

LAPORAN TUGAS BESAR

IF2211 STRATEGI ALGORITMA

Pengaplikasian Algoritma BFS dan DFS dalam Menyelesaikan Persoalan *Maze Treasure Hunt*



Disusun oleh

Jeffrey Chow	13521046
Wilson Tansil	13521054
Jimly Firdaus	13521102

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2023

DAFTAR ISI

DAFTAR ISI.....	i
BAB I DESKRIPSI TUGAS	1
BAB II LANDASAN TEORI	4
2.1 <i>Graph Traversal</i> , BFS, dan DFS.....	4
2.2 C# Desktop Application Development	6
BAB III ANALISIS PEMECAHAN MASALAH	7
3.1 Langkah - Langkah Pemecahan Masalah.....	7
3.2 <i>Mapping</i> Persoalan Menjadi Elemen - Elemen Algoritma BFS dan DFS	7
3.3 Ilustrasi Kasus Lain	8
BAB IV IMPLEMENTASI DAN PENGUJIAN	9
4.1 Implementasi Program	9
4.2 Penjelasan Struktur Data yang Digunakan.....	11
4.3 Penjelasan Tata Cara Penggunaan Program dan Komponen Program	12
4.4 Hasil Pengujian	16
4.5 Analisis dari Design Solusi Algoritma BFS dan DFS terhadap Pengujian.....	17
BAB V KESIMPULAN DAN SARAN	18
5.1 Kesimpulan.....	18
5.2 Saran.....	18
5.3 Refleksi.....	18
5.4 Tanggapan Terkait Tugas Besar Ini	18
BAB VI.....	19
DAFTAR PUSTAKA.....	19
LAMPIRAN.....	20
Tautan <i>remote repository</i>	20
Tautan video.....	20

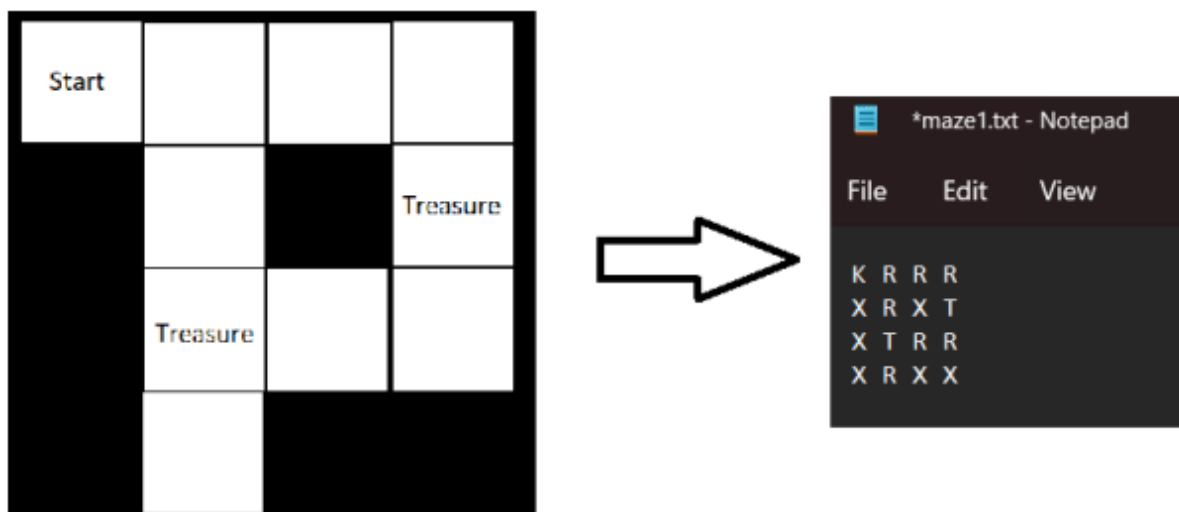
BAB I

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

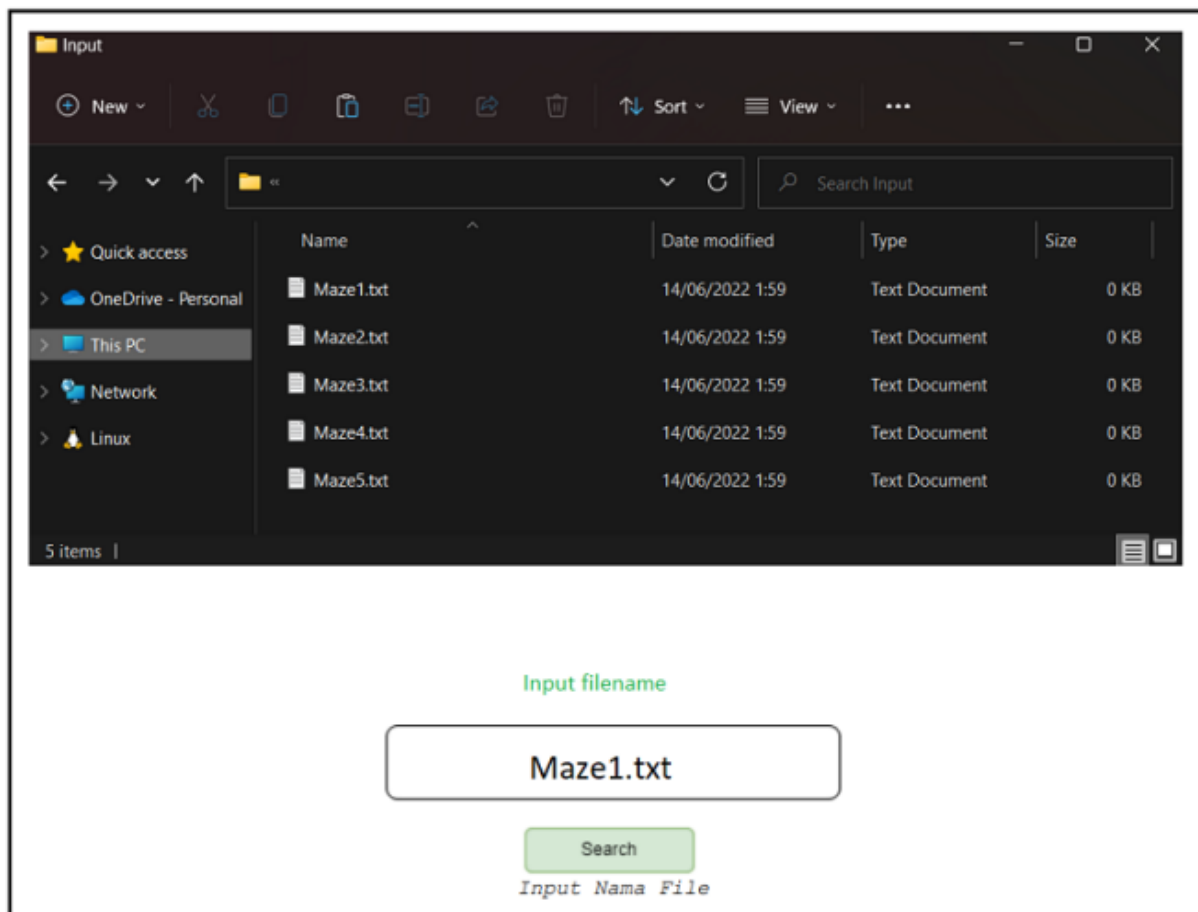
Contoh file input :



