$  yBefore) then
            { Visit map }
            nodes  $\leftarrow$  nodes + 1
            isVisited[x][y]  $\leftarrow$  True
            progress  $\leftarrow$  progress + (x, y, "GREEN")

            { Add map to track }
            track  $\leftarrow$  track + (x, y)

            { If it reached starting point }

```

```

        if (map[x][y] == 'K') then
            finish ← True
            path ← path + track
            track.Clear()
            exit
        else
            isVisited[x][y] ← True

        { DFS on L-U-R-D pattern }
        tspDFS(map, x, y - 1, x, y, isVisited, path, track, finish, progress, nodes,
M, N)

        { Stops searching if reached starting point }
        if (finish) then
            exit

        { Add backtrack to progress }
        if (progress.Last()[0] ≠ x or progress.Last()[1] ≠ y) then
            progress ← progress + (x, y, "RED")
            nodes ← nodes + 1

        tspDFS(map, x - 1, y, x, y, isVisited, path, track, finish, progress, nodes,
M, N)

        { Stops searching if all treasures have been found }
        if (finish) then
            exit

        { Add backtrack to progress }
        if (progress.Last()[0] ≠ x or progress.Last()[1] ≠ y) then
            progress ← progress + (x, y, "RED")
            nodes ← nodes + 1

        tspDFS(map, x, y + 1, x, y, isVisited, path, track, finish, progress, nodes,
M, N)

        { Stops searching if all treasures have been found }
        if (finish) then
            exit

        { Add backtrack to progress }
        if (progress.Last()[0] ≠ x or progress.Last()[1] ≠ y) then
            progress ← progress + (x, y, "RED")
            nodes ← nodes + 1

        tspDFS(map, x + 1, y, x, y, isVisited, path, track, finish, progress, nodes,
M, N)

        { Stops searching if all treasures have been found }
        if (finish) then
            exit

        { Add backtrack to progress }
        if (progress.Last()[0] ≠ x or progress.Last()[1] ≠ y) then
            progress ← progress + (x, y, "RED")
            nodes ← nodes + 1

        { If track is blocked and nowhere to continue }

```

```

    if (xBefore - 1  $\geq$  0 and xBefore + 1 < M and yBefore - 1  $\geq$  0 and yBefore + 1
    < N and isVisited[xBefore - 1][yBefore] and isVisited[xBefore][yBefore + 1] and
    isVisited[xBefore + 1][yBefore] and isVisited[xBefore][yBefore - 1]) then
        track.Clear()
    else if (track.Count > 0) then
        { Remove current map from track and continue add another map to track }
        track.RemoveAt(track.Count - 1)

```

4.1.5 Helper

```

function DataTableFromTextFile(input location)
{ Function to convert txt file to DataTable }

```

KAMUS LOKAL

```

i, j : integer
graph : Graph
result : DataTable
dc : DataColumn
dr : DataRow

```

ALGORITMA

```

graph  $\leftarrow$  Graph(location)

j traversal [0..graph.getN()-1]
    result.Columns  $\leftarrow$  result.Columns + dc

i traversal [0..graph.getM()-1]
    j traversal [0..graph.getN()-1]
        if (graph.getMap()[i][j] = 'X') then
            dr[j]  $\leftarrow$  Image.FromFile("../src/images/empty.png")
        else if (graph.getMap()[i][j] = 'R') then
            dr[j]  $\leftarrow$  Image.FromFile("../src/images/empty.png")
        else if (graph.getMap()[i][j] = 'K') then
            dr[j]  $\leftarrow$  Image.FromFile("../src/images/fatih.png")
        else if (graph.getMap()[i][j] = 'T') then
            dr[j]  $\leftarrow$  Image.FromFile("../src/images/wibu.jpg")
        result.Rows  $\leftarrow$  result.Rows + dr

 $\rightarrow$  result

```

```

procedure printMap(input map, input M, input N)
{ Function to print the map }

```

KAMUS LOKAL

```

i, j, M, N : integer
map : array of array of char

```

ALGORITMA

```

i traversal [0..M-1]
    j traversal [0..N-1]
        output(map[i][j])
    output(newline)

```

```

procedure memset(input/output buffer, input value, input M, input N)

```

```
{ Function to set a buffer with the given value }
```

KAMUS LOKAL

i, j, M, N : integer
buffer : array of array of char
value : char

ALGORITMA

```
i traversal [0..M-1]
  j traversal [0..N-1]
    buffer[i][j] ← value
```

procedure setSolution(input/output buffer, input path, input solution, input start)

{ Set map with solution path }

KAMUS LOKAL

coor, start : tuple(integer,integer)
buffer : array of array of char
path, solution : array of tuple(integer,integer)

ALGORITMA

{Function to plot the path to the treasures}

```
foreach (coor in path)
  buffer[coor[0]][coor[1]] ← 'P'
```

```
foreach (coor in solution)
  buffer[coor[0]][coor[1]] ← 'T'
```

```
buffer[start[0]][start[1]] ← 'K'
```

procedure setSolutionTSP(input/output buffer, input path, input start)

{ Set map with solution TSP path }

KAMUS LOKAL

coor, start : tuple(integer,integer)
buffer : array of array of char
path : array of tuple(integer,integer)

ALGORITMA

{ Function to plot the path to starting point }

```
foreach (coor in path)
  buffer[coor[0]][coor[1]] ← 'P'
```

```
buffer[start[0]][start[1]] ← 'K'
```

function getStartingPoint(input map, input M, input N)

{Returns the starting point}

KAMUS LOKAL

i, j, M, N : integer
map : array of array of char

ALGORITMA

{Function to get the starting point of the map}

```
i traversal [0..M-1]
  j traversal [0..N-1]
```

```

    if(map[i][j] = 'K') then
        → (i,j)

{If not found}
→ (0,0)

function sequenceMove(input progress)
{Returns the sequence of route}

KAMUS LOKAL
i : integer
firstNode,secondNode : tuple(integer, integer)
sequence : array of char

ALGORITMA
i traversal [0..length(progress)-2]
firstNode ← progress[i]
secondNode ← progress[i + 1]
if (firstNode[0] = secondNode[0] and secondNode[1] > firstNode[1]) then
    sequence ← sequence + ('R')
else if (firstNode[0] = secondNode[0] && secondNode[1] < firstNode[1]) then
    sequence ← sequence + ('L')
else if (firstNode[0] > secondNode[0] && secondNode[1] = firstNode[1]) then
    sequence ← sequence + ('U')
else
    sequence ← sequence + ('D')

→ sequence

```

4.1.6 ReadFile

```

function readFile(input pathfile)
{Function to read map from file}

KAMUS LOKAL
lines : array of string
idx, M, N, i, treasures, j : integer
line : string
map : array of array of char

ALGORITMA
{Try to read file}
lines ← read("../../test/" + pathfile)

{Remove spaces from each line}
idx traversal [0..length(lines)-1]
    lines[idx] ← lines[idx].replace(' ', '')

{Validation file}
{1. Empty File}
M ← length(lines)

if (M = 0) then
    throw exception("File kosong!")

