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

Kelompok MencariMaknaSpek :

Go Dillon Audris	(13521062)
Michael Leon Putra Widhi	(13521108)
Kenneth Dave Bahana	(13521145)

Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung

2023

DAFTAR ISI

BAB I
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.

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.

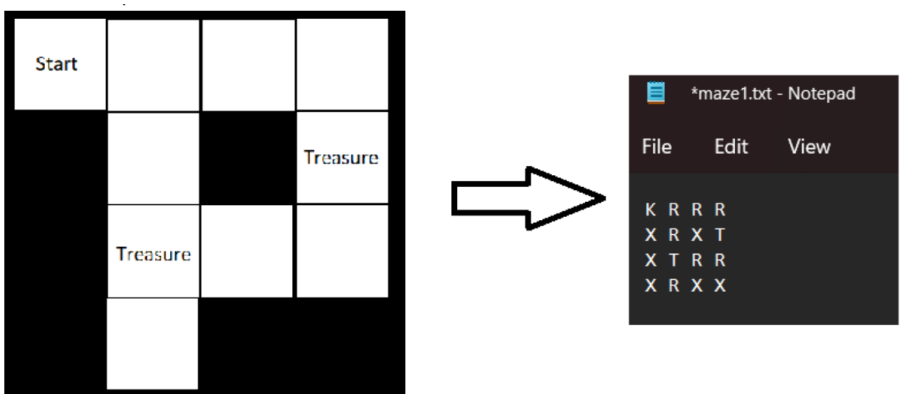
Dalam tugas besar ini, Anda akan diminta untuk membangun sebuah aplikasi dengan GUI sederhana yang dapat mengimplementasikan BFS dan DFS untuk mendapatkan rute memperoleh seluruh *treasure* atau harta karun yang ada. Program dapat menerima dan membaca input sebuah *file* .txt yang berisi *maze* yang 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

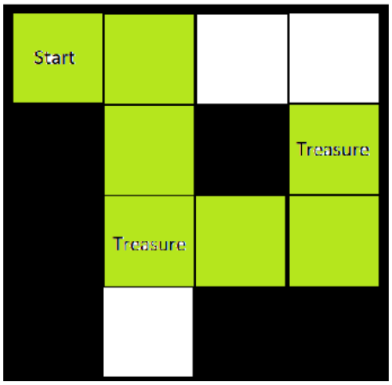
X : *Grid* halangan yang tidak dapat diakses



Gambar 1.1. *Input* ilustrasi *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.

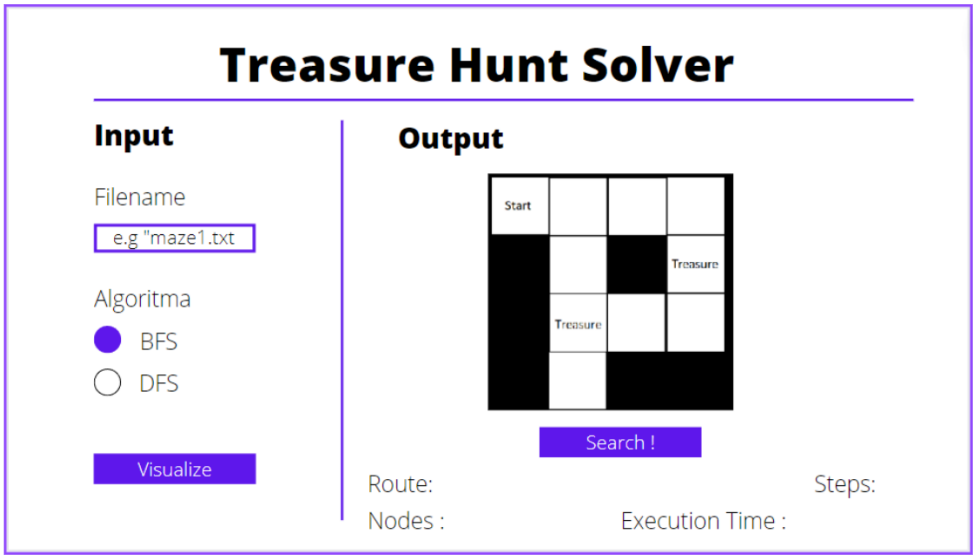
Daftar *input maze* akan dikemas dalam sebuah *folder* yang dinamakan *test* dan terkandung dalam *repository* program. *Folder* tersebut akan setara kedudukannya dengan *folder* src dan doc (struktur *folder repository* akan dijelaskan lebih lanjut di bagian bawah spesifikasi tubes). Cara *input maze*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



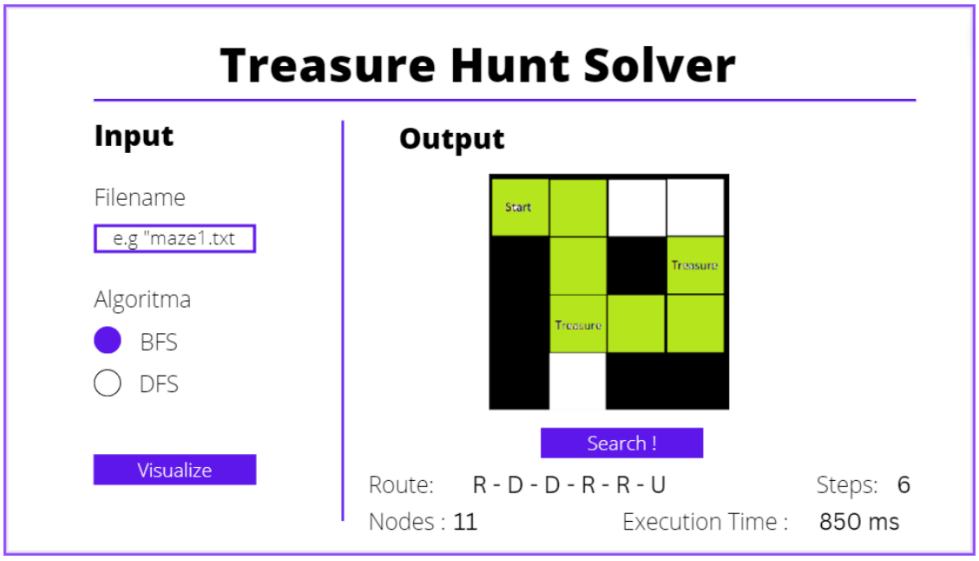
Gambar 1.2. Contoh *output* program untuk Gambar 1.1.

Setelah program melakukan pembacaan *input*, program akan memvisualisasikan *grid*-nya 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.

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



Gambar 1.3. Tampilan program sebelum dicari solusinya



Gambar 1.4. 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 *file*-nya.
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.

BAB II
LANDASAN TEORI

A. *Graph Traversal*

Graf digunakan untuk merepresentasikan objek-objek diskrit dan hubungan antara objek-objek tersebut, sehingga secara sederhana graf didefinisikan sebagai kumpulan titik yang dihubungkan oleh garis-garis/sisi. Sedangkan definisi matematis untuk graf adalah, pasangan terurut himpunan (V, E), dimana V merupakan himpunan beranggotakan titik-titik (*vertex*) dan E merupakan himpunan beranggotakan sisi-sisi (*edges*). Algoritma *graph traversal* atau traversal di dalam graf adalah algoritma yang memiliki mekanisme mengunjungi simpul-simpul dengan cara yang sistematik, dengan asumsi semua graf terhubung. Jika graf adalah sebuah representasi dari persoalan, maka *graph traversal* dapat diibaratkan sebagai pencarian solusi dari persoalan tersebut. Terdapat dua jenis representasi graf dalam proses pencarian, yaitu graf statis (graf yang sudah terbentuk sebelum proses pencarian dilakukan, biasa direpresentasikan dalam struktur data) dan graf dinamis (graf yang terbentuk saat proses pencarian dilakukan karena graf dibangun selama pencarian solusi). Sedangkan algoritma pencarian solusi secara umum dibagi menjadi dua, yaitu tanpa informasi (*uninformed / blind search*) seperti *Breadth First Search* (BFS), *Depth First Search* (DFS), *Depth Limited Search* (DLS), *Iterative Deepening Search* (IDS), *Uniform Cost Search* (UCS), dan pencarian dengan informasi (*informed search*) seperti Algoritma *Best First Search* dan A*.

B. *Breadth First Search* (BFS)

Breadth First Search atau BFS merupakan salah satu algoritma yang paling sederhana dalam melakukan proses pencarian pada graf, khususnya untuk pencarian tanpa informasi dalam representasi graf statis. Algoritma *minimum-spanning-tree* Prim dan Algoritma pencarian jarak terdekat Dijkstra juga menggunakan ide implementasi yang serupa dengan algoritma pencarian *Breadth First Search*. Dalam algoritma *Breadth First Search*, graf direpresentasikan dalam bentuk $G = (V, E)$ dengan V merupakan simpul awal penelusuran dan E adalah sisi pada graf.

Secara sederhana, algoritma *Breadth First Search* akan memulai penelusuran dari simpul V. Kemudian, akan dilakukan penelusuran terhadap semua simpul yang bertetangga dengan simpul V terlebih dahulu. Selanjutnya, lakukan penelusuran terhadap simpul-simpul lain yang belum pernah dikunjungi dan bertetangga dengan simpul-simpul yang telah dikunjungi sebelumnya. Langkah ini akan dilakukan terus-menerus hingga ditemukan simpul yang dicari atau hingga seluruh simpul telah ditelusuri.

Dalam implementasinya, graf $G = (V, E)$ pada algoritma *Breadth First Search* dapat direpresentasikan dengan matriks ketetanggaan (*adjacency matrix*). Untuk mengetahui simpul yang akan diperiksa, dapat digunakan struktur data antrian (*queue*). Terakhir, untuk mengetahui suatu simpul telah diperiksa atau belum, dapat digunakan struktur data senarai (*list*) atau larik-larik asosiatif (*hash table*) yang bertipe *boolean*. Pada sebuah graf dengan faktor percabangan b dan kedalaman d, Algoritma BFS memiliki kompleksitas waktu $O(b^d)$ dan kompleksitas ruang $O(b^d)$, sehingga relatif besar. Berikut adalah *pseudocode* umum untuk prosedur *Breadth First Search*.

```
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 true }
    MasukAntrian(q, v)     { masukkan simpul awal kunjungan ke dalam antian }

    { 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 then
                  write(w)
                  MasukAntrian(q, w)
                  dikunjungi[v] ← true
             endif
        endfor
    endwhile

    { AntrianKosong(q) }
```

C. *Depth First Search* (DFS)

Depth First Search atau DFS merupakan salah satu algoritma sederhana dalam melakukan proses pencarian tanpa informasi (*blind search*) pada graf selain BFS. Algoritma dimulai dari simpul akar pada struktur data pohon atau salah satu simpul pada graf, kemudian akan melakukan eksplorasi sedalam mungkin sebelum akhirnya harus melakukan runut-balik (*backtrack*). Umumnya, terdapat struktur data berupa *stack* yang digunakan untuk menyimpan simpul-simpul yang telah dikunjungi. Struktur data *stack* juga mempermudah proses *backtracking* pada algoritma DFS. Dalam algoritma *Depth First Search*, graf direpresentasikan dalam bentuk $G = (V, E)$ dengan V merupakan simpul awal penelusuran dan E adalah sisi pada graf.

Algoritma DFS bekerja dengan memulai eksplorasi atau penelusuran pada simpul V. Kemudian, salah satu simpul W yang bertetangga dengan simpul V akan dipilih dan algoritma DFS diulangi mulai dari simpul W tersebut. Ketika algoritma DFS mencapai suatu simpul U sedemikian hingga semua simpul yang bertetangga dengan simpul U sudah dikunjungi, maka akan akan dilakukan proses runut-balik ke simpul yang dikunjungi sebelumnya dan memiliki simpul W yang belum dikunjungi. Algoritma DFS akan berakhir ketika tidak ada lagi simpul yang dapat dikunjungi dari semua simpul yang telah dikunjungi.

Dalam implementasinya, graf $G = (V, E)$ pada algoritma *Depth First Search* dapat direpresentasikan dengan matriks ketetanggaan (*adjacency matrix*). Terdapat pula larik asosiatif bertipe *boolean* untuk menandai simpul yang telah dikunjungi. Pada graf eksplisit, algoritma DFS memiliki kompleksitas waktu $O(|V| + |E|)$ dan kompleksitas ruang $O(|V|)$. Sedangkan pada graf implisit dengan jumlah percabangan *b* dan kedalaman *d*, algoritma DFS memiliki kompleksitas waktu $O(b^d)$ dan kompleksitas ruang $O(bd)$. Berikut adalah *pseudocode* umum untuk prosedur *Depth First Search*.

```
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)                { cetak simpul awal yang dikunjungi }
    dikunjungi[v] ← true    { simpul v telah dikunjungi, tandai 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
```

D. *C# Desktop Application Development*

Desktop Application atau Aplikasi Berbasis *Desktop* merupakan suatu aplikasi atau software milik *desktop* (PC dan laptop) yang mampu beroperasi tanpa terhubung dengan koneksi internet (*offline*). Untuk menggunakannya, pengguna harus melakukan instalasi terlebih dahulu di sistem operasi pada laptop maupun komputer. Aplikasi berbasis *desktop* dibuat dengan menggunakan 3 bahasa pemrograman yaitu .NET, Java, dan Delphi. Adapun bahasa pemrograman .NET meliputi Visual Basic (VB), C++, dan C# (C Sharp). Keunggulan dari .NET yaitu memungkinkan *developer* membuat aplikasi *windows-base* yang di-*launch* melalui Internet Explorer sehingga dapat memanfaatkan *rich window component* untuk mengembangkan sebuah aplikasi *web*. Pengembangan aplikasi *desktop*, terlebih dalam hal tampilan pengguna pada tugas besar ini diimplementasikan dengan menggunakan WinForms dan *integrated development environment* (IDE) Visual Studio. Secara khusus, WinForms digunakan untuk mengembangkan aplikasi *desktop* berbasis Windows dalam bahasa pemrograman C#.

