

LAPORAN TUGAS BESAR 2
IF2211 - STRATEGI ALGORITMA

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

Kelompok BraveEagle69



Anggota Kelompok :

(13521061) Alex Sander

(13521097) Shidqi Indy Izhari

(13521106) Mohammad Farhan Fahrezy

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2022

Daftar Isi

Daftar Isi	2
Bab 1:	
Deskripsi Tugas	3
1.1 Latar Belakang	3
1.2 Deskripsi Tugas	4
Bab 2:	
Landasan Teori	6
2.1 Penjelasan Graf	6
2.2 Penjelasan Traversal Graf	6
2.3 Penjelasan Algoritma BFS	7
2.4 Penjelasan Algoritma DFS	8
Bab 3:	
Aplikasi Strategi BFS dan DFS	9
3.1 Langkah-langkah pemecahan masalah	9
3.2 Proses mapping persoalan menjadi elemen-elemen algoritma BFS dan DFS	9
3.3 Contoh ilustrasi kasus lain yang berbeda dengan contoh pada spesifikasi tugas	9
Bab 4:	
Analisis Pemecahan Masalah	10
4.1 Implementasi program	10
4.2 Penjelasan struktur data	10
4.3 Penjelasan tata cara penggunaan program	10
4.4 Hasil pengujian/Eksperimen	10
4.4 Analisis dari desain solusi algoritma BFS dan DFS yang diimplementasikan pada setiap pengujian yang dilakukan	10
Bab 5:	
Penutup	11
5.1 Kesimpulan	11
5.2 Saran	12
5.3 Refleksi	12
Daftar Pustaka	13
Lampiran	14

Bab 1:

Deskripsi Tugas

1.1 Latar Belakang

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



Gambar 1. Labirin di bawah Krusty Krab

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.

1.2 Deskripsi Tugas

Dalam tugas besar ini, penulis 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:

1. K = Krusty Krab (titik awal)
2. T = Treasure (harta karun yang ingin diambil)
3. R = Tile yang mungkin diakses/dilewati
4. X = Tile yang tidak mungkin diakses/dilewati

Dengan memanfaatkan algoritma Breadth First Search (BFS) dan Depth First Search (DFS), penulis 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. Penulis juga diminta untuk memvisualisasikan input file .txt menjadi suatu grid maze serta hasil pencarian rute solusinya. Cara visualisasi grid dibebaskan, sebagai contoh dalam bentuk matriks yang ditampilkan dalam GUI dengan keterangan berupa teks atau warna. Pemilihan warna dan maknanya dibebaskan ke masing-masing kelompok, asalkan dijelaskan di readme/laporan.

Daftar input maze akan dikemas dalam sebuah folder yang dinamakan test dan terkandung dalam repository program. Folder tersebut akan setara kedudukannya dengan folder src dan doc

(struktur folder repository akan dijelaskan lebih lanjut di bagian bawah spesifikasi tubes). Cara input maze boleh langsung input file atau dengan textfield sehingga pengguna dapat mengetik nama maze yang diinginkan. Apabila dengan textfield, harus menghandle kasus apabila tidak ditemukan dengan nama file tersebut.

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

Bab 2:

Landasan Teori

2.1 Penjelasan Graf

Graf merupakan salah satu struktur data yang digunakan untuk merepresentasikan kumpulan objek yang saling berkaitan atau terhubung. Objek-objek ini disebut simpul atau node, sedangkan hubungan antar simpul disebut sisi atau edge. Graf dapat digunakan untuk merepresentasikan berbagai macam informasi, seperti jaringan sosial, jaringan transportasi, dan jaringan komputer. Setiap graf memiliki properti khusus seperti jumlah simpul dan sisi, serta derajat simpul (yaitu jumlah sisi yang terhubung ke suatu simpul). Graf dapat digunakan untuk melakukan berbagai operasi, seperti mencari rute terpendek antara dua simpul, menemukan sirkuit tertentu dalam graf, atau memeriksa apakah suatu graf memiliki properti tertentu seperti sifat-sifat aliran. Pada kasus ini, graf akan digambarkan dalam bentuk labirin yang akan ditelusuri.

Terdapat beberapa jenis graf, seperti graf tak-berarah, graf berarah, dan graf campuran. Graf tak-berarah adalah graf yang tidak memiliki arah pada sisinya, sehingga sisi-sisi dapat dihubungkan ke simpul mana pun. Sedangkan graf berarah memiliki arah pada setiap sisinya, sehingga sisi hanya menghubungkan simpul pada arah tertentu. Graf campuran adalah gabungan dari kedua jenis graf tersebut, di mana sisi-sisi dapat memiliki arah atau tidak. Graf juga dapat direpresentasikan dalam bentuk matriks atau daftar ketetanggaan, tergantung pada kebutuhan aplikasi.

2.2 Penjelasan Traversal Graf

Traversal graf adalah proses mengunjungi setiap simpul atau titik dalam sebuah graf, dengan cara yang teratur dan sistematis. Proses traversal graf ini biasanya dilakukan untuk tujuan tertentu, seperti mencari rute terpendek antara dua simpul dalam graf, menemukan sirkuit atau lintasan tertentu dalam graf, atau memeriksa apakah suatu graf memiliki properti tertentu seperti sifat-sifat aliran.

Ada beberapa algoritma yang dapat digunakan untuk melakukan traversal graf, seperti algoritma Breadth-First Search (BFS) dan Depth-First Search (DFS). Dalam algoritma BFS, traversal

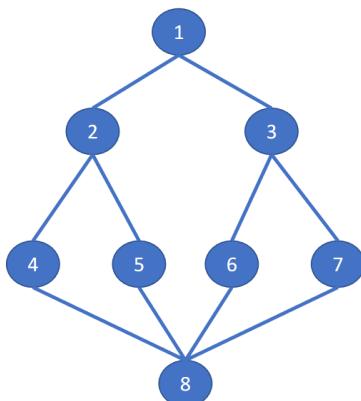
dimulai dari simpul awal dan mengunjungi semua simpul yang terhubung secara langsung ke simpul awal terlebih dahulu, kemudian bergerak ke simpul-simpul yang lebih jauh. Sementara dalam algoritma DFS, traversal dimulai dari simpul awal dan mengunjungi simpul terdekat yang belum dikunjungi, kemudian kembali ke simpul sebelumnya dan mengunjungi simpul yang belum dikunjungi lainnya.

Dalam program ini, algoritma akan dikhkususkan pada DFS dan BFS untuk menjalankan mengunjungi setiap simpul pada graf agar mendapatkan treasure pada graf statis.

2.3 Penjelasan Algoritma BFS

Algoritma BFS (Breadth First Search), sesuai nama dan artinya, adalah pencarian simpul graf secara melebar. Proses pengerjaan algoritma dimulai dari simpul awal, setelah itu mengunjungi semua simpul yang terhubung secara langsung ke simpul awal terlebih dahulu, baru kemudian bergerak ke simpul-simpul yang lebih jauh.

