

TUGAS BESAR II IF2211 STRATEGI ALGORITMA
SEMESTER II TAHUN 2022/2023
APLIKASI ALGORITMA BFS DAN DFS DALAM MENYELESAIKAN
PERSOALAN MAZE TREASURE HUNT



Disusun Oleh:

kelp juice

13521049 Brian Kheng

13521059 Arleen Chrysantha Gunardi

13521082 Farizki Kurniawan

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

2023

DAFTAR ISI

DAFTAR ISI	2
DAFTAR TABEL	4
DAFTAR GAMBAR	5
BAB I	
DESKRIPSI TUGAS	6
1.1 Deskripsi Tugas	6
BAB II	
LANDASAN TEORI	7
2.1 Traversal Graf	7
2.2 Algoritma Breadth First Search (BFS)	7
2.3 Algoritma Depth First Search (DFS)	8
2.4 Traveling Salesman Problem	9
2.5 C# Desktop Application Development	9
BAB III	
APLIKASI STRATEGI BFS DAN DFS	10
3.1 Langkah-Langkah Pemecahan Masalah	10
3.1.1 Langkah Algoritma Depth First Search (DFS)	10
3.1.2 Langkah Algoritma Breadth First Search (BFS)	11
3.1.3 Langkah Algoritma Traveling Salesman Problem	13
3.2 Proses Mapping Persoalan Menjadi Elemen-Elemen Algoritma BFS dan DFS	14
3.3 Ilustrasi Kasus Lain	15
3.3.1 Ilustrasi dengan Algoritma Depth First Search (DFS)	15
3.3.2 Ilustrasi dengan Algoritma Breadth First Search (BFS)	17
3.3.3 Ilustrasi dengan Algoritma Traveling Salesman Problem (TSP)	19
BAB IV	
IMPLEMENTASI DAN PENGUJIAN	21
4.1 Implementasi Program	21
4.1.1 Implementasi Depth First Search (DFS)	21
4.1.2 Implementasi Breadth First Search (BFS)	23
4.1.3 Implementasi Traveling Salesman Problem	28
4.2 Struktur Data	30
4.2.1 Map	31
4.2.2 Point	32
4.2.2 PointDir	32
4.2.3 Solver	33
4.2.4 DFSSolver	34

4.2.5 BFSSolver	34
4.2.6 TSPSolver	35
4.2.7 Parser	35
4.3 Tata Cara Penggunaan Program	36
4.4 Hasil Pengujian	36
4.4.1 Pengujian 1	36
4.4.2 Pengujian 2	39
4.4.3 Pengujian 3	42
4.4.4 Pengujian 4	43
4.4.5 Pengujian 5	46
4.5 Analisis Desain Solusi	49
4.5.1 Analisis Pengujian 1	49
4.5.2 Analisis Pengujian 2	51
4.5.3 Analisis Pengujian 3	52
4.5.4 Analisis Pengujian 4	52
4.5.5 Analisis Pengujian 5	53
BAB V	
PENUTUP	56
5.1 Simpulan	56
5.2 Saran	56
DAFTAR REFERENSI	57
LAMPIRAN	58

DAFTAR TABEL

Tabel 4.4.1 Hasil Pengujian 1	38
Tabel 4.4.2 Hasil Pengujian 2	41
Tabel 4.4.3 Hasil Pengujian 3	42
Tabel 4.4.4 Hasil Pengujian 4	45
Tabel 4.4.5 Hasil Pengujian 5	48

DAFTAR GAMBAR

Gambar 2.2.1 Ilustrasi BFS	7
Gambar 2.2.2 Ilustrasi DFS	8
Gambar 3.3.1 Ilustrasi Kasus (Peta)	15
Gambar 3.3.1.1 Ilustrasi Kasus dengan DFS (Ilustrasi dengan Tree)	16
Gambar 3.3.1.2 Ilustrasi Kasus dengan DFS (Ilustrasi pada Peta)	17
Gambar 3.3.2.1 Ilustrasi Kasus dengan BFS (Ilustrasi dengan Tree)	18
Gambar 3.3.2.2 Ilustrasi Kasus dengan BFS (Ilustrasi pada Peta)	18
Gambar 3.3.3.1 Ilustrasi Kasus dengan TSP (Ilustrasi dengan Tree)	19
Gambar 3.3.3.2 Ilustrasi Kasus dengan TSP (Ilustrasi pada Peta)	20
Gambar 4.2.1 Diagram Kelas (Struktur Data) Solver	31
Gambar 4.4.1 Pengujian 1	38
Gambar 4.4.2 Pengujian 2	41
Gambar 4.4.3 Pengujian 3	42
Gambar 4.4.4 Pengujian 4	44
Gambar 4.4.1 Pengujian 5	47
Gambar 4.5.1 Analisis Pengujian 1	50
Gambar 4.5.2 Analisis Pengujian 2	51
Gambar 4.5.4 Analisis Pengujian 4	53
Gambar 4.5.5 Analisis Pengujian 5	55

BAB I

DESKRIPSI TUGAS

1.1 Deskripsi Tugas

Tuan Krabs menemukan sebuah labirin distorsi terletak tepat di bawah Krusty Krab bernama El Doremi yang Ia yakini mempunyai sejumlah harta karun di dalamnya dan tentu saja Ia ingin mengambil harta karunnya. Karena 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.

Tujuan utama tugas besar ini adalah 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

Dengan memanfaatkan algoritma Breadth First Search (BFS) dan Depth First Search (DFS), program menelusuri *grid* (simpul) yang mungkin dikunjungi hingga ditemukan rute solusi, baik secara melebar ataupun mendalam, tergantung alternatif algoritma yang dipilih. Rute solusi adalah rute yang memperoleh seluruh *treasure* pada *maze*. Rute yang diperoleh dengan algoritma BFS dan DFS dapat berbeda, dan banyak langkah yang dibutuhkan pun menjadi berbeda. Prioritas arah simpul yang dibangkitkan ditentukan pada program. Tidak ada pergerakan secara diagonal. Selain itu, *input txt* tersebut divisualisasikan menjadi suatu *grid maze* beserta dengan hasil pencarian rute solusinya.

BAB II

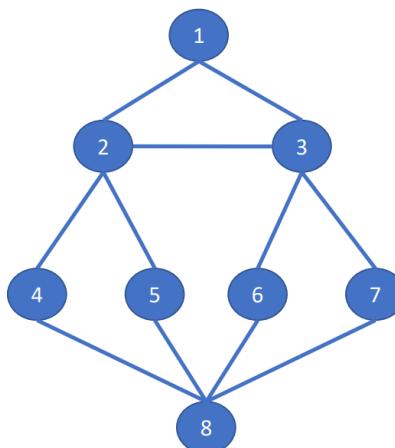
LANDASAN TEORI

2.1 Traversal Graf

Graf adalah suatu tipe data diskrit yang terdiri dari simpul (*vertex*) dan sisi (*edge*). Traversal graf merupakan proses untuk mengunjungi simpul secara sistematis. Apabila graf terhubung, – yaitu untuk setiap pasang simpul v_i dan v_j dalam himpunan V terdapat lintasan dari v_i ke v_j – maka traversal graf dapat dilakukan dengan dua macam algoritma: Breadth First Search (BFS) dan Depth First Search (DFS). Graf dapat merepresentasikan persoalan, sedangkan traversal graf merepresentasikan proses pencarian solusi persoalan. Dengan demikian, persoalan seperti mencari rute jalan keluar dari labirin dapat direpresentasikan sebagai graf dan diselesaikan dengan traversal graf.

2.2 Algoritma Breadth First Search (BFS)

Algoritma Breadth First Search (BFS) merupakan salah satu algoritma traversal graf. Algoritma ini melakukan pencarian secara melebar. Misalkan traversal graf dimulai dari simpul v , maka seluruh simpul yang bertetangga dengan v dikunjungi. Kemudian, kunjungi seluruh simpul yang bertetangga dengan simpul yang telah dikunjungi pada langkah sebelumnya. Hal ini dilakukan terus hingga seluruh simpul telah dikunjungi.



Gambar 2.2.1 Ilustrasi BFS

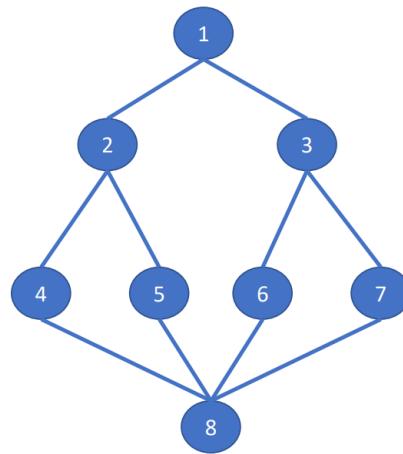
Sumber: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf>

Dalam implementasinya, algoritma BFS memiliki beberapa komponen struktur data yang memiliki peran dalam pemecahan permasalahan dengan algoritma ini:

1. Matriks ketetanggaan $A = [a_{ij}]$ dengan a_{ij} bernilai 1 jika simpul i bertetangga dengan simpul j , atau 0 jika simpul i tidak bertetangga dengan simpul j .
2. Antrian (queue) q yang menyimpan simpul-simpul yang telah dikunjungi
3. Tabel Boolean *dikunjungi* bertipe *array of Boolean* dengan nilai elemen *true* jika simpul telah dikunjungi, atau *false* jika simpul belum dikunjungi.

2.3 Algoritma Depth First Search (DFS)

Algoritma Depth First Search (DFS) merupakan algoritma traversal graf yang melakukan pencarian secara mendalam. Misalkan traversal graf dimulai dari simpul v , maka simpul v dikunjungi terlebih dahulu. Kemudian, terdapat simpul w yang bertetangga dengan simpul v , maka kunjungi simpul w tersebut. Tahap mengunjungi tersebut diulangi dari simpul w hingga suatu saat algoritma mencapai suatu simpul u dan seluruh simpul yang bertetangga telah dikunjungi, sehingga *backtrack* (runut-balik) dilakukan ke simpul sebelumnya yang memiliki simpul tetangga yang belum dikunjungi. Pencarian ini dilakukan hingga semua simpul (simpul yang dapat dicapai dari simpul yang telah dikunjungi) sudah dikunjungi.



Gambar 2.2.2 Ilustrasi DFS

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

Dalam implementasinya, algoritma DFS memiliki beberapa komponen struktur data yang memiliki peran dalam pemecahan permasalahan dengan algoritma ini:

1. Matriks ketetanggaan $A = [a_{ij}]$ dengan a_{ij} bernilai 1 jika simpul i bertetangga dengan simpul j , atau 0 jika simpul i tidak bertetangga dengan simpul j .
2. Stack s yang menyimpan simpul-simpul yang telah dikunjungi
3. Tabel Boolean *dikunjungi* bertipe *array of Boolean* dengan nilai elemen *true* jika simpul telah dikunjungi, atau *false* jika simpul belum dikunjungi.

2.4 Traveling Salesman Problem

Traveling Salesman Problem merupakan salah satu permasalahan yang dapat diselesaikan dengan strategi algoritma. Pada permasalahan ini, terdapat sejumlah kota dengan jarak antarkota diketahui. Lalu, terdapat seorang pedagang (*salesman*) yang ingin mengunjungi seluruh kota tepat satu kali. Kota berangkat dan pulang sang pedagang merupakan kota yang sama. Untuk efisiensi kerja, sang pedagang harus memilih rute dengan jarak terpendek untuk mengunjungi semua kota tepat satu kali dari kota awal menuju kota akhir yang sama. Adapun rute yang dipilih pada persoalan ini adalah dengan menentukan sirkuit Hamilton, yaitu “rute” yang mengunjungi semua simpul dengan bobot minimum.

2.5 C# Desktop Application Development

C# Desktop Application Development merupakan salah satu metode pengembangan aplikasi berbasis desktop yang memanfaatkan bahasa pemrograman C#. Bahasa pemrograman C# merupakan salah satu bahasa yang dikembangkan oleh Microsoft untuk mengembangkan aplikasi berbasis *web*, *mobile*, *games*, dan lain-lain. Adapun *framework* yang digunakan adalah .NET Winforms yang dapat dijalankan pada sistem operasi Windows. Dengan memanfaatkan IDE (Integrated Development Environment), misalnya Visual Studio, aplikasi dapat dikembangkan. Pada tugas besar ini, implementasi pengembangan aplikasi berbasis desktop ini adalah untuk membuat aplikasi pencari harta karun.

BAB III

APLIKASI STRATEGI BFS DAN DFS

3.1 Langkah-Langkah Pemecahan Masalah

3.1.1 Langkah Algoritma Depth First Search (DFS)

Algoritma DFS yang diterapkan untuk mencari rute pencarian harta karun. Secara garis besar, program akan menelusuri simpul-simpul mulai dari titik awal (Krusty Krab) dengan algoritma DFS. Penelusuran dilakukan dengan cara mengunjungi simpul-simpul dengan urutan prioritas arah RLDU (*right, left, down, up*). Penelusuran dilakukan secara rekursif hingga penelusuran menemui buntu. Apabila buntu, program melakukan *backtrack* ke simpul sebelumnya. Penelusuran berakhir jika semua *treasure* pada peta telah diambil semua.

1. Inisialisasi:
 - a. Solusi (*solution*) berupa *path* diinisialisasi sebagai string kosong.
 - b. Boolean *all treasure found* diinisialisasikan sebagai *false*.
 - c. List *path point* dengan elemen bertipe PointDir untuk menyimpan semua simpul yang pernah dikunjungi beserta dengan arahnya (arah berupa indeks integer yang unik untuk setiap arah, yaitu 0 untuk R, 1 untuk L, 2 untuk D, dan 3 untuk U).
 - d. Matriks *visited* diinisialisasi sebagai *array of array of bool* untuk menyimpan informasi simpul yang telah dikunjungi, dengan nilai default *false* untuk semua elemen.
 - e. Banyaknya *node* (simpul) yang dikunjungi / dievaluasi diinisialisasikan sebagai nilai integer 0.
2. Fungsi rekursif dimulai dengan mengunjungi grid titik awal (Krusty Krab):
 - a. Tandai elemen posisi saat ini pada matriks *visited* dengan *true*. Banyaknya *node* bertambah 1.
 - b. Jika simpul yang sedang dikunjungi adalah *treasure*, maka banyaknya *treasure* yang tersisa di peta berkurang 1.
 - c. Basis:
 - i. Jika banyaknya *treasure* yang tersisa di peta adalah 0, maka boolean *all treasure found* bernilai *true* dan fungsi rekursi selesai.
 - d. Rekursi:

- i. Kunjungi (iterasi) simpul-simpul di sekitar posisi saat ini, mulai dari kanan, kiri, bawah, atas. Simpul yang dapat dikunjungi adalah simpul berupa R (jalan) atau T (treasure). Simpul X tidak dapat dikunjungi. Simpul yang tidak dapat dikunjungi akan dilewati (*skipped*). Untuk setiap simpul yang dapat dikunjungi, konkatenasi *path* solusi dengan arah simpul tersebut dan panggil fungsi rekursi (ulangi langkah ke-2).
 - e. Jika tidak ada simpul yang dapat dikunjungi lagi (buntu) tetapi belum semua *treasure* diambil, maka lakukan *backtrack* dengan cara mengkonkatenasi *path* solusi dengan arah simpul yang dibalik (*reversed*).
3. Penelusuran selesai. Solusi (*path*) ditemukan dari titik awal sampai *treasure* terakhir. Banyaknya *node* yang diperiksa didapatkan. Banyaknya langkah didapatkan berupa panjang *path* solusi.

3.1.2 Langkah Algoritma Breadth First Search (BFS)

Algoritma BFS diterapkan untuk mencari rute pencarian harta karun. Secara garis besar, program akan menelusuri simpul-simpul mulai dari titik awal (Krusty Krab) dengan algoritma BFS. Penelusuran dilakukan dengan cara mencatat simpul-simpul aktif dengan Queue. Ketika satu *treasure* ditemukan, maka penelusuran dihentikan dan Queue dikosongkan. Jika masih ada *treasure* yang belum diambil, maka penelusuran dilakukan kembali dari posisi *treasure* yang terakhir diambil. Penelusuran akan berhenti jika semua *treasure* pada peta sudah diambil.