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.

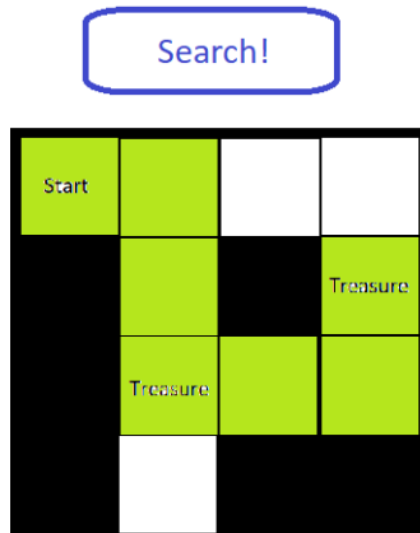
Contoh input aplikasi :



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 handle kasus apabila tidak ditemukan dengan nama file tersebut.

Contoh output Aplikasi :



Gambar 3. Contoh ouput 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 II

LANDASAN TEORI

2.1 *Graph Traversal*, BFS, dan DFS

Algoritma *graph traversal* atau pencarian graf adalah algoritma yang digunakan untuk menemukan solusi dari suatu persoalan dengan cara mengunjungi simpul-simpul pada graf secara sistematis. Terdapat dua jenis permasalahan dalam menggunakan algoritma *graph traversal*, yaitu pada permasalahan yang tidak memberikan informasi (*uninformed*, disebut dengan *blind search*), serta permasalahan yang memberikan informasi (*informed*). Dalam penyelesaiannya, pencarian solusi pada graf dapat menggunakan dua jenis pendekatan, yaitu dengan graf statis dan graf dinamis.

Dalam suatu graf, algoritma *graph traversal* bisa diimplementasikan dengan dua metode pencarian yang dikenal dengan pencarian *Breadth First Search* (BFS) serta pencarian *Depth First Search* (DFS). Algoritma pencarian BFS merupakan algoritma yang mengunjungi suatu simpul beserta simpul tetangganya yang belum dikunjungi sebelumnya terlebih dahulu sebelum mengunjungi simpul anaknya. Diperlukan 3 komponen dalam mengimplementasikan algoritma BFS, yaitu antrian (*queue*), simpul awal, dan simpul tujuan, berupa :

1. Graf Statis : graf yang telah terbentuk sebelum proses pencarian dilakukan
2. Graf Dinamis : graf yang terbentuk selama dilakukan pencarian

Mekanisme pencarian BFS secara umum adalah :

- Traversal dimulai dari simpul v .
- Algoritma :
 1. Kunjungi simpul v
 2. Kunjungi semua simpul yang bertetangga dengan simpul v terlebih dahulu
 3. Kunjungi simpul yang belum dikunjungi dan bertetangga dengan simpul-simpul yang tadi dikunjungi, demikian seterusnya

Berikut algoritma BFS secara umum.

```

Procedure BFS (input v: integer)
{Traversal graf dengan alforitma 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 a : 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 simpula 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)
            MasukAntrian(w) ← true
        endif
    endfor
endwhile
{AntrianKosong(q)}

```

Algoritma pencarian mendalam atau *Depth First Search* (DFS) adalah algoritma pencarian graf yang mengunjungi simpul-simpul pada graf secara vertikal terlebih dahulu sebelum mengunjungi simpul-simpul tetangganya. Dalam pencarian DFS, algoritma akan mengunjungi simpul anak dari simpul yang sedang dikunjungi sebelum mengunjungi simpul tetangganya. DFS biasanya digunakan untuk mencari jalur terpendek pada graf yang tidak memiliki bobot pada setiap sisi graf.