N ← length(lines[0])

```

```

{Initialize variable and array}
i ← 0
treasures ← 0
K ← 0

{Loop through all lines}
foreach (line in lines)
  if (length(line) ≠ N) then
    throw exception("Ukuran bukan segi empat!")

  j traversal [0..N-1]
  {3. Unknown character}
  if (line[j] ≠ 'K' and line[j] ≠ 'R' and line[j] ≠ 'T' and line[j] ≠ 'X') then
    throw exception("Karakter tidak sesuai isi dari peta!")
  else
    map[i][j] ← line[j]
    if (line[j] = 'T') then
      treasures ← treasures + 1

  i ← i + 1

if(K = 0) then
  throw exception("Tidak ada starting point pada peta!")
else if (K > 1) then
  throw exception("Starting point terdapat lebih dari satu!")
else if (treasures = 0) then
  throw exception("Tidak ada treasure pada peta!")
{Return map}
→ (map, treasures)

```

4.2 Penjelasan Struktur Data

Penjelasan detail tentang penggunaan struktur data pada salah bagian algoritma dapat dilihat pada bab 3. Bagian ini membahas secara umum fungsi dan kegunaan masing-masing struktur data dalam keseluruhan program.

4.2.1 List

List memiliki fungsi yang bervariasi dalam program kami. List digunakan untuk keperluan berikut :

- Menyimpan sekuens rute, *progress traversal*, dan *tile* yang dikunjungi,
- Pengganti struktur data *stack* dalam algoritma DFS (kenapa tidak menggunakan *stack* adalah karena perlu untuk melihat ke bagian tengah *stack* ketika menghasilkan rute)

4.2.2 Tuple

Tuple memiliki fungsi yang juga beragam dalam program kami, ia digunakan untuk mengelompokkan data-data yang terkait erat. Penggunaan tuple adalah sebagai berikut :

- Tuple dua elemen digunakan untuk menyimpan koordinat x dan y.
- Tuple tiga elemen digunakan untuk menyimpan koordinat x, koordinat y, dan sebuah string yang mengindikasikan apakah koordinat tersebut sudah pernah dikunjungi atau tidak. Tuple jenis ini digunakan dalam pembuatan *progress traversal* yang nantinya ditampilkan di GUI. Sesuai spek, dibutuhkan informasi apakah sebuah koordinat sudah pernah ditampilkan atau belum sehingga struktur data ini dipakai.

4.2.3 PrioQueue

Struktur data ini dipakai dalam algoritma BFS dan di dalam program kami, digunakan untuk menyimpan koordinat peta yang ditemukan selama proses BFS. Kami membuat sebuah struktur data baru untuk mendampingi PrioQueue ini yaitu **ELPrioQueue** yang selain mengandung elemennya, memiliki prioritas dari elemen tersebut. Sehingga untuk melakukan *enqueue* sebuah elemen ke dalam *queue*, ia perlu diubah menjadi **ELPrioQueue** dan didefinisikan prioritasnya. Ini dikarenakan algoritma kami membutuhkan definisi prioritas yang tidak bisa di-*infer* dari koordinat sendiri, melainkan didefinisikan prioritasnya oleh algoritma kami ketika ia akan dimasukkan (detail tentang prioritas dan penggunaan prioritas untuk algoritma BFS ada pada subbab [3.3.2](#) dan [4.1.1](#)) ke dalam *queue*.

```
public class PrioQueue<T>
{
    private int neff;
    private List<ELPrioQueue<T>> queue;

    public PrioQueue()
    {
        this.neff = 0;
        this.queue = new List<ELPrioQueue<T>>();
    }

    public void Enqueue(ELPrioQueue<T> el)
    {
```

```

        if (this.neff == 0)
        {
            this.queue.Add(el);
            this.neff++;
        }
        else
        {
            for (int i = this.neff - 1; i ≥ 0; i--)
            {
                if (el.getPriority() ≥ this.queue.ElementAt(i).getPriority())
                {
                    this.queue.Insert(i + 1, el);
                    this.neff++;
                    return;
                }
            }
            this.queue.Insert(0, el);
            this.neff++;
        }
    }

    public T Dequeue()
    {
        if (this.neff > 0)
        {
            this.neff--;
            ElPrioQueue<T> el = this.queue.First();
            this.queue.RemoveAt(0);
            return el.getCoor();
        }
        else
        {
            throw new Exception("Queue sudah kosong");
        }
    }

    public void Clear()
    {
        this.neff = 0;
        this.queue.Clear();
    }

    public int getNeff()
    {
        return this.neff;
    }

    public List<ElPrioQueue<T>> getQueue()
    {
        return this.queue;
    }
}

```

```

public class ElPrioQueue<T>
{
    private T coor;
    private int priority;

    public ElPrioQueue(T coor, int priority)
    {

```



```

        this.coor = coor;
        this.priority = priority;
    }

    public int getPriority()
    {
        return this.priority;
    }

    public T getCoor()
    {
        return this.coor;
    }
}

```

4.2.4 Graf

Graf digunakan untuk merepresentasikan peta yang ingin dicari prosesnya. Kami menggunakan struktur data matriks dua dimensi untuk merepresentasikan graf yang di-load dari *file* dan kemudian dicari solusinya oleh algoritma kami. Bagian penting dari struktur data ini adalah ia memiliki *method Solve* yang menghasilkan solusi dari dirinya sehingga program kami memecahkan masalah dengan paradigma OOP di mana objek melakukan sesuatu sendiri dan bukan dimanipulasi oleh fungsi eksternal.

```

class Graph
{
    private int M;
    private int N;
    private char[,] map;
    private int treasures;

    public Graph(String pathfile)
    {
        Tuple<char[,], int> config = ReadFile.readFile(pathfile);
        this.map = config.Item1;
        this.treasures = config.Item2;
        this.M = this.map.GetLength(0);
        this.N = this.map.GetLength(1);
    }

    public Graph(Graph other)
    {
        this.map = new char[other.M, other.N];
        for(int i = 0; i < other.M; i++)
        {

```

```

        for(int j = 0; j < other.N; j++)
        {
            this.map[i, j] = other.map[i, j];
        }
    }
    this.treasures = other.treasures;
    this.M = other.M;
    this.N = other.N;
}

public int getM()
{
    return this.M;
}

public int getN()
{
    return this.N;
}

public char[,] getMap()
{
    return this.map;
}
// Method solve bisa dilihat pada source code di tautan Github
}

```

4.2.5 Solution

Struktur data ini adalah pembungkus untuk solusi dari algoritma kami yang mencakup jumlah *nodes* yang dikunjungi, *progress traversal*, rute, jumlah langkah, waktu eksekusi pencarian. Selain itu, struktur data ini juga menyimpan daftar posisi harta karun dan peta. Ia dikembalikan oleh *method Solve* pada struktur data *Graph*.

```

class Solution
{
    private int nodes;
    private List<Tuple<int, int, string>> progress;
    private string route;
    private int steps;
    private System.Diagnostics.Stopwatch executionTime;
    private char[,] solutionMap;
    private List<Tuple<int, int>> treasures;

    public Solution(int nodes, List<Tuple<int, int, string>> progress,
        List<char> route, int steps, System.Diagnostics.Stopwatch executionTime, char[,]
        map, List<Tuple<int, int>> treasures)
    {

```

```

{
    this.nodes = nodes;
    this.progress = progress;
    string routeFinal = "";
    for (int i = 0; i < route.Count; i++)
    {
        if (i != route.Count - 1)
        {
            routeFinal += (route.ElementAt(i) + " - ");
        }
        else
        {
            routeFinal += (route.ElementAt(i));
        }
    }
    this.route = routeFinal;
    this.steps = steps;
    this.executionTime = executionTime;
    this.solutionMap = map;
    this.treasures = treasures;
}

public int getNodes()
{
    return this.nodes;
}

public List<Tuple<int, int, string>> getProgress()
{
    return this.progress;
}

public string getRoute()
{
    return this.route;
}

public int getSteps()
{
    return this.steps;
}

public System.Diagnostics.Stopwatch getExecutionTime()
{
    return this.executionTime;
}