Algoritma BFS yang diterapkan adalah penelusuran melebar dengan urutan prioritas langkah ke kanan (*right*), kiri (*left*), bawah (*down*), dan atas (*up*). Jadi, simpul-simpul dibangkitkan berdasarkan urutan prioritas RLDU (*right, left, down, up*).

Berikut merupakan langkah algoritma BFS mencari rute pencarian harta karun.

1. Inisialisasi:
 - a. Solusi (*solution*) berupa path diinisialisasi sebagai string kosong.
 - b. Matriks *visited* diinisialisasi sebagai *array of array of bool* untuk menyimpan informasi simpul yang telah dikunjungi, dengan nilai *default false* untuk semua elemen.

- c. Queue diinisialisasi dengan elemen bertipe Point untuk menyimpan semua simpul yang aktif.
 - d. List *path point* dengan elemen bertipe PointDir untuk menyimpan semua simpul yang pernah dikunjungi beserta dengan arahnya (arah berupa indeks integer yang unik untuk setiap arah, yaitu 0 untuk R, 1 untuk L, 2 untuk D, dan 3 untuk U).
 - e. List posisi *treasure* yang sudah diambil sebagai *list of Point*.
 - f. Posisi saat ini (*current row* dan *current col*) diinisialisasikan dengan posisi titik awal (Krusty Krab)
 - g. Titik awal sementara (*temp start*) diinisialisasikan dengan posisi titik awal (Krusty Krab)
 - h. Tambahkan (*enqueue*) simpul saat ini ke *queue*.
 - i. Tambahkan simpul saat ini ke *path point* dengan arah -1 (titik awal tidak dicapai oleh arah mana pun).
2. Kunjungi simpul saat ini. Tandai elemen simpul saat ini pada matriks *visited* dengan *true*. Hapus kepala *queue* (*dequeue*), yaitu simpul saat ini. Cek apabila *treasure* yang tersisa pada peta adalah 0, maka lakukan langkah ke-6.
3. Kunjungi (iterasi) simpul-simpul di sekitar posisi saat ini, mulai dari kanan, kiri, bawah, atas. Simpul yang dapat dikunjungi adalah simpul berupa R (jalan) atau T (*treasure*). Simpul X tidak dapat dikunjungi. Setiap simpul yang dikunjungi ditambahkan (*enqueue*) ke *queue* dan *path point*, kemudian tandai elemen matriks *visited* dengan *true* untuk titik posisi simpul yang sedang dikunjungi.
- a. Jika simpul yang dikunjungi terdapat *treasure* dan *treasure* tersebut belum pernah diambil, maka buat path dari informasi titik-titik pada *path points* dan konkatenasi path tersebut dengan solusi. Tambahkan simpul yang dikunjungi tersebut ke list posisi *treasure* yang sudah diambil. Kosongkan *queue*. Kemudian, *queue* diisi kembali dengan simpul yang dikunjungi tersebut. Ubah semua elemen pada matriks *visited* menjadi *false*. Ubah titik awal sementara (*temp start*) menjadi simpul yang dikunjungi tersebut. Lakukan langkah ke-4.
 - b. Jika tidak, maka lanjutkan iterasi untuk simpul lain di sekitar posisi saat ini (langkah ke-3).
4. Ubah posisi saat ini menjadi posisi pada kepala *queue* (kunjungi titik berikutnya).

5. Jika belum semua *treasure* diambil dan *queue* belum kosong, ulangi dari langkah ke-2.
6. Penelusuran selesai. Solusi (*path*) ditemukan dari titik awal sampai *treasure* terakhir. Banyaknya *node* yang diperiksa adalah banyaknya elemen *path point*, sedangkan banyaknya langkah adalah panjang *path* solusi.

3.1.3 Langkah Algoritma Traveling Salesman Problem

Algoritma TSP diterapkan untuk mencari rute pencarian harta karun. Secara garis besar, algoritma TSP mencari rute terpendek untuk mengambil semua *treasure* yang terdapat pada peta. Program akan mencari rute terpendek dari semua kemungkinan urutan *treasure* yang akan diambil. Kemudian, penelusuran dilakukan secara BFS dari urutan titik-titik yang ditentukan. Penelusuran dengan rute terpendek akan diambil sebagai rute solusi.

1. Inisialisasi:
 - a. Nilai *count solution* diinisialisasikan sebagai nilai integer dengan nilai yang sangat besar.
2. Iterasi untuk semua kemungkinan (permutasi) urutan *treasure* yang akan diambil:
 - a. Inisialisasi:
 - i. Rute solusi diinisialisasikan dengan string kosong.
 - ii. List *path point* dengan elemen bertipe Point untuk menyimpan semua simpul yang pernah dikunjungi.
 - b. Iterasi:
 - i. Cari rute solusi dari titik awal (Krusty Krab) ke *treasure* pertama.
 - ii. Jika semua *treasure* sudah diambil, cari rute solusi dari titik *treasure* yang terakhir diambil ke titik awal (Krusty Krab). Jika masih ada *treasure* yang belum diambil, cari rute solusi dari titik *treasure* yang terakhir diambil ke titik *treasure* berikutnya. Konkatenasi rute solusi tersebut dengan rute solusi keseluruhan. Ulangi langkah ini sebanyak jumlah *treasure* yang ada pada peta.
3. Cari permutasi urutan *treasure* yang akan diambil dengan rute terpendek (panjang string rute solusi terpendek).
4. Penelusuran selesai. Solusi (*path*) terpendek ditemukan dari titik awal sampai *treasure* terakhir dengan urutan tertentu lalu kembali ke titik awal. Banyaknya *node* yang diperiksa

adalah banyaknya elemen *path point*, sedangkan banyaknya langkah adalah panjang *path* solusi.

3.2 Proses Mapping Persoalan Menjadi Elemen-Elemen Algoritma BFS dan DFS

Program akan menerima masukan sebuah *file* teks (.txt) yang akan dikonversi menjadi Map yang terdiri dari matriks peta, banyaknya baris dan kolom, titik awal penelusuran, banyaknya *treasure*, dan lokasi *treasure*. Kemudian, program juga akan menerima masukan pilihan metode penelusuran, yaitu DFS, BFS, atau TSP.

Untuk algoritma DFS, *mapping* persoalan adalah sebagai berikut:

1. Matriks boolean *visited* yang mencatat apakah suatu simpul sudah dikunjungi.
2. List *path point* yang mencatat simpul-simpul yang telah dikunjungi.
3. String *path* solusi dari titik awal penelusuran hingga posisi *treasure* yang terakhir ditemukan.

Untuk algoritma BFS, *mapping* persoalan adalah sebagai berikut:

1. Matriks boolean *visited* yang mencatat apakah suatu simpul sudah dikunjungi.
2. List *path point* yang mencatat simpul-simpul yang telah dikunjungi beserta arahnya.
3. Queue yang menyimpan simpul-simpul yang aktif.
4. List lokasi *treasure* yang sudah diambil / ditemukan.
5. String *path* solusi dari titik awal penelusuran hingga posisi *treasure* yang terakhir ditemukan.

Untuk algoritma TSP, *mapping* persoalan adalah sebagai berikut:

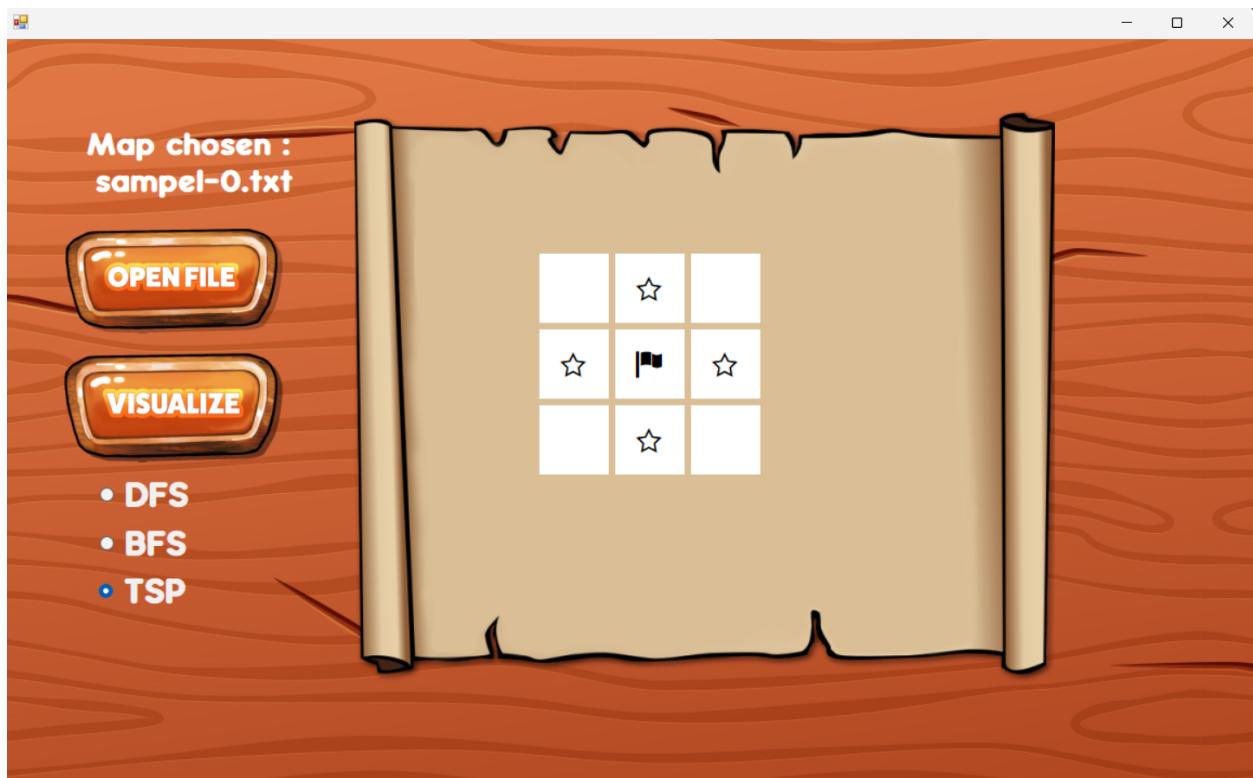
1. Integer *cntSolution* untuk mengecek panjang rute solusi.
2. List *path point* yang mencatat simpul-simpul yang telah dikunjungi.
3. String *path* solusi dari titik awal kemudian menelusuri semua *treasure* dan kembali ke titik awal.

3.3 Ilustrasi Kasus Lain

Misalkan terdapat peta dengan konfigurasi sebagai berikut:

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

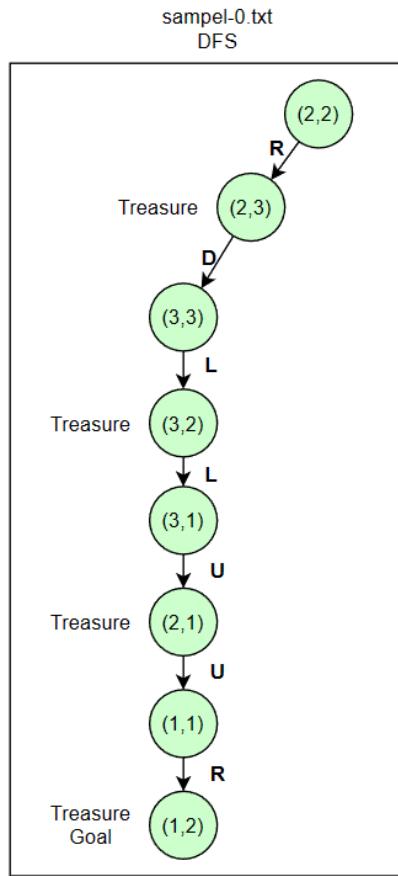
Peta berukuran 5×5 dengan 4 buah *treasure* yang berlokasi di grid (1,2), (2,1), (2,3), dan (3,2). Penelusuran dimulai dari Krusty Krab yang berlokasi di (2,2). Penelusuran rute pencarian harta karun dapat ditemukan oleh program. Adapun tampilan program untuk konfigurasi peta yang disimpan dalam file sampel-0.txt adalah sebagai berikut.



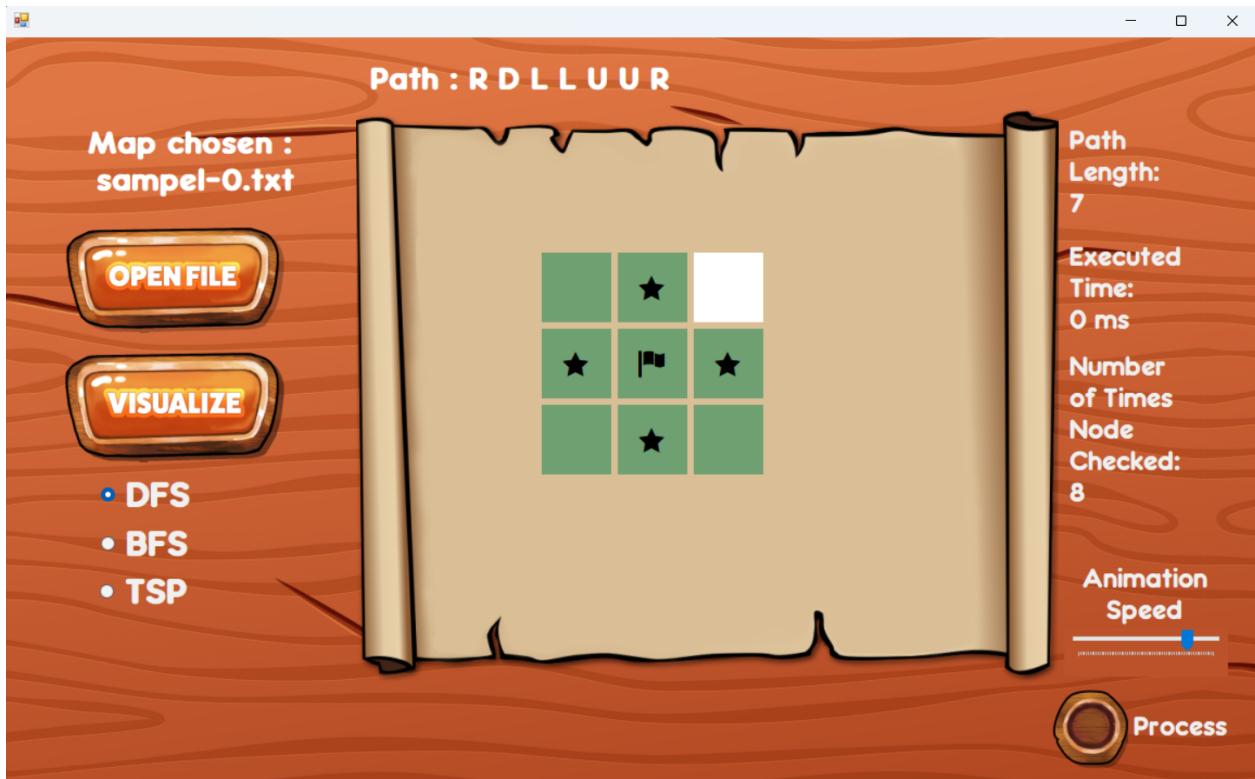
Gambar 3.3.1 Ilustrasi Kasus (Peta)

3.3.1 Ilustrasi dengan Algoritma Depth First Search (DFS)

Berikut merupakan visualisasi penyelesaian dengan algoritma DFS.