Berikut merupakan algoritma DFS secara umum.

```

Procedure DFS (input v: integer)
{Mengunjungi seluruh simpul graf dengan algoritma pencarian DFS.
Masukan : v adalah simpul awal kunjungan
Keluaran : semua simpul yang dikunjungi dicetak 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

```

Mekanisme pencarian DFS secara umum adalah :

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

2.2 C# Desktop Application Development

C# Desktop Application Development adalah pengembangan aplikasi desktop menggunakan bahasa pemrograman C#. C# adalah bahasa pemrograman yang dikembangkan oleh Microsoft dan merupakan bagian dari .NET language. C# memungkinkan pengembang untuk merancang aplikasi desktop berbasis Windows dengan mudah dan efisien. Dalam pengembangan aplikasi desktop, C# biasanya digunakan bersama dengan Windows Forms, sebuah framework yang memungkinkan pengembang untuk membuat aplikasi desktop dengan antarmuka grafis yang menarik dan mudah digunakan.

Selain itu, C# juga dapat digunakan untuk membuat aplikasi desktop dengan Windows Presentation Foundation (WPF) dan Universal Windows Platform (UWP). WPF adalah sebuah framework yang memungkinkan pengembang untuk membuat aplikasi desktop dengan antarmuka grafis yang menarik dan modern. Sedangkan UWP adalah sebuah platform pengembangan aplikasi yang memungkinkan pengembang untuk membuat aplikasi desktop yang dapat berjalan pada berbagai perangkat Windows, seperti PC, tablet, dan smartphone.

BAB III

ANALISIS PEMECAHAN MASALAH

3.1 Langkah - Langkah Pemecahan Masalah

Penyelesaian persoalan “*Maze Treasure Hunt*” diimplementasikan dalam file *SolveMaze.cs* dan beberapa file pendukung yaitu :

1. *Goblin.cs* : Untuk keperluan instansiasi objek *Goblin* sehingga dapat memanggil algoritma pencarian BFS dan DFS.
2. *GoblinForm.cs* : Untuk keperluan visualisasi dari hasil pencarian, baik dengan algoritma BFS maupun dengan DFS.

Pendekatan dalam penyelesaian persoalan “*Maze Treasure Hunt*” dengan algoritma pencarian BFS dan DFS.

a. Penyelesaian dengan BFS

Pada algoritma BFS, digunakan struktur data *Queue* untuk menyimpan *node – node* tetangga yang belum dikunjungi terhadap *node* sekarang. Awalnya akan dicatat *root node* atau *starting point* ke dalam *Queue*. Setiap *node* yang sudah dilalui akan disimpan sehingga untuk pengecekan selanjutnya tidak akan dilakukan pada *node* yang sudah pernah dikunjungi. Selama *loop* pencarian secara BFS belum berhenti (*Queue* masih belum kosong), maka akan dilakukan pengecekan pada anak atau tetangga dari *node* tersebut. Apabila masih tidak ditemukan semua *Treasure* dalam *maze* yang diberikan, pencarian akan terus dilanjutkan. *Node* selanjutnya yang diperiksa adalah *node* yang di-*dequeue* dari *Queue* BFS secara berurutan. Ketika ditemukan *treasure*, maka akan disimpan total *treasure* yang sudah ditemukan sejauh ini. Jika total *treasure* yang ditemukan sudah sama dengan jumlah *treasure* yang ada dalam *map*, maka *loop* BFS akan *terminat* dan mengembalikan *route* untuk mencapai semua *treasure* dari *start point*.

b. Penyelesaian dengan DFS

Pada algoritma DFS, digunakan struktur data *Stack* dan *Hash Map* untuk menyimpan *node* yang telah ditandai dan menandai *code* yang telah dilewati. Setiap *node* yang telah dilewati akan dimasukkan ke dalam *Stack* dan *node* akan dijadikan sebagai *index* untuk mengakses *Hash Map* untuk menandakan bahwa *node* ini telah dilalui. Ketika ditemukan *treasure* maka *route* akan disimpan di *result* dan akan dicari *route* selanjutnya lagi. *Loop* DFS akan mati apabila *Stack* telah kosong.

Keduanya memiliki prioritas arah berupa *left*, *up*, *right*, dan *down* yang masing-masing didefinisikan dengan L, U, R, dan D. Hasil dari kedua algoritma pencarian ini adalah *route* yang digunakan untuk mencapai semua *treasure* dalam *maze* dari *start point* yang diberikan pada *maze*. Diberikan juga total *nodes* serta *steps* yang dihasilkan pada proses penelusuran *treasure* dalam *maze*.

3.2 Mapping Persoalan Menjadi Elemen - Elemen Algoritma BFS dan DFS

Elemen – elemen dari algoritma pencarian BFS dan DFS untuk persoalan “*Maze Treasure Hunt*” direpresentasikan sebagai berikut.

Elemen – elemen	Algoritma BFS	Algoritma DFS
Pohon ruang status	Graf yang terbentuk dari simpul – simpul yang telah tercatat pada pohon ruang status	
Simpul	Akar : <i>start point</i> ('K') Daun : <i>treasure</i> dalam <i>maze</i>	
Cabang	<i>Enqueue</i> setiap <i>node</i> yang belum pernah di- <i>visit</i> dari <i>parent node</i> ke dalam <i>queue</i>	<i>Push</i> setiap <i>node</i> yang telah divisit ke dalam <i>Stack</i> .
Ruang status	Simpul – simpul yang telah tercatat dalam <i>queue</i>	Simpul-simpul yang terdapat dalam <i>Hash Map</i> .
Ruang solusi	Himpunan seluruh <i>route</i> menuju semua <i>treasure</i> dalam <i>maze</i>	

3.3 Ilustrasi Kasus Lain

Ilustrasi dengan Pendekatan Exhaustive Search

Untuk pendekatan dengan exhaustive search, dilakukan enumerasi untuk semua kemungkinan path yang dapat dibentuk dari posisi 'K' hingga mencapai semua 'T' dalam *maze*. Kemudian simpan path-path tersebut dan jika semua kemungkinan path sudah dienumerasi, pilih 1 path yang memenuhi aturan yang sebelumnya (dari posisi 'K' hingga mencapai semua 'T' dalam *maze*). Kemudian tampilkan *route*-nya.

