LAPORAN TUGAS BESAR II
IF2211 STRATEGI ALGORITMA
APLIKASI ALGORITMA BFS DAN DFS DALAM MENYELESAIKAN MAZE TREASURE HUNT

Disusun oleh:

13521043	Nigel Sahl
13521058	Ghazi Akmal Fauzan
13521070	Akmal Mahardika Nurwahyu Pratama

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

BANDUNG 2023

Daftar Isi

BAB I

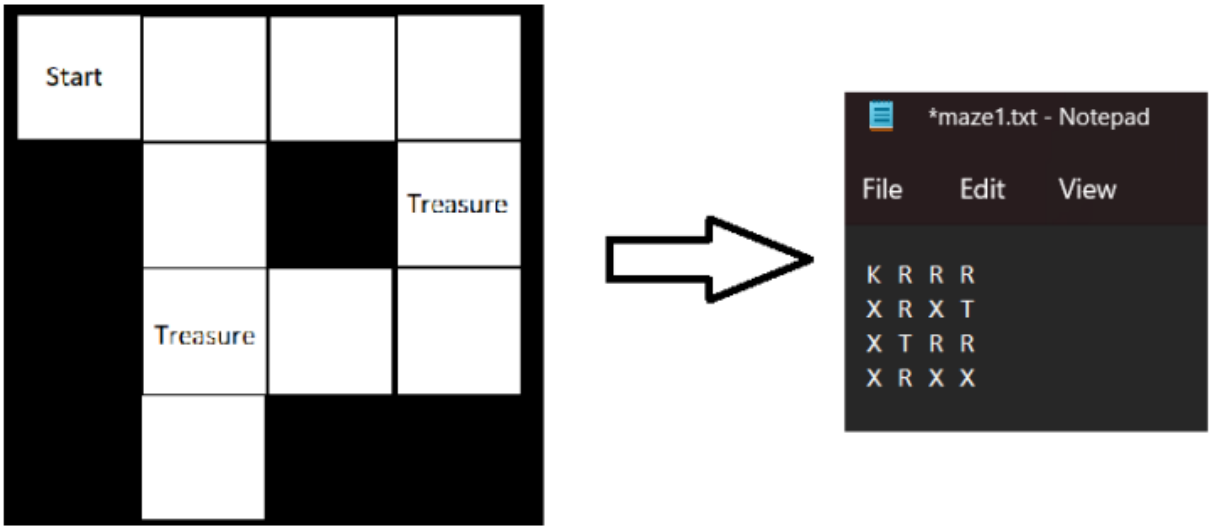
DESKRIPSI TUGAS

1 Deskripsi Tugas

Dalam tugas besar ini, Anda akan diminta untuk membangun sebuah aplikasi dengan GUI sederhana yang dapat mengimplementasikan BFS dan DFS untuk mendapatkan rute memperoleh seluruh treasure atau harta karun yang ada. Program dapat menerima dan membaca input sebuah file txt yang berisi maze yang akan ditemukan solusi rute mendapatkan treasure-nya. Untuk mempermudah, batasan dari input maze cukup berbentuk segi-empat dengan spesifikasi simbol sebagai berikut :

- K : Krusty Krab (Titik awal)
- T : Treasure
- R : Grid yang mungkin diakses / sebuah lintasan
- X : Grid halangan yang tidak dapat diakses

Contoh file input :

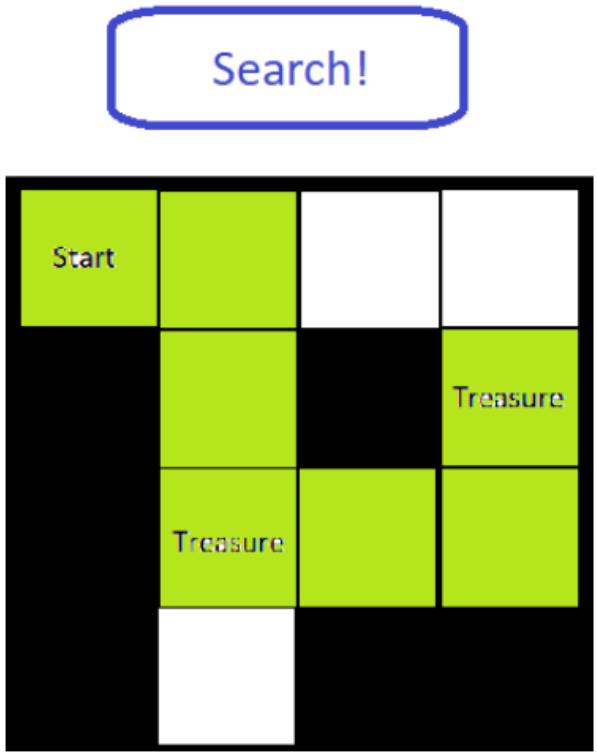


Gambar 1.1 Contoh maze dari input file

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.

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 :

