PENCARIAN RUTE PADA *MAZE TREASURE HUNT* DENGAN MEMANFAATKAN ALGORITME **BFS** DAN **DFS** DALAM BAHASA PEMROGRAMAN **C#**

Disusun untuk memenuhi laporan tugas besar mata kuliah IF2211 Strategi Algoritma semester 4 di Institut Teknologi Bandung.

Disusun oleh kelompok sebut_kok_stu:

Yanuar Sano Nur Rasyid (13521110)

Saddam Annais Shaquille (13521121)

Ryan Samuel Chandra (13521140)

PROGRAM STUDI TEKNIK INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

Jl. Ganesa No. 10, Lb. Siliwangi, Kecamatan Coblong,
Kota Bandung, Jawa Barat, 40132

2023

PRAKATA

Sebagai manusia yang dilengkapi dengan akal budi, kita sadar dunia bisa berkembang dengan pesat hingga saat ini karena tidak lepas dari bimbingan dan pertolongan Tuhan yang Maha Esa. Tidak henti-hentinya kami panjatkan syukur kepada-Nya, sebab pada akhirnya kami bisa mempersembahkan laporan ini tepat waktu, setelah bereksplorasi dan rutin melakukan kerja sama pemrograman selama tiga minggu terakhir ini. Kami mengucapkan terima kasih kepada Bapak Dr. Ir. Rinaldi Munir, M.T., juga Ibu Dr. Nur Ulfa Maulidevi, S.T, M.Sc., serta para asisten mata kuliah IF2211 Strategi Algoritma, yang telah membagikan ilmu serta memberikan kesempatan bagi kami untuk belajar lebih lagi tentang dunia informatika.

Dokumen ini kami beri judul "Pencarian Rute pada *Maze Treasure Hunt* dengan Memanfaatkan Algoritme BFS dan DFS dalam Bahasa Pemrograman C#". Laporan ini selain dibuat agar membawa manfaat bagi masyarakat, juga secara khusus disusun untuk memenuhi salah satu tugas besar mata kuliah IF2211 Strategi Algoritma di semester 4 Teknik Informatika Institut Teknologi Bandung.

Dengan usaha keras dalam mengumpulkan informasi terkait, serta uji coba program kolektif yang dibahas pun dilakukan secara saksama, kami mengeluarkan kemampuan terbaik untuk memenuhi tujuan pembuatan yang telah ditetapkan, meskipun berada di tengah tekanan waktu dan hambatan lingkungan.

Kami berharap percobaan kami ini dapat menjadi sesuatu yang memuaskan bagi semua pembaca. Tetapi, kami pun menyadari bahwa masih banyak kekurangan yang bisa diperbaiki. Maka dari itu, kami mohon kritik dan saran yang membangun untuk perkembangan percobaan kami. Terima kasih, selamat membaca.

Bandung, 24 Maret 2023

Penyusun Laporan

Daftar Isi

BAB 1
DESKRIPSI TUGAS

Gambar 1.1 Labirin di Bawah Krusty Krab
(Sumber: https://static.wikia.nocookie.net/theloudhouse/images/e/ec/Massive_Mustard_Pocket.png/revision/latest?cb=20180826170029)

Dalam tugas besar ini, mahasiswa 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 :

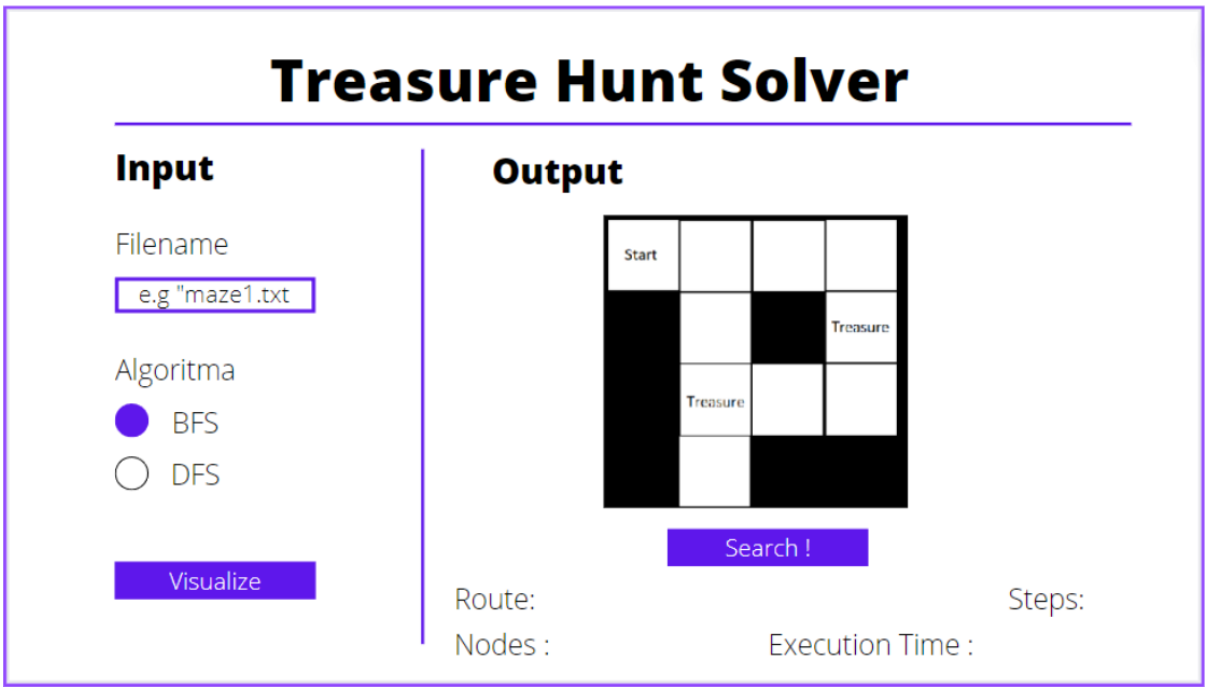
```
K R R R R R X
X R X T X R R
X T X R X R X
X R X X X T X
```

Gambar 1.2 Ilustrasi Masukan *Maze*
(Sumber: Dokumentasi Pribadi)

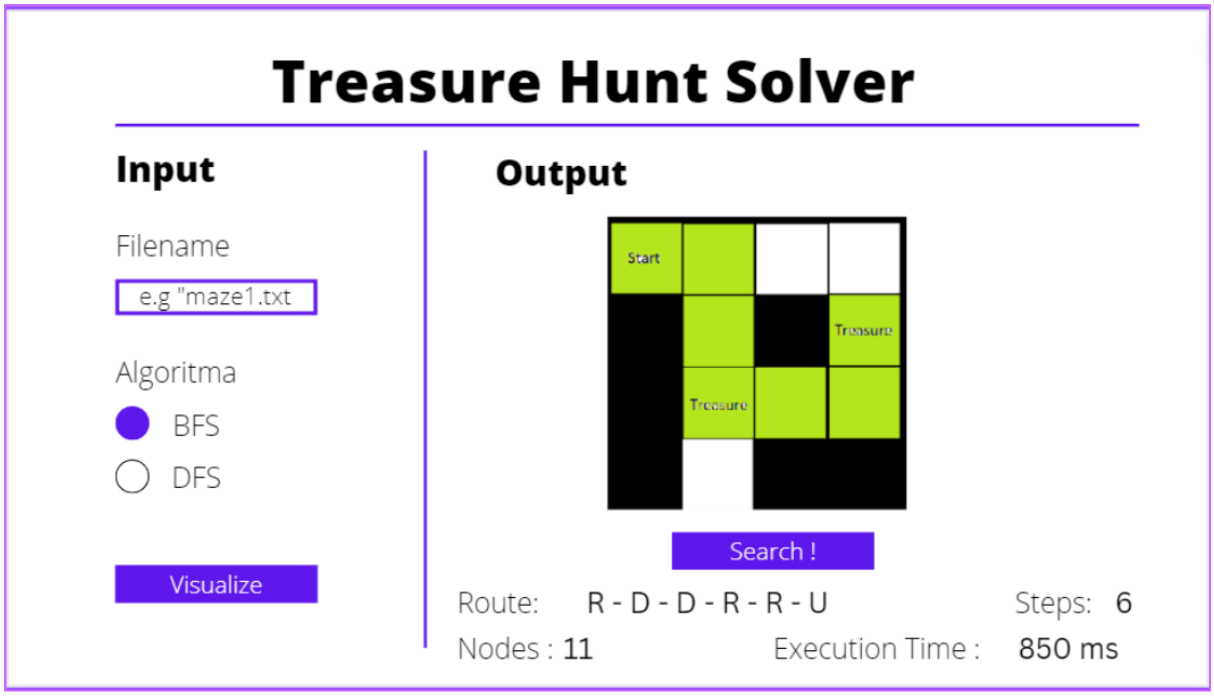
Dengan memanfaatkan algoritma *Breadth First Search* (BFS) dan *Depth First Search* (DFS), anda dapat menelusuri *grid* (simpul) yang mungkin dikunjungi hingga ditemukan rute solusi, baik secara melebar ataupun mendalam bergantung alternatif algoritma yang dipilih. Rute solusi adalah rute yang memperoleh 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. Mahasiswa juga diminta untuk memvisualisasikan input *txt* tersebut menjadi suatu *grid maze* serta hasil pencarian rute solusinya.