BAB IV

IMPLEMENTASI DAN PENGUJIAN

4.1 Implementasi Program

Algoritma program ini diimplementasikan pada *file* GoblinForm.cs, Utility.cs, Goblin.cs, dan SolveMaze.cs. Setiap *file* memiliki perannya masing-masing dan file-file ini berjalan sebagai sebuah kesatuan yang tidak dapat dipisah. Adapun pseudocode bagaimana logika program berjalan sebagai berikut:

ALGORITMA MainProgram

```
if file is chosen then
    if file content is not valid then
        output "File not valid"
    else
        save filename
        show filename

if (BFS.Checked or DFS.Checked) and (filename is saved) then
    enable runButton

if runButton is clicked then
    disable all buttons
    start stopwatch
    if checkBFS.Checked and tspCheckbox.Checked then
        TSPwithBFS
    else if checkDFS.Checked and tspCheckbox.Checked then
        TSPwithDFS
    else if checkBFS.Checked then
        solveWithBFS
    else
        SolveWithDFS()
    stop stopwatch

    save route
    save steps
    show route
    show route path
    show nodes count
    show steps count
    show execution time
    enable all buttons

if resetButton is clicked then
    reset panel color

if showRouteButton is clicked then
    disable all buttons
    reset panel color
    show route
    enable all buttons

if showStepButton.Clicked then
    disable all buttons
    reset panel color
    show steps
    enable all buttons
```

<pre> FUNCTION TSPwithBFS outRoute <- solveWithBFS copy maze modify copy maze, current position become start point, start point become 'T' backRoute <- solveWithBFS for modified copy maze travelRoute <- concat(outRoute, backRoute) return travelRoute </pre>
<pre> FUNCTION TSPwithDFS solveWithDFS("TSP") </pre>
<pre> FUNCTION solveWithBFS (int totalTreasure) create queue bfsQ save (v) mark (v) as true enqueue(bfsQ, v) foundAll <- false encounteredTreasure <- 0 while(!foundAll) do if (bfsQ is not empty) then dequeue(bfsQ, v) for each node w that is adjacent to node v do if (not mark(w)) then saveRouteTo(w) mark(w) as true enqueue(bfsQ, w) endif if w == 'T' then encounteredTreasure <- encounteredTreasure + 1 endif if (encounteredTreasure == totalTreasure) foundAll <- true break endif endfor else find an active node x find a route from the current position to x saveRouteTo(x) mark(x) as true enqueue(bfsQ, x) endif endwhile return savedRoute </pre>
<pre> FUNCTION solveWithDFS(string choice = " ") ResetGoblin() DECLARE hmp as Hashmap SET hmp to all states with key value coordinates and all values set to false DECLARE route as the result of DFS(initialStartingPoint, choice, hmp) IF choice == TSP THEN SET route to CONCAT(DFS(lasttreasureStartingPoint, choice), route, hmp) END IF RETURN route END FUNCTION </pre>
<pre> FUNCTION DFS(int startingPoint, string choice, Hashmap hmp) SET current point as starting point DECLARE a Stack </pre>

```

PUSH current point to Stack
DECLARE a LastMove Stack
COPY construct hmp to new Hashmap hmp_copy
SET hmp_copy with key current point to true
DECLARE rute as the result
DECLARE temporary rute as dummy
SET FoundHome to False

WHILE (Stack is not empty)
  IF (choice == "TSP" && no where to move)
    RESET hmp_copy
  ELSE
    SET FoundHome to True

  IF(FoundHome)
    SET all directions as backtrack
  ELSE
    GET available directions

  IF (direction != backtrack)
    SET all coordinates according to directions
    PUSH current coordinates to Stack
    PUSH opposite directions to LastMove Stack
    ADD coordinate to temporary rute
  ELSE IF direction == backtrack
    POP the Stack value
    ADD the popped value of LastMove Stack to temporary rute
    SET current position as the top element of the Stack

  IF(find target and hmp_copy[current position])
    SET rute to CONCAT(rute and temporary rute)
    CLEAR temporary rute
    SYNCH hmp with hmp_copy

  SET hmp_copy[current position] to true

RETURN rute

```

4.2 Penjelasan Struktur Data yang Digunakan

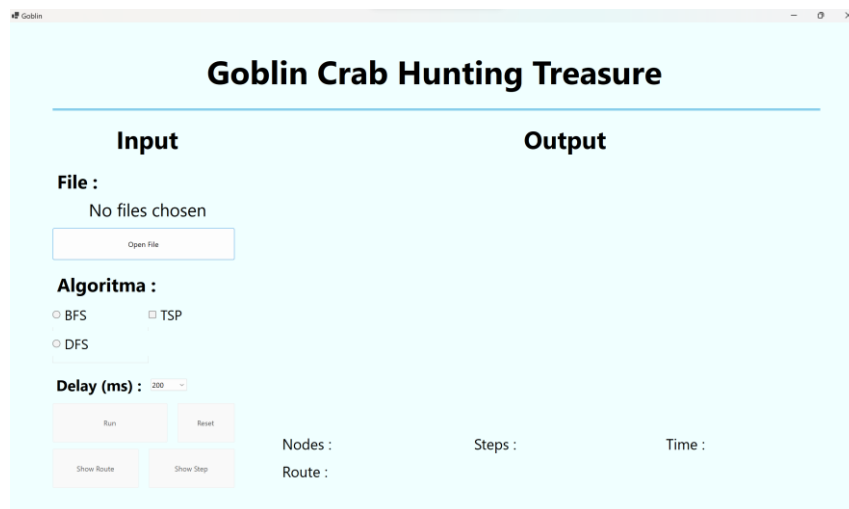
Pada implementasi program, kami menggunakan beberapa struktur data untuk membantu keberjalanan program, yaitu sebagai berikut :

