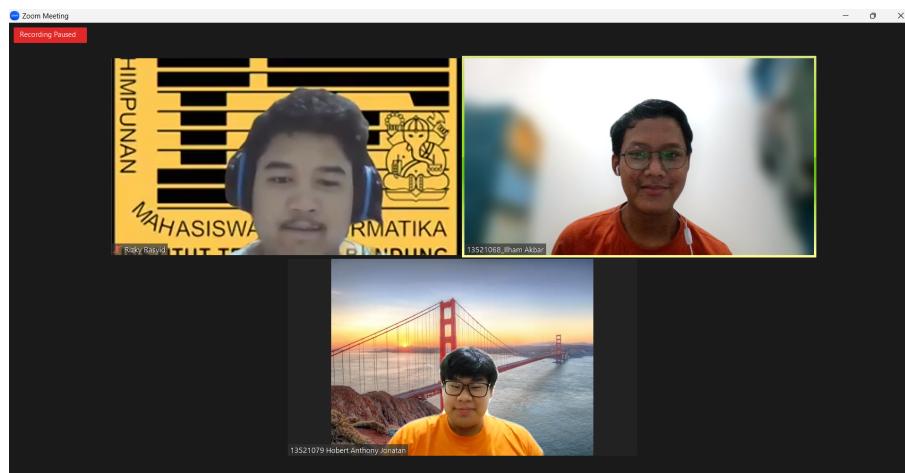


Tugas Besar 2 IF2211 Strategi Algoritma

Pemanfaatan Algoritma BFS dan DFS untuk Mencari Harta

Tuan Krab



Disusun oleh :

Kelompok CHastag

Ilham Akbar **13521068**

Hobert Anthony Jonatan **13521079**

Rizky Abdillah Rasyid **13521109**

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

2023

DAFTAR ISI

Bab 1 Deskripsi Tugas	2
Bab 2 Landasan Teori	8
2.1 Graf Traversal	8
2.2 Breadth-First Search	8
2.3 Depth-First Search	10
2.4 C# Desktop Application Development	11
Bab 3 Aplikasi Algoritma BFS dan DFS	13
3.1 Langkah-langkah Pemecahan Masalah	13
3.2 Elemen-elemen Algoritma BFS dan DFS	14
3.2.1. Breadth-first search	14
3.2.2. Depth-first search	14
3.3 Ilustrasi dengan Kasus Lain	14
Bab 4 Implementasi dan Analisis Pemecahan Masalah	17
4.1 Implementasi Program	17
4.1.1 huntBFS	17
4.1.2 runDFS	18
4.2 Struktur Data dalam Program	20
4.2.2 Graph	20
4.2.2.1 Node	20
4.2.3 Queue	21
4.2.4 Stack	21
4.2.5 List	21
4.3 Tata Cara Penggunaan Program	22
4.4 Analisis dan Hasil Pengujian	22
4.4.1 Tampilan Antarmuka Program	22
4.4.2 Pengujian program dengan test case Sampel	25
Bab 5 Kesimpulan dan Saran	37
5.1 Kesimpulan	37
5.2 Saran	37
5.3 Refleksi	37
5.4 Tanggapan Terkait Tugas Besar	38
DAFTAR PUSTAKA	39

Bab 1

Deskripsi Tugas

Tuan Krabs menemukan sebuah labirin distorsi terletak tepat di bawah Krusty Krab bernama El Doremi yang Ia yakini mempunyai sejumlah harta karun di dalamnya dan tentu saja Ia ingin mengambil harta karunnya. Dikarenakan labirinnya dapat mengalami distorsi, Tuan Krabs harus terus mengukur ukuran dari labirin tersebut. Oleh karena itu, Tuan Krabs banyak menghabiskan tenaga untuk melakukan hal tersebut sehingga Ia perlu memikirkan bagaimana caranya agar Ia dapat menelusuri labirin ini lalu memperoleh seluruh harta karun dengan mudah.



Gambar 1.1 Labirin di Bawah Krusty Krab

(Sumber: https://static.wikia.nocookie.net/theloudhouse/images/e/ec/Massive_Mustard_Pocket.png/revision/latest?cb=20180826170029)

Setelah berpikir cukup lama, Tuan Krabs tiba-tiba mengingat bahwa ketika Ia berada pada kelas Strategi Algoritma-nya dulu, Ia ingat bahwa Ia dulu mempelajari algoritma BFS dan DFS sehingga Tuan Krabs menjadi yakin bahwa persoalan ini dapat diselesaikan menggunakan kedua algoritma tersebut. Akan tetapi, dikarenakan sudah lama tidak menyentuh algoritma, Tuan Krabs telah lupa bagaimana cara untuk menyelesaikan persoalan ini dan Tuan Krabs pun kebingungan. Tidak butuh waktu lama, Ia terpikirkan sebuah solusi yang brilian. Solusi tersebut adalah meminta mahasiswa yang saat ini sedang berada pada kelas Strategi Algoritma untuk menyelesaikan permasalahan ini.

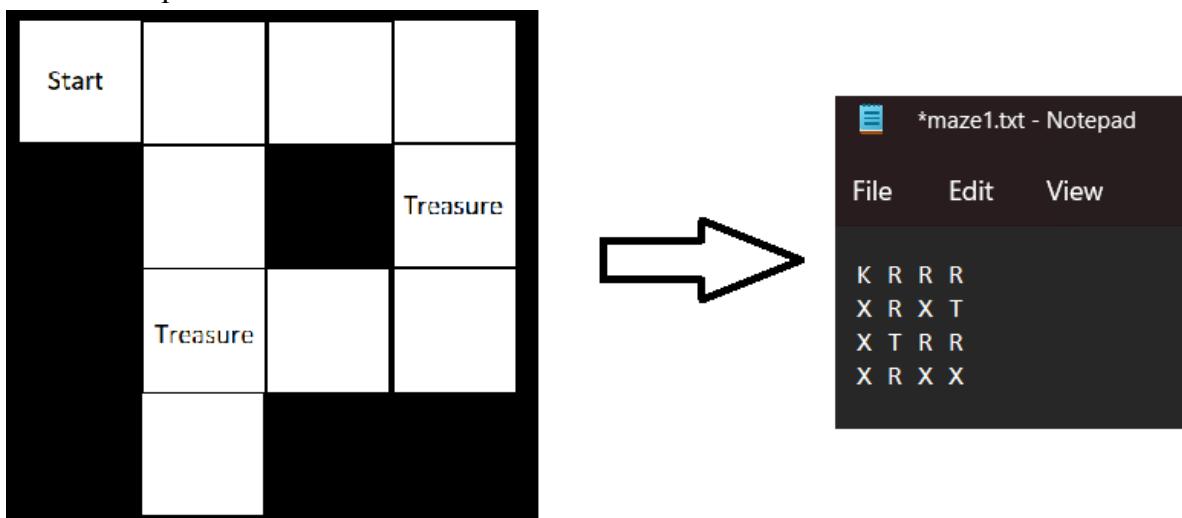
Deskripsi tugas:

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 segiempat 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

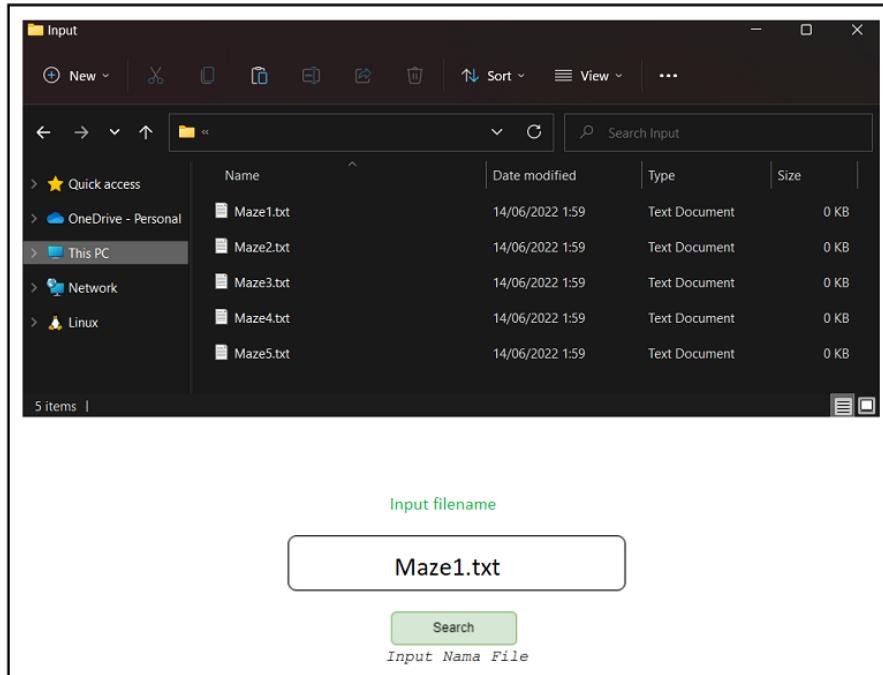
Contoh file input :



Gambar 1.2 Ilustrasi input file maze

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

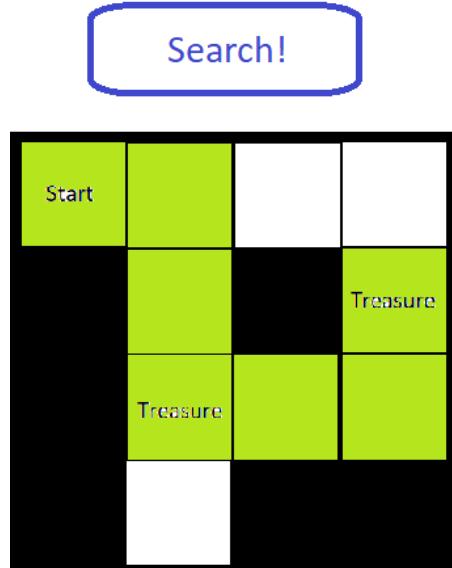
Contoh input aplikasi :



Gambar 1.3 Contoh input program

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

Contoh output Aplikasi :



Gambar 1.4 Contoh output program untuk gambar 2

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

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
e.g "maze1.txt"