Berikut adalah beberapa prosedur sederhana mengembangkan *desktop application* dengan menggunakan WinForms yang dilakukan dengan IDE Visual Studio :

1. Membuat *project* baru
    a. Buka aplikasi Visual Studio versi terbaru.
    b. Pada laman awal program, pilih menu *Create a new project*.

c. Pada laman yang sama, pilih *Windows Forms App* (*.NET Framework*) *template* untuk bahasa pemrograman C#.
d. Pada laman *Configure your new project*, isi kolom *project name* dengan nama *project* yang diinginkan. Kemudian, tekan tombol *Create*.

2. Membuat aplikasi

Pembuatan project ini berkaitan erat dengan mengakses komponen - komponen yang dapat digunakan dalam menampilkan visualisasi berbagai hasil pencarian dari algoritma yang akan dibuat. Semua penambahan komponen ini dapat ditemukan dalam [Nama Arsip GUI].design pada repository. Dalam arsip design ini, disediakan *Toolbox* yang berisi seluruh komponen yang dapat ditambahkan pada GUI yang akan dibuat. Berikut beberapa komponen yang ditambahkan pada GUI yang telah dimodelkan.

a. Menambahkan tombol

Penambahannya cukup menarik (*click and drag*) komponen bernama "*Button*" ke dalam layar utama GUI. Jenis tombol yang dapat digunakan antara lain *default button*, yaitu tombol pencet yang berisi keterangan nama tombolnya, *radio button*, yang merupakan titik opsi seperti yang hanya dapat memilih satu opsi dari beberapa opsi atau dari beberapa *radio button* lainnya, *check box* yang merupakan kotak yang apabila dipencet akan muncul tanda centang untuk menunjukkan apakah suatu opsi ingin digunakan. Pemodelan, warna, keterangan tombol, ukuran, dan berbagai aspek lainnya dapat dimodifikasikan secara langsung pada properti yang dapat diakses untuk setiap komponen dan hasilnya akan menjadi keadaan awal komponen tersebut ketika GUI dijalankan. Perubahan aspek - aspek pada komponen ini dapat dimanipulasikan lagi pada penjelasan penambahan kode.

b. Menambahkan *label*

Penambahan komponen label secara teknis bersifat sama dengan tombol. Perbedaannya adalah penggunaannya sebagai penamaan ini tidak secara langsung dapat menjalankan suatu algoritma, namun dapat berubah berdasarkan suatu algoritma yang dijalankan seperti komponen tombol

yang dapat mengubah nilai atau isi dari suatu label. Modifikasi properti pada label ini bisa memodifikasikan isi dari teksnya sendiri, ukuran, *font*, warna, serta letak label itu sendiri dalam arsip *design*.

c. Menambahkan *Data Grid*

Data Grid ini merupakan komponen utama tempat visualisasi dari algoritma yang ingin ditunjukkan. Karena algoritma utama dari hasil pencarian rute BFS dan DFS, Cara memasukkan komponen ini ke dalam arsip *design* sama juga, cukup di tekan komponen *datagridview* pada *toolbox* yang disediakan dan ditarik ke dalam tampilan utama GUI sesuai dengan posisi yang diinginkan. Penambahan kode akan dijelaskan pada selanjutnya, namun pada datagridview ini, penambahan kode akan meliputi beberapa aspek seperti pembacaan peta yang digunakan dan format hasil pembacaan peta tersebut ke dalam grid sesuai peraturan pengaturan warna dan teks yang ingin ditampilkan.

d. Menambahkan kode

Penambahan kode dapat dilakukan baik dari GUI maupun tidak. Untuk menyediakan algoritma yang dijalankan dalam fungsional komponen-komponen pada GUI ini, cukup tekan dua kali wujud fisik yang telah diletakkan pada arsip *design* GUI dan akan muncul fungsi berupa *void* pada file utama GUI yang merupakan [Nama Arsip GUI].cs dimana inisialisasi komponen - komponen pada arsip *design* dibuat. Dalam fungsi *void* ini baru hasil dari pemencetan tombol ketika GUI dijalankan dapat dimanipulasi dan ditambahkan algoritma yang dibutuhkan dalam bahasa c#

3. Menjalankan aplikasi
    a. Tekan tombol *start* pada menu.
    b. Coba fungsionalitas pada laman yang berhasil dibuat.
    c. Tutup *dialog box* untuk menghentikan eksekusi program.

BAB III
ANALISIS PEMECAHAN MASALAH

A. Langkah - langkah Pemecahan Masalah

Langkah awal yang dilakukan dalam memecahkan permasalahan utama dari implementasi program ini (*Maze Treasure Hunt*) adalah membagi permasalahan ini menjadi dua topik utama penyelesaian *graph traversal*, yaitu menggunakan algoritma *Breadth First Search* (BFS) dan *Depth First Search* (DFS). Permasalahan yang berikutnya adalah menampilkan hasil pencarian tersebut dalam sebuah *desktop application*, sehingga secara umum, pemecahan masalah ini dibagi menjadi 3 bagian, yaitu :

1. **Langkah - Langkah Pemecahan Masalah dengan Algoritma BFS**

   Secara singkat, Algoritma *Breadth First Search* (BFS) mengutamakan pencarian secara melebar daripada pencarian secara mendalam, sehingga implementasi dari langkah-langkah berikut adalah skema pencarian melebar yang diimplementasikan dalam program. Solusi pemecahan ini dibangun menggunakan konsep graf dinamis sehingga perlu dilakukan penyimpanan rute untuk mempermudah melakukan perunutan langkah pencarian selanjutnya. Berikut adalah prosedurnya.

   a. Mencari koordinat *start* pada peta. Sesuai dengan spesifikasi tugas besar yang diberikan, koordinat *start* dinyatakan dengan simbol *K* besar sebagai koordinat lokasi awal dimana *Krusty Krab* berada. Arahkan simpul aktif (simpul posisi koordinat saat ini) ke koordinat *start*.

   b. Inisiasi sebuah antrian (*queue*) dengan elemen rute pencarian (*route*). Rute adalah sebuah struktur data bentukan yang merupakan representasi dari senarai koordinat. Tujuan inisiasi antrian ini adalah untuk menjawab pertanyaan implementasi solusi graf dinamis pada awal eksplanasi. Sebagai langkah awal, inisiasi sebuah rute-satu-elemen yang berisi koordinat lokasi *start* berada.

   c. Melakukan analisis terhadap simpul jelajah. Sebagai contoh, pada awal pencarian, simpul jelajah adalah semua titik koordinat yang “bertetangga”

dengan titik *start* dari pencarian pada peta. Pada proses lanjutan, simpul jelajah yang dianalisis adalah koordinat terakhir dari rute pertama yang akan dieksekusi pada antrian yang telah dideklarasikan sebelumnya. Adapun pengambilan rute pertama dalam antrian dilakukan dengan cara dihapus dari antrian (*dequeue*). Proses analisis terhadap simpul jelajah dilakukan dengan mempertimbangkan beberapa prioritas yang diatur sebagai berikut :

1) Karena setiap simpul dapat dikunjungi lebih dari satu kali, maka akan diprioritaskan simpul jelajah yang lebih "jarang" dikunjungi.
2) Prioritas pencarian simpul yang digunakan adalah menggunakan prinsip ULDR (*Up - Left - Down - Right*) sehingga simpul yang berada di bagian atas koordinat akan lebih diprioritaskan, diikuti dengan posisi simpul jelajah pada urutan tersebut.

d. Selain menentukan prioritas simpul jelajah pada poin sebelumnya, juga dilakukan analisis terhadap simpul jelajah yang telah ditentukan. Jika simpul jelajah bukan merupakan area yang tidak bisa diakses (simpul yang tidak bisa diakses dinyatakan dengan simbol *X* sesuai spesifikasi tugas besar) dan berada pada koordinat peta yang valid (tidak lebih besar dari absis dan ordinat batas peta maupun lebih kecil dari 0), maka simpul jelajah akan dimasukkan dalam senarai simpul jelajah yang valid dan layak untuk dieksplorasi.
e. Langkah pada bagian c dan d kemudian dielaborasi hingga diperoleh sebuah rute dengan mekanisme sebagai berikut.
    1) Skema ini akan dilakukan hingga ditemukan sebuah rute yang memiliki jumlah harta karun sama dengan jumlah harta karun yang terdapat pada peta secara keseluruhan.
    2) Selama kondisi 1) belum terpenuhi, maka lakukan langkah c (penentuan prioritas simpul jelajah dan penghapusan elemen pertama dari antrian) yang dilanjutkan dengan langkah d (penentuan solusi semua simpul jelajah yang valid).
    3) Untuk setiap koordinat yang dihasilkan dari senarai simpul jelajah pada poin 2), lakukan hal berikut :

a) Inisiasi sebuah rute baru yang merupakan salinan dari rute yang dihapus dari antrian pada poin c.
b) Ubah posisi koordinat saat ini ke koordinat tersebut.
c) Lakukan analisis terhadap koordinat tersebut. Jika koordinat tersebut ternyata adalah sebuah lokasi harta karun (dalam spesifikasi tugas besar, dinyatakan dengan simbol *T*) dan belum pernah dikunjungi oleh rute tersebut, maka daftarkan koordinat tersebut dalam senarai koordinat harta karun yang pernah dikunjungi. Senarai ini merupakan atribut dari struktur data bentukan rute.
d) Setelah proses analisis selesai dilakukan, tambahkan koordinat ini ke dalam rute yang telah diinisialisasi pada awal bagian 3), lalu masukkan antrian (*enqueue*) rute ini ke dalam antrian yang telah dideklarasikan pada awal prosedur.

Rute yang dihasilkan dari prosedur ini adalah sebuah rute final yang berhasil mendapatkan jumlah harta karun sebanyak harta karun yang terdapat dalam peta. Rute ini dan seluruh rangkaian rute proses pencarian diatas kemudian disimpan untuk kemudian ditampilkan pada *desktop application* yang dibangun.

Pada Algoritma *Breadth First Search* juga diimplementasikan Permasalahan *Traveling Salesman Problem* (TSP) yang memungkinkan sebuah jalur kembali ke posisi awalnya dengan biaya jelajah (*cost*) seminimal mungkin. Secara umum skema yang digunakan mirip seperti algoritma BFS yang telah dijelaskan diatas. Yang menjadi pembeda adalah proses pencarian dilakukan dengan mengarahkan koordinat terakhir dari rute solusi algoritma BFS sebagai koordinat awal pencarian rute dan menjadikan posisi *start* sebagai koordinat akhir dari pencarian rute secara keseluruhan. Selain itu, prioritas simpul jelajah juga diubah dengan arah lawannya menjadi RDLU (*Right - Down - Left - Up*) sehingga simpul yang berada di bagian kanan koordinat analisis akan lebih diprioritaskan, diikuti dengan posisi simpul jelajah dengan urutan tersebut. Pada bagian akhir, disimpan pula rute akhir dari skema implementasi permasalahan TSP pada struktur BFS dalam senarai koordinat penampung yang berbeda dengan solusi orisinil (tidak mengimplementasikan TSP).

2. Langkah - Langkah Pemecahan Masalah dengan Algoritma DFS

Secara singkat, algoritma *Depth First Search* (DFS) mengutamakan pencarian secara mendalam pada simpul yang sedang dikunjungi, sehingga implementasi dari langkah-langkah pemecahan masalah berupa pencarian mendalam. Solusi pemecahan ini dibangun menggunakan konsep graf dinamis sehingga akan disimpan suatu rute solusi untuk menyimpan urutan simpul yang telah dikunjungi oleh algoritma DFS. Berikut adalah prosedurnya.

a. Mencari koordinat *start* pada peta. Sesuai dengan spesifikasi tugas besar yang diberikan, koordinat *start* dinyatakan dengan simbol *K* besar sebagai koordinat lokasi awal dimana *Krusty Krab* berada. Simpul aktif (simpul yang sedang dikunjungi) diinisialisasi menjadi koordinat *start*.
b. Kunjungi koordinat *start* untuk menandai bahwa simpul telah dikunjungi di awal penelusuran dan tambahkan koordinat *start* ke dalam rute solusi.
c. Tentukan simpul jelajah yang dapat dikunjungi oleh algoritma DFS. Simpul jelajah yang dapat dikunjungi merupakan simpul yang bertetangga dengan simpul aktif yang sedang dikunjungi. Simpul jelajah juga harus merupakan simpul yang valid untuk dikunjungi. Suatu simpul valid untuk dikunjungi jika simpul merupakan suatu area yang dapat dikunjungi (tidak dinyatakan dengan simbol *X* sesuai spesifikasi tugas besar) dan koordinat simpul berada dalam jangkauan koordinat peta yang valid (tidak lebih besar dari baris dan kolom peta serta tidak lebih kecil dari 0).
d. Setelah didapat simpul jelajah yang dapat dikunjungi, akan ditentukan satu simpul untuk dikunjungi oleh algoritma DFS. Penentuan dilakukan berdasarkan prioritas tertentu dan diimplementasikan lewat proses analisis sebagai berikut.
    1) Karena setiap simpul dapat dikunjungi lebih dari satu kali, maka akan diprioritaskan simpul jelajah yang lebih “jarang” dikunjungi.
    2) Prioritas pencarian simpul yang digunakan adalah menggunakan prinsip ULDR (*Up - Left - Down - Right*) sehingga simpul yang berada di bagian atas koordinat akan lebih diprioritaskan, diikuti dengan posisi simpul jelajah pada urutan tersebut.