BFS diimplementasikan berdasarkan struktur data queue. Proses dimulai dengan memasukkan simpul awal ke dalam antrian. Selanjutnya, simpul pertama dalam antrian diambil dan seluruh simpul yang terhubung langsung dengan simpul tersebut dimasukkan ke dalam antrian. Setelah itu, simpul kedua dalam antrian diambil dan semua simpul yang terhubung langsung dengan simpul tersebut yang belum dikunjungi dimasukkan ke dalam antrian, dan seterusnya hingga semua simpul terhubung telah dikunjungi. Berikut adalah ilustrasi dari pemakaian algoritma BFS:



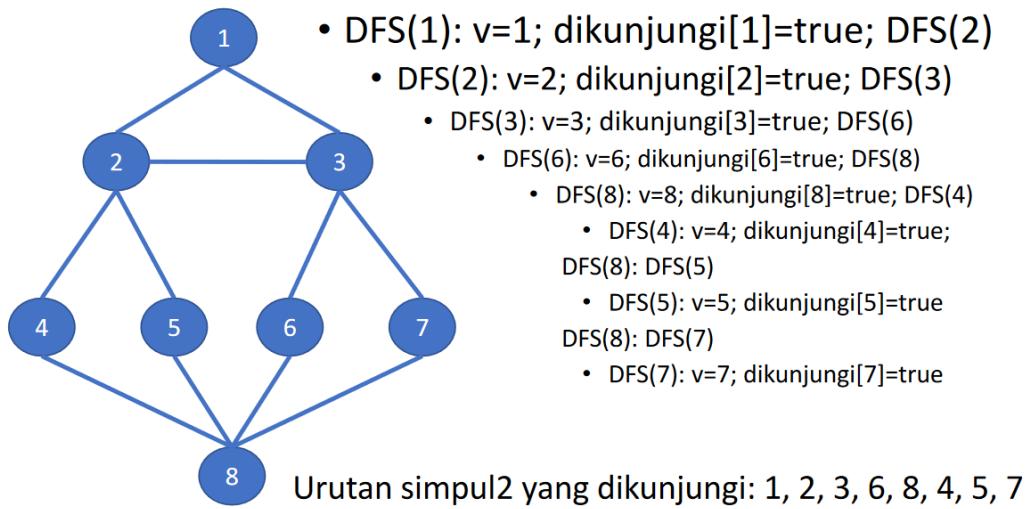
Iterasi	V	Q	dikunjungi							
			1	2	3	4	5	6	7	8
Inisialisasi	1	{1}	T	F	F	F	F	F	F	F
Iterasi 1	1	{2,3}	T	T	T	F	F	F	F	F
Iterasi 2	2	{3,4,5}	T	T	T	T	T	F	F	F
Iterasi 3	3	{4,5,6,7}	T	T	T	T	T	T	T	F
Iterasi 4	4	{5,6,7,8}	T	T	T	T	T	T	T	T
Iterasi 5	5	{6,7,8}	T	T	T	T	T	T	T	T
Iterasi 6	6	{7,8}	T	T	T	T	T	T	T	T
Iterasi 7	7	{8}	T	T	T	T	T	T	T	T
Iterasi 8	8	{}	T	T	T	T	T	T	T	T

Urutan simpul2 yang dikunjungi: 1, 2, 3, 4, 5, 6, 7, 8

Gambar 2. Ilustrasi BFS

2.4 Penjelasan Algoritma DFS

DFS (Depth first search), merupakan traversal graf dengan cara mendalam yang menggunakan dasar struktur data stack. Proses dimulai dengan memasukkan simpul awal ke dalam tumpukan. Selanjutnya, simpul pertama dalam tumpukan diambil dan semua simpul yang terhubung langsung dengan simpul tersebut yang belum dikunjungi dimasukkan ke dalam tumpukan. Setelah itu, simpul kedua dalam tumpukan diambil dan semua simpul yang terhubung langsung dengan simpul tersebut yang belum dikunjungi dimasukkan ke dalam tumpukan, dan seterusnya hingga semua simpul terhubung telah dikunjungi. Berikut adalah ilustrasi dari pemakaian algoritma DFS:



Gambar 3. Ilustrasi DFS

Keuntungan dari algoritma DFS adalah dapat digunakan untuk mencari sirkuit atau lintasan tertentu dalam graf. Algoritma ini juga dapat digunakan untuk memeriksa apakah suatu graf memiliki properti tertentu seperti sifat-sifat aliran, serta digunakan dalam beberapa masalah optimasi dan analisis graf. Namun, algoritma DFS dapat terjebak dalam siklus tak terhingga jika graf memiliki sirkuit tak terbatas. Selain itu, karena algoritma ini cenderung menjelajahi satu cabang simpul hingga ke dasarnya sebelum kembali ke simpul lain, maka proses traversal dapat menjadi lambat jika graf sangat dalam atau bercabang banyak.

Bab 3:

Aplikasi Strategi BFS dan DFS

3.1 Langkah-langkah pemecahan masalah

Langkah pertama dari alur program adalah menerima maze dan menentukan titik awal dari pemain yang akan dieksekusi kemudian. Kemudian, program akan berjalan sesuai dengan algoritma yang dipilih oleh pengguna; BFS/DFS, dan mencari jalan pulangnya menggunakan algoritma TSP (Travelling salesman problem). Seluruh jalan dari solusinya akan divisualisasikan pada aplikasi.

3.2 Proses mapping persoalan menjadi elemen-elemen algoritma BFS dan DFS

Sesuai dengan yang telah dijelaskan pada bab sebelumnya, algoritma BFS memiliki dasar struktur data queue dan algoritma DFS memiliki dasar struktur data stack. Sehingga, queue dan stack merupakan salah satu hal utama dari pembuatan algoritma keduanya. Berikut adalah rinciannya:

1. BFS

Pertama, algoritma akan selalu memvalidasi di manakah tile sedang berada berdasarkan value setelah melakukan dequeue pada queue yang tersedia. Jika tile tersebut bukan merupakan treasure, maka algoritma akan memeriksa elemen terdekat dari tile (atas, kanan, bawah, kiri) dan melakukan enqueue terhadap queue jika elemen terdekat bukan merupakan dinding. Algoritma ini akan terus dipanggil selama jumlah treasure belum bernilai 0.

2. DFS

Untuk algoritma DFS, algoritma akan selalu memvalidasi di manakah tile sedang berada berdasarkan value setelah melakukan pop pada stack yang tersedia. Jika tile tersebut bukan merupakan treasure, maka algoritma akan memeriksa elemen terdekat dari tile (atas, kanan, bawah, kiri) dan melakukan push terhadap stack jika elemen terdekat bukan merupakan dinding. Algoritma ini akan terus dipanggil selama jumlah treasure belum bernilai 0.