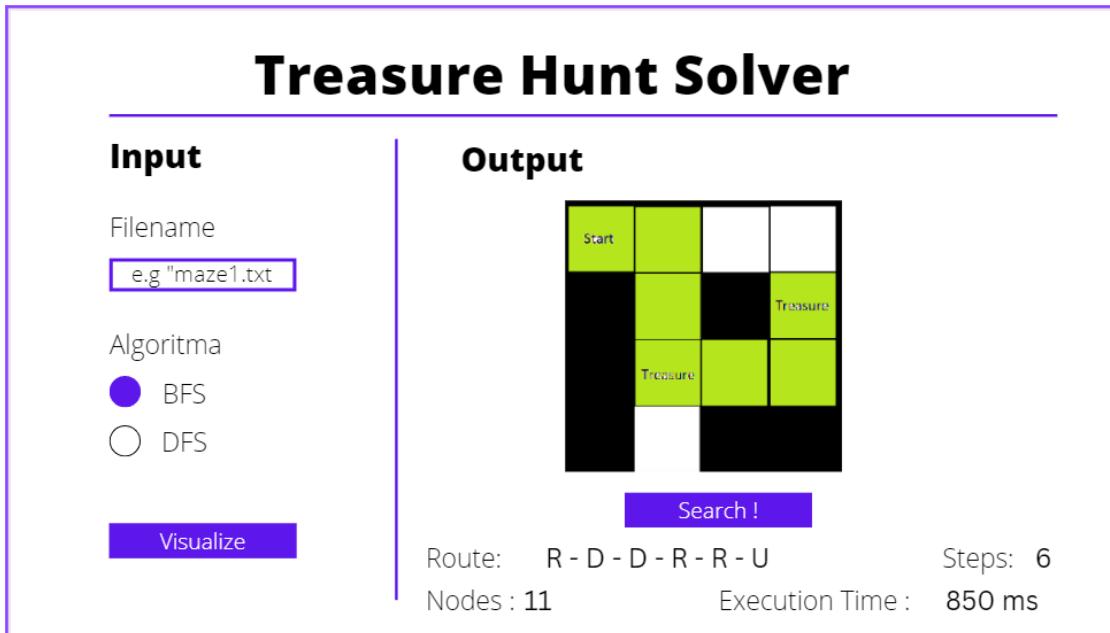
Algoritma
 BFS
 DFS

Output

Route:
Nodes :
Steps:
Execution Time :

Search !

Gambar 1.5 Tampilan Program Sebelum dicari solusinya



Gambar 1.6 Tampilan Program setelah dicari solusinya

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

Spesifikasi GUI:

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

8. GUI dapat dibuat **sekreatif** mungkin asalkan memuat 5 (7 jika mengerjakan bonus) spesifikasi di atas.

Program yang dibuat harus memenuhi spesifikasi wajib sebagai berikut:

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

Proses visualisasi ini boleh memanfaatkan pustaka atau kakas yang tersedia. Sebagai referensi, salah satu kakas yang tersedia untuk memvisualisasikan matrix dalam bentuk grid adalah **DataGridView**. Berikut adalah panduan singkat terkait penggunaannya <http://csharp.net-informations.com/datagridview/csharp-datagridview-tutorial.htm>

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

Bab 2

Landasan Teori

2.1 Graf Traversal

Graf traversal adalah proses melintasi atau memeriksa setiap simpul (node) dalam sebuah graf. Graf traversal sering digunakan dalam berbagai aplikasi seperti pemrosesan teks, pengenalan wajah, dan pemodelan kecerdasan buatan. Dalam graf traversal, kita mulai dari simpul awal (*start node*) dan melintasi graf untuk mengunjungi setiap simpul satu per satu. Terdapat 2 teknik graf traversal yang akan dibahas dalam laporan ini, yaitu *Breadth-First Search* (BFS) dan *Depth-First Search* (DFS)

2.2 Breadth-First Search

Breadth-First Search (BFS) adalah salah satu teknik paling sederhana untuk melakukan traversal pada graf. Teknik ini dimulai dari simpul awal (*start node*) dan melintasi graf secara horizontal (meluas/*breadth*) terlebih dahulu sebelum menjelajahi simpul-simpul yang berdekatan secara vertikal (kedalaman/*depth*).

Algoritma BFS bekerja dengan cara sebagai berikut:

1. Tentukan simpul awal yang akan dilintasi.
2. Masukkan simpul awal ke dalam sebuah *queue*.
3. Selama *queue* tidak kosong, lakukan langkah-langkah berikut:
 - Keluarkan simpul pertama dari *queue*.
 - Periksa apakah simpul tersebut adalah simpul tujuan yang dicari. Jika iya, proses traversal dihentikan dan simpul tersebut dikeluarkan sebagai hasil.
 - Jika simpul tersebut bukan simpul tujuan, tambahkan semua simpul yang berdekatan (tetangga) ke dalam *queue*.
4. Jika semua simpul sudah dilintasi dan simpul tujuan tidak ditemukan, kembalikan hasil null atau pesan bahwa simpul tujuan tidak ada dalam graf.

Berikut adalah contoh pseudocode untuk proses 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 w←1 to n do
        if A[v,w] = 1 then      { v dan w bertetangga }
            if not dikunjungi[w] then
                write(w)           { cetak simpul yang dikunjungi }
                MasukAntrian(q,w)
                dikunjungi[w]←true
            endif
        endif
    endfor
endwhile
{ AntrianKosong(q) }

```

Gambar 2.1 Pseudocode untuk proses BFS

sumber :

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

BFS biasanya digunakan dalam pencarian jalur terpendek pada graf dengan bobot (*weighted graph*) positif atau pencarian semua simpul pada graf yang terhubung. BFS juga dapat digunakan untuk mencari jalur terpendek pada sebuah peta, mencari kata-kata dalam kamus, atau menyelesaikan permasalahan yang membutuhkan penelusuran seluruh simpul pada graf.

Kelebihan dari BFS adalah setiap simpul akan dikunjungi secara merata dan jika simpul tujuan berada pada level yang dangkal, maka algoritma ini akan menemukannya dengan cepat. Namun, kekurangan dari BFS adalah jika graf sangat besar dan dalam, maka akan memakan banyak memori untuk menyimpan simpul-simpul pada level yang dalam tersebut.

2.3 Depth-First Search

Depth-First Search (DFS) adalah salah satu teknik sederhana lainnya untuk melakukan traversal pada graf. Teknik ini dimulai dari simpul awal (*start node*) dan melintasi setiap simpul pada graf secara vertikal (*kedalaman/depth*) terlebih dahulu sebelum menjelajahi simpul-simpul yang berdekatan secara horizontal (*meluas/breadth*).

Algoritma DFS bekerja dengan cara sebagai berikut:

1. Tentukan simpul awal yang akan dilintasi.
2. Tandai simpul awal sebagai sudah dikunjungi.
3. Jika simpul tersebut adalah simpul tujuan yang dicari, proses traversal dihentikan dan simpul tersebut dikeluarkan sebagai hasil.
4. Jika simpul tersebut bukan simpul tujuan, kunjungi simpul-simpul yang berdekatan (tetangga) secara rekursif.
5. Jika semua simpul yang terhubung dengan simpul awal sudah dikunjungi, proses traversal kembali ke simpul sebelumnya dan kunjungi simpul-simpul yang belum dikunjungi.

Berikut adalah contoh pseudocode untuk proses 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

```

Gambar 2.1 Pseudocode untuk proses DFS

sumber :

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

DFS biasanya digunakan dalam pencarian jalur pada graf dengan atau tanpa bobot (*weight*), pengurutan topologis (*topological sort*), dan deteksi siklus pada graf. DFS juga dapat digunakan dalam permasalahan pemecahan sudoku, navigasi maze, atau penyelesaian masalah yang memerlukan pengambilan keputusan.

Kelebihan dari DFS adalah algoritma ini hanya memerlukan sedikit memori untuk menyimpan simpul-simpul yang belum dikunjungi dan cocok untuk graf yang sangat besar atau dalam. Namun, kekurangan dari DFS adalah algoritma ini tidak menjamin menemukan jalur terpendek pada graf dan dapat mengalami infinite loop jika tidak diimplementasikan dengan benar.

2.4 C# Desktop Application Development

C# desktop application development adalah proses pengembangan aplikasi desktop menggunakan bahasa pemrograman C# dan .NET Framework. C# adalah bahasa pemrograman modern yang kuat dan mudah dipelajari yang dapat digunakan untuk membuat aplikasi desktop yang kompleks dan interaktif.

Proses pengembangan aplikasi desktop C# melibatkan pembuatan antarmuka pengguna (*user interface*), logika bisnis, pengolahan data, dan fitur-fitur lainnya yang dibutuhkan dalam aplikasi desktop. Dalam pengembangan *desktop application*, beberapa teknologi seperti Windows Presentation Foundation (WPF) dan Windows Forms dapat digunakan untuk membangun aplikasi desktop.

Pengembangan aplikasi desktop C# dapat dilakukan dengan menggunakan berbagai alat pengembangan yang tersedia seperti Microsoft Visual Studio, MonoDevelop, dan SharpDevelop. Keuntungan penggunaan C# dalam pengembangan aplikasi desktop adalah mudahnya integrasi dengan platform Windows, dukungan untuk berbagai jenis *database*, serta adanya dukungan komunitas yang luas.

Bab 3

Aplikasi Algoritma BFS dan DFS

3.1 Langkah-langkah Pemecahan Masalah

Treasure Hunt Solver adalah permainan yang mengharuskan menemukan sebuah harta karun di suatu tempat dengan mengikuti petunjuk-petunjuk yang diberikan. Pada umumnya, *Treasure Hunt Solver* dapat diselesaikan dengan menggunakan algoritma *Breadth-First Search* (BFS) dan *Depth-First Search* (DFS). Berikut adalah langkah-langkah pemecahan masalah *Treasure Hunt Solver* dengan menggunakan algoritma BFS dan DFS:

BFS:

1. Tentukan *node* awal(K), yaitu tempat di mana pemain berada saat memulai permainan.
2. Buat sebuah antrian yang akan digunakan untuk menyimpan *node* yang akan diproses selanjutnya.
3. Masukkan *node* awal ke dalam antrian.
4. Selama antrian tidak kosong, ambil node paling depan dari antrian.
5. Jika *node* tersebut adalah harta karun yang dicari, maka simpan node harta karun tersebut. Jika tidak, tambahkan semua tetangga dari node tersebut yang belum pernah diproses ke dalam antrian.
6. Jika node tetangga adalah halangan/*obstacle*(X) maka lanjut memproses tetangga yang lain dengan node yang mungkin diakses(R).
7. Tandai *node* tersebut sebagai sudah diproses.
8. Ulangi langkah 4 sampai 7 sampai rute semua harta karun ditemukan.

