

**LAPORAN TUGAS BESAR 2  
IF2211 STRATEGI ALGORITMA**

**PENGAPLIKASIAN ALGORITMA BFS DAN DFS DALAM  
MENYELESAIKAN PERSOALAN MAZE TREASURE HUNT**



**SEMESTER II TAHUN 2022/2023**

**DISUSUN OLEH**

**DANIEL EGIANT SITANGGANG 13521056  
FRANKIE HUANG 13521092  
I PUTU BAKTA HARI SUDEWA 13521150**

**PROGRAM STUDI TEKNIK INFORMATIKA  
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA  
INSTITUT TEKNOLOGI BANDUNG  
2022/2023**

# DAFTAR ISI

<b>DAFTAR ISI</b>	<b>1</b>
<b>BAB I DESKRIPSI TUGAS</b>	<b>3</b>
<b>BAB II LANDASAN TEORI</b>	<b>4</b>
2.1. Teori dasar	4
2.1.1. Algoritma BFS (Breadth First Search)	4
2.1.2. Algoritma DFS (Depth First Search)	5
2.1.3. Persoalan TSP (Traveling Salesman Problem)	5
2.2. C# Desktop Application Development	6
<b>BAB III PEMETAAN MASALAH</b>	<b>6</b>
3.1. Langkah Pemecahan Masalah	6
3.1.1. Algoritma BFS	7
3.1.2. Algoritma DFS	7
3.1.3. Algoritma Brute Force untuk menyelesaikan persoalan TSP	7
3.2. Pemetaan Persoalan	7
3.3. Ilustrasi Kasus	8
3.3.1. BFS	8
3.3.2. DFS	8
3.3.3. TSP	9
<b>BAB IV ANALISIS PEMECAHAN MASALAH</b>	<b>9</b>
4.1. Implementasi Program	9
4.1.1. BFS dan DFS	9
4.1.1.1. Kelas Pathfinder sebagai base class	9
4.1.1.2. Kelas DFS	9
4.1.1.3. Kelas BFS	11
4.1.2. Kelas TSP	13
4.2. Struktur Data yang Digunakan	17
4.2.1. Kelas Pathfinder	17
4.2.2. Kelas TSP	19
4.3. Langkah Penggunaan Program	20
4.4. Pengujian Persoalan	23
4.4.1. sample-1.txt	23
4.4.2. sample-2.txt	25
4.4.3. sample-3.txt	26
4.4.4. sample-4.txt	27
4.4.5. sample-5.txt	28
4.5. Analisis Desain Solusi Algoritma	30
<b>BAB V KESIMPULAN DAN SARAN</b>	<b>31</b>
5.1. Kesimpulan	31

5.2. Saran	31
5.3. Refleksi	31
5.4. Tanggapan	31
<b>DAFTAR PUSTAKA</b>	<b>32</b>
<b>PRANALA</b>	<b>32</b>

## **BAB I DESKRIPSI TUGAS**

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

Dengan memanfaatkan algoritma Breadth First Search (BFS) dan Depth First Search (DFS), anda dapat menelusuri grid (simpul) yang mungkin dikunjungi hingga ditemukan rute solusi, baik secara melebar ataupun mendalam bergantung alternatif algoritma yang dipilih. Rute solusi adalah rute yang memperoleh seluruh treasure pada maze. Perhatikan bahwa rute yang diperoleh dengan algoritma BFS dan DFS dapat berbeda, dan banyak langkah yang dibutuhkan pun menjadi berbeda. Prioritas arah simpul yang dibangkitkan dibebaskan asalkan ditulis di laporan ataupun readme, semisal LRUD (left right up down). Tidak ada pergerakan secara diagonal. Anda juga diminta untuk memvisualisasikan input txt tersebut menjadi suatu grid maze serta hasil pencarian rute solusinya. Cara visualisasi grid dibebaskan, sebagai contoh dalam bentuk matriks yang ditampilkan dalam GUI dengan keterangan berupa teks atau warna. Pemilihan warna dan maknanya dibebaskan ke masing - masing kelompok, asalkan dijelaskan di readme / laporan.

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

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

## BAB II LANDASAN TEORI

### 2.1. Teori dasar

#### 2.1.1. Algoritma BFS (Breadth First Search)

BFS atau Breadth-First Search adalah salah satu algoritma pencarian dalam graf. Algoritma BFS efektif untuk menemukan jalur terpendek dari simpul awal ke simpul akhir. Algoritma ini bersifat *vertex-based* karena algoritma ini melakukan pencarian dengan mempertimbangkan setiap simpul / *vertex* secara langsung, yaitu dengan menjelajahi semua simpul yang berdekatan pada satu level sebelum beralih ke level selanjutnya. Secara umum, algoritma ini menggunakan struktur data *Queue* untuk menyimpan simpul mana yang hendak dikunjungi berikutnya.

Kompleksitas dari algoritma BFS adalah  $O(V + E)$  dengan  $V$  adalah banyaknya simpul dan  $E$  adalah banyaknya sisi. Berikut pseudocode dari algoritma BFS

```
procedure BFS(input v:integer)
{ Traversal graf dengan algoritma pencarian BFS.

  Masukan: v adalah simpul awal kunjungan
  Keluaran: semua simpul yang dikunjungi dicetak ke layar
}
Deklarasi
  w : integer
  q : antrian;

  procedure BuatAntrian(input/output q : antrian)
  { membuat antrian kosong, kepala(q) diisi 0 }

  procedure MasukAntrian(input/output q:antrian, input v:integer)
  { memasukkan v ke dalam antrian q pada posisi belakang }

  procedure HapusAntrian(input/output q:antrian,output v:integer)
  { menghapus v dari kepala antrian q }

  function AntrianKosong(input q:antrian) → boolean
  { true jika antrian q kosong, false jika sebaliknya }

Algoritma:
  BuatAntrian(q)          { buat antrian kosong }

  write(v)                 { cetak simpul awal yang dikunjungi }
  dikunjungi[v]←true      { simpul v telah dikunjungi, tandai dengan
                           true}
  MasukAntrian(q,v)        { masukkan simpul awal kunjungan ke dalam
                           antrian}

  { kunjungi semua simpul graf selama antrian belum kosong }
  while not AntrianKosong(q) do
    HapusAntrian(q,v)      { simpul v telah dikunjungi, hapus dari
                           antrian }
    for tiap simpul w yang bertetangga dengan simpul v do
      if not dikunjungi[w] then
        write(w)           {cetak simpul yang dikunjungi}
        MasukAntrian(q,w)
        dikunjungi[w]←true
      endif
    endfor
  endwhile
  { AntrianKosong(q) }
```

Gambar 1 Pseudocode Algoritma BFS

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

### 2.1.2. Algoritma DFS (Depth First Search)

Selain BFS, terdapat pula algoritma Depth-First Search atau DFS. Algoritma ini bersifat *edge-based* karena algoritma ini melakukan pencarian dengan mempertimbangkan setiap edge secara langsung. Dalam DFS, algoritma mencari terlebih dahulu satu simpul, kemudian mengikuti satu *edge* ke simpul berikutnya. Proses ini diulangi hingga tidak ada lagi edge yang dapat diikuti dan kemudian kembali ke simpul sebelumnya untuk mengeksplorasi edge lainnya. Versi iteratif dari algoritma DFS dapat menggunakan struktur data *Stack* untuk menyimpan simpul yang hendak dikunjungi berikutnya.