3.3 Contoh ilustrasi kasus lain yang berbeda dengan contoh pada spesifikasi tugas

Selain DFS dan BFS, masih terdapat beberapa algoritma lain untuk menyelesaikan persoalan treasure hunter pada maze ini. Sebagai salah satu contohnya adalah algoritma A* (A star). Algoritma ini menggunakan fungsi heuristik untuk memilih jalur terbaik pada setiap langkah pencarian. Fungsi heuristik ini membantu algoritma A* untuk menentukan jarak yang lebih optimal dari setiap simpul ke tujuan. Algoritma A* biasanya digunakan dalam masalah pencarian rute terpendek pada graf berbobot atau graf dengan jarak Euclidean. Untuk persoalan treasure hunt, jarak euclidean dapat digunakan untuk mencari jarak terpendek antara starting point dengan treasure. Pada algoritma A*, struktur data yang digunakan adalah priority queue.

Bab 4:

Analisis Pemecahan Masalah

4.1 Implementasi program

Pengimplementasian algoritma ditulis dalam bahasa pemrograman C# dan dibantu oleh aplikasi Visual Studio 2022. Berikut adalah rincian dari program yang telah dibuat:

1. BFS

```
class BFS{
    // Algoritma BFS untuk menemukan jalur yang melewati seluruh
    // treasure di maze
    public List<char> Solver(Maze maze){
        Queue<Tile> queue = new Queue<Tile>();
        List<char> path = new List<char>();
        bool searchTSP = false;

        // Mengambil tile pertama yang akan dieksekusi
        Tile start = maze.getTile(maze.getStart().getX(), maze.getStart().getY());
        start.addNumOfStepped();
        enqueue(queue, start, 'S');

        // Melakukan algoritma BFS hingga ditemukan seluruh
        // treasure
        while(maze.getTreasureCount() != 0){
            doTheThing(maze, queue, path, searchTSP);
        }

        // Mencari jalan pulang (TSP)
        if (Global.getTSPstatus())
        {
            searchTSP = true;
            while (queue.getSize() != 0)
            {
                doTheThing(maze, queue, path, searchTSP);
            }
        }
    }
}
```

```

        return path;
    }

    // Algoritma BFS
    public void doTheThing(Maze maze, Queue<Tile> queue,
List<char> path, boolean searchTSP) {
    Tile check = dequeue(queue);
    int checkX = check.getPosition().getX();
    int checkY = check.getPosition().getY();

    // Mengecek apakah tile sekarang adalah Treasure
    if(check.getType()=='T') {
        maze.decreaseTreasureCount();
        maze.getPath(check, maze, path);
        clearQueue(queue);
        maze.resetStatusMaze();
        enqueue(queue, check, 'S');
    }
    else if (searchTSP && check.getType() == 'K')
    {
        maze.getPath(check, maze, path);
        clearQueue(queue);
        maze.resetStatusMaze();
    }

    // Mengecek adjacent Tile
    else {
        // Check Up
        Tile tileCheck = maze.getTile(checkX, checkY-1);
        if(tileCheck.getType() != 'X' &&
tileCheck.getStatus() < check.getStatus())
        {
            enqueue(queue, tileCheck, 'D');
        }
        // Check Right
        tileCheck = maze.getTile(checkX+1, checkY);
        if(tileCheck.getType() != 'X' && tileCheck.getStatus()
< check.getStatus())
        {
            enqueue(queue, tileCheck, 'L');
        }
    }
}

```

```

        }

        // Check Down
        tileCheck = maze.getTile(checkX, checkY+1);
        if(tileCheck.getType() != 'X' && tileCheck.getStatus()
< check.getStatus())
        {
            enqueue(queue, tileCheck, 'U');
        }

        // Check Left
        tileCheck = maze.getTile(checkX-1, checkY);
        if(tileCheck.getType() != 'X' && tileCheck.getStatus()
< check.getStatus())
        {
            enqueue(queue, tileCheck, 'R');
        }
    }

    // Fungsi yang memasukkan Tile ke dalam queue
    public void enqueue(Queue<Tile> queue, Tile add, char
pathBefore)
{
    if(!queue.isInside(add))
    {
        add.addStatus();
        add.setPathBefore(pathBefore);
        queue.enqueue(add);

        // Mewarnai tile yang masuk ke dalam queue
        if (add.getStatus() == 0 && add.getType() == 'R' &&
add.getNumOfStepped() == 0 && !add.getIsChecked())
        {
            Global.changeColor(add.getPosition().getX(),
add.getPosition().getY(), Color.LightGray);
        }
    }
}

// Fungsi yang mengeluarkan Tile dari queue
public Tile dequeue(Queue<Tile> queue)
{
    Tile deque = queue.dequeue();
    deque.addStatus();
}

```

```

        deque.setIsChecked(true);
        if (deque.getStatus() == 1 && deque.getType() == 'R' &&
deque.getNumOfStepped()==0)
        {
            Global.changeColor(deque.getPosition().getX(),
deque.getPosition().getY(), Color.Gray);
        }
        return deque;
    }

    // Mengosongkan queue
    public void clearQueue(Queue<Tile> queue) {
        if(queue.getSize()!=0){
            for(int i = 0;i<queue.getSize();i++){
                queue.getElmt(i).setStatus(queue.getElmt(i).getStatus()-1);
                if (queue.getElmt(i).getStatus() == -1 &&
queue.getElmt(i).getType() == 'R' &&
!queue.getElmt(i).getIsChecked())
                {
                    Global.changeColor(queue.getElmt(i).getPosition().getX(),
queue.getElmt(i).getPosition().getY(), Color.White);
                }
            }
            queue.clear();
        }
    }
}

```

2. DFS

```

class DFS{
    // Algoritma DFS untuk menemukan jalur yang melewati seluruh
Treasure di maze
    public List<char> Solver(Maze maze){
        Stack<Tile> stack = new Stack<Tile>();
        List<char> path = new List<char>();
        bool searchTSP = false;
    }
}

```

```

        // Mengambil tile pertama yang akan dieksekusi
        Tile      start      =
maze.getTile(maze.getStart().getX(),maze.getStart().getY());
        start.addNumOfStepped();
        push(stack,start,'S');

        // Melakukan algoritma DFS hingga ditemukan seluruh
Treasure
        while(maze.getTreasureCount () !=0) {
            doTheThing(maze,stack,path,searchTSP);

        }

        // Mencari jalan pulang (TSP)
        if (Global.getTSPstatus())
        {
            searchTSP = true;
            while (stack.getSize() !=0)
            {
                doTheThing(maze, stack, path, searchTSP);
            }
        }

        return path;
    }

    // Algoritma DFS
    public void doTheThing(Maze maze, Stack<Tile> stack,
List<char> path, bool searchTSP){
        Tile check = pop(stack);
        int checkX = check.getPosition().getX();
        int checkY = check.getPosition().getY();

        // Mengecek apakah tile sekarang adalah Treasure
        if(check.getType()=='T'){
            maze.decreaseTreasureCount ();
            maze.getPath(check,maze,path);
            clearStack(stack);
            maze.resetStatusMaze ();
        }
    }
}