Hasil akhir dari proses analisis adalah simpul valid yang bertetangga dengan simpul aktif yang sedang dikunjungi dan memiliki prioritas tertinggi diantara simpul valid tetangga lainnya.

e. Simpul yang telah dipilih kemudian akan dikunjungi. Mekanisme mengunjungi adalah sebagai berikut.
    1) Jika simpul merupakan sebuah harta karun (dinyatakan dengan simbol *T* besar), dan belum pernah dikunjungi sebelumnya. Maka naikkan jumlah harta karun yang sudah didapat sebanyak 1.
    2) Tambahkan jumlah kunjungan pada simpul tersebut sebanyak satu.
    3) Ubah simpul aktif yang sedang dikunjungi menjadi simpul tersebut, dan tambahkan simpul tersebut ke dalam rute solusi.
f. Langkah c, d, dan e kemudian akan dilakukan terus menerus hingga jumlah harta karun yang telah didapat sama dengan jumlah harta karun yang ada pada peta.

Rute yang dihasilkan dari prosedur ini adalah sebuah rute solusi yang telah mendapatkan jumlah harta karun sebanyak harta karun yang terdapat dalam peta. Rute solusi disimpan untuk kemudian ditampilkan pada *desktop application* yang dibangun.

Perlu diketahui bahwa algoritma DFS pada permasalahan ini tidak memanfaatkan proses runut-balik dalam pencariannya. Hal ini dikarenakan sifat permasalahan yang akan dipecahkan menyebabkan runut-balik menjadi suatu hal yang tidak mungkin terjadi. Simpul valid pada peta pasti memiliki tetangga dan pasti berhubungan satu sama lain (terdapat jalan yang menghubungkan *Krusty Krab* dengan semua harta karun pada peta). Selain itu, setiap simpul valid pada peta juga dapat dikunjungi lebih dari satu kali. Kedua kondisi ini menyebabkan algoritma DFS pasti memiliki simpul selanjutnya yang akan dikunjungi dari simpul yang sedang aktif pada setiap kedalaman pencarian dan proses penelusurannya akan terus semakin mendalam. Terjadinya runut-balik justru bertentangan dengan fakta bahwa: proses *backtrack* dimiliki oleh algoritma DFS ketika tidak ada lagi simpul yang mungkin dikunjungi dari simpul aktif, sedangkan premis permasalahan menunjukkan bahwa simpul dapat terus dikunjungi.

Pada algoritma *Depth First Search* juga diimplementasikan permasalahan *Traveling Salesman Problem* (TSP) yang memungkinkan sebuah jalur kembali ke posisi awalnya dengan biaya jelajah (*cost*) seminimal mungkin. Permasalahan TSP yang perlu diselesaikan adalah mencari jalur kembali ke koordinat *start* (*Krusty Krab*) setelah mendapatkan seluruh harta karun. Secara umum skema yang digunakan mirip seperti algoritma DFS yang telah dijelaskan diatas. Yang menjadi pembeda adalah sebelum penelusuran dilakukan, jumlah kunjungan semua simpul diinisialisasi menjadi 0. Proses penelusuran kemudian dilakukan dengan koordinat *start* berupa koordinat terakhir harta karun yang didapat, dan memiliki kondisi akhir berupa simpul aktif merupakan *Krusty Krab* (ditandai dengan simbol *K* besar). Rute solusi yang disimpan berupa rute untuk mendapatkan seluruh harta karun dan kembali ke *Krusty Krab*.

3. Mengasosiasikan Hasil Pencarian pada *Desktop Application*

Arsip - arsip pembentukkan awal yang disediakan *Windows Form Application* dalam pembentukan proyek ini berupa tiga arsip utama yang menjadi pengatur GUI yang ingin dibuat. Arsip yang dapat memodifikasi tampilan GUI merupakan [Nama Arsip].design, penyimpanan data komponen - komponen yang telah dibentuk pada [Nama Arsip].designer, dan inisialisasi tampilan utama beserta fungsionalitas komponen - komponen yang mungkin dibentuk pada [Nama Arsip].cs. Hasil implementasi utama dari algoritma BFS dan DFS dapat secara langsung digunakan pada arsip GUI ini dengan membentuk objek pemanggilan tipe data dari algoritma yang telah dibuat, yaitu dalam bentuk kelas.

Penggabungan algoritma utama dimulai dengan target utama visualisasi pada *Desktop Application* ini, yaitu visualisasi BFS dan DFS. Komponen *data grid view* merupakan komponen utama yang digunakan dalam mengasosiasikan hasil pencarian ke dalam bentuk visual. Pertama, telah dibentuk algoritma *parsing* yang digunakan untuk membaca arsip *txt*, kemudian pembacaan hasil *parsing* tersebut menggunakan algoritma lagi dengan membuat sebuah kelas *DataGrid* yang digunakan untuk datagridview nantinya membaca juga hasil peta, karena *parsing* dilakukan dua kali, yaitu satu sebagai peta yang digunakan untuk algoritma pencarian rute dan yang

menggunakan kelas *DataGrid* ini untuk pembacaan peta pada *data grid view*. Setelah berhasilnya dibaca, *data grid view* dimodifikasi sehingga menampilkan isi peta dengan cara dilakukannya *Cell Formatting* terlebih dahulu. *Cell Formatting* ini merupakan aturan format awal tampilan *data grid view* berdasarkan pembacaan data. Tujuan adanya format ini untuk mengubah suatu nilai hasil pembacaan setiap petak dan dilakukan penggantian warna pada petak tersebut untuk menunjukkan peta awal yang ditunjukkan. Misalkan, apabila suatu kotak dibaca "X" oleh pada tabel *data grid view* ini, maka dibuatkan *cell formatting* yang mengubah kotak tersebut menjadi warna hitam untuk menandakan pencarian rute solusi tidak bisa melewati kotak tersebut saat visualisasi. Berlaku juga untuk nilai - nilai lain yang memungkinkan pada kotak - kotak yang dibaca.

Menjalankan algoritma BFS dan DFSnya menggunakan komponen - komponen yang dapat diakses seperti beberapa jenis tombol dan penamaan fungsionalitas tersebut seperti tombol yang digunakan untuk mengakses algoritma BFS, DFS, serta keterangan tombol lainnya yang telah dijelaskan jenis - jenis tombol pada landasan teori. Tombol yang digunakan ini mengakses objek bertipe BFS atau DFS, sesuai pilihan user dan menjalankan algoritma utama *solve* yang telah dibentuk. Hasil dari pencarian algoritma tersebut menyimpan himpunan hasil berupa rute yang berisi koordinat - koordinat yang dikunjungi untuk setiap rute sehingga visualisasi cukup membuat suatu fungsi yang mengiterasi isi dari himpunan pencarian rute dan menghitung jumlah kunjungan setiap petak yang dikunjungi untuk mengatur penggunaan warna yang ingin digunakan dengan cara menampilkan warna yang lebih gelap untuk petak yang semakin sering dikunjungi.

B. Elemen-elemen Algoritma BFS dan DFS

Berdasarkan uraian diatas, berhasil disimpulkan elemen-elemen yang berperan sebagai hasil pemetaan dari permasalahan implementasi Algoritma BFS dan DFS. Berikut adalah penjelasan lebih mendalam mengenai masing-masing elemen tersebut.

1. Elemen-elemen Algoritma *Breadth First Search* (BFS)

Seperti yang telah disampaikan pada bagian langkah-langkah Algoritma, implementasi algoritma ini menggunakan pembangkitan solusi dengan prinsip graf

dinamis. Jika pada implementasi Algoritma BFS yang umum, graf direpresentasikan dalam bentuk $G = (V, E)$ dengan V merupakan simpul awal penelusuran dan E adalah sisi pada graf, Pada implementasi Algoritma untuk kasus ini, karena graf direpresentasikan dalam matriks $m \times n$ dengan m menyatakan jumlah baris dan n menyatakan jumlah kolom, maka proses traversal dimulai dari koordinat awal (dilambangkan dengan K pada peta) dan simpul dinyatakan sebagai pasangan absis-ordinat penyusun matriks peta. Hal ini membuat pencarian harta karun sebagai pencarian sebuah simpul khusus pada peta (dalam hal ini simpul T yang menyatakan lokasi harta karun) dengan restriksi batas jelajah yaitu simpul dengan lambang X dan ukuran baris dan kolom dari matriks jelajah.

Lebih lanjut, pemrosesan algoritma menggunakan antrian membuat tidak diperlukannya struktur data senarai atau larik-larik asosiatif bertipe data *boolean* untuk mengetahui suatu simpul telah diperiksa atau belum, cukup masukkan antrian dan hapus antrian hingga terdapat salah satu rute yang memiliki jumlah harta karun sama dengan jumlah harta karun yang terdapat pada peta. Prioritas pemilihan elemen senarai simpul jelajah yang didasarkan pada jumlah "seberapa sering" simpul tersebut dikunjungi ditangani dengan baik pada representasi matriks dari peta yang juga menyimpan jumlah berapa kali simpul yang bisa diakses (dilambangkan dengan R pada peta sesuai spesifikasi, atau simpul lain seperti T dan K) telah dikunjungi.

2. Elemen-elemen Algoritma *Depth First Search* (DFS)

Jika pada implementasi algoritma DFS yang umum, graf direpresentasikan dalam bentuk $G = (V, E)$ dengan V merupakan simpul awal penelusuran dan E adalah sisi pada graf, Pada implementasi algoritma untuk kasus ini, karena graf direpresentasikan dalam matriks simpul $m \times n$ dengan m menyatakan jumlah baris dan n menyatakan jumlah kolom, maka proses traversal dimulai dari koordinat awal (dilambangkan dengan K pada peta) dan simpul dinyatakan sebagai pasangan absis-ordinat penyusun matriks peta yang juga memiliki simbol dan jumlah kunjungan. Hal ini membuat pencarian harta karun sebagai pencarian sebuah simpul khusus pada peta (dalam hal ini simpul T yang menyatakan lokasi harta karun) dengan restriksi batas jelajah yaitu simpul dengan lambang X dan ukuran baris dan

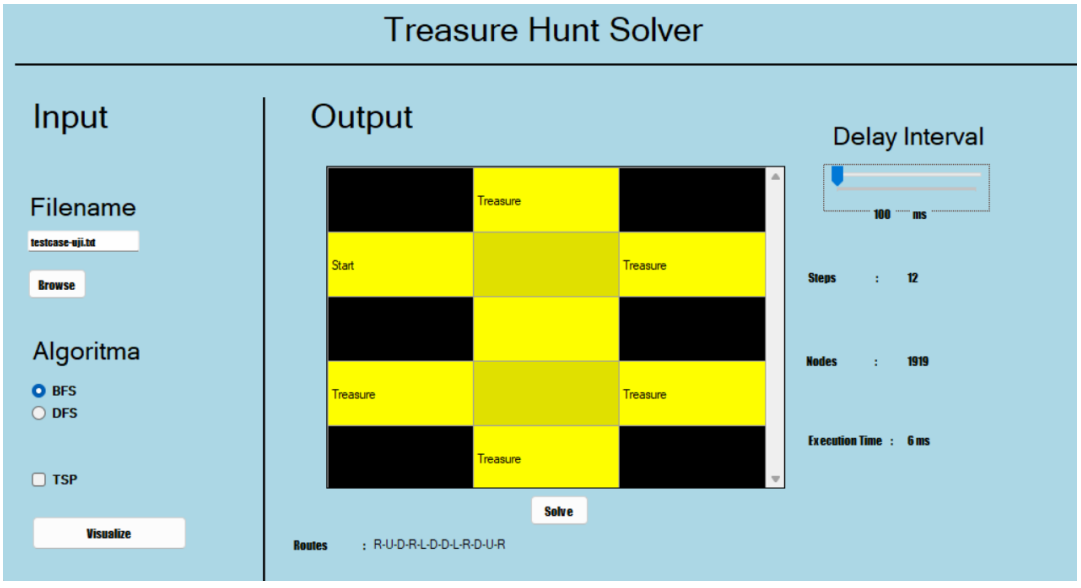
kolom dari matriks jelajah. Simpul *Krusty Krab* berperan sebagai simpul akar, dan pencarian algoritma DFS berhenti setelah seluruh harta karun didapatkan.

Selain menyimpan matriks simpul untuk merepresentasikan graf, algoritma DFS juga menyimpan rute penelusuran sebagai urutan koordinat simpul yang dikunjungi mulai dari simpul akar hingga seluruh harta karun didapatkan.

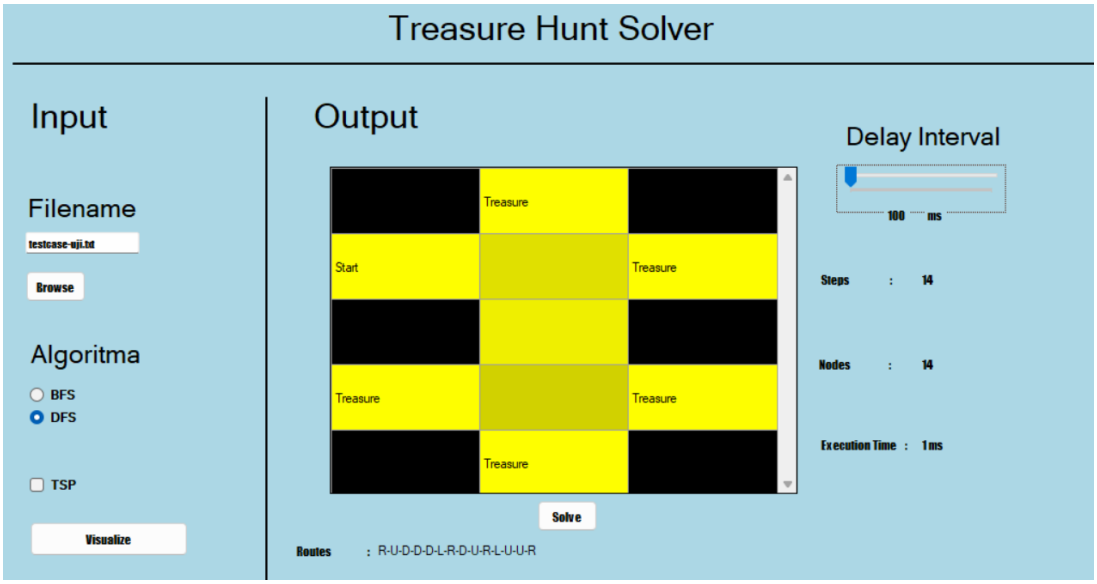
C. Ilustrasi Implementasi pada Kasus Lain

Contoh kasus yang akan dianalisis pada bagian ini adalah kasus yang memungkinkan proses pencarian melewati sebuah simpul lebih dari sekali. Berikut adalah hasilnya.