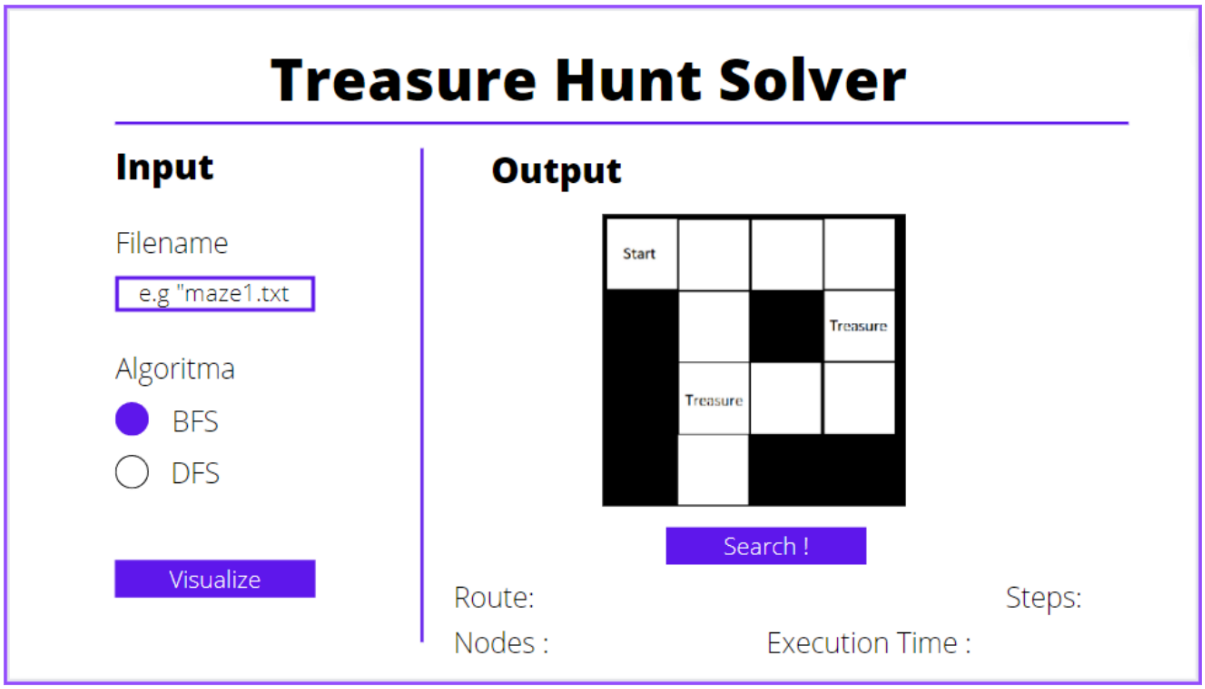


Gambar 1.2 Contoh output program untuk gambar 1

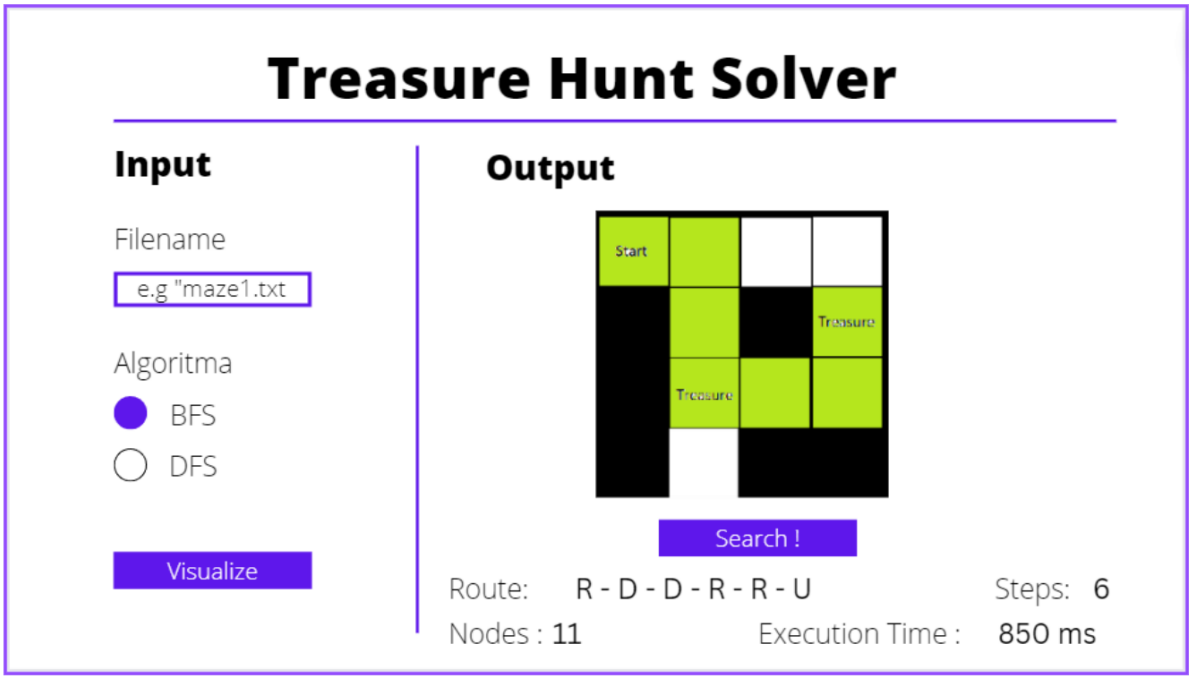
Setelah program melakukan pembacaan input, program akan memvisualisasikan gridnya terlebih dahulu tanpa pemberian rute solusi. Hal tersebut dilakukan agar pengguna dapat mengerjakan terlebih dahulu treasure hunt secara manual jika diinginkan. Kemudian, program menyediakan tombol solve untuk mengeksekusi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

2 Spesifikasi Program

Aplikasi yang akan dibangun dibuat berbasis GUI. Berikut ini adalah contoh tampilan dari aplikasi GUI yang akan dibangun



Gambar 1.2.1 Contoh tampilan program sebelum dicari solusinya



Gambar 1.2.2 Contoh 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 II

LANDASAN TEORI

1. Graph Traversal

Graf atau dalam bahasa Inggris disebut dengan "graph" merupakan salah satu jenis struktur data dalam matematika dan ilmu komputer yang digunakan untuk merepresentasikan hubungan antara objek-objek yang terhubung satu sama lain.

Graf memiliki beberapa komponen atau representasi :

1) Node/Vertex: merupakan titik dalam graf yang dapat direpresentasikan dengan suatu objek atau konsep.
2) Adjacency List: Daftar yang merepresentasikan semua node dan edge yang terhubung dengan node tersebut.

Algoritma traversal graf, yaitu algoritma untuk mengunjungi simpul dengan cara yang sistematik. Algoritma terbagi menjadi dua, yaitu Pencarian melebar (*breadth first search*/BFS) dan Pencarian mendalam (*depth first search*/DFS). BFS dan DFS merupakan salah satu pencarian solusi 'tanpa informasi' artinya kita tidak mengetahui apa-apa tentang letak node yang ingin dituju.

Dengan asumsi graf terhubung, graf dapat direpresentasikan sebagai persoalan dan traversal graf merupakan pencarian solusi. Representasi graf dalam proses pencarian memiliki 2 pendekatan

1) Graf statis: graf yang sudah terbentuk sebelum proses pencarian dilakukan (graf direpresentasikan sebagai struktur data)
2) Graf dinamis: graf yang terbentuk saat proses pencarian dilakukan
   (graf tidak tersedia sebelum pencarian, graf dibangun selama pencarian solusi)

2. BFS

BFS atau Breadth-First Search adalah salah satu algoritma traversal pada graf yang digunakan untuk menelusuri atau mencari informasi mengenai setiap node dalam graf. Algoritma BFS mengunjungi node-node dalam graf secara terurut, mulai dari satu node awal dan menyebar ke semua node tetangga yang belum dikunjungi, kemudian melanjutkan proses pada node-node tersebut secara berurut. Implementasi BFS biasanya menggunakan antrian (queue)

Jika merepresentasikan graf sebagai graf statis. Berikut adalah langkah-langkah algoritma BFS:

1) Mulai dari sebuah node awal dan tandai node tersebut sebagai telah dikunjungi.
2) Kunjungi semua simpul yang bertetangga dengan node awal, tambah ke antrian.
3) Kunjungi node yang bertetangga dengan simpul dalam antrian (setelah node dikeluarkan dari antrian) dan belum dikunjungi, kemudian tandai sebagai telah dikunjungi dan tambahkan ke dalam antrian.

4) Ulangi proses ini untuk setiap node yang telah diambil dari antrian atau sampai node yang dikunjungi atau dicek adalah node tujuan..

Mengacu pada referensi (Breadth/Depth First Search (BFS/DFS) bagian 1) berikut merupakan struktur data BFS :

1) Matriks ketetanggaan A = [aij] yang berukuran nxn,
   aij= 1, jika simpul i dan simpul j bertetangga,
   aij= 0, jika simpul i dan simpul j tidak bertetangga.
2) Antrian q untuk menyimpan simpul yang telah dikunjungi.
3) Tabel Boolean, diberi nama "dikunjungi"
   dikunjungi : array[1..n] of boolean
   dikunjungi[i] = true jika simpul i sudah dikunjungi
   dikunjungi[i] = false jika simpul i belum dikunjungi