```

```

        push(stack,check,'S');

    } else if (searchTSP && check.getType() == 'K')
    {
        maze.getPath(check, maze, path);
        clearStack(stack);
        maze.resetStatusMaze();
    }

    // Mengecek adjacent Tile
    else {
        // Check Left
        Tile tileCheck = maze.getTile(checkX-1,checkY);
        if(tileCheck.getType() != 'X' && tileCheck.getStatus()
< check.getStatus())
        {
            push(stack,tileCheck,'R');
        }
        // Check Down
        tileCheck = maze.getTile(checkX,checkY+1);
        if(tileCheck.getType() != 'X' && tileCheck.getStatus()
< check.getStatus())
        {
            push(stack,tileCheck,'U');
        }
        // Check Right
        tileCheck = maze.getTile(checkX+1,checkY);
        if(tileCheck.getType() != 'X' && tileCheck.getStatus()
< check.getStatus())
        {
            push(stack,tileCheck,'L');
        }
        // Check Up
        tileCheck = maze.getTile(checkX,checkY-1);
        if(tileCheck.getType() != 'X' && tileCheck.getStatus()
< check.getStatus())
        {
            push(stack,tileCheck,'D');
        }
    }
}

```

```

    // Fungsi yang memasukkan Tile ke dalam stack
    public void push(Stack<Tile> stack, Tile add, char
pathBefore) {
        if(stack.isInside(add)) {
            stack.removeEl(add);
        } else {
            add.addStatus();
        }
        stack.push(add);
        add.setPathBefore(pathBefore);
        // Mewarnai tile yang masuk ke dalam stack
        if (add.getStatus() == 0 && add.getType() == 'R' &&
!add.isChecked() && add.getNumOfStepped()==0)
        {
            Global.changeColor(add.getPosition().getX(),
add.getPosition().getY(), Color.LightGray);
        }
    }

    // Fungsi yang mengeluarkan Tile dari stack
    public Tile pop(Stack<Tile> stack){
        Tile pops = stack.pop();
        pops.addStatus();
        pops.setChecked(true);
        if (pops.getType() == 'R' && pops.getNumOfStepped() ==
0)
        {
            Global.changeColor(pops.getPosition().getX(),
pops.getPosition().getY(), Color.Gray);
        }
        return pops;
    }

    // Mengosongkan stack
    public void clearStack(Stack<Tile> stack) {
        if(stack.getSize()!=0) {
            for(int i = 0;i<stack.getSize();i++) {

stack.getElmt(i).setStatus(stack.getElmt(i).getStatus()-1);

```

```
        if (stack.getElmt(i).getStatus() == -1 &&
stack.getElmt(i).getType() == 'R' &&
!stack.getElmt(i).getIsChecked() &&
stack.getElmt(i).getNumOfStepped() == 0)
{
    Global.changeColor(stack.getElmt(i).getPosition().getX(),
stack.getElmt(i).getPosition().getY(), Color.White);
}
stack.clear();
}
}
```

4.2 Penjelasan struktur data

Kode utama dari program menggunakan prinsip pemrograman berbasis objek dalam kodennya sehingga struktur data yang digunakan merupakan class. Terdapat beberapa class penting dalam kodennya yang mewakili objek-objek penting pada permainan, diantaranya:

1. Queue/BFS

Queue merupakan dasar dari algoritma BFS yang dipakai untuk menyimpan antrian Node (Tile) yang akan di-traversal pada Maze dengan aturan FIFO (first in first out). Queue dibuat berdasarkan List pada System.Collection.Generic dengan template, sehingga struktur data ini dapat digunakan untuk berbagai kelas. Berikut adalah beberapa method pada struktur data queue:

- A. dequeue, berguna untuk menghapus atau mengeluarkan Tile pada antrian paling depan.
 - B. enqueue, berguna untuk menambahkan Tile pada antrian pada antrian paling belakang.
 - C. clearQueue, berguna untuk menghapus seluruh isi queue.
 - D. doTheThing, merupakan implementasi algoritma utama dari BFS
 - E. Solver, fungsi utama dari BFS

2. Stack/DFS

Stack adalah dasar dari algoritma DFS yang dipakai untuk menyimpan tumpukan Node

(Tile) yang akan di-traversal pada Maze dengan aturan LIFO (last in first out). Stack dibuat berdasarkan List pada System.Collection.Generic dengan template, sehingga struktur data ini dapat digunakan untuk berbagai kelas. Berikut adalah beberapa method pada struktur data stack:

- A. pop, berguna untuk menghapus atau mengeluarkan Tile pada tumpukan paling depan/atasi
 - B. push, berguna untuk menambahkan Tile pada antrian paling depan/atasi
 - C. clearStack, berguna untuk menghapus seluruh elemen dari tumpukan.
 - D. doTheThing, merupakan implementasi algoritma utama dari DFS
 - E. Solver, fungsi utama dari DFS
3. Reader

Reader, sesuai namanya, merupakan struktur data yang dipakai untuk membaca file agar program dapat menjalankan berdasarkan file input. Berikut adalah method pada kelas Reader:

- A. readFile, menerima parameter nama file yang akan dibaca yang akan dikonversikan ke dalam bentuk token. Method ini menggunakan System.IO sebagai
4. Point
- Berisi koordinat kartesius yang memuat integer x dan y sebagai dasar dari Maze dan juga yang menentukan posisi dari suatu Tile nantinya. Berikut adalah method pada Point:
- A. getX, mengembalikan nilai x
 - B. getY, mengembalikan nilai y
 - C. setX, mengganti nilai x
 - D. setY, mengganti nilai y
 - E. print, menuliskan output x dan y
5. Tile

Tile merupakan kotak-kotak pada maze yang memiliki posisi pada Point (x dan y). Suatu tile memiliki type (char), pathBefore (char), status (int), numOfStepped (int), position (Point), dan isChecked (bool). Kebanyakan dari elemen pada tile ini memiliki getter dan seternya masing-masing. Salah satu metode yang paling penting dari Tile adalah

printTile. Fungsi ini akan menampilkan warna yang berbeda untuk setiap type dan numOfStepped yang berbeda juga.

6. Maze

Maze adalah daerah yang akan ditelusuri nantinya. Sebuah Maze terdiri dari sizeX, sizeY, array of Tile, start (Point), dan treasureCount. Maze juga memiliki constructor, getter, dan setter untuk beberapa elemennya.

