IF2211 Strategi Algoritma

IMPLEMENTASI ALGORITMA DEPTH-FIRST SEARCH DAN BREADTH-FIRST SEARCH DALAM PENCARIAN HARTA KARUN

Laporan Tugas Besar II

Disusun untuk memenuhi tugas mata kuliah Strategi Algoritma

pada Semester II (dua) Tahun Akademik 2022/2023

Oleh

Kenneth Ezekiel Suprantoni 13521089

Chiquita Ahsanunnisa 13521129

Vanessa Rebecca Wiyono 13521151

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

BANDUNG

2023

DAFTAR ISI

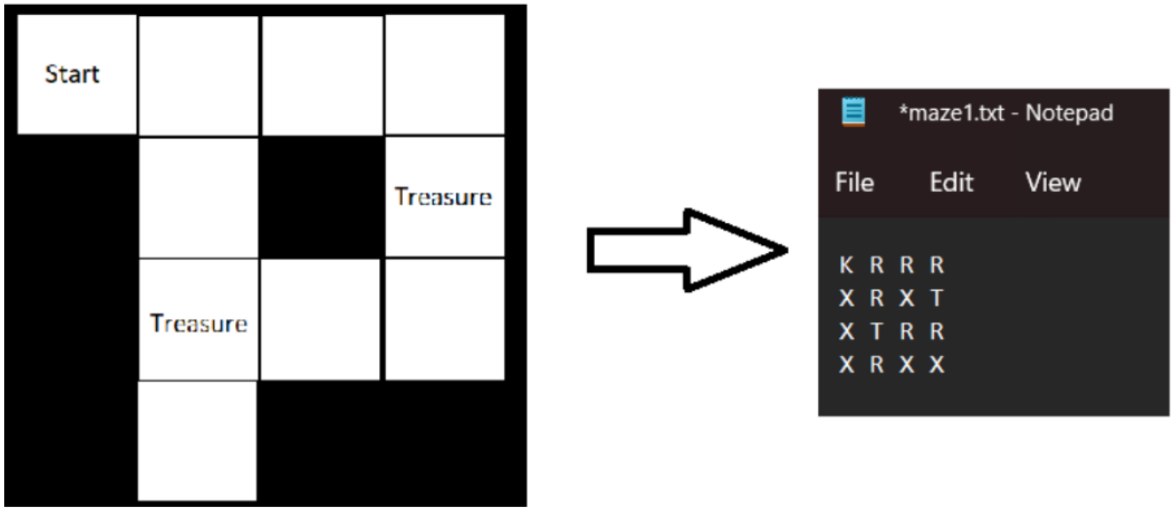
BAB I
DESKRIPSI MASALAH

Pada tugas ini, dibutuhkan sebuah algoritma pencarian untuk pencarian harta karun, yang diimplementasikan ke dalam sebuah aplikasi berbasis *desktop* dengan GUI dengan menggunakan algoritma BFS dan DFS untuk mendapatkan rute yang memperoleh seluruh harta karun yang ada. Program dapat menerima dan membaca sebuah *input file* dengan ektensi .txt yang berisi *maze* yang akan dicari solusi rute untuk harta karun-nya. Rute solusi adalah rute yang mengambil semua harta karun, dan rute hasil dari DFS dan BFS dapat berbeda. Akan dijelaskan untuk bonus visualisasi pencarian, bagaimana kelompok kami membedakan mana blok yang sedang dicari dan yang sudah dicari.

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

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

Contoh file input :



Gambar 2. Ilustrasi input file maze

Dengan memanfaatkan algoritma Breadth First Search (BFS) dan Depth First Search (DFS), anda dapat menelusuri grid (simpul) yang mungkin dikunjungi hingga ditemukan rute solusi, baik secara melebar ataupun mendalam bergantung alternatif algoritma yang dipilih. Rute solusi adalah rute yang memperoleh seluruh treasure pada maze. Perhatikan bahwa rute yang diperoleh dengan algoritma BFS dan DFS dapat berbeda, dan banyak langkah yang dibutuhkan pun menjadi berbeda. Prioritas arah simpul yang dibangkitkan dibebaskan asalkan ditulis di laporan ataupun readme, semisal LRUD (left right up down). Tidak ada pergerakan secara

diagonal. Anda juga diminta untuk memvisualisasikan input txt tersebut menjadi suatu grid maze serta hasil pencarian rute solusinya. Cara visualisasi grid dibebaskan, sebagai contoh dalam bentuk matriks yang ditampilkan dalam GUI dengan keterangan berupa teks atau warna. Pemilihan warna dan maknanya dibebaskan ke masing-masing kelompok, asalkan dijelaskan di readme / laporan.

BAB II
TEORI SINGKAT

2.1 Traversal Graf

Traversal graf adalah sebuah proses untuk mengunjungi semua simpul di dalam sebuah graf, dan urutan pengunjungan akan mengklasifikasikan tipe traversal graf. Sebuah spesialisasi dari traversal graf adalah traversal pohon, karena struktur data pohon merupakan sebuah spesialisasi dari struktur data graf, yang tidak memiliki sirkuit. Dalam traversal graf, terkadang terdapat kondisi dimana sebuah simpul dikunjungi lebih dari sekali, dan semakin banyak simpul yang ada di dalam graf, akan semakin banyak pula redundansi yang akan terlihat. Dalam pencarian rute, terdapat beberapa cara yang dapat digunakan agar graf yang sudah dilewati oleh rute yang sama tidak dikunjungi kembali, salah satu caranya yang kelompok kami terapkan adalah dengan *string matching* dari arah rute. Dalam traversal graf, graf dapat merepresentasikan persoalan yang ingin dipecahkan, dan traversal graf adalah proses pencarian solusinya. Aplikasi dari traversal dapat bermacam-macam, beberapa contohnya adalah Algoritma Cheney, mendapatkan rute terpendek antara dua simpul, dan algoritma penomoran Cuthill-McKee.

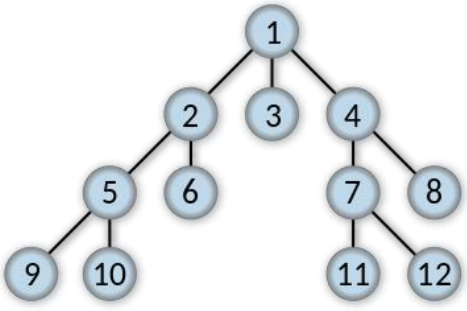
2.2 Algoritma *Breadth-First Search*

Algoritma *breadth-first search* (BFS) adalah salah satu algoritma traversal pencarian pada graf. Algoritma ini mengunjungi simpul-simpul pada graf secara "melebar".

Misalkan ada sebuah simpul *v* dari graf *G*. Akan dilakukan traversal graf *G* dengan algoritma BFS dimulai dari simpul *v*. Skema algoritmanya adalah sebagai berikut.

1. Kunjungi simpul *v*.
2. Kunjungi seluruh simpul yang bertetangga dengan simpul *v*.
3. Kunjungi simpul yang belum dikunjungi dan bertetangga dengan simpul-simpul yang sudah dikunjungi sebelumnya.
4. Lakukan langkah yang sama hingga seluruh simpul pada graf *G* telah dikunjungi.

Sebagai contoh nyata, pada graf yang ditunjukkan pada Gambar 2.2.1, urutan simpul yang dikunjungi pada algoritma BFS adalah 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12.



Gambar 2.2.1 Urutan Penjelajahan Algoritma BFS

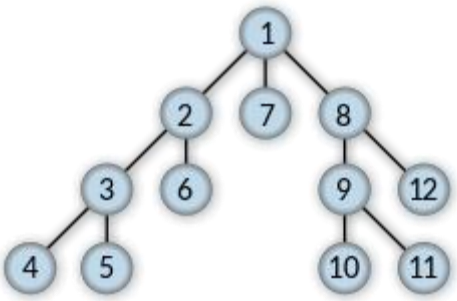
Algoritma BFS pada umumnya diimplementasikan dengan struktur data antrian/*queue*. Struktur data antrian ini digunakan untuk menyimpan simpul yang telah dikunjungi. Untuk lebih jelasnya, perhatikan *pseudocode* dari algoritma BFS di bawah ini.

```
procedure BFS(input v: simpul)
```

```
{ Melakukan traversal graf dengan algoritma pencarian BFS.
  Masukan: v adalah simpul awal traversal
  Luaran: Seluruh simpul graf dijelajahi dengan algoritma pencarian BFS }

Deklarasi
    w : simpul
    q : antrian

    procedure Kunjungi(input/output w : simpul)
    { Mengunjungi simpul w }
    function SudahDikunjungi(input w : w) → boolean
    { Mengecek apakah simpul w sudah dikunjungi }
    procedure BuatAntrian(input/output q : antrian)
    { Membuat antrian kosong }
    procedure MasukAntrian(input/output q : antrian, input v : simpul)
    { Memasukkan simpul v ke antrian q pada posisi belakang }
    procedure HapusAntrian(input/output q : antrian, output v : simpul)
    { Menghapus v dari kepala antrian q }
    function AntrianKosong(input q : antrian) → boolean
    { Mengecek apakah antrian q kosong }

Algoritma
    BuatAntrian(q)
    Kunjungi(v)
    MasukAntrian(q,v)

    { Kunjungi semua simpul graf selama antrian belum kosong }
    while not AntrianKosong(q) do
        HapusAntrian(q,v)
        for (tiap simpul w yang bertetangga dengan simpul v) do
            if not SudahDikunjungi(w) then
                Kunjungi(w)
                MasukAntrian(q,w)
            endif
        endfor
    endwhile
    { AntrianKosong(q) }
```

2.3 Algoritma *Depth-First Search*

Algoritma DFS adalah sebuah algoritma pencarian traversal yang umumnya digunakan untuk struktur data pohon atau graf. Pencarian dimulai di simpul akar dan mengeklorasi semua kemungkinan solusi dari semua *branch* sampai akhirnya mengalami *deadend*, dimana simpul tersebut merupakan simpul daun. Dalam DFS, biasanya digunakan sebuah struktur data *stack* untuk *keeping track* simpul mana yang akan dikunjungi berikutnya. Hal ini bisa terjadi karena *stack* menggunakan prinsip LIFO, yang sesuai dengan cara kerja DFS. Contohnya, jika sebuah simpul memiliki dua anak, misalnya anak kiri dan anak atas, maka anak akan dimasukkan ke dalam *stack* sesuai prioritasnya (dari yang terakhir, misal prioritas LURD), maka simpul anak kiri akan di-*push* terlebih dahulu ke *stack*, diikuti *push* anak atas ke *stack*, lalu dilakukan *pop* untuk mendapatkan simpul berikutnya yang akan dijelajahi, yaitu simpul anak atas, yang akan mem-*push* anak-anak-nya dan anak-anak tersebut akan ditelusuri terlebih dahulu sampai semuanya habis, baru akan menelusuri simpul anak kiri yang pertama kali di-*push*.



Gambar 2.2.2 Urutan Penjelajahan Algoritma DFS

Berikut adalah skema umum dari algoritma DFS.

```
Procedure DFS(G, v)
{ Melakukan pencarian dengan algoritma DFS
Masukan: graf atau pohon G dan simpul v
Luaran: Menjelajahi seluruh graf dengan skema DFS }

Algoritma
     tandai v sudah dijelajahi
     for semua sisi dari v ke w yang bersisian dengan simpul v:
          if simpul w belum dijelajahi then
              DFS(G, w)
          endif
     endfor
```

Pada algoritma diatas, tidak ditunjukkan struktur data *stack*-nya karena pemanggilan secara rekursif akan secara automatis menggunakan *stack* dikarenakan *call stack* yang digunakan untuk pemanggilan rekursif menggunakan struktur data *stack.*

2.4 C# *Desktop Application Development*

C# *desktop application development* merupakan sebuah alat untuk mengembangkan *dekstop application* yang dapat beroperasi tanpa koneksi internet. Salah satu bahasa pemrograman yang sering digunakan untuk pengembangan ini adalah C#. Dalam hal *user interface* (UI), pengembangan dilakukan melalui WinForms/WPF dengan bantuan *integrated development environment* (IDE) Visual Studio sehingga praktiknya lebih mudah. WPF digunakan dalam mengembangkan aplikasi yang berbasis Windows dengan menggunakan bahasa *native* yaitu bahasa C#. Berikut adalah langkah-langkah pengembangan aplikasi dekstop dengan menggunakan Windows Form App:

Membuat project baru:

Buka aplikasi visual studio 2022 lalu pilih create a new project. Setelah itu, pilih*Windows Forms App (.NET* Framework*).* Isi nama project, kemudian buat project dengan menekan tombol ‘create’

Membuat aplikasi:

Pilih menu *Toolbox* untuk membuka jendela Toolbox-fly-out lalu pilih komponen yang ingin digunakan untuk aplikasi dekstop. Modifikasi komponen dapat dilakukan dengan mengubah properti pada jendela *properties.*

Menambahkan kode pada form

Pilih jendela Form1.cs [Design] lalu double-click untuk membuka jendela Form1.cs. Konfigurasi kode dapat ditulis pada file Form1.cs

Menjalankan aplikasi

Aplikasi dapat dijalankan dengan menekan tombol *Start* pada *menu*.