public char[,] getSolutionMap()
{
    return this.solutionMap;
}

public List<Tuple<int, int>> getTreasures()
{
    return this.treasures;
}
}

```

4.3 Penjelasan Tata Cara Penggunaan Program

Program dijalankan dengan masuk ke *directory* bin/Debug/net6.0-Windows/ dan menjalankan *file* Tubes2Stima.exe.

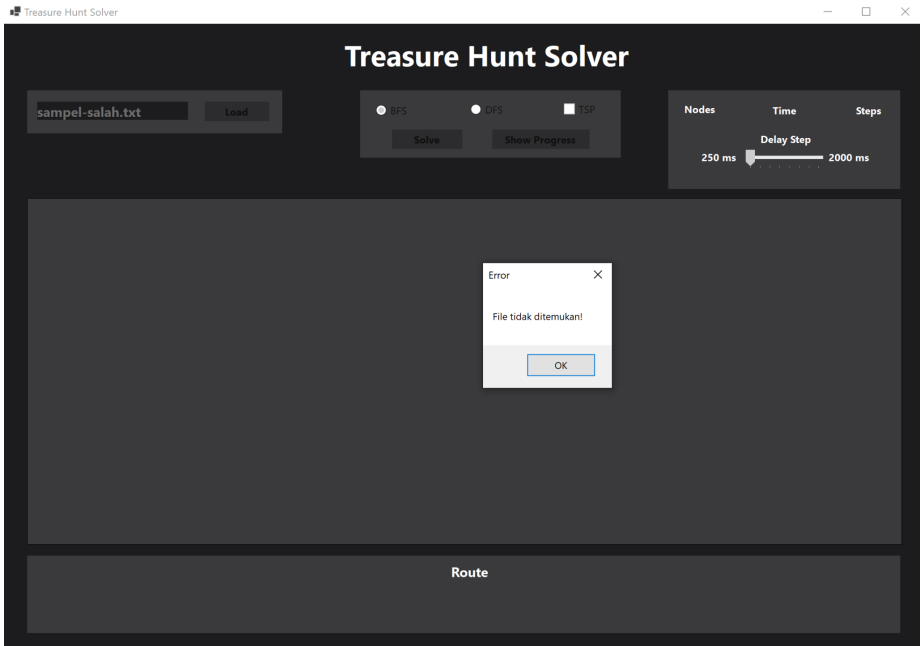
Untuk melakukan *loading* peta dari *file*, kita masukkan nama *file* yang telah diletakkan pada *directory* yang sama dengan *file* utama program. *Loading file* dilakukan dengan kotak yang ada di pojok kiri atas. Setelah nama yang benar dimasukkan, kita tekan tombol *load*. Jika *file* tidak ditemukan, maka akan muncul *pop-up* dengan pesan *error* bahwa *file* tidak ditemukan. Jika *directory* tidak ditemukan, maka akan muncul *pop-up* dengan pesan *error* bahwa *directory* tidak ditemukan. Jika *file* tidak sesuai dengan format yang semestinya, maka pesan *error* tersebut akan ditunjukkan. Jika *file* berhasil di-*load* dan dibaca, maka barulah akan ditampilkan peta di kotak besar yang ada di tengah program.

Setelah peta di-*load*, maka kita bisa melakukan pencarian dengan pengaturan yang ada di kotak tengah atas. Kita dapat memilih apakah pencarian DFS atau BFS serta apakah pencarian dilakukan dengan TSP (perlu kembali ke titik awal). Setelah *checkbox* dan *radio button* yang sesuai dengan keinginan kita di klik dan dipilih, kita dapat menekan tombol Solve atau Show Progress. Kedua tombol akan menampilkan jumlah *nodes* yang dikunjungi, waktu eksekusi, dan jumlah langkah *traversal* pada kotak di pojok kanan atas. Perbedaan kedua tombol tersebut adalah tombol Solve akan langsung menunjukkan rute *traversal* yang dikalkulasi program di kotak bagian tengah sementara Show Progress akan menunjukkan proses *traversal* yang dilakukan oleh program sesuai pengaturan pencarian yang dimasukkan, barulah di akhir akan ditunjukkan rute yang ditemukan. *Delay* antara langkah yang ditunjukkan ketika *progress traversal* sedang ditampilkan diatur oleh sebuah *slider* di dalam kotak pojok kanan atas yang memiliki opsi paling rendah $\frac{1}{4}$ detik dan paling tinggi 2 detik. Melalui *slider* tersebut, pengguna dapat mengatur seberapa cepat *progress traversal* ditunjukkan.