Gambar 3.1. Masukan *file maze*



Gambar 3.2. Hasil dan langkah penelusuran dengan algoritma BFS



Gambar 3.3. Hasil dan langkah penelusuran dengan algoritma DFS

Gambar 3.1. merupakan peta masukan yang akan dianalisis. Adapun alasan pemilihan desain peta tersebut adalah karena jalur yang ada memaksa adanya runut-balik untuk mendapatkan semua harta karun yang mengakibatkan satu simpul dapat dilalui lebih dari satu kali. Seperti yang teramati pada Gambar 3.2. yang merupakan penyelesaian dari algoritma *Breadth First Search* (BFS) pada kasus ini, memiliki panjang langkah 12 yaitu R-U-D-R-L-D-D-L-R-D-U-R dengan jumlah total simpul yang dikunjungi adalah 1919 simpul. Sedangkan pada algoritma *Depth First Search* (DFS) pada Gambar 3.3. memerlukan 14 langkah yaitu R-U-D-D-D-L-R-D-U-R-L-U-U-R dengan nilai jumlah simpul yang dikunjungi sama dengan jumlah langkah.

Pemetaan komponen dari algoritma BFS dan DFS pada penyelesaian sama seperti yang telah dijabarkan sebelumnya. Berdasarkan hasil implementasi dari program, diperoleh bahwa hasil yang diperoleh melalui algoritma BFS lebih mangkus daripada hasil dari algoritma DFS dengan jumlah langkah yang diperlukan untuk mendapatkan semua rute lebih sedikit. Akan tetapi, di sisi lain, algoritma DFS lebih sangkil daripada algoritma BFS karena jumlah simpul yang dikunjungi jauh lebih sedikit. Hal ini dapat terjadi karena algoritma DFS memiliki kompleksitas ruang yang jauh lebih baik daripada algoritma BFS.

BAB IV

IMPLEMENTASI DAN PENGUJIAN

A. Implementasi Program

Kelas Koordinat

```
class Coordinate
{ Struktur data yang menyimpan nilai absis dan ordinat dari masing-masing posisi
  dalam peta }

Deklarasi
    { atribut - private }
    x   : integer
    y   : integer

    { method - public }
    Coordinate ()
    { default konstruktor objek Coordinate }

    Coordinate (input x : integer, y : integer)
    { konstruktor objek Coordinate }

    function getX () → integer
    { getter atribut x }

    function getY () → integer
    { getter atribut y }

    function moveUp (input coordinate : Coordinate) → Coordinate
    { memindah posisi dari koordinat ke atas dengan membuat objek baru }

    function moveDown (input coordinate : Coordinate) → Coordinate
    { memindah posisi dari koordinat ke bawah dengan membuat objek baru }

    function moveLeft (input coordinate : Coordinate) → Coordinate
    { memindah posisi dari koordinat ke kiri dengan membuat objek baru }

    function moveRight (input coordinate : Coordinate) → Coordinate
    { memindah posisi dari koordinat ke kanan dengan membuat objek baru }
```

Kelas Node

```
class Node
{ Struktur data yang menyimpan setiap posisi pada peta dengan atribut pembantunya
  seperti tipe dan jumlah dikunjungi }

Deklarasi
    { atribut - private }
    type        : char
    loc         : Coordinate
    visitedTime : integer
```

```
    { method - public }
    Node ()
    { default konstruktor objek Node }

    Node (input type : char, x : integer, y : integer, visitedTime : integer)
    { konstruktor objek Node }

    function getType () → char
    { getter atribut type }

    function getVisitedTime () → integer
    { getter atribut visitedTime }

    procedure reset (input/output node : Node)
    { melakukan pengaturan ulang terhadap nilai atribut visitedTime }

    procedure visit (input/output node : Node)
    { melakukan penambahan nilai visitedTime sebanyak 1 }
```

Kelas MatrixNode

```
class MatrixNode
{ Struktur data yang menyimpan matriks simpul dan informasi lain terkait peta }

Deklarasi
    { atribut - private }
    map             : Matrix of Node
    start           : Coordinate
    row             : integer
    col             : integer
    numOfTreasure   : integer

    { method - public }
    MatrixNode ()
    { default konstruktor objek MatrixNode }

    function getRow () → integer
    { getter atribut row }

    function getCol () → integer
    { getter atribut col }

    function getStart () → Coordinate
    { getter atribut start }

    function getTreasure () → integer
    { getter atribut numOfTreasure }

    function getVisitedTime (input test : Coordinate) → integer
    { mengembalikan jumlah kunjungan pada simpul dengan koordinat test }

    function getMapElement (input i : integer, j : integer) → integer
    { getter elemen dengan baris ke-i dan kolom ke-j dari map }

    procedure clearVisits ()
    { mengubah jumlah kunjungan semua simpul pada peta menjadi 0 }
```

```
    procedure fillMatrix (input fileName : string)
    { menginisiasi seluruh atribut objek MatrixNode sesuai hasil parsing file }

    function isCoordinateValid (input test : Coordinate) → boolean
    { mengembalikan true jika test valid sebagai koordinat pada matriks, false
      jika tidak }

    function isCoordinatePassable (input test : Coordinate) → boolean
    { mengembalikan true jika simpul dengan koordinat test tidak memiliki simbol
      berupa simbol X, false jika tidak }

    function isTreasure (input test : Coordinate) → boolean
    { mengembalikan true jika simpul dengan koordinat test memiliki simbol
      berupa simbol T, false jika tidak }

    function isVisited (input test : Coordinate) → boolean
    { mengembalikan true jika simpul dengan koordinat test sudah pernah
      dikunjungi setidaknya satu kali, false jika tidak }

    procedure visitCoordinate (input test : Coordinate)
    { menaikkan jumlah kunjungan pada simpul dengan koordinat test sebesar 1 }

    function convertToStringArray () → List of string
    { mengubah setiap baris pada simpul menjadi suatu string berisi simbol
      simpul }
```

Kelas Rute

```
class Route
{ Struktur data bentukan rute yang menyimpan senarai koordinat }

Deklarasi
    { atribut - private }
    routes              : List of Coordinate
    visitedTreasure     : List of Coordinate
    numsTreasure        : integer
    length              : integer
    currentCoordinate   : Coordinate

    { method - public }
    Route ()
    { default konstruktor kelas Route }

    function getNumsTreasure () → integer
    { getter atribut numsTreasure }

    function getCurrentCoordinate () → Coordinate
    { getter atribut currentCoordinate }

    function getRouteLength () → integer
    { getter atribut length }

    function getRoute () → List of Coordinate
    { getter atribut routes }
```

```
    function getRouteTop () → Coordinate
    { getter nilai elemen terakhir dari routes }

    function getRouteTopX () → integer
    { getter nilai absis elemen terakhir dari routes }

    function getRouteTopY () → integer
    { getter nilai ordinat elemen terakhir dari routes }

    procedure addNumsTreasure (input/output route : Route, currentLoc :
                               Coordinate)
    { menambahkan jumlah treasure yang telah dikunjungi
      menambahkan koordinat lokasi treasure yang telah dikunjungi }

    function isTreasureVisited (input route : Route, currentLoc : Coordinate)
                                → boolean
    { melakukan traversal terhadap isi senarai hingga ditemukan Coordinate di
      dalam list atau tidak ditemukan }

    procedure addElement (input/output route : Route, currentLoc : Coordinate)
    { menambahkan Coordinate pada List of Coordinate visitedTreasure }

    procedure copyRoute (input route1 : Route, output route2 : Route)
    { menyalin seluruh atribut rute dari route1 ke route2 }

    function getSequenceOfDirection (input route : Route) → List of string
    { mengembalikan sekuens karakter arah dari setiap langkah dari awal
      pencarian hingga tiba di tujuan }
```

Kelas Parser

```
class Parser
{ Struktur data objek yang bertanggung jawab untuk melakukan parsing file dan
  menyimpan hasil parsing }

Deklarasi
    { atribut - private }
    map         : Matrix of Node
    fileName    : string
    row         : integer
    col         : integer
    i           : integer
    j           : integer
    startPoint  : integer
    startLoc    : Coordinate
    treasure    : integer

    { method - public }
    Parser ()
    { default konstruktor objek Parser }

    Parser (input fileName : string)
    { konstruktor objek Parser )

    function getRow () → integer
    { getter atribut row }
```

```
    function getCol () → integer
    { getter atribut col }

    function getTreasure () → integer
    { getter atribut treasure }

    function getStart () → Coordinate
    { getter atribut startLoc }

    procedure checkColumnInRow (input col : integer)
    { melemparkan exception jika col tidak sama dengan kolom mula-mula peta }

    procedure checkCharacterValidity (input s : string)
    { melemparkan exception jika s tidak sesuai dengan simbol T, R, X, atau K }

    procedure processCharacter (input c : character)
    { mengubah atribut objek Parser tergantung pada c, jika terdapat 2 simbol K
      yang muncul, maka akan melemparkan exception }

    function parseAndFill () → integer
    { mengembalikan matriks simpul hasil parsing }
```

Kelas BFS

```
class BFSSolver
{Melakukan pemrosesan traversal graf pencarian harta karun dengan algoritma BFS}

Deklarasi
    { atribut - private }
    maze              : MatrixNode
    currentLoc        : Coordinate
    queueRoute        : Queue of Route
    queueRouteTSP     : Queue of Route
    finalRoute        : Route
    finalRouteTSP     : Route
    sequenceOfRoute   : List of Route
    nodeCount         : integer

    { method - public }
    BFSSolver ()
    { default konstruktor kelas BFS }

    BFSSolver (input maze : MatrixNode)
    { konstruktor kelas BFS }

    function getSequence () → List of Route
    { getter atribut sequenceOfRoute }

    function getFinalRoute () → Route
    { getter atribut finalRoute }

    function getFinalRouteTSP () → Route
    { getter atribut finalRouteTSP }

    function expandWithBFS (input currentLoc : Coordinate, flag : integer) →
                            List of Coordinate
    { melakukan pencarian simpul jelajah dengan algoritma BFS
```

```
      menggunakan skema penentuan simpul jelajah yang dibahas pada bagian
      langkah-langkah.
      input : flag menyatakan apakah ekspan untuk BFS biasa atau TSP }

    function isSearchDone (input queueRoute : Queue of Route, maze : MatrixNode)
                           → boolean
    { melakukan analisis apakah proses pencarian telah selesai dilakukan
      dalam hal ini, terdapat rute yang memiliki jumlah harta karun sama dengan
      jumlah harta karun dalam peta }

    function isSearchDoneTSP (input queueRouteTSP : Queue of Route, maze :
                              MatrixNode) → boolean
    { melakukan analisis apakah proses pencarian TSP telah selesai dilakukan
      dalam hal ini, terdapat posisi koordinat akhir rute sudah tiba pada titik
      awal pencarian dimulai }

    procedure Enqueue (input/output q : Queue of Route, input r : Route)
    { memasukkan Route r ke dalam Queue of Route q pada posisi belakang }

    procedure Dequeue (input/output q : Queue of Route, output r : Route)
    { menghapus Route r dari kepala Queue of Route q }

    procedure Add (input/output l : List of Route, input r : Route)
    { memasukkan Route r ke dalam List of Route l pada posisi belakang }

    procedure solve (input maze : MatrixNode, currentLoc : Coordinate, output
                     finalRoute : Route)
    { menyelesaikan dengan algoritma BFS }

    procedure solveTSP (input maze : MatrixNode, currentLoc : Coordinate, output
                        finalRouteTSP : Route)
    { menyelesaikan dengan algoritma BFS lengkap dengan skema TSP }

    function numsOfNode (input maze : MatrixNode) → integer
    { mendapatkan jumlah node yang dikunjungi
      jika dikunjungi lebih dari 1 kali, maka juga diperhitungkan }

    function numsOfSteps (input finalRoute : Route) → integer
    { mendapatkan jumlah step yang diperlukan dari awal hingga solusi
      ditemukan dengan algoritma BFS }

    function numsOfStepsTSP (input finalRouteTSP : Route) → integer
    { mendapatkan jumlah step yang diperlukan dari awal hingga solusi
      ditemukan dengan algoritma BFS dengan TSP }

Algoritma
    procedure solve (input maze : MatrixNode, currentLoc : Coordinate, output
                     finalRoute : Route)

        Route(initial_route)   { buat rute kosong }
        addElement(initial_route, currentLoc)
        visitCoordinate(maze, currentLoc)
        nodeCount ← nodeCount + 1
        Enqueue(queueRoute, initial_route)
        Add(sequenceOfRoute, initial_route)

        { selama belum memenuhi kondisi selesai }
        while not isSearchDone(queueOfRoute, maze) do
            Dequeue(queueRoute, initial_route)
            currentLoc ← getRouteTop(initial_route)
```

```
            for coordinate in expandWithBFS(currentLoc, 1)do
                Route(temp_route)   { buat rute tmp kosong }
                copyRoute(initial_route, temp_route)
                currentLoc ← coordinate

                if isTreasure(maze, currentLoc) and not
                   isTreasureVisited(temp_route, currentLoc) then
                     addNumsTreasure(temp_route, currentLoc)
                endif

                visitCoordinate(maze, currentLoc)
                nodeCount ← nodeCount + 1
                addElement(temp_route, currentLoc)
                Enqueue(queueRoute, temp_route)
                Add(sequenceOfRoute, temp_route)
            endfor
        endwhile

        { Ditemukan solusi jalur menggunakan BFS }

    procedure solveTSP (input maze : MatrixNode, currentLoc : Coordinate, output
                        finalRouteTSP : Route)

        solve(maze, currentLoc, finalRoute)
        clearVisit(maze)
        Route(initial_route)   { buat rute kosong }
        copyRoute(finalRoute, initial_route)
        visitCoordinate(maze, currentLoc)
        Enqueue(queueRouteTSP, initial_route)
        Add(sequenceOfRoute, initial_route)

        { selama belum memenuhi kondisi selesai }
        while not isSearchTSPDone(queueOfRoute, maze) do
            Dequeue(queueRoute, initial_route)
            currentLoc ← getRouteTop(initial_route)
            for coordinate in expandWithBFS(currentLoc, 2) do
                Route(temp_route)   { buat rute tmp kosong }
                copyRoute(initial_route, temp_route)
                currentLoc ← coordinate

                visitCoordinate(maze, currentLoc)
                nodeCount ← nodeCount + 1
                addElement(temp_route, currentLoc)
                Enqueue(queueRouteTSP, temp_route)
                Add(sequenceOfRoute, temp_route)
            endfor
        endwhile

        { Ditemukan solusi jalur menggunakan BFS lengkap dengan TSP }
```