BAB III
APLIKASI ALGORITMA BFS DAN DFS

3.1 Langkah pemecahan masalah

Permasalahan yang akan dipecahkan adalah pencarian sebuah rute yang mendapatkan semua rute. Permasalahan tersebut dapat dipecah menjadi beberapa bagian, yaitu:

- Kelas Matrix yang menyimpan data posisi dari peta yang diberikan, yang memiliki atribut *two-dimensional array of* Block, dan sebuah Player.
- Kelas Block yang akan merepresentasikan sebuah simpul dalam graf, yang menjadi atribut dari Matrix
- Kelas Player yang akan merepresentasikan *active node* atau simpul yang sedang dijelajahi
- Kelas DFS yang akan memiliki atribut Struktur data Stack dan sebuah Dictionary untuk *mapping* id simpul dan rute perjalanan
- Kelas BFS yang memiliki atribut objek Matrix yang memiliki metode untuk mencari rute pencarian Treasure pada Matrix tersebut
- Kelas Basic yang menandakan Block yang Basic, bisa di Step
- Kelas Tembok yang menandakan Block tembok yang tidak bisa dilewati
- Kelas Start yang merupakan Block Basic dimana Player mulai, sehingga menjadi spesialisasi dari kelas Basic
- Kelas Treasure yang merupakan Block Basic dimana harta karun dapat ditemukan, sehingga menjadi spesialisasi dari kelas Basic

Permasalahan tersebut diselesaikan dengan prosedur berikut:

1. Masukan *file* dengan ekstensi .txt pada aplikasi akan dipakai dalam pembuatan kelas Matrix, dimana atribut map nya akan menggambarkan peta yang diberikan oleh *file* tersebut.
2. Semua Block yang disimpan didalam map akan disambungkan dengan tetangganya.
3. Dalam proses *mapping*, Block yang berbeda akan disesuaikan dengan tipenya, dan setiap Block Treasure yang dibaca, akan ditambahkan untuk hitungan TreasureCount.
4. Jalankan Algoritma pencarian BFS atau DFS sesuai pilihan sampai: semua harta karun sudah ditemukan (TreasureCount == TreasureTaken), atau pencarian selesai dan belum semua harta karun ditemukan (Tidak terdapat rute).
5. Algoritma akan mengeluarkan sebuah String moves yang menggambarkan Langkah-langkah yang akan diambil oleh player untuk mengambil seluruh harta karun, yang menjadi rute solusi.
6. String moves akan divisualisasikan oleh program menjadi animasi atau gambar dari rute hasil.

3.2 Proses *Mapping* Persoalan Menjadi Elemen dalam Algoritma DFS dan BFS

Setiap karakter (kecuali spasi) pada *file* .txt mewakili satu blok. Blok tersebut dapat dibagi menjadi dua jenis, yaitu blok yang dapat dilintasi dan yang tidak dapat dilintasi. Blok yang dapat diinjak yaitu blok yang diwakili dengan karakter ‘R’ (lintasan), ‘K’ (*start*), dan ‘T’ (*treasure*). Blok yang tidak dapat diinjak yaitu blok yang diwakili dengan karakter ‘X’ (halangan).

Berdasarkan pemaparan di atas, *mapping* dengan membuat kelas abstrak Block dan membuat turunannya yaitu kelas Basic dan kelas Tembok. Kelas Basic mewakili blok yang tidak dapat dilintasi, sedangkan kelas Tembok mewakili blok halangan. Kelas Basic dapat diturunkan lagi menjadi kelas Start yang mewakili blok *start* dan kelas Treasure yang mewakili blok *treasure*.

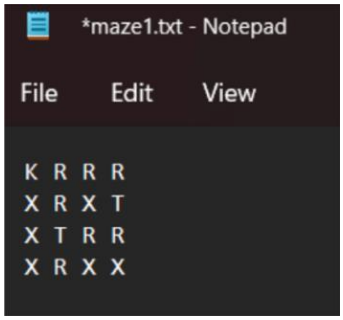
Berikut adalah *mapping* karakter pada *file* .txt dengan kelas turunan Block yang bersesuaian.

Tabel 3.2.1 *Mapping* Karakter dan Block

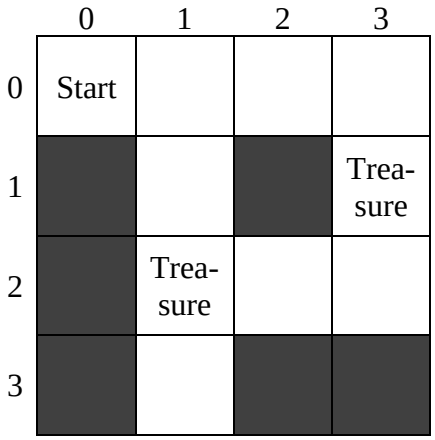
Karakter	***Base Class***	***Derived Class***
K	Basic	Start
T	Basic	Treasure
R	Basic	-
X	Tembok	-

Konfigurasi blok-blok tersebut akan disimpan pada sebuah matriks yang ditangani oleh kelas Matrix. Kelas Matrix adalah kelas yang menyimpan matriks yang berisi Block (beserta kelas turunannya, sesuai dengan tipe blok).

Secara umum, *mapping* dari matriks menjadi graf dapat dilakukan dengan melihat bahwa setiap elemen matriks adalah simpul dari graf, dengan catatan, yang dianggap simpul hanyalah blok yang dapat diinjak. Simpul-simpul yang bertetangga adalah simpul yang bersebelahan (dari sisi kiri, kanan, atas, atau bawah). Sebagai contoh, pada matriks pada Gambar 3.2.1, simpul blok indeks ke (0,1) bertetangga dengan simpul blok indeks ke (0,0), (1,1), dan (0,2).



File konfigurasi



Matriks

Gambar 3.2.1 Contoh *Mapping*

Setelah di-*mapping* menjadi graf dengan simpul dan sisi yang jelas, algoritma BFS dan DFS dapat diimplementasikan untuk menjelajahi matriks tersebut.

3.3 Ilustrasi Kasus Lain

Terdapat algoritma lain yang dapat digunakan dalam pencarian rute yang mendapatkan semua rute. Contoh dari algoritma lain tersebut adalah algoritma *iterative deepening search* atau IDS, dimana algoritma tersebut menggabungkan efisiensi ruang dari algoritma DFS dan penemuan cepat untuk simpul yang dekat dengan akar dari BFS. Dalam penerapannya, algoritma IDS memanfaatkan alforitma *depth limited search* (DLS) yang melakukan pembatasan kedalaman pencarian. Berikut merupakan contoh ilustrasi dari algoritma IDS.

Gambar 3.3.1 Ilustrasi Algoritma IDS

Berdasarkan ilustrasi tersebut, dapat dilihat bahwa IDS melakukan pencarian secara iteratif menggunakan penelusuran DLS dimulai dengan batasan level 0. Jika solusi belum ditemukan, maka akan dilakukan iterasi ke-2 dengan batasan level 1. Hal tersebut akan terus diulangi hingga solusi ditemukan. Penelusuran setiap proses pencariannya menggunakan prinsip DFS, yaitu dengan menelusuri satu node dalam setiap level dari yang paling kiri. Jika solusi masih belum ditemukan hingga pada level terdalam, maka pencarian akan dilanjutkan pada node sebelah kanan. Demikian seterusnya hingga solusi ditemukan.

Jika pemecahan masalah seperti pada poin 3.1 dilakukan dengan menggunakan IDS, maka berikut adalah langkah-langkahnya.

1. Buat kelas IDS dengan atribut yang sama seperti pada metode DFS, yaitu struktur data stack dan sebuah dictionary yang berguna untuk mapping id simpul sdan rute perjalanan
2. Buat metode search() pada kelas IDS yang akan memanggil metode search DFS dengan kedalaman level 0 hingga batas level tertentu. Batas kedalaman akan ditingkatkan pada setiap iterasi hingga akhirnya solusi ditemukan atau batas kedalaman maksimum tercapai.
3. Buat metode searchDFS(depth) pada kelas IDS yang berguna untuk melakukan pencarian DFS pada kedalaman yang kurang dari atau sama dengan batas kedalaman yang ditentukan.
4. Ketika metode search() pada kelas IDS dijalankan, cari solusi dengan cara menentukan batas kedalaman awal dan kemudian batas kedalaman ditingkatkan hingga akhirnya solusi ditemukan atau hingga batas kedalaman maksimum tercapai.

BAB IV

ANALISIS PEMECAHAN MASALAH

4.1 Implementasi Program

Implementasi dari program pencarian terdapat pada beberapa kelas, yaitu kelas Matrix, kelas DFS, dan kelas BFS. Pada kelas Matrix, diimplementasikan sistem pergerakan seperti penyambungan tetangga dari suatu simpul Block dan visualisasi. Pada kelas DFS dan BFS, diimplementasikan algoritma pencarian secara *depth-first* dan *breadth-first*, beserta fungsi-fungsi pembantunya.

4.1.1 Implementasi Algoritma DFS

Jalan algoritma pada metode pencarian DFS adalah sebagai berikut:

```
Procedure StartSearch(Block node, bool TSP) -> string
// Memulai pencarian secara depth-first
// Masukan: node – simpul yang sedang dikunjungi, TSP – indikasi apakah
TSP sedang dinyalakan atau tidak
// Keluaran: string – Gerakan yang perlu dilakukan untuk rute solusi }
Deklarasi
String moves
Char lastMove

Algoritma
lastMove = 'S'
moves = Search(node, reference lastMove, moves, TSP)

if not (Jumlah Treasure yang diambil == Jumlah Treasure) then
    // Tidak terdapat rute yang mendapatkan semua treasure
    return ""
return moves
```

```
Procedure Search(Block node, reference char lastMove, string moves, bool
TSP) -> string
// Melakukan pencarian secara depth-first
// Masukan: node – simpul yang sedang dikunjungi, lastMove – gerakan
terakhir yang dilakukan untuk          mencapai simpul yang dikunjungi,
moves – gerakan sejauh ini, TSP – indikasi apakah TSP sedang dinyalakan
atau tidak
// Keluaran: string – Gerakan yang perlu dilakukan untuk rute solusi }
Deklarasi
String currentMoves
Boolean notDeadend
Tuple (char, block) Child
Char nextMove
Block nextBlock
Int i

Algoritma
```

```
currentMoves = moves
if (jumlah treasure diambil == jumlah treasure) then
     // Pencarian sudah selesai
     moves = currentMoves
     return moves
else
    // Lanjutkan pencarian ke node yang sekarang dikunjungi
    node.step() // Injak simpul yang dikunjungi
    currentMoves += lastMove // Tambahkan Langkah terakhir ke pergerakan
    Masukkan mapping ID Block node dan currentMoves ke dictionary
attribute kelas DFS
    notDeadend = false
    insertChild(node, reference notDeadend) // Akan mengganti notDeadEnd
menjadi true jika masih ada jalan dari node dan push semua child ke stack
    if (notDeadend) then
             for i traversal 0..node.getNumOfChild()
                 Child = getChild() // Pop child dari stack
                 nextMove = Child.move
                 nextBlock = Child.node
                if not (nextBlock sudah pernah dilintasi dengan rute yang
sama, yaitu di dalam dictionary terdapat sebuah key dengan ID yang sama
dengan nextBlock dan string currentMoves diawali dengan value dari ID
block tersebut dalam dictionary) then
                    Search(nextBlock, reference nextMove, currentMoves,
TSP) // Lakukan search rekursif
                 endif
             endfor
             if (TSP) or (Belum mendapatkan semua treasure) then
                 reverseMove // Putar balik arah pergerakan, dilakukan agar
rute tidak patah dalam proses backtracking
             endif
         return currentMoves
    else // Deadend, maka backtrack
         reverseMove dan reverseLastMove
         return currentMoves
    endif
endif
```

Beberapa hal yang dapat di-*highlight* dalam algoritma DFS adalah sebagai berikut:

- Penyambungan rute saat *backtrack* sudah ter-*handle* sehingga akan tetap menghasilkan rute yang tidak terputus
- Dapat membedakan simpul berdasarkan rutenya menggunakan Dictionary, sehingga dalam rute yang sama simpul yang sama tidak akan ditelurusi kembali, tetapi jika terdapat perubahan rute (karena backtracking), maka simpul tersebut dapat ditelusuri kembali menggunakan *string matching*
- Penggunaan konsep Kelas dan Objek, sehingga setiap Block dapat secara automatis dibedakan tipenya dan semua *method* yang dipanggil akan memanggil method yang dimiliki setiap tipe, misal tipe Block treasure jika dipanggil *method* step nya, maka akan secara automatis bertambah hitungan berapa kali Block tersebut telah diinjak dan berapa jumlah harta karun yang sudah terambil

- Dalam Push tetangga dari sebuah simpul ke dalam stack, disertakan juga langkah yang akan diperlukan, sehingga tetangga atas akan tercata langkah yang diperlukannya sebagai ‘U’, kiri ‘L’, kanan ‘R’, dan bawah ‘D’
- Pencatatan rute yang dilakukan secara *string concatenation*, sehingga visualisasi dan pencarian rute dapat dibedakan secara modular

4.1.2 Implementasi Algoritma BFS

Pada implementasi pencarian Treasure dengan algoritma BFS, permasalahan dipecah menjadi beberapa langkah sebagai berikut.