Spesifikasi Program:
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
(Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2022-2023/Tubes2-Stima-2023.pdf)



Gambar 1.4 Tampilan Program Setelah Dicari Solusinya
(Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2022-2023/Tubes2-Stima-2023.pdf)

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.

Program yang dibuat harus memenuhi spesifikasi wajib sebagai berikut:

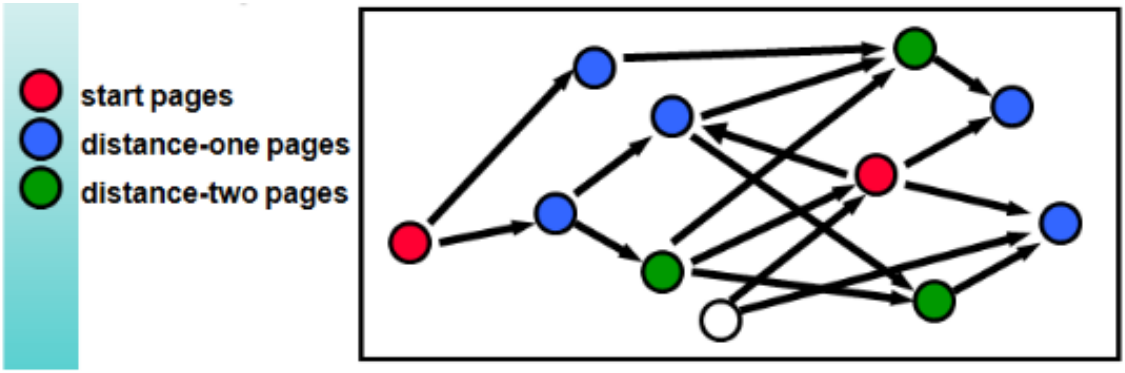
1. Membuat program dalam bahasa C# untuk mengimplementasi ***Treasure Hunt Solver*** sehingga diperoleh *output* yang diinginkan. Penelusuran harus memanfaatkan algoritma BFS dan DFS.
2. Awalnya program menerima *file* atau nama *file maze treasure hunt*.

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.
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.

Dikutip dari: Spesifikasi Tugas Besar 2 IF2211 2022/2023 (diakses tanggal 23 Maret 2023 dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2022-2023/Tubes2-Stima-2023.pdf)

BAB 2
LANDASAN TEORI

2.1 Traversal Graf



Gambar 2.1.1 Graf Jaringan *Web Page*
(Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf)

Graf adalah struktur data yang terdiri dari simpul atau *nodes* yang dihubungkan oleh *edge* atau sisi. Graf sering digunakan untuk merepresentasikan hubungan atau keterkaitan antara objek atau entitas dalam sebuah sistem. *Graph traversal* atau *traversal* graf adalah salah satu konsep yang digunakan untuk mengunjungi setiap simpul atau *nodes* pada sebuah graf secara sistematik.

Dalam algoritma traversal graf, terdapat dua jenis algoritma utama yaitu algoritma *Breadth First Search* (BFS) dan *Depth First Search* (DFS). Algoritma BFS digunakan untuk melakukan pencarian melebar pada graf, yaitu mengunjungi simpul-simpul pada level yang sama terlebih dahulu sebelum melanjutkan ke level berikutnya. Sedangkan algoritma DFS digunakan untuk melakukan pencarian mendalam pada graf, yaitu mengunjungi simpul-simpul pada jalur tertentu hingga habis sebelum kembali ke simpul awal dan melanjutkan ke jalur lainnya.

2.2 Algoritma *Breadth-First Search* (BFS)

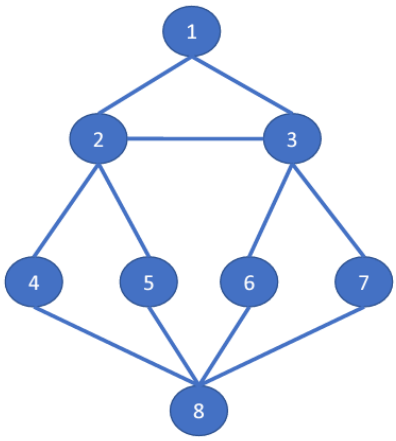
BFS atau biasa disebut sebagai pencarian melebar merupakan salah satu algoritma traversal graf yang sering digunakan dalam berbagai aplikasi. Algoritma ini memiliki tujuan utama untuk mencari jalur atau rute terpendek dari sebuah titik awal menuju titik tujuan pada sebuah graf.

Algoritma BFS akan mengunjungi semua simpul yang dapat dicapai dari titik awal terlebih dahulu sebelum melanjutkan ke simpul-simpul lainnya yang lebih jauh. Dalam hal ini, algoritma BFS termasuk ke dalam kategori *uninformed search* atau *blind search* karena tidak ada informasi tambahan yang diberikan selain struktur graf itu sendiri.

Algoritma BFS bekerja dengan cara memproses simpul satu per satu dengan mempertahankan urutan simpul yang belum diproses pada sebuah antrian. Setelah semua simpul yang terhubung dengan simpul awal telah diproses, algoritma akan melanjutkan ke simpul-simpul lainnya yang lebih jauh dan belum diproses.

Asumsikan pencarian dimulai dari simpul *v*. Maka pencarian solusi dengan algoritma BFS akan memiliki langkah-langkah sebagai berikut:

1. Algoritma BFS dimulai dengan mengunjungi simpul awal atau simpul *v*. Simpul *v* akan dijadikan simpul awal dari pencarian jalur terpendek.
2. Kemudian, semua simpul yang terhubung dengan simpul *v* secara langsung atau simpul-simpul yang bertetangga dengan simpul v akan dikunjungi setelahnya
3. Setelah dikunjungi, simpul ini akan ditandai sebagai simpul yang sudah dikunjungi.
4. Algoritma BFS akan mengunjungi simpul-simpul yang belum dikunjungi dan bertetangga dengan simpul-simpul yang sudah dikunjungi pada langkah kedua. Proses ini akan terus berlanjut hingga algoritma menemukan simpul tujuan atau simpul yang diinginkan.



Pada contoh gambar di samping, urutan simpul yang dicek secara BFS dari awal sampai akhir adalah:

1 – 2 – 3 – 4 – 5 – 6 – 7 – 8

Graf tersebut diasumsikan berjenis statis, yaitu sudah terbentuk sebelum proses pencarian dilakukan. Dengan demikian, graf statis dapat direpresentasikan sebagai sebuah struktur data.

Gambar 2.2.1 Contoh Graf Statis dengan Awal Simpul 1 (Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf)

Struktur data graf statis yang diperlukan untuk melakukan BFS adalah:

1. Matriks ketetanggaan $A = [a_{ij}]$ yang berukuran $n \times n$. a_{ij} bernilai 1 jika simpul *i* dan *j* bertetangga, dan bernilai 0 jika simpul *i* dan *j* tidak bertetangga.
2. Antrian (*Queue*) *Q* untuk menyimpan simpul yang telah dikunjungi.
3. Tabel *boolean*, diberi nama "dikunjungi" / "visited" (`array[1..n] of boolean`); dikunjungi[i] = *true* jika simpul *i* sudah dikunjungi, selebihnya bernilai *false*.

2.3 Algoritma *Depth-First Search* (DFS)

Depth First Search atau DFS adalah sebuah algoritma yang digunakan untuk melakukan traversal graf secara sistematis. Algoritma ini bekerja dengan cara mencari secara mendalam pada suatu cabang terlebih dahulu sebelum kemudian melanjutkan ke cabang berikutnya. Dengan menggunakan metode ini, DFS dapat mengeksplorasi setiap simpul pada graf hingga ke tingkat yang paling dalam sebelum kembali ke simpul awal dan melanjutkan pencarian ke simpul-simpul lainnya. Dalam melakukan pencarian, DFS menggunakan sebuah *stack* yang berfungsi untuk menyimpan simpul-simpul yang belum dieksplorasi sehingga nantinya dapat diambil kembali untuk dilakukan pengecekan lebih lanjut.