Berikut merupakan 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
  { AntrianKosong(q) }
```

Gambar 2.2.1 Algoritma BFS

3. DFS

DFS atau Depth-First Search adalah salah satu algoritma traversal pada graf yang digunakan untuk menelusuri atau mencari informasi mengenai setiap node dalam graf. Algoritma DFS mengunjungi node-node dalam graf secara rekursif, mulai dari satu node awal dan bergerak secara vertikal ke bawah sejauh mungkin sebelum kembali ke node sebelumnya dan melanjutkan proses pada cabang yang belum dijelajahi. Implementasi DFS biasanya menggunakan stack atau rekursif.

Jika merepresentasikan graf sebagai graf statis. berikut adalah langkah-langkah algoritma DFS:

1) Mulai dari sebuah node awal dan tandai node tersebut sebagai telah dikunjungi.
2) Kunjungi node (misal node w) yang terhubung dengan node awal, kemudian tandai sebagai telah dikunjungi.
3) Ulangi proses mulai dari node w.
4) Ketika mencapai suatu simpul sedemikian sehingga semua simpul yang bertetangga dengannya telah dikunjungi. Lakukan *backtrack* ke simpul terakhir yang dikunjungi sebelumnya dan mempunyai tetangga yang belum dikunjungi
5) Pencarian berakhir bila tidak ada lagi simpul yang belum dikunjungi yang dapat dicapai dari simpul yang telah dikunjungi atau ketika menemukan node yang dituju.

Berikut merupakan algoritma DFS (secara rekursif):

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

Masukan: v adalah simpul awal kunjungan
Keluaran: semua simpulyang 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 2.3.1 Algoritma DFS

4. C# Dekstop Aplication Development

C# Desktop Application Development adalah sebuah kakas pegembang desktop application yang mampu beroperasi tanpa menggunakan internet. Pada umumnya,

pengembangan desktop application menggunakan bahasa pemrograman C++, C#, ataupun Java. Pengembangan desktop application, terlebih dalam hal user interface (UI), dipermudah dengan menggunakan WinForms/WPF dan bantuan integrated development environment (IDE) Visual Studio. WPF digunakan untuk pengembangan desktop application berbasis Windows dengan menggunakan bahasa native-nya, yaitu bahasa pemrograman C#

Windows Forms (WinForms): Windows Forms adalah teknologi GUI (Graphical User Interface) untuk aplikasi desktop yang digunakan dalam C# Desktop Application Development. WinForms memungkinkan pengguna untuk membuat tampilan aplikasi yang interaktif dengan berbagai elemen seperti tombol, kotak teks, dan jendela dialog.

BAB III

APLIKASI ALGORITMA BFS DAN DFS

1. Pemecahan Masalah

Kasus blok-blok pada map yang melambangkan X untuk grid yang tidak bisa dilewati, dan yang bisa dilewati terdiri dari R untuk jalan biasa, K untuk jalan awal, dan T untuk treasure. Kami memecah persoalan menjadi masukan map yang berisi simbol-simbol karakter (X, R, K, dan T) menjadi matriks karakter. Masukan dari txt ini ditangani oleh kelas InputHandler. Kelas InputHandler membaca seluruh karakter dalam file dan dimasukan ke dalam array of string. Kemudian array of string ini dicek apakah karakter yang terdapat di dalamnya valid atau tidak. Apabila valid maka akan diproses oleh kelas map. Kelas map akan menghasilkan matriks karakter. Matriks karakter ini kemudian akan digunakan dalam kalkulasi BFS dan DFS.

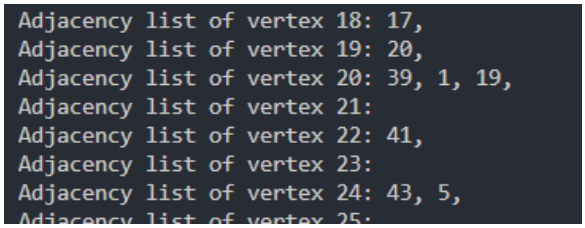
Selanjutnya kami memecah persoalan algoritma dalam sebuah kelas yang berisi dari DFS, BFS. Kami membuat representasi dari blok-blok dalam matriks karakter menjadi sebuah struct bernama mapElmt dengan 3 atribut yaitu index, row, dan col. Index bermakna dia ada di urutan ke berapa dari matriks karakter (pojok kiri atas itu 0 dan bertambah terus sesuai dengan urutan elemen pada matriks). Row bermakna baris pada matriks dan col bermakna kolom pada matriks.

2. InputHandler dan Map

Kelas InputHandler ini dibuat untuk menangani dan membaca file masukan txt. Dalam kelas ini terdapat 2 atribut, yaitu string[] inputFileData dan bool isValid. Terdapat salah satu method utama dalam kelas ini, yaitu method readFile dengan parameter filepath. Method ini akan membaca file txt yang dimasukan dan membaca seluruh karakter yang ada di dalam file tersebut. Setelah dibaca, seluruh karakter akan dijadikan string (per line) dan dimasukkan ke array inputFileData. Kemudian inputFileData ini akan dicek apakah seluruh elemennya valid atau tidak. inputFileData yang valid hanya terdiri dari karakter ‘K’, ‘R’, ‘X’, dan ‘T’. Apabila valid, inputFileData akan diproses oleh kelas Map dan akan diubah menjadi matriks karakter. Matriks karakter ini kemudian akan diproses oleh Solver (BFS dan DFS).

3. Solver

Kelas solver merupakan kelas untuk implementasi algortima BFS, DFS, dan kedua algoritma tersebut dalam versi TSP. Terdapat beberapa atribut utama dalam kelas Solver yaitu jumlah vertices dalam map, array of List dari mapElmt yang berfungsi sebagai list ketetanggan dari mapElmt misalnya pada index pertama terdapat sebuah mapElmt representasi ‘K’ pada map. MapElmt tersebut bertetanga dengan indeks 2, dan 8 pada list sehingga adjacency list vertex 1 : 2 8.



Gambar 3.3.1 Adjacency list

Atribut-atribut lain pada kelas ini adalah n_treasure sebagai jumlah treasure pada map, n_visited sebagai jumlah mapElmt yang sudah dikunjungi, List tressureIndex sebagai List of index dari tresure pada map. Kemudian kami menggunakan dua buah atribut untuk menyimpan data hasil dari algoritma yang kita buat. Pertama, List of tuple integer scannedPath sebagai List berelemen baris dan kolom pada map sebagai hasil dari *scanning* algoritma baik BFS maupun DFS. Kedua, list of tuple integer pathToTreasure sebagai list hasil akhir setelah *scanning* dari algoritma. Berikut merupakan method algortima dari kelas Solver:

a. BFS

Terdapat dua method dengan satu method utama yang digunakan untuk memecahkan persoalan dengan BFS. Method utama (Scanning) bernama BFSAlgorithmStrategies() memiliki konsep seperti pada landasan teori dengan tambahan:

1. Inisiasi queue (dengan linked List)
2. Kunjungi mapElmt awal (K)
3. Ketika queue belum kosong atau treasure belum semua ditemukan : kunjungi setiap node yang bertetangga dan belum dikunjungi dengan prioritas D-L-U-R
4. Masukan kedalam antrian untuk setiap list yang ditemukan
5. Perbedaannya terletak disini. Jika menemukan treasure, mapElmt pada treasure tersebut dimasukan ke ‘awal’ dari queue (seperti menyerobot) untuk memprioritaskan BFS dari treasure tersebut sehingga dapat membuat jalur dari K menuju semua treasure

Selain itu, terdapat BFS method pembantu dengan logika terbalik dari dari BFS utama. Logika terbalik dari BFS utama adalah, method ini mencari jalan pada mapElmt yang sudah dikunjungi. Method ini dijalankan setiap menemukan treasure dengan konsep menyerupai BBFS (terdapat 2 titik yang diketahui untuk disambungkan) tetapi hanya melakukan BFS dari mapElmt awal saja. Method ini membantu menemukan jalur dari satu titik menuju titik lain.

Oleh karena itu, saat menjalankan BFS terdapat 2 BFS yang berjalan. BFS yang bertanggungjawab untuk *scanning* dan BFS yang bertanggung jwab untuk ‘pencarian jalan’

b. DFS

Terdapat beberapa method untuk algoritma ini, pertama DFS versi normal dibuat dengan cara yang sudah diajarkan pada perkuliahan yaitu menggunakan stack.

1. Masukkan start index ke dalam stack

2. Selama stack belum kosong dan jumlah harta karun yang harus diambil belum nol iterasi poin selanjutnya dari poin 3 sampai 5
3. Pop stack dan masukan ke dalam current (mapElmt)
4. Iterasi mapElmt i pada List adjacency dari indeks current pada poin kelima
5. Jika indeks i belum dikunjungi, maka tandai true untuk indeks i tersebut, push i ke dalam stack, jika indeks i adalah 'T' maka kurangi jumlah harta karun yang harus diambil

Selanjutnya, kami membuat fungsi BFS untuk versi dapat melakukan backtrack (untuk keperluan GUI agar terlihat jelas *step by step*). Terdapat dua buah fungsi yaitu dfsbacktrack untuk menginisiasi elemen yang akan diberikan ke fungsi dfsBack (rekursif). Pada dfsbacktrack, inisiasi variabel pathUsed sebagai List of List dari mapElmt, sebuah stack mapElmt, dan dummy variabel pred sebagai (*predecesssor* dari current mapElmt) pendahulu dari start index. Kemudian fungsi ini memanggil dfsBack yang memiliki algoritma sebagai berikut:

1. Cek apakah current index sudah sama dengan dummy variabel (untuk nilainya yaitu -1)
2. Jika sudah maka terminasi fungsi ini, jika tidak sama maka
3. Cek apakah current itu 'T' dan belum dikunjungi, jika benar lakukan pengurangan terhadap n_treasure (jumlah harta yang harus diambil) dan jika n_treasure sudah nol maka tambahkan ke dalam scannedPath
4. Cek apakah jumlah yang sudah dikunjungi (n_visited) sudah sama dengan jumlah vertex atau apakah n_treasure sudah habis atau nol maka terminasi (return)
5. Tandai current index sudah dikunjungi dengan true
6. Tambahkan pathUsed dengan List of mapElmt dengan parameter pred dan current
7. Tambahkan scannedPath dengan bariss dan kolom dari current
8. Iterasi kepada node-node yang belum dikunjungi di dalam List adjacency pada indeks current (tetangga-tetangga dari current), **untuk setiap tetangga** panggil kembali fungsi ini (rekursif) jika **belum dikunjungi** dan **n_treasure masih belum habis atau nol**, dengan tetangga sebagai parameter current dan current sebagai parameter pred (karena x memiliki predecessor current)
9. Backtrack ke node kunjungan terakhir dengan mengiterasi dari nol sampai jumlah pathUsed, jika index dari pathUsed[y][1] adalah index current maka panggil fungsi ini (rekursif) dengan parameter current adalah pathUsed[y][1] dan parameter pred adalah current

c. TSP

1) BFS

TSP dengan BFS konsepnya menyerupai BFS ketika mencari jalur, sehingga caranya sama seperti poin a (BFS), tetapi ada perbedaan yaitu ketika menemukan treasure terakhir. Akan dijalankan BFS pembantu untuk mencari jalan dari titik treasure akhir ke titik start.

2) DFS

Terdapat juga penerapan dari TSP untuk DFS. Secara garis besar, dari scannedPath yang telah dihasilkan oleh dfsbacktrack, method untuk tsp ini mencari path yang bersesuaian dengan alur path hasil scanning dfs kemudian diiterasi dari setiap treasure ke treasure dan start index ('K'). Setiap iterasi antar dua parameter

tersebut (tresure satu ke yang lain), lakukan pengecekan adjacent dan kasus ketika indeks tersebut menemukan jalan buntu (terdapat dua buah indeks bernilai sama karena bactrack) dan jika memang jalan buntu maka lewati jalan tersebut. Setelah selesai dan menemukan sekuens path, gabungkan dengan hasil yang dibalik (reverse) sehingga dari start bisa kembali lagi ke start.

4. GUI

Kelas ini dibuat atas dasar spesifikasi dari tugas besar itu sendiri. Kami ditugaskan untuk membuat suatu program dengan basis antarmuka berupa GUI. Dalam GUI ini terdapat banyak sekali atribut berupa komponen-komponen yang dapat digunakan atau berinteraksi dengan user, antara lain button, text box, radio button, dll. Dalam GUI ini juga terdapat banyak method yang dibuat untuk menyatukan seluruh algoritma yang sudah dibuat menjadi satu kesatuan program.

BAB IV

Analisis Pemecahan Masalah

1. Implementasi Program

a. BFS

Procedure BFSAlgorithmStrategies()
{initial state semua atribute telah diinisiasi}
{final state : atribut hasil terisi denganhasil scanner dan hasil jalur}
Inisiasi BFSqueue(q)
Kunjungi(this.start)

