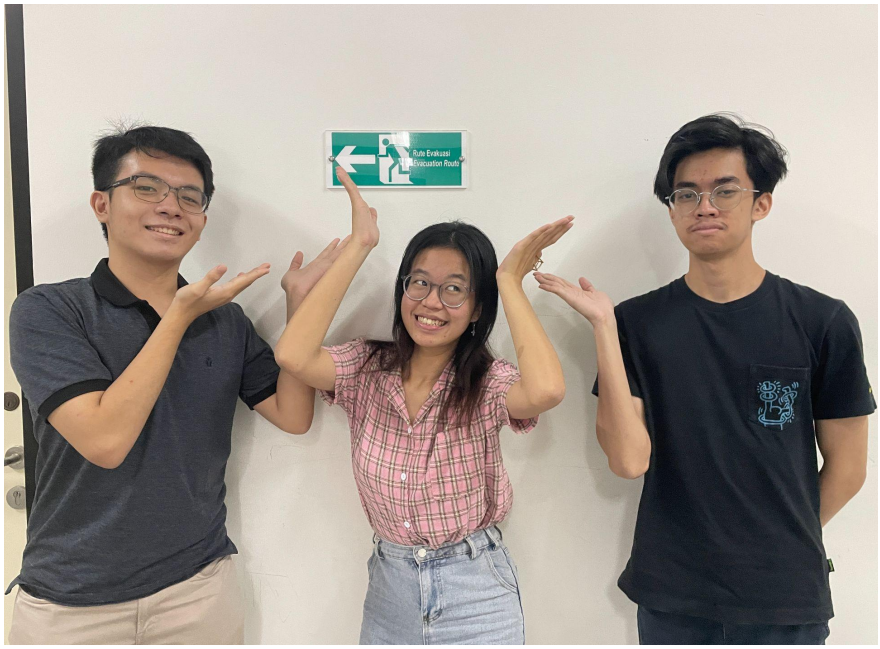
# Tugas Besar II IF2211 Strategi Algoritma

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

*Diajukan sebagai pemenuhan tugas besar II.*




Oleh:

Kelompok 24 ( **Spongebot**)

1. 13521094 - Angela Livia Arumsari
2. 13521100 - Alexander Jason
3. 13521134 - Rinaldy Adin

**PROGRAM STUDI TEKNIK INFORMATIKA**

**SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA**

**INSTITUT TEKNOLOGI BANDUNG**

**2023**

# DAFTAR ISI

# BAB 1
# DESKRIPSI MASALAH

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

Gambar 1. Labirin di Bawah Krusty Krab
(Sumber:https://static.wikia.nocookie.net/theloudhouse/images/e/ec/Massive_Mustard_Pocket.png/revision/latest?cb=20180826170029)

Setelah berpikir cukup lama, Tuan Krabs tiba-tiba mengingat bahwa ketika Ia berada pada kelas Strategi Algoritma-nya dulu, Ia ingat bahwa Ia dulu mempelajari algoritma BFS dan DFS sehingga Tuan Krabs menjadi yakin bahwa persoalan ini dapat diselesaikan menggunakan kedua algoritma tersebut. Akan tetapi, dikarenakan sudah lama tidak menyentuh algoritma, Tuan Krabs telah lupa bagaimana cara untuk menyelesaikan persoalan ini dan Tuan Krabs pun kebingungan. Tidak butuh waktu lama, Ia terpikirkan sebuah solusi yang brilian. Solusi tersebut adalah meminta mahasiswa yang saat ini sedang berada pada kelas Strategi Algoritma untuk menyelesaikan permasalahan ini.

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), dapat ditelusuri 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. Cara input maze boleh langsung input file atau dengan textfield sehingga pengguna dapat mengetik nama maze yang diinginkan. Apabila dengan textfield, harus menghandle kasus apabila tidak ditemukan dengan nama file tersebut

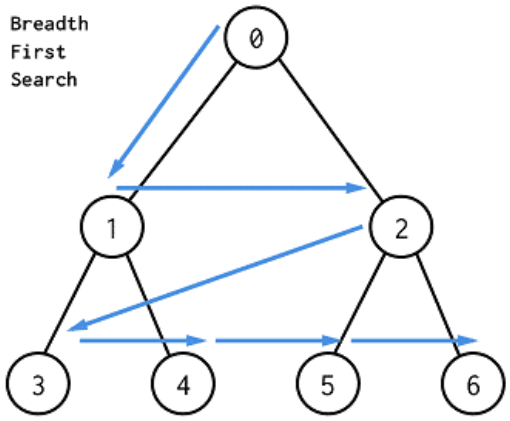
# BAB 2
# LANDASAN TEORI

## 2.1. Graph Traversal

Graph traversal adalah proses pengunjungan dan pemrosesan simpul dalam suatu graph. Pemrosesan dan pengunjungan tersebut dapat digunakan dalam masalah-masalah pencarian simpul, pencarian rute, dan lain-lain. Terdapat berbagai cara untuk melakukan traversal graf berdasarkan strategi pemilihan simpul untuk dikunjungi.

Dari berbagai strategi traversal graf yang ada, beberapa strategi pencarian paling sederhana adalah Breadth First Search dan Depth First Search. Kedua algoritma tersebut berbeda dari prioritas algoritma tersebut dalam memilih simpul yang akan dikunjungi setelahnya. Namun, terdapat kesamaan dalam kedua algoritma, yaitu kedua algoritma tersebut mencatat simpul mana yang sudah dikunjungi agar tidak mengunjungi simpul yang sama dua kali.

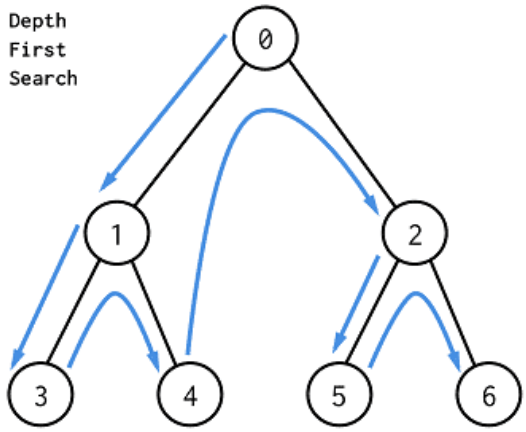
## *2.2. Breadth First Search (BFS)*




Gambar 2.2. Breadth First Search

Algoritma Breadth-First Search (BFS) memilih simpul berikutnya secara melebar. Dalam BFS, setelah suatu di simpul diproses, simpul simpul tetangganya akan ditambahkan ke dalam suatu *queue*, lalu akan memilih simpul berikutnya berdasarkan *queue* tersebut. Struktur data *queue* digunakan karena sifat *First In First Out* (FIFO) dari struktur data *queue*. Algoritma ini akan jalan sampai semua simpul sudah dikunjungi atau simpul tujuan sudah ditemukan. Algoritma BFS lebih optimal daripada DFS dalam pemecahan masalah rute terpendek dalam graph tidak berbobot karena sifat pencariannya yang melebar.

### *2.3. Depth First Search (DFS)*




Gambar 2.3. Depth First Search

Algoritma Depth-First Search (DFS) akan memilih simpul berikutnya secara mendalam. Dalam DFS, setelah suatu di simpul diproses, simpul simpul tetangganya akan ditambahkan ke dalam suatu *stack*, lalu akan memilih simpul berikutnya berdasarkan isi *stack* tersebut. Struktur data *stack* digunakan karena sifat *Last In First Out* (LIFO) dari struktur data *stack*. Algoritma ini akan jalan sampai semua simpul sudah dikunjungi atau simpul tujuan sudah ditemukan.

### *2.4. Desktop Application Development with C#*

C# (C sharp) adalah sebuah bahasa pemrograman berorientasi objek yang dikembangkan oleh Microsoft sebagai bagian dari inisiatif kerangka .NET Framework. Bahasa pemrograman ini dapat dimanfaatkan sebagai alat pengembang aplikasi desktop. Dengan menggunakan C#, pengembangan aplikasi dapat dimudahkan terutama dalam hal user interface (UI), salah satunya menggunakan WinForms/WPF dengan bantuan integrated development environment (IDE) Visual Studio.

# BAB 3
# APLIKASI ALGORITMA BFS DAN DFS

## 3.1. Langkah-langkah Pemecahan Masalah

Dalam membangun solusi berbasis C# dan .NET yang digunakan untuk memecahkan masalah pencarian Maze Treasure Hunt, kelompok kami melakukan langkah langkah berikut,

1. Memahami persoalan yang ingin diselesaikan, serta dekomposisi masalah tersebut menjadi elemen-elemen dalam persoalan graph.
2. Menemukan algoritma dasar BFS dan DFS yang sesuai dengan persoalan Maze Treasure Hunt
3. Mempelajari syntax dan semantics dari bahasa C#, serta framework .NET dan WPF yang akan digunakan untuk membangun algoritma graph traversal dan GUI yang akan digunakan untuk menampilkan solusi.
4. Membuat algoritma BFS dan DFS dalam bahasa C#.
5. Membuat algoritma untuk membaca dan memvalidasi file input.
6. Membuat elemen-elemen GUI, serta grid yang menampilkan maze menggunakan framework WPF.
7. Menghubungkan elemen GUI dan grid maze dengan algoritma BFS dan DFS yang telah dibuat

## 3.2. *Mapping Persoalan Maze Treasure Hunt* ke Elemen Algoritma BFS dan DFS

### 3.2.1. *Breadth-First Search*

Dalam pemecahan persoalan *maze treasure hunt* dengan algoritma BFS, algoritma ini menggunakan konsep algoritma *searching* dengan graf dinamis. Dalam merepresentasikan graf, digunakan daftar ketetanggan atau *adjacency list*. Graf menggunakan struktur data *queue* yang berisi simpul atau *node* yang akan dikunjungi. Setiap *node* berisi titik koordinat dari *maze.* Setiap koordinat yang dikunjungi akan dipetakan lagi menjadi koordinat lain yang bertetangga dengan koordinat tersebut.

| | |
|---|---|
| Nodes | Lokasi/koordinat *cell* pada Maze |
| Edges | Koordinat valid yang bertetangga dengan sebuah node |
| Goal State | *Treasure* pada *maze* |