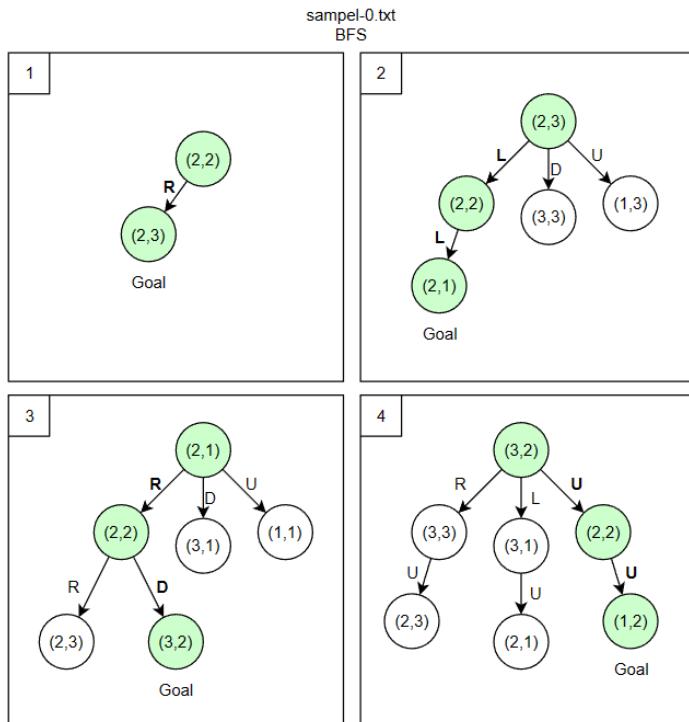
Gambar 3.3.1.1 Ilustrasi Kasus dengan DFS (Ilustrasi dengan Tree)



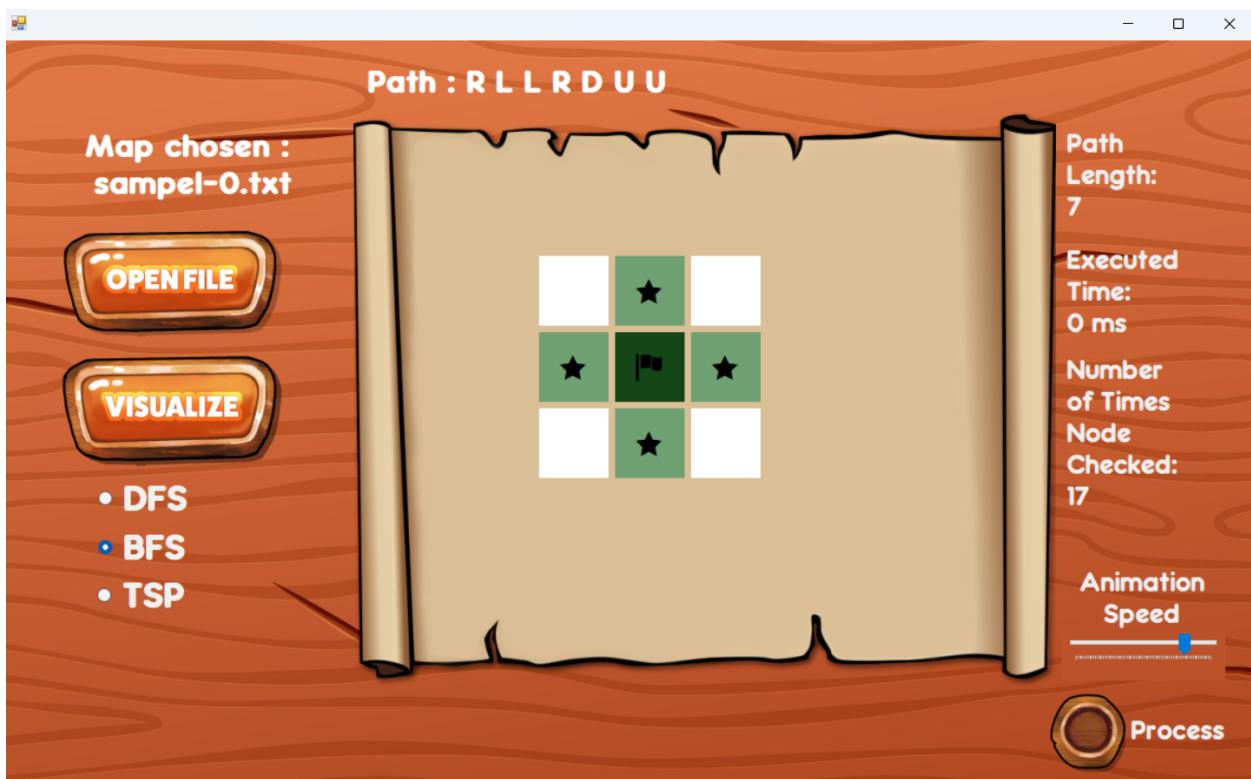
Gambar 3.3.1.2 Ilustrasi Kasus dengan DFS (Ilustrasi pada Peta)

3.3.2 Ilustrasi dengan Algoritma Breadth First Search (BFS)

Berikut merupakan visualisasi penyelesaian dengan algoritma BFS.



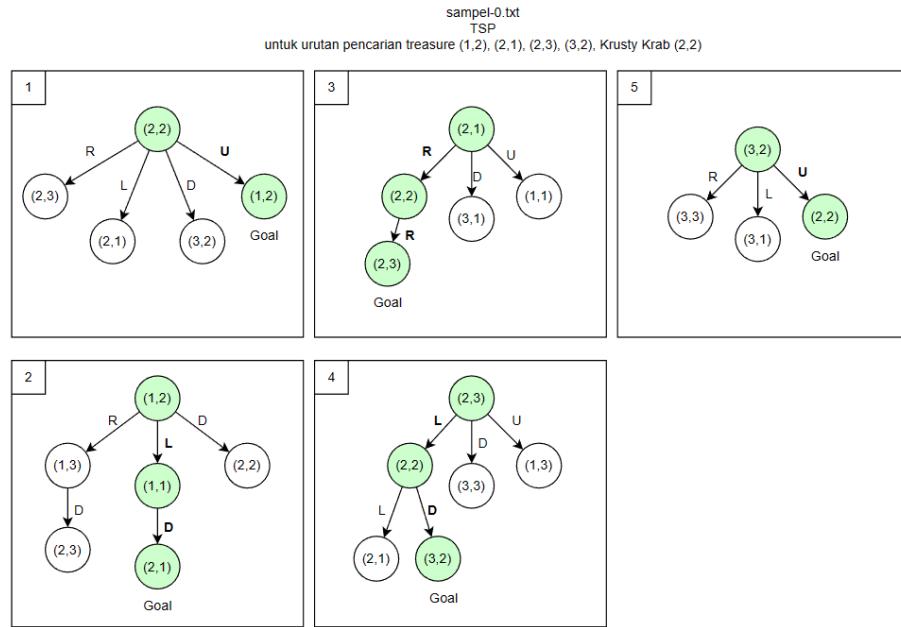
Gambar 3.3.2.1 Ilustrasi Kasus dengan BFS (Ilustrasi dengan Tree)



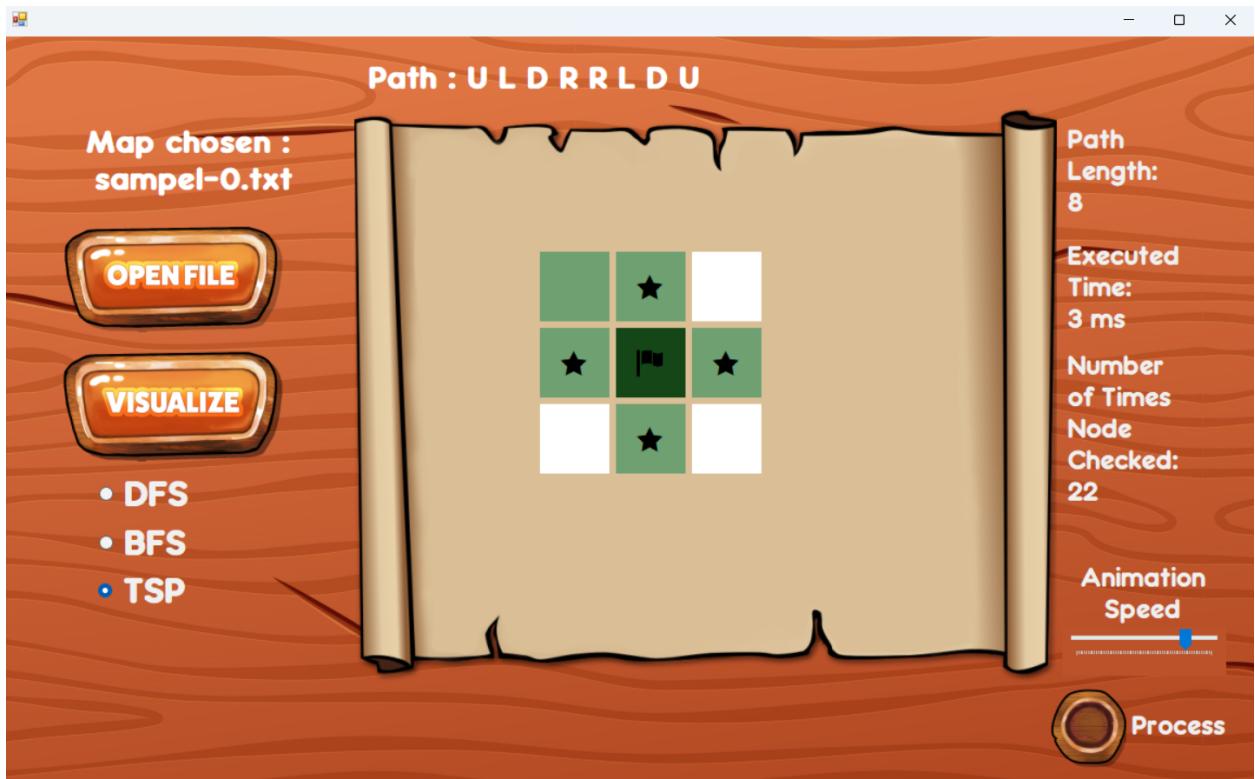
Gambar 3.3.2.2 Ilustrasi Kasus dengan BFS (Ilustrasi pada Peta)

3.3.3 Ilustrasi dengan Algoritma Traveling Salesman Problem (TSP)

Berikut merupakan visualisasi penyelesaian dengan algoritma TSP.



Gambar 3.3.3.1 Ilustrasi Kasus dengan TSP (Ilustrasi dengan Tree)



Gambar 3.3.3.2 Ilustrasi Kasus dengan TSP (Ilustrasi pada Peta)

BAB IV

IMPLEMENTASI DAN PENGUJIAN

4.1 Implementasi Program

Program diimplementasikan dengan menggunakan bahasa pemrograman C# dan GUI .NET 7.0. Pada antarmuka program, terdapat tombol untuk memilih *file* masukan, opsi algoritma penyelesaian, *field* visualisasi peta dalam bentuk grid, serta hasil penyelesaiannya, yaitu rute, banyaknya *node*, banyaknya langkah, dan waktu eksekusi. Visualisasi langkah-langkah penyelesaian juga dapat dilihat pada grid peta. Dengan demikian, program dapat menerima *input file* konfigurasi peta serta pilihan algoritma penelusuran, lalu mengeluarkan *output* visualisasi peta beserta langkah penelusuran dan juga informasi detail terkait penelusuran tersebut.

4.1.1 Implementasi Depth First Search (DFS)

```
namespace Solver {
    public class DFSSolver : Solver {
        static void DFS(int row, int col) {
            // Node check
            cntNode++
            visited[row, col] ← true
            // Found treasure
            if (map.grid[row, col] = 'T') then
                numOfTreasure--
            // All treasure found!
            if (numOfTreasure == 0) then
                allTreasureFound ← true
                return

            // All possible path to construct
            for i in range [0..4]
                newRow ← row + dy[i]
                newCol ← col + dx[i]
                if (newRow < 0 OR newRow ≥ map.rows OR newCol < 0 OR
                    newCol ≥ map.cols OR visited[newRow, newCol] OR
```

```

        map.grid[newRow, newCol] = 'X' OR allTreasureFound) then
            Continue

            // Recursive searching
            solution ← solution + direction[i]
            DFS(newRow, newCol)

            // Backtracking
            if (NOT allTreasureFound) then
                solution ← solution + reverseDirection[i]
            }

        public static void CallDFS(Map _map, ref string _solution, ref int
        _cntNode, ref List<Point> _pathPoints, ref long timeExec) {
            // Initiate
            cntNode ← 0
            allTreasureFound ← false
            solution ← ""
            map ← _map
            numOfTreasure ← map.numOfTreasure
            pathPoints ← new List<Point>() {
                new Point(map.start.rowId, map.start.colId) }

            visited ← new bool[map.rows, map.cols]

            // Time execution
            watch ← new System.Diagnostics.Stopwatch()
            watch.Start()
            DFS(map.start.rowId, map.start.colId)
            watch.Stop()

            // Construct pathPoints from the solution path
            for direction in range [0..solution.length()]
                switch (direction):
                    case 'R':
                        pathPoints.Add(new Point(pathPoints.Last().rowId+0,

```

```

        pathPoints.Last().colId + 1))
    break
case 'L':
    pathPoints.Add(new Point(pathPoints.Last().rowId+0,
    pathPoints.Last().colId - 1))
    break
case 'D':
    pathPoints.Add(new Point(pathPoints.Last().rowId+1,
    pathPoints.Last().colId + 0))
    break
case 'U':
    pathPoints.Add(new Point(pathPoints.Last().rowId-1,
    pathPoints.Last().colId + 0))
    break
default:
    output("Direction undefined!")
    break

_solution ← solution
_cntNode ← cntNode
_pathPoints ← pathPoints
timeExec ← watch.ElapsedMilliseconds
}
}
}

```

4.1.2 Implementasi Breadth First Search (BFS)

```

namespace Solver {
public class BFSSolver : Solver{
    public static void BFS(Map map, ref string sol, ref int num_node,
    ref List<Point> pathPoints, ref long timeExec){
        solution ← "" // solution path
        allTreasureFound ← false
        num0fTreasure ← map.num0fTreasure
    }
}
}

```

```

visited ← new bool[map.rows, map.cols] // is-visited info
queue ← new Queue<Point>{} // queue of to-be-visited nodes
(queue of Point)
pathPointDir ← new List<PointDir>(){} // list of PointDir
treasurePicked ← new List<Point>() {} // list of treasure
picked

// start timer
watch ← new System.Diagnostics.Stopwatch()
watch.Start()

// set current position
currentRow ← map.start.rowId
currentCol ← map.start.colId
tempStartRow ← currentRow
tempStartCol ← currentCol
queue.Enqueue(new Point(currentRow, currentCol))
// add first position to queue
pathPointDir.Add(new PointDir(currentRow, currentCol, -1))
// add first point to solution (x,y,direction index=-1)

while (NOT allTreasureFound AND queue.Count > 0) do
    visited[currentRow, currentCol] ← true; // visited
    queue.Dequeue() // dequeue

    if (numOfTreasure = 0) then
        allTreasureFound ← true // all treasure found
        break

    // visit node (direction priority: RLDU)
    for i in range [0..3]
        newRow ← currentRow + dy[i]
        newCol ← currentCol + dx[i]
        if (newRow < 0 OR newRow ≥ map.rows OR newCol < 0 OR
        newCol ≥ map.cols OR visited[newRow, newCol] OR
        map.grid[newRow, newCol]='X' OR

```

```

        allTreasureFound) then
            continue; // won't be visited
        // else
        queue.Enqueue(new Point(newRow, newCol)) //enqueue nodes
        pathPointDir.Add(new PointDir(newRow, newCol, i))
            // add to solution (x,y,direction index)
        visited[newRow, newCol] ← true // visited

        if (map.grid[newRow, newCol] = 'T' AND
        treasurePicked.FindIndex(p => p.rowId = newRow AND
        p.colId = newCol) = -1) then
            // treasure found! // mark as picked
            numOfTreasure-- // treasure found
            treasurePicked.Add(new Point(newRow, newCol))

            // create path
            createPath(pathPointDir, tempStartRow, tempStartCol,
            ref solution)

            // begin again
            queue.Clear() // delete queue
            queue.Enqueue(new Point(newRow, newCol))
                // add first node (the last treasure node /
                current position)
            visited ← new bool[map.rows, map.cols]
                // set all visited to false
            tempStartRow ← newRow // set new start point
            tempStartCol ← newCol // set new start point
            break

            // next
            currentRow ← queue.Head().rowId
            currentCol ← queue.Head().colId

        // stop watch
    
```

```

watch.Stop()

// result
pathPoints ← convertPathPoints(pathPointDir)
cntNode ← pathPoints.Count
num_node ← cntNode
sol ← solution
timeExec ← watch.ElapsedMilliseconds
}

public static void BFSOneGoal(Map map, Point start, Point end, ref
string solution, ref List<Point> _pathPoints)
{
    // Insert start point to the queue
    queue.Enqueue(new Point(start.rowId, start.colId))
    visited[start.rowId, start.colId] ← true
    from[start.rowId, start.colId] ← 'X'

    while (queue.Count > 0) do
        // Get the top point of the queue then pop
        curRow ← queue.Head().rowId
        curCol ← queue.Head().colId
        queue.Dequeue()

        // Add point to the pathPoints
        pathPoints.Add(new Point(curRow, curCol))

        // Check if current point = end point
        if (curRow = end.rowId AND curCol = end.colId) then
            break

        // Visit possible node (direction priority: RLDU)
        for i in range [0..3]
            newRow ← curRow + dy[i]
            newCol ← curCol + dx[i]
}

