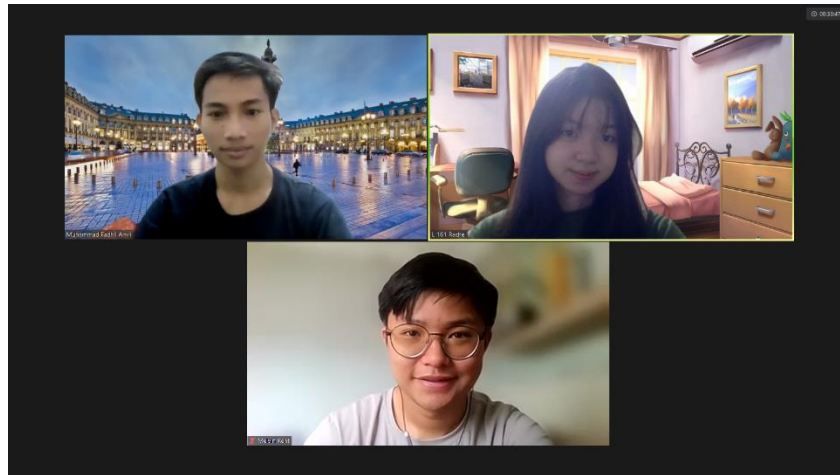


**LAPORAN TUGAS BESAR 2**  
**IF2211 STRATEGI ALGORITMA**

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



**KELOMPOK GOLD RUSH**

**ANGGOTA :**

- 1. 13521044 RACHEL GABRIELA CHEN**
- 2. 13521052 MELVIN KENT JONATHAN**
- 3. 13521066 MUHAMMAD FADHIL AMRI**

**PROGRAM STUDI TEKNIK INFORMATIKA**  
**SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA**  
**INSTITUT TEKNOLOGI BANDUNG**

**2023**

## BAB I

### 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.



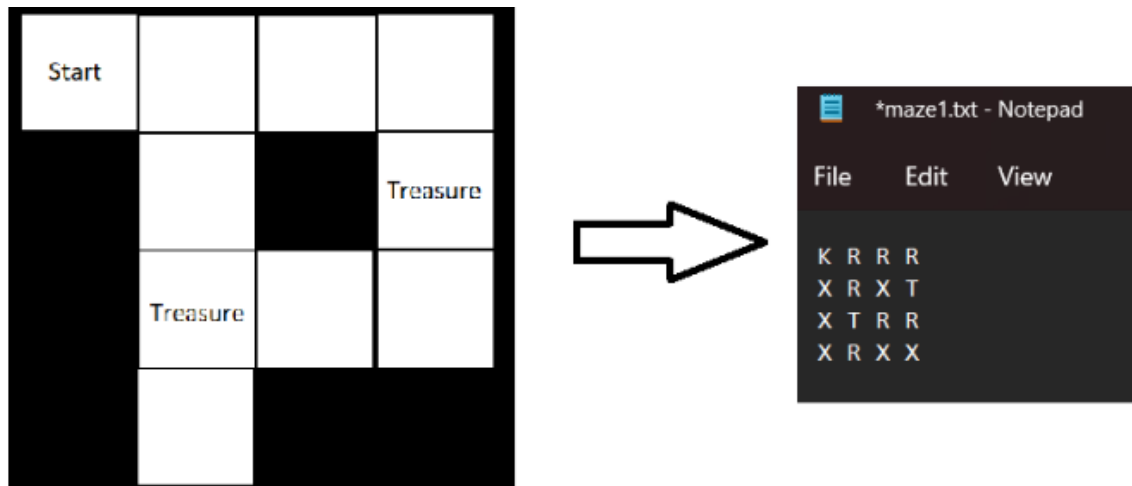
*Gambar 1. Labirin di Bawah Krusty Krab*

(Sumber: [https://static.wikia.nocookie.net/theloudhouse/images/e/ec/Massive\\_Mustard\\_Pocket.png/revision/latest?cb=20180826170029](https://static.wikia.nocookie.net/theloudhouse/images/e/ec/Massive_Mustard_Pocket.png/revision/latest?cb=20180826170029))

Permasalahan ini dapat diselesaikan dengan mengimplementasikan BFS dan DFS dalam sebuah program dengan GUI 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. Batasan dari input maze cukup berbentuk segi-empat dengan spesifikasi simbol sebagai berikut :

1. K : Krusty Krab (Titik awal)
2. T : Treasure
3. R : Grid yang mungkin diakses / sebuah lintasan
4. X : Grid halangan yang tidak dapat diakses

Contoh file input :



Gambar 2. Contoh file input

Dengan memanfaatkan algoritma Breadth First Search (BFS) dan Depth First Search (DFS), grid (simpul) yang mungkin dikunjungi hingga ditemukan rute solusi dapat ditelusuri baik secara melebar ataupun mendalam bergantung alternatif algoritma yang dipilih. Rute solusi adalah rute yang memperoleh seluruh treasure pada maze.