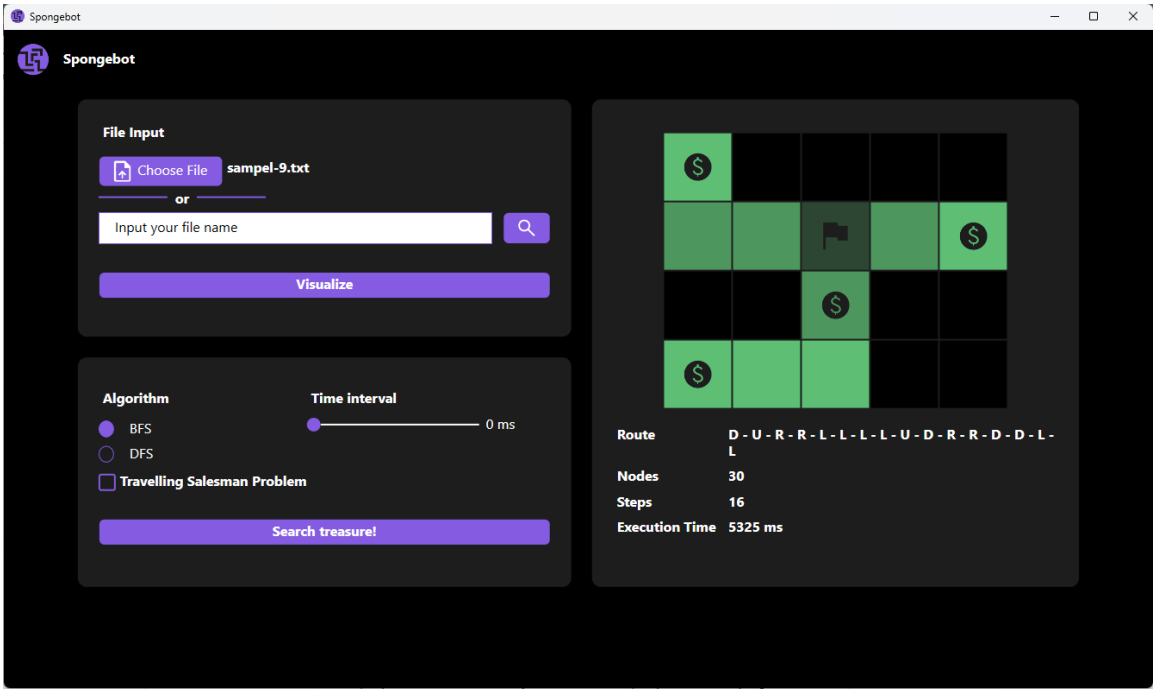
### 3.2.2. *Depth-First Search*

Dalam pemecahan persoalan *maze treasure hunt* dengan algoritma BFS, algoritma ini menggunakan konsep algoritma *searching* dengan graf dinamis. Dalam merepresentasikan graf, digunakan daftar ketetanggan atau *adjacency list*. Graf menggunakan struktur data *stack* yang berisi simpul atau *node* yang akan dikunjungi. Setiap *node* berisi titik koordinat dari *maze.* Setiap koordinat yang dikunjungi akan dipetakan lagi menjadi koordinat lain yang bertetangga dengan koordinat tersebut.

| | |
|---|---|
| Nodes | Lokasi/koordinat *cell* pada Maze |
| Edges | Koordinat valid yang bertetangga dengan sebuah node |
| Goal State | *Treasure* pada *maze* |

## 3.3. Ilustrasi Penyelesaian Kasus

### 3.3.1. Penyelesaian Kasus dengan BFS



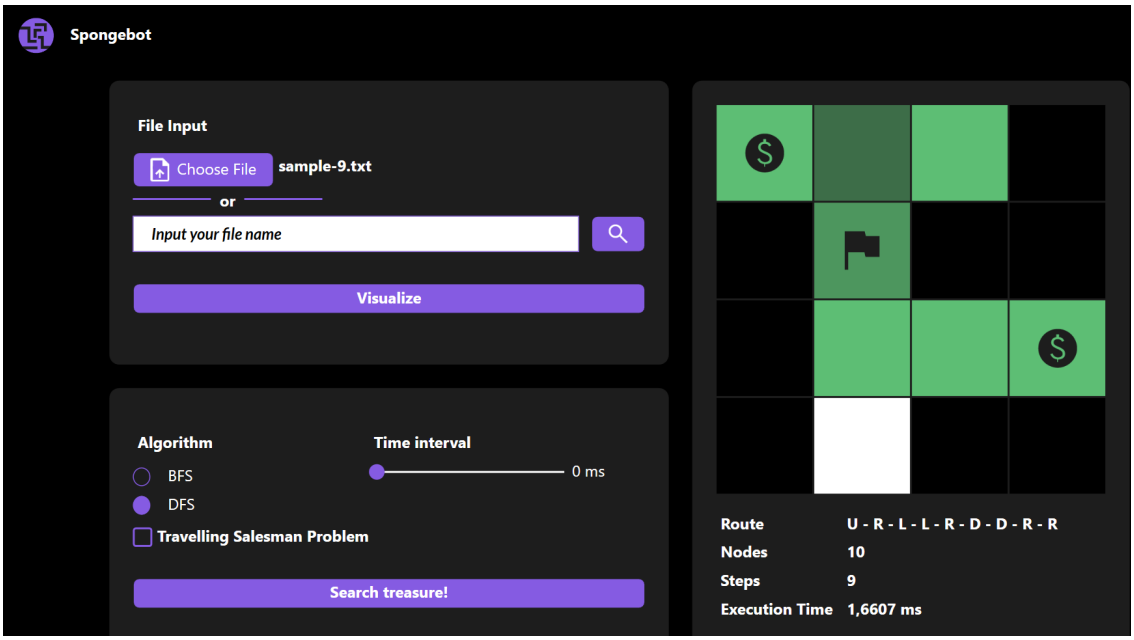

*Gambar 3.3.1.1 Ilustrasi Kasus BFS*

Gambar diatas mengilustrasikan sebuah penyelesaian pencarian treasure menggunakan algoritma BFS. Algoritma pencarian BFS kami memilih rute yang dilalui secara heuristik dengan melakukan pencarian BFS yang berawal dari start dan akan mencari treasure yang terdekat dari start, lalu dari treasure tersebut program kami akan mengosongkan queue pencariannya dan mengulang algoritma BFS untuk mencari treasure lain yang terdekat dengan treasure yang terakhir ditemukan. Algoritma BFS kami melakukan pendekatan heuristik tersebut sampai semua treasure sudah dikunjungi, dan dalam mode TSP, jika sudah kembali ke posisi start.

Ilustrasi penyelesaian di atas menunjukkan kasus terdapatnya 4 treasure yang berjarak berbeda dengan posisi start. Dari rute yang dihasilkan oleh BFS, dapat dilihat bahwa algoritma mengunjungi treasure di tengah terlebih dahulu, lalu treasure di kanan, lalu treasure di kiri atas, dan terakhir treasure di kiri bawah. Meskipun terdapat rute lebih cepat jika algoritma mengunjungi treasure tengah setelah mengunjungi treasure kanan dan kiri atas, yang akan mengunjungi semua treasure hanya dalam 14 step. Rute yang algoritma BFS kami pilih lebih

lambat karena heuristik algoritma kami yang selalu memilih treasure yang terdekat dengan posisi algoritma saat itu. Kami memilih heuristik tersebut karena kami nilai lebih cocok dengan menimbang bahwa kasus permasalahan dalam program ini cenderung memiliki jumlah treasure yang banyak. Tanpa heuristik tersebut, algoritma BFS akan membangkitkan rute yang sangat banyak jika terdapat maze yang rumit, serta jumlah treasure yang banyak.

### 3.3.2. Penyelesaian Kasus dengan DFS




*Gambar 3.3.2.1 Ilustrasi Kasus BFS*

Ilustrasi penyelesaian di atas menunjukkan kasus terdapatnya 2 treasure yang berada pada titik yang berseberangan. Dari rute yang dihasilkan oleh DFS, dapat dilihat bahwa algoritma mengunjungi treasure di atas kiri terlebih dahulu, lalu treasure di kanan bawah. Ketika sebuah cell berhadapan dengan jalan buntu atau cell yang bertetangga dengannya sudah pernah dikunjungi sebelumnya, Algoritma DFS kami akan melakukan proses backtrack dengan mengunjungi node lain. Dapat dilihat pada cell yang berwarna lebih gelap, menandakan bahwa node mengunjungi sebuah cell lebih dari 1 kali untuk mencapai treasure di kanan bawah.

# BAB 4
# ANALISIS DAN PEMECAHAN MASALAH

## 4.1. Implementasi program

### 4.4.1. Breadth-First Search

a. runNonTSP

```
procedure runNonTSP(output finalPath: MazePath, input Board: board)
{ Traversal board dengan algoritma penelusuran BFS tanpa TSP
  I.S. board terdefinisi dan tidak sembarang
  F.S. finalPath berisi rute yang berawal dari start dan melalui
       semua harta karun. }
KAMUS
  { Variabel }
  initialPath, currentPath : MazePath
  completePath : List of MazePath
  unvisitedTreasure : Set of Cell
  pathQ : Queue of MazePath
  neighborPositions : List of Point
  neighbor : Cell
  { Fungsi dan prosedur antara }
  procedure createQueue (input q : Queue)
  { Inisialisasi queue
    I.S. q belum terdefinisi
    F.S. q terdefinisi dan kosong }
  procedure Enqueue(input q : Queue, input mp: MazePath)
  { Menambahkan elemen ke queue
    I.S. q dan mp terdefinisi
    F.S. mp ditambahkan ke belakang Queue q }
  procedure appendList(input/ouput l : List, input mp: MazePath)
  { Menambahkan elemen ke list
    I.S. l dan mp terdefinisi
    F.S. mp ditambahkan ke belakang List l }
  procedure setRemove(input/ouput s : Set, input c: Cell)
  { Menghapus elemen c dari s
    I.S. s dan c terdefinisi
    F.S. s tidak mengandung elemen c }
  function createSet(input cells : List of Cell)
  { Mengembalikan Set baru berisi elemen dari cells }
  function MazePath createMazePath()
  { Mengembalikan MazePath kosong }
  function MazePath createMazePath(MazePath p, Cell cell)
  { Mengembalikan MazePath baru dari salinan p dengan ditambahkan cell }
  function MazePath createMazePath(List of MazePath mps)
  { Mengembalikan MazePath baru dari array mps }
  function MazePath createMazePath(Cell cell)
  { Mengembalikan MazePath baru berisi cell }
  function bool isEmpty(Queue q)
  { Mengembalikan apakah q kosong atau tidak }
  function bool setContains(Set s, Cell c)
  { Mengembalikan apakah Cell c terdapat Set s }
```

```
  function MazePath dequeue(Queue q)
  { Mengembalikan elemen terdepan q dan menghapuskannya dari queue }
  function Position createPoint(int x, int y)
  { Mengembalikan Point dengan koordinat (x, y) }
  function bool isValidPosition(Board board, Position p)
  { Mengembalikan apakah p merupakan posisi yang valid pada board }
  function bool cellIsVisited(Cell cell, MazePath path)
  { Mengembalikan apakah cell sudah dikunjungi di dalam path }
  function bool isWall(Cell cell)
  { Mengembalikan apakah cell bertipe wall }

ALGORITMA
  initialPath <- createMazePath(startCell)
  unvisitedTreasure <- createSet(board.treasureCells)
  createQueue(pathQ)

  enqueue(pathQ, initialPath)

  while (not isEmpty(pathQ)) do
     currentPath <- dequeue(pathQ)
     lastCell <- currentPath[currentPath.Length - 1]

     { reset path yang di dalam queue jika menemukan suatu treasure,
       dianggap seperti program melakukan traversal berawal
       dari posisi treasure yang terakhir ditemukan }
     if (setContains(unvisitedTreasure, lastCell)) then
       appendList(completePath, currentPath)
       clearQueue(pathQ)
       setRemove(unvisitedTreasure, lastCell)
       currentPath <- createMazePath()

     { return jika sudah tidak ada treasure yang belum dikunjungi }
     if (isEmpty(unvisitedTreasure)) then
       -> createMazePath(completePath)

     neighborPoisitions <- [
       createPoint(lastCell.Position.X, lastCell.Position.Y - 1),
       createPoint(lastCell.Position.X + 1, lastCell.Position.Y),
       createPoint(lastCell.Position.X, lastCell.Position.Y + 1),
       createPoint(lastCell.Position.X - 1, lastCell.Position.Y),
     ]

     { tambahkan rute melalui tetangga yang valid ke dalam queue }
     for neighborPosition in neighborPositions do
       if (isValidPosition(board, neighborPosition) and
           (cellIsVisited(board[neighborPosition], currentPath)) and
           (not isWall(board[neighborPosition])) then
         neighbor = board[neighborPosition]
         enqueue(pathQ, createMazePath(currentPath, neighbor))
```