```

```

    // Skip condition
    if (newRow < 0 OR newRow ≥ map.rows OR newCol < 0 OR
    newCol ≥ map.cols OR visited[newRow, newCol] OR
    map.grid[newRow, newCol] = 'X') then
        continue

    // Add to the queue, set to visited, and from direction
    queue.Enqueue(new Point(newRow, newCol))
    visited[newRow, newCol] ← true
    from[newRow, newCol] ← direction[i]

    // create path (backtracking)
    curRow ← end.rowId
    curCol ← end.colId
    while (from[curRow, curCol] ≠ 'X') do
        solution ← from[curRow, curCol] + solution
        switch (from[curRow, curCol]):
            case 'R':
                curCol--
                break
            case 'L':
                curCol++
                break
            case 'D':
                curRow--
                break
            case 'U':
                curRow++
                break
            default:
                output("Direction undefined!")
                Break

    _pathPoints ← pathPoints

```

```
    }  
}
```

4.1.3 Implementasi Traveling Salesman Problem

```
namespace Solver {  
  
    public class TSPSolver : Solver {  
        // Function to find all permutation  
        static IEnumerable<IEnumerable<Point>> GetPermutations(List<Point>  
list) {  
            if (list.Count = 0) then  
                → Enumerable.Empty<Point>()  
            else  
                for i in range [0..list.Count]  
                    item ← list[i]  
                    remaining ← new List<Point>(list)  
                    remaining.RemoveAt(i)  
                    for permutation in GetPermutations(remaining)  
                        →new[ ]{item}.Concat(permutation)  
        }  
  
        public static void CallTSP(Map _map, ref string _solution, ref int  
_cntNode, ref List<Point> _pathPoints, ref long timeExec) {  
            // Initiate  
            map ← _map  
            cntSolution ← int.MaxValue  
  
            // Time execution  
            watch ← new System.Diagnostics.Stopwatch()  
            watch.Start()  
  
            // Find & iterate over all permutation  
            allPermutations ← GetPermutations(_map.treasureLoc)  
            for permutation in allPermutations  
                // Initiate for current permutation
```

```

    permutationSolution ← ""
    permutationPathPoints ← new List<Point>()

    // Iterate current permutation
    for i in range [0..map.numOfTreasure]
        // Initiate for current route
        curSolution ← ""
        curPathPoints ← new List<Point>()

        // Find the solution from start to first treasure
        if (i = 0) then
            BFSSolver.BFSOneGoal(map, map.start,
                permutation.ElementAt(i), ref curSolution, ref
                curPathPoints)

        // Find the solution from last treasure to start
        else if (i = map.numOfTreasure) then
            BFSSolver.BFSOneGoal(map, permutation.ElementAt(i - 1),
                map.start, ref curSolution, ref curPathPoints)
            curPathPoints.RemoveAt(0)

        // Find the solution between 2 treasure
        else
            BFSSolver.BFSOneGoal(map, permutation.ElementAt(i - 1),
                permutation.ElementAt(i), ref curSolution, ref
                curPathPoints)
            curPathPoints.RemoveAt(0)

        // Add to the current permutation
        permutationSolution ← permutationSolution + curSolution
        permutationPathPoints.AddRange(curPathPoints)

        // Change the solution if taken shorter path in current
        // permutation
        if (permutationSolution.Length < cntSolution) then

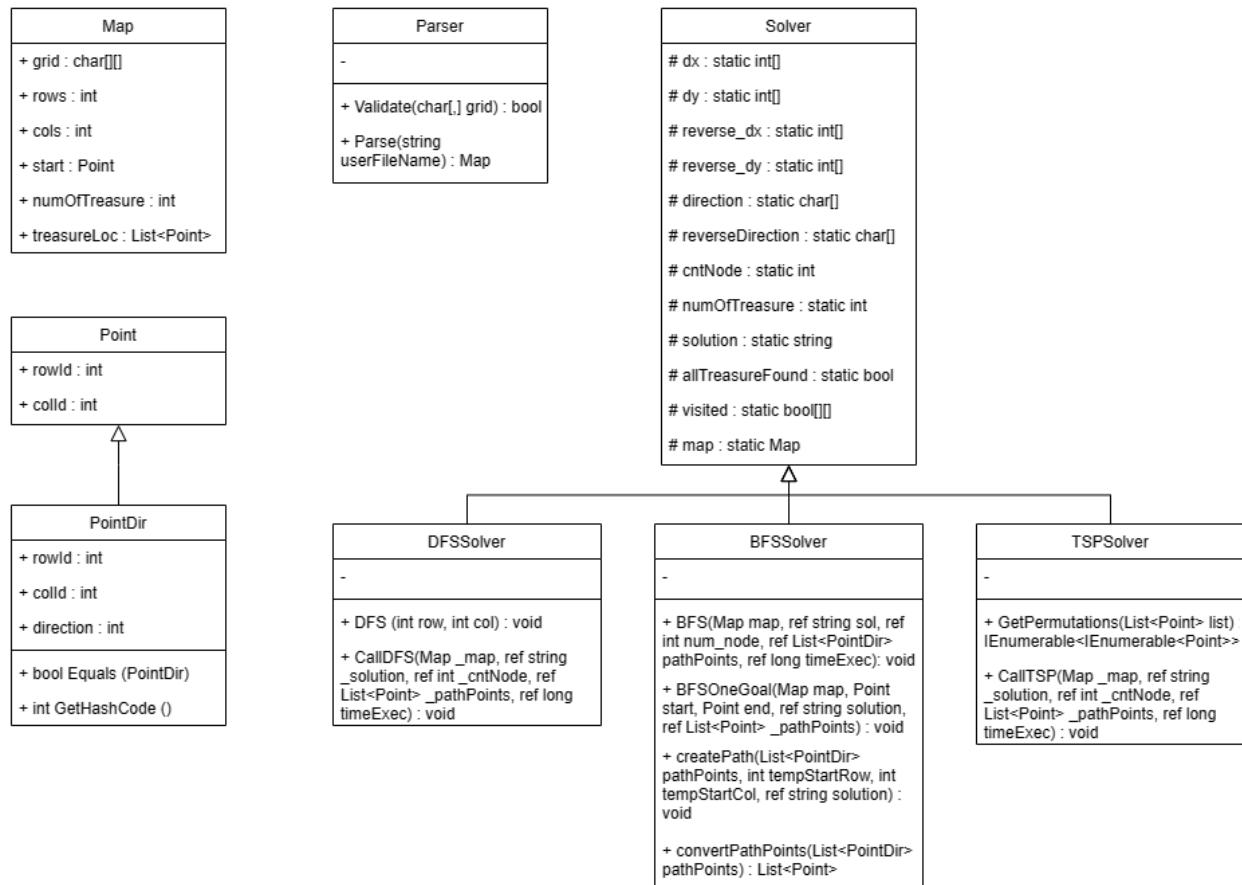
```

```
        cntSolution ← permutationSolution.Length
        _solution ← permutationSolution
        _pathPoints ← permutationPathPoints

        watch.Stop()
        _cntNode ← _pathPoints.Count
        timeExec ← watch.ElapsedMilliseconds
    }
}
}
```

4.2 Struktur Data

Program dirancang dengan pemrograman berorientasi objek. Adapun struktur data program ini terdiri dari sebuah namespace bernama Solver. Di dalam solver, terdapat beberapa kelas, yaitu Map, Point, PointDir, Solver, DFSSolver, BFSSolver, TSPSolver, dan Parser.



Gambar 4.2.1 Diagram Kelas (Struktur Data) Solver

Berikut merupakan penjelasan mengenai struktur data masing-masing kelas.

4.2.1 Map

Kelas Map merupakan kelas yang merepresentasikan peta pada program pencari rute Treasure Hunt.

Atribut:

1. grid: array of array of char, berisi peta
2. rows: int, merepresentasikan jumlah baris pada peta
3. cols: int, merepresentasikan jumlah kolom pada peta
4. start: Point, menyimpan titik mulai pencarian pada peta dalam notasi (row,col)
5. numOfTreasure: int, menyimpan banyaknya treasure pada peta.
6. treasureLoc : list of Point, berisi titik-titik lokasi *treasure* pada peta.

Konstruktor:

1. Konstruktor kelas Map dibuat dengan parameter bertipe array of array of char berupa matriks peta. Konstruktor akan mengiterasi matriks tersebut dan menentukan banyaknya baris dan kolom, banyaknya treasure beserta lokasinya pada peta, serta menentukan titik mulai pencarian pada peta.

4.2.2 Point

Kelas Point merupakan kelas yang merepresentasikan titik-titik pada peta yang terdiri dari indeks baris dan kolom peta.

Atribut:

1. rowId: int, merepresentasikan indeks / posisi baris pada peta
2. colId: int, merepresentasikan indeks / posisi kolom pada peta

Konstruktor:

1. Konstruktor kelas Point dibuat dengan dua parameter yang keduanya bertipe int, yaitu parameter yang merepresentasikan rowId dan juga colId.

4.2.2 PointDir

Kelas PointDir merupakan kelas turunan dari kelas Point yang merepresentasikan titik-titik pada peta yang terdiri dari indeks baris dan kolom peta disertai dengan indeks arah penelusuran.

Atribut:

1. direction: int [0..3], merepresentasikan indeks arah penelusuran titik. Atribut ini bernilai 0 jika arah penelusuran ke kanan (R), 1 jika ke kiri (L), 2 jika ke bawah (D), dan 3 jika ke atas (U).

Konstruktor:

1. Konstruktor kelas PointDir dibuat dengan tiga parameter yang seluruhnya bertipe int, yaitu parameter yang merepresentasikan rowId, colId, dan juga direction.

Metode:

1. bool Equals(object p): metode yang meng-*override* fungsi operator pembanding untuk tipe objek PointDir. Metode mengembalikan *true* jika aspek rowId, colId, dan direction dari dua objek PointDir sama. Metode mengembalikan *false* jika minimal terdapat salah satu aspek rowId, colId, atau direction dari dua objek PointDir berbeda.
2. int GetHashCode(): metode yang meng-*override* fungsi Hash Map. Metode ini mengembalikan kode hash yang unik untuk setiap elemennya.

4.2.3 Solver

Kelas Solver merupakan kelas yang merepresentasikan cara penyelesaian masalah mencari rute harta karun.

Atribut:

1. dx: array of int, berisi selisih kolom suatu titik ke titik-titik di sekitarnya, yaitu kanan, kiri, bawah, dan atas. Larik ini digunakan untuk mengevaluasi titik-titik di sekitar suatu titik dengan urutan prioritas RLDU (*right, left, down, up*). Selisih kolom suatu titik ke titik di sebelah kanannya adalah 1. Selisih kolom suatu titik ke titik di sebelah kirinya adalah -1. Selisih kolom suatu titik ke titik di sebelah bawah dan atasnya adalah 0.
2. dy: array of int, berisi selisih baris suatu titik ke titik-titik di sekitarnya, yaitu kanan, kiri, bawah, dan atas. Selisih baris suatu titik ke titik di sebelah kanan dan kirinya adalah 0. Selisih baris suatu titik ke titik di sebelah bawahnya adalah 1. Selisih baris suatu titik ke titik di sebelah atasnya adalah -1.
3. Reverse_dx: array of int, berisi selisih kolom suatu titik ke titik-titik di sekitarnya dengan arah yang dibalik, sehingga prioritas RLDU dibalik menjadi LRUD (*left, right, up, down*).
4. reverse_dy: array of int, berisi selisih baris suatu titik ke titik-titik di sekitarnya dengan arah yang dibalik.
5. direction: array of char, berisi arah evaluasi titik dalam notasi huruf, yaitu R, L, D, atau U.
6. reverseDirection: array of char, berisi arah evaluasi titik yang dibalik (*reversed*), yaitu L, R, U, atau D.
7. cntNode: int, merepresentasikan banyaknya node / simpul (titik) yang dievaluasi selama pencarian rute.

8. `numOfTreasure`: int, merepresentasikan banyaknya *treasure* yang perlu diambil pada peta.
9. `solution`: string, merepresentasikan rangkaian urutan arah gerakan untuk mencapai solusi, terdiri dari kombinasi huruf R, L, D, dan U.
10. `allTreasureFound`: bool, bernilai *true* jika seluruh *treasure* telah diambil atau *false* jika belum.
11. `visited`: array of array of bool, berisi nilai *true* atau *false* untuk setiap elemen. Elemen merepresentasikan titik pada peta. Elemen bernilai *true* jika titik tersebut telah dikunjungi, atau *false* jika belum.
12. `map`: Map, merepresentasikan peta yang ingin dicari rute harta karunnya.

4.2.4 DFSSolver

Kelas DFSSolver merupakan kelas turunan dari kelas Solver yang merepresentasikan cara penyelesaian masalah mencari harta karun dengan algoritma DFS.

Metode:

1. `void DFS(int row, int col)`: metode untuk melakukan penelusuran DFS secara rekursif terhadap suatu titik yang diidentifikasi dengan `row` dan `col`.
2. `void CallDFS(Map _map, ref string _solution, ref int _cntNode, ref List<Point> _pathPoints, ref long timeExec)`: metode untuk melaksanakan penelusuran DFS untuk peta yang terdapat pada parameternya. Metode mengembalikan hasil dalam bentuk *reference*, yaitu rute solusi (*string*), jumlah simpul yang dikunjungi (*int*), list simpul yang dikunjungi (*list of Point*), dan lama waktu eksekusi (*long*).

4.2.5 BFSSolver

Kelas BFSSolver merupakan kelas turunan dari kelas Solver yang merepresentasikan cara penyelesaian masalah mencari harta karun dengan algoritma BFS.

Metode:

1. `void BFS(Map map, ref string sol, ref int num_node, ref List<Point> pathPoints, ref long timeExec)`: metode untuk melaksanakan penelusuran BFS untuk seluruh *treasure* yang terdapat pada peta. Metode mengembalikan hasil dalam bentuk *reference*, yaitu rute

solusi (*string*), jumlah simpul yang dikunjungi (*int*), list simpul yang dikunjungi (*list of Point*), dan lama waktu eksekusi (*long*).

2. void BFSOneGoal(Map map, Point start, Point end, ref string solution, ref List<Point> _pathPoints): metode untuk melaksanakan penelusuran BFS dari titik awal sampai titik akhir yang ditentukan pada parameter. Metode mengembalikan hasil dalam bentuk reference, yaitu rute solusi (*string*) dan list simpul yang dikunjungi (*list of Point*).
3. List<Point> convertPathPoints(List<PointDir> pathPoints): metode untuk mengkonversi *list of PointDir* menjadi *list of Point*. Larik dengan elemen bertipe PointDir dibutuhkan untuk membuat *path* solusi, sedangkan tampilan luar hanya membutuhkan larik dengan elemen bertipe Point, sehingga tipe elemen larik perlu dikonversikan.

4.2.6 TSPSolver

Kelas TSPSolver merupakan kelas turunan dari kelas Solver yang merepresentasikan cara penyelesaian masalah mencari harta karun dengan algoritma Traveling Salesman Problem (TSP).