DFS:

1. Tentukan *node* awal (K), yaitu tempat di mana pemain berada saat memulai permainan.
2. Buat sebuah *stack* yang akan digunakan untuk menyimpan *node* yang akan diproses selanjutnya.
3. Masukkan *node* awal ke dalam *stack*.
4. Selama *stack* tidak kosong, ambil *node* paling atas dari *stack*.
5. Jika *node* tersebut adalah harta karun yang dicari, maka simpan node harta karun tersebut. Jika tidak, tambahkan semua tetangga dari node tersebut yang belum pernah diproses ke dalam *stack*.
6. Jika node tetangga adalah halangan/*obstacle*(X) maka lanjut memproses tetangga yang lain dengan *node* yang mungkin diakses(R).
7. Tandai *node* tersebut sebagai sudah diproses.
8. Ulangi langkah 4 sampai 7 sampai harta karun ditemukan.

Perbedaan antara algoritma BFS dan DFS adalah cara memproses *node* selanjutnya. Pada BFS, *node* diproses berdasarkan urutan kedalaman dari *node* awal. Artinya, *node* yang lebih dekat dengan *node* awal akan diproses terlebih dahulu. Sedangkan pada DFS, node diproses berdasarkan urutan kedalaman dari *node* yang sedang diproses saat itu. Artinya, *node* yang lebih dalam akan diproses terlebih dahulu. Karena itu, pada kasus tertentu, DFS dapat lebih cepat menemukan solusi daripada BFS. Namun, BFS memiliki keuntungan dalam menemukan solusi yang lebih optimal, yaitu dengan jarak minimum dari *node* awal ke *node* tujuan.

3.2 Elemen-elemen Algoritma BFS dan DFS

3.2.1. Breadth-first search

Algoritma *breadth-first search* pada permasalahan ini memanfaatkan konsep pencarian dengan graf statis, dimana graf telah terbentuk sebelum dimulainya proses pencarian. Graf untuk pencarian BFS pada program kami sesungguhnya hanya berupa sekumpulan *node* yang tersambung satu sama lain, tetapi tidak dibungkus lagi oleh struktur data graf sesungguhnya yang menampung kumpulan *nodes*. Digunakan juga sebuah *queue* untuk menyimpan *node-node* yang akan dikunjungi selanjutnya. Tidak digunakan *list* untuk menyimpan *node-node* yang telah dikunjungi karena dalam struktur *node* sendiri telah ada atribut yang menandakan apakah sebuah *node* sudah dikunjungi atau belum.

3.2.2. Depth-first search

Algoritma *depth-first search* pada permasalahan ini juga memanfaatkan konsep pencarian dengan graf statis, dimana graf telah terbentuk sebelum dimulainya pencarian. Graf direpresentasikan menggunakan sekumpulan *node*, dimana setiap *node* sendiri akan terhubung dengan *node* lainnya kemudian disimpan dalam sebuah *dictionary* untuk dinyatakan sebagai suatu graf utuh. Digunakan juga sebuah *stack* untuk menyimpan *node* yang akan dikunjungi dalam proses pencarian, selain itu dipakai juga suatu list untuk menyimpan *node-node* yang telah dikunjungi agar proses pencarian tidak dilakukan berulang-ulang pada node yang sama.

3.3 Ilustrasi dengan Kasus Lain

Berikut adalah ilustrasi pemecahan masalah dengan menggunakan algoritma BFS dan DFS pada kasus lain:

Kasus Mencari jalan terpendek dari titik A ke titik B dalam sebuah labirin merupakan salah satu kasus penerapan algoritma BFS atau DFS dalam pemecahan masalah. Dalam kasus ini, terdapat sebuah labirin yang terdiri dari berbagai rute atau jalan yang saling terhubung. Titik A dan titik B adalah dua titik di dalam labirin yang menandakan titik awal dan titik tujuan yang ingin dicapai.

Tugas yang harus dilakukan dalam kasus ini adalah mencari jalan terpendek yang dapat dilalui dari titik A ke titik B. Dalam menyelesaikan masalah ini, algoritma BFS dan DFS dapat digunakan untuk memeriksa semua kemungkinan jalan yang dapat dilalui dari titik A ke titik B. Pada dasarnya, algoritma ini akan menelusuri semua jalan yang tersedia di dalam labirin dan mencatat jalur terpendek yang ditemukan selama proses pencarian.

Pada kasus ini, labirin dapat direpresentasikan dalam bentuk matriks atau grafik, di mana setiap titik di dalam labirin direpresentasikan sebagai simpul atau node dalam grafik. Kemudian, untuk setiap node dalam grafik, akan ditentukan tetangga atau simpul terdekat yang dapat dicapai dari node tersebut. Setelah grafik dan semua tetangga dari setiap node telah ditentukan, maka algoritma BFS atau DFS dapat diterapkan pada grafik tersebut untuk mencari jalan terpendek dari titik A ke titik B.

Dalam penyelesaian kasus ini, penggunaan algoritma BFS atau DFS akan tergantung pada karakteristik dari labirin dan tujuan pencarian. Jika labirin memiliki cabang yang banyak dan jarak yang cukup jauh antara titik A dan titik B, maka algoritma BFS akan lebih efektif. Namun, jika labirin relatif linear dan jarak antara titik A dan titik B relatif pendek, maka algoritma DFS dapat digunakan.

BFS :

1. Tentukan *node* awal, yaitu titik A di dalam labirin.
2. Buat sebuah antrian yang akan digunakan untuk menyimpan *node* yang akan diproses selanjutnya.
3. Masukkan *node* awal ke dalam antrian.
4. Selama antrian tidak kosong, ambil *node* paling depan dari antrian.
5. Jika *node* tersebut adalah titik B, maka permainan selesai. Jika tidak, tambahkan semua tetangga dari *node* tersebut yang belum pernah diproses ke dalam antrian.
6. Tandai *node* tersebut sebagai sudah diproses.
7. Ulangi langkah 4 sampai 6 hingga titik B ditemukan.

DFS:

1. Tentukan *node* awal, yaitu titik A di dalam labirin.
2. Buat sebuah *stack* yang akan digunakan untuk menyimpan *node* yang akan diproses selanjutnya.
3. Masukkan *node* awal ke dalam *stack*.
4. Selama *stack* tidak kosong, ambil node paling atas dari *stack*.
5. Jika *node* tersebut adalah titik B, maka permainan selesai. Jika tidak, tambahkan semua tetangga dari *node* tersebut yang belum pernah diproses ke dalam *stack*.
6. Tandai *node* tersebut sebagai sudah diproses.
7. Ulangi langkah 4 sampai 6 hingga titik B ditemukan.

Perbedaan antara algoritma BFS dan DFS pada kasus ini adalah cara memproses *node* selanjutnya. Pada BFS, *node* diproses berdasarkan urutan jarak dari node awal. Artinya, *node* yang lebih dekat dengan *node* awal akan diproses terlebih dahulu. Sedangkan pada DFS, *node* diproses berdasarkan urutan kedalaman dari *node* yang sedang diproses saat itu. Artinya, *node* yang lebih dalam akan diproses terlebih dahulu. Karena itu, pada kasus tertentu, DFS dapat lebih cepat menemukan solusi daripada BFS. Namun, BFS memiliki keuntungan dalam menemukan solusi yang lebih optimal, yaitu dengan jarak minimum dari *node* awal ke *node* tujuan.

Bab 4

Implementasi dan Analisis Pemecahan Masalah

4.1 Implementasi Program

Source code program dapat diakses pada : https://github.com/rizkrysdy28/Tubes2_Chastag

Berikut adalah pseudocode untuk pencarian harta karun dengan algoritma BFS dan DFS.

4.1.1 huntBFS

```
procedure huntBFS (output GoPath : List of String, input nodes : Dictionary<Tuple<int, int, Node>> )
{ Mencari jalur pencarian treasure dalam maze
  I.S nodes terdefinisi, telah dibentuk nodes yang merepresentasikan maze
  F.S Terbentuk jalur pencarian treasure dengan BFS }

KAMUS
curr : Node
queue : Queue
nodes : Dictionary<Tuple<int, int>, Node>
GoPath : List of String
map : Matrix of Char ( Array of Array of Char )
last : Tuple<int, int>
{ this sendiri dalam pseudocode ini artinya adalah objek dari BFS (karena menggunakan OOP) }

ALGORITMA
{ inisialisasi node awal pencarian }
curr = nodes[this.start]

{ tandai node telah di kunjungi }
curr.setVisitNode()

{ loop utama untuk mencari treasure }
do
)
{ melakukan pencarian melebar ke node yang memungkinkan untuk dikunjungi }
{ prioritas = Utara, Timur, Selatan, Barat }

  if curr.getUtara() != null and !nodes[curr.getUtara()].isVisited() then
    nodes[curr.getUtara()].connectPath(curr)
    nodes[curr.getUtara()].setVisitNode()
    queue.Enqueue(nodes[curr.getUtara()])

  if curr.getTimur() != null and !nodes[curr.getTimur()].isVisited() then
    nodes[curr.getTimur()].connectPath(curr)
    nodes[curr.getTimur()].setVisitNode()
```

```

queue.Enqueue(nodes[curr.getTimur()])

if curr.getSelatan() != null and !nodes[curr.getSelatan()].isVisited()
then
    nodes[curr.getSelatan()].connectPath(curr)
    nodes[curr.getSelatan()].setVisitNode()
    queue.Enqueue(nodes[curr.getSelatan()])

if curr.getBarat() != null and !nodes[curr.getBarat()].isVisited() then
    nodes[curr.getBarat()].connectPath(curr)
    nodes[curr.getBarat()].setVisitNode()
    queue.Enqueue(nodes[curr.getBarat()])

if queue.Count = 0 then
    break

curr = queue.Dequeue()
curr.setVisitNode()

{ mengecek apakah curr node adalah treasure }

if map[curr.getCoor().Item2, curr.getCoor().Item1].Equals('T') then

    { kurangi 1 treasure }
    treasure = treasure - 1

    { hapus isi queue }
    queue.Clear()

    clearVisitNode(curr)
    map[curr.getCoor().Item2, curr.getCoor().Item1] = 'R'
    queue.Enqueue(nodes[curr.getCoor()])
    this.GoPath = curr.getPathString()
    this.last = curr.getCoor()
)while treasure != 0

{ cleaning up }
clearVisitedNode(curr)
curr.clearNodePath()
queue.Clear()