1. Cari jalan dari Start hingga mendapat Treasure yang belum diambil secara BFS. Simpan lokasi Treasure tersebut. Simpan rutenya.
2. Gunakan lokasi Treasure yang disimpan sebelumnya (Treasure yang terakhir diambil) sebagai lokasi *start* baru. Cari jalan dari lokasi *start* yang baru hingga mendapat Treasure yang belum diambil secara BFS. Simpan rutenya
3. Ulangi langkah 2 hingga seluruh Treasure terambil (untuk kasus seluruh Treasure *reachable*/graf terhubung) atau seluruh simpul telah dicek (untuk kasus tidak semua Treasure *reachable*/graf tidak terhubung). Simpan lokasi Treasure yang terakhir diambil. Simpan seluruh rutenya juga.
4. Jika diinginkan untuk mengaplikasikan TSP (kembali ke blok Start), gunakan lokasi Treasure yang disimpan sebelumnya (Treasure yang terakhir diambil) sebagai lokasi *start* baru. Cari jalan dari lokasi *start* yang baru hingga mencapai blok Start secara BFS. Simpan rutenya.
5. Gabungkan seluruh rute yang ada.

Pada implementasinya, keseluruhan pencarian dilakukan oleh fungsi `Search`, pencarian rute ke satu Treasure dengan algoritma BFS dilakukan oleh fungsi `SubBFS`, pencarian rute ke blok Start untuk TSP dilakukan oleh fungsi `SubBFSTSP`.

Pseudocode dari algoritma yang sudah dipaparkan di atas adalah sebagai berikut.

```
function Search(boolean tsp, reference string search) → string

Deklarasi
    moves : string
    took  : boolean
    next  : Position

Algoritma
    took ← false
    { Mulai mencari dari blok Start }
    next ← SubBFS(this.map.GetStartPos(), reference moves, reference
search, reference took)

    { Mencari hingga seluruh Treasure telah diambil atau seluruh simpul
telah dicek (took bernilai false) }
    while (took and not(Treasure.isAllTaken())) do
        map.resetMatrixStep()
        { Meng-assign start yang baru }
```

```
        next ← SubBFS(next, reference moves, reference search, reference
took)

    { Toggle untuk TSP }
    if (tsp) then
        map.resetMatrixStep()
        SubBFSTSP(next, reference moves, reference search)

    → moves
```

```
function SubBFS(Position startNode, reference string result, reference
string search, reference bool took) → Position

Deklarasi
    posQueue                              : Queue of List of Position
    posList, prev, searchPos, path, next : List of  Position
    nextPos, currNode                     : Position

Algoritma
    { Mencari dari startNode }
    posList.Add(startNode)
    posQueue.Enqueue(posList)

    { Mencari hingga posQueue kosong }
    while (posQueue.Count > 0) do
        path ← posQueue.Dequeue()
        currNode ← path[path.Count - 1]
        nextPos ←currNode

        { Mencatat rute pencarian }
        next ← path
        ChangeList(prev, next, reference searchPos)

        { Jika currNode adalah blok Treasure yang belum diambil, ambil
Treasure }
        if (this.map.GetBlock(currNode).isTreasure() and
not((Treasure)this.map.GetBlock(currNode)).isTaken()) then
            result ← result + PosToString(path)
            search ← search + PosToString(searchPos)
            took ← true
            this.map.GetBlock (currNode).step()
            { Tujuan tercapai: mendapatkan Treasure }
            → currNode

        { Jika currNode adalah blok biasa yang belum diinjak, injak blok
itu, lalu masukkan child dari node tersebut ke posQueue }
        if (this.map.GetBlock(currNode).canStep() and
not((Basic)this.map.GetBlock(currNode)).isStepped()) then
            InsertChild(path, currNode, reference posQueue)
            this.map.GetBlock(currNode).step()

        prev ← next

    { posQueue kosong, seluruh reachable node dari startNode telah dicek }
    search ← search + PosToString(searchPos)
```

```
    took ← false
    → nextPos
```

```
function SubBFSTSP(Position startNode, reference string result, reference
string search, reference bool took) → Position

Deklarasi
    posQueue                              : Queue of List of Position
    posList, prev, searchPos, path, next : List of  Position
    nextPos, currNode                     : Position

Algoritma
    { Mencari dari startNode }
    posList.Add(startNode)
    posQueue.Enqueue(posList)

    { Mencari hingga posQueue kosong }
    while (posQueue.Count > 0) do
        path ← posQueue.Dequeue()
        currNode ← path[path.Count - 1]
        nextPos ←currNode

        { Mencatat rute pencarian }
        next ← path
        ChangeList(prev, next, reference searchPos)

        { Jika currNode adalah Start, tujuan tercapai }
        if (currNode.isEqual(this.map.GetStartPos()))then
            result ← result + PosToString(path)
            search ← search + PosToString(searchPos)
            took ← true
            this.map.GetBlock (currNode).step()
            { Tujuan tercapai: kembali ke Start }
            → currNode

        { Jika currNode adalah blok biasa yang belum diinjak, injak blok
itu, lalu masukkan child dari node tersebut ke posQueue }
        if (this.map.GetBlock(currNode).canStep() and
not((Basic)this.map.GetBlock(currNode)).isStepped()) then
            InsertChild(path, currNode, reference posQueue)
            this.map.GetBlock(currNode).step()

        prev ← next

    { posQueue kosong, seluruh reachable node dari startNode telah dicek }
    search ← search + PosToString(searchPos)
    took ← false
    → nextPos
```

Beberapa hal yang menjadi perhatian di sini antara lain sebagai berikut.

- Pada implementasi di sini, simpul direpresentasikan melalu posisinya (baris dan kolomnya) pada matriks.
- Pada BFS, sebenarnya, rute pencarian (bukan rute hasil) tidak kontinu karena pada pengecekannya terjadi “lompatan” dari satu simpul ke simpul yang lain yang belum tentu

bertetangga. Hal ini terjadi karena pengecekan pada BFS bersifat melebar/menyamping. Namun, pada implementasi di sini, "lompatan" ditulis dengan "mundur" sehingga rute pencarian yang dicatat bersifat kontinu.

4.2 Struktur Data yang Digunakan dalam Program dan Spesifikasi Program

Struktur data yang digunakan dalam Program dapat dibedakan menjadi dua tipe, Kelas yang digunakan secara umum, dan struktur data yang dimiliki oleh sebuah kelas. Kelas yang terdapat program kami adalah:

- Kelas Player yang memiliki atribut posisi x dan y serta *method* untuk bergerak
- Kelas Matrix yang memiliki struktur data *two dimensional array of Block*, jumlah kolom dan baris peta, dan jumlah simpul selain tembok dari peta
- Kelas Block yang menjadi Parent dari kelas Basic dan Tembok yang memiliki atribut tetangga, ID, dan warna
- Kelas Basic yang merepresentasikan Block yang bisa "diinjak", dan dapat dispesialisasi menjadi Start dan Treasure, yang memiliki atribut jumlah berapa kali diinjak dan warna dan metode untuk "menginjak"
- Kelas Start yang merepresentasikan Block yang menjadi titik mulai
- Kelas Treasure yang akan melakukan *method overriding* terhadap *method* step yang disediakan oleh kelas Basic dan menyimpan sebuah Boolean sudah diambil atau belum serta sebuah atribut statik yang menghitung berapa treasure yang sudah terambil
- Kelas SearchAlgorithm yang menjadi interface untuk Kelas DFS dan BFS
- Kelas DFS yang menggunakan Struktur data Dictionary dan Stack serta mengimplementasikan *method* StartSearch, Search, insertChild, reverseMove, insertNode, dan getChild
- Kelas BFS yang memiliki atribut objek Matrix yang memiliki metode untuk mencari rute pencarian Treasure pada Matrix tersebut

Struktur data yang dimiliki oleh kelas-kelas yang digunakan dalam program adalah sebagai berikut.

- Sebuah Stack berisikan tuple (char, Block) yang digunakan untuk menyimpan Block yang disimpan untuk ditelusuri secara DFS dan langkah yang diperlukan untuk mencapai Block tersebut dari Block sebelumnya
- Sebuah Dictionary dengan key int dan value string dimana key merupakan Block ID, dan value merupakan rute yang sudah melewati Block tersebut
- Ini buat yang BFS
- Sebuah *two dimensional array of Block* yang digunakan untuk menyimpan peta yang sudah di-*mapping* menjadi Block

Berikut adalah program yang telah dibuat.