Kelas DFS

```
class DFSSolver
{ Melakukan pemrosesan pencarian harta karun pada matriks simpul dengan algoritma
  DFS }

Deklarasi
    { atribut - private }
```

```
    maze               : MatrixNode
    currentLoc         : Coordinate          { simpul aktif }
    route              : List of Coordinate
    treasureCollected  : integer

    { method - public }
    DFSSolver ()
    { default konstruktor objek DFSSolver }

    function getRoute() → List of Coordinate
    { getter atribut route }

    procedure fillMaze (input fileName : string)
    { melakukan parsing file untuk atribut MatrixNode }

    function checkNeighbourNode() → List of boolean
    { mengembalikan list of boolean yang menunjukkan apakah tetangga simpul aktif
      dapat dikunjungi atau tidak. Urutan simpul mengikuti prioritas ULDR }

    function checkNeighbourNodeVisits (input visitable : List of boolean)
                                    → List of integer
    { mengembalikan list of int yang menunjukkan jumlah kunjungan tetangga simpul
      aktif. Jika tetangga tidak dapat dikunjungi, maka jumlah kunjungan = -1.
      Urutan mengikuti prioritas ULDR }

    function expandWithDFS() → Coordinate
    { mengembalikan koordinat simpul yang akan dikunjungi selanjutnya oleh
      algoritma DFS }

    procedure visitCurrentLocation (input flag : integer)
    { melakukan prosedur mengunjungi simpul aktif. Jika flag tidak sama dengan 0
      maka jumlah harta karun yang didapat tidak akan naik meskipun simpul aktif
      adalah sebuah harta karun }

    procedure solve ()
    { melakukan algoritma DFS untuk mencari seluruh harta karun }

    procedure solveAndTSP ()
    { melakukan algoritma DFS untuk mencari seluruh harta karun dan kembali ke
      Krusty Krab }

    function getSequenceOfDirection () → List of string
    { mengubah rute solusi pencarian menjadi urutan-urutan arah untuk melakukan
      rute solusi }
```

Algoritma

```
    procedure solve ()
        visitCurrentLocation(0)   { kunjungi simpul start pertama )

        { ulangi proses penelusuran selama masih ada harta karun tersisa }
        while (treasureCollected ≠ jumlahHartaKarunDiPeta) do
            route = route + currentLoc {tambahkan simpul aktif ke rute solusi}
            currentLoc = expandWithDFS() { cari simpul aktif berikutnya }
            visitCurrentLocation(0)    { kunjungi simpul aktif }
        endwhile

        route = route + currentLoc { tambahkan simpul aktif terakhir }

        { pencarian selesai, simpul aktif terakhir adalah simpul harta karun
          terakhir }
```

```
procedure solveAndTSP ()
     solve() { temukan rute solusi ke seluruh harta karun}
     maze.clearVisits() { ubah jumlah kunjungan semua simpul menjadi 0}

     visitCurrentLocation(1)   { kunjungi simpul harta karun terakhir )

     { ulangi proses penelusuran selama belum mencapai Krusty Krab }
     while (koordinat simpul aktif ≠ koordinat simpul start) do
           currentLoc = expandWithDFS() { cari simpul aktif berikutnya }
           visitCurrentLocation(1)    { kunjungi simpul aktif }
           route = route + currentLoc {tambahkan simpul aktif ke rute solusi}
     endwhile

     { pencarian selesai, simpul aktif terakhir adalah simpul start }
```

B. Struktur Data dan Spesifikasi Program

Penjelasan mengenai struktur data yang digunakan dan spesifikasi implementasinya dalam program yang dibuat dijabarkan sebagai berikut.

1. Senarai/Larik (*Array/List*)

Struktur data larik banyak digunakan di dalam program. Struktur ini digunakan untuk melakukan berbagai hal dan menyimpan berbagai tipe data sebagai berikut.

1) *List of integer*

   Larik berisi tipe data *integer* digunakan untuk mencatat jumlah kunjungan setiap simpul tetangga dari simpul aktif yang sedang dikunjungi.

2) *List of boolean*

   Larik berisi tipe data *boolean* digunakan untuk menyimpan status apakah simpul tetangga dari simpul aktif yang sedang dikunjungi dapat dikunjungi atau tidak

3) *List of string*

   Larik berisi tipe data *string* digunakan untuk menyimpan urutan arah rute solusi (L, R, U, D) yang harus dilakukan untuk mendapatkan seluruh harta karun dan/atau kembali ke koordinat awal (*Krusty Krab*). Larik *string* juga digunakan untuk menyimpan simbol simpul dari setiap baris matriks. Hal ini diperlukan untuk memvisualisasikan matriks menjadi sebuah *DataGrid* pada GUI.

4) *List of Coordinate*

Larik berisi objek koordinat diperlukan untuk mencatat rute pencarian pada algoritma DFS dan BFS. Larik ini kemudian akan digunakan untuk memvisualisasikan hasil pencarian rute pada GUI.

5) *List of Route*:

Larik berisi objek rute diperlukan untuk menyimpan semua rute yang telah dikunjungi oleh algoritma BFS. Hal ini diperlukan karena visualisasi proses pencarian algoritma BFS pada GUI perlu menampilkan setiap rute yang dilakukannya.

2. Matriks (Larik multidimensi)

Struktur data matriks digunakan untuk melakukan representasi terhadap elemen-elemen yang terdapat pada graf. Graf yang direpresentasikan dalam bentuk $G = (V, E)$ dengan V merupakan simpul awal penelusuran dan E adalah sisi pada graf direpresentasikan dalam matriks simpul $m \times n$ dengan m menyatakan jumlah baris dan n menyatakan jumlah kolom. Secara umum, perwujudan dari matriks ini adalah sebuah *Matrix of Node* yang menyimpan sejumlah simpul dengan informasi tertentu. Pada bagian ini disimpan pula informasi terkait peta, lokasi *start*, jumlah baris, kolom, dan jumlah harta karun pada peta tersebut dan diimplementasikan dengan membentuk kelas bernama MatrixNode.

3. Simpul

Struktur data simpul digunakan untuk merepresentasikan petak pada peta yang berperan sebagai simpul pada graf. Implementasinya berupa sebuah objek *Node*. Simpul memiliki beberapa atribut berupa simbol simpul (K, R, T, X), koordinat simpul (berupa objek koordinat), dan jumlah kunjungan simpul tersebut. Adanya struktur data simpul memudahkan proses pencarian rute pada algoritma DFS dan BFS. Dengan adanya struktur data simpul, kedua algoritma tidak perlu lagi menyimpan matriks tersendiri untuk mencatat jumlah kunjungan setiap simpul pada matriks terpisah. Selain itu, atribut koordinat juga memudahkan kedua algoritma untuk menelusuri simpul pada peta.

4. Antrian (*Queue*)

Implementasi dari struktur data antrian pada program yang dibuat adalah dengan menggunakan *collection* dari kelas Systems.Collection.Generic milik bahasa

pemrograman C#. Adapun struktur data antrian digunakan untuk menyimpan daftar rute yang akan dikunjungi pada saat skema traversal graf dengan metode *Breadth First Search* (BFS) dilakukan. Beberapa metode dasar yang digunakan pada struktur data dari *Collection* ini adalah sebagai berikut.

a. Tambah antrian (*Enqueue*), berfungsi untuk memasukkan rute ke dalam antrian rute (*Queue of Route*)
b. Kurangi antrian (*Dequeue*), berfungsi untuk menghapus rute dari antrian rute.

Adapun alasan dipilihnya struktur data antrian adalah karena struktur ini menerapkan prinsip FIFO (*first-in first-out*) yang memudahkan skema pengecekan semua rute yang mungkin secara bergantian dan sistematis sehingga simpul jelajah yang dihasilkan lebih teratur dan terarah.

C. Tata Cara Penggunaan Program

Berikut adalah beberapa cara dan penjelasan mengenai cara penggunaan program, mulai cari cara kompilasi, cara menjalankan, hingga cara mengoperasikan *desktop application* yang telah dibangun :

1. **Cara Mengkompilasi dan Menjalankan Program**
    a. Lakukan clone *repository* melalui terminal dengan *command* berikut

    ```
    git clone https://github.com/mikeleo03/Tubes2_MencariMaknaSpek.git
    ```

    b. Buka solution dari *repository* ini (TreasureHunt.sln) pada Visual Studio.
    c. Lakukan kompilasi dengan menekan tombol *start* berwarna hijau pada Visual Studio.
    d. Jika kompilasi berhasil, maka akan muncul *file* bernama TreasureHunt.exe pada *folder* bin yang sejajar dengan *folder* src dan akan muncul tampilan dari *desktop application* pada layar.
    e. Selain cara tersebut, program dapat pula dijalankan dengan menggunakan *executable file* yang telah tersedia pada *folder* bin dengan cara masuk ke dalam direktori kode, lalu jalankan *command* berikut pada terminal

    ```
    bin/TreasureHunt
    ```

2. Cara Mengoperasikan Program

a. Pada bagian awal akan muncul tampilan muka dari *desktop application* yang telah dibangun

b. Pengguna dapat memasukkan *file maze* yang ingin dianalisis dengan menekan tombol Browse pada bagian kiri atas. Selanjutnya pengguna akan diarahkan pada laman pemilihan dokumen. Pilih *file* yang ingin diproses lalu tekan tombol open.

   Jika *file* Anda valid, maka aplikasi akan menunjukkan peta hasil pembacaan dari file yang telah digenerasi oleh aplikasi. Jika tidak, maka aplikasi akan mengabarkan *error* yang mengatakan *file* tidak valid sehingga Anda harus memberikan masukan ulang.

c. Setelah hasil generasi petak dimunculkan, pengguna diarahkan untuk memilih algoritma yang akan digunakan. Pilihan algoritma terdiri atas algoritma BFS dan DFS. Pemrosesan tidak akan dilanjutkan sebelum pengguna memilih satu jenis algoritma yang diinginkan.

d. Selanjutnya, pengguna akan diminta untuk memilih apakah ingin mengimplementasikan TSP pada proses pencarian. Jika iya, maka pengguna harus memberikan *checklist* pada kolom TSP yang tersedia. Secara *default* proses pencarian tidak mengimplementasikan TSP.

e. Setelah seluruh prosedur diatas dilakukan, pengguna dapat melihat hasil penyelesaian dari algoritma dan skema yang dipilih dengan menekan tombol Solve pada bagian tengah bawah. Informasi terkait proses pencarian meliputi rute, jumlah langkah, jumlah simpul yang dikunjungi, dan waktu eksekusi akan muncul sesaat setelah tombol tersebut ditekan.

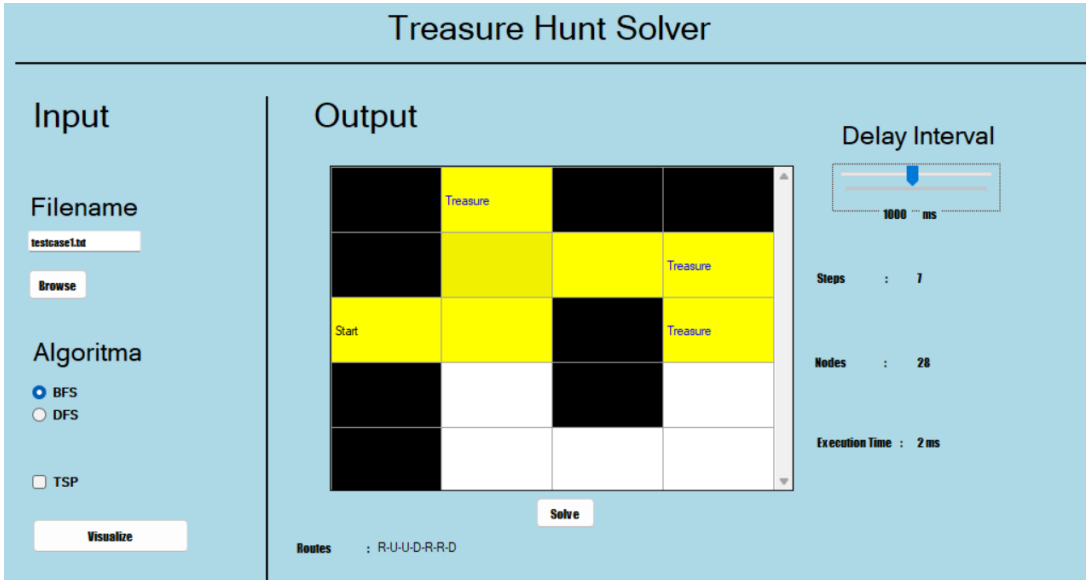
f. Sebagai pelengkap, pengguna juga dapat melihat hasil akhir dan proses pencarian menggunakan algoritma yang dipilih hingga ditemukan solusi rute yang memenuhi kondisi dengan menekan tombol Visualize. Pengguna juga dapat mengatur interval *delay* dari penampilan proses pencarian dengan menggerakkan *scroll bar* yang berada pada bagian kanan atas sesuai kebutuhan.