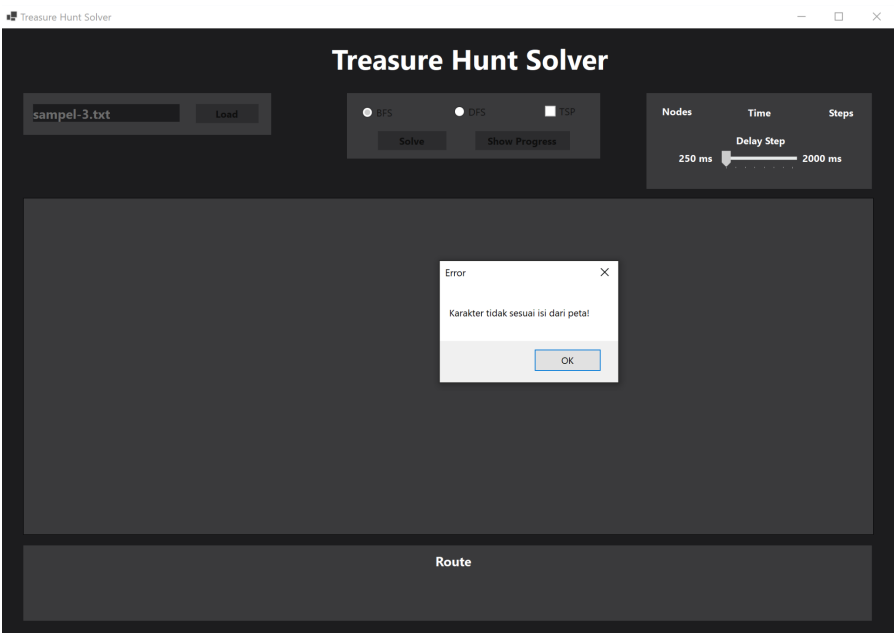
Ketika *file* sedang di-*load*, solusi sedang dikalkulasi, ataupun ketika *progress traversal* sedang ditampilkan, maka semua tombol yang ada dalam program – yaitu tombol yang ada di kotak bagian atas – akan di-*disable*.

4.4 Hasil Pengujian

4.4.1 Pengujian 1

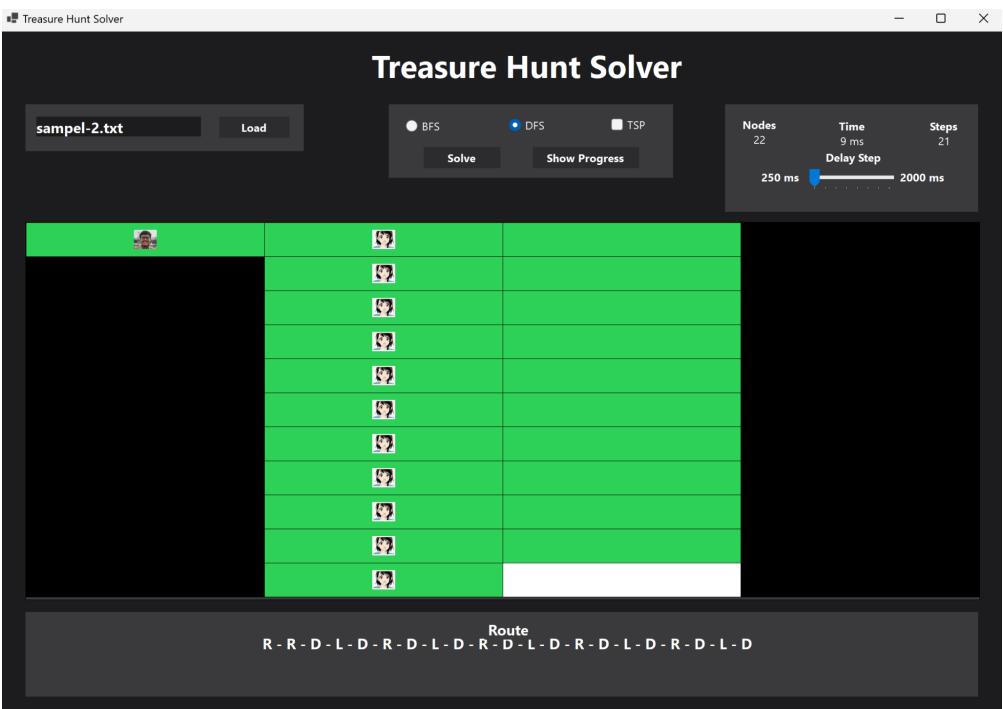
Peta	-
Nama file	sampel-salah.txt
Keterangan	Kasus nama file yang tidak valid
Hasil	

Peta	J A N G A N L U P A C E K Y A N G B E G I N I Y
Nama file	sampel-1.txt
Keterangan	Kasus isi peta memiliki karakter yang tidak valid

Hasil	
-------	--

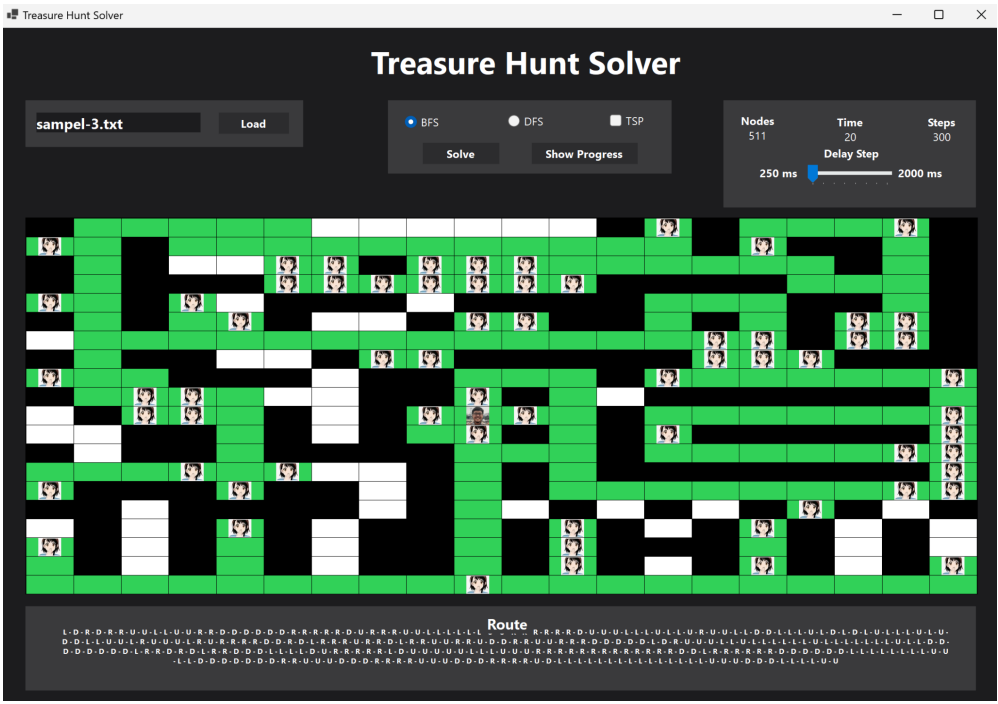
4.4.2 Pengujian 2

Peta	<pre> 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 X T R X X T R X </pre>
Nama file	sampel-2.txt
Keterangan	Kasus peta berbentuk asimetris

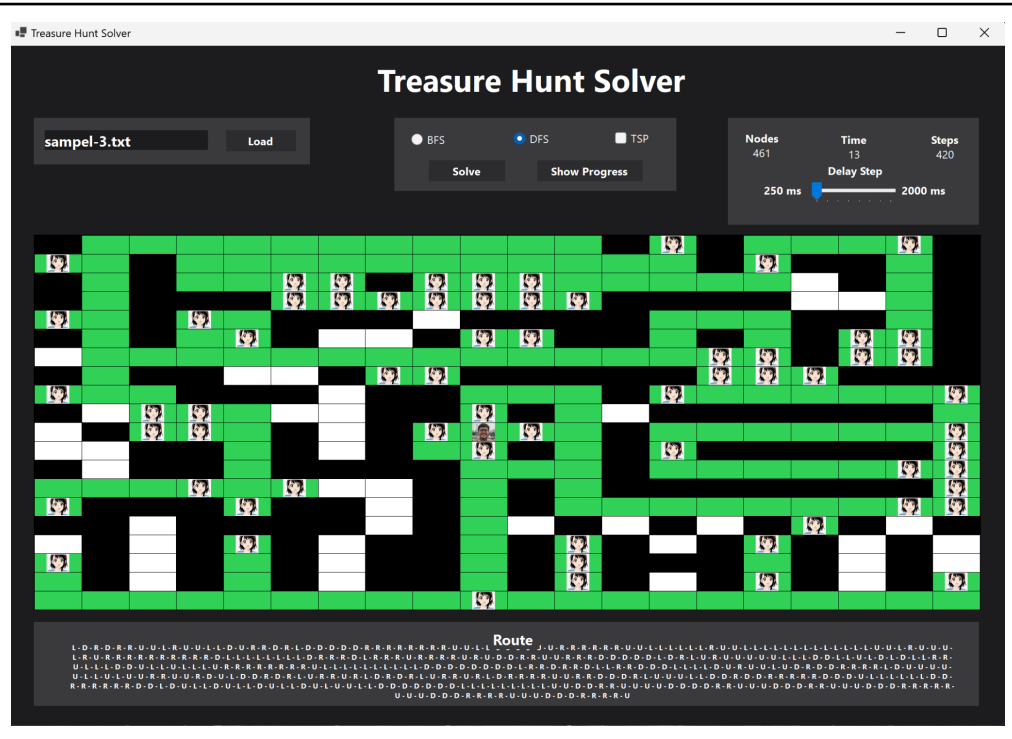
<p>Hasil (BFS)</p>	
<p>Hasil (DFS)</p>	

4.4.3 Pengujian 3

Peta	<pre> X R R R R R R R R R R R X T X R R R T X T R X R R R R R R R R R R R X T X X R X X R X R R T T X T T T R R R R R R X R X X R X X X T T T T T T X X X X R R R X T R X T R X X X R X X X R R R X X R X </pre>
------	--

	<pre> X R X R T X R R X T T X X R X R X T T X R R R R R R R R R R R R R R T T X T T X X R X X R R X T T X X X X T T T X X X T R R X X X R X X R R R X T R R R R R T X R T T R R R X X T X R R X X X X X X R R X T T R X R X T K T R X R R R R R R T R R X X R X R X R T X R X T X X X X X T X R X X R X X X X R R R X R R R R R T T R R R T R T R R X R X R X X X X X X T T X X X T X X R X R X R R R R R R R T T X X R X X X X R X R R X R X R X T X R X R X R X T X R X X R X T X R X T X R X R T X R X R X R X X R X T X X X R X R X R R X R X R X R X X R X T X R X T X R X T R R R R R R R R R T R R R R R R R R R R </pre>
Nama file	sampel-3.txt
Keterangan	Kasus peta berukuran sangat besar
Hasil (BFS)	 <p>The screenshot shows the 'Treasure Hunt Solver' application interface. It features a file input field with 'sampel-3.txt' and a 'Load' button. Below this are radio buttons for 'BFS' (selected), 'DFS', and 'TSP', along with 'Solve' and 'Show Progress' buttons. A statistics panel on the right displays 'Nodes: 511', 'Time: 20', and 'Steps: 300'. A 'Delay Step' slider is set to 250 ms. The main area shows a large maze map with a green path and small treasure icons. At the bottom, a 'Route' string is displayed, consisting of a long sequence of 'L', 'R', 'D', and 'U' characters.</p>