Matrix.cs

```
using System;
using Blocks;
using Positions;
using System.Drawing;
```

```
using System.IO;

namespace Matrices
{
    public class Matrix
    {
        private int nCol, nRow;
        private Block[,] mat;
        private Position start;
        public static int NumOfSteppableNodes = 0;
        private const string assetsPath = "./assets/";

        public Matrix(int rows, int cols)
        {
            mat = new Block[rows, cols];
            nRow = rows;
            nCol = cols;
            start = new Position();
        }

        public Matrix(string path)
        {
            Treasure.resetTreasure();
            NumOfSteppableNodes = 0;
            string[] rows = File.ReadAllLines(path);
            nRow = rows.Length;
            nCol = rows[0].Length / 2 + 1;
            mat = new Block[nRow, nCol];
            start = new Position();
            int id = 1;
            for (int i = 0; i < nRow; i++)
            {
                for (int j = 0; j < nCol; j++)
                {
                    char c = rows[i][j * 2];
                    switch (c)
                    {
                        case 'T':
                            mat[i, j] = new Treasure();
                            mat[i, j].setID(id);
                            id++;
                            NumOfSteppableNodes++;
                            break;
                        case 'K':
                            mat[i, j] = new Start();
                            mat[i, j].setID(id);
```

```
                    id++;
                    start.setI(i);
                    start.setJ(j);
                    NumOfSteppableNodes++;
                    break;
                case 'X':
                    mat[i, j] = new Tembok();
                    mat[i, j].setID(id);
                    id++;
                    break;
                case 'R':
                    mat[i, j] = new Basic();
                    mat[i, j].setID(id);
                    id++;
                    NumOfSteppableNodes++;
                    break;
                case ' ':
                    continue;
                default:
                    throw new Exception("Invalid characters detected");
            }
        }
    }
    for (int i = 0; i < nRow; i++)
    {
        for (int j = 0; j < nCol; j++)
        {
            getNeighbours(i, j);
        }
    }
}

public Block GetBlock(int i, int j) { return mat[i, j]; }

public Block GetBlock(Position p) { return mat[p.getI(), p.getJ()]; }

public Block GetStart() { return mat[start.getI(), start.getJ()]; }

public Position GetStartPos() { return start; }

public void getNeighbours(int i, int j)
{
    // Get all the neighbours of block i, j
    // Left
    if (j > 0)
    {
```

```
            // If can step, then not tembok
            if (this.mat[i, j - 1].canStep())
            {
                this.mat[i, j].setL(this.mat[i, j - 1]);
            }

        }
        // Right
        if (j < nCol - 1)
        {
            // If can step, then not tembok
            if (this.mat[i, j + 1].canStep())
            {
                this.mat[i, j].setR(this.mat[i, j + 1]);
            }
        }
        // Up
        if (i > 0)
        {
            // If can step, then not tembok
            if (this.mat[i - 1, j].canStep())
            {
                this.mat[i, j].setU(this.mat[i - 1, j]);
            }
        }
        // Down
        if (i < nRow - 1)
        {
            // If can step, then not tembok
            if (this.mat[i + 1, j].canStep())
            {
                this.mat[i, j].setD(this.mat[i + 1, j]);
            }
        }
    }

    public void resetMatrixStep() {
        for (int i = 0; i < nRow; i++)
        {
            for (int j = 0; j < nCol; j++) {
                if (this.mat[i, j].canStep())
                    ((Basic)this.mat[i, j]).resetStep();
            }
        }

    }
```

```
public void resetEverything()
{
    for (int i = 0; i < nRow; i++)
    {
        for (int j = 0; j < nCol; j++)
        {
            if (this.mat[i, j].canStep())
            {
                ((Basic)this.mat[i, j]).resetStep();
                if (this.mat[i, j].isTreasure())
                {
                    ((Treasure)this.mat[i, j]).resetTaken();
                }
            }
        }
    }
    Treasure.resetTreasureTaken();
}

public int GetNumOfSteppedNodes() {
    int NumOfSteppedNodes = 0;
    for (int i = 0; i < nRow; i++) {
        for (int j = 0; j < nCol; j++) {
            if (((Basic)this.mat[i, j]).getStepCount() > 0) {
                NumOfSteppedNodes++;
            }
        }
    }
    return NumOfSteppedNodes;
}

public override string ToString()
{
    string res = "";
    for (int i = 0; i < nRow; i++)
    {
        for (int j = 0; j < nCol; j++)
        {
            res += mat[i, j].ToString();
            if (j != nCol - 1)
            {
                res += " ";
            }
        }
        if (i != nRow - 1)
```

```
            {
                res += "\n";
            }
        }
        return res;
    }

    public void stepAt(int i, int j) { mat[i, j].step(); }

    public void stepAt(Position p) { mat[p.getI(), p.getJ()].step(); }

    public void walk(string walkPath)
    {
        resetEverything();
        Position currPos = new Position(this.start);
        foreach (char dir in walkPath)
        {
            this.stepAt(currPos);
            currPos.move(dir);
        }
        this.stepAt(currPos);
    }

    // VISUALIZATION
    const int squareSize = 200;
    const int pad = 20;
    const int sidePad = 20;

    public void animateWalk(string folderPath, string walkPath, string playerPath)
    {
        resetEverything();

        int minHeight, minWidth;

        minHeight = nRow * squareSize + (nRow - 1) * pad + 2 * sidePad;
        minWidth = nCol * squareSize + (nCol - 1) * pad + 2 * sidePad;

        Bitmap image = new Bitmap(minWidth, minHeight);
        Graphics graphic = Graphics.FromImage(image);

        visualize(ref graphic);

        Position curr = new Position(this.start);
        stepAt(curr);
        rerenderPtr(ref graphic, curr, playerPath);
        Position prev = new Position(curr);
```

```
            image.Save(folderPath + (1).ToString() + ".png", System.Drawing.Imaging.ImageFormat.Png);

            for (int i = 0; i < walkPath.Length; i++)
            {
                rerenderAt(ref graphic, prev, false);
                curr.move(walkPath[i]);
                stepAt(curr);
                rerenderPtr(ref graphic, curr, playerPath);
                prev.setI(curr.getI());
                prev.setJ(curr.getJ());
                image.Save(folderPath + (i + 2).ToString() + ".png", System.Drawing.Imaging.ImageFormat.Png);
            }
        }

        public void rerenderPtr(ref Graphics graphic, Position p, string playerPath)
        {
            int i, j, currX, currY;
            i = p.getI();
            j = p.getJ();
            currX = sidePad + j * (pad + squareSize);
            currY = sidePad + i * (pad + squareSize);

            Image player = resizeImage(Image.FromFile(assetsPath + playerPath), new Size(squareSize, squareSize));
            graphic.DrawImage(player, new Point(currX, currY));
        }

        public void rerenderAt(ref Graphics graphic, Position p, bool alphaOn)
        {
            int i = p.getI();
            int j = p.getJ();

            Image baseBlock = resizeImage(Image.FromFile(assetsPath + "baseblock.png"), new Size(squareSize,
squareSize));
            Image closedTreasure = resizeImage(Image.FromFile(assetsPath + "chest-closed.png"), new Size(squareSize,
squareSize));
            Image openedTreasure = resizeImage(Image.FromFile(assetsPath + "chest-opened.png"), new
Size(squareSize, squareSize));
            Image startBlock = resizeImage(Image.FromFile(assetsPath + "start.png"), new Size(squareSize, squareSize));

            if (!this.mat[i, j].isTembok())
            {
                Point currPoint = new Point(sidePad + j * (pad + squareSize), sidePad + i * (pad + squareSize));

                graphic.DrawImage(baseBlock, currPoint);

                SolidBrush stepBrush;
```

```
            if (((Basic)this.mat[i, j]).isStepped())
            {
                if (alphaOn)
                {
                    stepBrush = new SolidBrush(this.mat[i, j].getColor());
                } else
                {
                    stepBrush = new SolidBrush(Color.FromArgb(50, 255, 0, 0));
                }

                Rectangle stepRect = new Rectangle(currPoint, new Size(squareSize, squareSize));
                graphic.FillRectangle(stepBrush, stepRect);
            }

            if (this.mat[i, j].isTreasure())
            {
                if (((Treasure)this.mat[i, j]).isTaken())
                {
                    graphic.DrawImage(openedTreasure, currPoint);
                }
                else
                {
                    graphic.DrawImage(closedTreasure, currPoint);
                }
            }
            else if (i == this.start.getI() && j == this.start.getJ())
            {
                graphic.DrawImage(startBlock, currPoint);
            }
        }
    }

    public void visualize(string path)
    {
        int minHeight = nRow * squareSize + (nRow - 1) * pad + 2 * sidePad;
        int minWidth = nCol * squareSize + (nCol - 1) * pad + 2 * sidePad;

        Bitmap image = new Bitmap(minWidth, minHeight);
        Graphics graphic = Graphics.FromImage(image);

        visualize(ref graphic);

        image.Save(path, System.Drawing.Imaging.ImageFormat.Png);
    }

    public void visualize(ref Graphics graphic)
```

```
        {
            int minHeight = nRow * squareSize + (nRow - 1) * pad + 2 * sidePad;
            int minWidth = nCol * squareSize + (nCol - 1) * pad + 2 * sidePad;

            Image bg = new Bitmap(assetsPath + "background.png");
            TextureBrush tBrush = new TextureBrush(bg);
            graphic.FillRectangle(tBrush, new Rectangle(0, 0, minWidth, minHeight));

            for (int i = 0; i < nRow; i++)
            {
                for (int j = 0; j < nCol; j++)
                {
                    rerenderAt(ref graphic, new Position(i, j), true);
                }
            }
        }

        public void visualizeAll(string folderPath, string route, string search)
        {
            // clean directory
            Directory.CreateDirectory(folderPath);

            DirectoryInfo di = new DirectoryInfo(folderPath);

            foreach (FileInfo file in di.GetFiles())
            {
                file.Delete();
            }
            foreach (DirectoryInfo dir in di.GetDirectories())
            {
                dir.Delete(true);
            }

            // select player randomly
            string[] playersPath = { "gary.png", "mrkrabs.png", "patrick.png", "plankton.png", "sandy.png",
"squidward.png" };
            Random rnd = new Random();
            string player = playersPath[rnd.Next(playersPath.Length)];

            // visualize kosong (belom diapa-apain, 1 gambar)
            resetEverything();
            visualize(folderPath + "0.png");

            //// visualize search (len(search) + 1 gambar)
            animateWalk(folderPath, search, player);
```

```
            // visualize route (1 gambar)
            walk(route);
            visualize(folderPath + (search.Length + 2).ToString() + ".png");
        }

        public static Image resizeImage(Image imgToResize, Size size)
        {
            return new Bitmap(imgToResize, size);
        }
    }
}
```

SearchAlgorithm.cs

```
namespace Tubes2_Stima.src
{
    abstract public class SearchAlgorithm {}
}
```

DFS.cs

```
using Blocks;
using System;
using System.Collections.Generic;

namespace Tubes2_Stima.src
{
    class DFS : SearchAlgorithm
    {
        public Stack<(char move, Block node)> NodeMoves = new Stack<(char move, Block node)>();

        public Dictionary<int, string> BlockIDMovesMapping = new Dictionary<int, string>();

        public static int NumOfSteps = 0;

        public int numOfTreasure = 0;

        public DFS(int numOfTreasure)
        {
            this.numOfTreasure = numOfTreasure;
        }

        public void insertNode(Block n, char a)
        {
            (char move, Block node) temp = (a, n);
            this.NodeMoves.Push(temp);
        }
```

```
public void insertChild(Block n, char lastMove, ref bool notDeadend)
{
    // Priority : L U R D
    if (n.hasL && !(lastMove == 'R'))
    {
        insertNode(n.getL(), 'L');
        notDeadend = true;
    }
    if (n.hasU && !(lastMove == 'D'))
    {
        insertNode(n.getU(), 'U');
        notDeadend = true;
    }
    if (n.hasR && !(lastMove == 'L'))
    {
        insertNode(n.getR(), 'R');
        notDeadend = true;
    }
    if (n.hasD && !(lastMove == 'U'))
    {
        insertNode(n.getD(), 'D');
        notDeadend = true;
    }
}

public (char move, Block node) getChild()
{
    return this.NodeMoves.Pop();
}

public string startSearch(Block n, bool TSP)
{
    string moves = "";
    char lastMove = 'S';
    moves = Search(n, ref lastMove, moves, TSP);
    // tinggal return moves
    if (Treasure.getTreasureCount() != Treasure.getTreasureTaken())
    {
        Console.WriteLine("No path found");
        return "";
    }
    moves = moves.Replace("S", string.Empty);
    NumOfSteps = moves.Length;
    return moves;
}
```

```
    public void reverseMove(ref string currentMoves, char lastMove)
    {
        if (lastMove == 'L')
        {
            currentMoves += "R";
        }
        if (lastMove == 'U')
        {
            currentMoves += "D";
        }
        if (lastMove == 'R')
        {
            currentMoves += "L";
        }
        if (lastMove == 'D')
        {
            currentMoves += "U";
        }
    }

    public string Search(Block node, ref char lastMove, string moves, bool TSP)
    {
        string currentMoves = moves;

        if (Treasure.getTreasureCount() == Treasure.getTreasureTaken())
        {
            moves = currentMoves;
            return currentMoves;
        }
        else
        {
            node.step();
            // di step nya treasure, tambahin num of treasure++
            // terus nanti get num of gotten treasurenya, bandingin sama total treasure
            // tambahin currentMove. terus kasih ke block ID
            currentMoves += lastMove;
            this.BlockIDMovesMapping[node.getID()] = currentMoves;

            // cari child nya yang bukan parentnya
            bool notDeadend = false;
            insertChild(node, lastMove, ref notDeadend);

            // kalo TSP nanti pas treasure countnya dah sesuai, tambahin start sebagai treasure terakhir (beda
module aja)
            // Udah di taken, tinggal ditambahin treasure count, kalau misal udah sama yauda stop, kaloga mulai
balik ke start kalo TSP
```

```
        // if (treasure udah keambil semua) { kalo TSP, backtrack (reverse moves nya ato cari Start sbg Treasure terakhir, boleh, jangan lupa aja block ID nya di reset semua, kalau ga yauda beresin aja langsung return)}

        if (notDeadend)
        {
            // lanjut
            (char move, Block node) Child;
            char nextMove;
            Block nextBlock;
            if (lastMove == 'S')
            {
                for (int i = 0; i < node.getNumOfChild(); i++)
                {
                    // Search untuk semua child yang dimiliki oleh node yang sedang diinjak
                    Child = getChild();
                    nextMove = Child.move;
                    nextBlock = Child.node;
                    if (
                        BlockIDMovesMapping.ContainsKey(nextBlock.ID)
                        && currentMoves.StartsWith(BlockIDMovesMapping[nextBlock.ID])
                    )
                    {
                        // Block yang akan dikunjungi sudah pernah dikunjungi oleh track yang sama
                        // return currentMoves;
                    }
                    else
                    {
                        currentMoves = Search(nextBlock, ref nextMove, currentMoves, TSP);
                    }
                }

                if (TSP || !(Treasure.getTreasureCount() == Treasure.getTreasureTaken()))
                {
                    reverseMove(ref currentMoves, lastMove);
                }
            }
            else
            {
                for (int i = 0; i < node.getNumOfChild() - 1; i++)
                {
                    // Search untuk semua child yang dimiliki oleh node yang sedang diinjak
                    Child = getChild();
                    nextMove = Child.move;
                    nextBlock = Child.node;
                    if (
```

```
                    BlockIDMovesMapping.ContainsKey(nextBlock.ID)
                    && currentMoves.StartsWith(BlockIDMovesMapping[nextBlock.ID])
                )
                {
                    // Block yang akan dikunjungi sudah pernah dikunjungi oleh track yang sama
                    // return currentMoves;
                }
                else
                {
                    currentMoves = Search(nextBlock, ref nextMove, currentMoves, TSP);
                }
            }
            if (TSP || !(Treasure.getTreasureCount() == Treasure.getTreasureTaken()))
            {
                reverseMove(ref currentMoves, lastMove);
            }
        }

        return currentMoves;
    }
    else
    {
        // deadend, backtrack
        if (TSP || !(Treasure.getTreasureCount() == Treasure.getTreasureTaken()))
        {
            reverseMove(ref currentMoves, lastMove);
            if (lastMove == 'L')
            {
                lastMove = 'R';
            }
            else if (lastMove == 'U')
            {
                lastMove = 'D';
            }
            else if (lastMove == 'R')
            {
                lastMove = 'L';
            }
            else if (lastMove == 'D')
            {
                lastMove = 'U';
            }
        }

        return currentMoves;
    }
```

```
        }
      }
   }
}
```