```

4.1.2 runDFS

```

procedure runDFS ( output DFSPath : List of Tuple<int, int, string> , input
theGraph : Graph )
{ Mencari jalur pencarian treasure dalam maze
 I.S graph terdefinisi, telah dibentuk graph yang merepresentasikan      maze
 F.S Terbentuk jalur pencarian treasure dengan DFS }

KAMUS
theGraph : Graph
DFSPath : List of Tuple<int, int, string>

```

```

trs : Integer
dir : String
curr, next : Node
pileOfTreasure : List of Node { Node berisi treasure }
g : Tuple<int, int, string>
 kvp : KeyValuePair<int, Node>
check : Stack of Node
exploredNode : List of Node
c : Char
{ this sendiri dalam pseudocode ini artinya adalah objek dari DFS (karena
menggunakan OOP) }

```

ALGORITMA

```

{ inisialisasi stack dan mencari starting point (K) }
this.findStart()
trs = this.theGraph.getNumOfTreasure()
dir = "LDUR" { prioritas arah }

{ loop utama pencarian treasure }
while this.theGraph.getNumOfTreasure() > 0 do
    curr = check.Pop()

    if(curr.isTreasure()) then
        this.theGraph.setNumOfTreasures(this.theGraph.getNumOfTreasure() -1)
        this.pileOfTreasure.Add(curr)
        curr.setTreasure(false)

        { menyimpan path dari node treasure yang telah ditemukan }
        g = curr.getPath()
        g.Reverse()

        { menyimpan path dari node treasure ke DFSPath }
        for i = 0 to g.Count-1
            this.DFSPath.Add(g[i])

        { melakukan reset path dari setiap node dalam graph }
        for kvp in this.theGraph.nodes
            kvp.Value.clearPath()

        this.check.clear()
        this.exploredNode.clear()

        next = curr
        for c in dir
            if(curr.hasNeighbor())
                if (c = 'L' and curr.getLeft() != null) then
                    next = curr.getLeft()
                    next.connectPath(curr)

                else if (c = 'D' and curr.getLeft() != null) then
                    next = curr.getLeft()
                    next.connectPath(curr)

                else if (c = 'U' and curr.getLeft() != null) then
                    next = curr.getLeft()

```

```

        next.connectPath(curr)

        else if (c = 'R' and curr.getLeft() != null) then
            next = curr.getLeft()
            next.connectPath(curr)

        else if this.isNodeInList(next) then
            continue

        this.exploredNode.Add(next)
        this.check.Push(next)
    
```

4.2 Struktur Data dalam Program

4.2.2 Graph

Dalam menyelesaikan permasalahan *treasure hunt*, digunakan struktur data Graph (dibuat dalam bentuk class) yang diimplementasikan dengan menggabungkan sekumpulan Node. Tujuan dari penggunaan struktur data Graph ini adalah untuk merepresentasikan *maze treasure hunt* yang akan ditelusuri saat menjalankan algoritma DFS. Struktur data Graph tepat untuk digunakan karena dapat menggambarkan keterhubungan jalur antara suatu kotak dengan kotak lainnya dalam maze dengan baik. Tiap kotak dalam maze adalah Node dari Graph itu sendiri. Struktur data Graph sendiri merupakan agregasi dari struktur data Node.

4.2.2.1 Node

Node adalah sebuah struktur data dalam graph yang terdiri dari beberapa atribut. Dalam implementasi ini, atribut Node terdiri dari beberapa atribut, yaitu

- a. name yang bertipe string dan berfungsi sebagai identifikasi Node.
- b. x_kor bertipe int, berisi nilai koordinat X dari Node pada Graph (berkorespondensi dengan matriks pada file input).
- c. y_kor bertipe int, berisi nilai koordinat Y dari Node pada Graph (berkorespondensi dengan matriks pada file input).
- d. visited yang bertipe boolean dan menandakan apakah Node sudah dikunjungi atau belum.
- e. treasure bertipe boolean , berfungsi untuk menandai apakah Node merupakan lokasi treasure atau bukan.
- f. path yang bertipe List<Tuple<int, int, string>> berisi informasi jalur yang ditempuh untuk mencapai Node tersebut.
- g. up bertipe Node, berisi referensi ke Node yang posisinya berada di atas Node ini, null bila tidak ada.
- h. down bertipe Node, berisi referensi ke Node yang posisinya berada di bawah Node ini, null bila tidak ada.
- i. left bertipe Node, berisi referensi ke Node yang posisinya berada di kiri Node ini, null bila tidak ada.

- j. right bertipe Node, berisi referensi ke Node yang posisinya berada di kanan node ini, null bila tidak ada.

4.2.3 Queue

Struktur data Queue diimplementasikan dengan menggunakan *collection* pada kelas *System.Collections.Generic*. Tujuan dari penggunaan struktur data Queue ini adalah untuk menyimpan antrian Node, yang merepresentasikan jalur yang akan dikunjungi oleh program saat melakukan traversal dengan metode BFS (Breadth-First Search). Berikut adalah method yang digunakan pada struktur data Queue.

- a. Enqueue, berfungsi untuk memasukkan Node ke dalam Queue dengan aturan FIFO atau first in first out.
- b. Dequeue, berfungsi untuk mengeluarkan Node dari Queue dengan aturan FIFO atau first in first out.
- c. Count, berfungsi untuk mendapatkan jumlah Node yang sedang berada dalam Queue.
- d. Clear, berfungsi untuk mengosongkan Queue secara langsung.

4.2.4 Stack

Implementasi struktur data Stack menggunakan collection pada kelas *System.Collections.Generic*. Stack digunakan untuk menyimpan tumpukan Node yang akan dikunjungi oleh program saat melakukan traversal dengan metode DFS (Depth-First Search). Berikut adalah method yang digunakan pada struktur data Stack.

- a. Push, berfungsi untuk memasukkan Node ke dalam Stack dengan aturan LIFO atau last in first out.
- b. Pop, berfungsi untuk mengeluarkan Node dari Stack dengan aturan LIFO atau last in first out.
- c. Count, berfungsi untuk mendapatkan jumlah Node yang sedang berada dalam Stack.
- d. Clear, berfungsi untuk mengosongkan Stack secara langsung.

4.2.5 List

List merupakan struktur data dalam pemrograman yang dapat digunakan untuk menyimpan sekumpulan elemen atau objek dalam urutan tertentu. List ini memungkinkan penambahan, penghapusan, dan pencarian elemen. Berikut adalah method yang digunakan pada struktur data List.

- a. Add, berfungsi untuk menambahkan elemen ke dalam List.
- b. Clear, berfungsi untuk mengosongkan List.
- c. Count, berfungsi untuk menghitung jumlah elemen dalam List.

4.3 Tata Cara Penggunaan Program

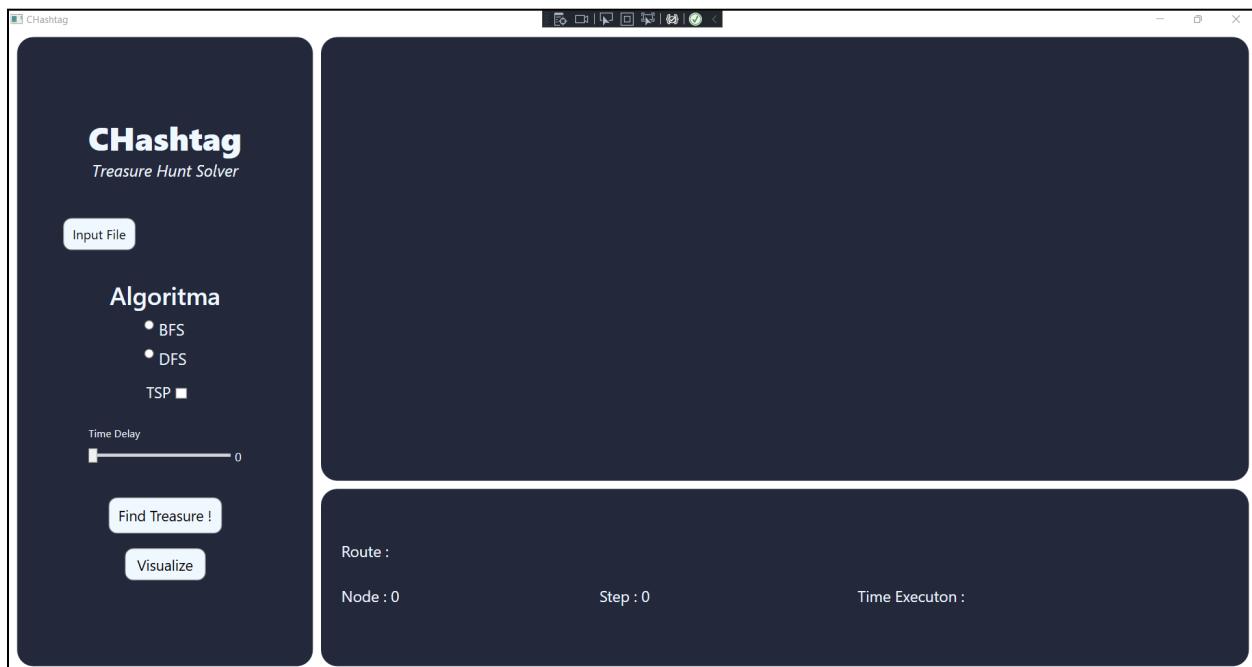
1. Pastikan sudah menginstall semua requirements yang diperlukan.

2. Jalankan Visual Studio Code.
3. Buka folder src\Chastag\Chastag.
4. Cari file MainWindow.xaml.cs lalu copy pathnya.
5. Buka terminal pada folder src\Chastag\Chastag
6. Jalankan program dengan cara menuliskan dotnet run “path MainWindow.xaml.cs” pada terminal.
7. Setelah tampilan program muncul, klik input file lalu pilih file .txt dari maze.
8. Kemudian, pilih algoritma yang ingin digunakan (BFS/DFS).
9. Centang pada bagian TSP apabila rute solusi yang diperoleh juga harus kembali ke titik awal setelah menemukan segala harta karunnya
10. Tekan Find Treasure ! untuk menemukan semua treasure yang ada.
11. Tekan Visualize untuk menampilkan rute yang mengunjungi semua treasure yang ada.