Metode:

1. IEnumerable<IEnumerable<Point>> GetPermutations(List<Point> list): metode untuk memperoleh semua kemungkinan permutasi urutan titik.
2. void CallTSP(Map _map, ref string _solution, ref int _cntNode, ref List<Point> _pathPoints, ref long timeExec): metode untuk melaksanakan penelusuran TSP dari titik awal, hingga mengambil semua *treasure*, lalu kembali ke titik awal. Metode mengembalikan hasil dalam bentuk reference, yaitu rute solusi (*string*), jumlah simpul yang dikunjungi (*int*), list simpul yang dikunjungi (*list of Point*), dan lama waktu eksekusi (*long*).

4.2.7 Parser

Kelas Parser merupakan kelas yang merepresentasikan cara pembacaan peta dari suatu *file teks* (.txt).

Metode:

1. bool Validate(char[,] grid): metode untuk pengecekan apakah peta yang dibaca valid atau tidak. Peta yang valid adalah teks yang terdiri dari huruf ‘K’, ‘R’, atau ‘X’, serta tiap huruf dipisahkan dengan spasi.

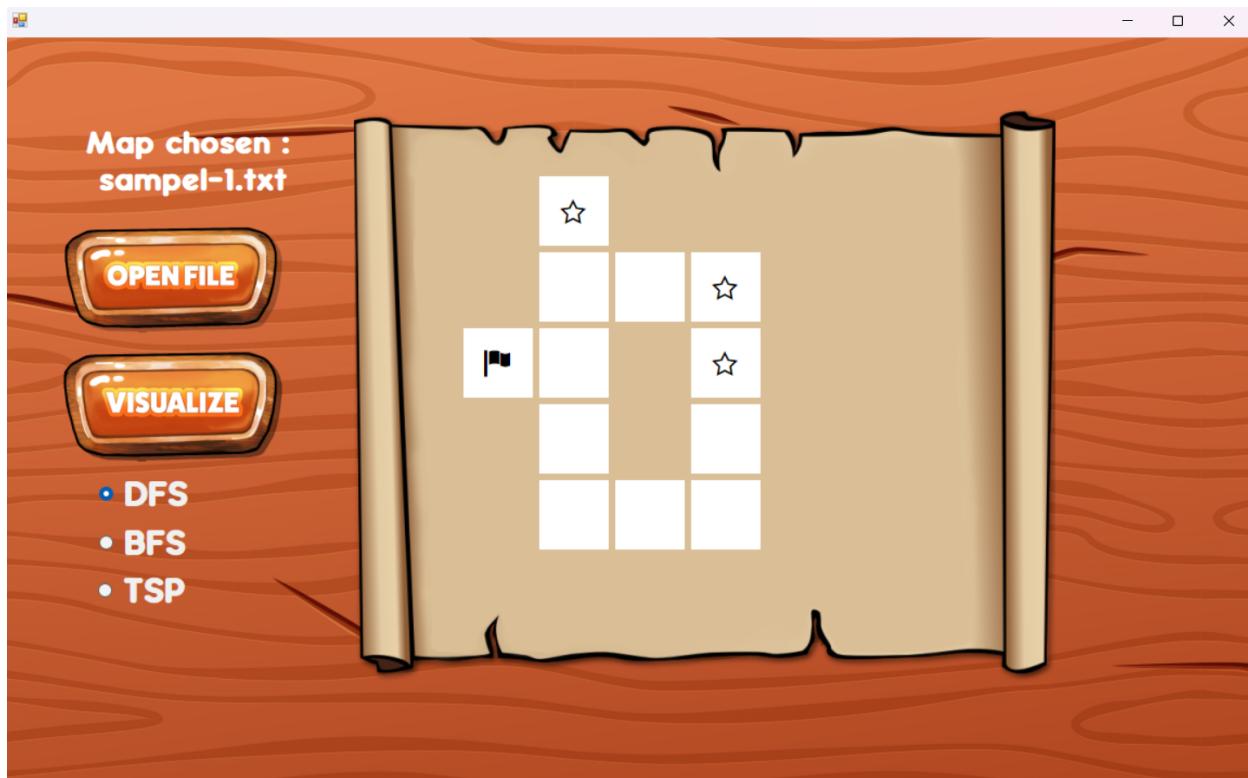
2. Map Parse(string userFileName): metode untuk membaca *file* teks peta dan mengubahnya ke dalam bentuk Map. Apabila peta yang dibaca tidak valid, maka metode akan melemparkan *exception* dengan pesan “File tidak valid!”.

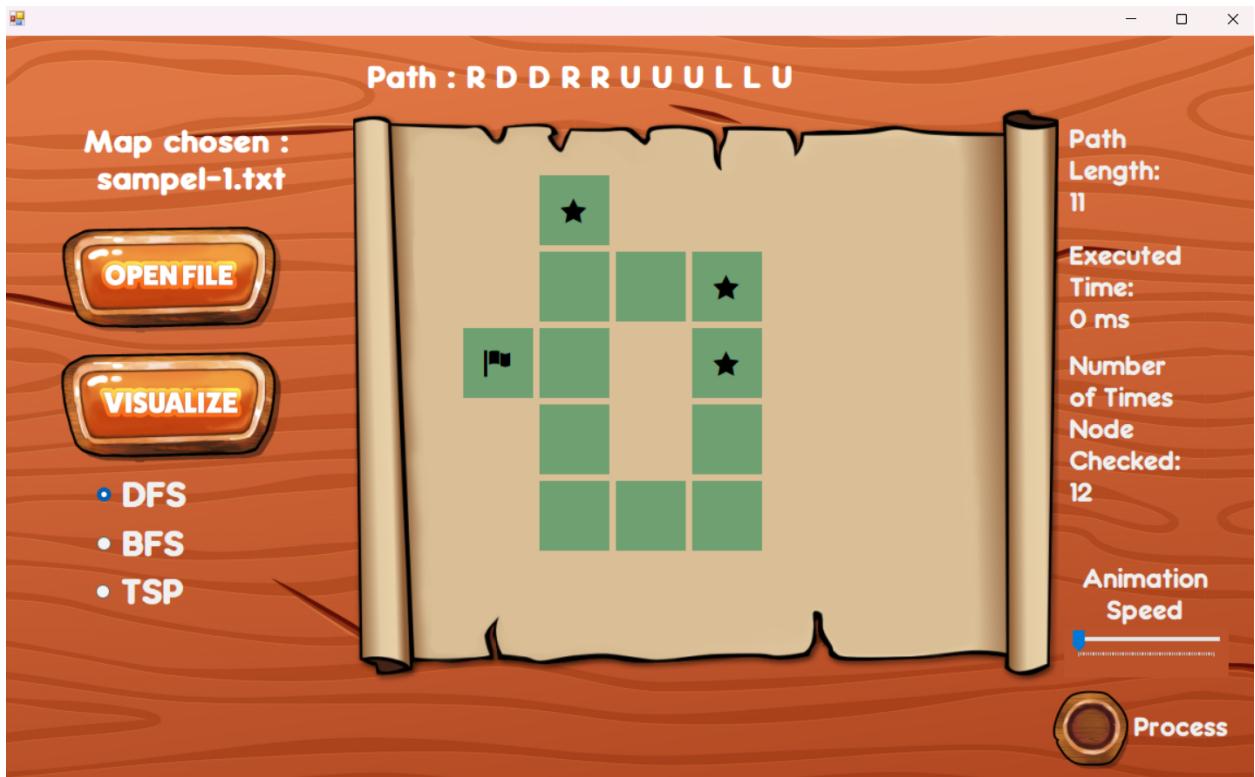
4.3 Tata Cara Penggunaan Program

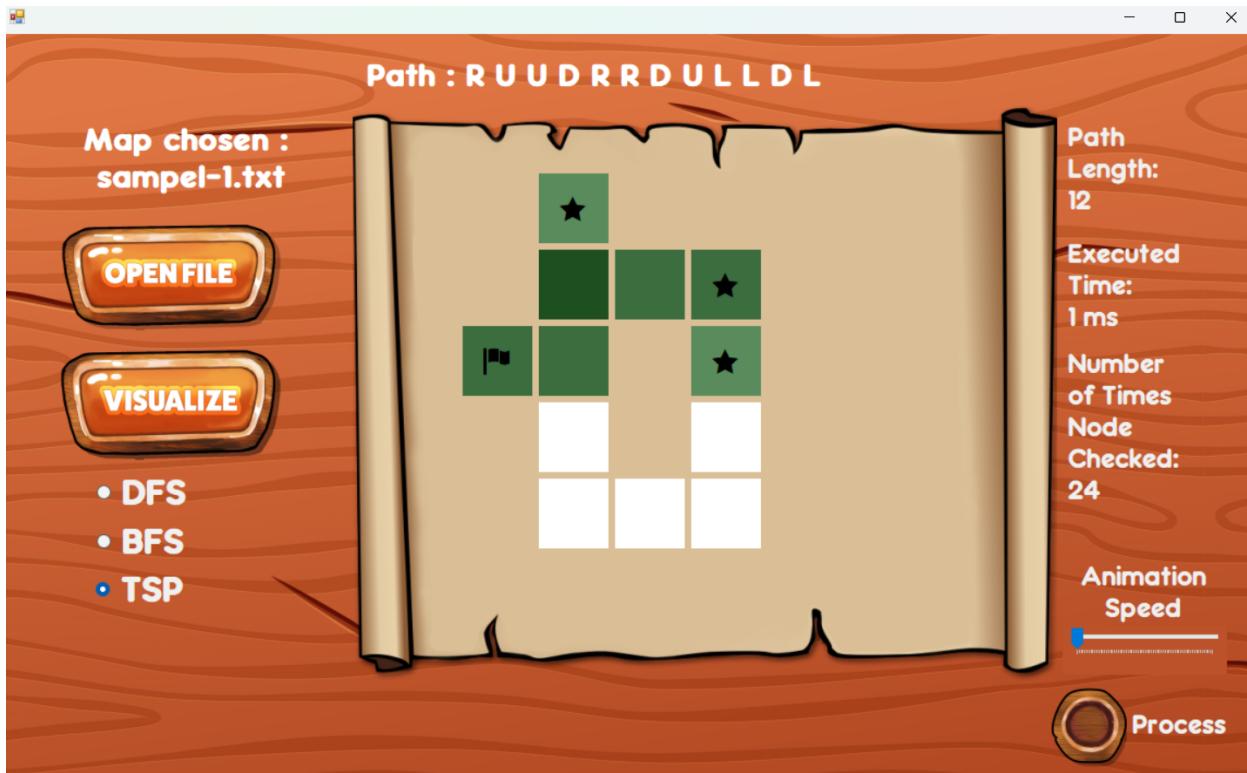
Jalankan TreasureHunt.exe yang tersimpan pada folder “bin”. Untuk menggunakan program, pengguna hanya perlu memilih *file* konfigurasi peta melalui tombol “open file” pada bagian kiri atas. Jika konfigurasi peta valid, maka peta divisualisasikan dalam bentuk grid. Jika tidak valid, maka program akan menampilkan pesan *error*. Pengguna juga dapat memilih algoritma penelusuran, yaitu DFS, BFS, dan TSP pada pilihan opsi pada kiri bawah layar. Setelah memilih, pengguna menekan tombol “visualize” untuk mendapatkan hasil rute solusi penelusuran. Urutan rute solusi akan ditampilkan pada bagian atas peta. Detail solusi, seperti panjang rute, waktu eksekusi, dan banyaknya *node* yang dicek, ditampilkan pada bagian kanan layar.