Kompleksitas dari algoritma DFS adalah  $O(V + E)$  dengan  $V$  adalah banyaknya simpul dan  $E$  adalah banyaknya sisi. Berikut pseudocode dari algoritma DFS

```
procedure DFS(input v:integer)
{Mengunjungi seluruh simpul graf dengan algoritma pencarian DFS}

Masukan: v adalah simpul awal kunjungan
Keluaran: semua simpul yang dikunjungi ditulis ke layar
}
Deklarasi
  w : integer

Algoritma:
  write(v)
  dikunjungi[v] ← true
  for w ← 1 to n do
    if A[v,w]=1 then {simpul v dan simpul w bertetangga }
      if not dikunjungi[w] then
        DFS(w)
      endif
    endif
  endfor
```

Gambar 2 Pseudocode Algoritma DFS

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

### 2.1.3. Persoalan TSP (Traveling Salesman Problem)

Persoalan TSP adalah sebuah permasalahan yang berusaha untuk menjawab pertanyaan berikut: “Diberikan  $n$  buah kota serta diketahui jarak antara setiap kota satu sama lain. Temukan perjalanan (*tour*) dengan jarak **terpendek** yang dilakukan oleh seorang pedagang sehingga ia melalui setiap kota tepat hanya sekali dan kembali lagi ke kota asal keberangkatan.”

Persoalan TSP tidak lain adalah persoalan untuk menemukan sirkuit Hamilton dengan bobot / jarak minimum. Dengan algoritma *brute-force*, persoalan ini dapat diselesaikan dengan kompleksitas  $O(n!)$ .

## 2.2. C# Desktop Application Development

*Desktop Application Development* atau pengembangan aplikasi desktop yang berbasis C# sangat mudah dijumpai saat ini. Untuk pengembangan aplikasi desktop C# memiliki beragam *framework* populer yang dapat dipilih untuk digunakan. Salah satu *framework* untuk membuat aplikasi desktop berbasis C# adalah WPF (*Windows Presentation Foundation*). Salah satu aplikasi yang dibuat dengan menggunakan WPF adalah Visual Studio dari Microsoft. WPF memiliki arsitektur yang cukup mudah dipahami oleh pemula, mulai dari UI yang berbasis XAML sebagai bahasa *markupnya* dan *design pattern* MVVM (*Model-View-View Model*) yang memudahkan aliran data serta interaktivitas antar komponen yang lebih dinamis.

C# sendiri adalah bahasa pemrograman berorientasi objek, yang berarti bahwa kode terorganisir ke dalam objek atau kelas yang memiliki sifat dan perilaku yang didefinisikan secara terpisah. C# juga menggunakan sintaks yang mirip dengan bahasa C++, sehingga mudah dipelajari oleh pemrogram yang sudah terbiasa dengan bahasa tersebut.

C# dijalankan pada lingkungan .NET Framework, yang menyediakan banyak fitur untuk mempermudah pengembangan aplikasi, termasuk pengelolaan memori yang otomatis, penyederhanaan tugas I/O, dan dukungan untuk bahasa pemrograman lain seperti Visual Basic dan F#.

## BAB III PEMETAAN MASALAH

### 3.1. Langkah Pemecahan Masalah

Perancangan solusi untuk menyelesaikan permasalahan pada tugas besar kali ini diawali dengan pembagian dua masalah utama yaitu pencarian langkah solusi peta dan menampilkan solusi persoalan tersebut dalam bentuk aplikasi desktop. Dalam menyelesaikan persoalan pencarian langkah solusi peta, ada tiga pendekatan/algorithm yang kami implementasikan yaitu breadth-first search, depth-first search, dan brute-force khusus untuk persoalan TSP.

Setelah menentukan permasalahan tersebut, perlu dirancang struktur data yang tepat untuk mengimplementasikan algoritma pencarian, yaitu BFS dan DFS. Dari kekhasan pada struktur data yang umum digunakan pada algoritma DFS dan BFS kemudian dirancang sebuah class sebagai abstraksi dari struktur data yang digunakan pada algoritma BFS dan DFS. Hasil abstraksi berupa kelas PathFinder ini dapat dilihat pada 4.1.1.1, 4.2.1, dan source-code program.

Langkah terakhir adalah mengimplementasikan algoritma pencarian solusi dengan DFS, BFS, dan brute-force. Berikut implementasinya.

### 3.1.1. Algoritma BFS

Algoritma BFS diawali dengan menginisialisasi sebuah queue lalu meng-enqueue titik awal pencarian solusi. Kemudian dimulai iterasi yang berakhir pada saat queue telah kosong. Langkah-langkah dalam iterasi tersebut adalah melakukan dequeue pada queue tersebut, lalu memproses point yang dikeluarkan dari queue tersebut apakah merupakan treasure atau tidak. Selanjutnya tiap simpul yang bertetangga dengan point tersebut akan di-enqueue ke dalam queue dengan prioritas UDLR dan iterasi dilanjutkan.

### 3.1.2. Algoritma DFS

Algoritma DFS diawali dengan menginisialisasi sebuah stack lalu melakukan operasi push titik awal pencarian solusi. Stack kemudian secara iteratif melakukan pop tersebut sampai stack kosong. Pada setiap pemrosesan point yang di pop, akan dicek apakah titik tersebut merupakan treasure atau tidak. Selanjutnya, simpul yang bertetangga dengan point tersebut akan di push ke dalam stack dengan prioritas UDLR dan iterasi akan dilanjutkan hingga stack kosong/berhenti.

### 3.1.3. Algoritma *Brute Force* untuk menyelesaikan persoalan TSP

Mirip seperti algoritma BFS, algoritma *brute-force* TSP yang digunakan adalah dengan membuat graf berbobot penuh yang memiliki bobot jarak antara kedua titik T dan K. Jarak ini dapat dihitung dengan melakukan BFS pada titik asal hingga semua T terkunjungi. Jarak-jarak ini kemudian dimasukkan ke dalam sebuah matriks adjacency yang merupakan representasi dari graf berbobot penuh tersebut.

Setelah jarak antar semua titik telah didapatkan, kemudian akan dicari algoritma *brute-force* untuk mencari permutasi titik-titik graf yang akan memberikan jarak terpendek. Hasil pencarian akan disimpan dan perhitungan akan berjalan hingga semua permutasi yang berawal pada K terjalani, sehingga akan dipastikan didapatkan jarak yang terpendek.

## 3.2. Pemetaan Persoalan

Pada program, persoalan akan dimasukkan dengan membuka sebuah file yang berisi format peta yang sudah terdefinisi. Program kemudian akan membaca peta secara iteratif hingga EOF dan kemudian membaca nilai karakter yang ada dipisahkan dengan spasi. Setelah itu, program akan membentuk sebuah matriks yang merupakan hasil pembacaan dari file tersebut, dimana setiap elemen bertipe char dan setiap char berkorespondensi dengan hasil pembacaan pada file tersebut.

Hasil pembacaan kemudian akan disimpan pada sebuah matriks yang berisi element of string. Matriks inilah yang nantinya akan menjadi representasi dari “graf”, dimana koordinat



yang valid adalah kotak yang ditandai dengan huruf K, T, atau R; sedangkan koordinat yang tidak valid ditandai dengan kotak dengan huruf X.

### 3.3. Ilustrasi Kasus

Untuk ilustrasi kasus, akan diambil contoh kasus sebagai berikut

	0	1	2
0	K	R	R
1	R	X	R
2	R	R	T

#### 3.3.1. BFS

Head	Tail
(0, 0)	(0, 1)   (1, 0)
(0, 1)	(1, 0)   (0, 2)
(1, 0)	(0, 2)   (2, 0)
(0, 2)	(2, 0)   (1, 2)
(2, 0)	(1, 2)   (2, 1)
(1, 2)	(2, 1)   (2, 2)
(2, 1)	(2, 2)
(2, 2)	Treasure ditemukan, pencarian berhenti

#### 3.3.2. DFS

Head	Body
(0, 0)	(0, 1)   (1, 0)
(0, 1)	(0, 2)   (1, 0)
(0, 2)	(1, 2)   (1, 0)
(1, 2)	(2, 2)   (1, 0)
(2, 2)	Treasure ditemukan, pencarian berhenti

### 3.3.3. TSP

Pada algoritma TSP, titik K dan T akan diubah menjadi sebuah matriks adjacency yang akan berbentuk

	0	1
0	0	4
1	4	0

Kemudian, kita akan mencari permutasi titik yang akan menghasilkan jarak paling minimal. Karena jumlah titik hanya ada dua, maka dapat langsung dipecahkan bahwa jarak terpendek yang bisa dicapai adalah titik K-T.