4.3 Penjelasan tata cara penggunaan program

To run the Maze Solver program, follow these steps:

1. Clone this repository or download it as a ZIP file and extract it to your local machine.
2. Build the executable by following the above instructions.
3. Open the `./bin/MazeSolver.exe` file.
4. Select the directory where your maze file `maze.txt` is located by clicking on the search bar.
5. Click on `Visualize` to display the maze.
6. Select either `Depth First Search` or `Breadth First Search` for your maze algorithm
7. Click on `Search` to display the solution.
8. Adjust the speed of the algorithm by sliding the `Speed` slider. You can also use `Turbo` to instantly complete the maze.
9. Activate the `TSP` toggle to find the solution to the Traveling Salesman Problem.

4.4 Hasil pengujian/Eksperimen

Untuk menguji strategi algoritma BFS dan DFS pada persoalan treasure hunt, asisten telah menyiapkan 5 buah test case untuk dicoba. Berikut adalah hasil percobaan program yang telah kami tulis dengan test case tersebut:

1. sampel-1

Nama	Foto

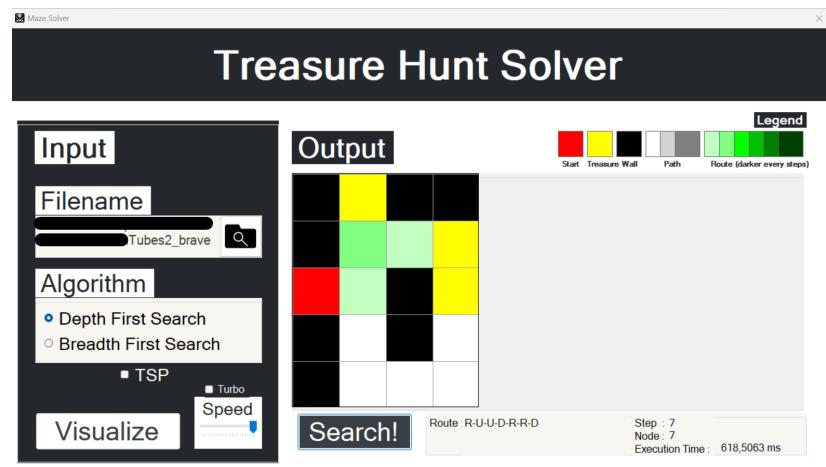
sampel-1.txt

A screenshot of a Windows Notepad window titled "sampel-1.txt - Notep...". The content of the file is a 5x5 grid of characters representing a maze:

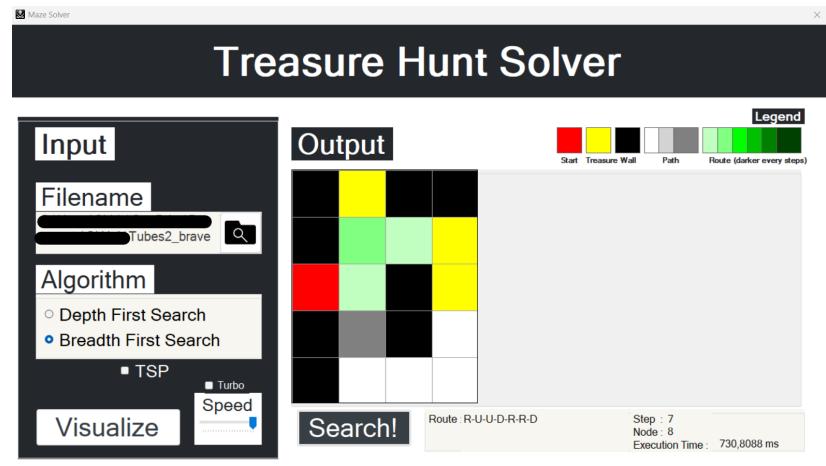
```
X T X X  
X R R T  
K R X T  
X R X R  
X R R R
```

The status bar at the bottom shows "Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8".

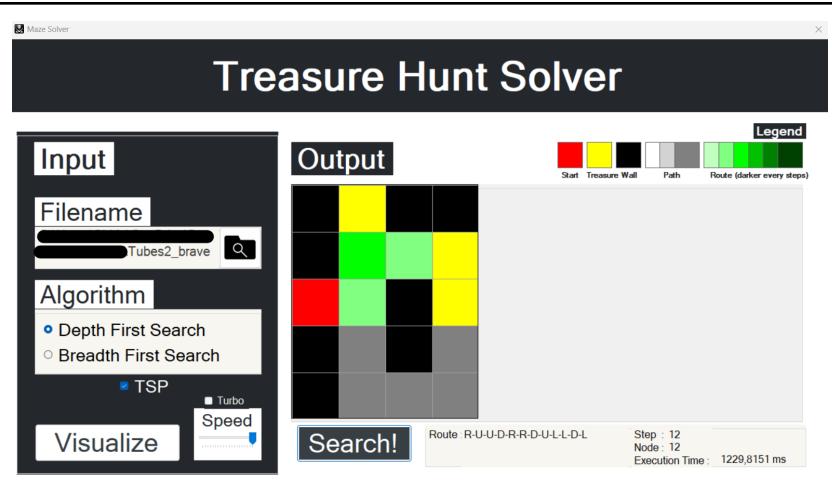
Pencarian solusi menggunakan algoritma DFS



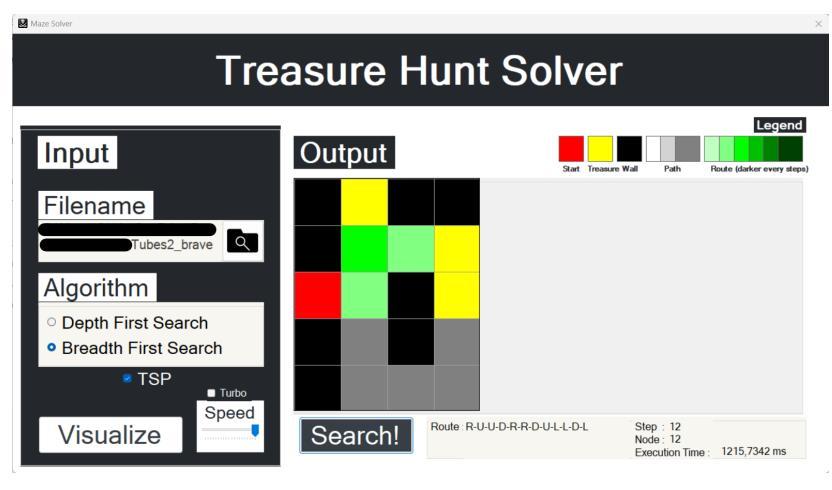
Pencarian solusi menggunakan algoritma BFS



Pencarian solusi menggunakan algoritma DFS dan kembali ke semula menggunakan algoritma TSP