4.4 Analisis dan Hasil Pengujian

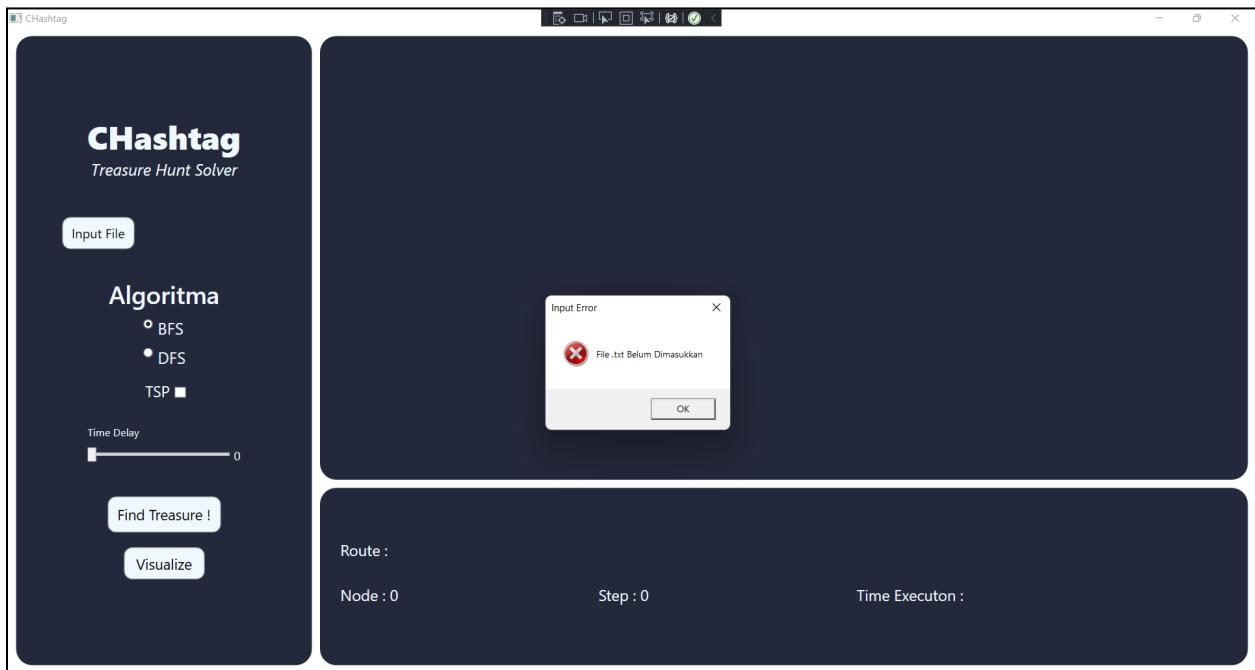
4.4.1 Tampilan Antarmuka Program

Berikut adalah *screenshot* tampilan antarmuka program saat program dimulai



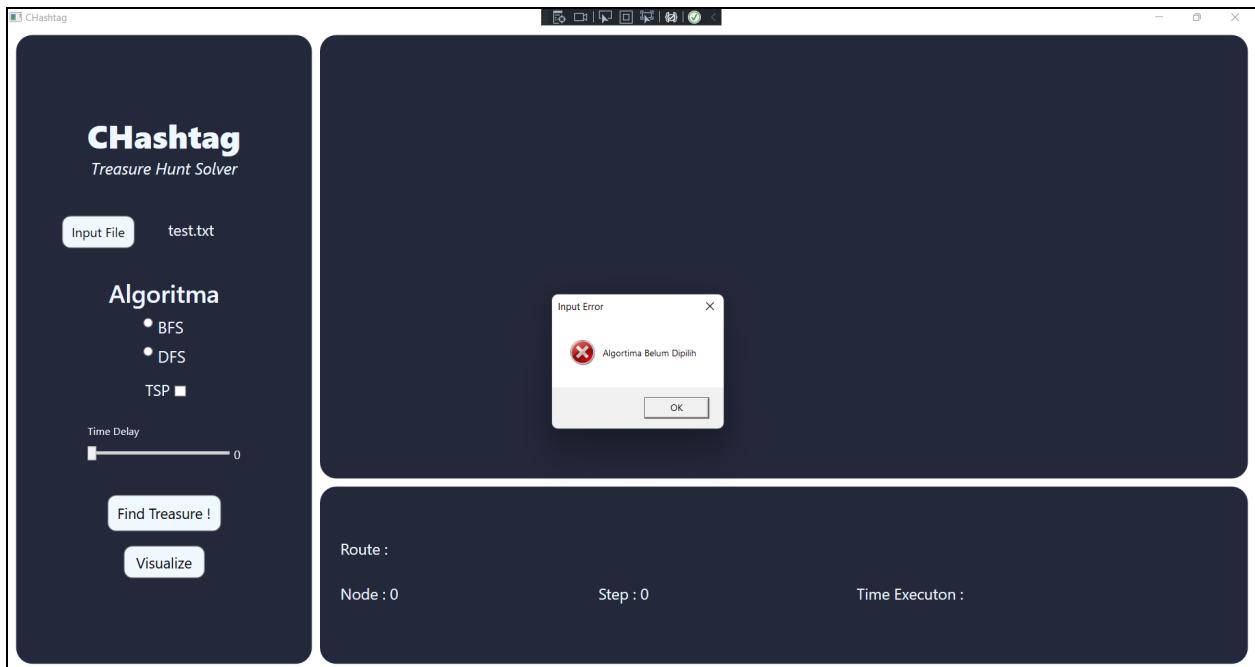
Gambar 4.1 Tampilan saat program dimulai

Berikut adalah tampilan antarmuka program saat algoritma dijalankan tetapi *file input* belum dimasukan



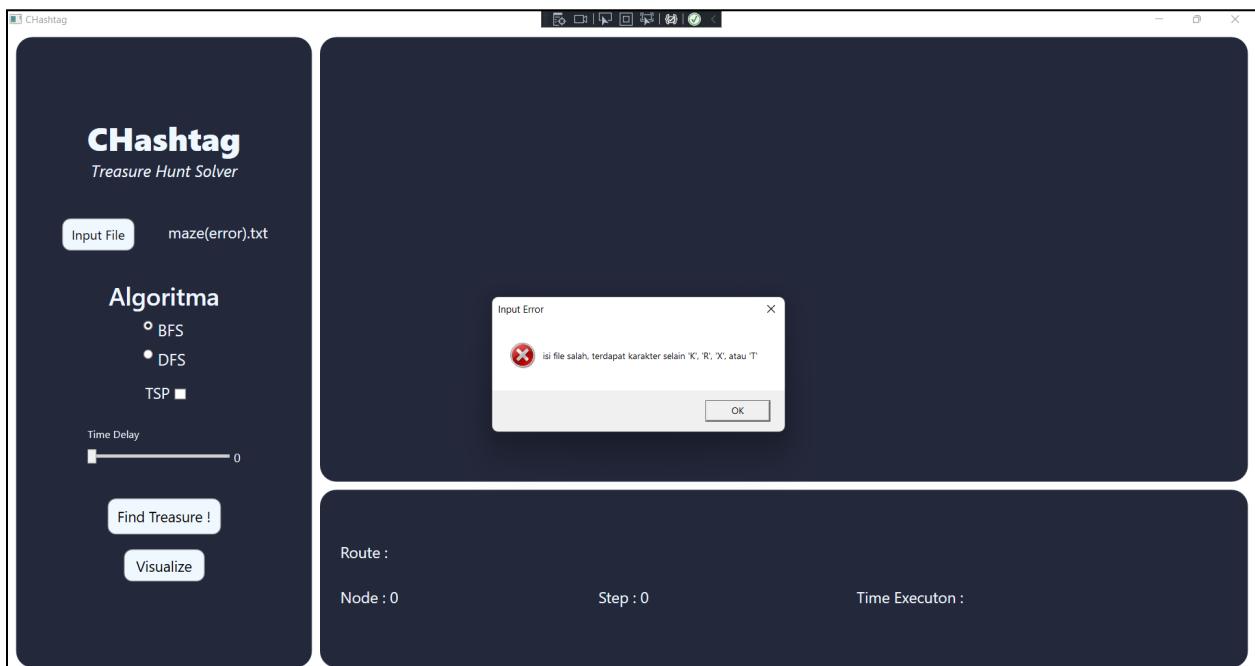
Gambar 4.2 Tampilan *exception handling* program saat *file* lupa dimasukan

Berikut adalah tampilan antarmuka program saat dijalankan tetapi algoritma tidak dipilih terlebih dahulu



Gambar 4.3 Tampilan *exception handling* program saat algoritma tidak dipilih

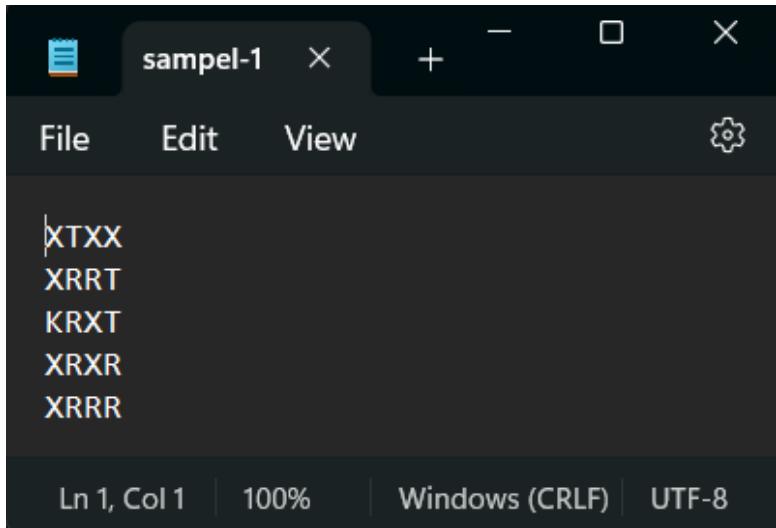
Berikut adalah tampilan antarmuka program saat *file input* memiliki karakter selain K, X, R, T



Gambar 4.4 Tampilan *exception handling* saat *file input* memiliki karakter tidak valid