b. runTSP

```
procedure runNonTSP(output finalPath: MazePath, input Board: board)
{ Traversal board dengan algoritma penelusuran BFS tanpa TSP
  I.S. board terdefinisi dan tidak sembarang
```

```
  F.S. finalPath berisi rute yang berawal dari start dan melalui
       semua harta karun. }
KAMUS
  { Variabel }
  initialPath, currentPath : MazePath
  completePath : List of MazePath
  unvisitedTreasure : Set of Cell
  pathQ : Queue of MazePath
  neighborPositions : List of Point
  neighbor : Cell
  { Fungsi dan prosedur antara }
  procedure createQueue (input q : Queue)
  { Inisialisasi queue
    I.S. q belum terdefinisi
    F.S. q terdefinisi dan kosong }
  procedure Enqueue(input q : Queue, input mp: MazePath)
  { Menambahkan elemen ke queue
    I.S. q dan mp terdefinisi
    F.S. mp ditambahkan ke belakang Queue q }
  procedure appendList(input/ouput l : List, input mp: MazePath)
  { Menambahkan elemen ke list
    I.S. l dan mp terdefinisi
    F.S. mp ditambahkan ke belakang List l }
  procedure setRemove(input/ouput s : Set, input c: Cell)
  { Menghapus elemen c dari s
    I.S. s dan c terdefinisi
    F.S. s tidak mengandung elemen c }
  function createSet(input cells : List of Cell)
  { Mengembalikan Set baru berisi elemen dari cells }
  function MazePath createMazePath()
  { Mengembalikan MazePath kosong }
  function MazePath createMazePath(MazePath p, Cell cell)
  { Mengembalikan MazePath baru dari salinan p dengan ditambahkan cell }
  function MazePath createMazePath(List of MazePath mps)
  { Mengembalikan MazePath baru dari array mps }
  function MazePath createMazePath(Cell cell)
  { Mengembalikan MazePath baru berisi cell }
  function bool isEmpty(Queue q)
  { Mengembalikan apakah q kosong atau tidak }
  function bool setContains(Set s, Cell c)
  { Mengembalikan apakah Cell c terdapat Set s }
  function MazePath dequeue(Queue q)
  { Mengembalikan elemen terdepan q dan menghapuskannya dari queue }
  function Position createPoint(int x, int y)
  { Mengembalikan Point dengan koordinat (x, y) }
  function bool isValidPosition(Board board, Position p)
  { Mengembalikan apakah p merupakan posisi yang valid pada board }
  function bool cellIsVisited(Cell cell, MazePath path)
  { Mengembalikan apakah cell sudah dikunjungi di dalam path }
  function bool isWall(Cell cell)
  { Mengembalikan apakah cell bertipe wall }
  function bool isStart(Cell cell)
  { Mengembalikan apakah cell bertipe start }

ALGORITMA
  initialPath <- createMazePath(startCell)
```

```
  unvisitedTreasure <- createSet(board.treasureCells)
  createQueue(pathQ)

  enqueue(pathQ, initialPath)

  while (not isEmpty(pathQ)) do
     currentPath <- dequeue(pathQ)
     lastCell <- currentPath[currentPath.Length - 1]

     { reset path yang di dalam queue jika menemukan suatu treasure
       atau sudah menemukan start kembali pada akhir TSP,
       dianggap seperti program melakukan traversal berawal
       dari posisi treasure yang terakhir ditemukan }
     if (setContains(unvisitedTreasure, lastCell) or
         (isEmpty(unvisitedTreasure) and isStart(lastCell))) then
       appendList(completePath, currentPath)
       clearQueue(pathQ)
       setRemove(unvisitedTreasure, lastCell)
       currentPath <- createMazePath()

     { return jika sudah tidak ada treasure yang belum dikunjungi
       dan rute sudah sampai cell start }
     if (isEmpty(unvisitedTreasure) and isStart(lastCell)) then
       -> createMazePath(completePath)

     neighborPoisitions <- [
       createPoint(lastCell.Position.X, lastCell.Position.Y - 1),
       createPoint(lastCell.Position.X + 1, lastCell.Position.Y),
       createPoint(lastCell.Position.X, lastCell.Position.Y + 1),
       createPoint(lastCell.Position.X - 1, lastCell.Position.Y)
     ]

     { tambahkan rute melalui tetangga yang valid ke dalam queue }
     for neighborPosition in neighborPositions do
       if (isValidPosition(board, neighborPosition) and
           (cellIsVisited(board[neighborPosition], currentPath)) and
           (not isWall(board[neighborPosition])) then
         neighbor = board[neighborPosition]
         enqueue(pathQ, createMazePath(currentPath, neighbor))
```

### 4.1.2. Depth-First Search

Keterangan: MazePath merupakan sebuah kelas dengan atribut path (array of Cell) dan prevCell (Stack of Cell) yang menyimpan node/cell yang aktif. prevCell berguna dalam proses backtracking

**a. run**

```
procedure run(output finalPath: MazePath, output totalSteps:
integer, output visitedNodes: integer, input board: Board, input
isTSP: boolean)
{I.S. board terdefinisi dan valid
 F.S. menghasilkan jalur dari algoritma DFS beserta jumlah langkah
dan nodes}
```

```
KAMUS
 {variabel}
 visitedNodes : integer
 totalSteps: integer
 pathS : Stack of MazePath
 initialPath: MazePath
 currentPath: MazePath
 lastCell: Cell
 neighborPositions : List of Point
 neighbor : Cell
 deadend: boolean

 {prosedur dan fungsi antara}
 procedure createMazePath(input/output m: MazePath, input
startCell:Cell)
   {Inisialisasi Path dengan cell awal startCell dan push startCell
ke dalam prevCell (keterangan tertera di awal bagian 4.1.2)
  I.S. m belum terdefinisi
  F.S. m terdefinisi dan kosong}
 procedure Stack (input/output s: Stack)
  {Inisialisasi Stack
  I.S. s belum terdefinisi
  F.S. s terdefinisi dan kosong}
 procedure Push (input/output s: Stack, input n: MazePath)
  {menambah elemen Stack
  I.S. s terdefinisi
  F.S. n masuk ke dalam s dengan aturan LIFO}
 function Pop (s: Stack) -> MazePath
 {menghapus path pada stack s dengan aturan LIFO dan mengembalikan
path tersebut}
 function isStackEmpty(s: Stack) -> boolean
  {mengembalikan true jika stack kosong dan false jika sebaliknya}
 procedure Clear(input/output s: Stack)
  {menghapus seluruh isi stack
  I.S. s terdefinisi
  F.S. s terdefinisi dan kosong}

 procedure addPath(input/output n: MazePath, input c: Cell)
  {Menambahkan cell pada path n
  I.S. n dan c terdefinisi
  F.S. c masuk dalam MazePath}
 function getTreasureCount1(m: MazePath)-> integer
  {mengembalikan jumlah treasure pada path yang dilalui}
 function getTreasureCount2(b: Board)-> integer
  {mengembalikan jumlah treasure pada board maze}
 function Length (m: MazePath)-> integer
  {mengembalikan panjang path m}
 function finalRoute(m: MazePath)-> string
  {mengembalikan rute path m dalam bentuk string (R,L,U,D)}
 function TSP(c: Cell)-> MazePath
  {mengembalikan rute path dari cell c kembali ke titik start}
```

```
 function Position createPoint(int x, int y)
  { Mengembalikan Point dengan koordinat (x, y) }
 function bool isValidPosition(Board board, Position p)
  { Mengembalikan apakah p merupakan posisi yang valid pada board }
 function bool cellIsVisited(Cell cell, MazePath path)
  { Mengembalikan apakah cell sudah dikunjungi di dalam path }
 function bool isWall(Cell cell)
  { Mengembalikan apakah cell bertipe wall }
 function bool isStart(Cell cell)
  { Mengembalikan apakah cell bertipe start }

ALGORITMA
 {Inisialisasi variabel}
 deadend <- True
 visitedNodes <- 0
 totalSteps <- 0
 createMazePath(initialPath, startCell)
 Stack(pathS)

 {push cell yang menjadi titik start}
 Push(pathS, initialPath)

 {memulai pencarian DFS}
 while(not isStackEmpty(pathS)) do
  currentPath <- Pop(pathS)
  lastCell <- currentPath[ Length(currentPath)-1 ]

  {jika semua treasure sudah dikunjungi}
    if  (getTreasureCount1(currentPath)  =  getTreasureCount2(board))
then
     if (isTSP) then
      currentPath   <-   MazePath   (currentPath,   TSP   (currentPath
[Length(currentPath)-1]

     visitedNodes<-Length(currentPath)
     totalSteps<-visitedNodes-1

     Print (route(currentPath))
     Print (visitedNodes)
     Print (totalSteps)

     Clear(pathS) {DFS berakhir}
  else
     neighborPoisitions <- [
      createPoint(lastCell.Position.X - 1, lastCell.Position.Y),
      createPoint(lastCell.Position.X, lastCell.Position.Y + 1),
      createPoint(lastCell.Position.X + 1, lastCell.Position.Y),
      createPoint(lastCell.Position.X, lastCell.Position.Y - 1),
    ]

    for neighborPosition in neighborPositions do
```

```
      if (isValidPosition(board, neighborPosition) and
         (cellIsVisited(board[neighborPosition], currentPath)) and
         (not isWall(board[neighborPosition])) then
        neighbor = board[neighborPosition]
        Push(pathS, addPath(currentPath, neighbor))
        deadend<-False

    if (deadend) then
      Pop(currentPath.prevCell)
      Push(pathS, addPath(currentPath, Pop(currentPath.prevCell)))
```