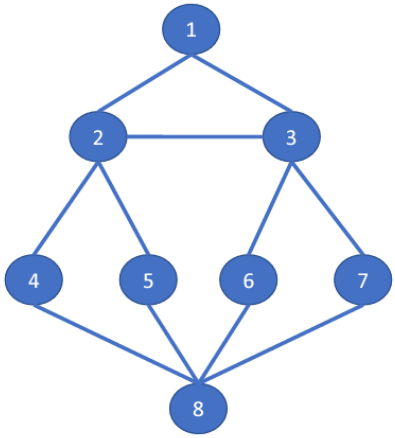
Algoritma DFS secara umum adalah sebagai berikut.

1. Mengunjungi simpul *root* atau asal.
2. Mengunjungi simpul, misal *a*, yang bertetangga dengan root.

3. Mengulangi langkah DFS dimulai dari simpul *a*.
4. Jika suatu simpul sudah tidak memiliki tetangga maka akan dilakukan *backtracking* (runut balik) ke simpul yang memiliki tetangga yang belum dikunjungi.
5. Pencarian berakhir ketika sesuatu yang dicari sudah ditemukan atau semua simpul sudah dikunjungi.

Untuk menerapkan algoritma tersebut, diperlukan struktur data yang sama dengan algoritma BFS, kecuali pada bagian *queue* sebagai tempat penyimpanan susunan simpul atau *nodes* yang akan dicari, diganti dengan menggunakan struktur data *stack*.

Contoh penerapan algoritma DFS:



Pada gambar di samping, yang mirip dengan contoh pada BFS, penelusuran simpul jika menggunakan algoritma DFS dengan prioritas memasuki angka yang lebih kecil terlebih dahulu akan menghasilkan susunan.

1 – 2 – 4 – 8 – 5 – 3 – 6 - 7

Gambar 2.3.1 Contoh Graf Statis dengan Awal Simpul 1 (Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf)

2.4 C# *Desktop Application Development*

C# adalah suatu bahasa pemrograman dari Microsoft yang digunakan untuk mengembangkan aplikasi *desktop*, *web*, dan *mobile*. WPF (Windows Presentation Foundation) adalah salah satu *framework* di .NET yang menggunakan C# untuk membuat aplikasi *desktop* di Windows. WPF menyediakan *user interface* yang dapat diisi dengan animasi dan efek visual. *User interface* dibuat menggunakan XAML, yaitu bahasa *markup* berbasis XML. Pembuatan *user interface* yang kompleks dapat dipermudah dengan penggunaan XAML.

Beberapa manfaat menggunakan C# dan WPF untuk pengembangan aplikasi desktop adalah sebagai berikut:

1. C# mudah dipelajari dan dipahami karena memiliki sintaks yang mirip dengan C maupun dengan C++.
2. WPF menyediakan *interface* yang mudah untuk dimodifikasi dan dapat ditambah efek visual sehingga aplikasi terlihat modern dan menarik.
3. WPF dapat menghasilkan aplikasi yang responsif sehingga dapat mengolah interaksi pengguna secara efisien.

Namun, di balik beberapa kelebihan yang telah disebutkan di atas, C# dan WPF memiliki beberapa kekurangan seperti berikut:

1. Aplikasi WPF hanya dapat berjalan di sistem operasi Windows sehingga pengguna dengan sistem operasi lain tidak dapat menggunakan *codebase* yang sama.

2. Pengembangan aplikasi C# dan WPF lebih mahal karena terdapat biaya lisensi untuk perangkat lunak seperti Visual Studio.

Berikut dijelaskan pula langkah-langkah dilaksanakan secara umum dalam melakukan pengembangan aplikasi *desktop* menggunakan C# dan WPF:

1. Buka aplikasi Visual Studio Code.
2. Pilih "*Create New Project*".
3. Pilih "WPF (*Windows Presentation Foundation*)" dan masukkan nama *project*.
4. Buka MainWindow.xaml dan tambahkan *interface* sesuai kebutuhan pengguna seperti teks, *button*, dan lain-lain.
5. Buat *handler function* pada C# untuk menangani *action* di *interface* pengguna.
6. Jalankan aplikasi untuk menguji *interface* dan fungsionalitas serta gunakan fitur *debug* milik Visual Studio untuk menemukan dan memperbaiki kesalahan pada kode C#.

BAB 3
ANALISIS PEMECAHAN MASALAH

3.1 Langkah Penyelesaian Masalah

Proses pemecahan masalah dimulai dengan memecah permasalahan utama yang kompleks menjadi upamasalah (*subproblem*) yang lebih mudah dan terstruktur. Dalam hal ini, permasalahan utama adalah mencari harta karun dalam labirin 2D dengan menggunakan algoritma BFS dan DFS.

Langkah awal dalam proses pemecahan masalah adalah memodelkan labirin dari *file* input menjadi sebuah graf. Proses ini dapat dilakukan dengan membaca input dari *file* eksternal yang berisi informasi lokasi awa, jalan, penghalang, dan harta karun dalam labirin. Setelah itu, informasi tersebut diubah menjadi graf dengan menghubungkan setiap titik dalam labirin yang dapat dilalui dengan tepat satu langkah. Pada graf ini, setiap titik jalan dan harta karun dalam labirin merupakan simpul graf dan setiap jalur yang dapat dilalui antara titik-titik tersebut merupakan sisi graf.

Setelah graf terbentuk, langkah selanjutnya adalah membuat algoritma BFS dan DFS untuk mencari harta karun pada graf tersebut. Algoritma BFS (*Breadth-First Search*) bekerja dengan cara menjelajahi simpul-simpul pada level yang sama terlebih dahulu sebelum mengeksplorasi simpul-simpul pada level di bawahnya. Sementara itu, algoritma DFS (*Depth-First Search*) bekerja dengan cara mengeksplorasi simpul-simpul sejauh mungkin pada satu jalur sebelum kembali ke simpul awal dan mengeksplorasi jalur lainnya. Pada program yang akan dibuat, prioritas simpul yang dipilih oleh algoritma untuk diperiksa terlebih dahulu adalah simpul sebelah kanan, bawah, kiri, dan atas.

Setelah algoritma BFS dan DFS selesai dibuat, langkah terakhir adalah membuat GUI menggunakan WPF sebagai aplikasi desktop untuk memudahkan pengguna dalam melihat hasil pencarian. GUI ini akan menampilkan labirin yang telah dimodelkan menjadi graf bersama dengan jalur yang telah ditemukan oleh algoritma BFS atau DFS. Selain itu, GUI juga akan menampilkan informasi mengenai langkah-langkah yang telah dilakukan oleh algoritma dalam menemukan harta karun.

3.2 Elemen-Elemen Algoritma BFS dan DFS

3.2.1 Breadth-First Search

Algoritma ini berada dalam kelas “MazeBFS”, menerima sebuah parameter berupa graf. Awalnya, ia akan memanggil dua buah *method* dari kelas “InputFile”, yaitu untuk mencari jumlah *treasure* dan titik awal pencarian. Tidak lupa, dicatat pula sebuah variabel untuk menyatakan jumlah *treasure* yang telah ditemukan.

Setelah itu, program akan membuat dua buah struktur data *queue* yang merupakan ciri khas utama dari BFS. Antrian pertama akan menyimpan satu per satu titik yang akan diperiksa, dan jalur menuju titik tersebut akan disimpan dalam antrian kedua. Setiap pemeriksaan sebuah titik akan menjadikan titik tersebut “dikunjungi”, sehingga tidak akan masuk lagi ke antrian meskipun menjadi simpul ekspan (tetangga dari simpul yang diperiksa).

Sementara kedua *queue* melakukan pemeriksaan menurut algoritma BFS pada umumnya, sebuah larik didefinisikan untuk secara kontinu menyimpan urutan pemeriksaan titik-titik. Hal ini menyebabkan fungsi bisa mengembalikan dua buah alur, yaitu alur nyata (menyambung dari titik awal sampai menemukan semua *treasure*) dan alur pemeriksaan (titik per titik yang diperiksa secara BFS; belum tentu terus terhubung).

Secara umum, BFS hanya bisa digunakan untuk mencari jalur menemukan satu tujuan saja. Maka dari itu, BFS harus dimodifikasi agar bisa memberikan jawaban untuk banyak *treasure* sekaligus. Perubahan besar-besaran dilakukan pada saat program berhasil menemukan satu dari sekian *treasure* yang ada, yaitu apabila belum semua harta karun terlacak, program seolah mengatur ulang semua simpul selain selain simpul *treasure* yang sudah dikunjungi, menjadi "belum dikunjungi", kemudian kedua *queue* dikosongkan. Algoritma BFS disambung kembali dengan simpul *treasure* yang baru diperiksa tersebut sebagai titik awalnya, tanpa menghapus jalur dari titik asal yang sebenarnya ke titik tersebut.