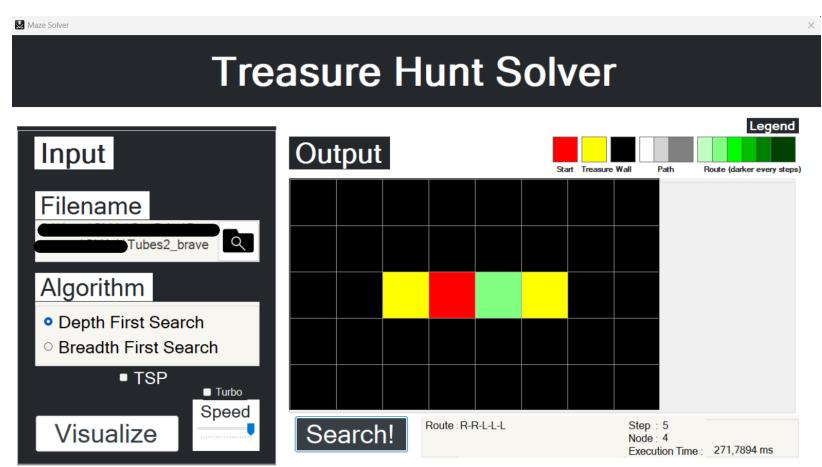
Pencarian solusi menggunakan algoritma BFS dan kembali ke semula menggunakan algoritma TSP



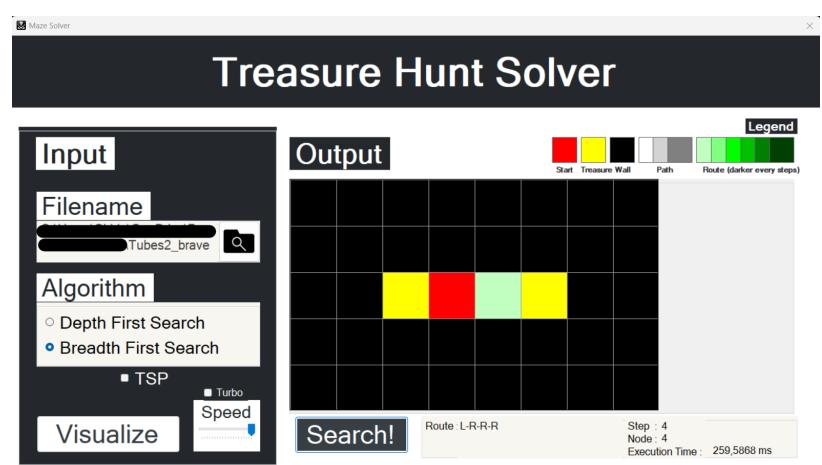
2. sampel-2

Nama	Foto
sampel-2.txt	<p>The screenshot shows a Notepad window with the file 'sampel-2.txt' open. The content of the file is a 5x5 grid of 'X' characters, representing a maze. The Notepad window includes standard menu options like File, Edit, View, and a status bar at the bottom showing 'Ln 1, Col 1 100% Windows (CRLF) UTF-8'.</p>

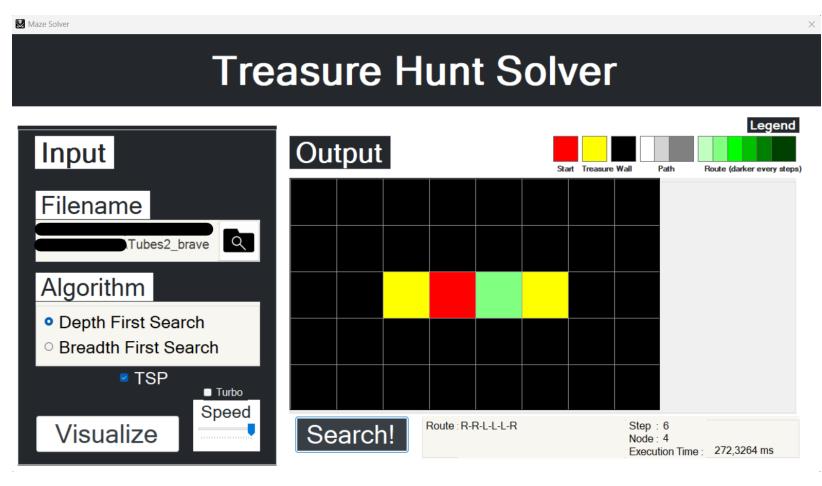
Pencarian solusi menggunakan algoritma DFS



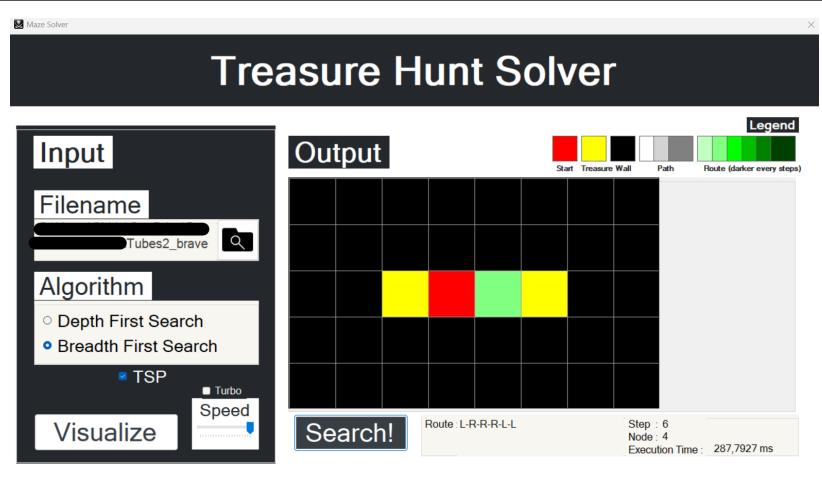
Pencarian solusi menggunakan algoritma BFS



Pencarian solusi menggunakan algoritma DFS dan kembali ke semula menggunakan algoritma TSP