### b. TSP

```
function TSP(input lastTreasure: Cell) -> MazePath
{mengembalikan rute dari lastTreasure ke titik start}

KAMUS
 {variabel}
 pathS : Stack of MazePath
 tspPath: MazePath
 initialPath: MazePath
 currentPath: MazePath
 lastCell: Cell
 neighborPositions : List of Point
 neighbor : Cell
 deadend: boolean
 i: integer

 {prosedur dan fungsi antara}
 procedure createMazePath(input/output m: MazePath, input
startCell:Cell)
   {Inisialisasi Path dengan cell awal startCell dan push startCell
ke dalam prevCell (keterangan tertera di awal bagian 4.1.2)
  I.S. m belum terdefinisi
  F.S. m terdefinisi dan kosong}
 procedure Stack (input/output s: Stack)
  {Inisialisasi Stack
  I.S. s belum terdefinisi
  F.S. s terdefinisi dan kosong}
 procedure Push (input/output s: Stack, input n: MazePath)
  {menambah elemen Stack
  I.S. s terdefinisi
  F.S. n masuk ke dalam s dengan aturan LIFO}
 function Pop (s: Stack) -> MazePath
 {menghapus path pada stack s dengan aturan LIFO dan mengembalikan
path tersebut}
 function isStackEmpty(s: Stack) -> boolean
  {mengembalikan true jika stack kosong dan false jika sebaliknya}
 procedure Clear(input/output s: Stack)
  {menghapus seluruh isi stack
```

```
  I.S. s terdefinisi
  F.S. s terdefinisi dan kosong}

 procedure addPath(input/output n: MazePath, input c: Cell)
  {Menambahkan cell pada path n
  I.S. n dan c terdefinisi
  F.S. c masuk dalam MazePath}
 function Length (m: MazePath)-> integer
  {mengembalikan panjang path m}

 function Position createPoint(int x, int y)
  { Mengembalikan Point dengan koordinat (x, y) }
 function bool isValidPosition(Board board, Position p)
  { Mengembalikan apakah p merupakan posisi yang valid pada board }
 function bool cellIsVisited(Cell cell, MazePath path)
  { Mengembalikan apakah cell sudah dikunjungi di dalam path }
 function bool isWall(Cell cell)
  { Mengembalikan apakah cell bertipe wall }
 function bool isStart(Cell cell)
  { Mengembalikan apakah cell bertipe start }

ALGORITMA
 {Inisialisasi variabel}
 deadend <- True
 visitedNodes <- 0
 totalSteps <- 0
 createMazePath(initialPath, lastTreasure)
 Stack(pathS)

 {push cell treasure terakhir sebagai start}
 Push(pathS, initialPath)

 {memulai pencarian DFS}
 while(not isStackEmpty(pathS)) do
  currentPath <- Pop(pathS)
  lastCell <- currentPath[ Length(currentPath)-1 ]

  visitedNodes <- visitedNodes + 1;
   {jika node yang dikunjungi merupakan titik start maka algoritma
selesai}
  if (isStart(lastCell)) then
     Clear(pathS) {DFS berakhir}
  else
     neighborPositions <- [
      createPoint(lastCell.Position.X - 1, lastCell.Position.Y),
      createPoint(lastCell.Position.X, lastCell.Position.Y + 1),
      createPoint(lastCell.Position.X + 1, lastCell.Position.Y),
      createPoint(lastCell.Position.X, lastCell.Position.Y - 1),
    ]

    for neighborPosition in neighborPositions do
      if (isValidPosition(board, neighborPosition) and
```

```
          (cellIsVisited(board[neighborPosition], currentPath)) and
          (not isWall(board[neighborPosition])) then
        neighbor = board[neighborPosition]
        Push(pathS, addPath(currentPath, neighbor))
        deadend<-False

    if (deadend) then
      Pop(currentPath.prevCell)
      Push(pathS, addPath(currentPath, Pop(currentPath.prevCell)))

{menghapus path start agar tidak terduplikasi}
i<-1
while i < Length(currentPath) do
   addPath(tspPath, currentPath[i]
   i <- i+1
-> tspPath
```

### 4.2. Struktur Data

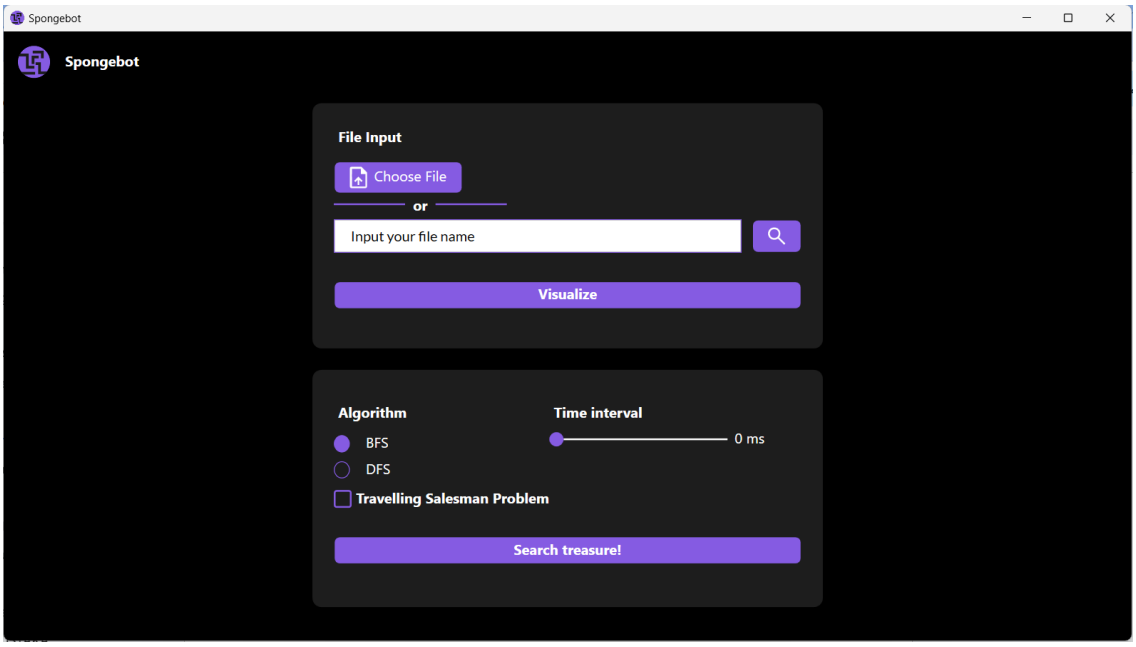
Pada program kami, kami mengembangkan program dengan menggunakan paradigma Object Oriented Programming. Kami menggunakan struktur data class untuk merepresentasikan objek/elemen yang ada dalam persoalan ini. Kode utama program berada pada kelas MainWindow.xaml.cs yang merupakan *code behind* dari GUI yang telah di design. Kelas utama di MainWindow.xaml.cs akan memanggil algoritma validasi file, traversal graph, dan algoritma-algoritma lain. Adapun kelas-kelas lain yang mendukung program utama.

| No | Nama Kelas | Penjelasan |
|---|---|---|
| 1 | MainWindow | Kelas ini merupakan kelas utama yang berisi *code behind* dari GUI berbasis XAML. Kelas ini berisi binding variabel dan fungsi yang digunakan dan dijalankan ketika terjadi interaksi dengan elemen-elemen GUI. |
| 2 | BFS | Kelas ini berisi penanganan algoritma BFS untuk menyelesaikan persoalan Maze Treasure Hunt |
| 3 | DFS | Kelas ini berisi penanganan algoritma DFS untuk menyelesaikan persoalan Maze Treasure Hunt |
| 4 | FileIO | Kelas ini akan menangani pencarian, pembacaan, dan validasi file konfigurasi maze agar dapat dijadikan struktur data yang dapat diolah dengan algoritma BFS dan DFS. |
| 5 | Board | Kelas ini merepresentasikan objek maze yang menyimpan matriks dari Cell. Kelas ini menangani pengaksesan Cell yang terdapat di dalam maze. |
| 6 | Cell | Kelas ini merepresentasikan suatu Cell yang terdapat di dalam Maze/Board. Kelas ini menyimpan informasi |

| | | |
|---|---|---|
| | | mengenai posisi, warna, dan tipe dari suatu elemen Cell di dalam Maze/Board |
| 7 | Point | Kelas ini merepresentasikan suatu posisi yang berada di dalam Maze/Board |
| 8 | MazePath | Kelas ini merepresentasikan suatu rute yang dilalui di dalam Maze. Kelas ini menyimpan larik berisi Cell sebagai rute yang dilalui. Kelas ini juga dapat mengubah warna dari semua Cell yang terdapat dalam Path. |

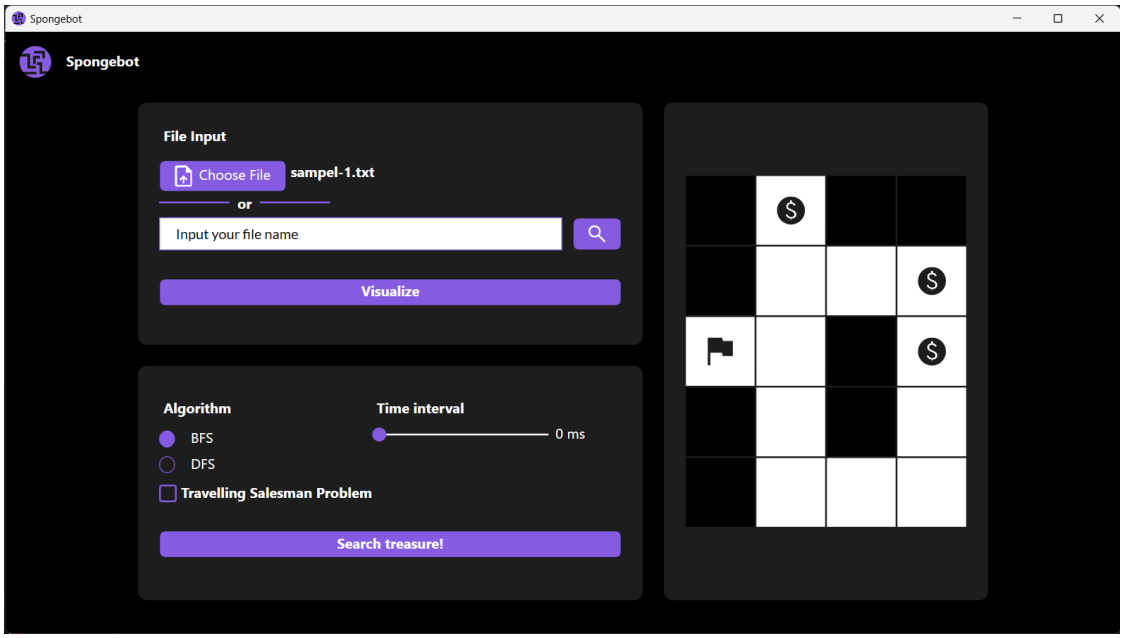
### 4.3. Tata Cara Penggunaan Program

Program dapat dijalankan dengan membuka langsung file Spongebot.exe pada folder bin di root folder. Selain cara menjalankan tersebut, program juga dapat dijalankan dengan menjalankan command dotnet Spongebot.dll pada folder bin. Ketika program dijalankan, program akan memiliki tampilan utama sebagai berikut.




Gambar 4.3.1. Tampilan Awal Program

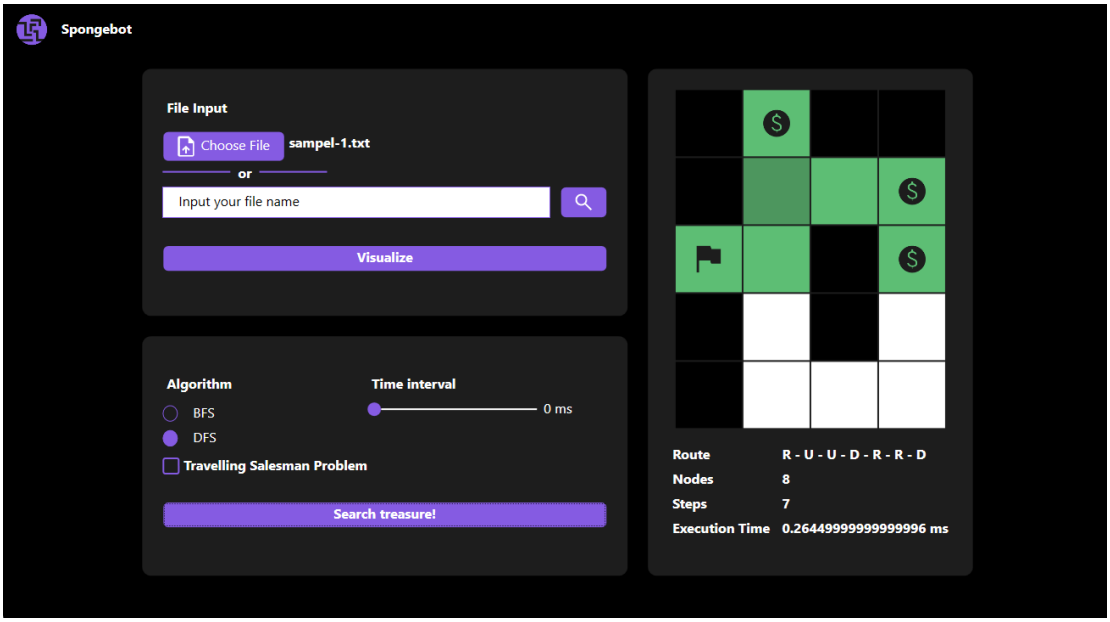
Pada program terdapat dua opsi untuk melakukan input file dengan extension berupa .txt. Opsi pertama yaitu memilih file dari file explorer dengan menekan tombol Choose File. Opsi kedua yaitu mengetik nama file yang terdapat pada folder test di root folder program. Pada opsi ini akan dilakukan validasi dari nama file yang diketik. Pada bagian file input akan terdapat beberapa validasi dari isi file yang dipilih oleh user. Program akan mengeluarkan pesan error jika terdapat lebih dari satu start atau tidak ada start atau treasure. Selain itu, program hanya menerima file dengan karakter yang valid, yaitu K, R, T, dan X.