3.2.2 Depth-First Search

Algoritma ini menerima masukan berupa graf yang akan ditelusuri, titik awal pencarian sebagai simpul awal, dan jumlah *treasure* yang harus dicari. Pada awal pencarian, algoritma DFS akan memulai dari titik awal pencarian yang pertama-tama akan disimpan ke dalam larik rute. Selain itu, simpul tersebut ditandai dengan status "sudah dikunjungi". Setelah itu, program akan menyimpan simpul-simpul tetangga yang belum dikunjungi ke dalam sebuah stack.

Kemudian, secara rekursif akan dipanggil kembali algoritma DFS dengan titik baru yang didapat dari *stack*. Proses ini akan terus berulang sampai semua simpul tetangga sudah dikunjungi. Jika semua simpul tetangga sudah dikunjungi, algoritma akan melakukan backtrack menuju simpul sebelumnya yang memiliki tetangga yang belum terkunjungi.

Proses pencarian akan berakhir ketika semua *treasure* yang harus dicari telah ditemukan. Selama proses pencarian, algoritma DFS akan terus mencari simpul-simpul tetangga yang belum dikunjungi, sehingga dapat menemukan solusi dengan menggunakan pendekatan "depth-first" atau "mendalam" saat pencariannya.

3.3 Contoh Kasus

X R X X
K R R T
X R X X
X R R T

Pengecekan BFS: (1,0) → (1,1) → (1,2) → (2,1) → (0,1) → (1,3) → (1,2) → (1,1) → (2,1) → (1,0) → (0,1) → (3,1) → (3,2) → (3,3)
Solusi BFS: R – R – R – L – L – D – D – D – R

Pengecekan DFS: (1,0) → (1,1) → (1,3) → (2,1) → (3,1) → (3,2) → (3,3)
Solusi DFS: R – R – R – L – L – D – D – R – R

T R R T R X X R R X X R K R R T

Pengecekan BFS: (3,0) → (3,1) → (2,0) → (3,2) → (1,0) → (3,3) → (3,2) → (2,3) → (3,1) → (1,3) → (3,0) → (0,3) → (1,3) → (0,2) → (2,3) → (0,1) → (0,0)
Solusi BFS: R – R – R – U – U – U – L – L – L

Pengecekan DFS: (3,0) → (3,1) → (3,2) → (3,3) → (2,3) → (1,3) → (0,3) → (0,2) → (0,1) → (0,0)
Solusi DFS: R – R – R – U – U – U – L – L – L

BAB 4
IMPLEMENTASI DAN PENGUJIAN

4.1 Implementasi Program

4.1.1 *Function* "findPathBFS"

```
function findPathBFS(g : Graph) → array[0..1] of array of Point
{Mengembalikan sebuah List<List<Point>>, yaitu indeks 0 berisi jalur terhubung, dan indeks kedua berisi alur pemeriksaan dari titik-titik pada g secara BFS}

KAMUS
{Deklarasi Variabel}
num_T : integer
num_found : integer
QP : Queue[(jumlah nodes g) * 1000] of Point
QLP : Queue[(jumlah nodes g) * 1000] of array of Point
checkedPath, path, path1, pathOther : array of Point
otherTrash, pathEnqueue : array of Point
now, trash : Point
returnValue : array[0..1] of array of Point

ALGORITMA
if (jumlah node pada graf g != 0) then
    g.resetGraph() {memastikan semua simpul pada graf "belum dikunjungi"}
    num_T ← findNumberOfTreasure(g)
    num_found ← 0

    //INISIALISASI (PENGECEKAN PERTAMA)
    now ← findStartingPoint(g)
    now.Found ← true;  {set titik awal "sudah dikunjungi"}

    path ← path + now
    checkedPath ← checkedPath + now

    for (setiap titik di value adjecency list pada g dengan key=now)
        if (titik.Found == false) then
            masukAntrian (QP, titik)
            masukAntrian (QLP, path)

    //PENGECEKAN BERIKUTNYA
    while ((num_found < num_T) and (QP.isEmpty() = false)) do
        now ← keluarAntrian (QP)
        now.Found = true

        pathOther ← keluarAntrian (QLP)
        pathOther ← pathOther + now
        path = pathOther
        checkedPath ← checkedPath + now

        if (now.Type = TypeGrid.Treasure) then
            for (setiap simpul di g)
```

```
                if (p.Type != "Treasure") then
                    p.Found = false
            while (not isEmpty(QP)) do  {antrian belum kosong}
                trash ← keluarAntrian (QP)
                otherTrash ← keluarAntrian (QLP)
            num_found ← num_found + 1

        for (setiap titik di value adjecency list pada g dengan key=now)
            if (titik.Found = false) then
                masukAntrian (QP, titik)
                masukAntrian (QLP, path)
                pathEnqueue ← pathOther
                masukAntrian (QLP, pathEnqueue)

    returnValue ← returnValue + path
    returnValue ← returnValue + checkedPath
    → returnValue
else
    returnValue ← returnValue + path
    returnValue ← returnValue + checkedPath
    → returnValue
```

4.1.2 *Procedure* "dfs"

```
procedure dfs(Input source : Point, Input numOfTreasure : integer)
{I.S. Path terdefinisi atau berisi jalur tidak komplit
 F.S. Path menunjukkan jalur komplit asumsi semua Treasure memiliki jalan}

KAMUS
{Deklarasi Variabel}
pathStack : Stack
Path, pathVisited, backtrackPath, neighbours : List<Point>
G : Graph
Step, treasureCount : integer
Next : Point

{Deklarasi Fungsi dan Prosedur}
procedure pointFound()
{ I.S. Point terdefinisi
  F.S. Attribut found dari Point berubah menjadi True}

function getNeighboursNotVisited(input Point) -> int
{  Mengembalikan jumlah tetangga yang belum dikunjungi dari Input Point }

procedure addBacktrackPath(input Point)
{ I.S. backtrackPath terdefinisi
  F.S. backtrackPath berisi jalur menuju simpul sebelumnya yang memiliki
tetangga belum dikunjungi }
```

```
ALGORITMA
        {Menandakan Point telah ditemukan, menambah step, menyimpan Point}
        step <- step + 1
        source.pointFound()
        pathVisited.Add(source)
        path.Add(source)

        if (source.Type = TypeGrid.Treasure) then
            {Jika Point berjenis Treasure}
            treasureCount <- treasureCount + 1

        if (treasureCount = numOfTreasure) then
            {Jika semua treasure telah ditemukan}
            ->

        if (g.getNeighboursNotVisited(source) = 0) then
            {Jika  Point/simpul  tidak  memiliki  tetangga  yang  belum
dikunjungi}
            addBacktrackPath(source)
            {Melakukan backtrack ke Point yang belum dikunjungi}
            foreach (Point node in backtrackPath) do
                path.Add(node)
                step <- step + 1
            backtrackPath.Clear()
         else
            neighbours <- g.getNeighbours(source)
            {Menambah tetangga Point ke dalam stack}
            for (int i <- neighbours.Count-1; i >= 0; i <- i -1) do
                if (!neighbours[i].Found)
                    pathStack.Push(neighbours[i])

        if (pathStack.Count = 0) then
            ->
        {Mencari Point yang belum dikunjungi dari stack}
        Point next <- pathStack.Pop()
        do while (next.Found)
            if (pathStack.Count = 0) then
                ->
            next <- pathStack.Pop()
        {Rekursif ke Point selanjutnya}
        dfs(next, numOfTreasure)
```

4.2 Struktur Data dan Detail Program

4.2.1. *Class* "Point"

Kelas "Point" dibuat sebagai representasi dari setiap *grid* yang ada pada sebuah *map* atau peta. Representasi *point* terdiri dari dua buah angka yaitu X (≥ 0) dan Y (≥ 0) yang masing-masing menunjukkan posisi titik tersebut seolah berada dalam sebuah matriks. Misalnya posisi $X = 0$, $Y = 0$ (ditulis 0,0) menunjukkan *grid* yang berada di kiri atas. Perubahan Y menggeser lokasi pemantauan ke kanan/kiri, sementara perubahan X akan menggeser lokasi pemantauan ke atas/bawah.