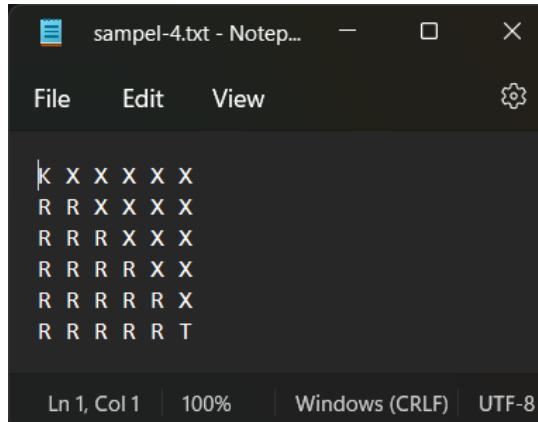
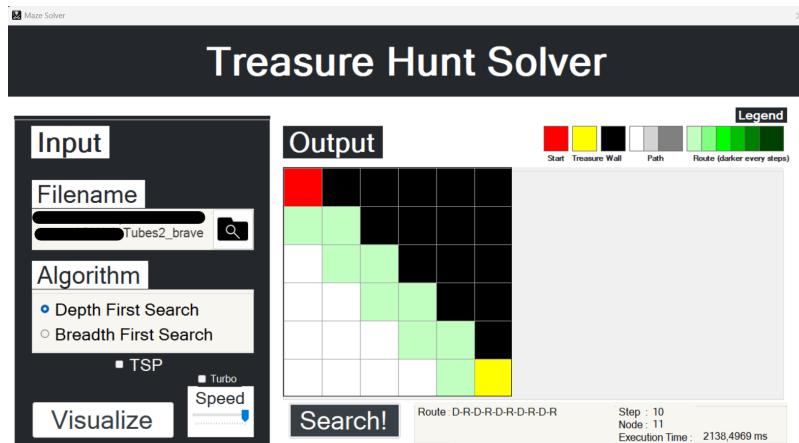
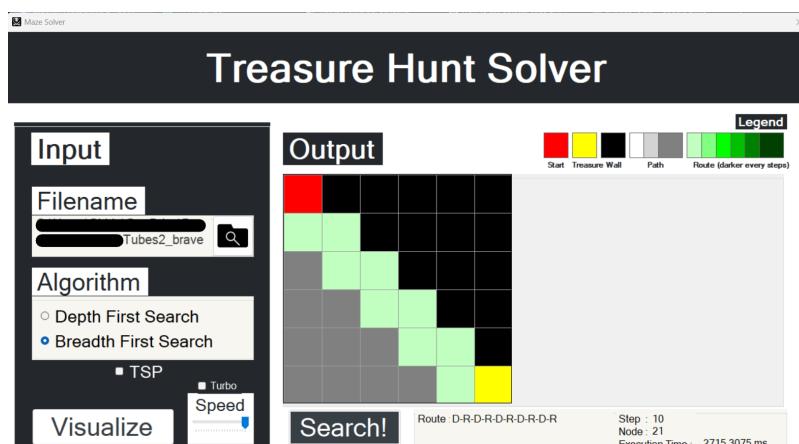
Pencarian solusi menggunakan algoritma BFS dan kembali ke semula menggunakan algoritma TSP



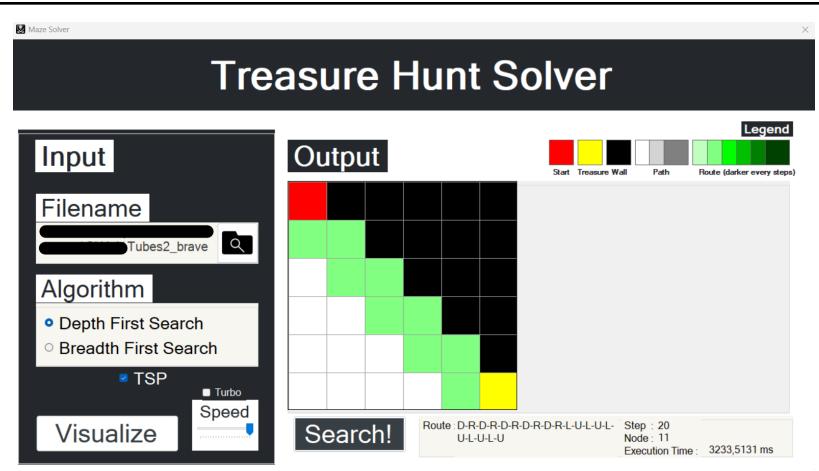
3. sampel-3

Nama	Foto
sampel-3.txt	
Output (mewakili semua karena input tidak valid)	

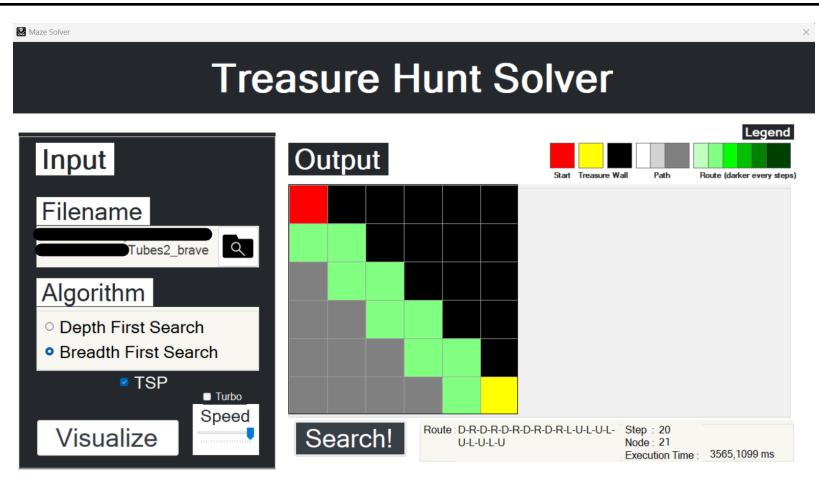
4. sampel-4

Nama	Foto
sampel-4.txt	 <p>Ln 1, Col 1 100% Windows (CRLF) UTF-8</p>
Pencarian solusi menggunakan algoritma DFS	
Pencarian solusi menggunakan algoritma BFS	

Pencarian solusi menggunakan algoritma DFS dan kembali ke semula menggunakan algoritma TSP



Pencarian solusi menggunakan algoritma BFS dan kembali ke semula menggunakan algoritma TSP