```
While (BFSqueue.Count != 0 and n_treasure != 0) do
        w = deque(q)
        Foreach tetangga w
                kunjungi(this.start)
                If (tetangga(w) = treasure)
                        GetPathBFSAlgorithmStrategies(titik1, titik2) // cari jalur dati
```

b. dfsbacktrack

Procedure dfsbacktrack **( )**
{initial state : atribut hasil (scannedPath) dan dari kelas Solver di reset atau dalam keadaan awal **}**
{final state : atribut hasil sudah terisi dengan hasil algoritma **}**

Insiasi List of List mapElmt
mapElmt pred ← new mapElmt(-1,-1,-1)
DFSBack(start, pred, visited, pathUsed)

c. dfsBack

Procedure dfsBack **(input/output** mapElmt **:** current, **input/output** mapElmt **:** pred, **input/output** mapElmt **:** current, **input/output** List of bool **:** visited, List of List mapElmt : PathUsed **)**
{initial state : atribut hasil (scannedPath) dan dari kelas Solver di reset atau dalam keadaan awal **}**
{final state : atribut hasil sudah terisi dengan hasil algoritma **}**

```
if (current.index != -1)
{
   // check if get treasure
   if (map.getElement(current.row, current.col) == 'T' and  !visited[current.index])
   {
      N_treasure ← N_treasure - 1
      if (n_treasure == 0){
```

```
            // add to scannedPath a tuple of u.row and u.col
            this.scannedPath.Add((current.row, current.col))
        }
        // print jumlah scannedPath
    }

    // If all the node is visited return
    if (this.n_visited == v || n_treasure == 0){
        terminasi
    }

    // Mark not visited node as visited
    visited[current.index] = true
    this.n_visited++
    // Track the current edge
    pathUsed.Add(new List<mapElmt>() {pred, current})
    // add to scannedPath a tuple of current.row and current.col
    this.scannedPath.Add((current.row, current.col))

    // Check for not visited node and proceed with it.
    foreach (mapElmt x in adj[current.index])
    {
        // call the DFs function if not visited
        if (!visited[x.index] && n_treasure != 0)
        {
            dFSBack(x, current, visited, pathUsed)
        }
    }

    // Backtrack through the last visited nodes
    for (int y = 0; y < pathUsed.Count; y++)
    {
        // jika pathUsed[y][1] sama dengan current maka panggil lagi dFSBack
        // dengan pathUsed[y][0] sebagai current dan pathUsed[y][1] sebagai pred
        if (pathUsed[y][1].index == current.index)
        {
            dFSBack(pathUsed[y][0], current, visited, pathUsed)
        }
    }
}
```

d. getPathToTreasureDFS

```
getPathToTreasureDFS(int i1, int i2, List<(int,int)> subPath){
    // find path from last to first index of scannedPath with the skip index
    subPath.Clear();
```

```
        int idx = i2;
        var t = scannedPath[i2];
        subPath.Add(t);
                    Console.WriteLine("scannedPath[i2]  " + scannedPath[i2].Item1 + " " +
scannedPath[i2].Item2);
        // idx--;
        while (subPath.Last() != scannedPath[i1]){
           // int count = 0;
           for (int i = i1; i < idx; i++){
              t = scannedPath[i];
              // change t to mapElmt
              // cek apakah t adjacent dengan subPath.Last()
                  if ( (isAdjacent(t, subPath.Last()) && scannedPath[i+1]==subPath.Last() ) || t ==
scannedPath[idx-1]){
                    // add to subPath
                    subPath.Add(t);
                    // decrement idx
                    idx--;
                    break;
              }
           }
        }
        // reverse subPath
        subPath.Reverse();
```

2. Struktur data

Struktur data yang kami gunakan dapat dibagi menjadi 3 bagian besar, InputHandler dan Map, Algoritma (BFS, DFS, TSP), dan GUI.

1. InputHandler dan Map : Berisikan atribut inputFileData dengan tipe data array of string, isValid dengan tipe data boolean, dan mapData dengan tipe data matriks karakter dengan tambahan atribut pendukung berupa mapWidth, mapHeight, mapSize. Method yang terdapat pada InputHandler dan Map secara garis besar menangani dan melakukan pemrosesan terhadap file masukan.
2. Algoritma : Pada kelas algoritma terdapat beberapa atribut sebagai berikut.