1. Panel : digunakan untuk kepentingan GUI
2. Point : digunakan untuk menggambarkan posisi pencarian dalam koordinat kartesian
3. List of char : digunakan untuk menyimpan seluruh *direction* yang pernah digunakan
4. Label : digunakan untuk kepentingan GUI
5. Color : digunakan untuk kepentingan GUI
6. Button : digunakan untuk kepentingan GUI
7. Goblin : digunakan untuk menginstansiasi objek Goblin untuk menyelusuri *maze*

8. List of point : digunakan untuk menampung seluruh *history movement* dari Goblin
9. BFSNode : kelas yang merepresentasikan *node – node* pada algoritma pencarian BFS
10. Queue of BFSNode : digunakan sebagai *Queue* BFS
11. Set list of points : digunakan untuk menyimpan *node-node* apa saja yang sudah pernah dikunjungi
12. Dictionary : digunakan untuk menandakan bahwa node tersebut telah dilewati atau belum
13. Stack of char : menyimpan node-node yang telah dilewati untuk melakukan backtrack.

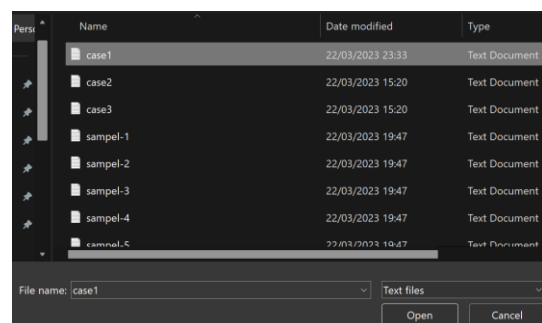
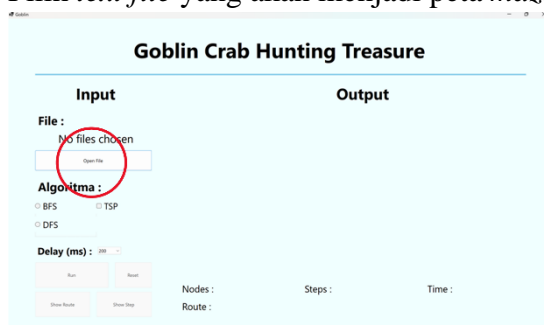
4.3 Penjelasan Tata Cara Penggunaan Program dan Komponen Program

Tampilan awal program adalah sebagai berikut :

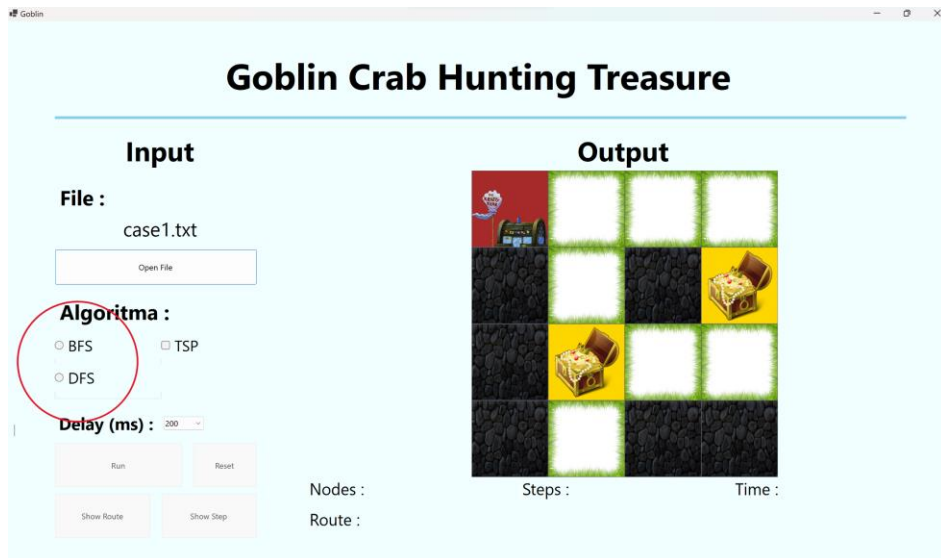


Tata cara penggunaan perangkat lunak adalah sebagai berikut:

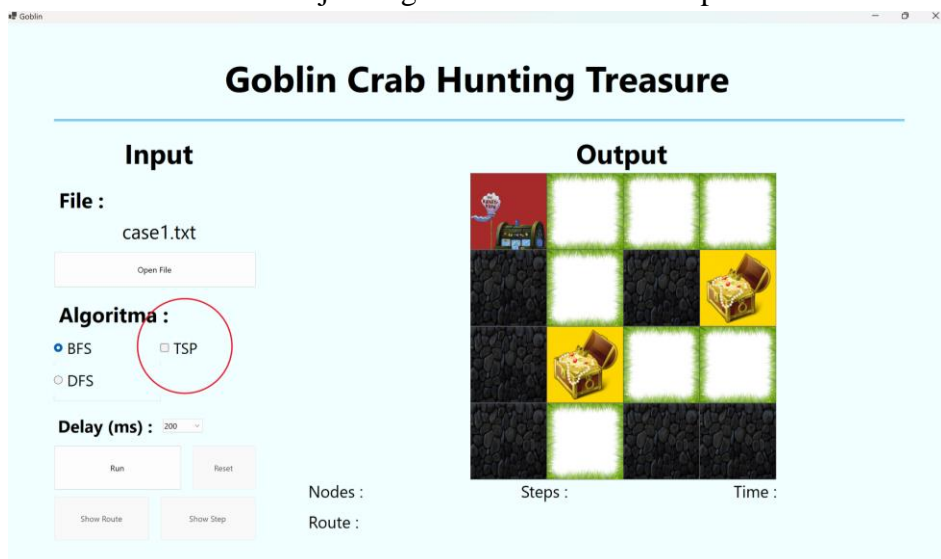
1. Pilih *text file* yang akan menjadi peta *maze*