BFS.cs

```
using Blocks;
using Matrices;
using Positions;
using System.Collections.Generic;
using System;

namespace Tubes2_Stima.src
{
   class BFS : SearchAlgorithm
   {
      private Matrix map;
      public BFS(Matrix map) { this.map = map; }


      public void InsertNode(ref List<Position> list, Position p)
      {
         list.Add(p);
      }
      public void InsertChild(List<Position> path, Position currNode, ref Queue<List<Position>> posQueue)
      {
         if (this.map.GetBlock(currNode).hasL)
         {
            Position temp = currNode.getLPos();
            if (!((Basic)this.map.GetBlock(temp)).isStepped())
            {
               List<Position> newPath = new List<Position>(path);
               InsertNode(ref newPath, temp);
               posQueue.Enqueue(new List<Position>(newPath));
            }
         }
         if (this.map.GetBlock(currNode).hasU)
         {
            Position temp = currNode.getUPos();
            if (!((Basic)this.map.GetBlock(temp)).isStepped())
            {
               List<Position> newPath = new List<Position>(path);
               InsertNode(ref newPath, temp);
               posQueue.Enqueue(new List<Position>(newPath));
            }
         }
         if (this.map.GetBlock(currNode).hasR)
```

```
        {
            Position temp = currNode.getRPos();
            if (!((Basic)this.map.GetBlock(temp)).isStepped())
            {
                List<Position> newPath = new List<Position>(path);
                InsertNode(ref newPath, temp);
                posQueue.Enqueue(new List<Position>(newPath));
            }
        }
        if (this.map.GetBlock(currNode).hasD)
        {
            Position temp = currNode.getDPos();
            if (!((Basic)this.map.GetBlock(temp)).isStepped())
            {
                List<Position> newPath = new List<Position>(path);
                InsertNode(ref newPath, temp);
                posQueue.Enqueue(new List<Position>(newPath));
            }
        }

    }

    public static string PosToString(List<Position> moves)
    {
        string movesDir = "";
        for (int i = 0; i < moves.Count - 1; i++)
        {
            movesDir += moves[i].getDirTo(moves[i + 1]);
        }
        return movesDir;
    }

    public static void PrintQueue(Queue<List<Position>> q)
    {
        Console.Write("{");
        foreach (List<Position> l in q)
        {
            Console.Write("[");
            foreach (Position p in l)
            {
                Console.Write(p);
            }
            Console.Write("] ");
        }
        Console.Write("}");
        Console.Write("\n");
```

```
    }

    public static void PrintList(List<Position> l)
    {
        Console.Write("[");
        foreach (Position p in l)
        {
            Console.Write(p);
        }
        Console.Write("] ");
        Console.Write("\n");
    }

    public static void ChangeList(List<Position> prev, List<Position> curr, ref List<Position> search)
    {
        if (prev.Count >= 2)
        {
            int lastEq = 0;
            for (int i = 0; i < prev.Count; i++)
            {
                if (!prev[i].isEqual(curr[i]))
                {
                    break;
                }
                else
                {
                    lastEq = i;
                }
            }
            for (int i = prev.Count - 2; i >= lastEq; i--)
            {
                search.Add(prev[i]);
            }
            for (int i = lastEq + 1; i < curr.Count; i++)
            {
                search.Add(curr[i]);
            }
        }
        else
        {
            search.Add(curr[curr.Count - 1]);
        }
    }

    public Position SubBFS(Position startNode, ref string result, ref string search, ref bool took)
    {
```

```
    Queue<List<Position>> posQueue = new Queue<List<Position>>();

    List<Position> posList = new List<Position>();
    posList.Add(startNode);
    posQueue.Enqueue(new List<Position>(posList));

    List<Position> prev = new List<Position>();
    List<Position> searchPos = new List<Position>();
    Position nextPos = new Position();

    while (posQueue.Count > 0)
    {
        List<Position> path = new List<Position>(posQueue.Dequeue());
        Position currNode = new Position(path[path.Count - 1]);
        nextPos = new Position(currNode);
        List<Position> next = path;
        ChangeList(prev, next, ref searchPos);

        if (this.map.GetBlock(currNode).isTreasure() && !((Treasure)this.map.GetBlock(currNode)).isTaken())
        {
            result += PosToString(path);
            search += PosToString(searchPos);
            took = true;
            this.map.GetBlock (currNode).step();
            return currNode;
        }
        if (this.map.GetBlock(currNode).canStep() && !((Basic)this.map.GetBlock(currNode)).isStepped())
        {
            InsertChild(path, currNode, ref posQueue);
            this.map.GetBlock(currNode).step();
        }
        prev = next;
    }
    search += PosToString(searchPos);
    took = false;
    return nextPos;
}

public Position SubBFSTSP(Position startNode, ref string result, ref string search)
{
    Queue<List<Position>> posQueue = new Queue<List<Position>>();

    List<Position> posList = new List<Position>();
    posList.Add(startNode);
    posQueue.Enqueue(new List<Position>(posList));
```

```
        List<Position> prev = new List<Position>();
        List<Position> searchPos = new List<Position>();

        while (posQueue.Count > 0)
        {
            List<Position> path = new List<Position>(posQueue.Dequeue());
            Position currNode = new Position(path[path.Count - 1]);
            List<Position> next = new List<Position>(path);
            ChangeList(prev, next, ref searchPos);

            if (currNode.isEqual(this.map.GetStartPos()))
            {
                search += PosToString(searchPos);
                result += PosToString(path);
                this.map.GetBlock(currNode).step();
                return currNode;
            }
            if (this.map.GetBlock(currNode).canStep() && !((Basic)this.map.GetBlock(currNode)).isStepped())
            {
                InsertChild(path, currNode, ref posQueue);
                this.map.GetBlock(currNode).step();
            }
            prev = next;
        }
        search += PosToString(searchPos);
        return new Position();
    }

    public string Search(bool tsp, ref string search)
    {
        string moves = "";
        bool took = false;
        Position next = SubBFS(this.map.GetStartPos(), ref moves, ref search, ref took);

        while (took && !Treasure.isAllTaken())
        {
            map.resetMatrixStep();
            next = new Position(SubBFS(next, ref moves, ref search, ref took));
        }

        if (tsp)
        {
            map.resetMatrixStep();
            SubBFSTSP(next, ref moves, ref search);
        }
```

```
            return moves;
        }
    }
}
```

4.3 Tata Cara Penggunaan Program

Untuk menjalankan program, pengguna dapat menjalankan dengan beberapa cara. Cara yang pertama adalah untuk masuk ke IDE Visual Studio dan melakukan 'Build Solution'. Cara lainnya adalah melakukan navigasi ke folder Tubes2_Stima/bin/Release dan menjalankan Tubes2_Stima.exe. Setelah berada di aplikasi, lakukan:

1. Masukkan file dengan tombol filename
2. Pilih algoritma (DFS/BFS)
3. Pilih menggunakan TSP atau tidak
4. Tekan tombol search
5. Setelah tulisan pada routes, execution time, nodes, dan step muncul, lakukan penggeseran pada slider untuk mengatur delay time visualisasi
6. Tekan tombol visualisasi
7. Jika visualisasi sudah selesai atau ingin diulang, tekan tombol visualisasi lagi untuk mematikan, dan sekali lagi untuk menyalakan kembali
8. Pengaturan kecepatan animasi bisa dilakukan ditengah visualisasi
9. Jika sudah selesai, tekan tombol visualisasi kembali, bagian gambar akan hilang menjadi warna coklat
10. Ulangi langkah 1 untuk pencarian berikutnya