D. Hasil Pengujian

Berikut adalah hasil pengujian dari program. Adapun *test case* yang digunakan adalah *test case* yang diberikan oleh asisten pada spesifikasi tugas besar dan *test case* tambahan.

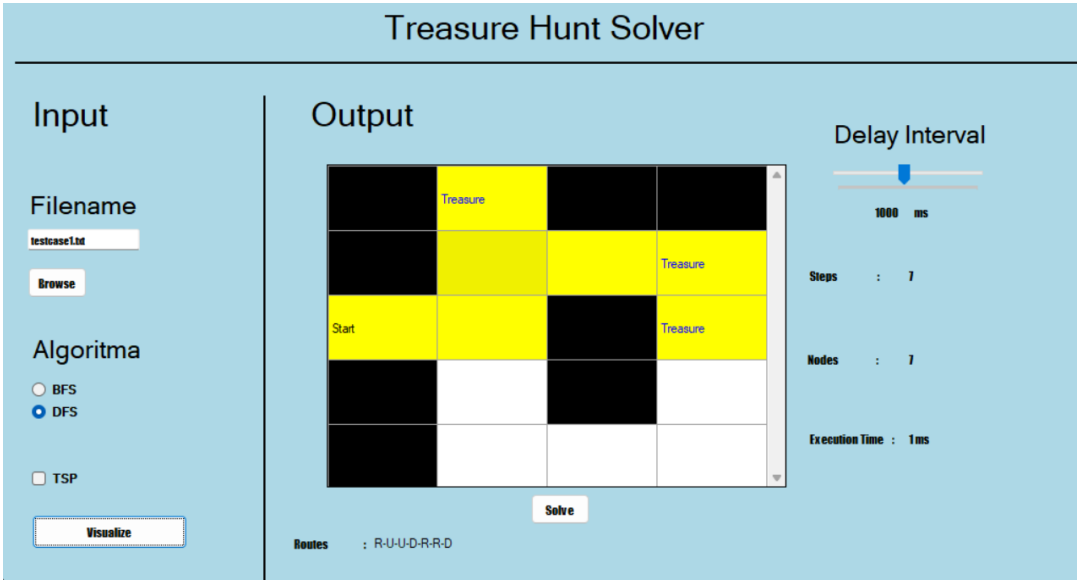
1. *Test Case* 1

Gambar 4.1.1. Masukan *file maze* untuk *test case* 1



Gambar 4.1.2. Hasil dan langkah penelusuran *test case* 1 dengan algoritma BFS

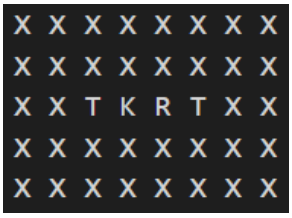
Untuk *test case* 1, algoritma BFS melakukan kunjungan terhadap 26 simpul atau *nodes* selama proses pencariannya. Rute solusi yang ditemukan untuk menemukan semua harta karun adalah R-U-U-D R-R-D dengan total 7 langkah.



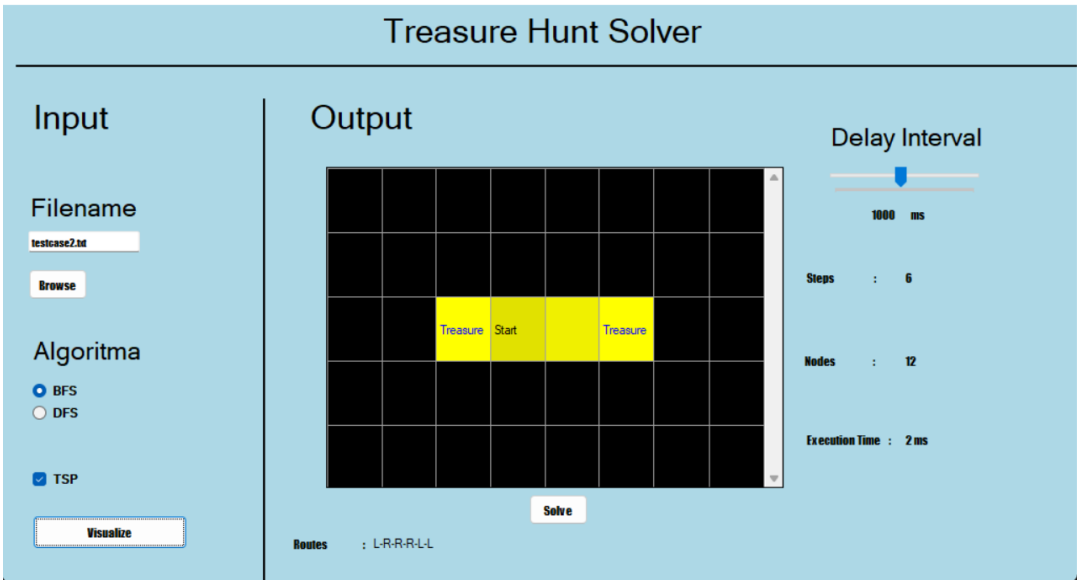
Gambar 4.1.3. Hasil dan langkah penelusuran *test case* 1 dengan algoritma DFS

Untuk *test case* 1, algoritma DFS melakukan kunjungan terhadap 7 simpul atau *nodes* selama proses pencariannya. Rute solusi yang ditemukan untuk menemukan semua harta karun adalah R-U-U-D R-R-D dengan total 7 langkah.

2. *Test Case* 2

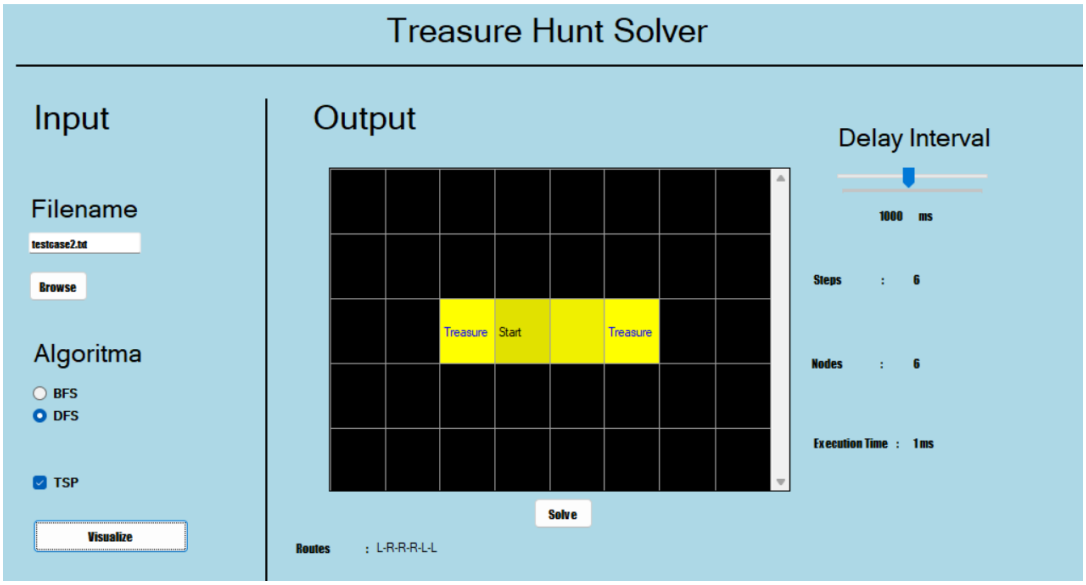


Gambar 4.2.1. Masukan *file maze* untuk *test case* 2



Gambar 4.2.2. Hasil dan langkah penelusuran *test case* 2 serta TSP dengan algoritma BFS

Untuk *test case* 2, algoritma BFS melakukan kunjungan terhadap 12 simpul atau *nodes* selama proses pencariannya. Rute solusi yang ditemukan untuk menemukan semua harta karun dan kembali ke posisi awal atau *start* adalah L-R-R-R-L-L dengan total 6 langkah.

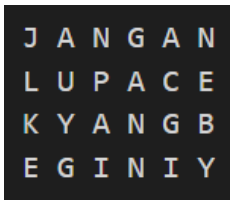


Gambar 4.2.3. Hasil dan langkah penelusuran *test case* 3 serta TSP dengan algoritma DFS

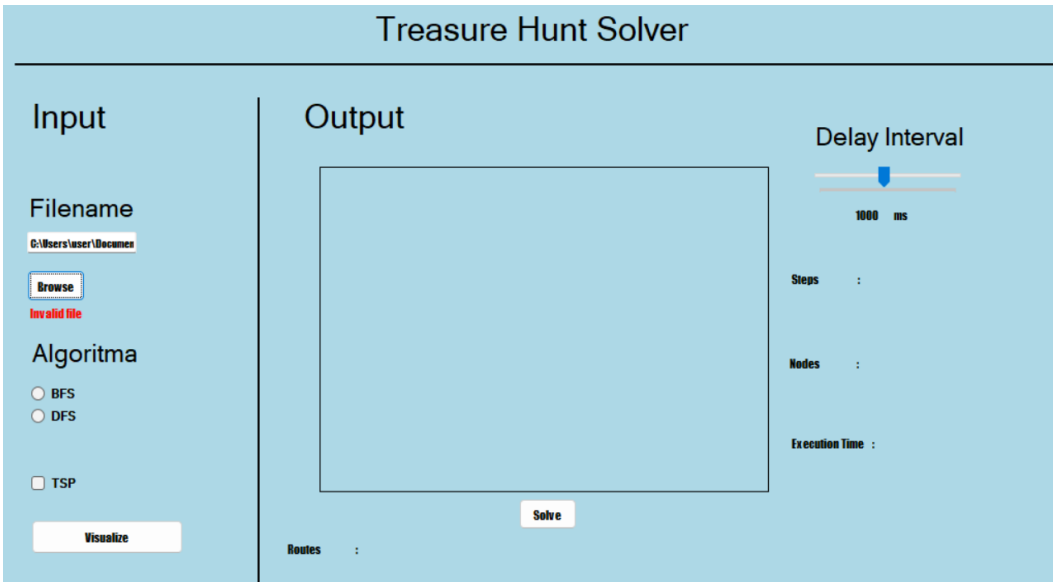
Untuk *test case* 2, algoritma DFS melakukan kunjungan terhadap 6 simpul atau *nodes* selama proses pencariannya. Rute solusi yang ditemukan untuk

menemukan semua harta karun dan kembali ke posisi awal atau *start* adalah L-R-R-R-L-L dengan total 6 langkah.

3. ***Test Case* 3**



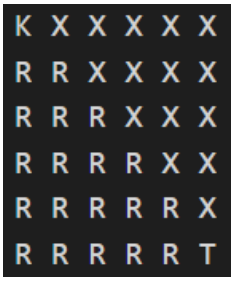
Gambar 4.3.1. Masukan *file maze* untuk *test case* 3



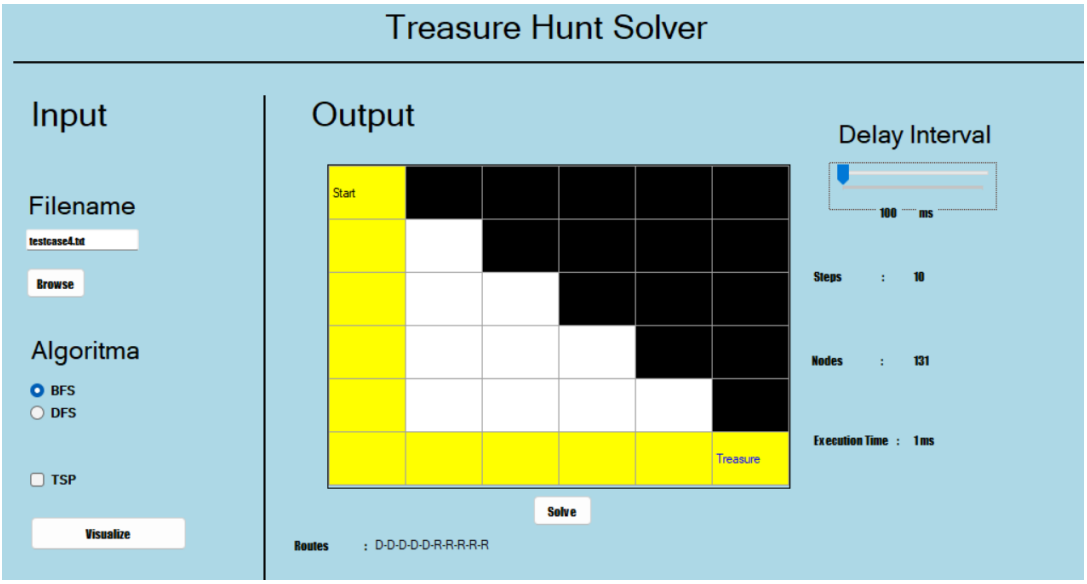
Gambar 4.3.2. Pesan *error* berupa "*Invalid file*" muncul di layar

Pada *test case* 3, ditampilkan pesan *error* yang menunjukkan terjadinya kegagalan ketika program sedang melakukan *parsing* terhadap file *input*. Kegagalan terjadi karena file mengandung simbol selain K, R, T, dan X.