```
private int v
private List<mapElmt>[] adj
private mapElmt start
private List<bool> visited
private MyMap map
private int n_treasure
private int n_visited
private List<(int, int)> scannedPath
private List<(int, int)> pathToTreasure
private List<char> step
public List<int> treasureIndex
```

3. GUI : Berisikan atribut berupa komponen-komponen yang dapat digunakan dalam GUI. Komponen-komponen tersebut antara lain terdapat background image, icon, dua buah button, dua buah radio button, satu buah check box, dua buah text box, tiga buah label, dan satu buah datagridview. Method-method yang terdapat pada kelas ini juga merupakan pemrosesan hasil algoritma BFS, DFS, dan TSP agar dapat menjadi satu kesatuan program.

3. Tata cara penggunaan program

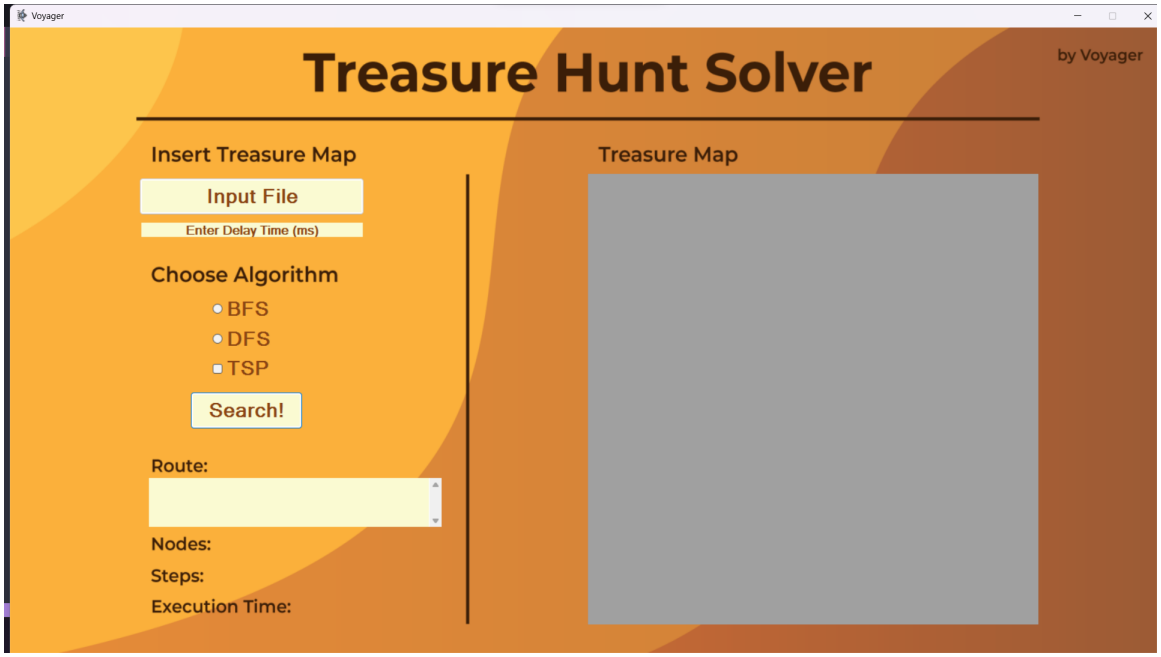
Program dibuat dalam bahasa C# dengan memanfaatkan Visual Studio .NET sehingga sebelum menjalankan program lakukan instalasi .NET untuk menjalankan program. Kelompok kami menggunakan *Third-party Tools* yaitu chocolatey untuk melakukan instalasi command 'make' dalam CLI Windows. Jika memiliki hal yang sama cara menjalankan program cukup :

1) Pastikan path cmd/powershell berapa pada Tubes2_Voyager (jika belum di rename)
2) Ketikan 'make run'.

Akan tetapi, jika ingin menjalankan program dengan .NET command :

1) Pastikan path cmd/powershell berada pada Tubes2_Voyager/src
2) Ketikan dotnet run

Setelah itu, program menampilkan :



Gambar 4.3.1 Tampilan awal program Treasure Hunt Solver

Berikut penjelasan komponen pada tampilan program sekaligus langkah-langkah untuk menjalankan program

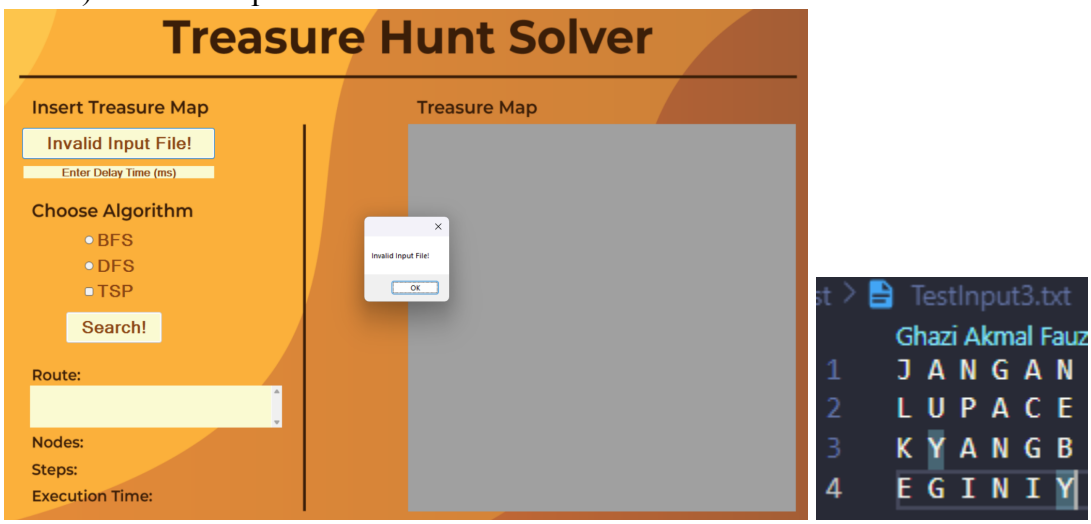
1) Kotak "Input File" untuk memilih file yang yang akan menjadi maze.Klik lalu pilih file yang akan menjadi 'maze'

2) Kotak “Enter Delay Time (ms)” untuk memasukan delay time dalam menampilkan hasil pencarian maze.