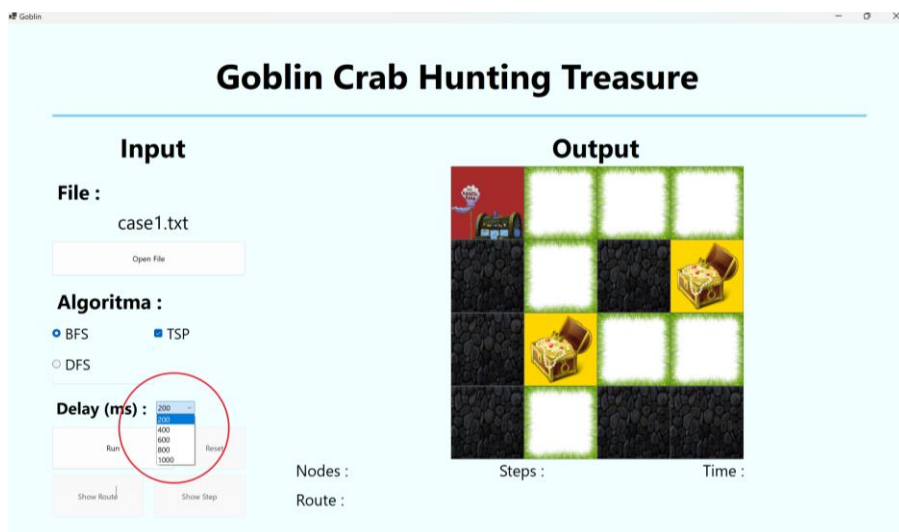
2. Pilih algoritma yang akan digunakan untuk mencari solusi (BFS/DFS)



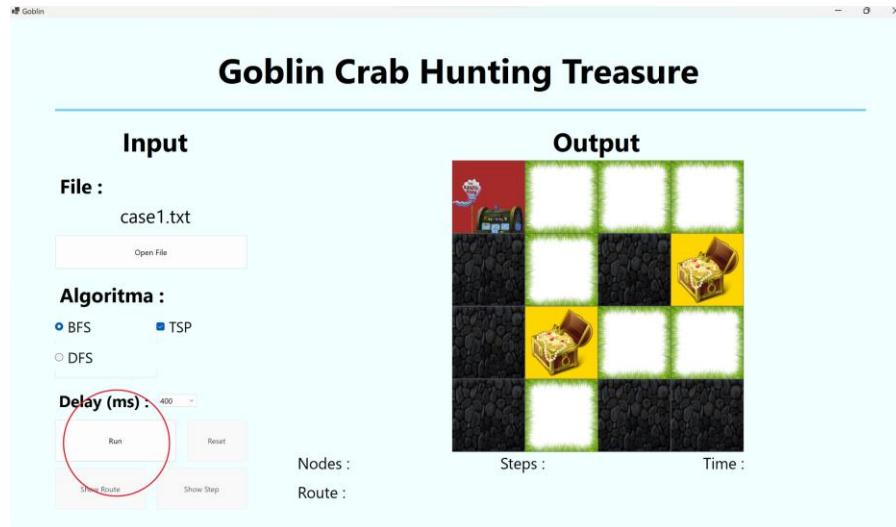
3. Aktifkan *checkbox* TSP jika ingin mencari solusi berupa TSP



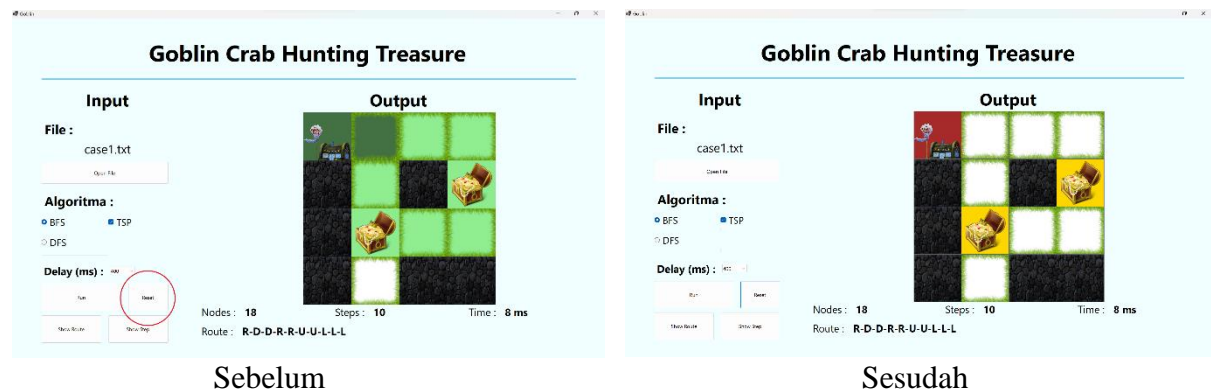
4. Atur *delay time* animasi program, secara default adalah 200 ms