## BAB IV ANALISIS PEMECAHAN MASALAH

### 4.1. Implementasi Program

#### 4.1.1. BFS dan DFS

##### 4.1.1.1. Kelas Pathfinder sebagai *base class*

```
class Pathfinder
protected List<List<string>> map
protected bool[,] visited
protected List<Point> solution
protected int numberOfTreasureAvail
protected Point startPoint
protected bool tsp
protected List<Point> trace
endclass
```

##### 4.1.1.2. Kelas DFS

```
class DFS : Pathfinder
function visit(Point point, int numberOfTreasureFound) -> bool
{ mengunjungi Point point dan mengembalikan True jika point
merupakan bagian dari himpunan solusi, false jika tidak }
KAMUS
anyTreasureFoundHere : boolean
ALGORITMA
{menginisialisasikan boolean anyTreasureFound sebagai flag
apakah node ini merupakan himpunan bagian dari solusi yaitu
jika merupakan treasure atau merupakan langkah menuju
treasure}
bool anyTreasureFound <- false

{menambahkan point ke trace agar kemudian dapat
divisualisasikan langkah pencarian dengan algoritma dfs}
trace.Add(point)

{menandai point sudah dikunjungi}
remember(point)

{memasukkan point ke himpunan solusi}
solution.Add(point)
```

```

        {jika point merupakan treasure}
        if (isTreasure(point)) then
            numberOfTreasureFound <- numberOfTreasureFound + 1
            anyTreasureFound <- True
        endif

        {jika sudah mencapai goal maka tidak perlu melanjutkan
        pencarian, langsung return}
        if (numberOfTreasureFound = numberOfTreasureAvail) then
            -> true
        endif

        {mencoba mengunjungi seluruh simpul yang bertetangga
        dengan point}
        Point[] directions <- [Point(0, 1), Point(0, -1), Point(-1,
        0), Point(1, 0)]
        for each (var dir in directions) do
            {inisialisasi point next, yaitu setiap simpul yang
            bertetangga dengan point}
            Point next <- Point(point.X + dir.X, point.Y + dir.Y)

            {jika point next valid, belum dikunjungi, dan bukan
            merupakan BLOCK, maka dikunjungi}
            if (isIdxValid(next) and not isVisited(next) and not
            isBlock(next))
                {mengunjungi simpul next}
                anyTreasureFoundHere <- visit(next, ref
                numberOfTreasureFound)

                if (numberOfTreasureFound = numberOfTreasureAvail)
                    {jika saat kembali sudah mencapai goal dan harus
                    kembali ke titik asal maka point dimasukkan ke himpunan solusi
                    (pulang)}

                    if tsp then
                        solution.Add(point)
                        {jika tidak perlu kembali ke titik asal,
                        langsung return}
                        -> true;
                    endif
                endif
                {jika point next merupakan himpunan bagian dari
                himpunan solusi, maka point dimasukkan kembali ke himpunan solusi
                (berjalan mundur)}
                if anyTreasureFoundHere then
                    anyTreasureFound <- true;
                    solution.Add(point)
                endif
            endif
        endfor

        {jika ternyata tidak terdapat treasure pada point dan point
        bukan merupakan langkah menuju treaure,
        point di dikeluarkan dari himpunan solusi}
        if not anyTreasureFound then
            solution.RemoveLast()
        endif

        -> anyTreasureFound

    endfunction

```

```

function findPathDFS(bool) -> (List<string>, string)
{ Mengembalikan urutan langkah pencarian secara DFS dan solusi
persoalan berupa string }
KAMUS
    numberOfTreasureFound: int
ALGORITMA
    {jika tidak ada treasure yang tersedia, maka solusi adalah
himpunan kosong}
    if (numberOfTreasureAvail = 0) then
        -> (trace, "");
    endif

    {inisialisasi banyaknya treasure yang sudah ditemukan}
    numberOfTreasureFound <- 0;

    {mengunjungi point awal yaitu krusty-krab dengan algoritma
bfs}
    visit(startPoint, numberOfTreasureFound);

    {jika setelah pencariin secara dfs solusi dicapai,
mengembalikan trace dan string solusi}
    if (numberOfTreasureFound = numberOfTreasureAvail) then
        -> (trace , solution)

    {jika tidak mengembalikan trace dan string kosong}
    else
        -> return (trace, "")
    endif
end function

endclass

```

#### 4.1.1.3.Kelas BFS

```

class BFS : Pathfinder
    procedure enqueueNeighbour(input currentPath: list of string ,
input/output: queue: queue of point)
    { Mengenqueue seluruh vertex yang bertetangga dengan
currentPoint,
    I.S. -
    F.S. seluruh vertex yang bertetangga dengan point sudah
di enqueue
    }
    KAMUS
        temp : list of string
    ALGORITMA
        {mencoba meng-enqueue seluruh simpul yang bertetangga
dengan point}
        directions <- [Point(0, 1), Point(0, -1), Point(-1, 0),
Point(1, 0)]
        for each (var dir in directions) do
            {inisialisasi point next, yaitu setiap simpul yang
bertetangga dengan point}
            next <- Point(point.X + dir.X, point.Y + dir.Y)
            {jika point next valid, belum dikunjungi, dan bukan
merupakan BLOCK, maka dikunjungi}
            if (isIdxValid(next) and not isVisited(next) and not
isBlock(next))
                {menyalin path dari point dan menambahkan next
ke path tersebut, lalu path di enqueue}

```

```

        temp <- currentPath
        temp.Add(point)
        queue.Enqueue(temp)
        remember(path)
    endif
end for
endprocedure

function findPathBFS(bool tsp = false) -> (List<string>, string)
    { Mengembalikan urutan langkah pencarian secara BFS dan
    solusi persoalan berupa string }
    KAMUS
        queue : queue of list of string
        temp : list of string

    ALGORITMA :
        doneTsp <- false;

        {jika tidak ada treasure yang tersedia, maka solusi
        adalah himpunan kosong}
        if (numberOfTreasureAvail = 0)
            -> (trace, "")
        endif
        {inisialisasi banyaknya treasure yang sudah ditemukan
        yaitu 0, dan queue kosong}
        queue <- []
        numberOfTreasureFound <- 0

        {mengenqueue point awal yaitu krusty-krab ke dalam
        queue}
        remember(startPoint)
        temp <- [startPoint]
        queue.Enqueue(temp)

        {iterasi hingga queue kosong yaitu hingga seluruh vertex
        yang terhubung dengan vertex awal sudah dikunjungi}
        while (queue.Count != 0) do
            {mendequeue point dari queue dan dikunjungi}
            currentPath <- queue.Dequeue()
            currentPoint <- currentPath.Last()
            trace.Add(currentPoint)

            if (isTreasure(currentPoint)) do
                numberOfTreasureFound++;

                if (numberOfTreasureFound >=
                _numberOfTreasureAvail)
                    {jika goal sudah tercapai dan tidak
                    perlu kembali ke titik awal atau sudah dari titik awal, maka return
                    solusi}

                    if (not tsp or doneTsp) do
                        -> (trace, currentPath)
                    else
                        {jika goal sudah tercapai namun harus
                        kembali ke titik awal, menjadikan titik awal sebagai treasure agar
                        menjadi goal untuk dikunjungi}
                        map[startPoint.Y][startPoint.X] =
                        TREASURE;

                        doneTsp <- true;
                    endif
                endif
            endif
        endwhile
    endwhile
endfunction