3) Radio tombol (BFS/DFS) untuk memilih metode algoritma
4) Kotak centang (TSP) untuk memilih apakah ingin melakukan TSP dengan metode Algoritma yang dipilih sebelumnya
5) Kotak “Search!” untuk memulai program

4. Hasil Pengujian

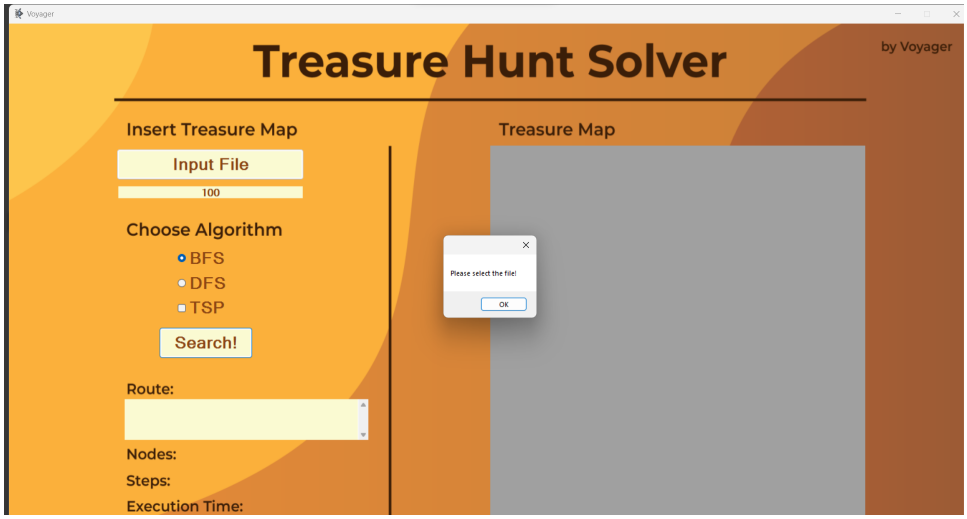
Berikut merupakan hasil pengujian

1) Case 1 : Input File Tidak Valid



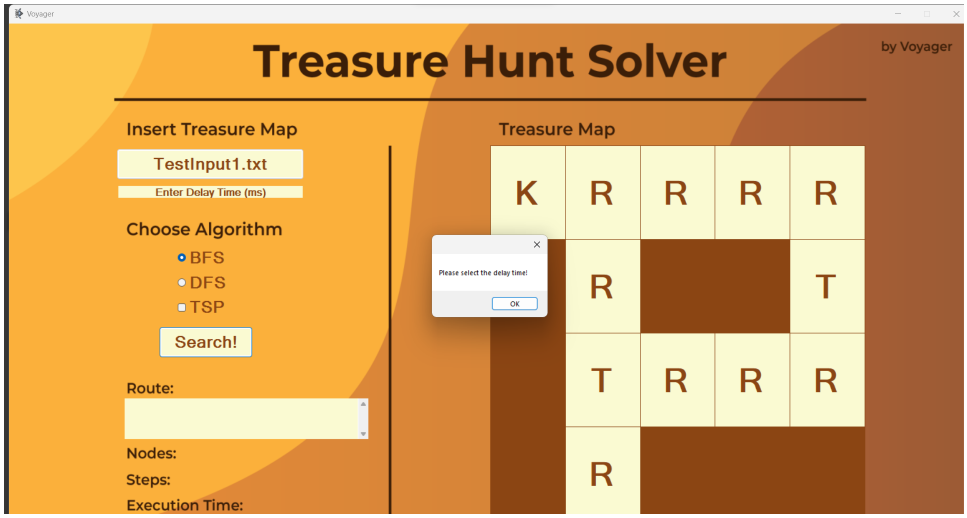
Gambar 4.4.1 Contoh input file yang tidak valid

2) Case 2 : Treasure Map belum dimasukkan



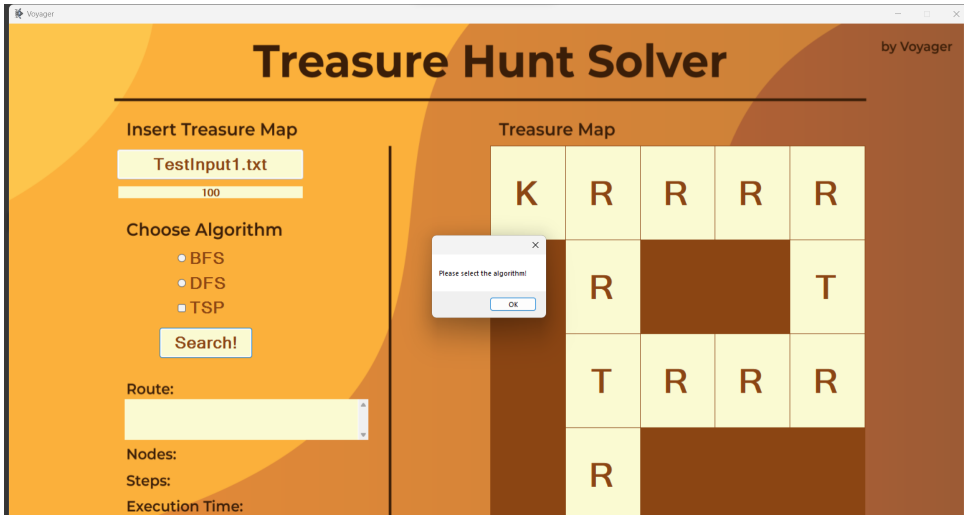
Gambar 4.4.2 Contoh Treasure Map belum dimasukkan

3) Case 3 : Delay Time belum dimasukkan



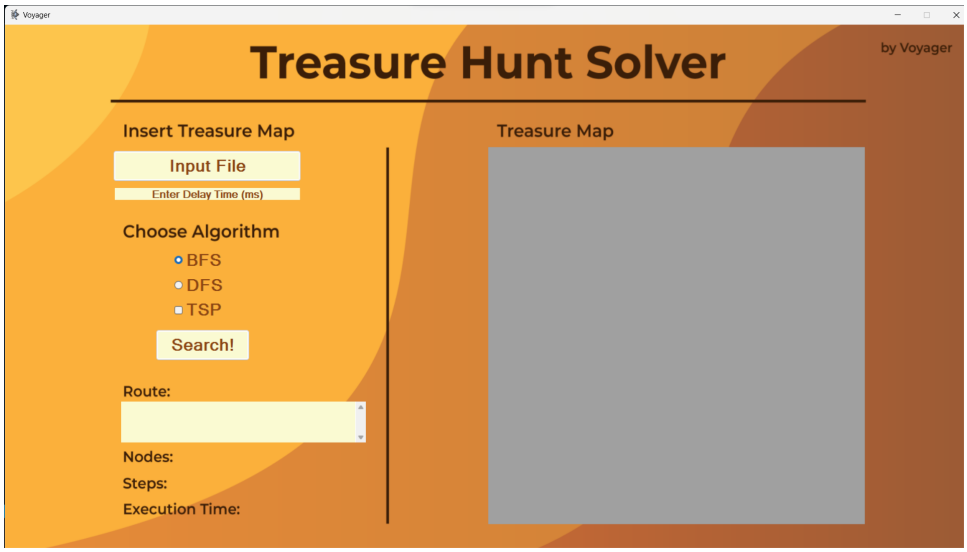
Gambar 4.4.3 Contoh Delay Time belum dimasukkan

4) Case 4 : Algoritma belum dipilih



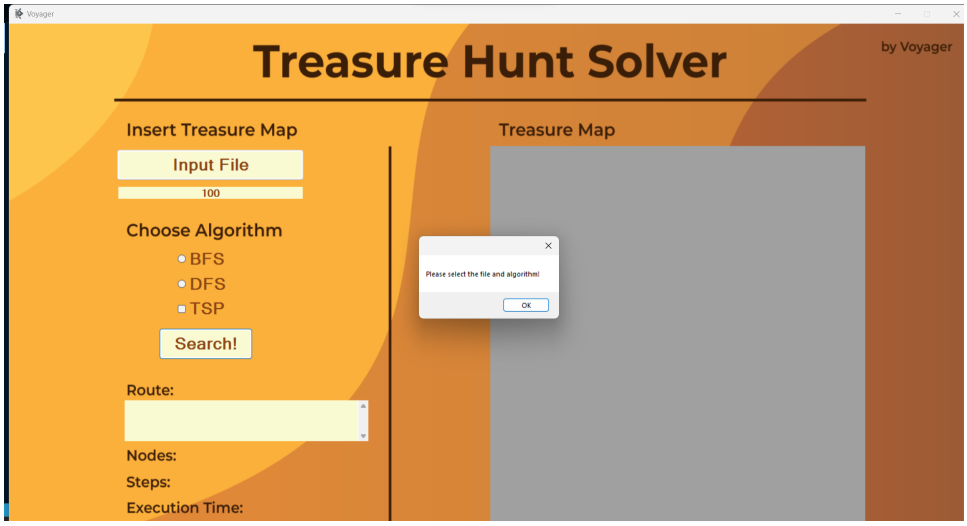
Gambar 4.4.4 Contoh Algoritma belum dipilih

5) Case 5 : Treasure Map dan Delay Time belum dimasukkan



Gambar 4.4.5 Contoh Treasure Map dan Delay Time belum dipilih

6) Case 6 : Treasure Map dan Algoritma belum dipilih

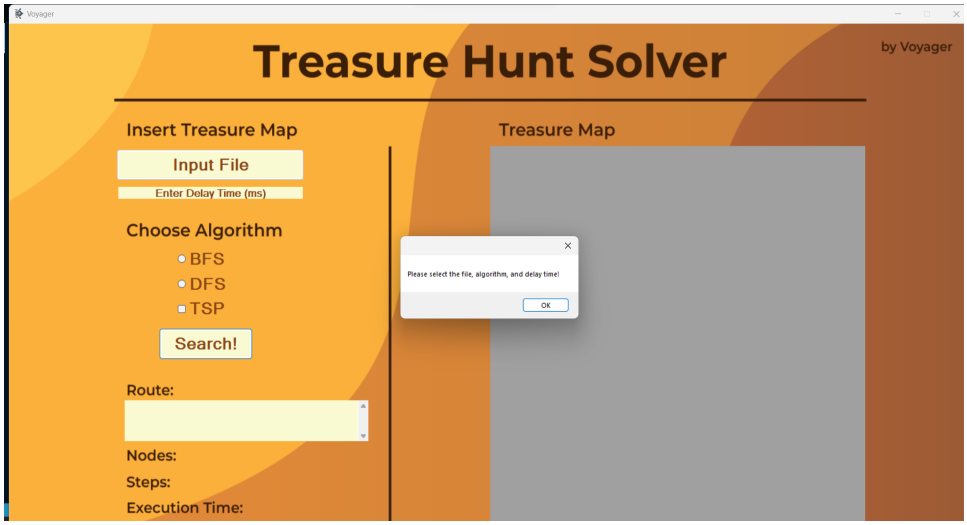


Gambar 4.4.6 Contoh Treasure Map dan Algoritma belum dipilih

7) Case 7 : Delay Time dan Algoritma belum dipilih

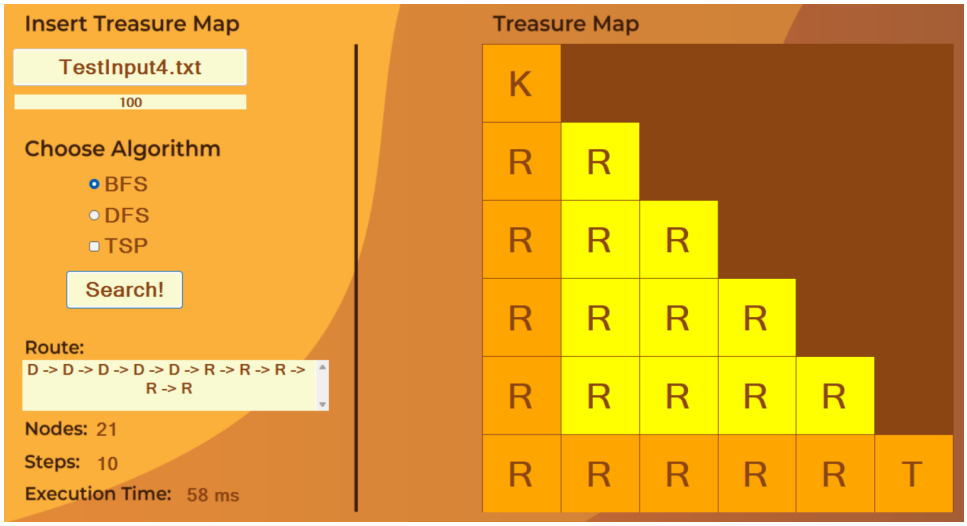
Gambar 4.4.7 Contoh Delay Time dan Algoritma belum dipilih

8) Case 8 : Seluruh input belum dimasukkan



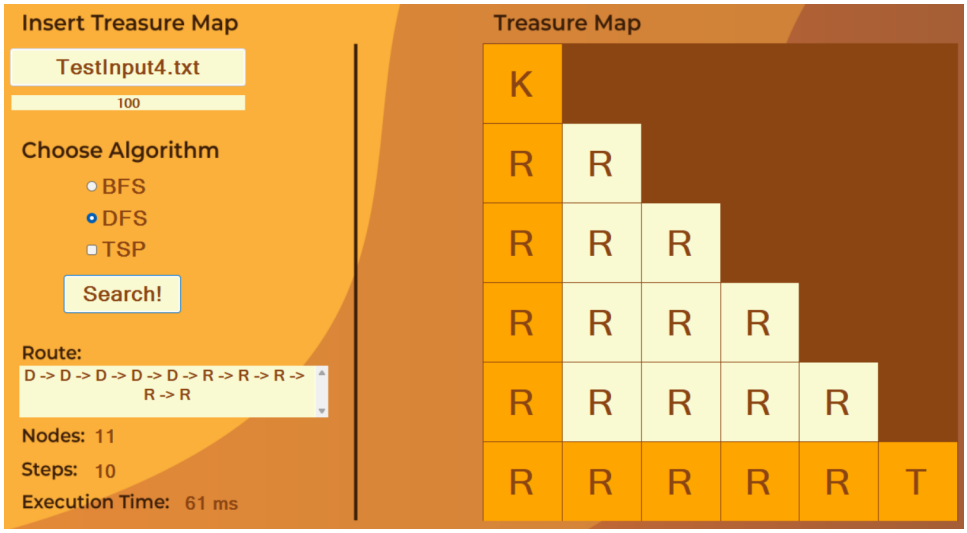
Gambar 4.4.8 Contoh seluruh input belum dimasukkan

9) Case 9 : BFS



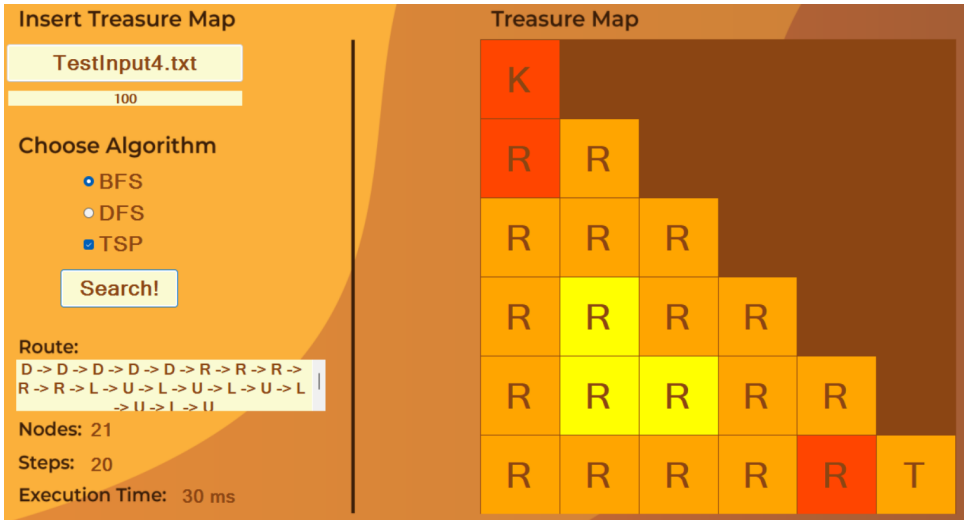
Gambar 4.4.9 Contoh kasus BFS

10) Case 10 : DFS



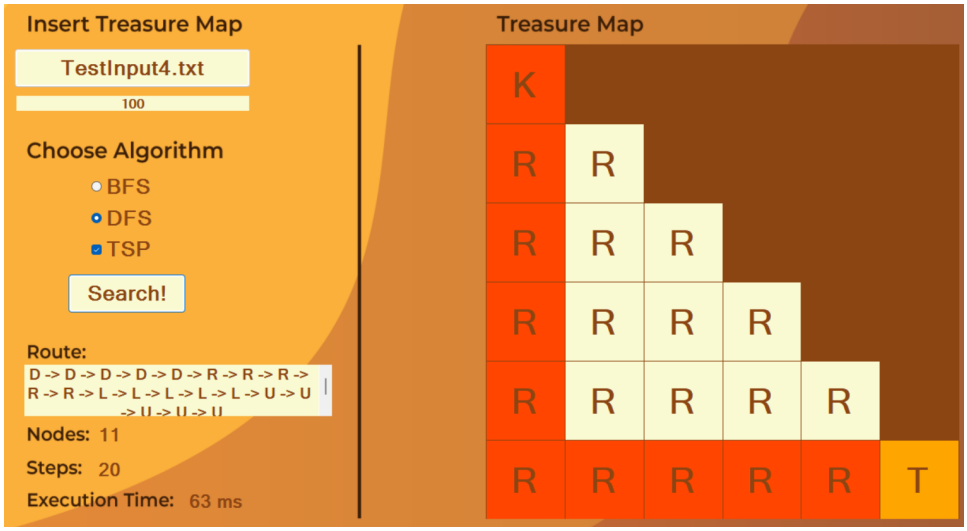
Gambar 4.4.10 Contoh Kasus DFS

11) Case 11 : TSP dengan BFS



Gambar 4.4.11 Contoh kasus TSP dengan BFS

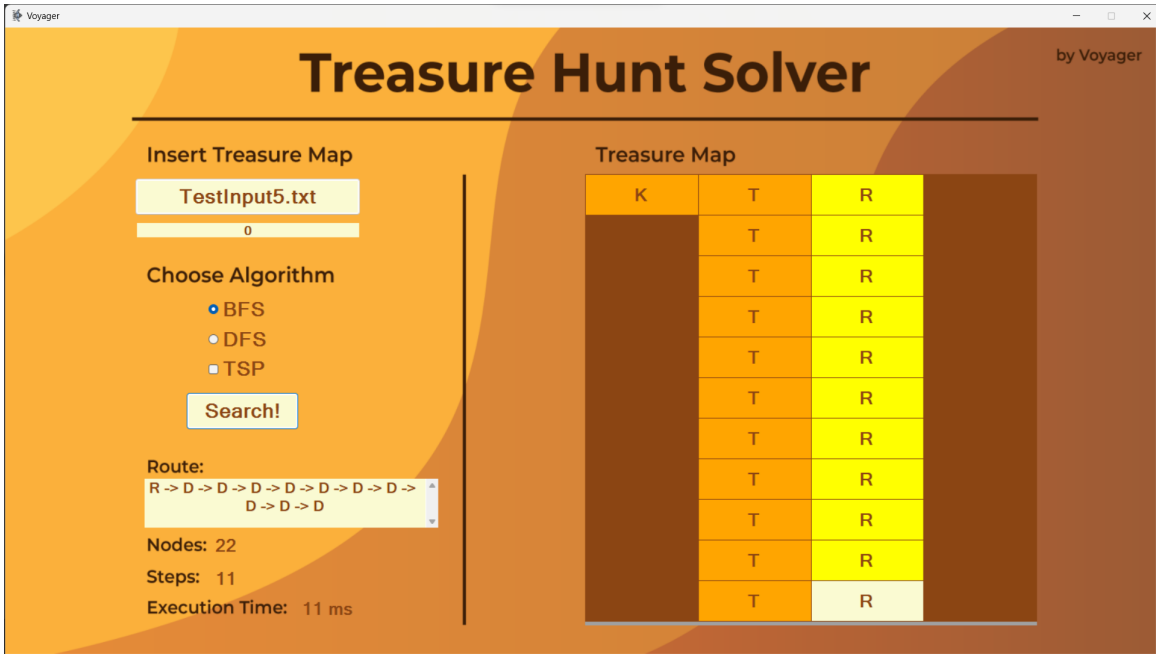
12) Case 12 : TSP dengan DFS

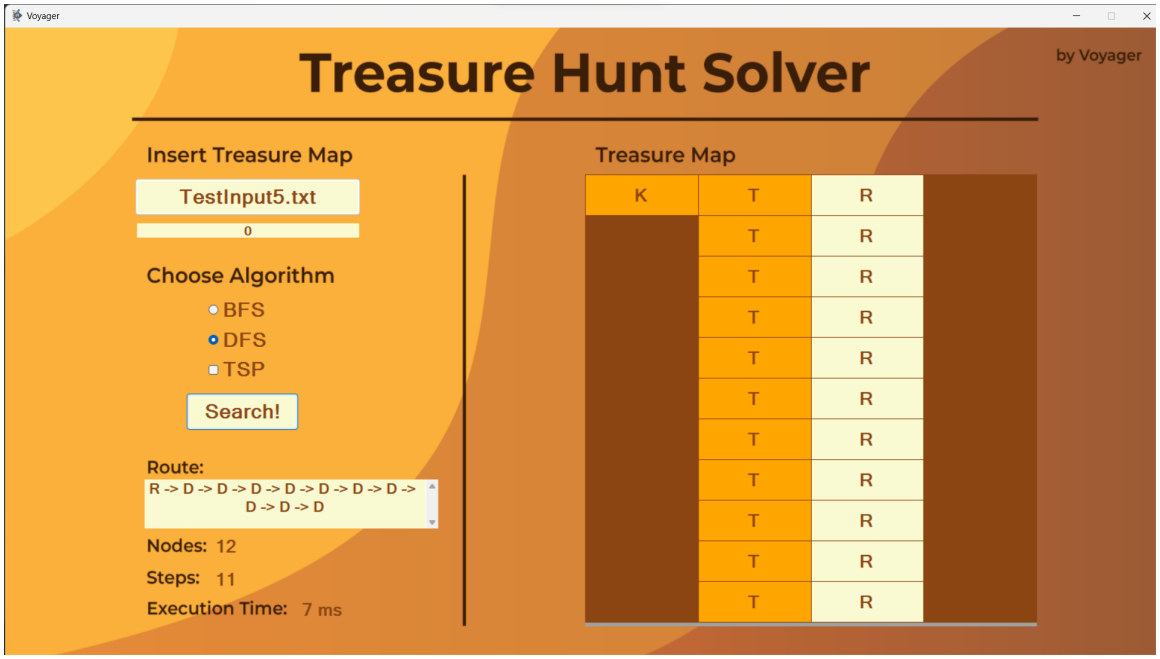