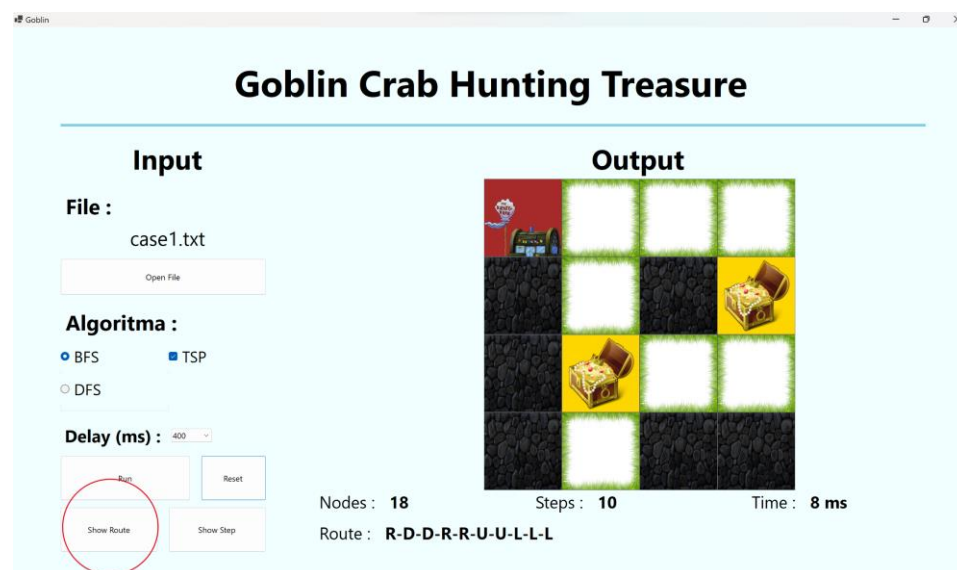
5. Tekan tombol “Run” untuk mencari solusi



6. Tekan tombol “Reset” untuk mengembalikan warna menjadi *default*

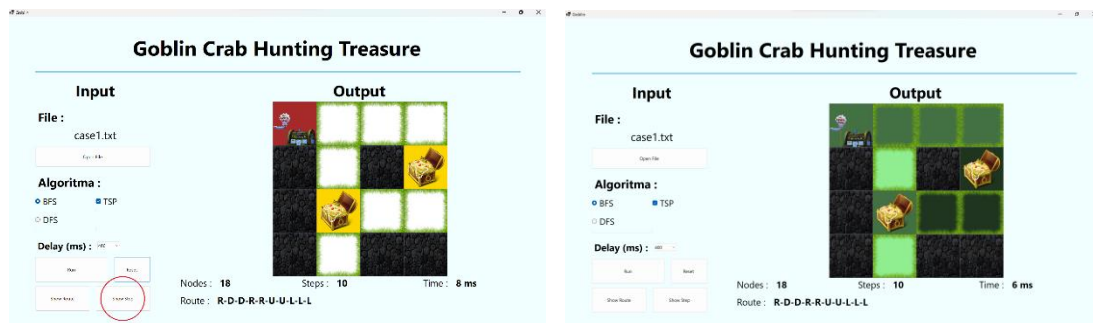


7. Tekan tombol “Show Route” untuk menunjukkan kembali rute pencarian solusi



Hasil akan sama seperti hasil di poin 6




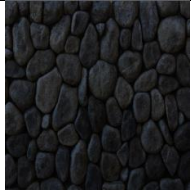
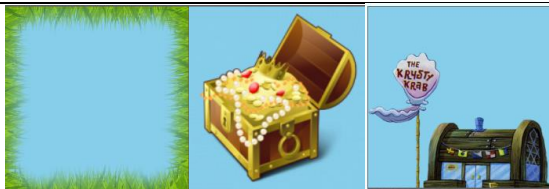
8. Tekan tombol “Show Step” untuk melakukan visualisasi langkah pencarian dengan algoritma terkait.


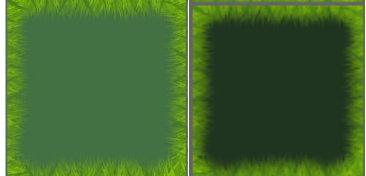


Sebelum


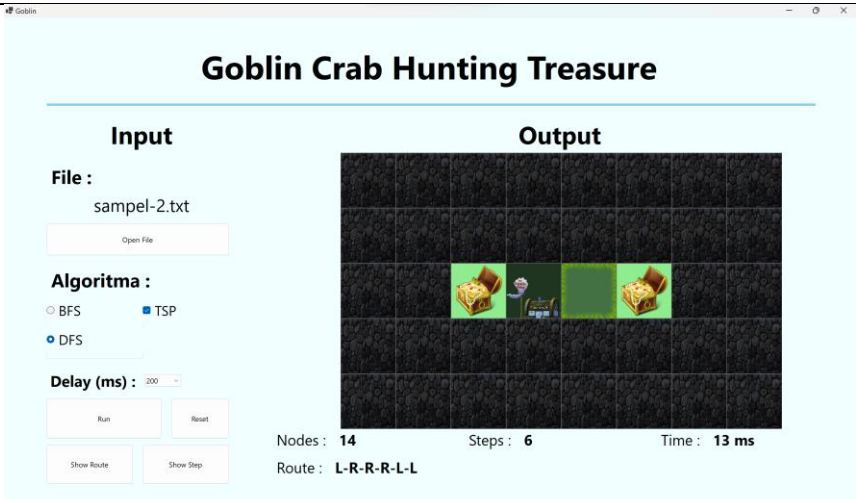
Sesudah

Komponen pada *output* program adalah sebagai berikut :

Komponen	Penjelasan
	Karakter ‘K’ pada <i>text file</i> akan divisualisasi pada GUI dengan gambar Krusty Krab dan <i>background color</i> merah.
	Karakter ‘R’ pada <i>text file</i> akan divisualisasi pada GUI dengan border rumput dan <i>background color</i> putih.
	Karakter ‘T’ pada <i>text file</i> akan divisualisasi pada GUI dengan gambar harta karun dan <i>background color</i> emas.
	Karakter ‘X’ pada <i>text file</i> akan divisualisasi pada GUI dengan gambar batu berwarna kehitaman.
	Saat program melakukan visualisasi terhadap jalur solusi, warna biru muda akan menjadi <i>background color</i> komponen yang sedang dilewati.