4. ***Test Case* 4**

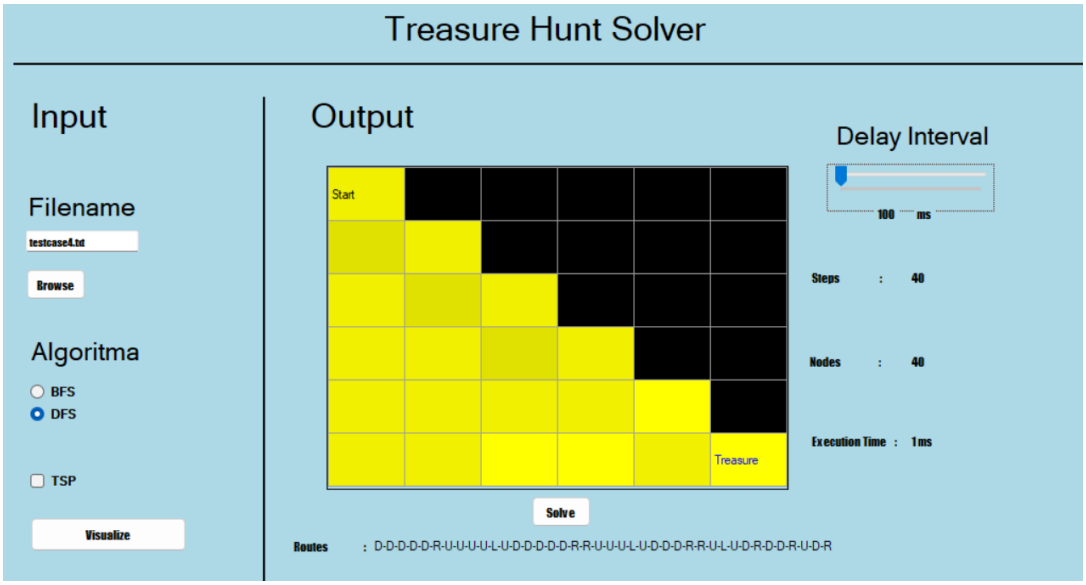


Gambar 4.4.1. Masukan *file maze* untuk *test case* 4



Gambar 4.4.2. Hasil dan langkah penelusuran *test case* 4 dengan algoritma BFS

Untuk *test case* 4, algoritma BFS melakukan kunjungan terhadap 131 simpul atau *nodes* selama proses pencariannya. Rute solusi yang ditemukan untuk menemukan semua harta karun adalah D-D-D-D-D-R-R-R-R-R dengan total 10 langkah.

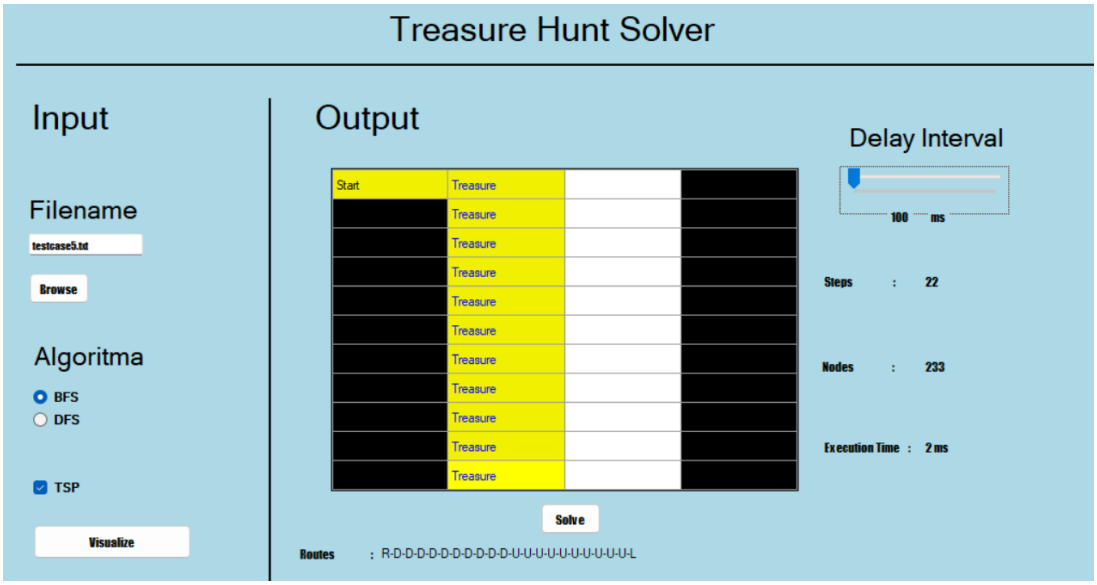


Gambar 4.4.3. Hasil dan langkah penelusuran *test case* 4 dengan algoritma DFS

Untuk *test case* 4, algoritma BFS melakukan kunjungan terhadap 40 simpul atau *nodes* selama proses pencariannya. Rute solusi yang ditemukan untuk menemukan semua harta karun adalah D-D-D-D-D-R-U-U-U-U-L-U-D-D-D-D-D-R-R-U-U-U-L-U-D-D-D-R-R-U-L-U-D-R-D-D-R-U-D-R dengan total 40 langkah.

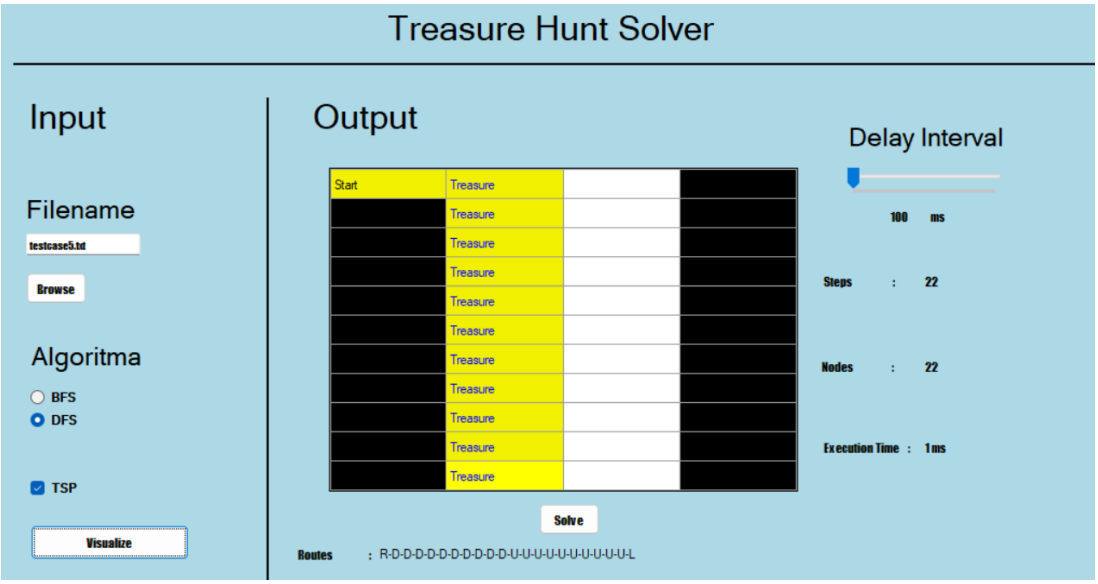
5. *Test Case* 5

Gambar 4.5.1. Masukan *file maze* untuk *test case* 5



Gambar 4.5.2. Hasil dan langkah penelusuran *test case* 5 serta TSP dengan algoritma BFS

Untuk *test case* 5, algoritma BFS melakukan kunjungan terhadap 233 simpul atau *nodes* selama proses pencariannya. Rute solusi yang ditemukan untuk menemukan semua harta karun dan kembali ke posisi awal atau *start* adalah R-D-D-D-D-D-D-D-D-D-D-D-U-U-U-U-U-U-U-U-U-U-L dengan total 22 langkah.

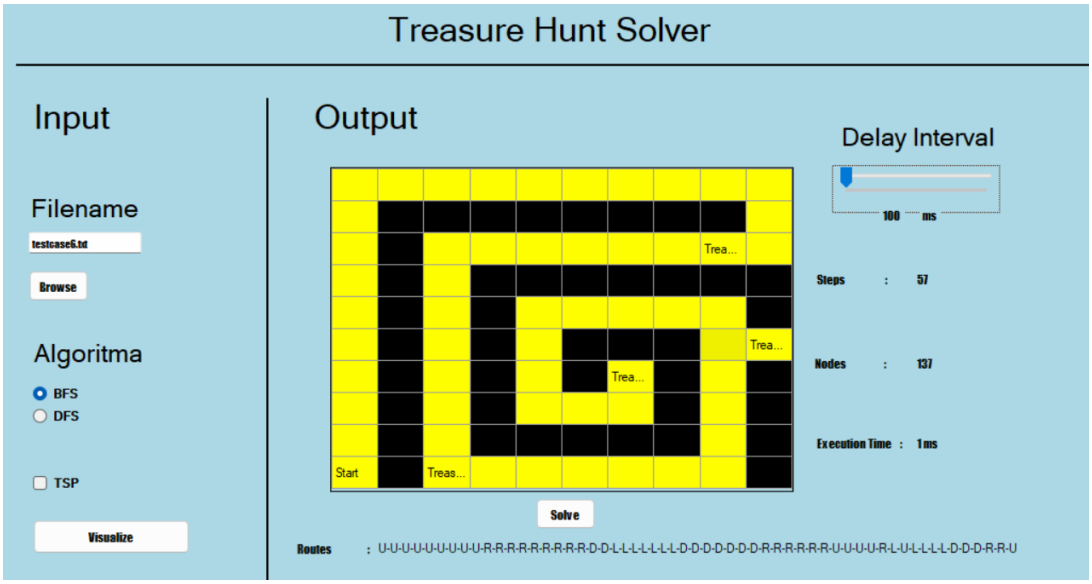


Gambar 4.5.3. Hasil dan langkah penelusuran *test case* 5 serta TSP dengan algoritma DFS

Untuk *test case* 5, algoritma DFS melakukan kunjungan terhadap 22 simpul atau *nodes* selama proses pencariannya. Rute solusi yang ditemukan untuk menemukan semua harta karun dan kembali ke posisi awal atau *start* adalah R-D-D-D-D-D-D-D-D-D-D-U-U-U-U-U-U-U-U-U-U-L dengan total 22 langkah.

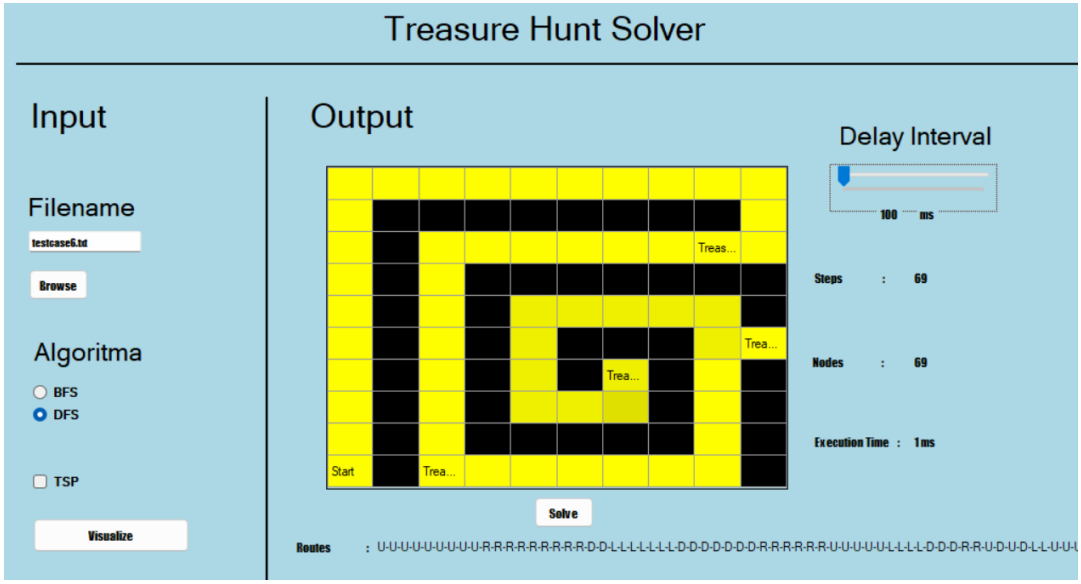
6. *Test Case* 6 (Tambahan)

Gambar 4.6.1. Masukan *file maze* untuk *test case* 6



Gambar 4.6.2. Hasil dan langkah penelusuran *test case* 6 dengan algoritma BFS

Untuk *test case* 6, algoritma BFS melakukan kunjungan terhadap 137 simpul atau *nodes* selama proses pencariannya. Rute solusi yang ditemukan untuk menemukan semua harta karun adalah U-U-U-U-U-U-U-U-U-R-R-R-R-R-R-R-R-R-D-D-L-L-L-L-L-L-L-D-D-D-D-D-D-D-R-R-R-R-R-R-U-U-U-U-R-L-U-L-L-L-L-L-D-D-D-R-R-U dengan total 57 langkah.