Gambar 4.4.12 TSP dengan DFS

5. Analisis Desain Solusi Algoritma BFS dan DFS

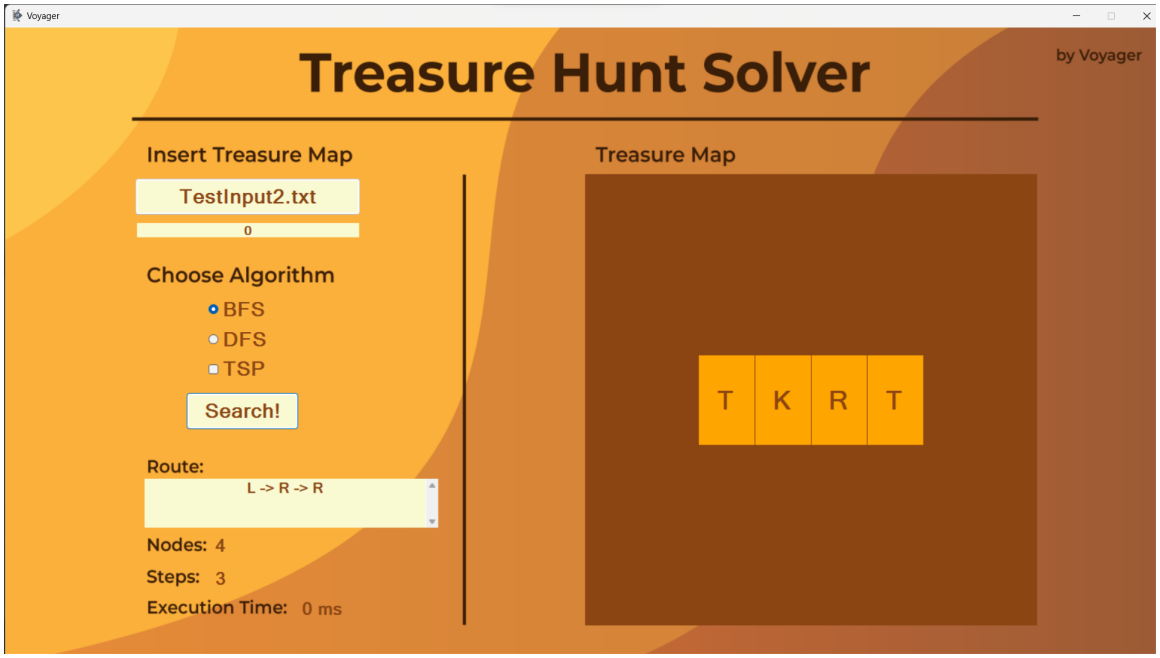
- Percobaan 1 (Algoritma DFS lebih baik daripada Algoritma BFS)

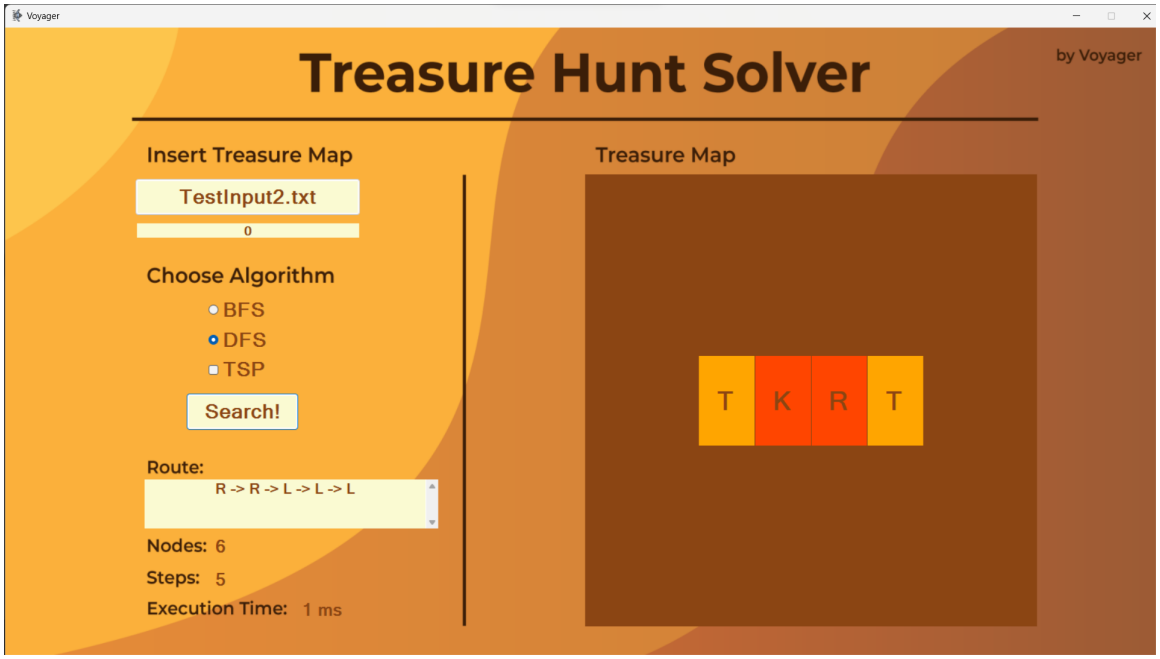




Pada percobaan dengan TestInput5.txt. Algoritma DFS memberikan hasil yang lebih baik daripada Algoritma BFS. Hal ini juga didukung dengan urutan Treasure yang berurutan dalam satu *branch*. Algoritma DFS memperoleh sebanyak 11 steps dengan 12 nodes dan waktu eksekusi 7ms. Sedangkan Algoritma BFS memperoleh sebanyak 11 steps dengan 22 nodes dan waktu eksekusi 11ms.

- Percobaan 2 (Algoritma BFS lebih baik daripada Algoritma DFS)





Pada percobaan dengan TestInput2.txt. Algoritma BFS memberikan hasil yang lebih baik daripada Algoritma DFS. Hal ini juga didukung dengan urutan Treasure yang tersebar di dua branch. Algoritma BFS memperoleh sebanyak 3 steps dengan 4 nodes dan waktu eksekusi sekitar 0ms. Sedangkan Algoritma DFS memperoleh sebanyak 5 steps dengan 6 nodes dan waktu eksekusi 1ms.

BAB V

KESIMPULAN DAN SARAN

1. Kesimpulan

BFS adalah algoritma yang digunakan untuk melintasi atau mencari dalam struktur data grafik atau pohon. Algoritme dimulai dari root node atau source node dan menjelajahi semua node pada kedalaman saat ini sebelum berpindah ke node pada level kedalaman berikutnya. BFS diimplementasikan menggunakan struktur data antrian, dimana node dimasukkan ke dalam antrian dan kemudian diproses dalam urutan FIFO (first in, first out).

DFS, di sisi lain, adalah algoritma yang digunakan untuk melintasi atau mencari dalam struktur data grafik atau pohon. Algoritme dimulai dari root node atau source node dan menjelajah sejauh mungkin sepanjang setiap cabang sebelum melakukan backtracking. DFS dapat diimplementasikan menggunakan stack atau rekursi. Saat menggunakan tumpukan, node didorong ke tumpukan dan kemudian diproses dalam urutan LIFO (masuk terakhir, keluar pertama). Saat menggunakan rekursi, fungsi memanggil dirinya sendiri untuk setiap node anak hingga mencapai kasus dasar.