4.4 Hasil Pengujian

4.4.1 Pengujian 1







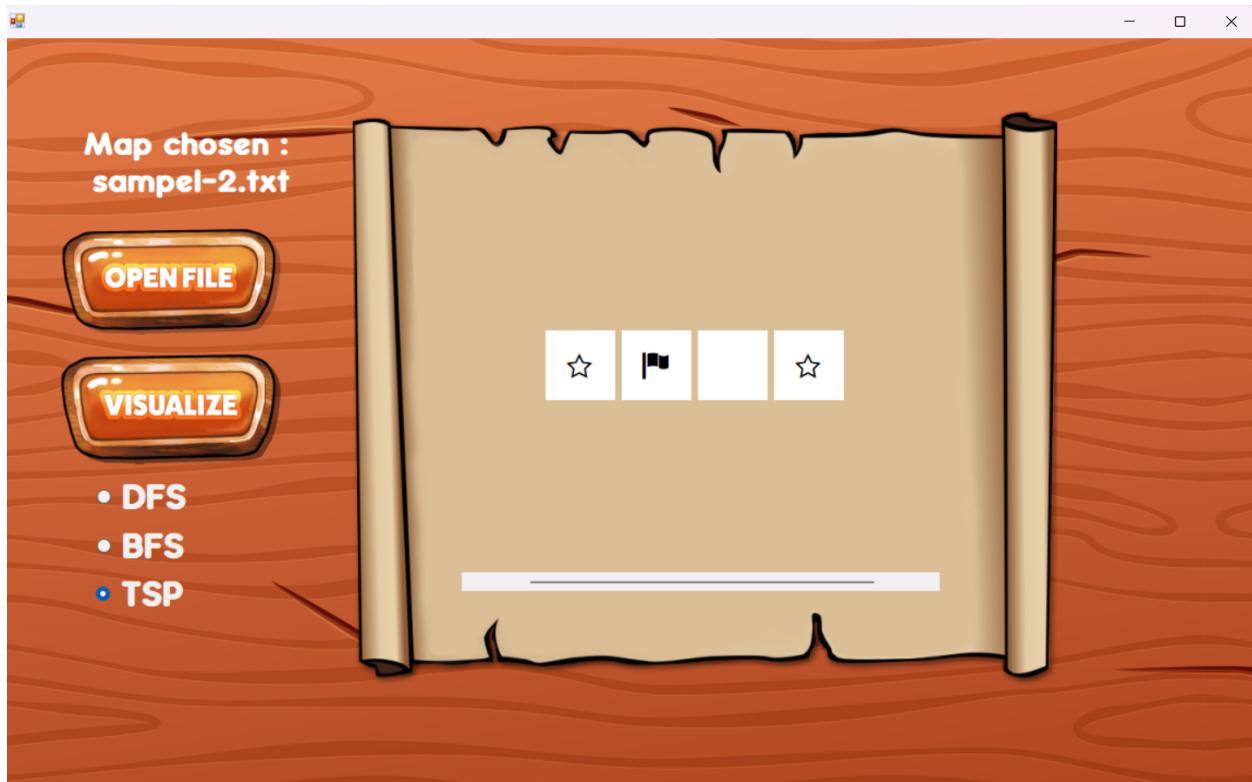
Gambar 4.4.1 Pengujian 1

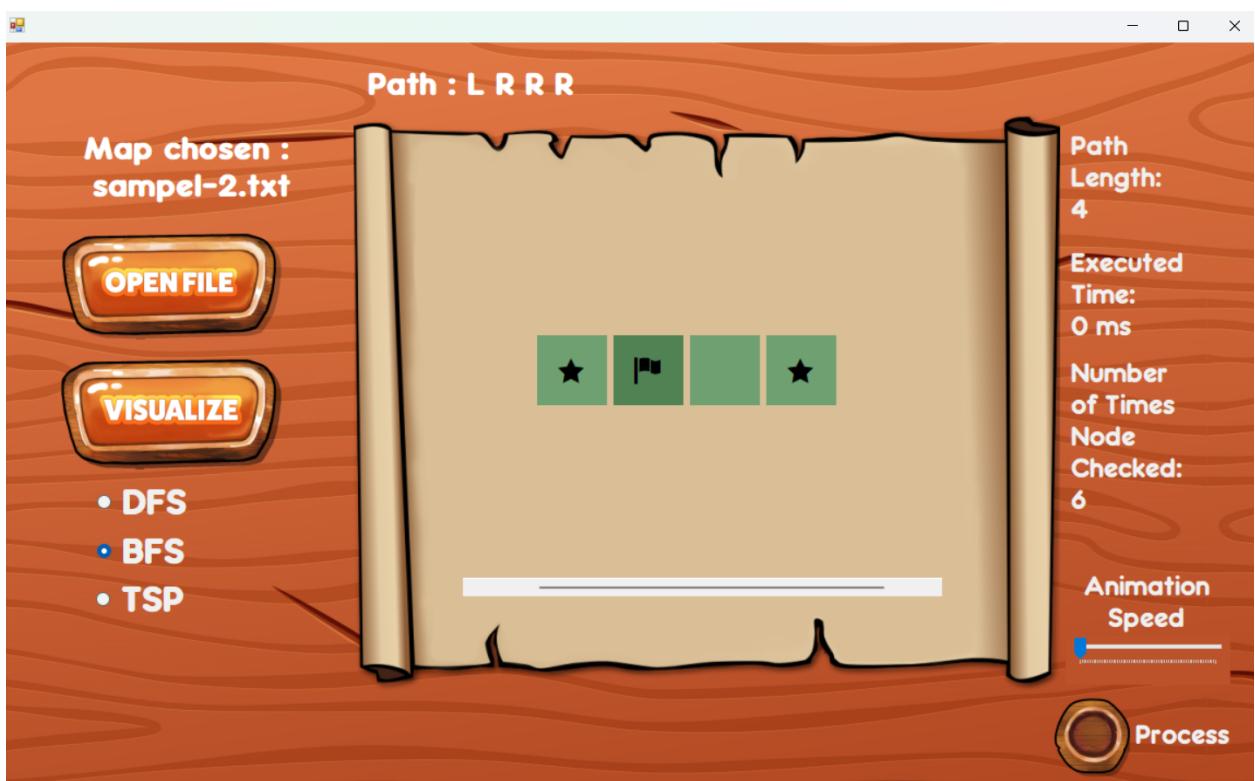
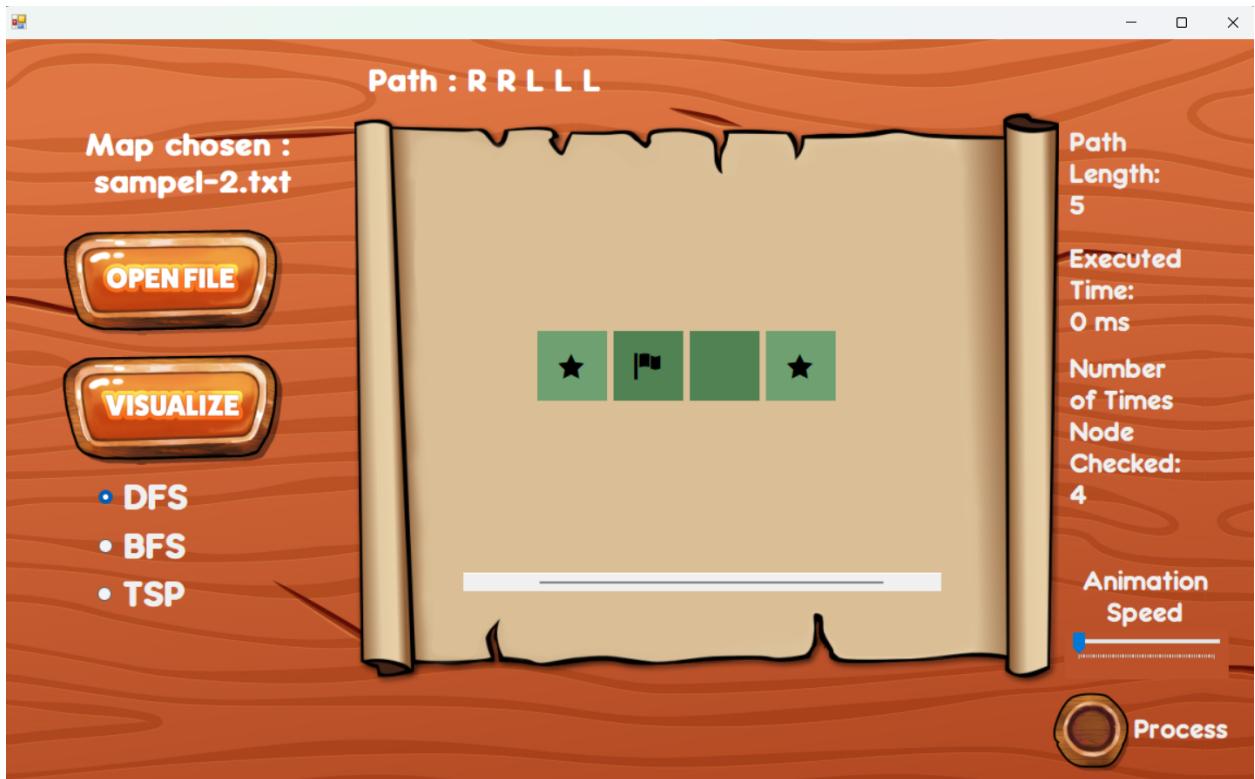
Tabel 4.4.1 Hasil Pengujian 1

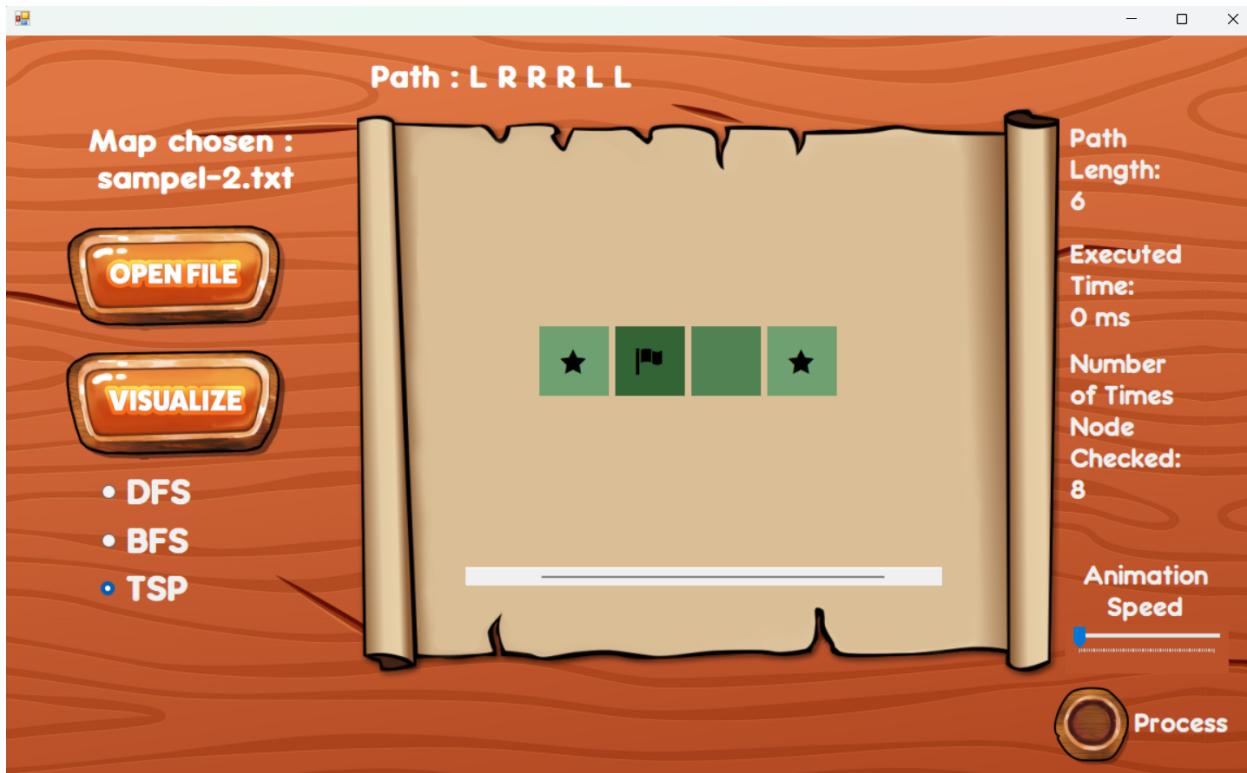
Nama file	sampel-1.txt
DFS	
Rute	R D D R R U U U L L U
Banyak node	12
Banyak langkah	11
Waktu eksekusi	0 ms
BFS	
Rute	R U U D R R D
Banyak node	13
Banyak langkah	7
Waktu eksekusi	4 ms
TSP	

Rute	R U U D R R D U L L D L
Banyak node	24
Banyak langkah	12
Waktu eksekusi	4 ms

4.4.2 Pengujian 2







Gambar 4.4.2 Pengujian 2

Tabel 4.4.2 Hasil Pengujian 2

Nama file	sampel-2.txt
DFS	
Rute	R R L L L
Banyak node	4
Banyak langkah	5
Waktu eksekusi	0 ms
BFS	
Rute	L R R R
Banyak node	6
Banyak langkah	4
Waktu eksekusi	0 ms
TSP	

Rute	L R R R L L
Banyak node	8
Banyak langkah	6
Waktu eksekusi	0 ms

4.4.3 Pengujian 3

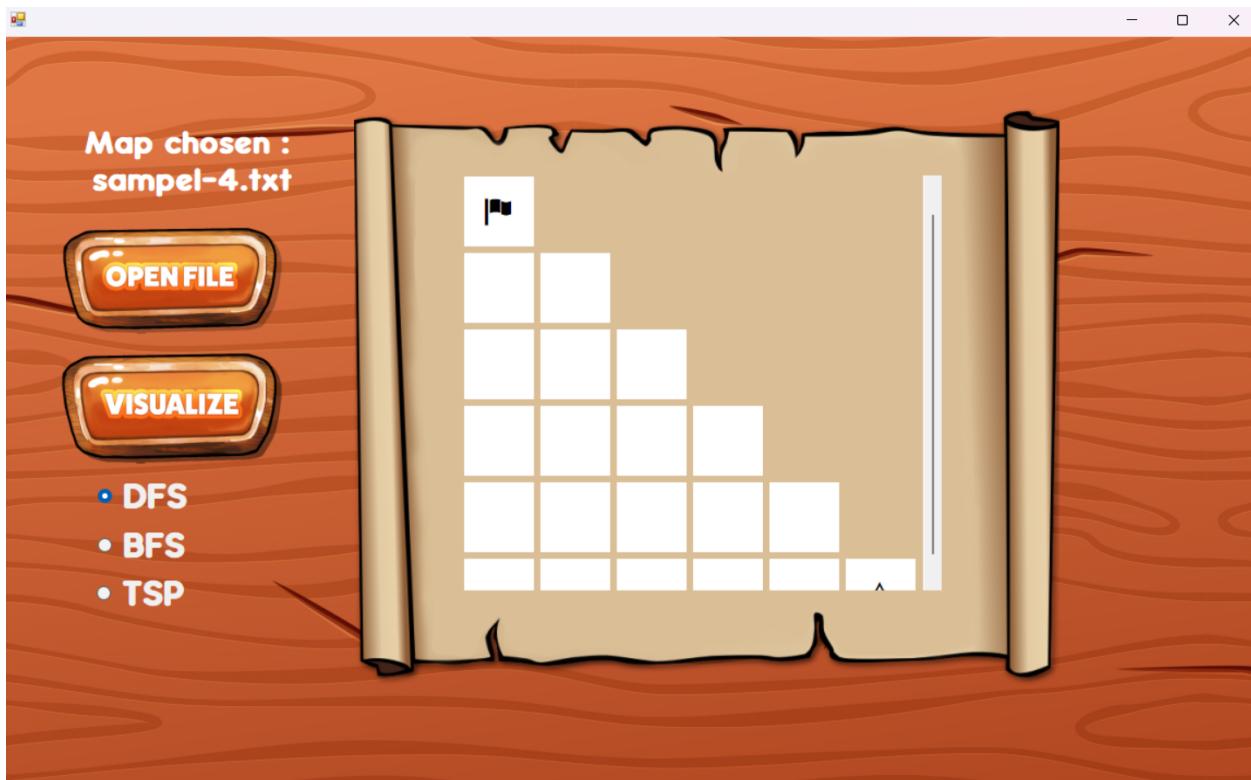


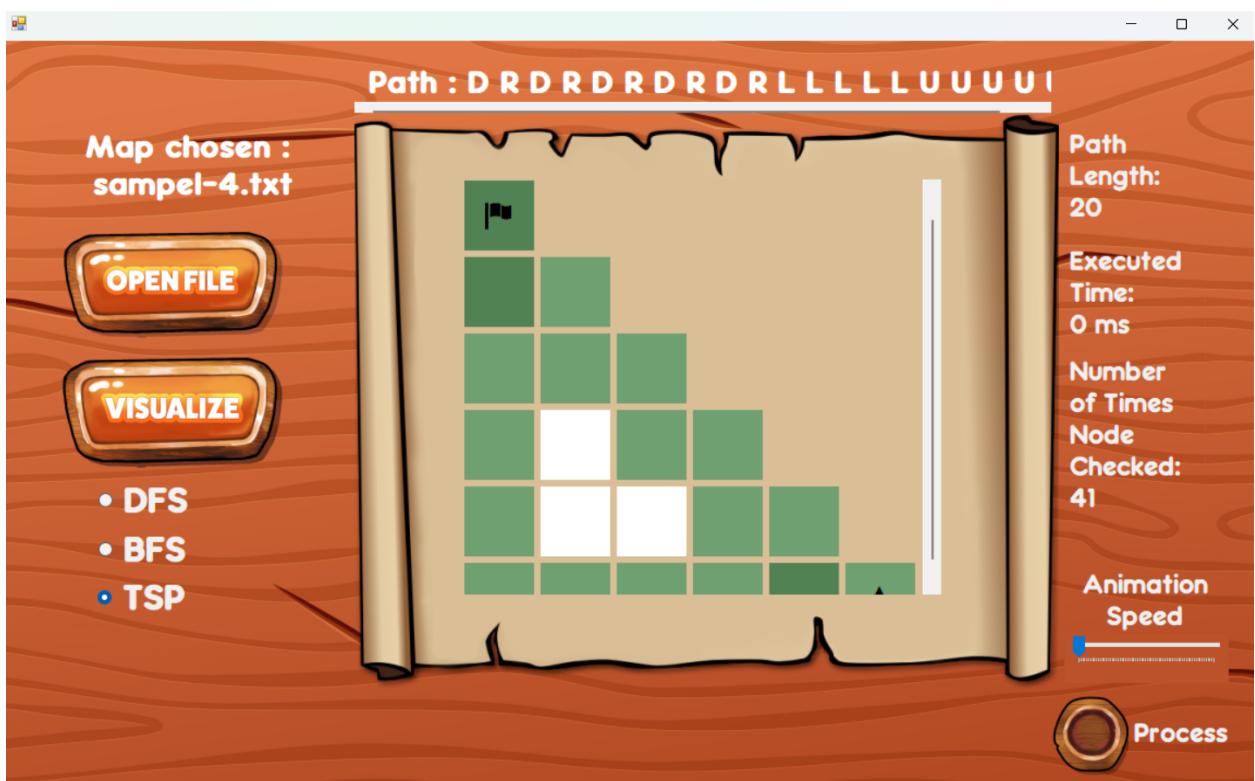
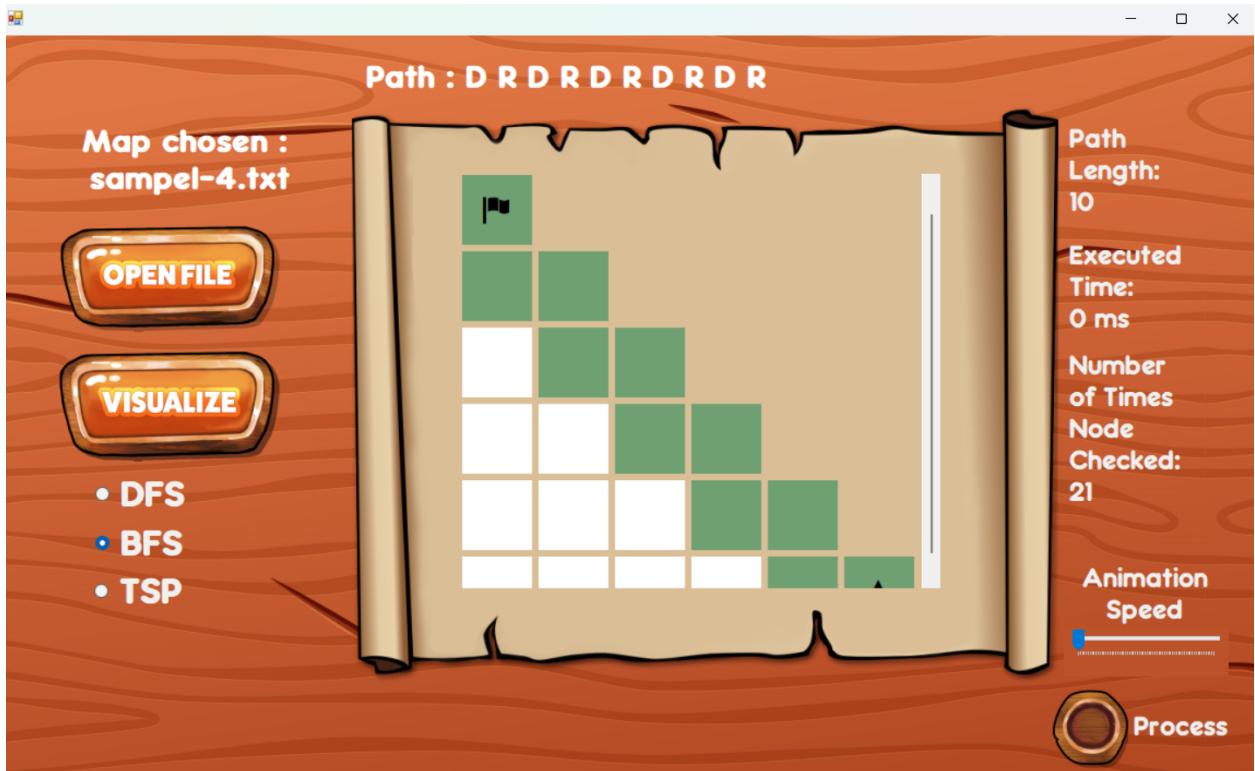
Gambar 4.4.3 Pengujian 3

Tabel 4.4.3 Hasil Pengujian 3

Nama file	sampel-3.txt
Hasil tidak dapat diperoleh karena peta tidak valid.	