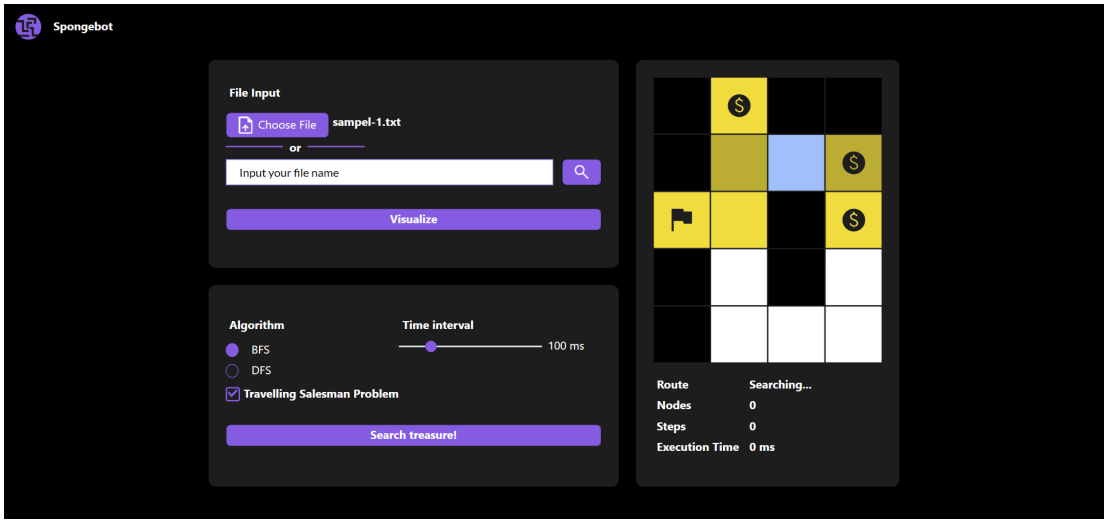

Gambar 4.3.2. Tampilan Board Maze

Setelah tombol Visualize ditekan, file input akan divisualisasikan dalam bentuk board. User dapat memilih algoritma yang ingin digunakan. Pemilihan BFS atau DFS akan menentukan algoritma yang digunakan untuk pencarian treasure. Checkbox Travelling Salesman Problem akan mengaktifkan pencarian rute untuk kembali ke start setelah menemukan seluruh treasure. Slide Time Interval akan mengatur jeda waktu dalam penampilan step pencarian treasure. Setelah user memilih algoritma yang diinginkan, user dapat melakukan pencarian treasure dengan menekan tombol Search Treasure!. Berikut merupakan contoh tampilan dari pencarian treasure.

Pada time interval 0 ms, program akan langsung menampilkan rute yang telah ditemukan algoritma. Rute akan ditampilkan secara bertahap dengan warna hijau yang semakin gelap ketika sebuah node dikunjungi lebih dari sekali. Jika time interval ditetapkan menjadi suatu angka tertentu, sebelum menampilkan rute akhir, program akan menampilkan tahapan pencarian dengan jeda waktu sesuai yang ditetapkan. Kotak yang telah dikunjungi akan diberi warna kuning, sedangkan kotak yang sedang dicek akan diberi warna biru.




Gambar 4.3.3. Tampilan Rute Hasil Pencarian



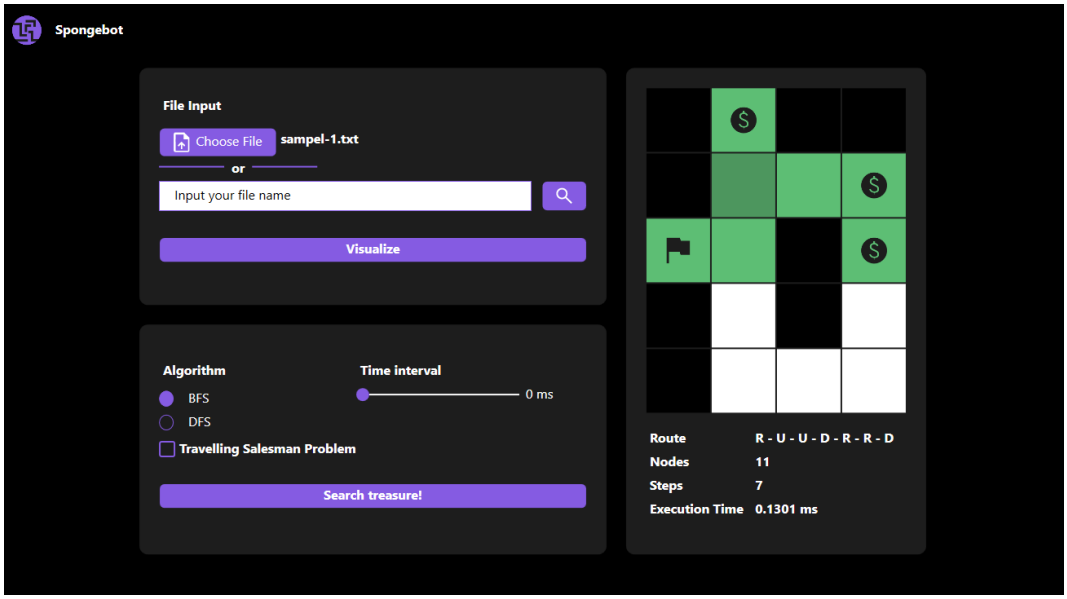

Gambar 4.3.4. Tampilan Tahap Pencarian

## 4.4. Hasil Pengujian

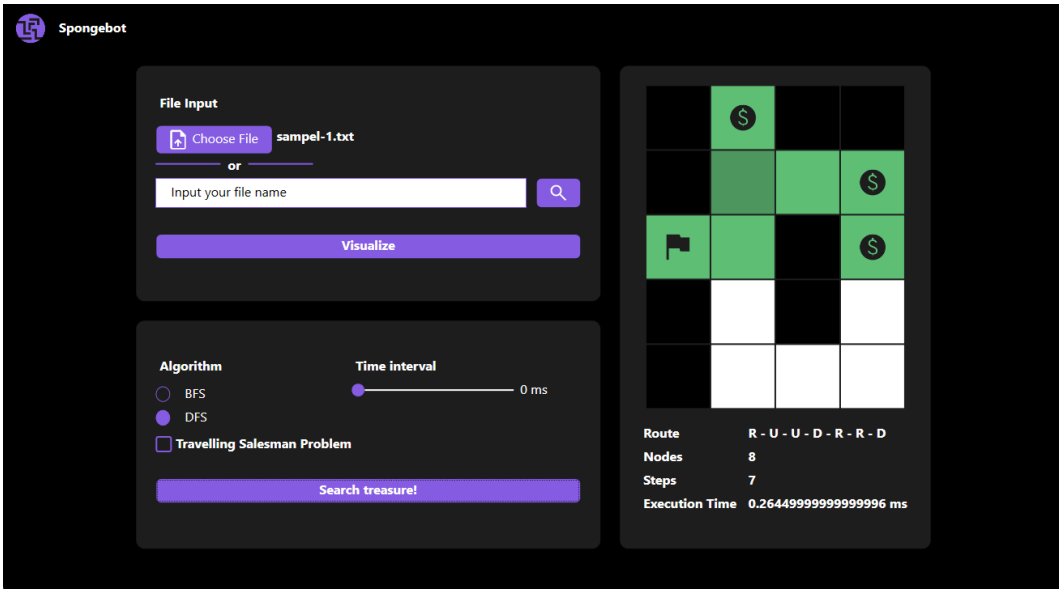
### 4.4.1. Pengujian 1

Berikut merupakan file input yang digunakan pada pengujian pertama.


*Gambar 4.4.1.1 File Maze*




*Gambar 4.4.1.2 Solusi BFS Non-TSP*



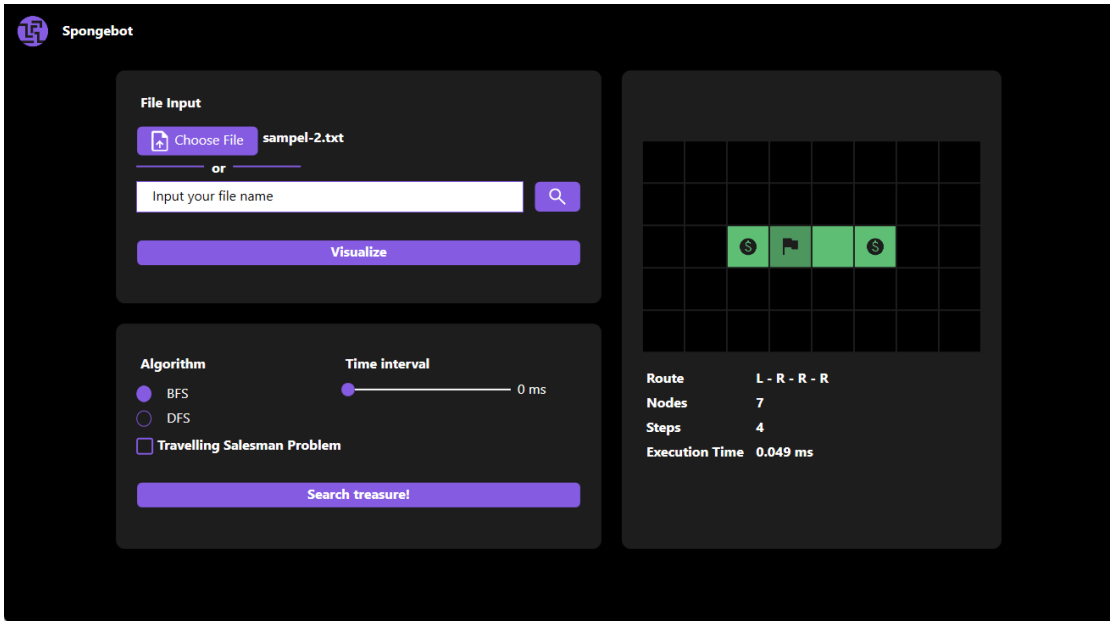

*Gambar 4.4.1.3 Solusi DFS Non-TSP*

Pada pengujian pertama, dilakukan pencarian dengan menggunakan BFS dan DFS tanpa mengaktifkan TSP. Pada pengujian ini tidak digunakan time interval untuk menampilkan tiap tahap. Hasil rute dari algoritma BFS maupun DFS sama, tetapi jumlah nodes yang dikunjungi BFS lebih banyak dibanding DFS. Hal ini dikarenakan pencarian BFS yang melebar.