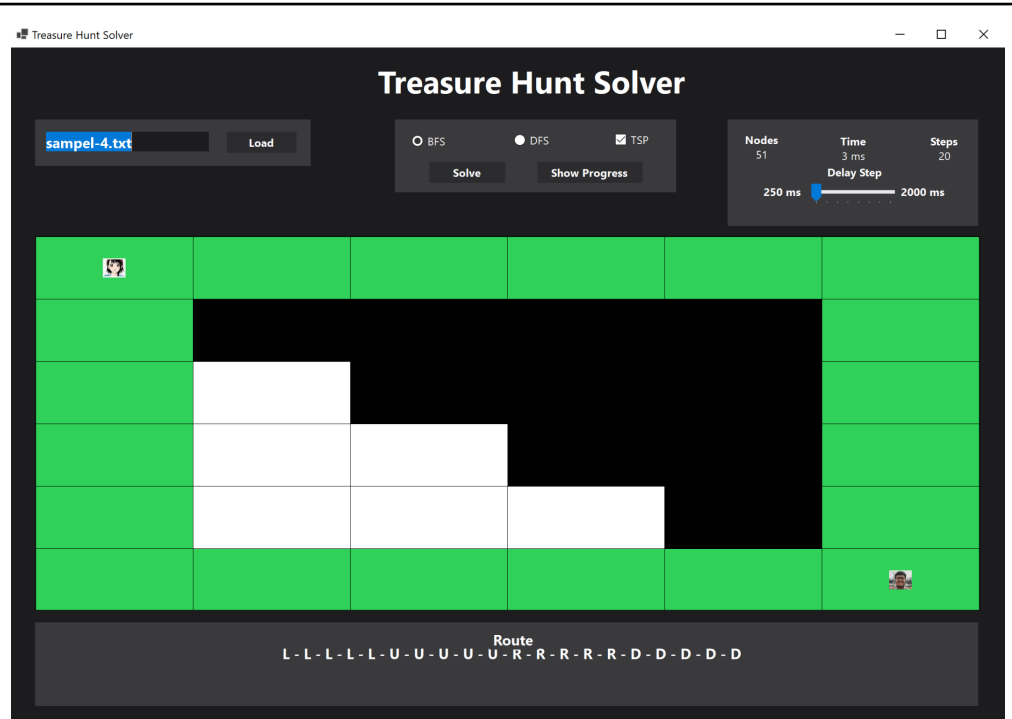
Hasil (DFS)



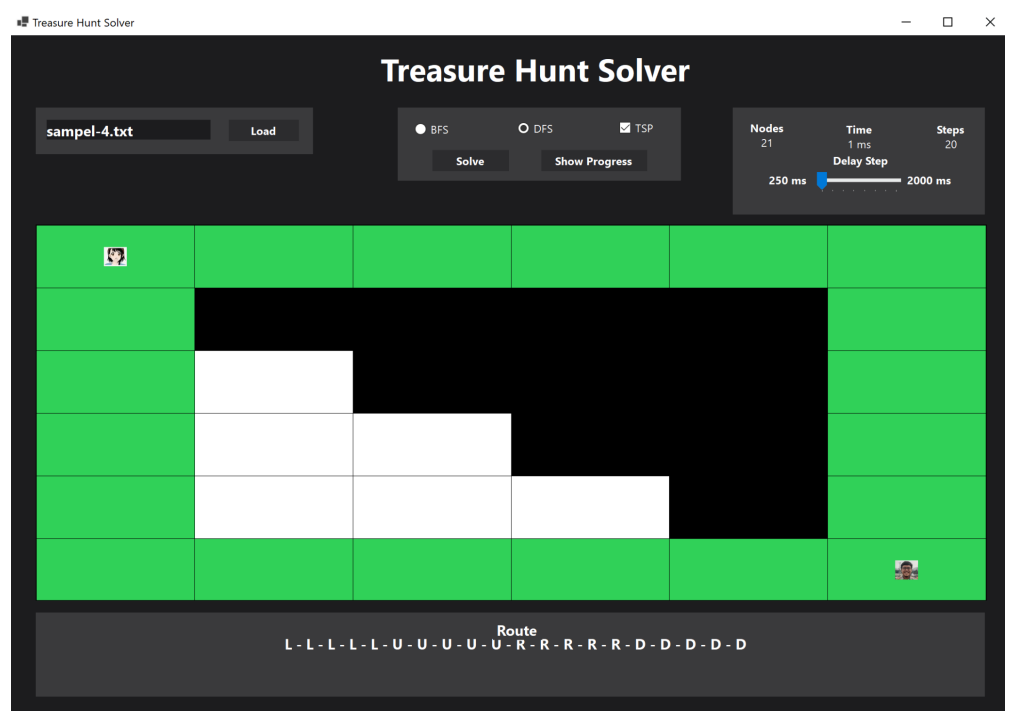
4.4.4 Pengujian 4

Peta	T R R R R 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 R K
Nama file	sampel-4.txt
Keterangan	Kasus <i>toggle</i> TSP dinyalakan

Hasil (BFS)



Hasil (DFS)



4.4.5 Pengujian 5

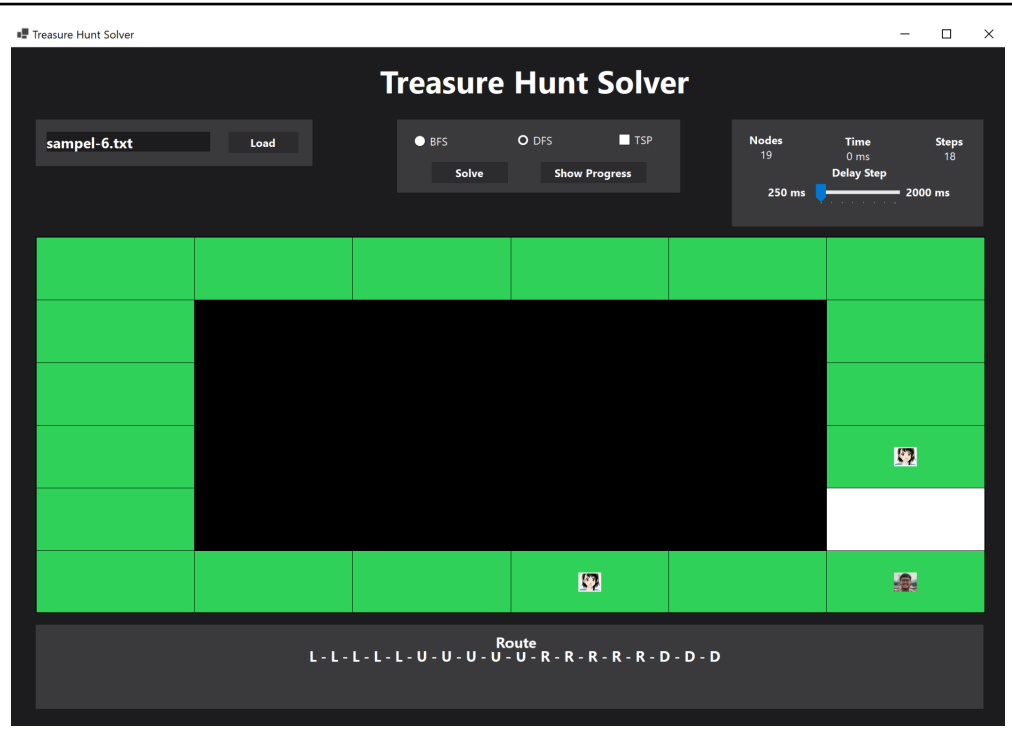
Peta	T R R R R R
	R X X X X R
	R R X X X R
	R R R X X R
	R R R R X R

	R R R R R K
Nama file	sampel-5.txt
Keterangan	Kasus BFS lebih baik daripada DFS
Hasil (BFS)	<div></div>
Hasil (DFS)	<div></div>

4.4.6 Pengujian 6

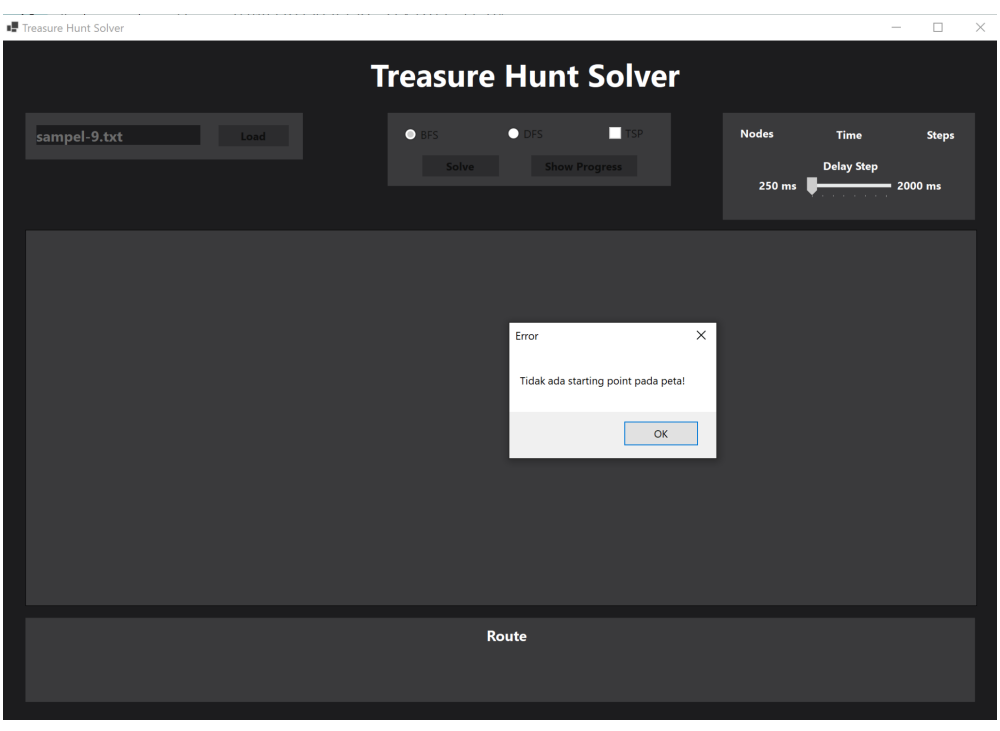
Peta	<pre> R R R R R R R X X X X R R X X X X R R X X X X T R X X X X R R R R T R K </pre>
Nama file	sampel-6.txt
Keterangan	-
Hasil (BFS)	<p>The screenshot shows the 'Treasure Hunt Solver' application interface. It includes a file input field with 'sampel-6.txt', a 'Load' button, and radio buttons for 'BFS' (selected), 'DFS', and 'TSP'. There are 'Solve' and 'Show Progress' buttons. On the right, statistics show 20 nodes, 0 ms time, and 18 steps. A 'Delay Step' slider is set to 250 ms. The main grid shows a 6x6 map with a black obstacle area. The path is highlighted in green. The route is: L-L-L-L-L-U-U-U-U-U-R-R-R-R-D-D-D.</p>

Hasil (DFS)



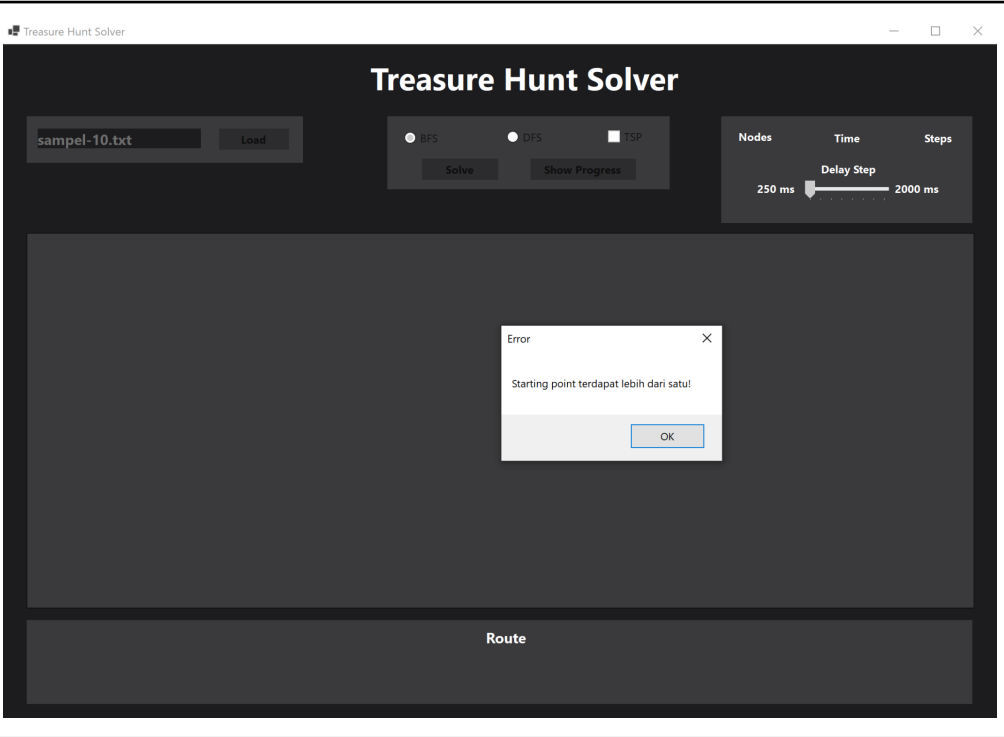
4.4.7 Pengujian 7

Peta	R X X R R X X R R R R T X R X R X R X T
Nama file	sampel-9.txt
Keterangan	Kasus tidak ada starting point pada peta

Hasil	
-------	--

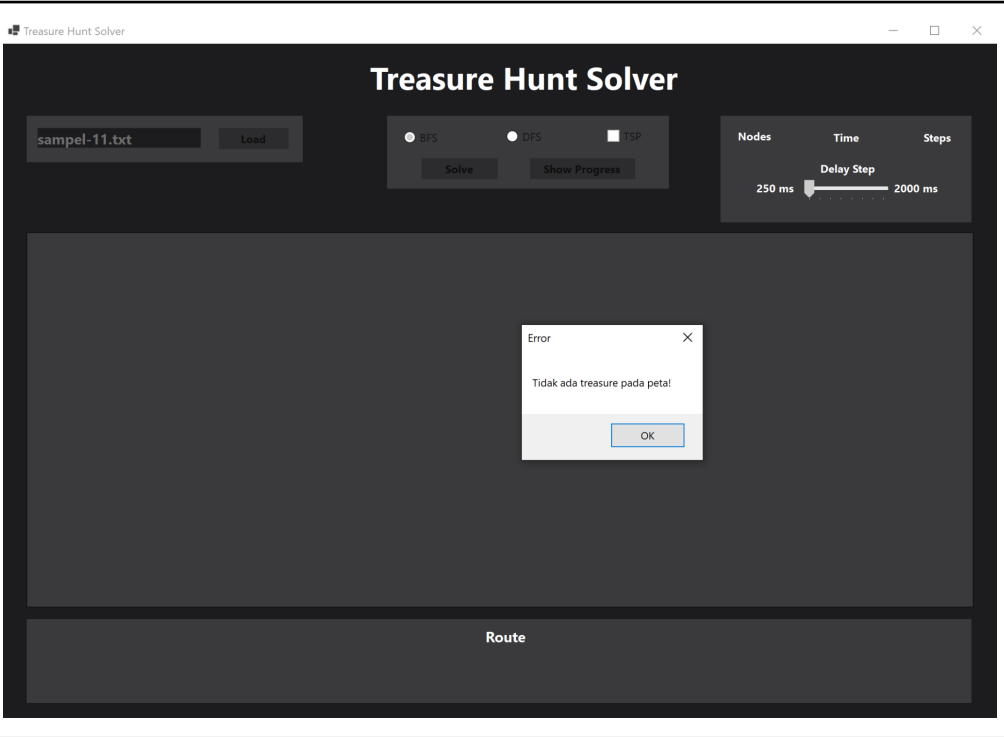
4.4.8 Pengujian 8

Peta	R X X R R X X R R R R T X R X R X R X T
Nama file	sampel-10.txt
Keterangan	Starting point terdapat lebih dari satu

Hasil	
-------	--

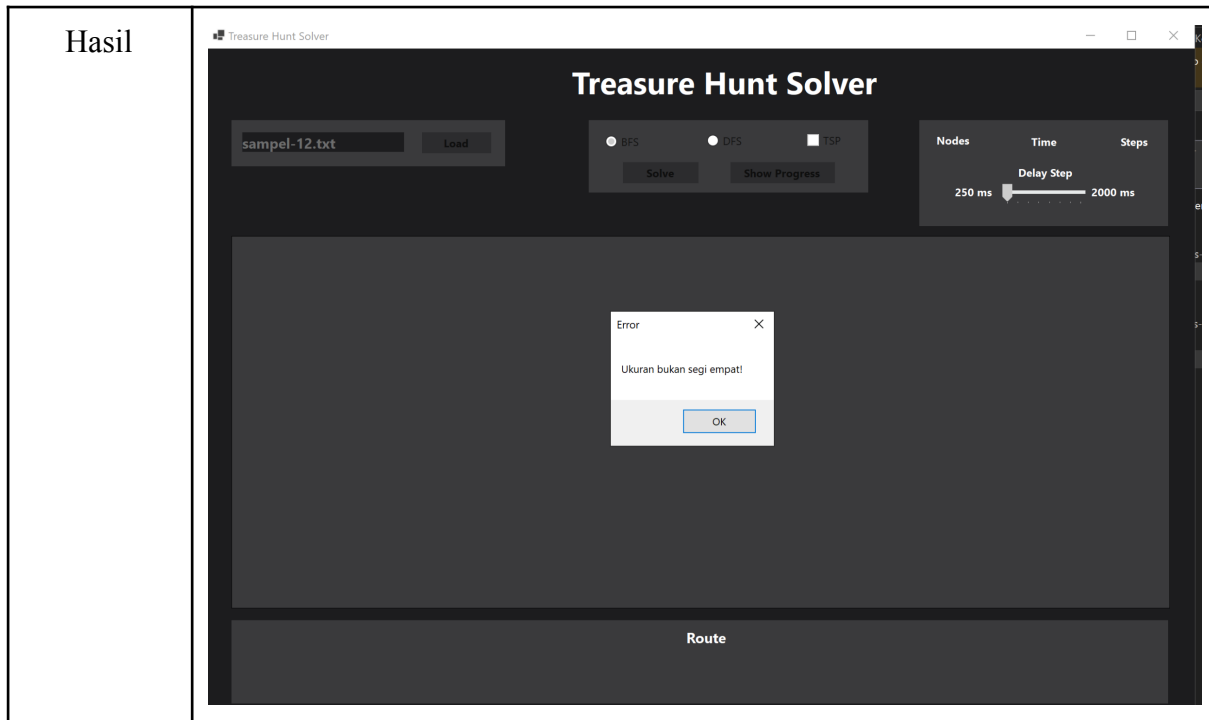
4.4.9 Pengujian 9

Peta	R X X R R X X R K R R R X R X R X R X R
Nama file	sampel-11.txt
Keterangan	Kasus tidak ada treasure pada peta