4.4.4 Pengujian 4



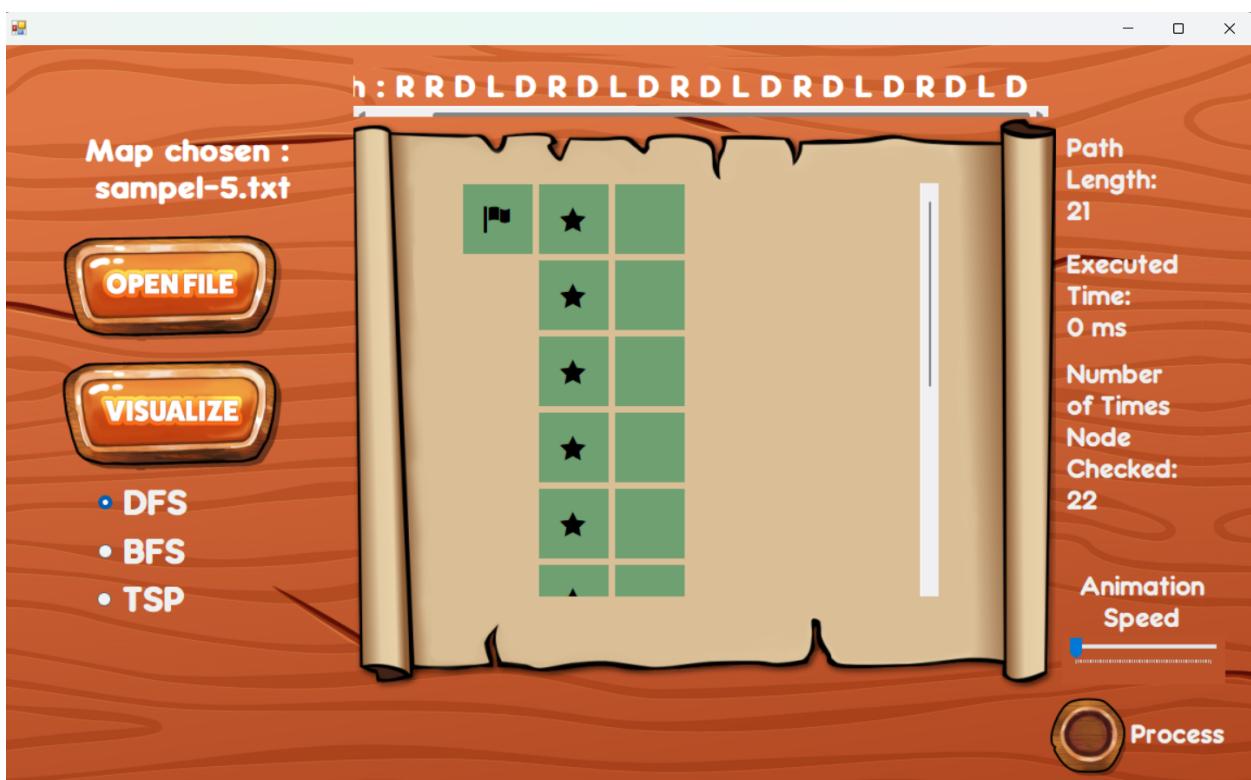


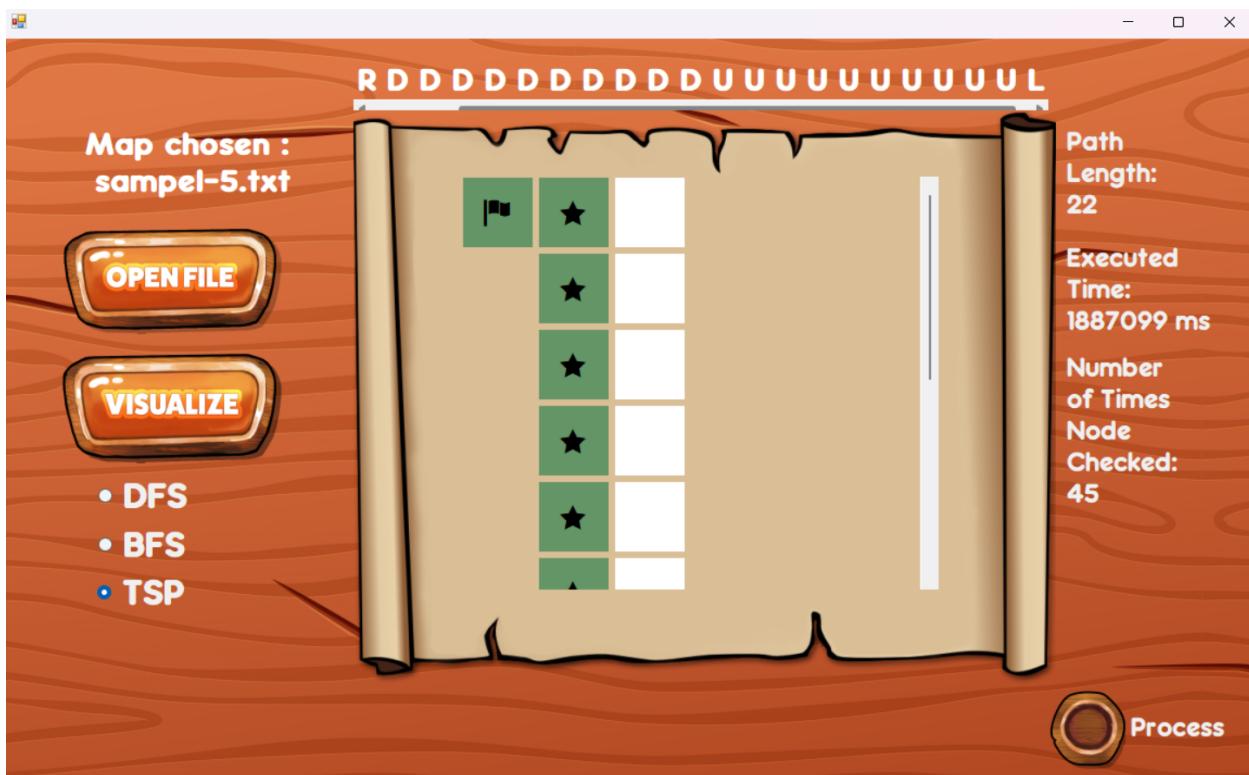
Gambar 4.4.4 Pengujian 4

Tabel 4.4.4 Hasil Pengujian 4

Nama file	sampel-4.txt
DFS	
Rute	D R D R D R D R D R
Banyak node	11
Banyak langkah	10
Waktu eksekusi	0 ms
BFS	
Rute	D R D R D R D R D R
Banyak node	21
Banyak langkah	10
Waktu eksekusi	0 ms
TSP	
Rute	D R D R D R D R D R L L L L U U U U
Banyak node	41
Banyak langkah	20
Waktu eksekusi	0 ms

4.4.5 Pengujian 5





Gambar 4.4.1 Pengujian 5

Tabel 4.4.5 Hasil Pengujian 5

4.5 Analisis Desain Solusi

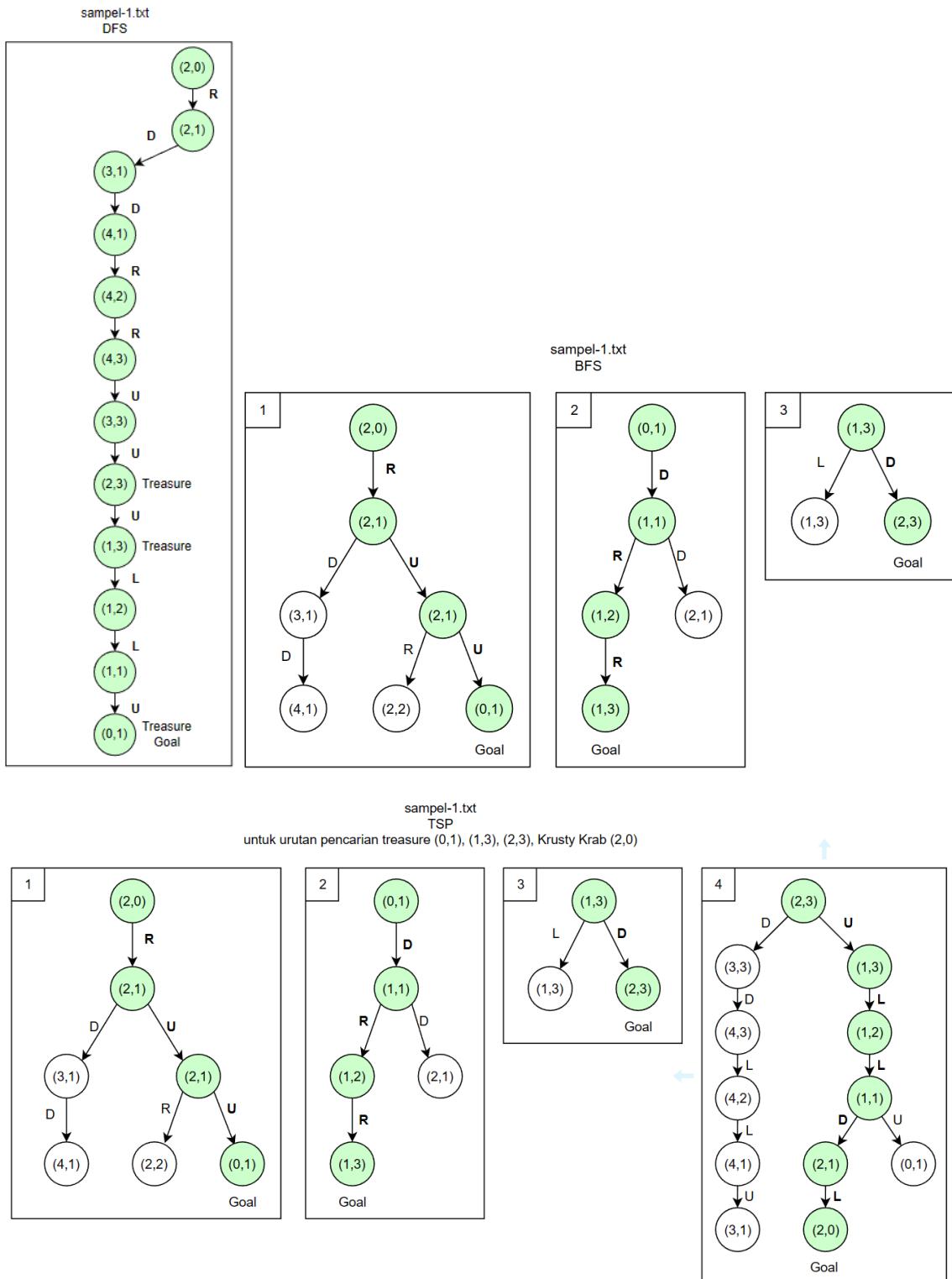
4.5.1 Analisis Pengujian 1

Dengan DFS, penelusuran dilakukan dengan urutan prioritas arah RLDU. Program akan melakukan penelusuran secara mendalam hingga semua *treasure* ditemukan. Penelusuran dimulai dari titik awal (2,0) hingga mencapai *treasure* pertama pada (2,3), lalu melanjutkan pencarian mendalamnya hingga mencapai *treasure* kedua pada (1,3), hingga terakhir mencapai *treasure* terakhir pada (0,1).

Dengan BFS, penelusuran dilakukan secara melebar hingga semua *treasure* ditemukan. Penelusuran dimulai dari titik awal (2,0) hingga mencapai *treasure* pertama pada (0,1). Kemudian, penelusuran dilakukan kembali dimulai dari titik (0,1) hingga mencapai *treasure* kedua pada (1,3). Penelusuran dilakukan kembali dimulai dari titik (1,3) hingga mencapai *treasure* terakhir pada (2,3).

Dengan TSP, program akan mencari seluruh solusi untuk semua kemungkinan (permutasi) urutan *treasure* yang diambil terlebih dahulu. Dari seluruh solusi tersebut, diambil solusi dengan rute terpendek. Pada kasus ini, urutan pengambilan *treasure* yang menciptakan rute terpendek adalah *treasure* pada (0,1), (1,3), (2,3), lalu kembali ke titik awal.

Pada kasus ini, rute dengan BFS (7 langkah) lebih pendek dibandingkan dengan rute dengan DFS (11 langkah). Akan tetapi, jumlah simpul yang dicek pada DFS (12 simpul) lebih sedikit dibandingkan BFS (13 simpul). Hal ini mungkin terjadi karena BFS mengunjungi lebih banyak simpul (pencarian melebar), sedangkan DFS mengunjungi simpul secara mendalam sehingga rutennya lebih panjang. Untuk kasus ini, rute yang dihasilkan TSP sama dengan rute pada BFS. Ini menandakan bahwa rute BFS merupakan rute yang paling efektif untuk kasus ini. Rute TSP lebih panjang (12 langkah) karena terdiri dari 7 langkah untuk mengambil semua *treasure* dan 5 langkah untuk kembali ke titik awal. Jumlah simpul yang dicek pada TSP cenderung banyak (24 simpul) karena TSP juga mengecek simpul-simpul untuk mendapatkan rute dari *treasure* terakhir ke titik awal sebanyak 11 simpul.



Gambar 4.5.1 Analisis Pengujian 1

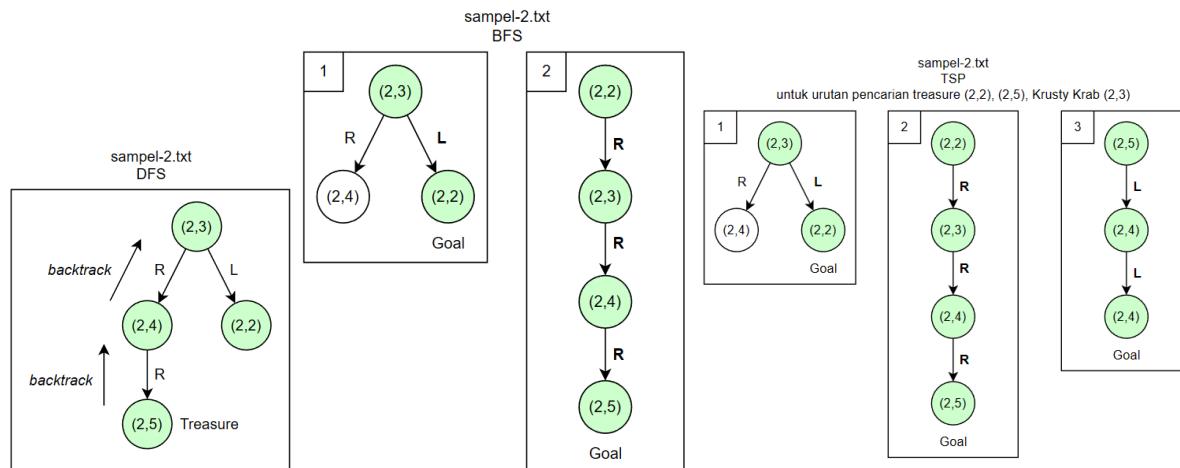
4.5.2 Analisis Pengujian 2

Dengan DFS, penelusuran dilakukan secara mendalam hingga semua *treasure* ditemukan. Penelusuran dimulai dari titik awal (2,3) hingga mencapai *treasure* pertama pada (2,5). Karena buntu, maka akan dilakukan *backtrack* hingga ke titik (2,3), lalu penelusuran dilanjutkan hingga mencapai *treasure* kedua pada (2,2).