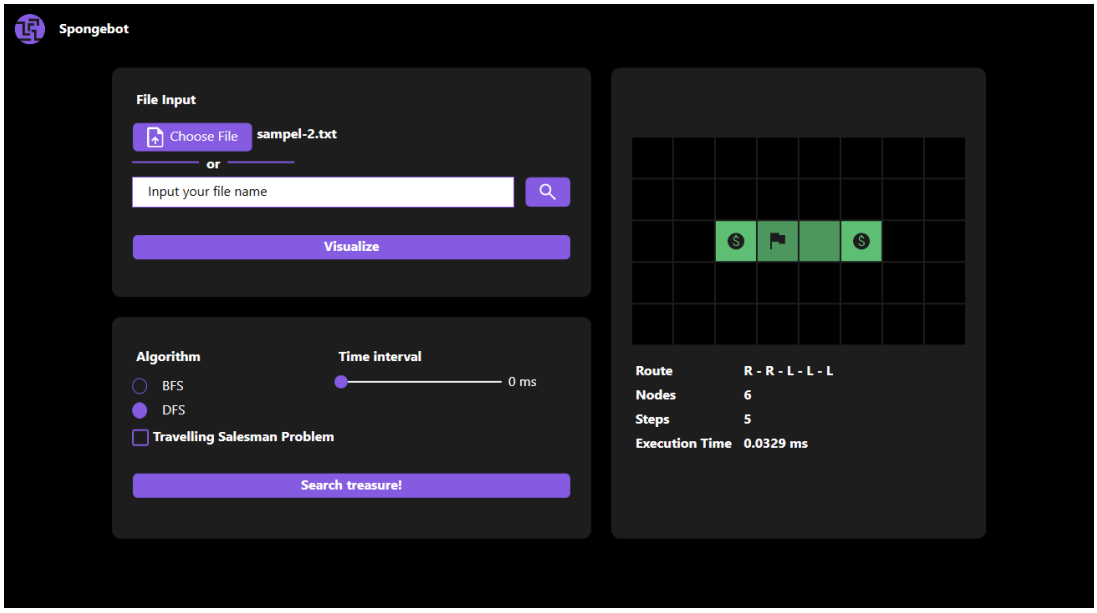
### 4.4.2. Pengujian 2

Berikut merupakan file yang digunakan untuk pengujian kedua.


*Gambar 4.4.2.1 File Maze*




*Gambar 4.4.2.2 Solusi BFS Non-TSP*

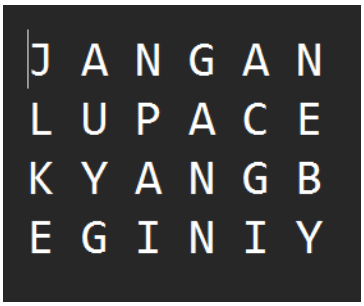



*Gambar 4.4.2.3 Solusi DFSNon-TSP*

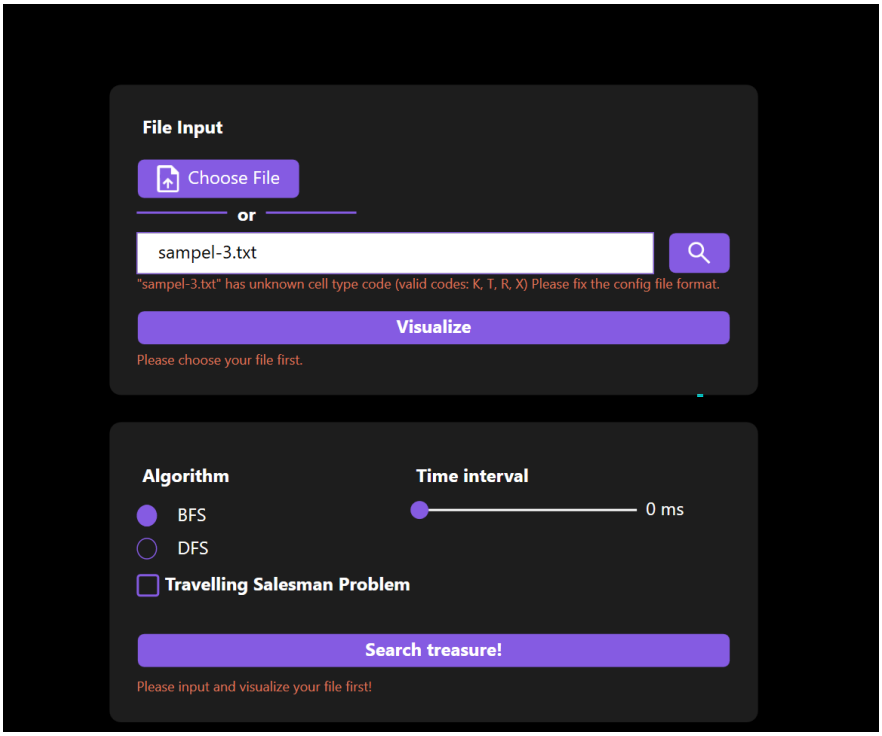
Pada pengujian kedua, dilakukan pencarian treasure dengan algoritma BFS dan DFS tanpa TSP. Sama seperti sebelumnya, kotak yang dikunjungi BFS akan lebih banyak dari DFS. Pada hasil rute yang ditampilkan, dapat diamati rute memiliki kotak yang dikunjungi lebih dari sekali. Hal ini dikarenakan untuk mendapatkan dua treasure dari start dipastikan harus melewati start dua kali. Namun, kotak yang dilalui dua kali pada BFS dan DFS akan berbeda diakibatkan prioritas pencarian dan heuristik masing-masing algoritma.

### 4.4.3. Pengujian 3

Berikut merupakan file yang digunakan untuk pengujian ketiga




*Gambar 4.4.3.1 File Maze*



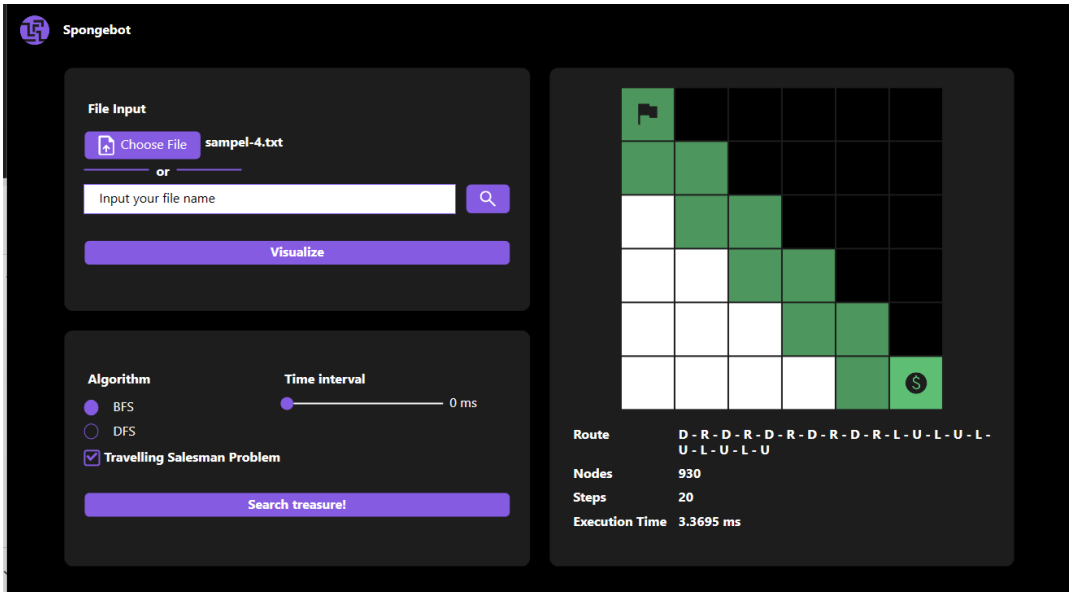

*Gambar 4.4.3.2 Tampilan Error GUI*

Pada pengujian ini, input file tidak valid dan dikeluarkan pesan error. File yang tidak valid disebabkan oleh karakter selain K, R, T, dan X. Jika input file tidak valid, pesan error juga akan dimunculkan ketika tombol Visualize dan Search Treasure! Ditekan.

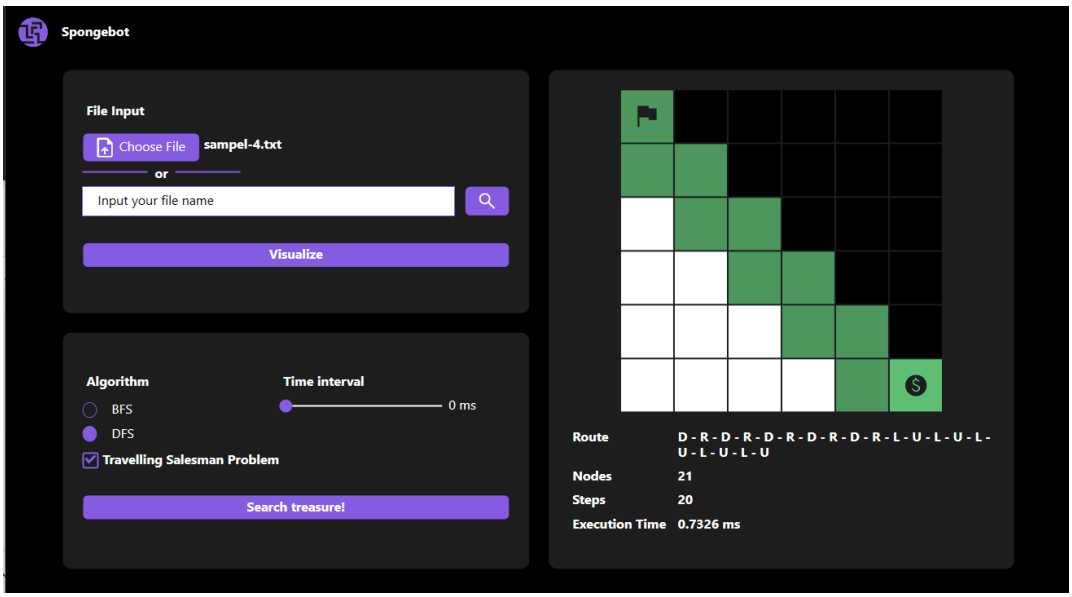
### 4.4.4. Pengujian 4

Berikut adalah file yang digunakan untuk pengujian keempat.