Terdapat pula “enum TypeGrid” sebagai tipe dari *grid* yang mungkin muncul, yaitu “KK” sebagai Krusty Krab alias titik mulai pencarian, “Lintasan” sebagai jalur, “Treasure” sebagai *treasure*, dan “X” sebagai penanda halangan.
Kelas ini terdiri dari atribut:

1. `X : integer`
2. `Y : integer`
3. `Type : enum TypeGrid`
4. `Found : boolean`

Terdapat pula beberapa *method* yang didefinisikan, yaitu:

1. `pointFound()`
2. `resetPoint()`
3. `isLeft() -> boolean`
4. `isRight() -> boolean`
5. `isUp() -> boolean`
6. `isDonwn() -> boolean`
7. `isAdjecent() -> boolean`
8. `print()`

4.2.2. *Class* “Graph”

Kelas “Graph” merupakan sebuah struktur data yang digunakan sebagai data masukan bagi algoritma BFS ataupun DFS. *Nodes* atau simpul dalam graf ini berupa kumpulan Point dan representasi *edges* atau sisi penghubung antar-Point direpresentasikan dengan *adjecency list* dalam bentuk `Dictionary<Point, List<Point>>`.
Kelas ini memiliki beberapa atribut seperti

1. `Nodes : List<Point>`
2. `adjList : Dictionary<Point , List<Point>>`

Method-method penting dari kelas ini

1. `addEdge(Point source, Point destionation)`
2. `getNeighbours(Point Node) -> List<Point>`

4.2.3. *Class* “Queue”

Kelas “Queue” merupakan struktur data yang digunakan dalam algoritma BFS (*Breadth-First Search*). Queue adalah sebuah struktur data yang digunakan untuk menyimpan elemen dengan aturan *First In First Out* (FIFO), artinya elemen yang pertama kali dimasukkan akan menjadi elemen pertama yang keluar. Dalam algoritma BFS, Queue digunakan untuk menyimpan simpul yang akan dikunjungi selanjutnya. Proses penyimpanan simpul dalam Queue dilakukan dengan cara memasukkan semua simpul yang bertetanggan dengan simpul yang sedang diperiksa saat ini. Proses pengeluaran simpul dari Queue dilakukan Ketika suatu simpul selesai diperiksa.

Dengan menggunakan Queue, proses pencarian simpul dapat dilakukan dengan sistematis, yaitu dengan mengunjungi simpul-simpul yang terdekat terlebih dahulu sebelum mengunjungi simpul-simpul yang lebih jauh. Hal ini memungkinkan algoritma

BFS untuk menemukan jalur terpendek dari simpul awal ke simpul tujuan dalam sebuah graf.

4.2.4. *Data Structure* "Stack"

Pada algoritma DFS (*Depth-First Search*), struktur data yang digunakan untuk menyimpan simpul yang akan dikunjungi adalah Stack. Stack adalah struktur data yang digunakan untuk menyimpan elemen dengan aturan *Last In First Out* (LIFO), artinya elemen yang terakhir dimasukkan akan menjadi elemen pertama yang keluar. Dalam algoritma DFS, Stack digunakan untuk menyimpan simpul yang akan dikunjungi selanjutnya. Proses penyimpanan simpul dalam Stack dilakukan dengan cara memasukkan semua simpul yang bertetanggan dengan simpul yang sedang diperiksa saat ini. Proses pengeluaran simpul dari Stack dilakukan ketika suatu simpul telah selesai dikunjungi.

Dengan menggunakan Stack, proses pencarian simpul dilakukan secara vertikal atau menelusuri satu jalut graf terlebih dahulu sebelum menelusuri jalur yang lain. Hal ini memungkinkan algoritma DFS untuk menemukan jalur dari simpul awal ke simpul tujuan dengan lebih efisien namun jalur yang dihasilkan belum tentu yang terpendek.

4.2.5. Data Structure "Dictionary"

Struktur data Dictionary merupakan struktur data yang digunakan dalam pembuatan graf dengan representasi *adjacency list*. *Adjacency list* adalah salah satu metode yang digunakan untuk merepresentasikan graf, dimana setiap simpul dalam graf direpresentasikan sebagai *key* dalam sebuah dictionary dan *value* dari kunci tersebut adalah list yang berisi simpul-simpul terhubung dengan simpul tersebut.

Dengan menggunakan Dictionary, berbagai operasi dapat dilakukan pada graf dengan efisien, seperti mencari simpul tertentu, menambah atau menghapus simpul, mengecek apakah ada hubungan antara simpul-simpul tertentu, dll. Oleh karena itu, struktur data Dictionary sangat penting dalam pembuatan dan pengolahan graf yang menggunakan representasi *adjacency list*.

4.3 Tata Cara Penggunaan Program

Beberapa dependensi diperlukan untuk melakukan pengaturan program ini, antara lain:

1. .NET Framework dengan versi minimal 4.7.2
2. Sistem operasi Windows dengan versi minimal Windows 7
3. .NET Core dengan versi minimal 7.

Berikut adalah langkah-langkah untuk setup program:

1. Buka command line di direktori folder utama program.
2. Jalankan perintah "dotnet build".
3. Setelah proses build selesai, program dapat dijalankan dengan membuka file "TreasureHuntSolver.exe" yang terdapat pada direktori folder "bin\Release\net7.0-windows10.0.17763.0".

Untuk menggunakan program TreasureHuntSolver, langkah-langkahnya adalah sebagai berikut:

1. Buka aplikasi TreasureHuntSolver.
2. Pilih file yang berisi peta labirin dengan format yang sesuai melalui tombol "Open file".
3. Tentukan metode pencarian dengan memilih salah satu dari dua opsi yang tersedia: BFS atau DFS.
4. Sesuaikan kecepatan durasi tiap langkah dengan menggeser slider yang tersedia.
5. Tekan tombol "Search!" untuk memulai pencarian.

4.4 Hasil Pengujian

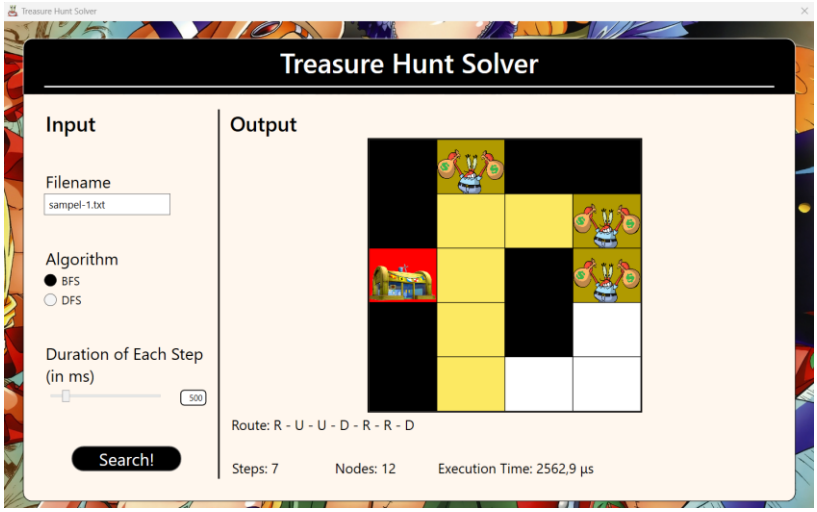
Catatan : Prioritas simpul yang dicari terlebih dahulu adalah kanan, bawah, kiri, atas

1. Pengujian 1

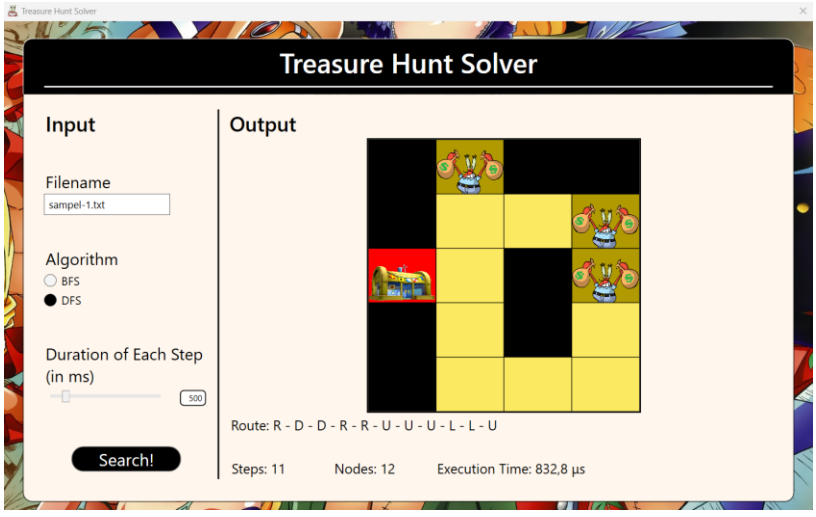
   Peta pengujian:

   X T X X
   X R R T
   K R X T
   X R X R
   X R R R

   Penelusuran menggunakan BFS:



   Penelusuran menggunakan DFS:

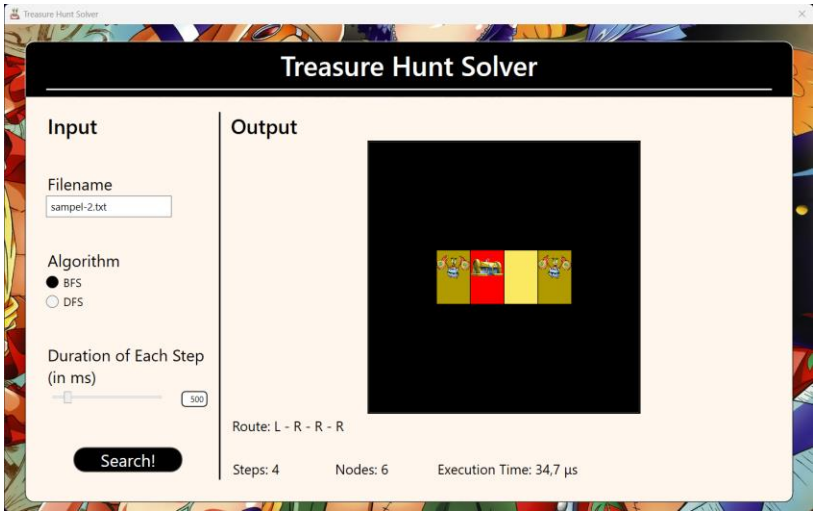


2. Pengujian 2

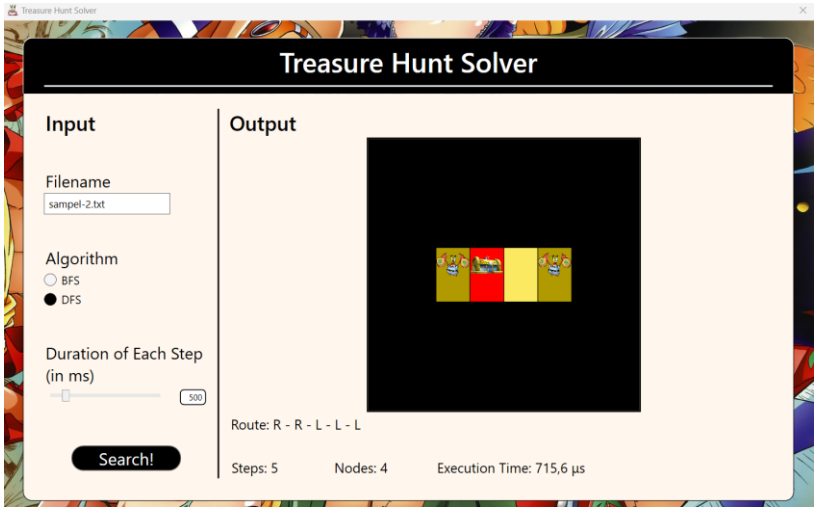
Peta pengujian:

```
X X X X X X X X
X X X X X X X X
X X T K R T X X
X X X X X X X X
X X X X X X X X
```

Penelusuran menggunakan BFS:



Penelusuran menggunakan DFS:



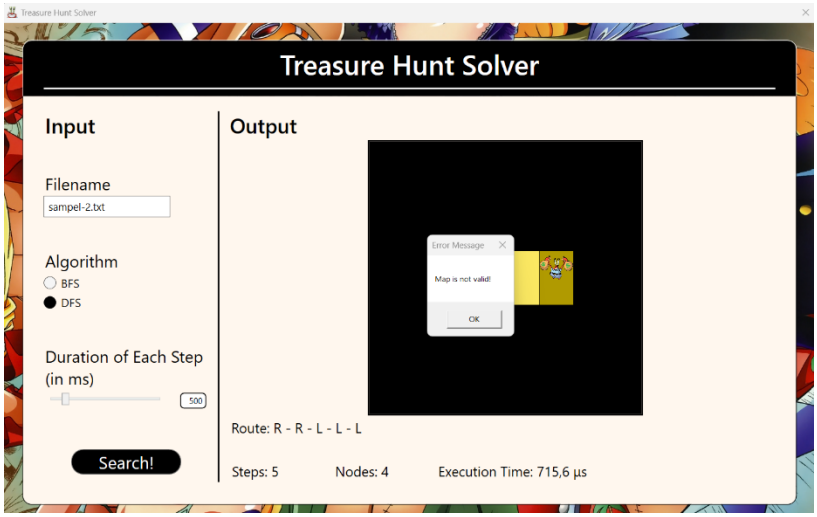
3. Pengujian 3

Peta pengujian:

```
J A N G A N
L U P A C E
K Y A N G B
E G I N I Y
```

Hasil :

Muncul *message box* bertuliskan “Map is not valid!” dan *file* tidak masuk ke program

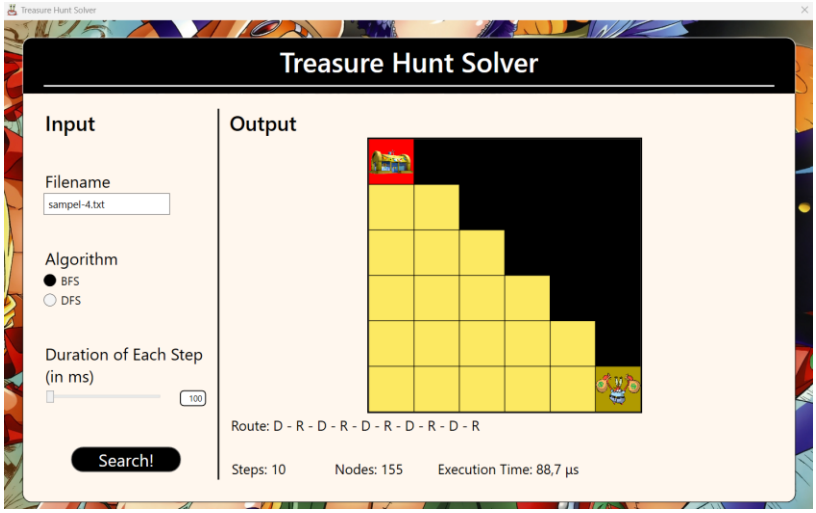


4. Pengujian 4

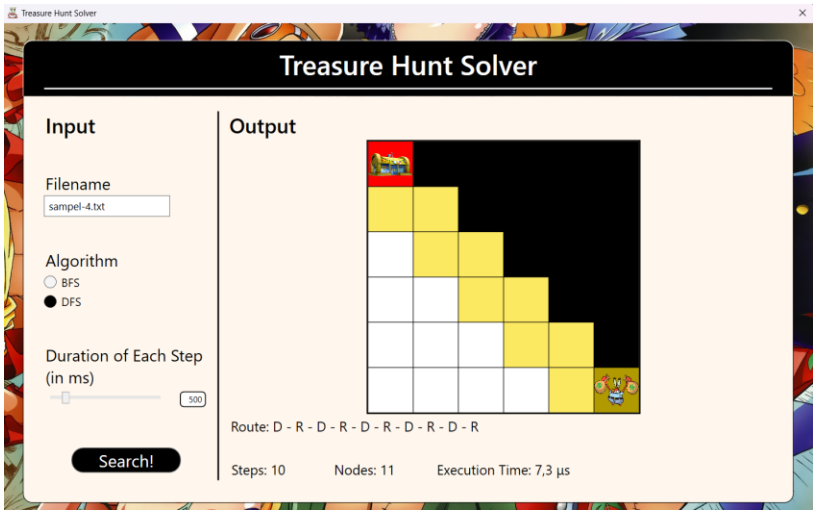
Peta pengujian:

```
K X X X X X
R R X X X X
R R R X X X
R R R R X X
R R R R R X
R R R R R T
```

Penelusuran menggunakan BFS:



Penelusuran menggunakan DFS:

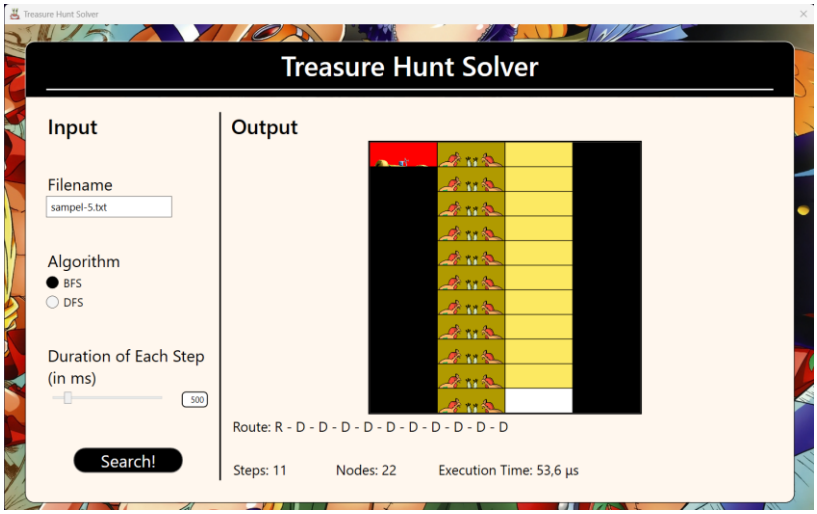


5. Pengujian 5

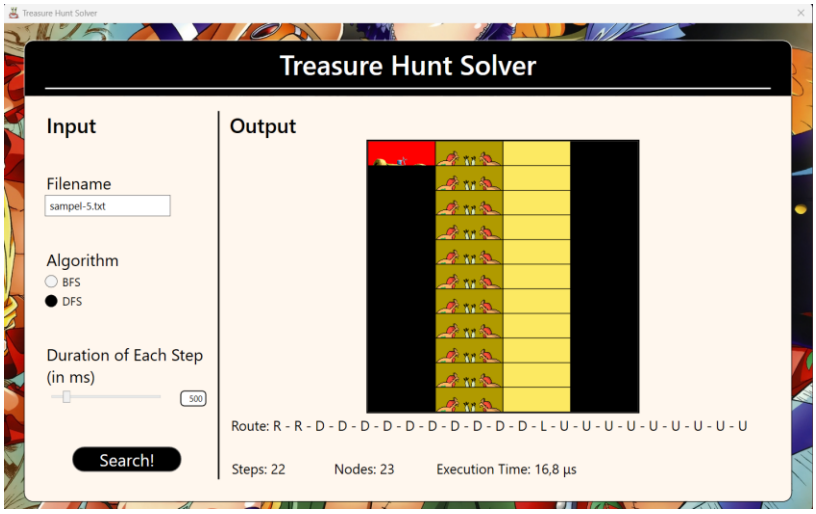
Peta pengujian:

```
K T R X
X T R X
X T R X
X T R X
X T R X
X T R X
X T R X
X T R X
X T R X
X T R X
X T R X
```

Penelusuran menggunakan BFS:



Penelusuran menggunakan DFS:

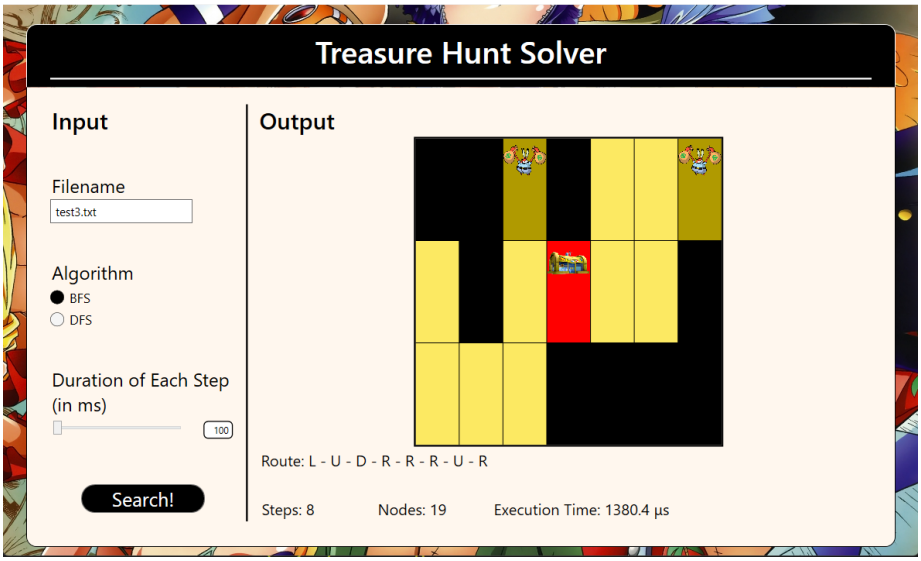


6. Pengujian 6

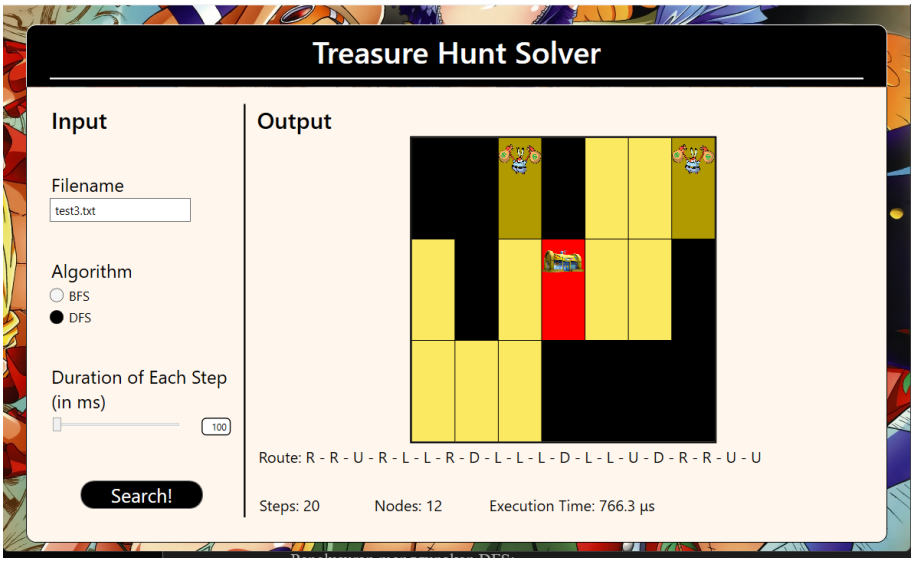
Peta Pengujian :

```
X X T X R R T
R X R K R R X
R R R X X X X
```

Penelusuran menggunakan BFS:



Penelusuran menggunakan DFS:

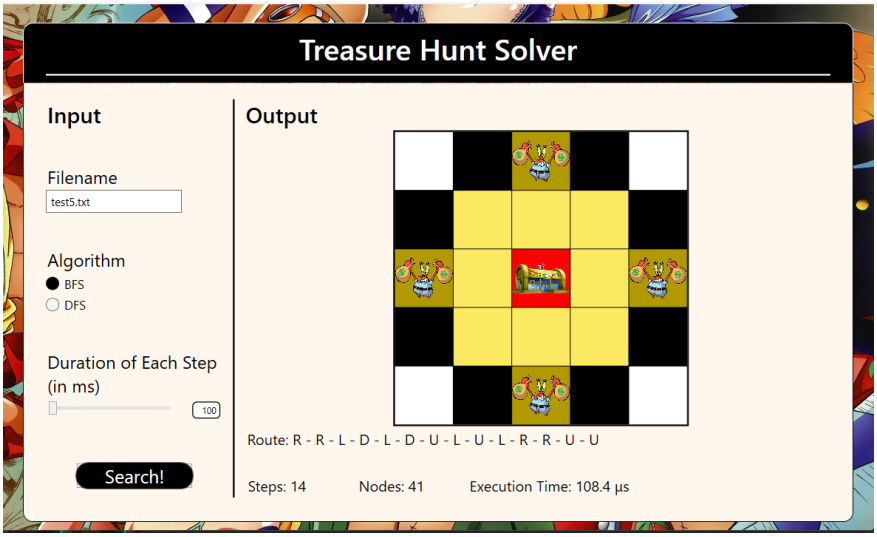


7. Pengujian 7

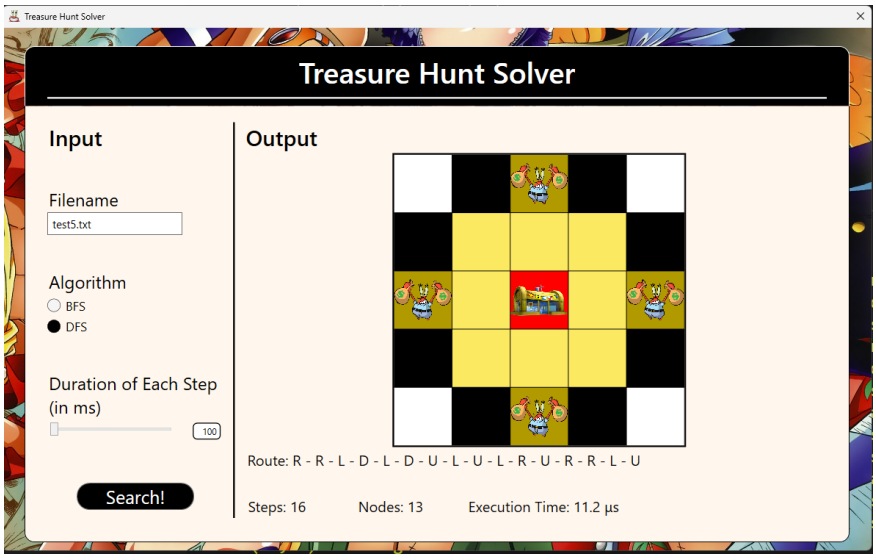
Peta Pengujian:

R X T X R
X R R R X
T R K R T
X R R R X
R X T X R

Penelusuran menggunakan BFS:



Penelusuran menggunakan DFS:



4.5 Analisis Desain Solusi

Berdasarkan pengujian yang telah dilakukan. Terdapat beberapa perbedaan mengenai hasil dari penerapan algoritma BFS dan DFS. Algoritma BFS melakukan pencarian secara melebar pada semua simpul yang terhubung pada simpul awal sehingga algoritma ini cenderung dapat menemukan rute yang lebih pendek dibandingkan DFS. Karena sifatnya yang memeriksa semua simpul yang bertetanggaan terlebih dahulu, jumlah simpul yang diperiksa pun menjadi lebih banyak. Jumlah simpul yang harus diperiksa menjadi sangat banyak seiring dengan bertambahnya jarak antara simpul awal dengan harta karun. Pada saat BFS melakukan pencarian, ia akan menelusuri semua simpul yang terhubung pada simpul awal terlebih dahulu sebelum melanjutkan ke simpul yang lebih jauh. Hal ini berarti BFS akan mengeksplorasi daerah sekitar titik awal terlebih dahulu sebelum mencari di daerah yang lebih jauh. Jadi, jika harta karun berada di dekat titik awal, maka BFS akan menemukannya dengan cepat. Oleh karena itu, BFS cocok untuk mencari harta karun jika harta karun tersebut dekat dengan titik awal.

Sementara itu, DFS atau *Depth-First Search* adalah algoritma yang melakukan pencarian secara mendalam pada setiap simpul yang terhubung pada simpul awal. Pada saat DFS melakukan pencarian, ia akan menelusuri simpul yang terhubung pada simpul awal hingga tidak ada lagi simpul yang terhubung sebelum melanjutkan ke simpul yang lebih jauh. Hal ini berarti DFS akan langsung menuju simpul yang terhubung dengan titik awal tanpa mengeksplorasi daerah sekitarnya terlebih dahulu. Oleh karena itu, jika harta karun berada di daerah yang jauh dari titik awal, DFS akan menemukannya dengan cepat dan lebih efisien.

BAB 5
PENUTUP

5.1 Kesimpulan

Traversal graf merupakan proses yang penting dalam *programming* untuk menelusuri setiap simpul dalam sebuah graf. Traversal graf terbagi menjadi dua tergantung dari susunan simpul yang akan dicari terlebih dahulu yaitu *Breadth-First Search* (BFS) dan *Depth-First Search* (DFS). Salah satu implementasi traversal graf adalah mencari rute pada sebuah peta yang memenuhi syarat mengunjungi petak-petak tertentu. Penentuan rutenya dapat dicari dengan algoritma BFS maupun DFS.

Algoritma BFS melakukan penyelesaian dengan menelusuri jalur secara lebar terlebih dahulu, yaitu dengan menelusuri seluruh simpul tetangga dari simpul asal sebelum melanjutkan ke simpul-simpul yang lebih dalam. Ketika mencari *treasure*, algoritma BFS akan mencari terlebih dahulu pada simpul-simpul yang dapat dijangkau dari simpul asal sebelum melanjutkan pencarian ke simpul yang lebih jauh. Jika terdapat jalan buntu, algoritma akan mencoba semua jalur yang masih tersedia sebelum menghentikan pencarian jika semua jalur sudah dijelajahi. Algoritma akan berhenti ketika semua *treasure* sudah ditemukan atau semua simpul/kotak sudah dikunjungi. Keuntungan dari algoritma BFS adalah dapat menemukan solusi terpendek dari suatu permasalahan, namun membutuhkan memori yang cukup besar jika graf yang digunakan memiliki jumlah simpul yang sangat besar.

Penyelesaian dengan algoritma DFS dilakukan dengan menelusuri sebuah jalur secara mendalam terlebih dahulu yaitu menelusuri salah satu simpul tetangga dari simpul asal sampai tidak ada tetangga lagi. Setelah itu, penelusuran akan dilanjutkan dengan menelusuri simpul lainnya secara mendalam lagi. Dalam permasalahan ini, algoritma DFS akan mencari treasure di setiap kotak dan jika mencapai jalan buntu maka penelusuran akan dilanjutkan dengan melakukan *backtracking* hingga mencapai rute yang memiliki jalan lain. Algoritma berhenti ketika semua *treasure* sudah ditemukan atau semua simpul/kotak sudah dikunjungi.

Perbedaan antara algoritma BFS dan DFS selain pada cara pemilihan simpul untuk penelusuran, untuk masalah ini, kedua algoritma tersebut memiliki perbedaan pada rute yang dihasilkan. Jika *treasure* pada map berada pada dekat area sekitar *krusty krab* tetapi tidak berada di satu jalur algoritma BFS akan menghasilkan rute yang lebih baik daripada DFS. Kasus selanjutnya adalah jika *treasure* berada jauh dari *krusty krab* dan berada di satu jalur algoritma DFS akan menghasilkan rute yang lebih baik daripada BFS. Selain itu, algoritma BFS akan menggunakan memori yang lebih banyak dibandingkan DFS karena algoritma BFS memeriksa semua tetangga terlebih dahulu yang akan bertambah seiring dengan banyaknya simpul yang diperiksa sedangkan DFS hanya memeriksa satu kemungkinan jalur terlebih dahulu.

5.2 Saran

1. Sebaiknya, ada sesi diskusi mendalam yang dilakukan bersama dengan seluruh anggota kelompok sebelum memulai pengerjaan tugas. Hal ini bertujuan untuk menyamakan persepsi tim, sehingga tidak ada bagian program yang tidak *connect* saat hasil implementasi digabungkan.

2. Akan lebih baik jika para pemrogram melakukan banyak komunikasi berkala seiring membangun algoritma. Dengan demikian, diharapkan tidak ada duplikasi fungsi umum, atau struktur program yang tidak seragam.
3. Seharusnya, waktu yang diberikan untuk pengerjaan tugas ini lebih banyak, dengan pertimbangan bahwa mahasiswa membutuhkan banyak sekali eksplorasi mandiri. Ditambah lagi, waktu yang tersedia terpotong oleh UTS selama satu minggu.
4. Penerapan konsep OOP yang sudah dipelajari dari matkul lain seharusnya bisa lebih dipakai sehingga pembuatan kode bisa lebih efisien.

5.3 Refleksi

Tugas besar kedua memberikan banyak ilmu baru dan juga memperkuat pemahaman yang sudah di pelajari dalam pemrograman dengan bahasa C# yang berbasis OOP. Meskipun hasil program yang dibuat belum maksimal, kami mempelajari implementasi OOP dalam bahasa baru yang baru pertama kali dipakai. Pengerjaan juga dapat menjadi lebih baik jika pembagian waktu untuk mengerjakan tugas ini dilakukan dengan lebih baik.

5.4 Tanggapan

Dulu kami masih bisa mengerjakan tugas-tugas besar yang bersisian, tetapi sekarang kami tidak mampu mengerjakan tugas besar yang beririsan satu sama lain. Seolah belum cukup menantang, UTS pun masih dibanjiri dengan tugas, termasuk tugas ini.

Memang pengalaman yang sungguh luar biasa, dapat bekerja dalam tim yang hebat. Namun rintangan demi rintangan yang tak terduga terus menghantui kami setiap hari, mematikan secercah harapan yang awalnya kami pandang pasti.

Kiranya apa pun yang kami telah kerjakan, segala jerih payah, tawa dan tangis, dan pengorbanan dapat membuahkan hasil yang baik suatu saat nanti.

LAMPIRAN

Pranala *Repository*:

https://github.com/RyanSC06/Tubes2_sebut_kok_stu.git

Pranala Video Youtube:

https://youtu.be/y8CyWllW46E

Tabel A Runut Pengerjaan Tugas

POIN	YA	TIDAK
1. Program berhasil dikompilasi tanpa ada kesalahan.	☑	
2. Program dapat menerima masukan dan menuliskan luaran.	☑	
3. Luaran program sudah benar (solusi labirin benar).	☑	
4. Bonus progres pencarian *grid* dikerjakan.	☑	
5. Bonus TSP dikerjakan.		☑
6. Bonus pembuatan video dikerjakan	☑	

DAFTAR REFERENSI

1. Munir, Rinaldi. 2021. *Breadth/Depth First Search (BFS/DFS) (Bagian 1)*. (Dikases pada tanggal 23 Maret 2023 dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf)
2. Munir, Rinaldi. 2021. *Breadth/Depth First Search (BFS/DFS) (Bagian 2)*. (Dikases pada tanggal 23 Maret 2023 dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf)
3. speak2rk09, dkk. 8 Februari 2023. *Depth First Search or DFS for a Graph*. (Diakses dari https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/ pada tanggal 23 Maret 2023).