4.4.2 Pengujian program dengan *test case* Sampel

1. TestCase1 : Sampel-1.txt

Sampel-1.txt	 <p>The screenshot shows a dark-themed text editor window titled "sampel-1". The content of the file is as follows:</p> <pre> XTXX XRRT KRXT XRXR XRRR</pre> <p>At the bottom of the editor, it displays "Ln 1, Col 1 100% Windows (CRLF) UTF-8".</p>
(BFS) Find Treasure !	 <p>The screenshot shows the CHashtag Treasure Hunt Solver application interface. The left panel shows the input file "sampel-1.txt" and the algorithm selection (BFS is selected). The right panel displays a 5x5 grid representing a treasure map. The grid contains several treasures (green squares), a KrustyKab (yellow square), and some gray squares. The route taken by the algorithm is highlighted in yellow, starting from the KrustyKab and leading to a treasure. The output panel at the bottom shows the route "R - U - U - D - R - R - D", 12 nodes visited, 7 steps, and a execution time of 0.0009 microsecond.</p>
(BFS) Visualize	 <p>The screenshot shows the CHashtag Treasure Hunt Solver application interface, similar to the previous one but with a different color scheme for the tokens. The left panel shows the input file "sampel-1.txt" and the algorithm selection (BFS is selected). The right panel displays a 5x5 grid representing a treasure map. The grid contains several tokens (green, yellow, and black squares), a KrustyKab (yellow square), and some gray squares. The route taken by the algorithm is highlighted in yellow, starting from the KrustyKab and leading to a token. The output panel at the bottom shows the route "R - U - U - D - R - R - D", 12 nodes visited, 7 steps, and a execution time of 0.0009 microsecond.</p>

(BFS) TSP

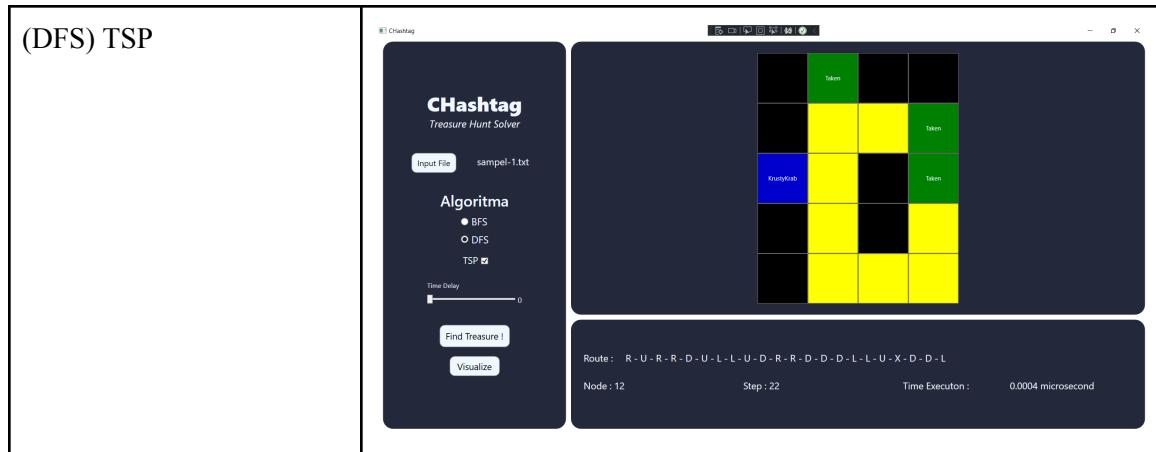


(DFS) Find Treasure !



(DFS) Visualize

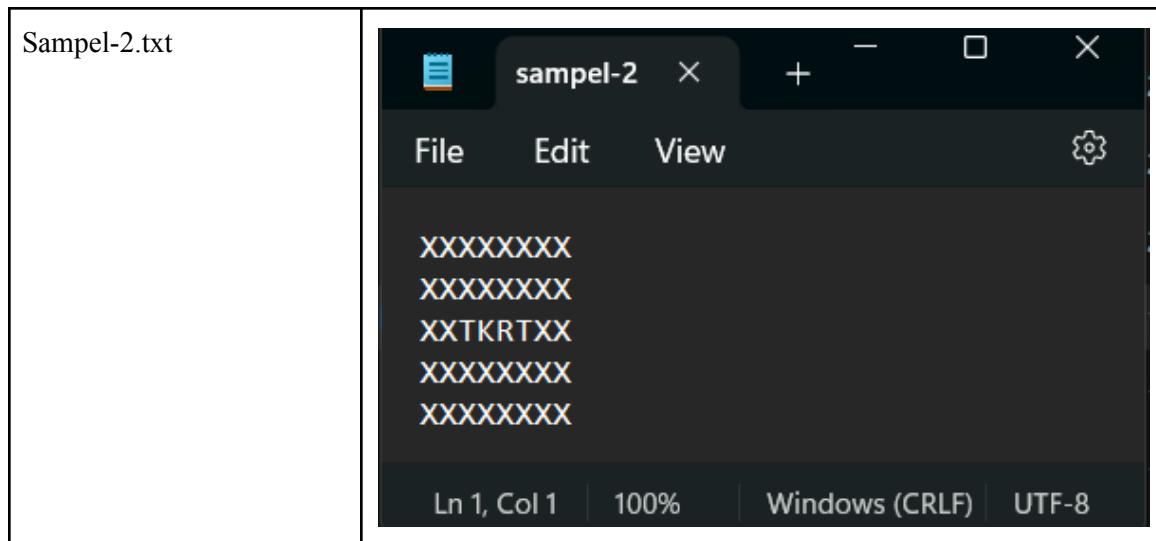




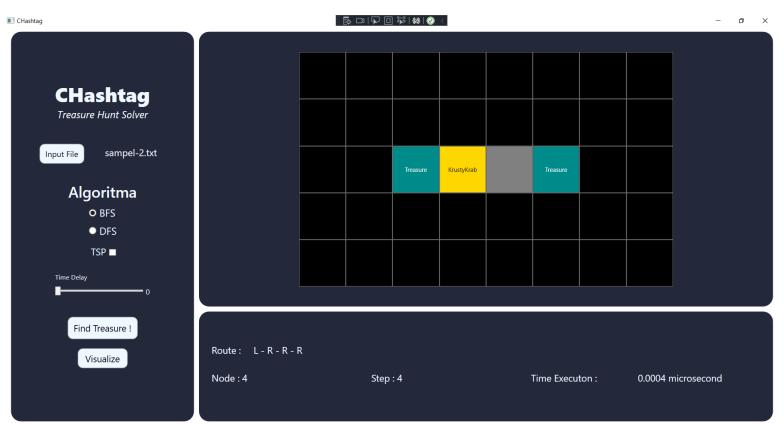
Analisis Hasil Pengujian :

Pada *test case* pertama, dapat dilihat bahwa jumlah *step* dan waktu yang diperlukan oleh metode BFS lebih sedikit dibanding yang diperlukan oleh metode DFS, hal ini menandakan metode BFS lebih efisien dalam *test case* pertama. Salah satu alasan yang menyebabkan metode DFS lebih tidak efisien adalah karena posisi *treasure* pada map dan juga prioritas gerakan dari DFS itu sendiri yang menyebabkan jalannya dalam mencari *treasure* menjadi sedikit bolak balik dan lebih memakan waktu.

2. TestCase2 : Sampel-2.txt



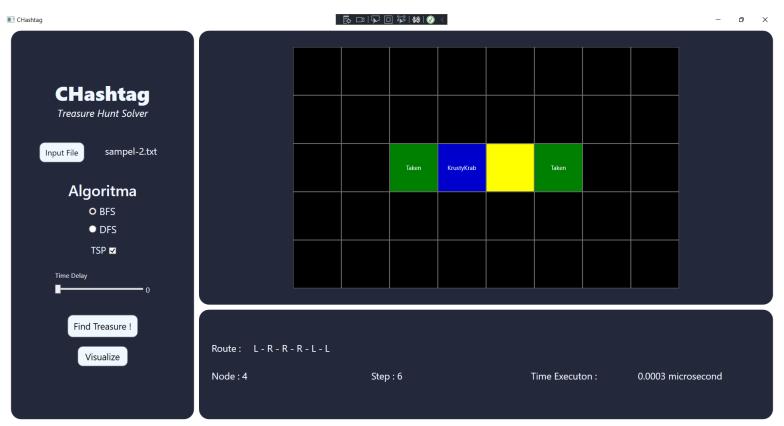
(BFS) Find Treasure !



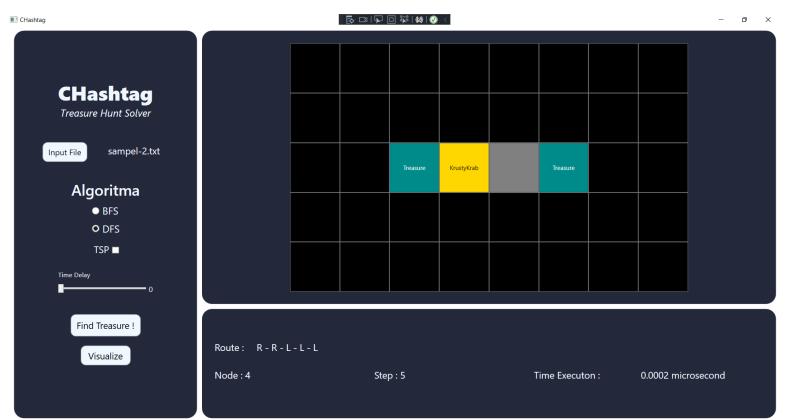
(BFS) Visualize



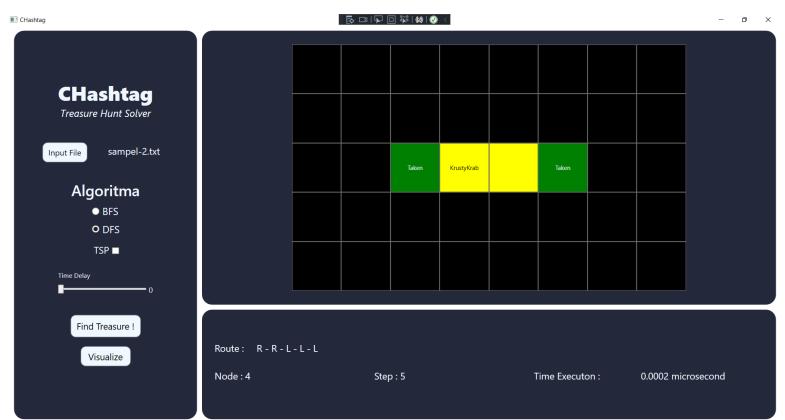
(BFS) TSP



(DFS) Find Treasure !