*Gambar 4.4.4.1 File Maze*




*Gambar 4.4.4.2 Solusi BFS TSP*



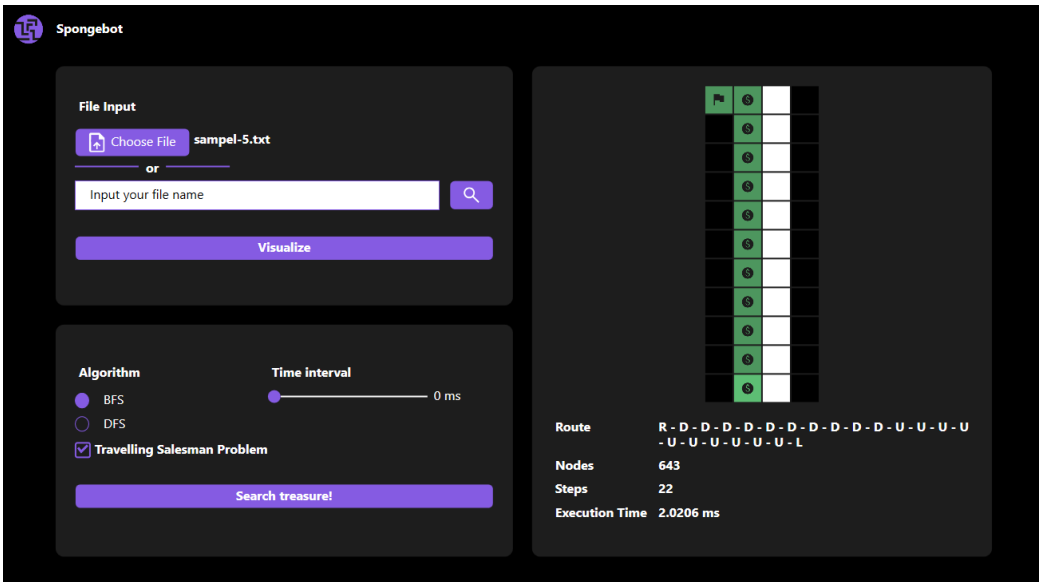

*Gambar 4.4.4.3 Solusi DFS TSP*

Pada pengujian keempat, dilakukan pengujian untuk algoritma BFS dan DFS dengan TSP, tetapi tanpa time interval. Pada pengujian kali ini, terlihat perbedaan signifikan pada jumlah nodes yang dikunjungi algoritma BFS dan DFS. Hal ini dikarenakan tiap nodes pada file input pengujian ini banyak yang bertetangga sehingga pencaarian BFS akan mengecek tiap cabang nodes dan membangkitkan nodes yang cukup banyak. Sedangkan pada DFS, karena digunakan prioritas U - R - D - L, algoritma ini dapat mencari rute optimal pada file input keempat.

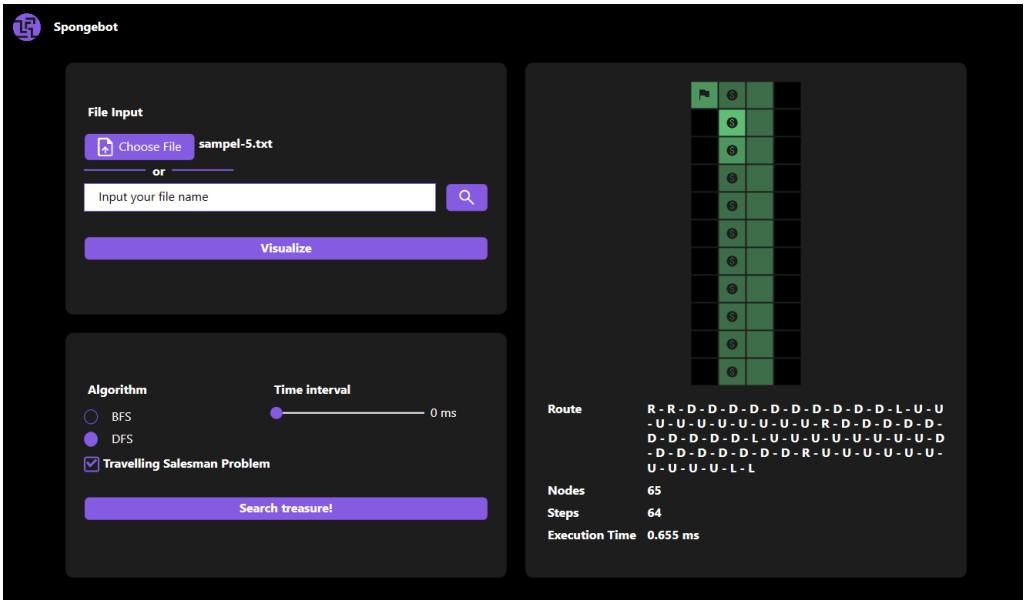
### 4.4.5. Pengujian 5

Berikut adalah file yang digunakan untuk pengujian keempat.


*Gambar 4.4.5.1 File Maze*




*Gambar 4.4.5.2 Solusi BFS TSP*




*Gambar 4.4.5.3 Solusi DFS TSP*

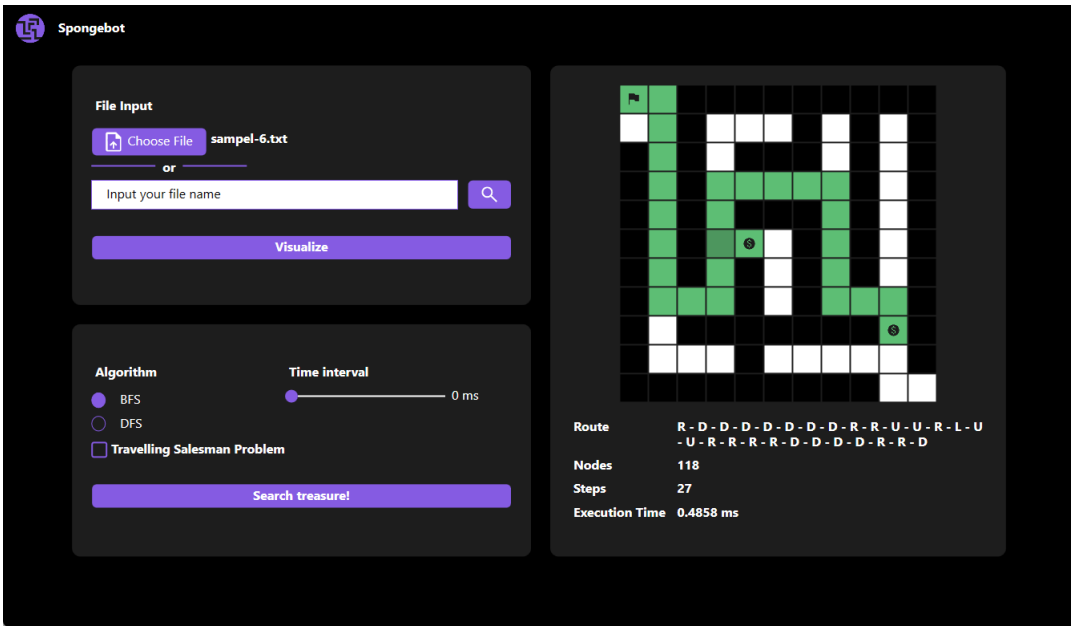
Pada pengujian kelima, dilakukan pengujian untuk algoritma BFS dan DFS dengan TSP tanpa time interval. Pada pengujian kali ini, terdapat perbedaan signifikan nodes yang dikunjungi BFS dan DFS. Namun, rute yang dihasilkan DFS jauh lebih lama dibanding rute yang dihasilkan BFS. Hal ini juga memengaruhi execution time dikarenakan tampilan rute akhir termasuk dalam execution time.

### 4.4.6. Pengujian 6

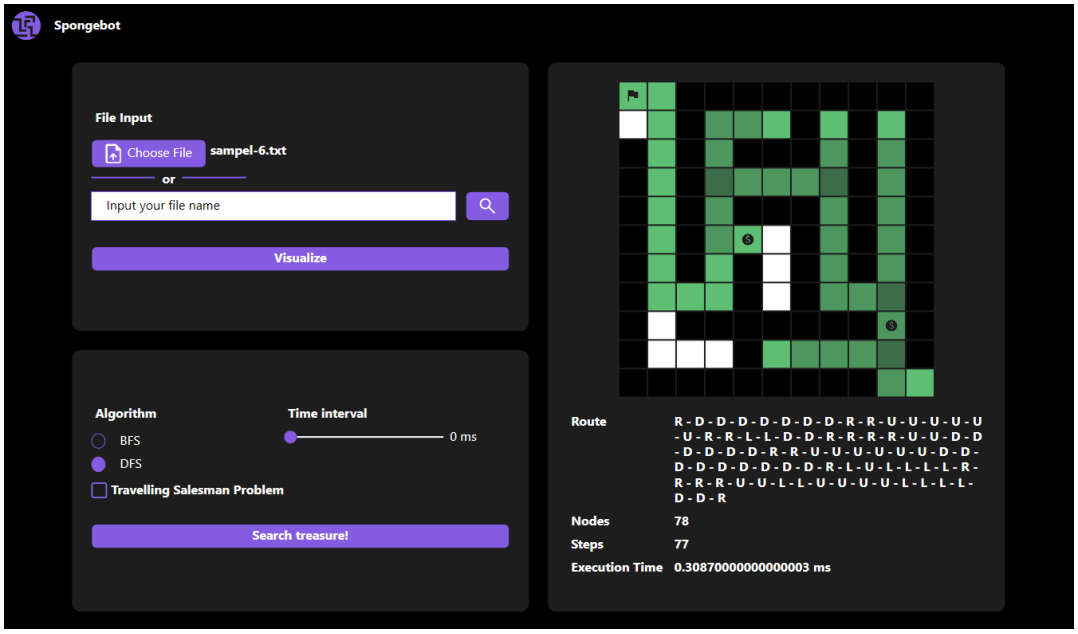
Berikut adalah file yang digunakan untuk pengujian keempat.

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

*Gambar 4.4.6.1 File Maze*




*Gambar 4.4.6.2 Solusi BFS Non-TSP*



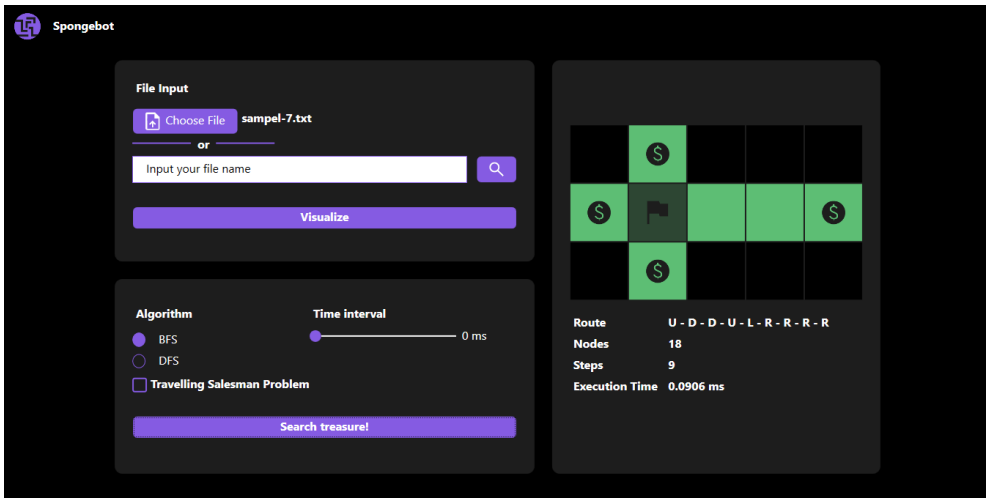

*Gambar 4.4.6.3 Solusi DFS Non-TSP*

Pada pengujian keenam, dilakukan pengujian algoritma BFS dan DFS tanpa TSP dan time interval. Pada pengujian kali ini, nodes yang dibangkitkan BFS lebih banyak dibanding DFS. Namun, dapat dilihat pada rute akhir, rute yang dihasilkan DFS jauh lebih tidak optimal dibanding yang dihasilkan BFS.