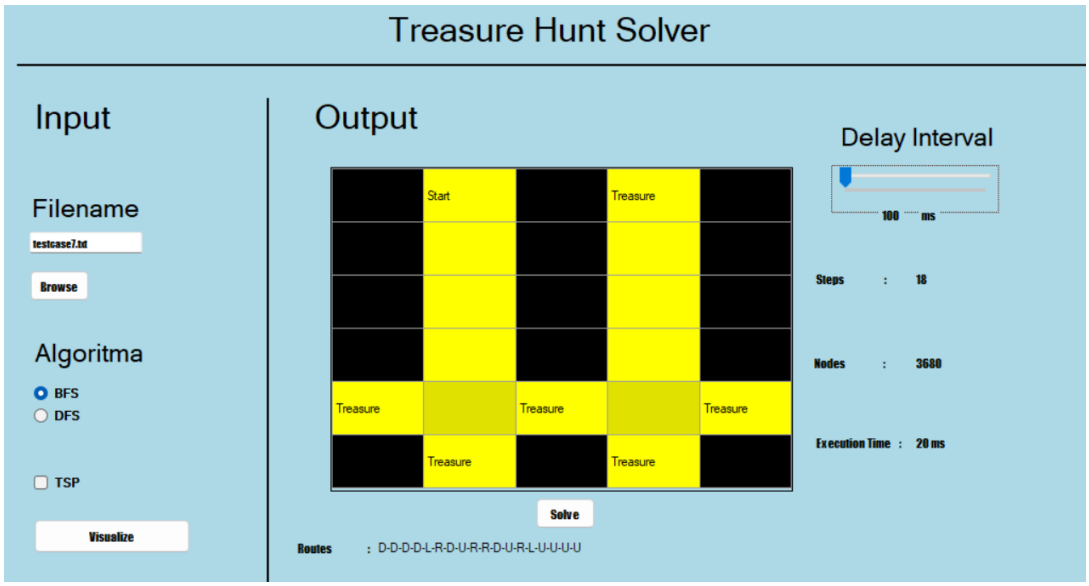
Gambar 4.6.3. Hasil dan langkah penelusuran *test case* 6 dengan algoritma DFS

Untuk *test case* 6, algoritma DFS melakukan kunjungan terhadap 69 simpul atau *nodes* selama proses pencariannya. Rute solusi yang ditemukan untuk menemukan semua harta karun adalah U-U-U-U-U-U-U-U-U-R-R-R-R-R-R-R-R-R

-R-D-D-L-L-L-L-L-L-L-D-D-D-D-D-D-D-R-R-R-R-R-R-U-U-U-U-U-L-L-L-L-D-D-D-R-R-U-D-U-D-L-L-U-U-U-R-R-R-R-D-R dengan total 69 langkah.

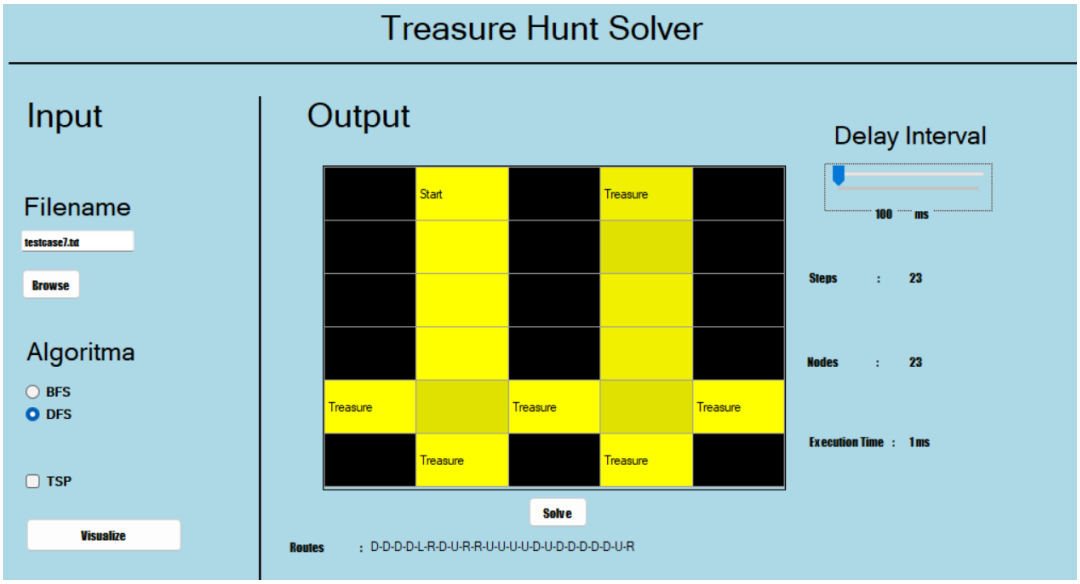
7. ***Test Case* 7 (Tambahan)**

Gambar 4.7.1. Masukan *file maze* untuk *test case* 7



Gambar 4.7.2. Hasil dan langkah penelusuran *test case* 7 dengan algoritma BFS

Untuk *test case* 7, algoritma BFS melakukan kunjungan terhadap 3680 simpul atau *nodes* selama proses pencariannya. Rute solusi yang ditemukan untuk menemukan semua harta karun adalah D-D-D-D-L-R-D-U-R-R-D-U-R-L-U-U-U-U dengan total 18 langkah.



Gambar 4.7.3. Hasil dan langkah penelusuran *test case* 7 dengan algoritma DFS

Untuk *test case* 7, algoritma DFS melakukan kunjungan terhadap 23 simpul atau *nodes* selama proses pencariannya. Rute solusi yang ditemukan untuk menemukan semua harta karun adalah D-D-D-D-L-R-D-U-R-R-U-U-U-U -D-U-D-D-D-D-D-U-R dengan total 23 langkah.

E. Analisis Desain Solusi Algoritma BFS dan DFS

Berdasarkan desain solusi yang dijabarkan dari Bab III hingga proses pengujian yang dilakukan pada Bab IV bagian D, diperoleh bahwa solusi pencarian menggunakan Algoritma DFS lebih sangkil dari segi kompleksitas ruang. Hal ini disebabkan karena solusi yang dibangun menggunakan algoritma DFS merupakan solusi satu jalur yang sekaligus merupakan solusi dari jalur yang diharapkan, terlepas dari karakteristik DFS yang menjadi semu karena adanya kemungkinan 1 petak dapat dilalui lebih dari 1 kali karena harus membentuk 1 rute sehingga skema runut-balik kurang terimplementasi dengan baik. Jumlah simpul yang perlu dikunjungi pun lebih sedikit karena setiap jalur hanya perlu kembali ke percabangan rute (implementasi pada DFS menggunakan prinsip runut-balik) hingga semua harta karun dapat tercapai dalam satu rute tersebut.

Hal yang membuat solusi menggunakan algoritma BFS lebih tidak sangkil adalah karena diperlukannya sebuah struktur data antrian untuk menyimpan rute yang akan dianalisis berikutnya. Implementasi ini membuat alokasi memori yang diperlukan untuk

merealisasikannya lebih besar sehingga tidak ramah dari segi kompleksitas ruang. Akan tetapi, pencarian solusi menggunakan algoritma BFS lebih mangkus karena menjamin adanya solusi terpendek akibat proses pencarian dilakukan pada semua rute yang mungkin. Jika ditinjau dari aspek waktu eksekusi, secara umum tidak jauh berbeda karena proses penelusuran simpul jelajah dilakukan satu demi satu secara terstruktur (untuk algoritma DFS) dan konkuren (untuk algoritma BFS).

Pada beberapa kasus pengujian pada bab IV bagian D, umumnya algoritma BFS harus mengunjungi lebih banyak simpul atau *node* dibandingkan dengan algoritma DFS. Namun algoritma BFS secara umum memberikan rute solusi yang lebih pendek dan efisien dibandingkan dengan algoritma DFS. Perbedaan ini semakin terlihat jelas ketika peta memiliki banyak simpul percabangan, dimana algoritma BFS harus mengunjungi *node* yang sangat banyak, sedangkan algoritma DFS memberikan rute solusi yang lebih panjang. Hal ini dapat terjadi karena sifat algoritma BFS yang melakukan pencarian secara melebar terlebih dahulu. Pada petak dengan percabangan yang, hal ini menyebabkan algoritma BFS harus terlebih dahulu mengunjungi semua cabang yang mungkin sebelum masuk ke kedalaman berikutnya dalam pohon ruang status pencarian. Untuk setiap cabang tersebut, algoritma BFS juga harus melakukan percabangan kembali. Sedangkan pada algoritma DFS, petak dengan banyak percabangan memungkinkan algoritma DFS untuk berputar-putar pada simpul yang telah dikunjungi sebelum akhirnya menemukan simpul yang belum dikunjungi. Hal ini menyebabkan rute solusi yang lebih panjang dibandingkan dengan algoritma BFS.

BAB V
KESIMPULAN DAN SARAN

A. Kesimpulan

Algoritma *Breadth First Search* (BFS) dan *Depth First Search* (DFS) merupakan dua jenis algoritma sederhana yang banyak digunakan untuk memecahkan berbagai masalah yang berkaitan dengan pencarian solusi ataupun penelusuran graf dan pohon. Dalam tugas besar ini, kedua algoritma dimanfaatkan untuk mencari suatu solusi berbentuk rute yang menemukan seluruh harta karun yang terdapat pada peta. Algoritma BFS digunakan dengan mencari rute secara melebar, sedangkan algoritma DFS mencari rute secara mendalam. Dapat disimpulkan bahwa kedua algoritma memiliki kelemahan dan kelebihan tersendiri dalam memecahkan permasalahan. Algoritma BFS memiliki kelebihan dimana dapat menemukan rute solusi yang efisien dan paling pendek untuk memecahkan masalah, namun membutuhkan memori yang lebih besar. Di sisi lain, algoritma DFS lebih sangkil untuk menemukan rute solusi dibandingkan algoritma BFS, serta tidak membutuhkan memori yang besar. Namun, rute solusi algoritma DFS belum tentu merupakan rute solusi yang efektif dan singkat untuk memecahkan permasalahan. Kedua algoritma memiliki kelebihannya masing-masing dan dapat digunakan sesuai dengan kebutuhan permasalahan yang ada.

B. Saran

Pengembangan sebuah *desktop application* pada tugas besar kali ini tentu saja tidak lepas dari adanya kekurangan batasan pengembangan. *Windows Form Application* sendiri, sesuai dengan namanya, hanya mendukung proses pengembangan sebuah *desktop application* pada sistem operasi Windows. Meskipun tidak menghambat pengembangan program dari kelompok kami karena semua menggunakan sistem operasi Windows, tetapi hal ini kemudian membuat program menjadi tidak *versatile.* Selain itu, algoritma pencarian BFS dan DFS juga dapat ditingkatkan dengan menambahkan beberapa metode heuristik ataupun dengan menggunakan algoritma selain BFS dan DFS untuk memecahkan permasalahan yang diberikan. Oleh sebab itu, kedepannya disarankan untuk dikembangkan pada bahasa pemrograman atau pengembang aplikasi desktop yang berbeda, secara khusus yang dapat

membuat aplikasi dapat dikembangkan pada sistem operasi lain. Peningkatan metode heuristik juga dapat dilakukan agar eksekusi program menjadi lebih mangkus dan sangkil.

C. Refleksi

Proses pengimplementasian program, secara umum, berjalan dengan lancar. Walaupun sempat terkendala dengan jumlah waktu yang relatif minimum, tetapi dengan koordinasi yang baik dan menyempatkan diri untuk melakukan pertemuan baik secara daring maupun luring, tugas besar ini dapat selesai dengan cukup baik. Beberapa poin yang menjadi refleksi dari tugas besar ini adalah :

1. Pemilihan aturan penamaan yang seragam membuat kode menjadi lebih intuitif.
2. Penggunaan komentar (*comment*) dalam jumlah banyak sangat membantu dalam memahami program dan langkah dari setiap kode yang diimplementasikan.
3. Desain algoritma yang jelas dan runtut sejak awal perencanaan membuat proses implementasi kode menjadi lebih terstruktur dan tidak terdapat kendala yang cukup berarti selama proses pembuatan dan berhasil tetap menjadi konvensi DRY (*Don't Repeat Yourself*).
4. Enkapsulasi menjadi hal yang penting dalam pembuatan pemrograman berorientasi objek. Berpikir sebagai objek membuat kami berhasil memahami bagaimana interaksi antar objek berlangsung pada saat *run-time*.
5. Pentingnya komunikasi yang baik antar anggota. Ini menjadi aspek yang tidak kalah penting dan menjadi salah satu kunci proses implementasi yang berlangsung dengan kendala yang minimal.
6. Pentingnya melakukan rekam cadang (*back-up*) bagi segmen kode yang memiliki signifikansi besar pada keseluruhan program. Salah satu kendala yang sempat kami hadapi adalah hilangnya bagian kode yang sangat penting tanpa memiliki rekaman yang terstruktur.
7. Penggunaan .gitignore menjadi penting dalam proses merapikan program dari *file* yang tidak sepenuhnya diperlukan bagi fungsionalitas program.

D. Tanggapan

Bisa disimpulkan tugas besar ini adalah tugas besar yang menarik. Terlepas dari minimumnya jumlah waktu yang tersedia untuk mengembangkan program dan membangun sebuah *desktop application* secara keseluruhan, tetapi ilmu yang diperoleh melalui tugas besar ini sangat membantu dalam memahami algoritma *Breadth First Search* (BFS) dan *Depth First Search* (DFS) serta cara membangun sebuah aplikasi desktop sederhana secara lebih baik dan menyeluruh. Akhir kata, semoga kedepannya ilmu yang didapat bisa berguna dan diimplementasikan bagi kehidupan di masa mendatang dan berguna bagi orang banyak.

BAB VI
DAFTAR PUSTAKA

Munir, R. (2023). Breadth/Depth First Search (BFS/DFS) (Bagian 1). IF2211 Strategi Algoritma - Semester II Tahun 2022/2023. Diakses pada 23 Maret 2023, pukul 13.57 dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf

Munir, R. (2023). Breadth/Depth First Search (BFS/DFS) (Bagian 2). IF2211 Strategi Algoritma - Semester II Tahun 2022/2023. Diakses pada 23 Maret 2023, pukul 14.23 dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf

Cormen, Thomas H. dkk. (2022). Introductions to Algorithm, 4th EditionMIT Press and McGraw-Hill. Tanggal 13 Maret 2023, pukul 23.48.

Imrona, Mahmud dan Rian Febrian Umbara. (2014). Matematika Deskret, Graph. School Of Computing, Telkom University. Diakses pada 22 Maret 2023, pukul 18.09 dari https://lmsspada.kemdikbud.go.id/mod/resource/view.php?id=47638

Lee, Terry G. dkk . (2023). Create a Windows Forms app in Visual Studio using C#. Microsoft. Diakses pada 6 Maret 2023, pukul 10.12 dari https://learn.microsoft.com/en-us/visualstudio/ide/create-csharp-winform-visual-studio?view=vs-2022

LAMPIRAN

Link Repository :

https://github.com/mikeleo03/Tubes2_MencariMaknaSpek

Link Video :

https://youtu.be/VUd5uzVgUdY