	<p>Saat program melakukan visualisasi terhadap jalur solusi, warna hijau muda akan menjadi <i>background color</i> komponen yang telah dilewati.</p>
	<p>Saat program melakukan visualisasi terhadap jalur solusi, <i>background color</i> komponen yang sudah dilewati berkali-kali akan menjadi semakin hijau/gelap.</p>

4.4 Hasil Pengujian

Pengujian	Hasil
<p>sampel-1.txt</p> <pre> X T X X X R R T K R X T X R X R X R R R </pre> <p>Algoritma : TSP dengan BFS</p>	
<p>sampel-2.txt</p> <pre> X X X X X X X X X X X X X X X X X X T K R T X X X X X X X X X X X X X X X X X X </pre> <p>Algoritma : TSP dengan DFS</p>	

<p>sampel-3.txt</p> <pre> J A N G A N L U P A C E K Y A N G B E G I N I Y </pre> <p>Text file mengandung karakter selain 'K', 'T', 'R', 'X'</p>	
<p>sampel-4.txt</p> <pre> K X X X X X R R X X X X R R R X X X R R R R X X R R R R R X R R R R R T </pre> <p>Algoritma : BFS</p>	
<p>sampel-5.txt</p> <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 </pre> <p>Algoritma : DFS</p>	

4.5 Analisis dari Design Solusi Algoritma BFS dan DFS terhadap Pengujian

Design Solusi Algoritma *Breadth First Search* dan *Depth First Search* memiliki keunggulan masing-masing dalam menyelesaikan sebuah permasalahan. Misalnya untuk BFS sendiri memiliki keunggulan apabila *treasure* yang dicari berdekatan dengan titik mulai dan DFS memiliki keunggulan apabila *treasure* terletak jauh di peta. Keduanya juga dipengaruhi *priority scale* dimana bisa saja apabila direction yang diambil benar maka akan mendapatkan hasil yang lebih efisien.

BAB V

KESIMPULAN DAN SARAN

5.1 Kesimpulan

Untuk Tugas Besar ini, kami telah berhasil menyelesaikan persoalan “*Maze Treasure Hunt*” dengan menggunakan algoritma pencarian *Breadth First Search* (BFS) dan algoritma pencarian *Depth First Search* (DFS). Karena pada persoalan ini, posisi *start* dan semua posisi *treasures* sangat berpengaruh kepada efisiensi dari hasil algoritma pencarian BFS maupun DFS, maka efisiensi kedua algoritma tidak dapat ditebak secara pasti. Generalisasinya adalah ketika *start point* tidak berada jauh dengan semua *treasure* yang ada, maka algoritma pencarian BFS akan lebih optimal dibandingkan dengan DFS. Namun jika posisi *start point* dengan semua *treasure* relatif jauh, maka algoritma pencarian DFS akan lebih optimal jika dibandingkan dengan algoritma pencarian BFS.

5.2 Saran

Saran untuk Tugas Besar ini adalah :

1. Pengembangan UI dapat lebih dimaksimalkan sehingga lebih interaktif.
2. Kode program lebih modular.
3. Meningkatkan efisiensi baik dari segi *space-complexity*.

5.3 Refleksi

Dengan selesainya Tugas Besar ini, kami mendapatkan *insight-insight* baru seperti beragam alternatif untuk *edge case* dari algoritma pencarian BFS yaitu saat pencarian sudah “*stuck*” yang artinya sudah tidak ada simpul yang bisa dicapai dari *node* terakhir BFS (*Queue* kosong). *Insight* lain yang kami dapatkan yaitu kami bisa mengenal dengan bahasa C# dan manfaatnya secara langsung dalam pembuatan GUI untuk keperluan Tugas Besar ini.

5.4 Tanggapan Terkait Tugas Besar Ini

Adanya persoalan – persoalan pada konfigurasi *maze* yang dapat menghasilkan algoritma BFS menjadi seakan-akan seperti dengan DFS yaitu saat algoritma pencarian BFS sudah mencapai *node* terakhir pada *Queue* BFS sehingga penyelesaian sisa pencarian dengan “*teleport*” ke *node* aktif yang lain akan memunculkan kebingungan dalam logika karena kemampuan *teleport* tersebut dalam proses menelusuri maupun dengan penyelesaian sisa pencarian dengan menelusuri kembali jalan yang sama untuk keluar dari keadaan “*stuck*” sehingga adanya *backtracking* pada algoritma pencarian BFS yang hal tersebut merupakan ciri khas dari algoritma DFS bukan BFS.

BAB VI

DAFTAR PUSTAKA

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

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

LAMPIRAN

Tautan *remote repository*

Berikut adalah tautan *remote repository* yang berisi *source code* untuk tugas ini.

https://github.com/Jimly-Firdaus/Tubes2_Goblin

Tautan video

<https://youtu.be/PuHY-YyCRas>