```

```

                {jika currentPoint adalah treasure, maka di set
sebagai path biasa}
                map[currentPoint.Y][currentPoint.X] <- PATH

                {melupakan seluruh simpul yang sudah dikunjungi,
dan mengosongkan queue}
                forgetAll()
                queue.Clear()
                remember(currentPoint)

            endif

            {mencoba enqueue neighbour dari currentPoint}
            enqueueNeighbour(currentPath, ref queue)
        endwhile

        {jika sudah keluar dari while-loop tanpa mereturn
solusi, berarti tidak ada solusi yang ditemukan}
        throw SolutionNotFoundException
    endfunction
endclass

```

#### 4.1.2. Kelas TSP

```

class TravellingSalesman

    KAMUS
    { kelas yang digunakan untuk mencatat titik }
    class Pair
        KAMUS
        y : koordinat titik y
        x : koordinat titik x

        procedure Pair(y: int, x: int)
            this.y <- y
            this.x <- x
        endprocedure
    endclass

    { kelas yang digunakan untuk mencatat jalur antar treasure/krusty
krab }
    class Path
        KAMUS
        point: titik pada koordinat
        path: string berisi jalur yang diambil
        distance: jarak antara titik asal dan titik tujuan

        procedure Path(point: Pair, path: string, distance: int)
            this.point <- point
            this.path <- path
            this.distance <- distance
        endprocedure

        procedure Path(point: Pair, path: string)
            this.point <- point
            this.path <- path
        endprocedure
    endclass

    direction : Path

    height : int
    width : int
endclass

```

```

amount: int

visited : [1...height] array of [1...width] array of boolean

index : [1...amount] array of Pair

distance : [1...amount] array of [1...amount] array of int

stringPaths : [1...amount] array of [1...amount] array of string

permutation : [1...amount!] array of [1...amount] array of int

ALGORITMA
{ Mereset visited array sehingga semua titik belum dikunjungi }
procedure refreshVisited(map: array of array of string)
    ALGORITMA
        for i in [0...height] do
            for j in [0...width] do
                if (map[i][j] = "X") then
                    visited[i][j] = true
                else
                    visited[i][j] = false

{ Check if index is visitable in map }
function isValidIndex(row: int, col: int) -> boolean
    ALGORITMA
        if (row < 0 || col < 0) then
            -> false
        if (row >= height || col >= width) then
            -> false
        if (visited[i][j] = true) then
            -> false
        -> true

{ Ubah matrix menjadi adjacency matrix (berisi jarak antar treasure) }
function transformToGraph(map: array of array of string, start: Pair) -> array of string
    KAMUS

        stringPath : array of string
        q : queue

        startY : int
        startX : int

        startIndex : int

        p: Path

        y : int
        x : int
        path : string
        distance : int

        adjY : int
        adjX : int
        direction : string

    ALGORITMA
        q.enqueue(new Path(start, "", 0))

```

```

startY <- start.y
startX <- start.x

visited[startY, startX] <- true

{ Cari index dimana "start" berada di array index }
startIndex <- 0
for i in [1...amount] do
    if (index[i].y = startY and index[i].x = startX) then
        startIndex <- 0

{ Loop hingga queue habis }
while (q.Count > 0) do
    p <- q.Dequeue()

    y <- p.point.y
    x <- p.point.x
    path <- p.path
    distance <- p.distance

    { Jika berada pada titik start atau treasure, catat
jarak dan simpan string jalur }
    if (map[y][x] = "K" || map[y][x] = "T") then
        for i in [0...amount] do
            if (index[i].y = y and index[i].x = x) then
                distance[i, startIndex] <- distance
                distance[startIndex, i] <- distance

                stringPath[i] <- path

                break

    { Loop sebanyak semua arah }
    for i in [0...4] do
        adjY <- y + direction[i].point.y
        adjX <- x + direction[i].point.x
        direction <- direction[i].path

        if (isValidIndex(adjY, adjX)) then
            q.Enqueue(new Path(new Pair(adjY, adjX), path
+ direction, distance + 1))
            visited[adjY, adjX] <- true

-> stringPath

{ Simpan semua permutasi bilangan dari 0 hingga n-1 }
procedure generatePermutation(numbers: array of int, size: int)
    KAMUS
        temp : array of int

    ALGORITMA
        if (size = 1) then
            for i in [i...amount] do
                temp.Add(numbers[i])

            permutation.Add(temp)

        for i in [i...size] do
            generatePermutation(numbers, size - 1)

        if (size % 2 = 1) then

```



```

        numbers[0], numbers[size - 1] <- numbers[size-1],
numbers[0]
        else
            numbers[i], numbers[size - 1] <- numbers[size -
1], numbers[i]

    { Hitung jarak dari semua bilangan pada array numbers secara
berurutan }
    function calculateDistance(numbers: array of int) -> int
        KAMUS
            dist : int

        ALGORITMA
            dist <- 0

            for i in [0...amount-1] do
                distance <- distance +
distance[numbers[i]][numbers[i+1]]

            dist <- dist + distance[numbers[amount - 1]][numbers[0]]

    { Fungsi utama }
    function doTSP(map: array of array of string) -> string
        KAMUS
            len : int

            numbers : array of int

            shortestIndex : int

            finalPath : string

        ALGORITMA
            height <- map.Count
            width <- map.Count

            { Hitung jumlah K dan T }
            amount <- 0
            for i in [0...height] do
                for j in [0...width] do
                    if (map[i][j] = "K" or map[i][j] = "T") then
                        amount <- amount + 1

            { Catat koordinat K dan T }
            len <- 1
            for i in [0...height] do
                for j in [0...width] do
                    if (map[i][j] = "K") then
                        index[0] <- new Pair(i, j)
                    if (map[i][j] = "T") then
                        index[len] <- new Pair(i, j)
                        len <- len + 1

            { Hitung jarak antara K dan T membentuk graf penuh }
            for i in [0...amount] do
                refreshVisited(map)
                stringPaths.Add(transformToGraph(map, index[i]))

            { Simpan semua permutasi }
            for i in [1...amount] do
                numbers.Add(i)

```

```

generatePermutation(numbers, numbers.Count)

{ Cari semua permutasi, lalu simpan nilai terpendek }
shortestIndex <- 0
for i in [1...amount!] do
  if (permutation[i][0] != 0) then
    continue

    if (calculateDistance(permutation[i]) <
calculateDistance(permutation[shortestIndex])) then
      shortestIndex <- i

{ Gabung semua string, lalu return fungsi }
finalPath <- ""
for i in [0...amount-1] do
  finalPath <- finalPath +
stringPaths[permutation[shortestIndex][i]][permutation[shortestIndex][i+1]
]

  finalPath <- finalPath +
stringPaths[permutation[shortestIndex][amount-1]][permutation[shortestIndex][0]]

-> finalPath

```

## 4.2. Struktur Data yang Digunakan

### 4.2.1. Kelas Pathfinder

Kelas Pathfinder adalah kelas yang digunakan untuk menyimpan data yang dibutuhkan selama proses PathFinding. Kelas PathFinding menjadi base class bagi kelas DFS dan BFS yang kemudian kelas-kelas tersebut akan mengimplementasikan path-findingnya masing-masing. Kelas Pathfinder adalah struktur data komposit yang terdiri dari list of point, list of list of string, 2d array of boolean, integer, point, dan boolean. Berikut adalah implementasi dari kelas Pathfinder