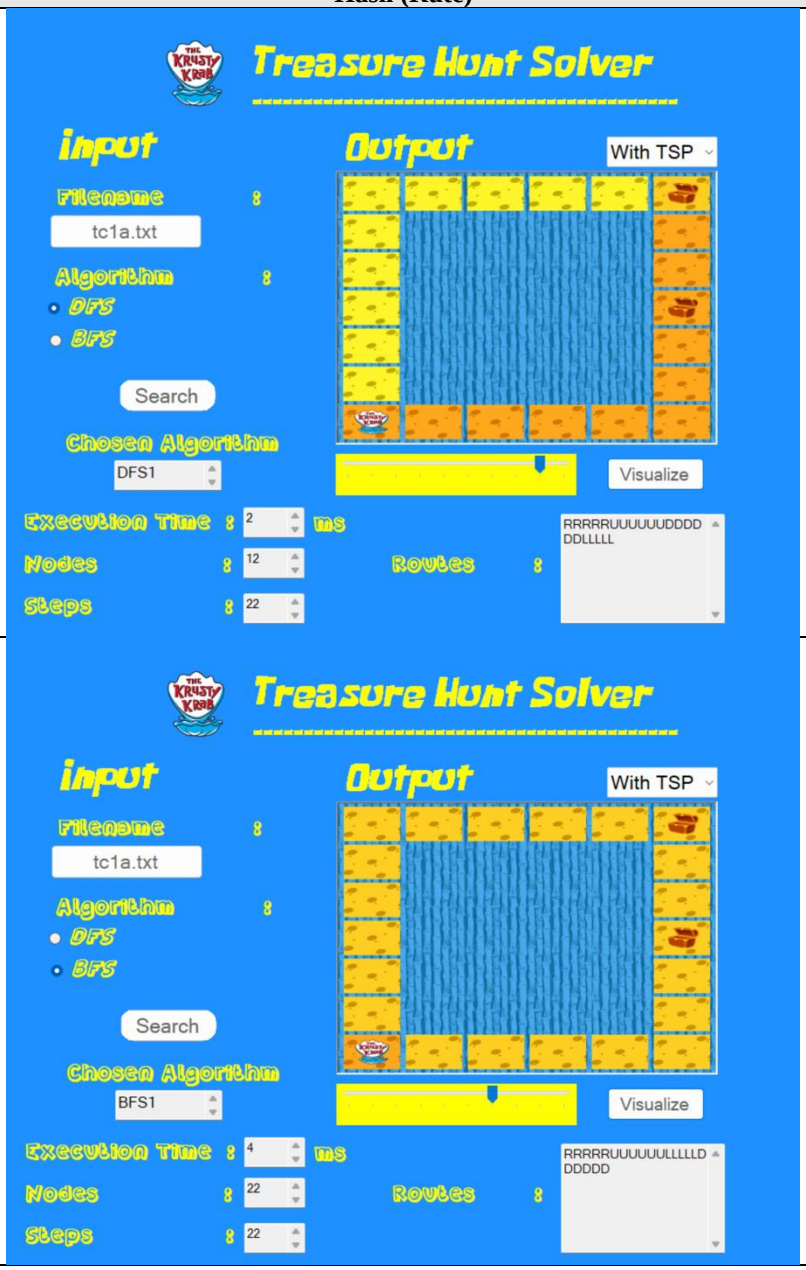
4.4 Hasil Pengujian

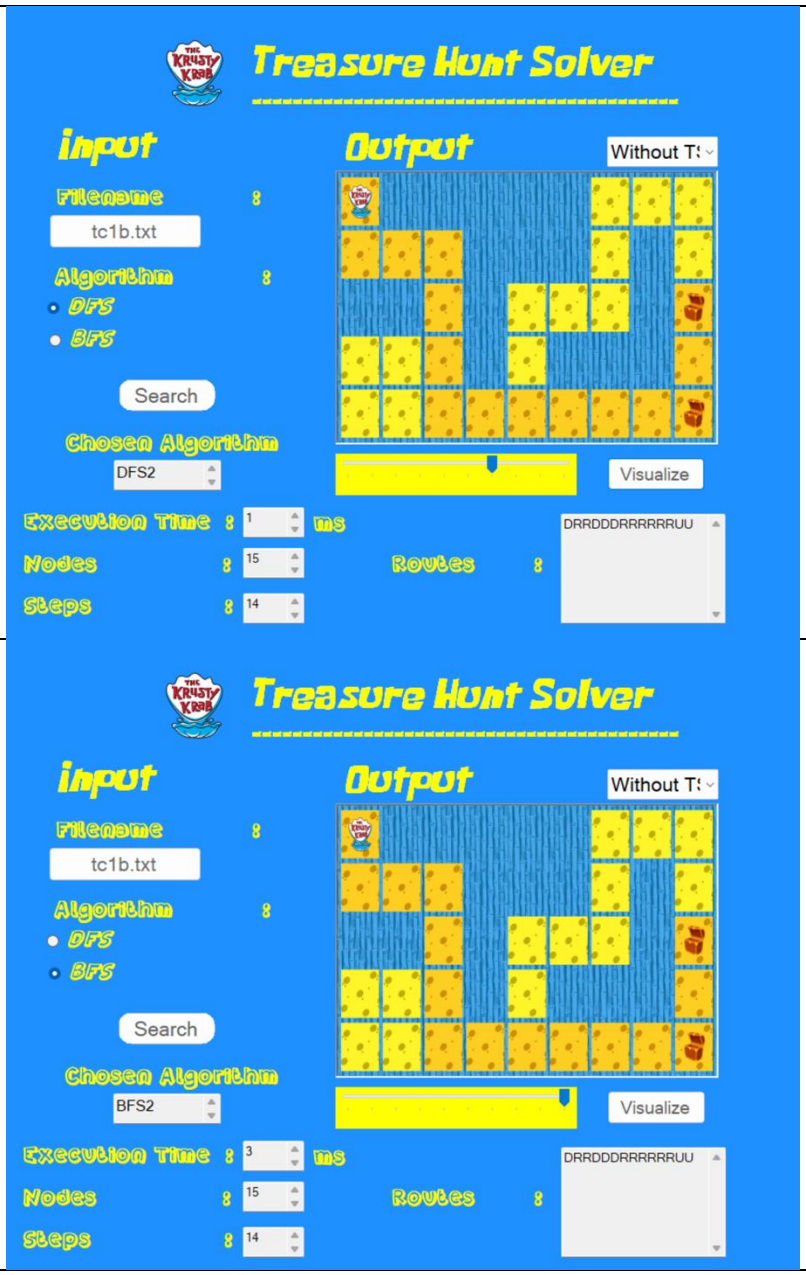
Sebagai keterangan, berikut adalah legenda untuk visualisasi pada hasil pengujian.

1. Kotak berwarna kuning bermotif Spongebob melambangkan blok yang dapat diinjak.
2. Ikon "Krusty Krab" melambangkan blok Start.
3. Ikon peti terbuka melambangkan blok Treasure yang sudah diambil.
4. Ikon peti tertutup melambangkan blok Treasure yang belum diambil.
5. Ikon karakter, yaitu salah satu dari Patrick, Mr. Krabs, Squidward, Gary, Plankton, atau Sandy (dipilih secara acak), melambangkan pemain yang sedang mengecek balok yang diinjak oleh karakter tersebut.
6. Kotak yang berwarna lebih tua (kemerahan) melambangkan blok yang sudah diinjak. Semakin tua warna suatu kotak, semakin banyak pula jumlah langkah yang menginjak kotak (blok) tersebut.

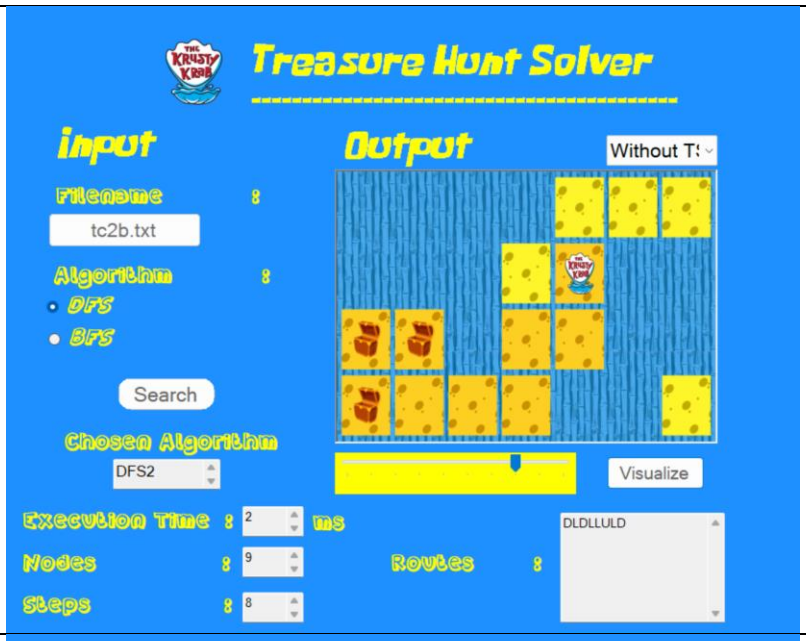
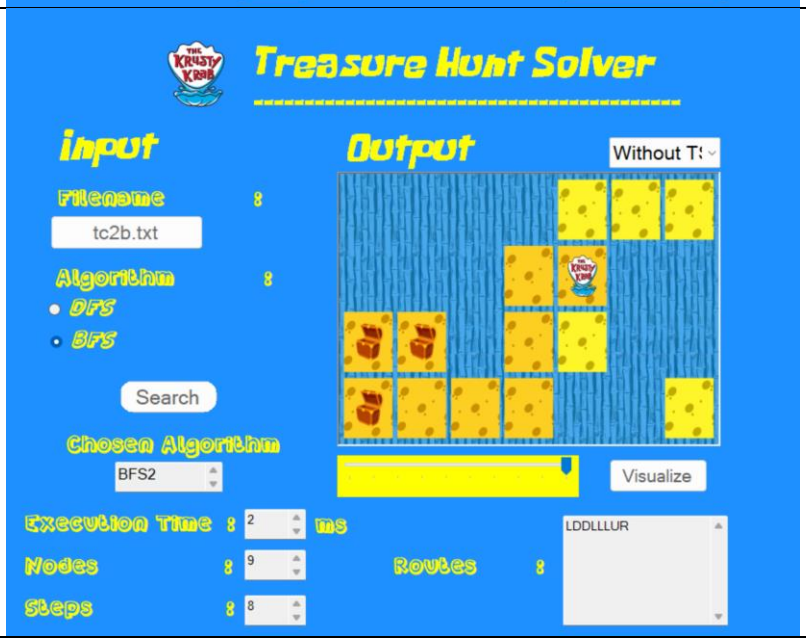
Berikut merupakan masukan dan keluaran program.

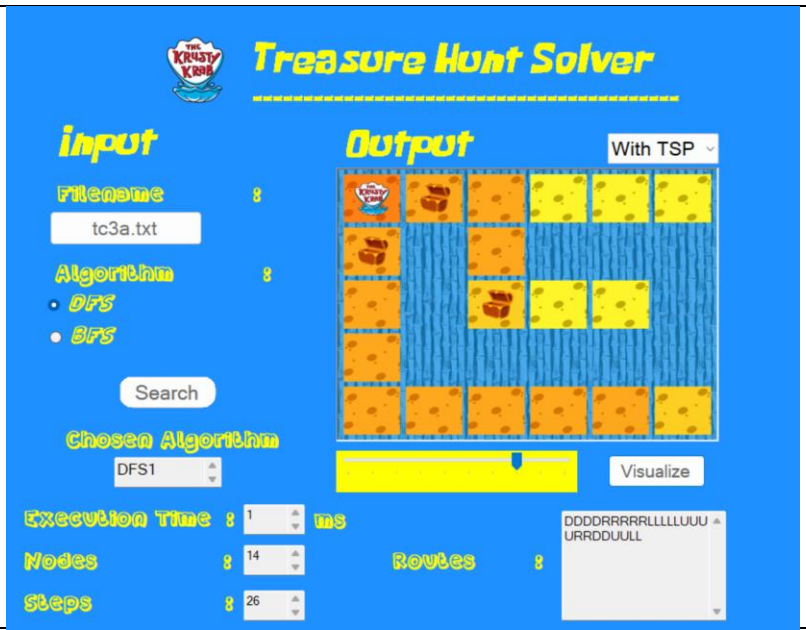
Tabel 4.4.1 Hasil Pengujian

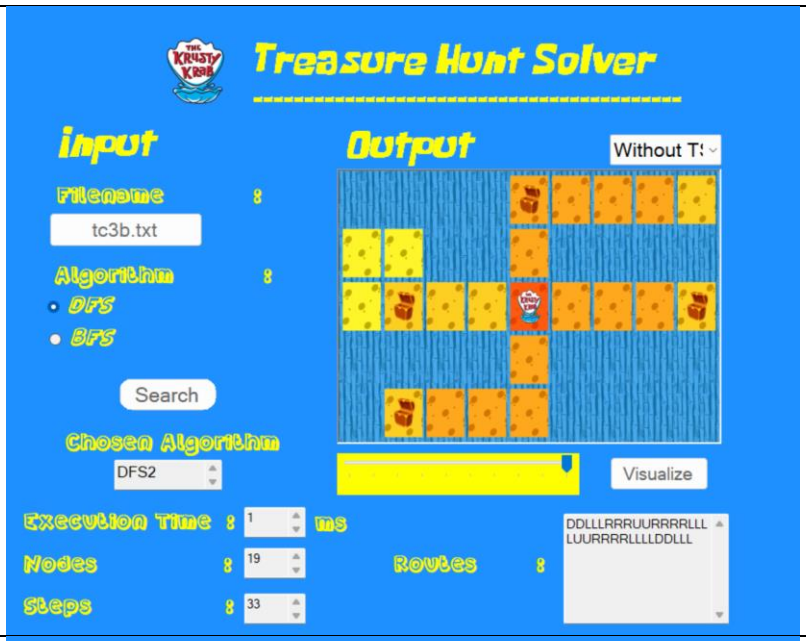
Kode	Masukan *File*	Hasil (Rute)
1a	R R R R R T R X X X X R R X X X X R R X X X X T R X X X X R R X X X X R K R R R R R	