Hasil	 The screenshot shows the 'Treasure Hunt Solver' application window. At the top, there's a title bar and a main title. Below the title, there are input fields for a file name ('sampel-11.txt') and a 'Load' button. To the right, there are radio buttons for 'BFS' and 'TSP', and a 'Solve' button. Further right, there are tabs for 'Nodes', 'Time', and 'Steps', and a 'Delay Step' slider ranging from 250 ms to 2000 ms. A large dark gray area in the center is where the map would be displayed. At the bottom, there's a 'Route' section. An error dialog box is open in the center, with the text 'Tidak ada treasure pada petal' and an 'OK' button.
-------	---

4.4.10 Pengujian 10

Peta	X X X X X X K T X X
Nama file	sampel-12.txt
Keterangan	Kasus peta bukan segiempat



4.5 Analisis dari Desain Solusi Algoritma

4.5.1 BFS

Solusi rute yang ditawarkan oleh BFS bisa dipastikan merupakan rute dengan langkah terpendek apabila hanya terdapat satu treasure yang ada. Namun hal ini memiliki *cost* tersendiri, yaitu jumlah nodes yang dikunjungi akan jauh lebih banyak dibanding dengan memakai alternatif DFS. Contohnya pada pengujian 3 dan 4 dimana banyak nodes yang dikunjungi lebih banyak dari cara DFS.

Namun skema BFS tidak murni digunakan jika terdapat dua treasure atau lebih. Terdapat skema untuk *me-reset* BFS ketika sudah menemukan tiap treasure. Lalu prioritas dari BFS baru ini adalah node yang sebelumnya belum pernah dikunjungi. Hal ini menyebabkan terkadang rute yang diambil algoritma lebih panjang dari seharusnya. Seperti contoh pada pengujian 6, pada hasil BFS, karena treasure ditemukan pertama kali di bagian kiri bawah, maka untuk mencari treasure yang kedua, algoritma harus berputar jauh.

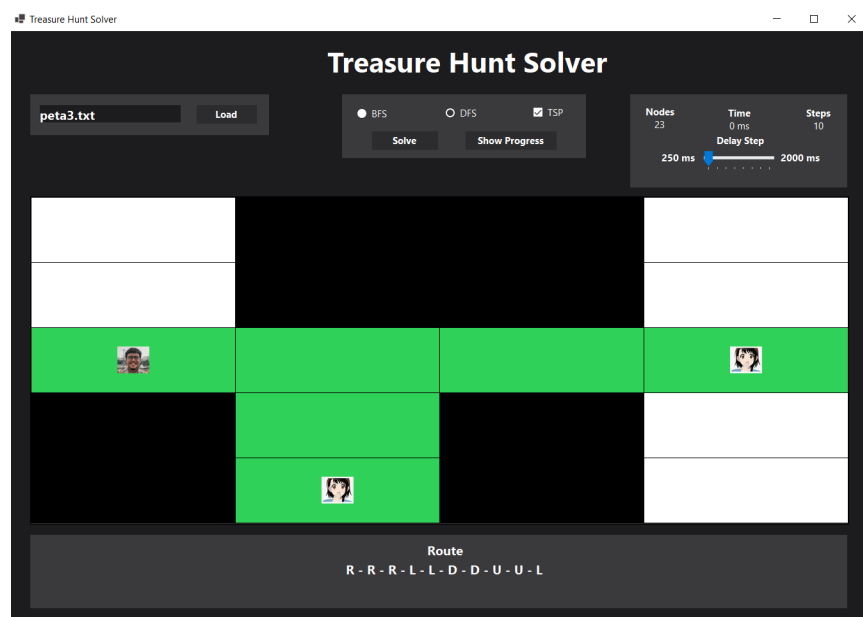
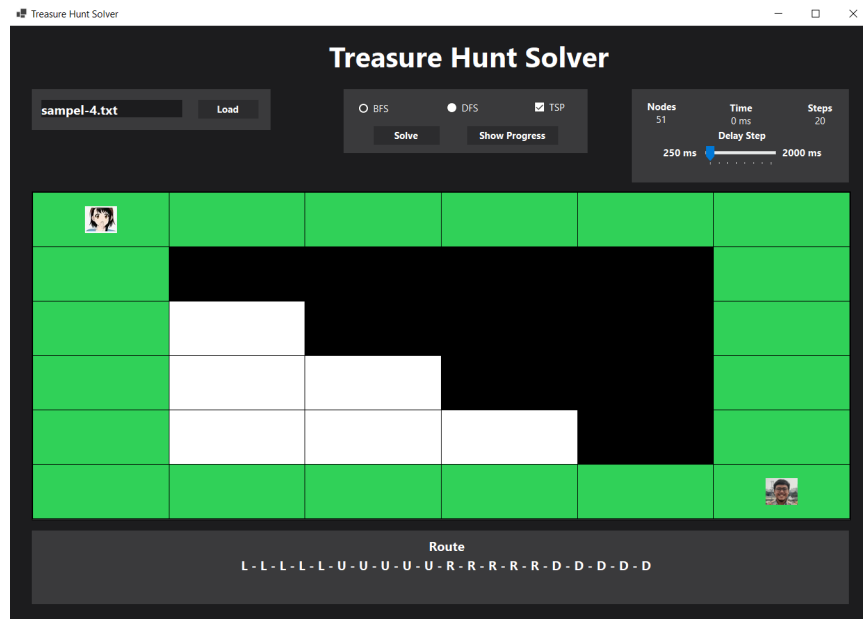
4.5.2 DFS

Solusi Rute yang ditawarkan DFS umumnya akan lebih panjang daripada yang diberikan BFS. Seperti yang diperlihatkan pada pengujian 5. Terlihat bahwa skema DFS bisa dengan mudah melewatkan treasure yang ada di dekat rute pencarian karena prioritas pencarian adalah LURD (Left, Up, Right, Down). Hal ini berbeda dengan BFS yang memeriksa area di dekat start secara menyeluruh, sehingga ditemukan treasure yang tersembunyi terlebih dahulu. Hal ini menyebabkan pada kebanyakan kasus, rute yang terbentuk dari pencarian DFS lebih panjang dari BFS.