Nama Field	Struktur Data	Keterangan
_solution	List<Point>	<p>List of point merepresentasikan path/jalur solusi dari peta persoalan, terdapat beberapa alternatif untuk merepresentasikan path dari peta persoalan yaitu Linked-List yang diimplementasikan secara terbalik (seperti tree) atau tree.</p> <p>Pada algoritma DFS, operasi yang umum dilakukan oleh path adalah add-last, dan delete-last (seperti push dan pop). Struktur data List&lt;Point&gt; dipilih karena</p>

		<p>kompleksitas dari add-last dan delete-last adalah <math>O(1)</math>, sedangkan pada struktur data baik Tree maupun Linked-List kompleksitas untuk melakukan add-last adalah <math>O(N)</math> dan kompleksitas untuk melakukan delete-last adalah <math>O(1)</math>, jika telah diketahui reference pada node akhirnya. Karena itu, dipilih struktur data List&lt;Point&gt; untuk merepresentasikan path pada algoritma DFS.</p> <p>Pada algoritma BFS, operasi yang umum dilakukan oleh path adalah copy dan add-last. Copy diperlukan saat me-requeue tiap node tetangga dari node yang sedang di proses, node tersebut harus memiliki salinan dari path yang telah dilalui node sebelumnya. Kompleksitas untuk melakukan copy pada struktur data List&lt;Point&gt; adalah <math>O(N)</math> dan kompleksitas untuk melakukan add-last adalah <math>O(1)</math>, sedangkan pada struktur data baik Tree ataupun Linked-List, kompleksitas untuk melakukan copy (as reference) adalah <math>O(1)</math> dan kompleksitas untuk melakukan insert-last adalah <math>O(N)</math>. Pada algoritma DFS di program ini, dipilih struktur data List&lt;Point&gt; untuk merepresentasikan path karena dirasa lebih mudah untuk diimplementasikan.</p>
<code>_map</code>	<code>List&lt;List&lt;string&gt;&gt;</code>	Matrix/2d list of string untuk menyimpan map/peta persoalan.
<code>_visited</code>	<code>bool[, ]</code>	Matrix/2d array of boolean yang menyimpan status dari tiap koordinat pada peta tersebut apakah sudah dikunjungi atau belum.

<code>_numberOfTreasureAvail</code>	Int	Integer untuk menyimpan banyaknya treasure yang ada pada peta.
<code>_startPoint</code>	Point	Point untuk menyimpan koordinat “Krusty-Krab” atau titik awal pada peta

#### 4.2.2. Kelas TSP

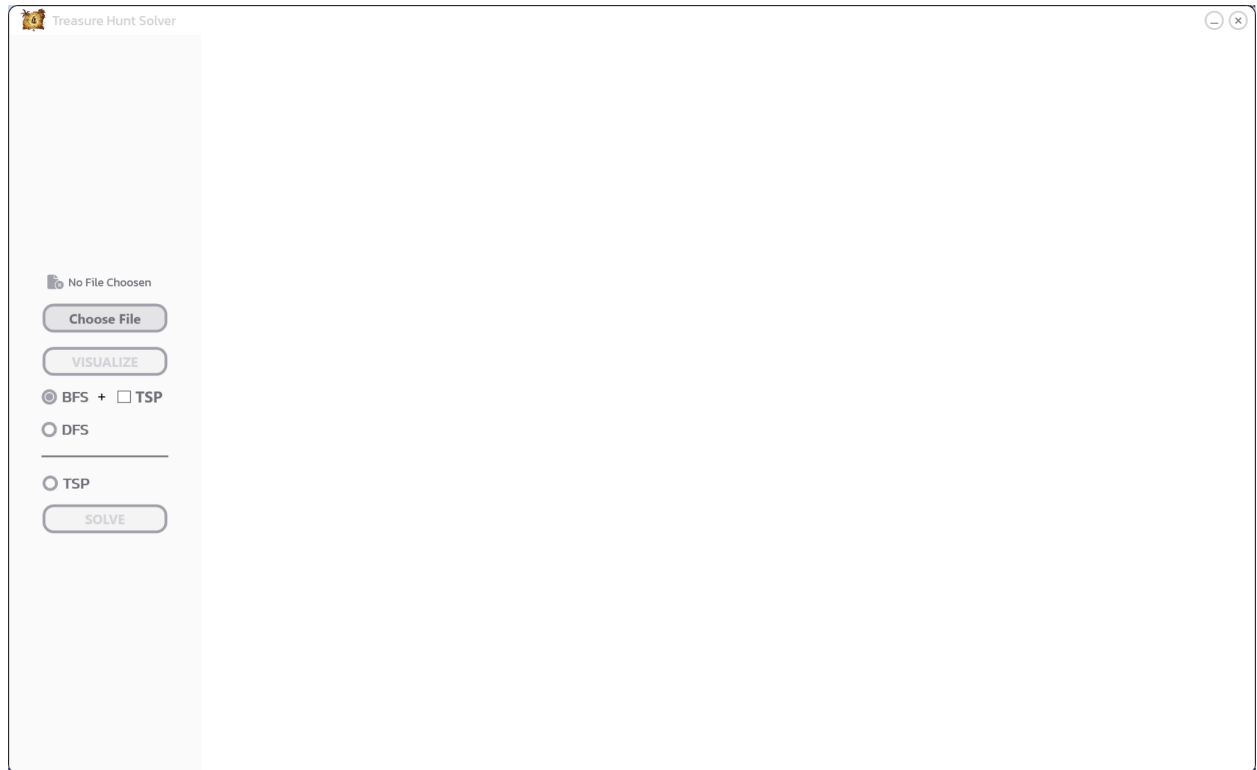
Kelas TSP adalah kelas yang digunakan untuk mencari jarak terpendek dari seluruh treasure dan kembali ke tempat semula. Kelas ini memiliki struktur data komposit yang terdiri dari Pair, Path, array of Pair, array of bool, queue of Path, list of list of string, dan list of list of int.

Nama Field	Struktur Data	Keterangan
<code>direction</code>	<code>Path[]</code>	Berisi langkah yang dapat diambil pada suatu titik tertentu ( <i>up, down, left, dan right</i> )
<code>visited</code>	<code>bool[, ]</code>	Berisi boolean apakah pasangan titik (i, j) sudah pernah dikunjungi
<code>index</code>	<code>Pair[]</code>	Berisi koordinat kotak yang bernilai “K” dan “T”
<code>distance</code>	<code>int[, ]</code>	Merupakan matriks adjacency yang berisi jarak dari treasure i ke treasure j
<code>stringPaths</code>	<code>List&lt;List&lt;String&gt;&gt;</code>	Berisi jalur yang ditempuh pada matriks agar sebuah kotak dapat dicapai dari suatu titik ke titik lainnya
<code>permutation</code>	<code>List&lt;List&lt;int&gt;&gt;</code>	Berisi permutasi nilai-nilai dari 0 hingga banyaknya “T” dan “K”
<code>stringPath</code>	<code>List&lt;String&gt;</code>	Berisi string yang merupakan kumpulan path yang diambil agar sampai ke suatu titik
<code>q</code>	<code>Queue&lt;Path&gt;</code>	Berisi nilai-nilai Path yang diambil dengan metode BFS
<code>p</code>	<code>Path</code>	Berisi koordinat beserta string path yang telah diambil saat ini
<code>map</code>	<code>List&lt;List&lt;Stri</code>	Berisi kumpulan karakter yang merupakan