## BAB II

### LANDASAN TEORI

#### 2.1 *Graph Traversal*

*Graph traversal* merujuk pada proses mengunjungi setiap simpul atau *node* di sebuah struktur data graf tepat sekali. *Graph traversal* merupakan operasi dasar dalam struktur data graf dan digunakan untuk menyelesaikan banyak masalah, seperti mencari jalur terpendek antara dua simpul, mendeteksi siklus, dan mencari pola dalam data set besar.

Ada dua pendekatan utama dalam *graph traversal*: pencarian dengan *depth-first search* (DFS) dan *breadth-first search* (BFS).

#### 2.2 Depth-First Search (DFS)

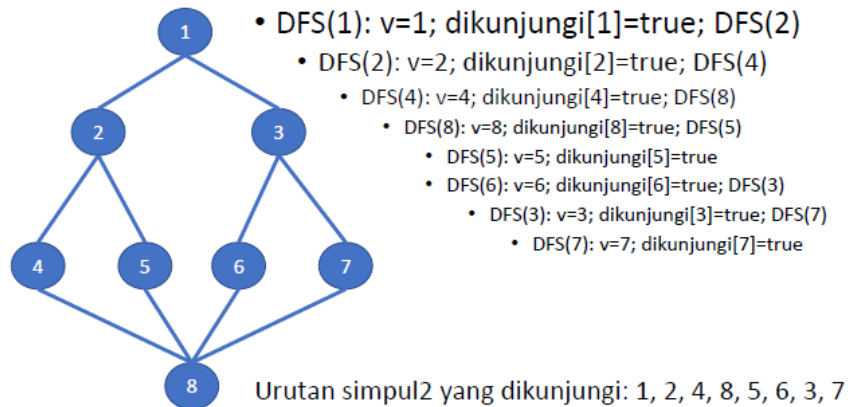
Depth-first search (DFS) merupakan salah satu algoritma *graph traversal*. Algoritma ini merupakan algoritma pencarian mendalam yang dimulai dari sebuah simpul (*node*) awal dilanjutkan dengan hanya mengunjungi *node* yang bertetangga dengan *node* awal tersebut sesuai dengan prioritas tertentu hingga tidak ada lagi *node* yang bisa dikunjungi.

Secara umum, algoritma DFS adalah sebagai berikut:

Misalkan traversal dimulai dari simpul,

1. Kunjungi simpul *v*,
2. Kunjungi simpul *w* yang bertetangga dengan simpul *v*,
3. Ulangi DFS dimulai dari simpul *w*,
4. Ketika mencapai simpul *u* sedemikian sehingga semua simpul yang bertetangga dengannya telah dikunjungi, pencarian dirunut balik ( *backtrack* ) ke simpul terakhir yang dikunjungi sebelumnya dan mempunyai simpul *w* yang belum dikunjungi.
5. Pencarian berakhir bila tidak ada lagi simpul yang belum dikunjungi yang dapat dicapai dari simpul yang telah dikunjungi.

Berikut ini adalah contoh ilustrasi dari algoritma DFS:



Gambar 3. Ilustrasi DFS

### 2.3 Breadth-First Search (BFS)

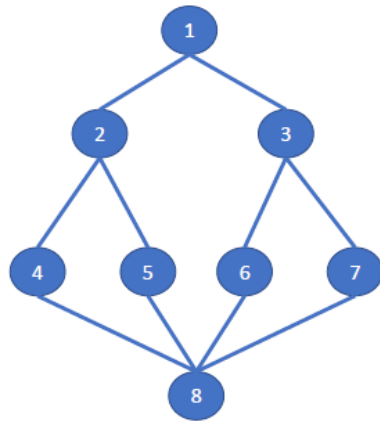
Breadth-First Search (BFS) adalah algoritma *graph traversal* yang umumnya digunakan untuk mencari jalur terpendek antara dua node atau untuk mencari komponen terhubung pada sebuah graf. Algoritma ini dimulai dengan mengunjungi sebuah node awal, kemudian mengunjungi setiap node yang terhubung langsung dengan node tersebut, dan seterusnya mengunjungi setiap node pada jarak yang sama dari node awal.

Secara umum, algoritma DFS adalah sebagai berikut:

Misalkan traversal dimulai dari simpul,

1. Kunjungi simpul v
2. Kunjungi semua simpul yang bertetangga dengan simpul v terlebih dahulu.
3. Kunjungi simpul yang belum dikunjungi dan bertetangga dengan simpul-simpul yang tadi dikunjungi, demikian seterusnya.