Kesimpulannya, secara umum BFS akan lebih baik apabila terdapat lebih banyak persimpangan, apalagi jika *maze* memiliki lebar jalan yang lebih dari satu satuan. Hal ini terjadi karena jika terdapat banyak persimpangan, maka kesempatan DFS untuk melewatkan treasure akan lebih tinggi. Namun perlu diingat kembali nodes yang akan dihasilkan oleh BFS akan lebih banyak dari DFS.

4.6 Bonus TSP

Untuk salah satu bonus yang diterapkan pada algoritma kami yaitu persoalan TSP dimana setelah semua harta karun ditemukan, harus dicari rute kembali ke starting point. Untuk kasus BFS, persoalan TSP diatasi dengan melakukan BFS ulang dari petak harta karun terakhir yang ditemukan. BFS masih menggunakan struktur data yang menyimpan berapa banyak suatu petak telah dikunjungi, maka BFS akan tetap memprioritaskan petak-petak yang belum dikunjungi jika memungkinkan. Untuk kasus DFS, persoalan TSP diatasi dengan melakukan DFS ulang juga dari petak harta karun terakhir yang ditemukan. Berikut adalah contoh solusi dari suatu peta labirin dengan menyalakan toggle TSP pada aplikasi secara BFS maupun DFS.



Dapat dilihat pada rute solusi ditujukan untuk mendapatkan seluruh harta karun dan jalan balik menuju starting point kembali.

BAB V

KESIMPULAN, SARAN, DAN REFLEKSI

5.1 Kesimpulan

Pada Tugas Besar 2 IF2211 Strategi Algoritma ini telah diimplementasikan sebuah aplikasi C# untuk pencarian rute dalam maze treasure hunt menggunakan algoritma BFS dan DFS yang telah dipelajari selama pertengahan semester 4 di mata kuliah Strategi Algoritma dan tambahan materi yang kami eksplor secara mandiri, yaitu pembuatan GUI dalam C# menggunakan WinForm serta membuat animasi progress algoritma yang digunakan.

Program ini dibuat dalam bahasa pemrograman C# dan dibuat dalam 4 folder, yaitu src (berisi source code), doc (berisi laporan), bin (berisi aplikasi), dan test (berisi test case yaitu peta-peta maze).

Adapun metode-metode yang diterapkan pada aplikasi pencarian rute maze ini, seperti penggunaan struktur data priority queue untuk efisiensi BFS dan DFS yang teroptimisasi dengan penghentian pencarian ketika semua harta karun telah ditemukan.

5.2 Saran

Dari proses pengerjaan tugas besar ini, kami memiliki banyak kendala selama mengerjakan seperti kurangnya informasi dan ambiguitas nama persoalan dari spesifikasi yang telah dibuat. Oleh karena itu, kami memiliki beberapa saran agar tugas ini bisa menjadi lebih baik, yaitu