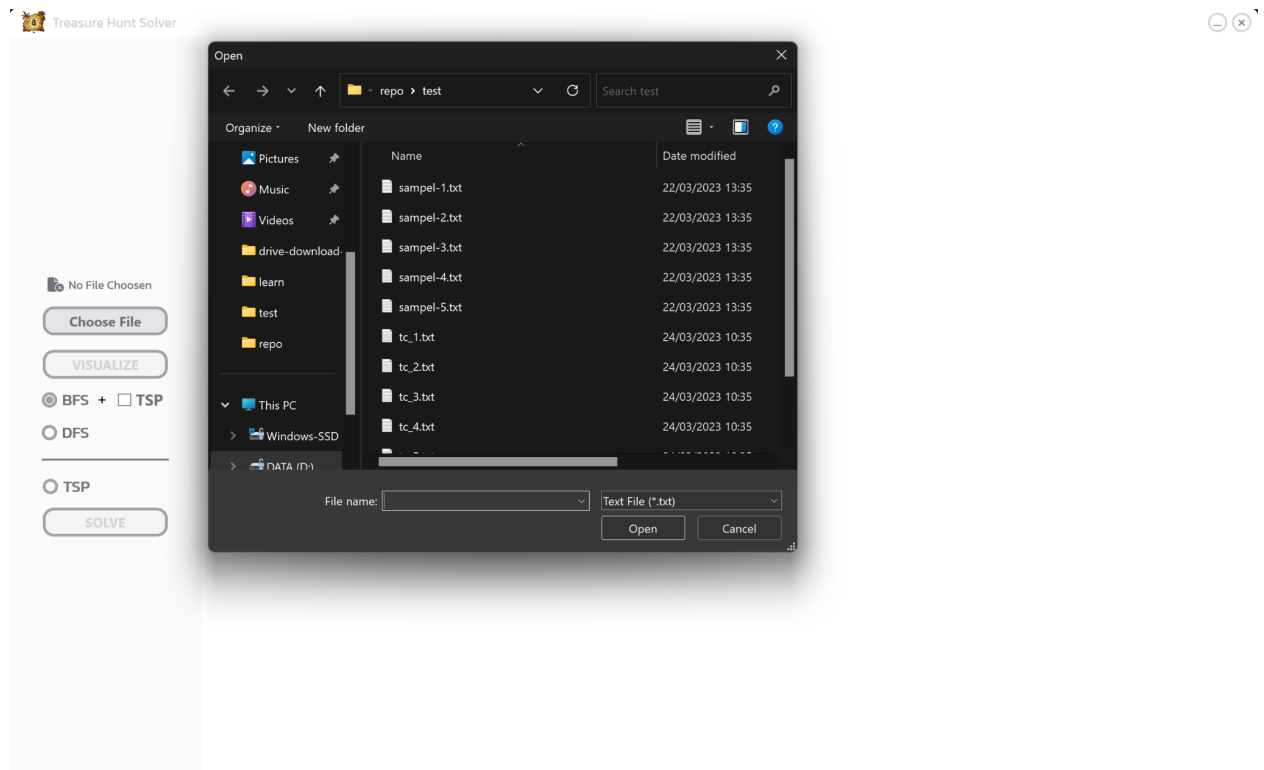
	ng>>	peta permainan
--	------	----------------

### 4.3. Langkah Penggunaan Program

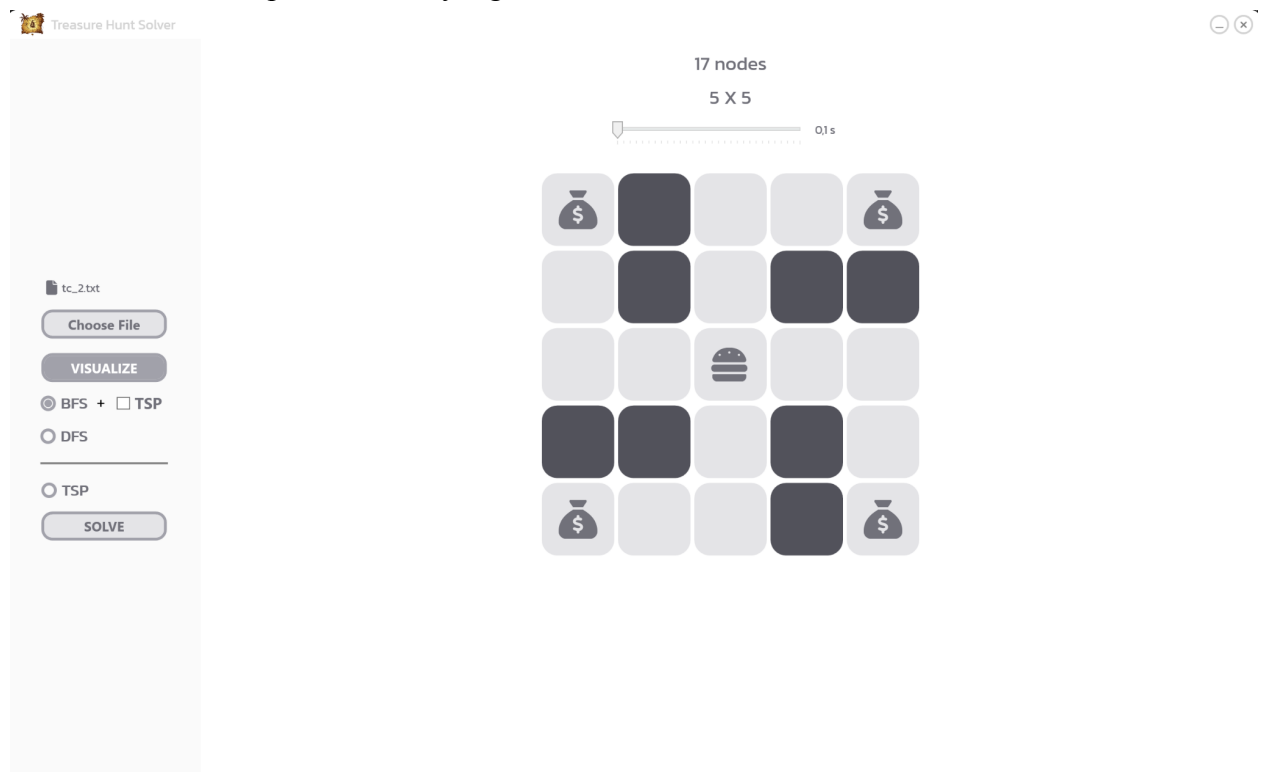
Buka program dengan menuju folder bin pada direktori.



Jika program dapat berjalan dengan sesuai maka akan terlihat tampilan seperti gambar di atas.