(DFS) Visualize



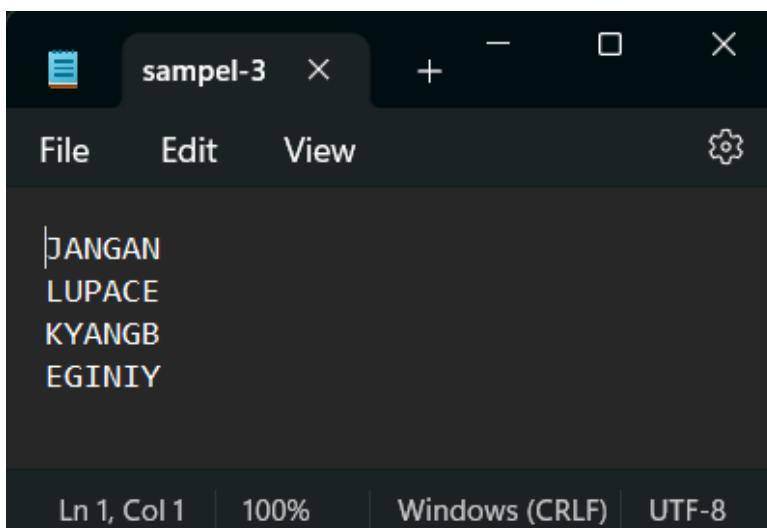
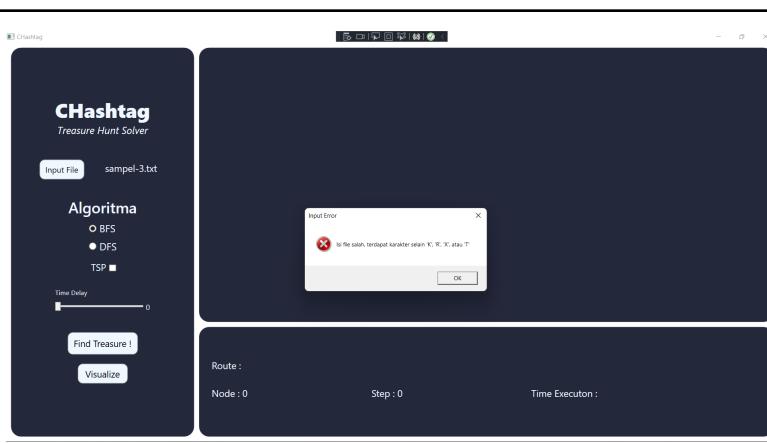
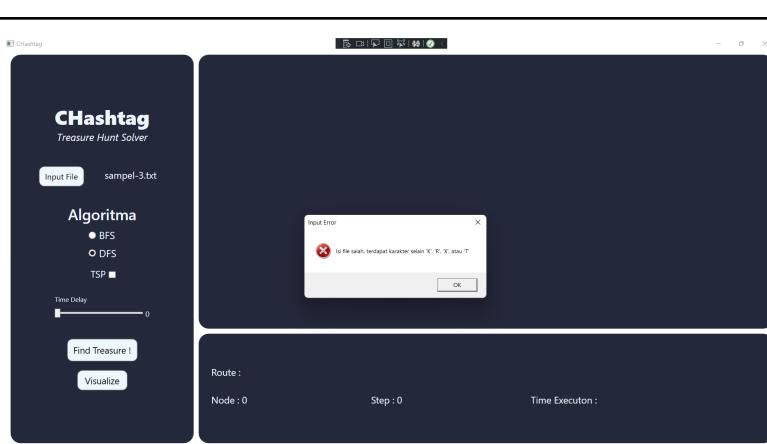
(DFS) TSP



Analisis Hasil Pengujian :

Pada *test case* kedua, dapat dilihat bahwa jumlah langkah yang dihasilkan oleh metode BFS lebih sedikit dibanding metode DFS. Namun, waktu yang diperlukan oleh metode DFS lebih singkat dibanding BFS, hal ini menandakan bahwa BFS pada kasus *treasure hunt* ini memang akan selalu menghasilkan langkah yang paling optimal (minimum), tetapi jika dilihat dari waktu yang diperlukan untuk pemrosesannya maka metode DFS bisa saja lebih unggul pada kasus tertentu, contohnya seperti kasus pada *test case* kedua ini.

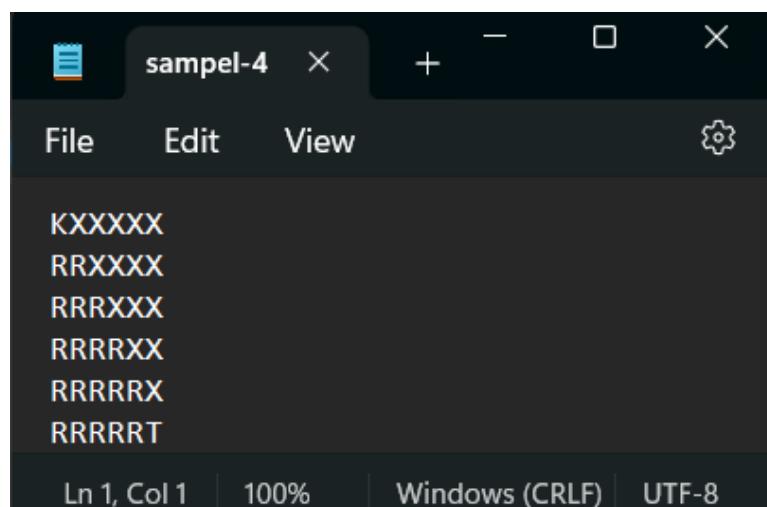
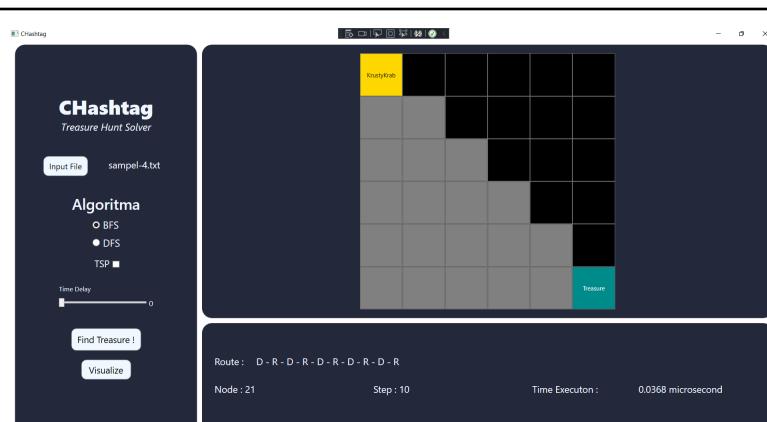
3. TestCase3 : Sampel-3.txt

Sampel-3.txt	
(BFS) Find Treasure !	
(DFS) Find Treasure !	

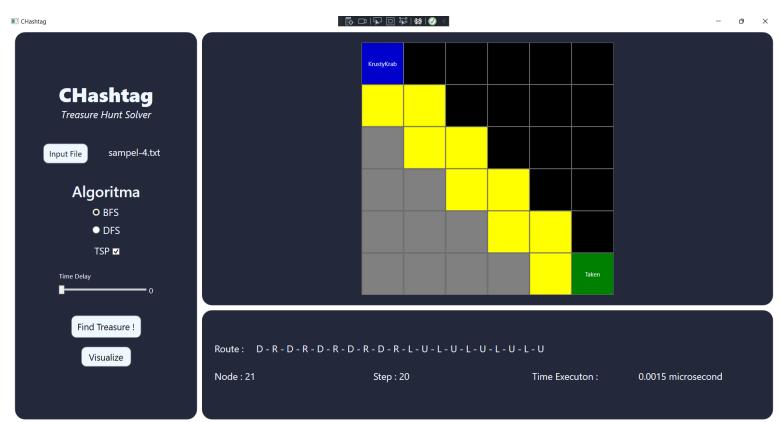
Analisis Hasil Pengujian :

Program telah berhasil mendeteksi *error* akibat adanya karakter yang tidak sesuai pada *file input* dan berhasil menampilkan pesan *error* ke layar.