- a. Penamaan fitur yang tepat mengenai persoalan TSP mengingat simpul yang dikunjungi sebenarnya hanya boleh satu kali saja namun pada tugas besar ini boleh lebih dari satu kali.
- b. Constraint yang lebih jelas untuk pencarian rute dengan BFS mengingat simpul tujuan (treasure) dari persoalan ini bisa saja lebih dari satu.

5.3 Refleksi

Tugas Besar 2 IF2211 Strategi Algoritma ini merupakan salah satu tugas besar yang menantang bagi kami karena banyak sekali hal-hal yang harus dipelajari seperti penggunaan bahasa C# dan pembuatan aplikasi yang berbasis *Graphical User Interface*. Namun, dengan tantangan ini, kami mendapatkan kesempatan untuk melakukan eksplorasi lebih dalam sehingga tugas besar ini sangat mengembangkan skill yang kami miliki.

Tantangan yang dialami kami selain dari GUI adalah optimisasi dan eksplorasi terhadap algoritma yang terbaik untuk BFS dan DFS sehingga pencarian rute untuk lebih dari satu simpul tujuan dapat lebih efisien dan bisa menghasilkan rute terpendek jika memungkinkan. Tantangan ini dihadapi dengan mempelajari lebih lanjut struktur data yang dapat membantu serta kontribusi dari setiap anggota kelompok yang saling membantu dan mengerjakan tugasnya dengan baik.

5.4 Tanggapan

Tanggapan dari kelompok kami mengenai tugas ini adalah tugas ini merupakan salah satu tugas yang cukup seru untuk dibahas dan dapat merasakan kepuasan setelah menyelesaikan aplikasi dengan baik karena aplikasi dapat dibuat dengan cantik dan efisien. Masa-masa eksplorasi juga kadang membuat sakit kepala namun menjadi sebuah kepuasan bagi kami ketika suatu permasalahan dapat diperbaiki dan diatasi dengan baik.

DAFTAR PUSTAKA

- Munir, Rinaldi. (2023). IF2211 Strategi Algoritma - Semester II Tahun 2022/2023. Institut Teknologi Bandung. Diakses pada 7 Maret 2023, dari <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf>
- Munir, Rinaldi. (2023). IF2211 Strategi Algoritma - Semester II Tahun 2022/2023. Institut Teknologi Bandung. Diakses pada 7 Maret 2023, dari <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf>

LAMPIRAN

[Tautan repository Github](#)

[Tautan video penjelasan](#)