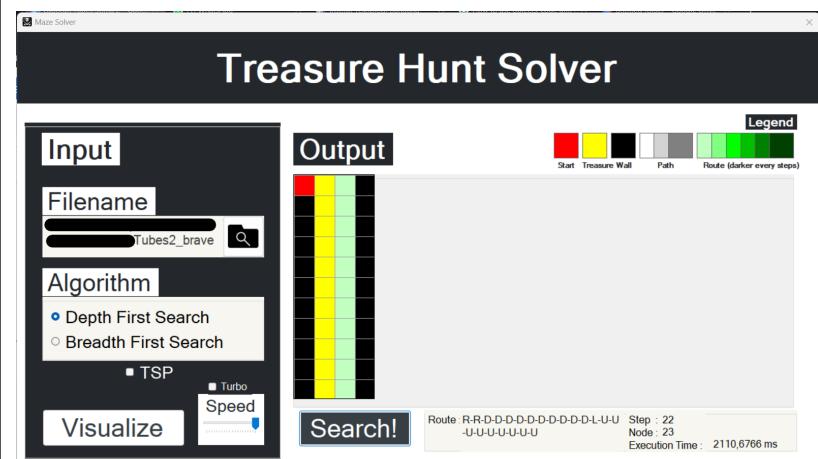


5. sampel-5

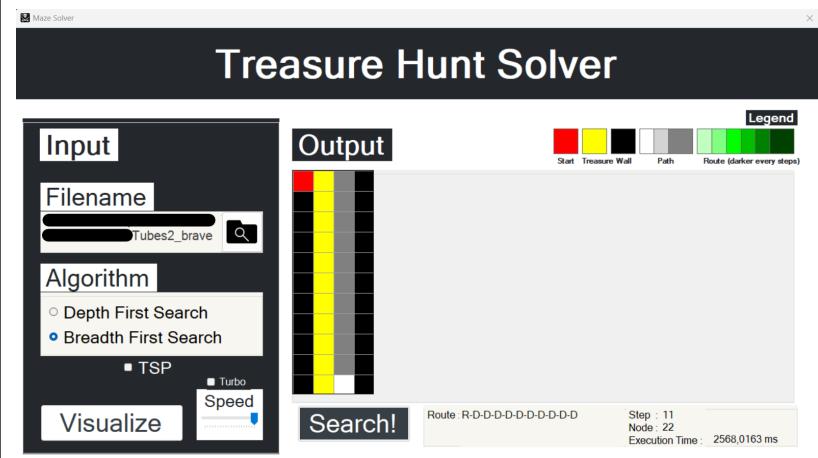
Nama	Foto
------	------

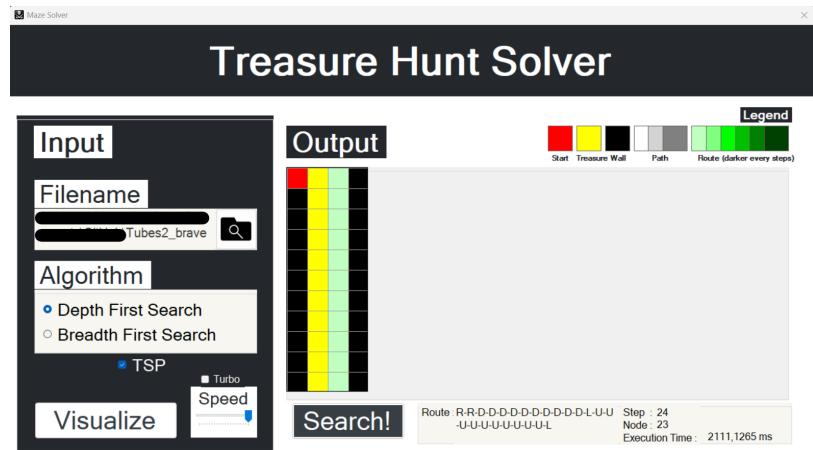
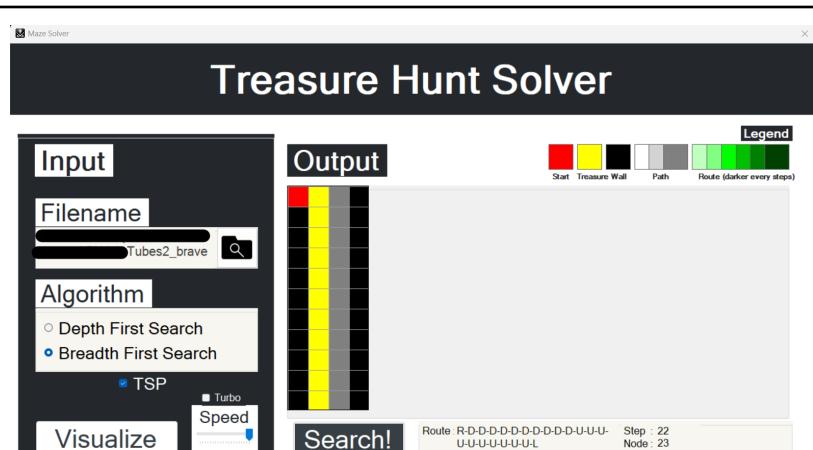
sampel-5.txt

Pencarian solusi menggunakan algoritma DFS



Pencarian solusi menggunakan algoritma BFS



Pencarian solusi menggunakan algoritma DFS dan kembali ke semula menggunakan algoritma TSP	
Pencarian solusi menggunakan algoritma BFS dan kembali ke semula menggunakan algoritma TSP	

4.4 Analisis dari desain solusi algoritma BFS dan DFS yang diimplementasikan pada setiap pengujian yang dilakukan

Jika ditinjau berdasarkan uji eksperimen bab di atas, terdapat beberapa kasus di mana BFS lebih unggul dibandingkan dengan DFS; begitupun sebaliknya. Tetapi, secara general, DFS memiliki keunggulan dalam kasus graf yang sangat dalam dan memiliki banyak cabang dengan kedalaman yang tidak terlalu besar. Namun, dalam kasus graf yang lebih lebar dan dangkal, DFS dapat memakan waktu lebih lama karena kemungkinan untuk mengunjungi banyak node yang tidak relevan sebelum menemukan solusi. Sementara itu, BFS akan lebih efektif dalam kasus graf yang lebih lebar dan dangkal. Hal ini karena BFS lebih cenderung menemukan solusi lebih awal, terutama jika solusi berada pada level yang dangkal atau dekat dengan node awal. Namun, pada kasus graf yang sangat dalam dengan banyak cabang, BFS mungkin memakan waktu lebih lama daripada DFS karena harus mengunjungi banyak node pada setiap level sebelum

menemukan solusi. Hal ini menunjukkan bahwasannya kedua algoritma dapat digunakan secara optimal pada maze yang tepat dan sesuai.

Bab 5:

Penutup

5.1 Kesimpulan

Pembuatan treasure finder untuk sebuah maze atau labirin dapat dilakukan menggunakan bahasa pemrograman C# dengan cara mengimplementasikan strategi algoritma BFS dan DFS yang telah diajarkan dasarnya pada mata kuliah IF2211 - Strategi Algoritma. Setelah membuat program ini, penulis dapat disimpulkan penggunaan algoritma BFS dan DFS dapat menjadi salah satu solusi untuk mendapatkan treasure pada suatu labirin.

5.2 Saran

Berikut adalah beberapa saran yang telah kami diskusikan dan kumpulkan untuk pembuatan program serupa kedepannya agar program dapat membawa hasil yang jauh lebih baik:

1. Menambahkan variasi algoritma seperti A* (A-Star) atau Algoritma Dead End Filling agar pencarian menjadi lebih efektif secara umum.
2. Untuk asisten agar memberikan test case segera setelah spek dirilis.

5.3 Refleksi

Pembuatan tugas besar mata kuliah IF2211 Strategi Algoritma yang kedua ini merupakan salah satu pengalaman yang sangat berharga bagi penulis. Kami belajar bagaimana suatu pengimplementasian strategi algoritma dapat sangat berguna pada kehidupan sehari-hari,

khususnya pembuatan program untuk memecahkan suatu masalah, dalam kasus ini, pencari treasure. Meskipun demikian, penulis merasa masih terdapat beberapa kekurangan pada program sehingga penulis sangat berharap untuk pembuatan algoritma serupa kedepannya dapat dieksekusi dengan lebih baik.

Daftar Pustaka

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

Lampiran

Link Repository GitHub : https://github.com/farhanfahreezy/Tubes2_braveEagle69

Link Video YouTube : <https://bit.ly/VideoTubes2braveEagle69>