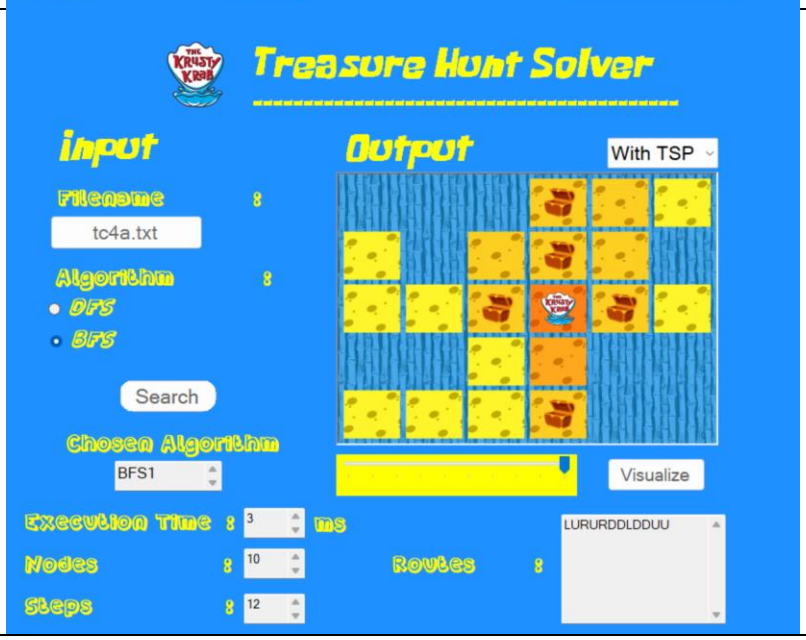
1b	K X X X X X R R R R R R X X X R X R X X R X R R R X T R R R X R X X X R R R R R R R R R T	

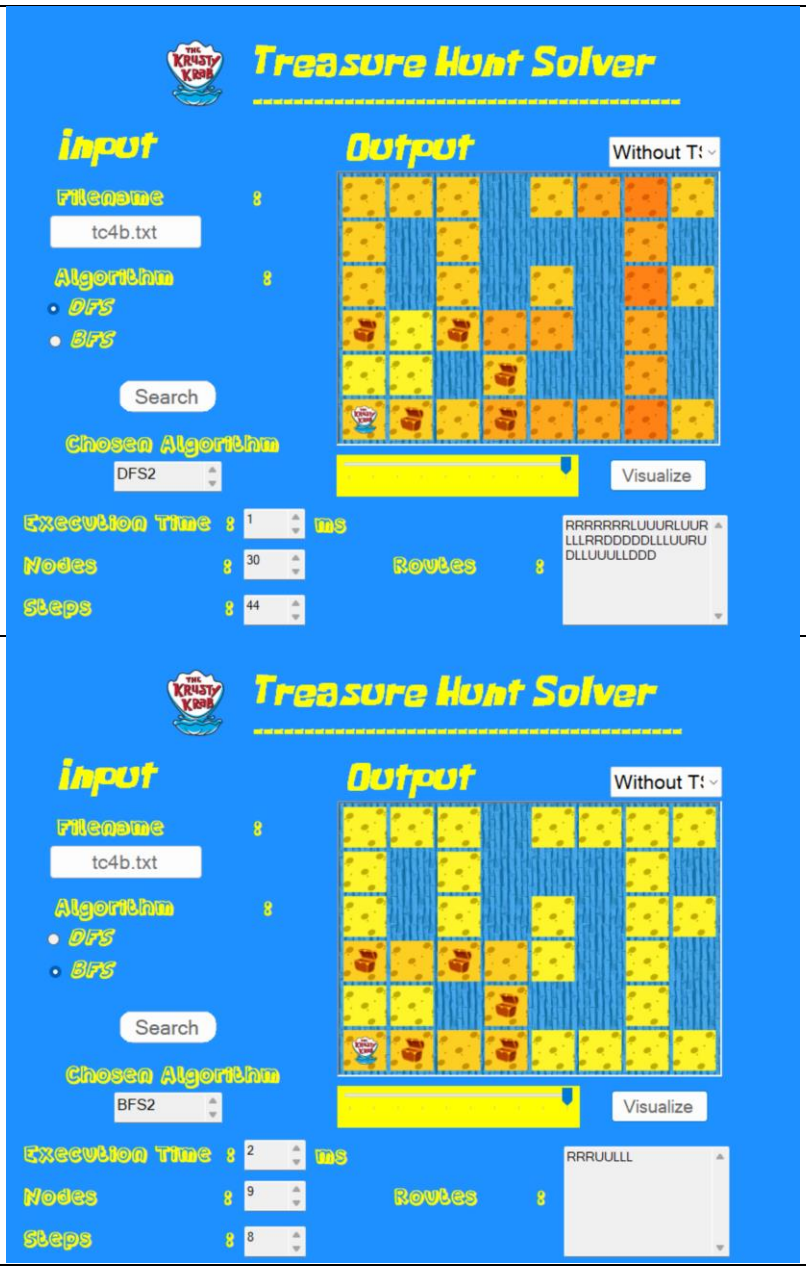
2a	R R R X T R X X T T R X R R R K R R X X	 

2b	X X X X R R R X X X R K X X T T X R R X X T R R R X X R	 

3a	K T R R R R T X R X X X R X T R R X R X X X X X R R R R R R	 

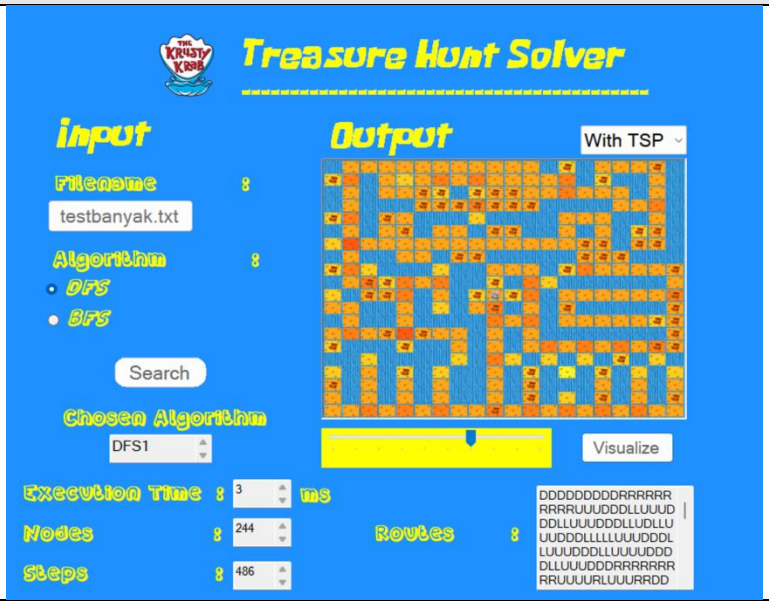
3b	X X X X T R R R R R R X X R X X X X R T R R K R R R T X X X X R X X X X X T R R R X X X X	 

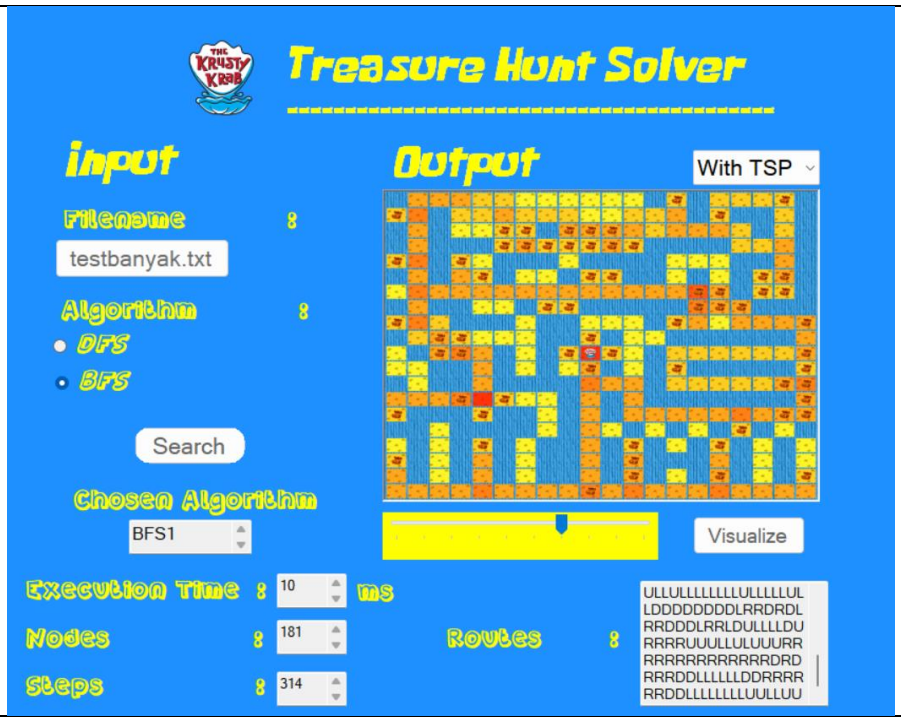
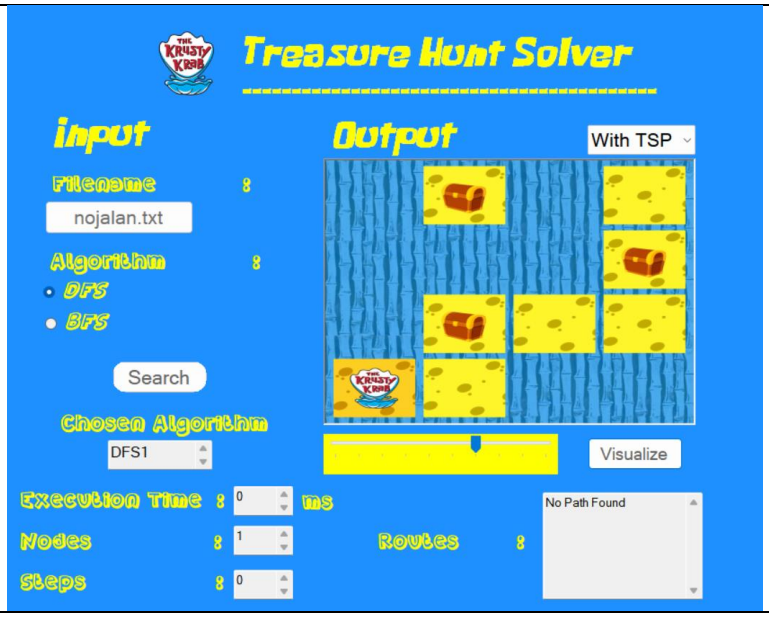
4a	X X X T R R R X R T R X R R T K T R X X R R X X R R R T X X	 

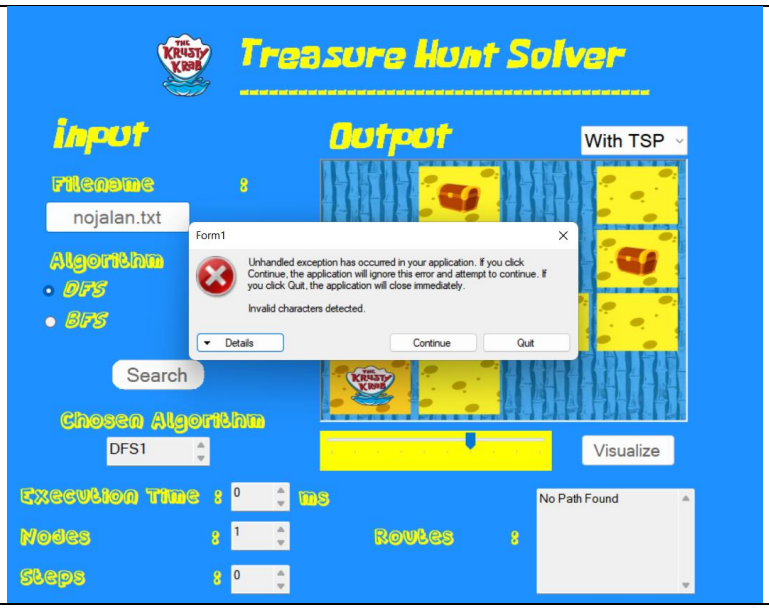
4b	R R R X R R R R R X R X X X R X R X R X R X R R T R T R R X R X R R X T X X R X K T R T R R R R	

Berikut merupakan hasil dari permasalahan menarik yang penulis temukan selama mengerjakan tugas kecil ini.