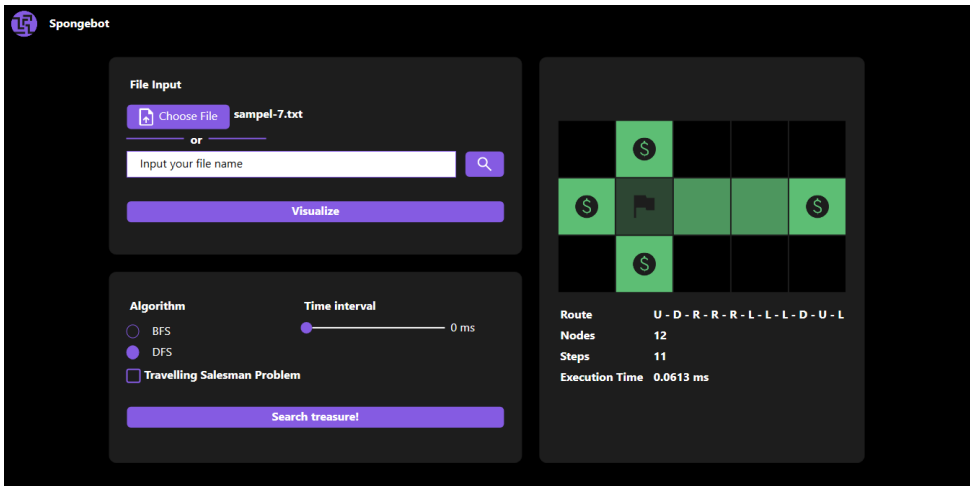
### 4.4.7. Pengujian 7

Berikut adalah file masukan yang digunakan untuk pengujian ketujuh.


*Gambar 4.4.7.1 File Maze*




*Gambar 4.4.7.2 Solusi BFS Non-TSP*



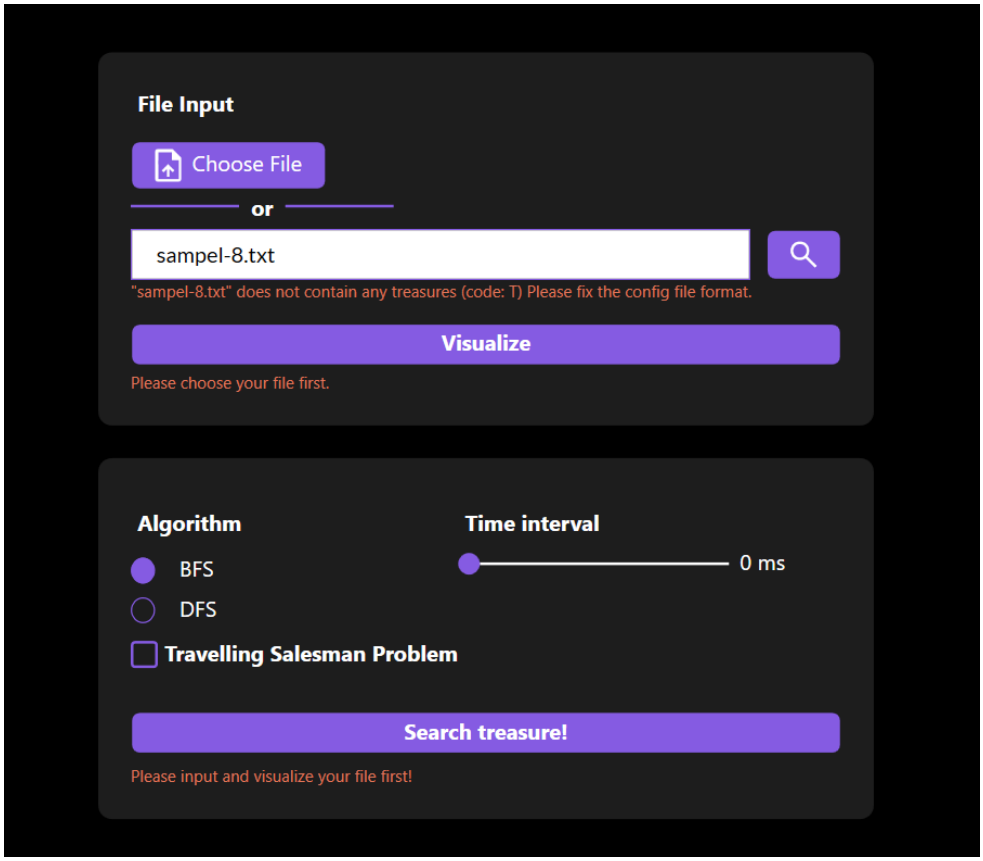

*Gambar 4.4.7.3 Solusi DFSNon-TSP*

Pada pengujian ketujuh, dilakukan pengujian algoritma BFS dan DFS tanpa TSP dan Time Interval. Pada file input kali ini, kotak start akan banyak dilalui karena posisi treasure yang hanya bisa dicapai melalui kotak start. Dari pengujian ini dpat dilihat pula perbedaan algoritma BFS dan DFS dalam mencari rute. Rute yang ditemukan BFS akan lebih efisien walaupun nodes yang dibangkitkan lebih banyak.

### 4.4.8. Pengujian 8

Berikut adalah file yang digunakan untuk pengujian kedelapan.


*Gambar 4.4.8.1 File Maze*




*Gambar 4.4.8.2 Tampilan Error GUI*

Pada pengujian kali ini, input file tidak valid karena file tidak mengandung treasure. Sama seperti pengujian ketiga, akan ditampilkan pesan error pada program. Pesan error juga akan tampil ketika tombol Visualize atau Search treasure! ditekan.

## 4.5. Analisis Desain Solusi

Secara keseluruhan, desain solusi yang diimplementasikan sesuai dengan karakteristik masing-masing algoritma. Berdasarkan landasan teori dan seluruh hasil pengujian yang telah dilakukan, dapat diamati karakteristik dari algoritma BFS dan DFS dalam mencari rute. Algoritma BFS yang membangkitkan nodes tiap cabang akan cenderung memeriksa nodes dalam jumlah yang banyak. Nodes yang dibangkitkan jumlahnya makin masif ketika banyak ketetanggaan antar nodes pada maze yang diberikan. Karakteristik ini akan menyebabkan algoritma BFS selalu menuju ke treasure yang paling dekat dengan titik start dan paling sesuai

dengan prioritas. Pendekatan heuristik yang digunakan pada BFS menganggap treasure yang sudah ditemukan sebagai titik start baru untuk mencari treasure lainnya. Pada beberapa kasus khusus, pendekatan ini akan menghasilkan rute yang tidak paling optimal. Namun, berdasarkan pengujian, hasil rute yang didapatkan jauh lebih efektif dan optimal daripada rute pada DFS.

Algoritma DFS akan menelusuri suatu nodes hingga dead-end baru diikuti dengan backtracking dan menuju nodes pada cabang yang berbeda. Karakteristik utama yang terlihat dari DFS yaitu rute akhir yang dihasilkan sama dengan tahap pencarian yang dilakukan oleh algoritma itu sendiri. Pada kebanyakan kasus, hal ini membuat nodes yang dibangkitkan oleh algoritma DFS tidak sebanyak pada algoritma BFS. Seberapa optimal rute yang dihasilkan oleh DFS akan sangat bergantung dengan prioritas yang digunakan dan maze yang sedang ditelusuri. Namun, secara keseluruhan, rute yang dihasilkan DFS menjadi lebih tidak optimal dibandingkan hasil algoritma BFS. Secara khusus, dapat dilihat rute DFS menjadi sangat tidak optimal pada maze yang berbentuk seperti “labirin” seperti halnya pada pengujian keenam. DFS memiliki keunggulan dalam efektifitas ruang.

Dari kedua karakteristik algoritma yang diterapkan, dapat dilihat bahwa masing-masing algoritma memiliki kelebihan dan kekurangannya masing-masing. Algoritma BFS akan menghasilkan rute yang optimal tetapi melalui proses yang lebih memakan waktu dan memori. Sedangkan algoritma DFS akan lebih optimal secara waktu dan memori, tetapi menghasilkan rute yang lebih tidak optimal.

# BAB 5
# PENUTUP

## 5.1. Kesimpulan

Dalam tugas besar IF2211 Strategi Algoritma ini, kami berhasil membuat aplikasi desktop yang mengaplikasikan algoritma BFS dan DFS dalam menyelesaikan persoalan pencarian titik/treasure, serta berhasil mengimplementasikan variasi solusi yang menemukan rute untuk kembali ke posisi awal (Travelling Salesman Problem). Program desktop yang kami buat memiliki tampilan yang interaktif dan mudah digunakan, serta berhasil menampilkan rute yang dihasilkan, serta langkah-langkah yang dilakukan untuk menemukan rute tersebut.

Dari pengerjaan tugas ini, kami mendapatkan beberapa kesimpulan, yaitu

- Algoritma BFS dapat menemukan rute optimal/terpendek dari suatu persoalan yang dapat direpresentasikan sebagai persoalan graph tidak berbobot. Akan tetapi, algoritma BFS cenderung menggunakan lebih banyak waktu dan memori sehingga perlu dilakukan optimasi agar pencarian tidak membangkitkan simpul yang terlalu banyak dan memakan waktu yang terlalu banyak.
- Algoritma DFS dapat menemukan rute dengan pencarian ruang seminimal mungkin dari suatu persoalan. Akan tetapi algoritma DFS cenderung menghasilkan rute yang tidak optimal karena hanya berfokus pada ruang yang sedikit.

## 5.2. Saran

Beberapa saran yang kami dapatkan dari pengerjaan pengembangan aplikasi ini adalah sebagai berikut,

1. Perlu dilakukan pendalaman dan pemahaman terlebih dahulu terhadap penggunaan bahasa pemrograman C#, IDE visual studio, framework WPF, serta tools-tools lain yang digunakan dalam pengembangan program ini.
2. Perlunya perencanaan terlebih dahulu mengenai struktur dari program dan kelas-kelas apa saja yang akan digunakan untuk menyelesaikan persoalan.
3. Perlu perencanaan terlebih dahulu mengenai algoritma BFS dan DFS yang ingin dikembangkan dapat menyelesaikan persoalan dengan optimal.

## 5.3. Refleksi

*Gambar 5.3 Refleksi Diri*

Dalam pengerjaan tugas ini sebaiknya dikerjakan dengan perancangan yang matang agar tidak merubah alur kode di tengah-tengah pengerjaan.

# DAFTAR PUSTAKA

Munir, R., MT., & Ulfa Maulidevi, N., S. T, M. Sc. (n.d.). *Breadth/Depth First Search (BFS/DFS) (Bagian 2)* [Slide show]. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf

Munir, R., & Ulfa Maulidevi, N., S. T, M. Sc. (n.d.). *Breadth/Depth First Search (BFS/DFS) (Bagian 1)* [Slide show]. https://informatika.stei.itb.ac.id. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf

*Pengaplikasian Algoritma BFS dan DFS dalam Menyelesaikan Persoalan Maze Treasure Hunt*. (n.d.). https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2022-2023/Tubes2-Stima-2023.pdf

# LAMPIRAN

## LINK REPOSITORY

Link repository GitHub : https://github.com/liviaarumsari/Tubes2_Spongebot

Link Youtube : https://youtu.be/PDnTwT-rfRI

## PEMBAGIAN TUGAS

| NIM | Nama | Tugas |
|---|---|---|
| 13521094 | Angela Livia Arumsari | GUI, Laporan |
| 13521100 | Alexander Jason | Algoritma DFS, TSP, Laporan |
| 13521134 | Rinaldy Adin | Algoritma BFS, TSP, Laporan |