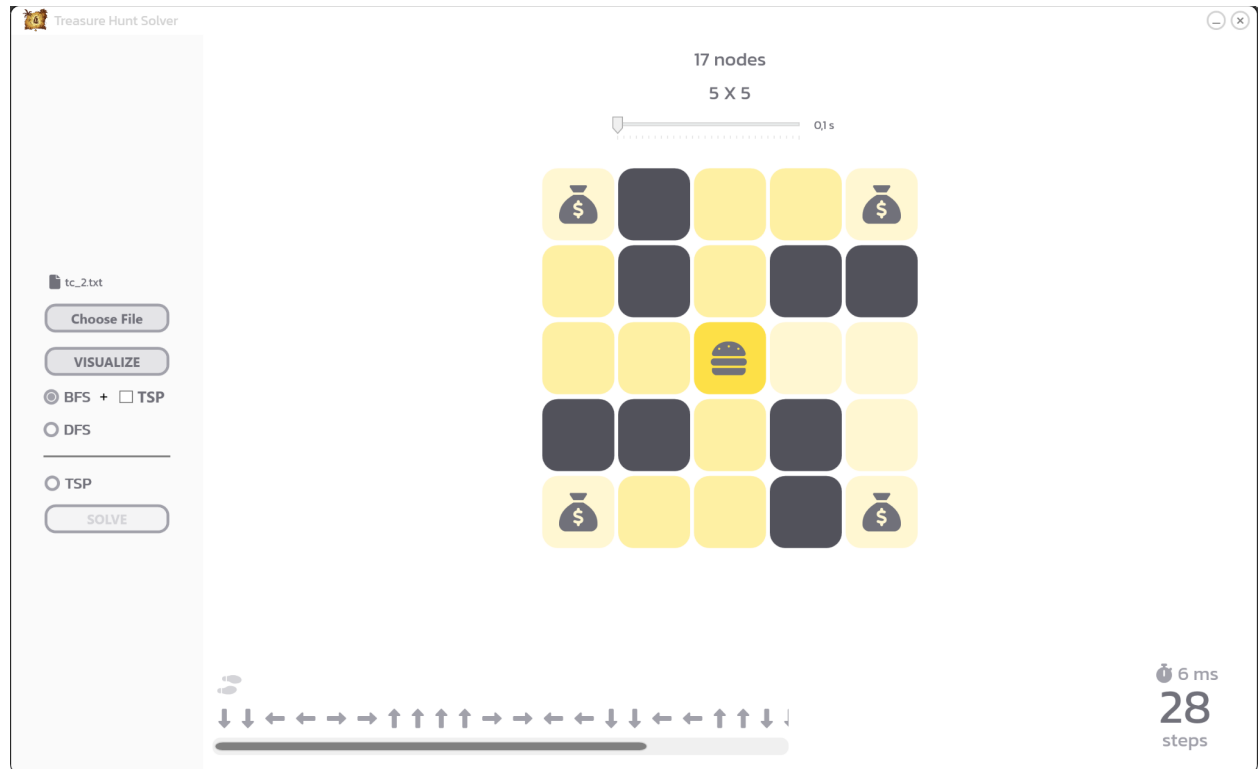
Berikutnya, Anda dapat memilih salah satu dari peta *treasure hunt* yang tersedia pada folder test direktori atau dapat membuat yang baru.



Setelah memilih file, peta *treasure hunt* dapat divisualisasikan dengan menekan tombol *VISUALIZE*.



Jika file yang dipilih mengandung peta yang tidak sesuai dengan format, maka tombol *VISUALIZE* akan *terdisable* sehingga visualisasi tidak dapat dilakukan. Anda harus memperbaiki kesalahan format atau memilih file lainnya.



Selanjutnya, Anda dapat memilih algoritma apa yang akan digunakan untuk menyelesaikan *treasure hunt* tersebut, kemudian tekan tombol *SOLVE* untuk melihat progres pencarian dan langkah serta waktu eksekusi yang diperlukan untuk menyelesaikan map tersebut. Anda dapat mereset map dengan menekan kembali tombol *VISUALIZE* setelah proses pencarian sebelumnya selesai atau menggunakan map yang baru.

## 4.4. Pengujian Persoalan

### 4.4.1. sample-1.txt

File Sample	
	X T X X
	X R R T
	K R X T
	X R X R
	X R R R

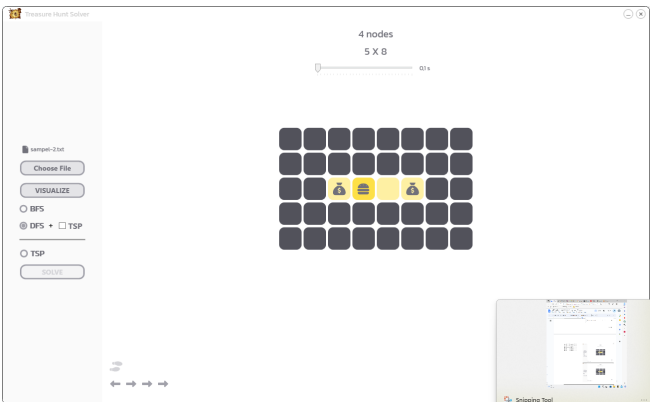


K e l u a r a n	BFS	
	DFS	

	<b>TSP</b>	

4.4.2. sample-2.txt

<b>File Sample</b>	<pre> X X X X X X X X X X X X X X X X X X T K R T X X X X X X X X X X X X X X X X X X </pre>	
<b>K e l u a r a n</b>	<b>BFS</b>	

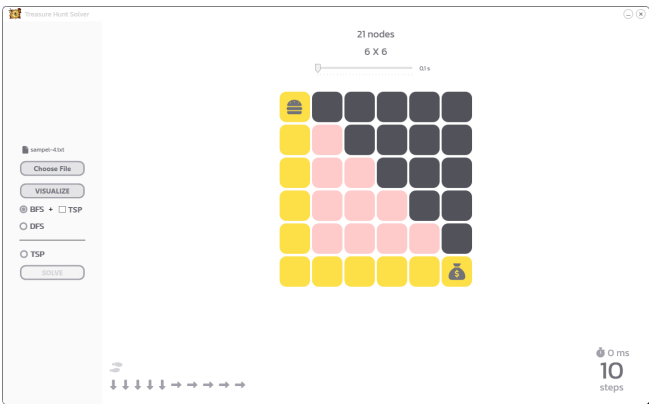
	<p><b>DFS</b></p>	 
	<p><b>TSP</b></p>	

4.4.3. sample-3.txt

<p><b>File Sample</b></p>	<p>J A N G A N L U P A C E K Y A N G B E G I N I Y</p>
---------------------------	--