Dengan BFS, penelusuran dilakukan secara melebar hingga semua *treasure* ditemukan. Penelusuran dimulai dari titik awal (2,3) hingga mencapai *treasure* pertama pada (2,2). Kemudian, penelusuran dilakukan kembali dimulai dari titik (2,2) hingga mencapai *treasure* kedua pada (2,5).

Dengan TSP, program akan mencari seluruh solusi untuk semua kemungkinan (permutasi) urutan *treasure* yang diambil terlebih dahulu. Dari seluruh solusi tersebut, diambil solusi dengan rute terpendek. Pada kasus ini, urutan pengambilan *treasure* yang menciptakan rute terpendek adalah *treasure* pada (2,2), (2,5), Krusty Krab (2,3).

Pada kasus ini, rute BFS (4 langkah) lebih pendek dibandingkan rute DFS (5 langkah). Akan tetapi, jumlah simpul yang dikunjungi DFS (4 simpul) lebih sedikit dibandingkan BFS (6 simpul). Hal ini terjadi karena DFS melakukan pencarian mendalam, sedangkan BFS pencarian melebar sehingga jumlah simpul yang dikunjungi lebih banyak. Rute pencarian *treasure* dengan TSP sama dengan rute yang dihasilkan dengan BFS. Ini menunjukkan bahwa pencarian BFS efektif untuk kasus ini. Jumlah langkah pada rute TSP adalah 4 langkah ditambah dengan rute kembali ke titik awal sebanyak 2 langkah, sedangkan jumlah simpul yang dikunjungi adalah 6 simpul ditambah dengan simpul yang dikunjungi untuk kembali ke titik awal sebanyak 2 simpul.



Gambar 4.5.2 Analisis Pengujian 2

4.5.3 Analisis Pengujian 3

Konfigurasi peta pada kasus ini tidak valid karena peta yang valid adalah peta yang terdiri dari satu buah huruf ‘K’ dan kombinasi antara huruf ‘R’, ‘T’, dan ‘X’. Peta yang memiliki komponen huruf selain huruf-huruf tersebut tidak valid. Karena itu, solusi tidak dapat ditemukan untuk peta yang tidak valid.

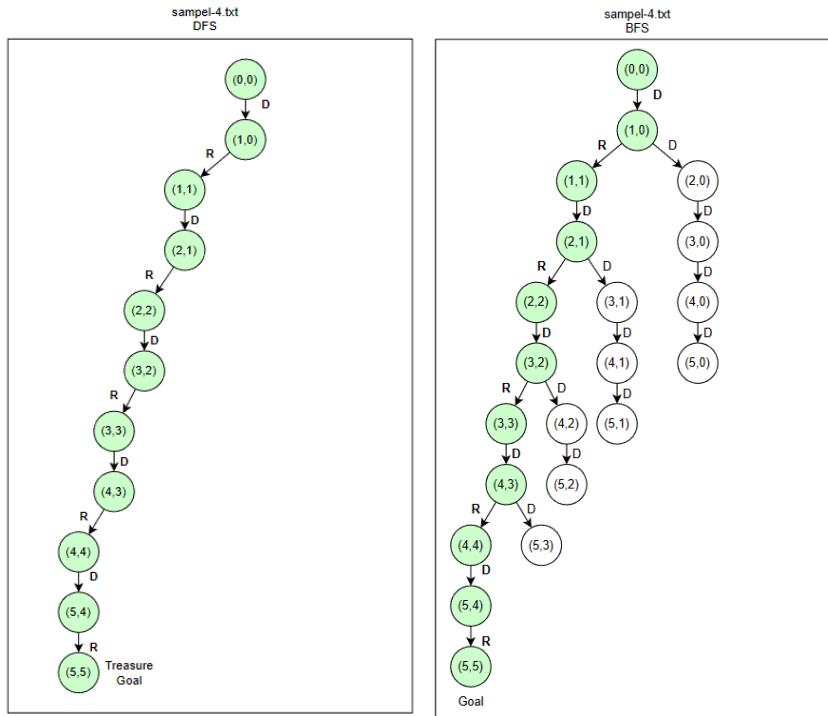
4.5.4 Analisis Pengujian 4

Dengan DFS, penelusuran dilakukan secara mendalam hingga semua *treasure* ditemukan. Penelusuran dimulai dari titik awal (0,0) hingga mencapai *treasure* pada (5,5). Penelusuran dilakukan secara mendalam hingga menemukan *treasure* atau hingga menemui jalan buntu.

Dengan BFS, penelusuran dilakukan secara melebar hingga semua *treasure* ditemukan. Penelusuran dimulai dari titik awal (0,0) hingga mencapai *treasure* pada (5,5). Penelusuran dilakukan secara melebar dengan mengunjungi semua simpul di sekitar simpul yang sedang dikunjungi.

Karena hanya terdapat satu buah *treasure* pada peta, maka algoritma TSP sama dengan algoritma BFS, hanya saja rute penelusuran ditambahkan dengan rute dari *treasure* (5,5) ke titik awal (0,0).

Untuk kasus ini, rute dengan DFS dan BFS memiliki rute dan panjang yang sama. Akan tetapi, jumlah simpul yang dikunjungi BFS (21 simpul) lebih banyak dibandingkan DFS (11 simpul). Hal ini terjadi karena DFS hanya melakukan pencarian mendalam hingga *treasure* ditemukan, sedangkan BFS mengunjungi semua simpul tetangga dari simpul yang sedang dikunjungi. Dengan TSP, dapat dibuktikan bahwa rute yang diperoleh dengan DFS dan BFS keduanya efektif. Panjang rute TSP adalah 10 langkah untuk mencapai *treasure* dan 10 langkah untuk kembali ke titik awal. Jumlah simpul yang dikunjungi oleh TSP adalah 21 simpul untuk mencapai *treasure* dan 20 simpul untuk mencapai titik awal kembali.



Gambar 4.5.4 Analisis Pengujian 4

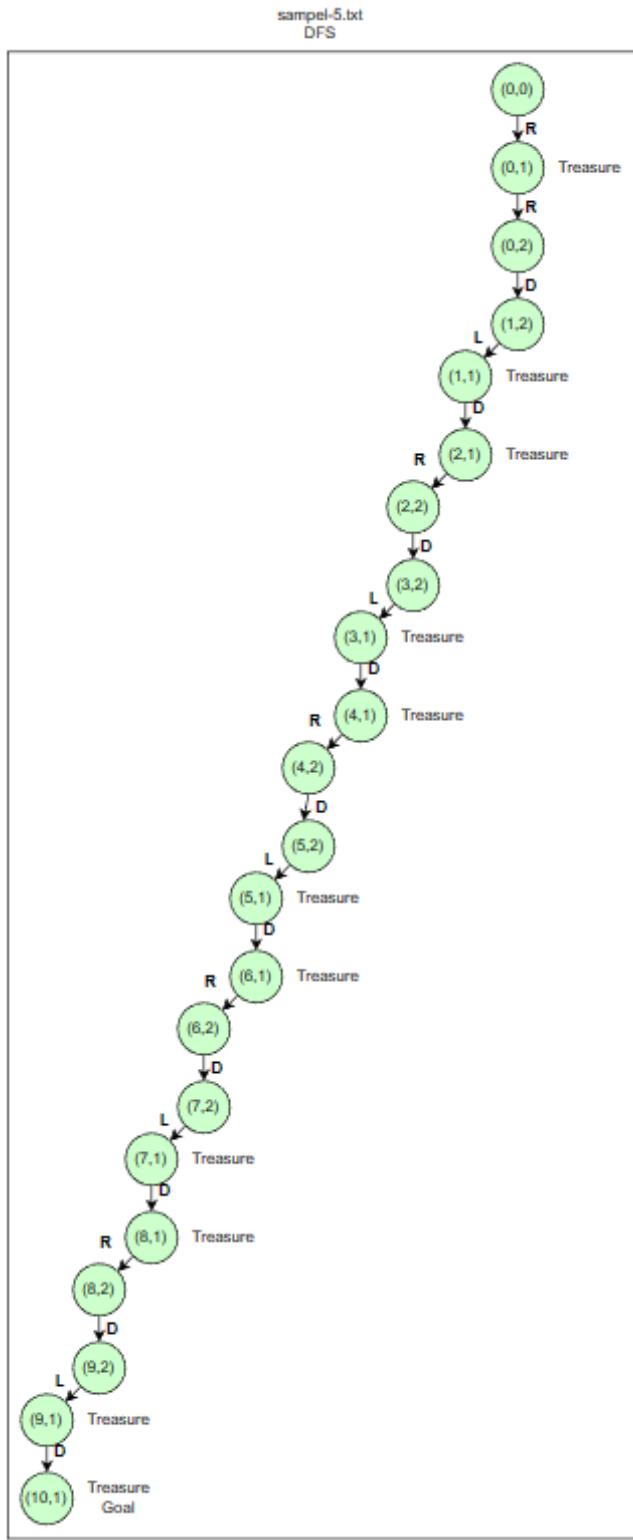
4.5.5 Analisis Pengujian 5

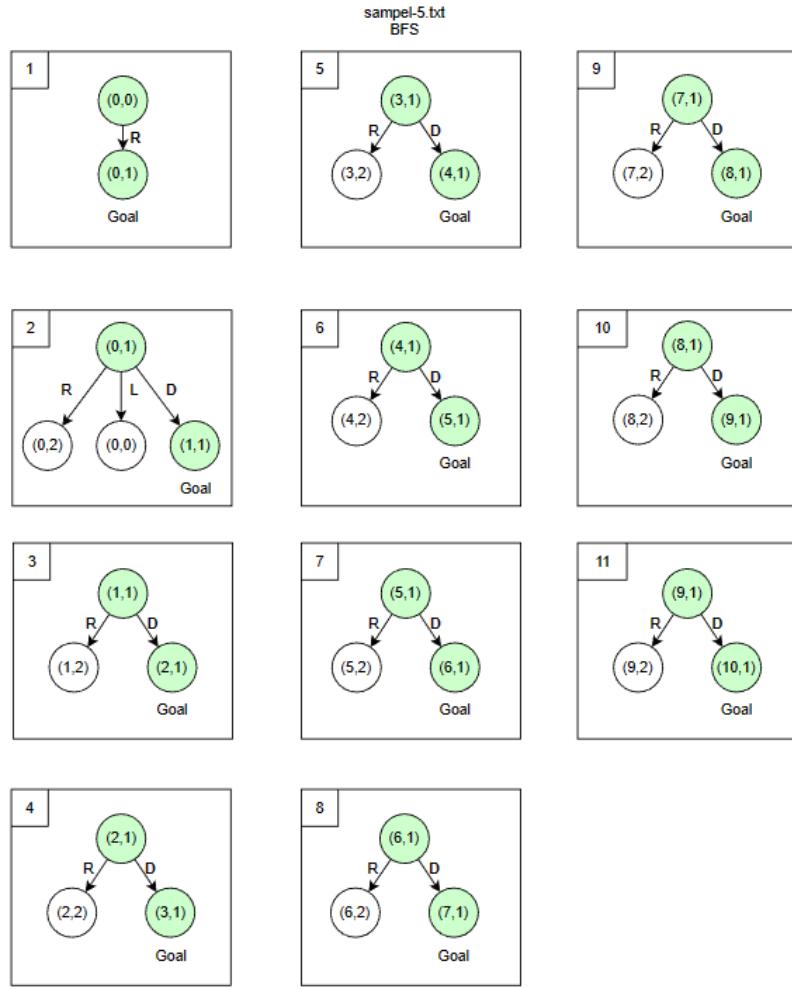
Dengan DFS, penelusuran dilakukan secara mendalam hingga semua *treasure* ditemukan. Penelusuran dimulai dari titik awal (0,0) hingga mencapai *treasure* pertama pada (0,1), lalu melanjutkan pencarian mendalamnya hingga mencapai *treasure* kedua pada (1,1). Penelusuran terus dilakukan secara mendalam hingga mencapai *treasure* terakhir pada (10,1).

Dengan BFS, penelusuran dilakukan secara melebar hingga semua *treasure* ditemukan. Penelusuran dimulai dari titik awal (0,0) hingga mencapai *treasure* pertama pada (0,1). Kemudian, penelusuran dilakukan kembali dimulai dari titik (0,1) hingga mencapai *treasure* kedua pada (1,1). Penelusuran melebar kembali dilakukan hingga semua *treasure* ditemukan.

Untuk kasus ini, rute oleh BFS (11 langkah) lebih pendek dibandingkan rute oleh DFS (21 langkah). Akan tetapi, jumlah simpul yang dikunjungi oleh DFS (22 simpul) lebih sedikit dibandingkan BFS (23 simpul). Hal ini terjadi karena DFS menelusuri secara mendalam dengan urutan arah RLDU, berbeda dengan BFS yang mencari secara melebar sehingga dapat mengambil semua *treasure* dengan rute yang lebih pendek. Untuk TSP, solusi yang harus dicari

adalah sebanyak $11!$ rute (ada $11!$ kemungkinan urutan *treasure*), sehingga kami tidak melakukan pengecekan untuk algoritma TSP pada kasus ini.





Gambar 4.5.5 Analisis Pengujian 5

BAB V

PENUTUP

5.1 Simpulan

Program *treasure hunt* merupakan program yang dapat melakukan penelusuran terhadap peta untuk mengambil seluruh *treasure* yang terdapat pada peta. Dalam menentukan rute tersebut, program dapat memanfaatkan algoritma Depth First Search (DFS), Breadth First Search (BFS), atau Traveling Salesman Problem (TSP). Setiap algoritma memiliki cara yang berbeda dalam melakukan penelusuran. DFS melakukan penelusuran peta secara mendalam, BFS melakukan penelusuran secara melebar, sedangkan TSP melakukan penelusuran peta untuk menentukan rute terpendek. Dengan demikian, program *treasure hunt* dirancang.

5.2 Saran

Penyelesaian TSP dalam *treasure hunt* ini dapat dioptimalisasi lagi dengan menggunakan algoritma-algoritma yang lebih efisien. Silakan gunakan program ini untuk berburu harta karun dan menjadi kaya.

DAFTAR REFERENSI

Microsoft Learn. .NET Documentation.

<https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic?view=net-8.0>

Munir, Rinaldi. 2023. Breadth First Search (BFS) dan Depth First Search (DFS) (Bagian 1).

Homepage Rinaldi Munir Sekolah Teknik Elektro dan Informatika (STEI) ITB.

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

Munir, Rinaldi. 2023. Breadth First Search (BFS) dan Depth First Search (DFS) (Bagian 1).

Homepage Rinaldi Munir Sekolah Teknik Elektro dan Informatika (STEI) ITB.

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

Munir, Rinaldi. 2022. Graf (Bagian 3). Homepage Rinaldi Munir Sekolah Teknik Elektro dan Informatika (STEI) ITB.

<https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian3.pdf>

net-informations. C# DataGridView Tutorial.

<http://csharp.net-informations.com/datagridview/csharp-datagridview-tutorial.htm>

LAMPIRAN

Tautan repository tugas besar: https://github.com/arleenchr/Tubes2_kelp-juice

Tautan video: <https://youtu.be/y0aytDtdDGs>