Baik BFS maupun DFS memiliki kelebihan dan kekurangan masing-masing tergantung pada situasinya. BFS berguna saat kita ingin mencari jalur terpendek pada graf atau pohon yang tidak berbobot, sedangkan DFS berguna saat kita ingin mencari node tertentu atau mencari jalur pada graf atau pohon berbobot.

2. Saran

Pada pengerjaan tugas besar ini, terdapat kendala waktu karena pengerjaan tugas besar ini dibarengi dengan adanya Ujian Tengah Semester. Agar waktu mengerjakan tugas cukup, berikut adalah hal yang perlu dilakukan:

1. Segera mencicil tugas besar.
2. Mempelajari materi UTS lebih awal sehingga masih ada waktu untuk mengerjakan tugas besar di waktu UTS.
3. Tidak menunda-nunda pengerjaan tugas.

3. Komentar dan Refleksi

Pada tugas besar kali ini sangat menuntut untuk memikirkan strategi DFS dan BFS terbaik yang dapat diimplementasikan ke dalam program. 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.

Referensi

GeeksforGeeks. (n.d.). Depth First Search or DFS for a Graph. Retrieved March 24, 2023, from https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

JavaTpoint. (n.d.). DFS Algorithm in Data Structure. Retrieved March 24, 2023, from https://www.javatpoint.com/dfs-algorithm-in-data-structure

Breadth/Depth First Search (BFS/DFS) bagian 1 https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf

Breadth/Depth First Search (BFS/DFS) bagian 2 https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf

Lampiran

Repository
https://github.com/NerbFox/Tubes2_Voyager

Link Youtube
https://youtu.be/jMTXlsCsaqU