<b>Keluaran</b>	
-----------------	--

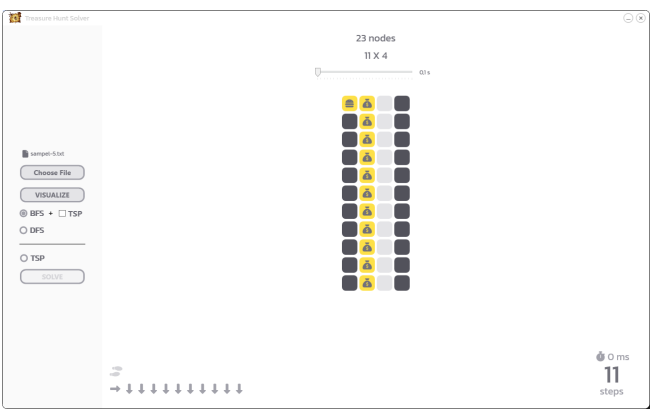
4.4.4. sample-4.txt

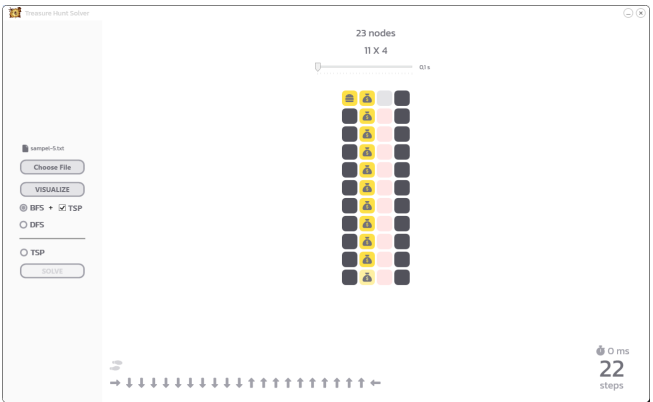
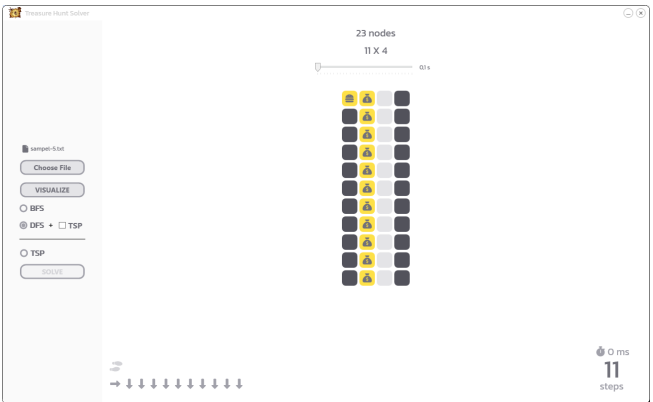
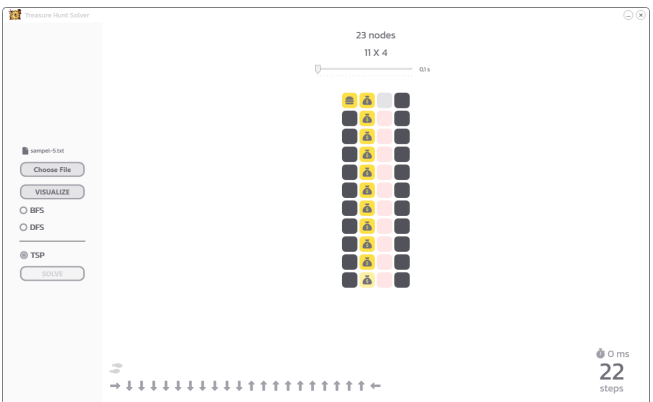
<b>File Sample</b>		<pre> K X X X X X R R X X X X R R R X X X R R R R X X R R R R R X R R R R R T </pre>
<b>K e l u a r a n</b>	<b>BFS</b>	

		
	<b>DFS</b>	 

	<b>TSP</b>	
--	------------	--

4.4.5. sample-5.txt

	<b>File Sample</b>	<pre> K T R X X T R X X T R X X T R X X T R X X T R X X T R X X T R X X T R X X T R X X T R X X T R X </pre>
<b>K e l u a r a n</b>	<b>BFS</b>	

		
	DFS	
	TSP	

#### 4.5. Analisis Desain Solusi Algoritma

Algoritma *breadth-first search* yang diimplementasikan efektif untuk mencari jarak terpendek dari Krusty-Krab menuju seluruh goal/treasure. Kompleksitas dari algoritma ini juga terbilang cukup mangkus yaitu  $O(n * goals)$  dengan n adalah banyaknya simpul pada peta dan goals adalah banyaknya treasure pada map atau banyaknya treasure + 1 jika harus kembali ke titik awal jika dibandingkan dengan kompleksitas brute-force yaitu  $O(n!)$ .

Namun, pada beberapa kasus, pada contoh ilustrasi kasus di *sub-bab* 3.3, algoritma *depth-first search* tetap dapat memperoleh solusi optimal dengan langkah pencarian yang lebih pendek.

Algoritma *depth-first search* yang diimplementasikan belum efektif untuk mencari jarak terpendek dari titik awal (Krusty-Krab) menuju semua goals/treasure. Hal ini dikarenakan DFS secara algoritma belum tentu akan mendapatkan jarak terpendek dari sebuah jalur, namun DFS mengurangi jumlah percabangan yang harus diperiksa. DFS sendiri memiliki kompleksitas algoritma  $O(n)$  dengan  $n$  adalah banyaknya simpul pada peta. Algoritma *depth-first search* dapat dibuat menjadi lebih efektif dalam menyelesaikan persoalan jika dibuat menjadi algoritma *iterative depth search*.

Algoritma *brute-force* untuk menyelesaikan persoalan TSP yang digunakan bisa dikatakan sangat buruk, karena memiliki kompleksitas program pada skala faktorial  $O(n!)$  dimana  $n$  adalah jumlah treasure. Pada test-case file sample-5.txt, diperlukan perangkat dengan kekuatan komputasi yang tinggi agar persoalan dapat diselesaikan, karena terdapat 11 buah treasure yang harus dipermutasi, sehingga membuat program yang dijalankan menjadi lebih lambat.

## BAB V KESIMPULAN DAN SARAN

### 5.1. Kesimpulan

Algoritma *breadth-first search* dan *depth-first Search* dapat digunakan untuk menyelesaikan permasalahan Maze Treasure Hunt secara efisien. Algoritma *breadth-first search* akan mencari koordinat-koordinat yang berjarak 1 kotak dari koordinat di atas queue sehingga akan mencari secara “melebar”. Algoritma *depth-first search* akan mencari koordinat-koordinat yang berjarak 1 kotak dari koordinat di atas Stack sehingga akan mencari secara “mendalam”.

Untuk menyelesaikan permasalahan TSP, dapat dilakukan *backtracking* jalur pada treasure terakhir atau juga dapat dilakukan pencarian secara *BFS* atau *DFS* lagi untuk mencari titik asal. Selain itu, bisa juga digunakan algoritma *brute-force* untuk menyelesaikan permasalahan TSP walaupun kompleksitas program yang digunakan sangat besar sehingga tidak akan efisien pada jumlah titik yang banyak.

Pengembangan *desktop application* menggunakan bahasa pemrograman C# dapat menggunakan bantuan IDE Visual Studio yang sangat membantu dalam proses debugging program dan langkah *publish* pada program.



## 5.2. Saran

Algoritma *brute-force* TSP yang diimplementasikan dapat diimplementasikan dengan lebih efisien dengan menggunakan algoritma *Dynamic Programming*. Algoritma tersebut akan menurunkan kompleksitas program dari  $O(n!)$  menjadi  $O(2^n n^2)$ . Selain itu, dapat juga diimplementasikan kasus-kasus dimana terdapat treasure yang tidak bisa diakses dan algoritma DFS juga dapat dibuat lebih efisien dengan menerapkan algoritma IDS.

## 5.3. Refleksi

Melalui tugas besar yang diberikan ini, kami dapat secara langsung membuat algoritma terkait materi BFS dan DFS yang sudah diajarkan di kelas. Selain itu, kami juga belajar cara membuat *desktop application* menggunakan Framework C# *desktop application development*. Selain itu, tugas besar ini melatih kemampuan kerjasama dalam tim sebagai *softskill* yang harus dimiliki oleh seorang programmer.

## 5.4. Tanggapan

Pengembangan *desktop application* yang menggunakan bahasa pemrograman C# tidak *versatile* karena IDE Visual Studio hanya tersedia pada sistem operasi Windows. Hal ini menyebabkan salah satu anggota pada kelompok ini harus menginstall sistem operasi Windows pada virtual machine yang mempersulit pengerjaan tugas besar. Selain itu, *desktop application* yang dikembangkan hanya bisa dijalankan pada sistem operasi Windows, sehingga program tidak bersifat *cross-platform*. Kedepannya, diharapkan tugas besar akan menggunakan bahasa yang *cross-platform* agar pengerjaan tugas besar tidak sulit.

## DAFTAR PUSTAKA

- R. Munir, (2023). *Tubes2-Stima-2023* [Online]. Tersedia: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2022-2023/Tubes2-Stima-2023.pdf>
- R. Munir, (2023). *BFS-DFS-2021-Bag1* [Online]. Tersedia: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf>
- R. Munir, (2023). *BFS-DFS-2021-Bag2* [Online]. Tersedia: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf>

- Ramandeep8421. *Breadth First Traversal ( BFS ) on a 2D array* [Online]. Tersedia: <https://www.geeksforgeeks.org/breadth-first-traversal-bfs-on-a-2d-array/>
- GeeksforGeeks. *Heap's Algorithm for generating permutations* [Online]. Tersedia: <https://www.geeksforgeeks.org/heaps-algorithm-for-generating-permutations/>

## **PRANALA**

Link Github : [https://github.com/sozyGithub/Tubes2\\_bebas](https://github.com/sozyGithub/Tubes2_bebas)