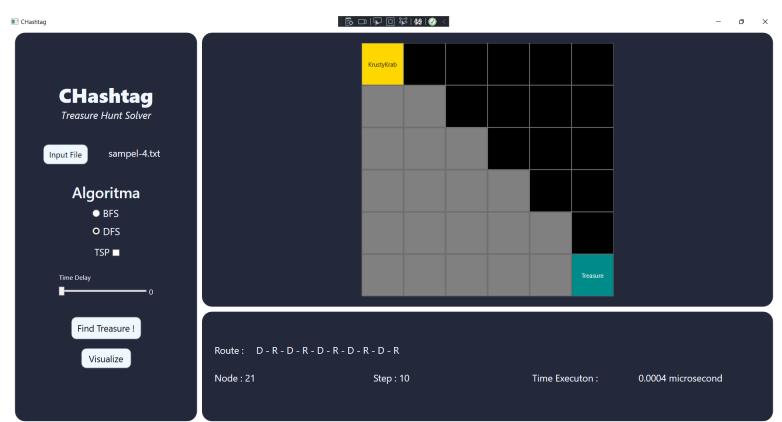
4. TestCase4 : Sampel-4.txt

Sampel-4.txt	
(BFS) Find Treasure !	
(BFS) Visualize	

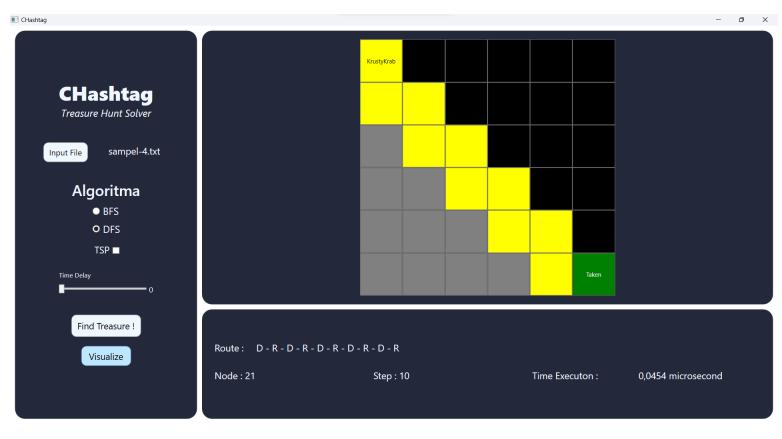
(BFS) TSP

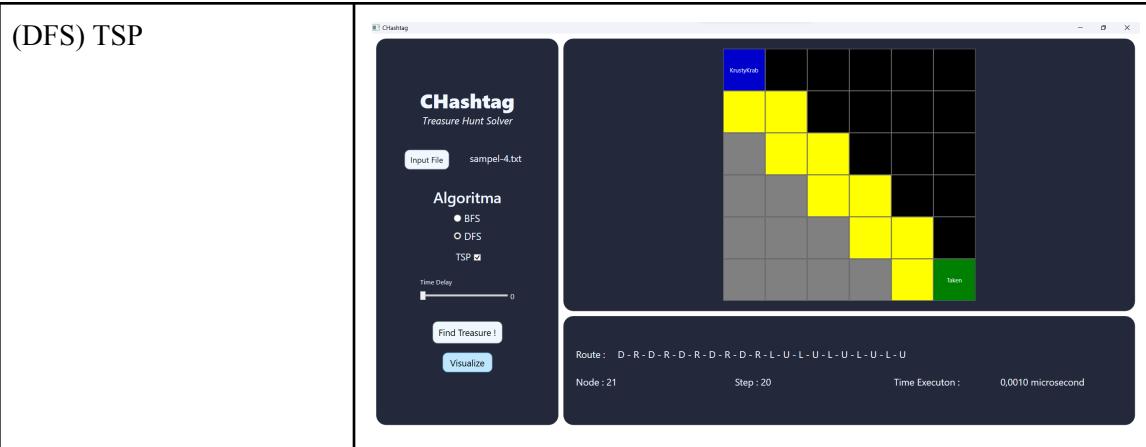


(DFS) Find Treasure !



(DFS) Visualize

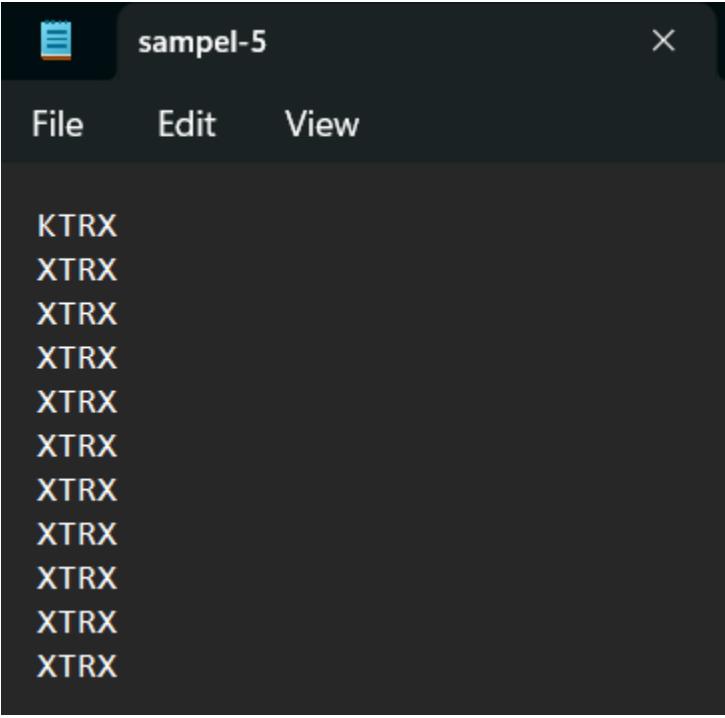


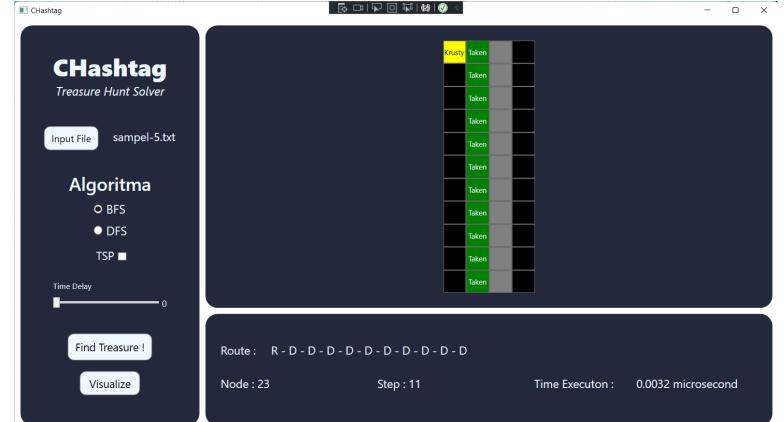


Analisis Hasil Pengujian :

Pada *test case* keempat, metode BFS dan DFS sama-sama menghasilkan langkah yang minimum untuk mendapatkan *treasure*, kesamaan langkah ini terjadi karena pada metode DFS, prioritas pergerakan yang ditetapkan adalah *Right* (kanan), *Up* (atas), *Down* (bawah), *Left* (kiri), yang menyebabkan jalur pencarinya seperti pada hasil pengujian. Secara kebetulan, jalur pencarinya juga adalah salah satu jalur tercepat dan terpendek untuk mencapai *treasure* yang dalam hal ini pasti sama dengan yang dihasilkan oleh metode BFS. Secara waktu eksekusi, metode DFS tetap lebih unggul dibanding BFS saat menghasilkan jalur pencarian akhir yang sama, hal ini karena ruang pencarian di belakang layar yang dilakukan oleh algoritma BFS sesungguhnya lebih luas daripada hasil akhir yang ditampilkan, tetapi pada metode DFS pencarian yang ditampilkan sebagai hasil akhir adalah benar-benar pencarian yang dilakukan oleh algoritma.

5. TestCase5 : Sampel-5.txt

Sampel-5.txt	 <p>The terminal window displays the following text: KTRX XTRX XTRX XTRX XTRX XTRX XTRX XTRX XTRX XTRX</p>
(BFS) Find Treasure !	 <p>The application interface shows the following details: Input File: sampel-5.txt Algorithm: DFS (selected) TSP: ■ Time Delay: 0 Buttons: Find Treasure!, Visualize Route: R - D - D - D - D - D - D - D - D - D - D Node: 23 Step: 11 Time Execution: 0.0032 microsecond A grid visualization on the right shows a path from 'Start' to 'Treasure' through various nodes.</p>

(BFS) Visualize	
(BFS) TSP	
(DFS) Find Treasure !	

(DFS) Visualize	
(DFS) TSP	

Analisis Hasil Pengujian :

Pada *test case* kelima, dapat dilihat bahwa langkah yang dihasilkan oleh metode BFS jauh lebih minim dibanding langkah yang dihasilkan oleh metode DFS, hal ini karena urutan prioritas langkah dalam metode DFS dan juga implementasi DFS yang akan melupakan *node-node* yang pernah dikunjunginya segera setelah menemukan sebuah treasure dan hanya akan menyambung jalur dari penemuan treasure tersebut. Hal inilah yang menyebabkan metode DFS banyak berputar-putar dalam *maze* sehingga tidak menghasilkan jalur yang optimal dan juga menyebabkan waktu eksekusi menjadi cukup lama.

Bab 5

Kesimpulan dan Saran

5.1 Kesimpulan

Algoritma *breadth-first search* dan *depth-first search* dapat digunakan untuk menyelesaikan permasalahan pencarian *treasure hunt* di dalam sebuah *maze*, permasalahan ini dapat digeneralisir menjadi pencarian objek di dalam *maze*. Algoritma *breadth-first search* akan melakukan penelusuran secara melebar terlebih dahulu menyusuri semua langkah dalam *maze* secara melebar, sedangkan algoritma *depth-first search* akan bergerak mengikuti satu jalur yang terus bersambung sampai menemukan kondisi *stuck*, baru melakukan *backtrack* dan mencari di jalur lain

Pengembangan *desktop application* dapat dilakukan dengan menggunakan IDE berupa Visual Studio dan bahasa pemrograman C#. Pengembangan GUI sangat dipermudah dengan adanya WinForm.

5.2 Saran

Pada pengerjaan tugas besar ini, terdapat kendala waktu karena diperlukan waktu untuk mempelajari dan memahami cara kerja algoritma BFS dan DFS dalam pemecahan kasus Mencari Harta Tuan Krab sebelum membuat implementasi program. Agar waktu mengerjakan tugas cukup, berikut adalah hal yang perlu dilakukan:

1. Segera pelajari fungsi-fungsi yang belum dipahami sehingga dapat diidentifikasi masalah seperti apa yang akan ditemui dan seberapa lama waktu yang dibutuhkan untuk menyelesaikan permasalahan tersebut.
2. Segera membagi tugas agar pekerjaan dapat segera diselesaikan sesuai dengan kesepakatan pembagian tugas.
3. Tidak menunda-nunda pengerjaan tugas.

5.3 Refleksi

Dengan adanya tugas besar 2 IF2211 Strategi Algoritma, mahasiswa menjadi semakin paham pentingnya algoritma BFS dan DFS dalam memecahkan permasalahan yang mungkin kita temui sehari-hari. Pada tugas kali ini juga diperlukan koordinasi antar anggota penyusun agar dapat menyelesaikan setiap masalah yang terjadi. Meskipun dapat menyelesaikan tugas dengan tepat waktu, mungkin masih terdapat beberapa kesalahan.

5.4 Tanggapan Terkait Tugas Besar

Tugas besar 2 IF2211 Strategi Algoritma dengan topik algoritma BFS dan DFS membuat kami sebagai mahasiswa IF'21 dapat berlatih menerapkan algoritma BFS dan DFS yang telah dipelajari di kelas langsung ke dalam bentuk program yang bisa menyelesaikan masalah sesungguhnya. Tugas besar ini sangat bermanfaat untuk latihan berpikir bagi mahasiswa, selain itu tugas ini juga memberikan kesempatan bagi mahasiswa untuk mengeksplor bahasa baru yang belum pernah digunakan pada tugas mata kuliah lain sebelumnya, yaitu C#, dan juga menggunakan IDE Visual Studio untuk mengembangkan desktop application yang memiliki GUI yang menarik. Secara keseluruhan, tugas besar ini sangat membantu dan mendorong mahasiswa untuk berkembang lebih baik lagi.

DAFTAR PUSTAKA

Link Repository : https://github.com/rizkyrsyd28/Tubes2_ChashTag

Link Video Youtube : <https://youtu.be/oWim0CspdV0>

REFERENSI :

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

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

https://www.youtube.com/watch?v=fZxZswmC_BY&list=PLA8ZIAm2I03hS41Fy4vFpRw8AdYNBXmNm