Tabel 4.4.2 Hasil Pengujian yang Menarik

No	Hasil (Rute)
1	
	DDDDDDDDDRRRRRRRRRRUUUDDDLLUUUDDDLLUUUDDDLLUDLLUUUDDDLLLLLUUUDDD LLUUUDDDLLUUUUDDDDLLUUUDDDRRRRRRRRRUUUURLUUURRDDRDURRDURRDURRDU RUUUUUULLLLULDLUURUULLDDLLLULDLDLUULDLDLUUULDDLLDDDRDRRDDDDURRRDD UULLLLLLUULUDRDDLDURRRRUUUURRDDUUUDLLLRDLLUUDLULRUUUUUUUURRDDRRD RRRDURRRURRRRDRRDDDLUDRUUUUUULLLDURRRDDDLLULLLUUDLLULDDLUULDDUL ULDDULULDURDRURDRURDDRUURDRRDLLDLLLLLLULLUULLDLRDDDLRDDLRRRUURDDD RURURDDRURURDRRRUURRDDLDDLRRURDRRRDDLLLLLLDDRRRRRLLLLLUURRRRRRDDD DLLLLLLLLUUUUURLULLDURRDDLRDDLLULUDRU

		
	LRUDRLDDDDDDDDDRRUUUDDDRRRRUUUDDDRRRRUDLLLLLLLLLLLLLLLLUUUDDDLLLLU UDDRRRRRRRRRUUUUUUUURRDDRRRRRDURRRUULRUUUULLLULLURLDDLRUULLLLULDLD LULLLULUDDLLUULRUUULRURRRRDDRDLRRRURRDLRRUURRUDDRRUURRRDDDDDLDRU UULLULLLLLLLLLULLLLLULLDDDDDDDDDLRRDRDLRRDDDLRRLDULLLLDURRRRUUULLULU UURRRRRRRRRRRRRRDRDRRRDDLLLLLLDDRRRRRRDDLLLLLLLLUULLUU	
2		
	No Path Found	

3		
	Test Case 3 dari Asisten J A N G A N L U P A C E K Y A N G B E G I N I Y	

4.5 Analisis Desain Solusi Algoritma BFS dan DFS yang Diimplementasikan

Pada kasus secara umum, pada pencarian rute, algoritma DFS akan selalu lebih baik daripada algoritma BFS, tetapi BFS lebih sering akan menemukan juga rute dengan jarak terdekat. Hal tersebut terjadi karena DFS akan hanya melakukan *backtracking* saat bertemu jalan buntu, sedangkan BFS akan mengulangi pencarian setiap ditemukan harta kartun. Tetapi ada juga kasus dimana kasus BFS menjadi lebih efisien dibandingkan kasus DFS, yaitu saat terdapat banyak percabangan dan semua harta karun terletak didekat percabangan tersebut seperti pada contoh 3a.

Berikut analisis untuk setiap kasus *testcase* pada bagian 4.4.

Nomor	Algoritma 1	Algoritma 2	Analisis
1a	DFS-TSP	BFS-TSP	Jumlah simpul yang dicek oleh DFS-TSP lebih sedikit daripada yang dicek oleh BFS-TSP, sehingga DFS-TSP lebih baik, tetapi BFS-TSP memberikan jalur pulang alternatif. Hal ini dikarenakan DFS akan langsung *backtracking* sementara BFS akan melakukan pencarian ulang dari simpul harta karun.
1b	DFS	BFS	DFS dan BFS memberikan solusi yang sama persis, walaupun pada kemungkinan besar, BFS membutuhkan pencarian yang lebih lama
2a	DFS-TSP	BFS-TSP	BFS mendapatkan rute dengan jumlah simpul dan langkah yang lebih sedikit, hal ini dikarenakan DFS memiliki prioritas langkah sehingga akan membentuk rute yang sesuai prioritas, walau tidak yang paling dekat
2b	DFS	BFS	DFS dan BFS mengembalikan rute yang serupa, hanya memiliki perbedaan yang terlihat diawal, yang dikarenakan prioritas pengambilan langkah pada DFS
3a	DFS-TSP	BFS-TSP	BFS akan memberikan rute yang lebih efisien, tetapi dengan pengunjungan simpul yang lebih banyak dalam fase pencarian rute. Hal ini dikarenakan DFS mengambil jalur yang panjang namun buntu, sehingga membutuhkan proses *backtrack* yang panjang. Sementara BFS akan mengulang pencarian setiap ditemukan harta karun, maka mendapatkan rute teroptimal, namun perlu banyak simpul yang dicek sekaligus.
3b	DFS	BFS	BFS akan memberikan rute yang lebih efisien, tetapi dengan pengunjungan simpul yang lebih banyak dalam fase pencarian rute. Sama halnya seperti kasus 3a, terdapat harta karun yang tersebar pada banyak cabang, sehingga DFS perlu melakukan *backtrack* untuk mencapai semua harta karun tersebut.
4a	DFS-TSP	BFS-TSP	BFS memberikan juga rute dengan pengunjungan simpul dan langkah yang lebih minimum. Hal ini

			dikarenakan untuk kasus dengan simpul yang terkumpul secara terpusat, pencarian dengan melebar akan dapat dengan lebih mudah mendapatkan semua harta karun dibandingkan jika harus menelusuri 1 jalur sampai buntu dahulu lalu melakukan *backtracking*. Tetapi, akan dibutuhkan lebih banyak waktu karena pencarian yang dilakukan BFS akan lebih banyak.
4b	DFS	BFS	Pada kasus ini, semakin terlihat bahwa pencarian secara BFS akan lebih baik untuk simpul-simpul harta karun yang terkumpul dan terdapat jalur-jalur buntu yang panjang. Karena pada DFS, jalur buntu yang panjang tersebut akan ditelusuri dan akan dilakukan *backtracking*, sehingga akan semakin banyak simpul dan langkah yang dibutuhkan. Tetapi, akan dibutuhkan lebih banyak waktu karena pencarian yang dilakukan BFS akan lebih banyak.
Kasus menarik 1	DFS	BFS	Pada kasus ini, sekilas terlihat bahwa BFS memberikan rute dan langkah yang lebih optimal, tetapi, dibalik layer, terdapat perbedaan kompleksitas yang signifikan. Dimana kasus DFS hanya melakukan sekitar 500 kali pengecekan pencarian simpul, sesuai dengan jumlah langkah yang diambil, sedangkah kasus BFS perlu melakukan sekitar 5000 kali pengecekan. Terdapat perbedaan 10 kali lipat lebih banyak pengecekan dalam BFS, tetapi dengan hasil yang lebih optimal pula. Hal ini dapat dibuktikan dengan counter image (pada folder visualization, DFS memiliki sekitar 500 gambar visualisasi, sedangkan BFS memerlukan sekitar 5000 gambar visualisasi), dan juga waktu eksekusi
Kasus menarik 2	DFS/BFS	-	Pada kasus ini, tidak terdapat sebuah rute yang mendapatkan semua harta karun, sehingga pada Routes akan tertulis “No Path Found”
Kasus menarik 3	-	-	Pada kasus ini, file gagal melewati tahap *preprocessing* atau *mapping* dari .txt menjadi sebuah objek matrix, sehingga di-*throw* sebuah *exception*.

Dari keseluruhan kasus di atas, dapat ditarik kesimpulan sebagai berikut.

1. DFS lebih unggul dibanding BFS saat target-target (Treasure) berada relatif jauh dari titik awal (Start).
2. BFS lebih unggul dibanding DFS saat target-target (Treasure) berada relatif dekat dari titik awal (Start).

3. DFS lebih unggul dibanding BFS saat jalur graf yang diberikan relatif lebih “lurus” dan tidak banyak bercabang.
4. BFS lebih unggul dibanding DFS saat jalur graf yang diberikan memiliki banyak cabang.

BAB V
KESIMPULAN DAN SARAN

5.1 Kesimpulan

Pada tugas besar II IF2211 Strategi Algoritma ini telah diimplementasikan algoritma *breath-first search* dan *depth-first search* beserta fungsi-fungsi pendukung dalam tujuan untuk menyelesaikan permasalahan pencarian harta karun pada sebuah peta, dimana simpul/node mencerminkan jalur yang dapat dilalui. Pengembangan aplikasi dekstop dibuat melalui Visual Studio dengan bahasa pemrograman C# dengan pengembangan GUI menggunakan WinForm.

Dengan demikian, penulis menyimpulkan bahwa melalui Tugas Besar II IF2211 Strategi Algoritma ini, terbukti bahwa algoritma DFS dan BFS mampu menyelesaikan permasalahan pencarian rute harta karun pada sebuah peta, baik melalui penelusuran yang menyamping pada BFS maupun penelusuran mendalam yang berfokus untuk terus menelusuri anak dari suatu simpul pada DFS.

5.2 Saran

Tugas Besar 2 IF2211 Strategi Algoritma Semester II Tahun 2022/2023 menjadi salah satu tugas yang memberikan pelajaran baru bagi penulis. Berdasarkan pengalaman penulis mengerjakan tugas ini, berikut merupakan saran untuk pembaca yang ingin melakukan atau mengerjakan hal yang serupa.

1. Keefektifan dalam kerja sama tim merupajan hal yang penting dalam mengerjakan tugas ini. Tugas ini sangat terbantu oleh pemakaian *real-time collaboration app*. Selain itu, pemakaian aplikasi pengelola version control seperti Github sangat disarankan agar memudahkan untuk mengelola pekerjaan secara asinkron.
2. Terdapat keterbatasan dalam pengembangan aplikasi berbasis Bahasa C# di Sistem Operasi lain, karena itu pembaca sangat disarankan menggunakan Sistem Operasi Windows untuk pengembangan aplikasi.
3. Sinkronisasi versi merupakan hal yang penting dalam kolaborasi pengembangan pada Bahasa C#, sehingga pembaca sangat disarankan untuk menggunakan versi C# dan Visual Studio yang sama, lebih baik jika versi terbaru.
4. Bahasa C# merupakan bahasa yang *high-level* sehingga tidak terdapat *pointer*, yang dibutuhkan untuk penyimpanan objek secara dinamis, ditambah dengan keterbatasan *return value* yang akan selalu mengembalikan hasil *copy* dari objek yang di-*return*, maka pembaca disarankan menggunakan konsep *reference*.

5.3 Refleksi

Tugas Besar 2 IF2211 Strategi Algoritma Semester II Tahun 2022/2023 menjadi salah satu tugas yang memberikan pelajaran baru bagi penulis. Berdasarkan pengalaman penulis mengerjakan tugas ini, berikut merupakan saran untuk pembaca yang ingin melakukan atau mengerjakan hal yang serupa.

1. Keefektifan dalam kerja sama tim merupajan hal yang penting dalam mengerjakan tugas ini, mulai dari melakukan set-up hingga memfinalisasi program, terlebih karena program ditulis dengan menggunakan Visual Studio dan bahasa pemrograman yang jarang digunakan pada tubes lainnya yaitu C#.. pemakaian aplikasi pengelola version control seperti Github sangat disarankan agar memudahkan untuk mengelola pekerjaan secara asinkron.
2. Mengingat bahwa tugas ini mengharuskan pemrogram mampu memvisualisasikan hasil pencarian, maka penting untuk menata program secara rapih pada foldernya masing-masing untuk mempermudah integrasi program bagian satu sama lain.

5.4. Tanggapan Terkait Tugas Besar

Tugas Besar 2 IF2211 Strategi Algoritma Semester II Tahun 2022/2023 menjadi Tugas Besar yang cukup menantang bagi penulis, tak hanya dari algoritmanya, tetapi juga dari Bahasa baru yang dipelajari, dan *developer environment* yang benar-benar baru. Secara keseluruhan, Tugas Besar ini tergolong menyenangkan bagi penulis karena dapat mengeksplorasi pengembangan algoritma pencarian BFS dan DFS, menerapkan konsep-konsep orientasi objek, mengeksplorasi kemungkinan-kemungkinan *edge case* dari pencarian rute yang dilakukan pada graf, bukan sebuah pohon seperti *string matching*, *children nodes*, dan juga men-*develop* sebuah aplikasi berbasis desktop menggunakan Bahasa C#. Tanggapan terakhir terkait Tugas Besar ini adalah Tugas Besar ini memiliki persoalan yang menarik, dan juga menantang penulis untuk lebih mendalami algoritma pencarian graf DFS dan BFS.

DAFTAR PUSTAKA

Graph Traversal. https://en.wikipedia.org/wiki/Graph_traversal

Depth-First Search https://en.wikipedia.org/wiki/Depth-first_search

Breadth-First Search https://en.wikipedia.org/wiki/Breadth-first_search

Munir, R. (2022). Breadth First Search (BFS) dan Depth First Search (DFS) (1). Retrieved from Homepage

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

Munir, R. (2022). Breadth First Search (BFS) dan Depth First Search (DFS) (2). Retrieved from Homepage

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

LAMPIRAN

Lampiran 1 Pranala Repositori Github

https://github.com/KenEzekiel/Tubes2_thehashslingingslasher

Lampiran 2 Pranala Video Youtube

https://youtu.be/LmmB5A8BeZo