Berikut adalah contoh ilustrasi dari algoritma BFS:



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

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

Gambar 4. Ilustrasi BFS

## 2.4 C# Desktop Application Development

C# desktop application development adalah pengembangan aplikasi desktop menggunakan bahasa pemrograman C#. C# adalah bahasa pemrograman yang populer untuk pengembangan aplikasi desktop karena memiliki sintaks yang mudah dipahami dan kuat untuk mengakses fitur-fitur sistem operasi.

Terdapat beberapa tools yang dapat memudahkan C# Desktop Application Development, seperti Microsoft Visual Studio yang menyediakan lingkungan pengembangan yang lengkap untuk mempermudah pembuatan aplikasi desktop.

Dalam pengembangan aplikasi desktop menggunakan C#, terdapat *framework* yang dapat digunakan untuk memudahkan pengembangan, seperti .NET. .NET adalah *framework* perangkat lunak yang dikembangkan oleh Microsoft yang digunakan untuk memudahkan pengembangan aplikasi dengan menggunakan berbagai bahasa pemrograman, termasuk C#. .NET dapat digunakan dengan WinForms atau WPF untuk memudahkan pengembangan GUI aplikasi.

## BAB III

### APLIKASI ALGORITMA BFS DAN DFS

#### 3.1 Langkah - Langkah Pemecahan Masalah

Berikut adalah langkah-langkah pemecahan masalah dengan algoritma DFS.

1. Tentukan *node* awal pencarian *treasure*('K') dan semua *node* ('R' dan 'T') yang terdapat pada *maze*.
2. *Push node* awal ke dalam *stack* yang merepresentasikan urutan pemeriksaan *node*.
3. Evaluasi *node* yang sedang diperiksa, yaitu *node* yang berada pada posisi *top* dari *stack*.
4. Jika *node* yang diperiksa adalah *treasure*, pencatatan *treasure* yang tersisa akan berkurang.
5. Lakukan pengecekan untuk *node* tetangga yang belum dikunjungi dari *node* yang sedang diperiksa.
6. *Push* semua *node* tetangga yang belum diperiksa ke dalam *stack*.
7. Jika semua *node* tetangga sudah diperiksa, atau tidak memiliki *node* tetangga. Lakukan *backtracking* dengan *pop node* yang sedang diperiksa dari *stack*.
8. Ulangi dari langkah ke-3 hingga semua *treasure* telah diperiksa (*treasure* yang tersisa 0).

Berikut adalah langkah-langkah pemecahan masalah dengan algoritma TSP-DFS.

1. Terapkan algoritma DFS untuk mencari semua *treasure* pada *maze*.
2. Evaluasi *node* yang sedang diperiksa, yaitu *node* yang berada pada posisi *top* dari *stack*.
3. Jika *node* yang diperiksa adalah *node* awal, TSP selesai.
4. Jika *node* yang diperiksa bukan *node* awal, lakukan pengecekan untuk *node* tetangga yang belum dikunjungi dari *node* yang sedang diperiksa.
5. *Push* semua *node* tetangga yang belum diperiksa ke dalam *stack*.
6. Jika semua *node* tetangga sudah diperiksa, atau tidak memiliki *node* tetangga. Lakukan *backtracking* dengan *pop node* yang sedang diperiksa dari *stack*.
7. Ulangi dari langkah ke-2 *node* awal diperiksa.

Berikut adalah langkah-langkah pemecahan masalah dengan algoritma BFS:



1. Inisialisasi seluruh petak peta menjadi sebuah *node* yang memiliki keterhubungan sesuai pada peta.
2. Tentukan *node* awal yang menjadi titik dimulainya pencarian. *Enqueue node* tersebut ke dalam *queue*.
3. *Dequeue node* paling awal dari *queue* dan cek apakah *node* tersebut merupakan *treasure*.
  - Apabila bukan, *enqueue node-node* tetangganya dengan prioritas mulai dari tetanggan kanan, bawah, kiri, atas. Pastikan bahwa *node* tetangga masih dalam status *unvisited*. Tandai pula *node* yang sedang diperiksa sebagai *visited*. Catat pula langkah-langkah yang telah dilalui dalam mencapai *node* tetangga tersebut.
  - Apabila iya, tambahkan *sequence* langkah yang telah dilalui ke pencatatan langkah solusi sebelumnya. Kemudian *reset* seluruh catatan *visited node*. Lanjutkan proses pencarian *treasure* berikutnya dengan dimulai dari *node treasure* terakhir.
4. Ulangi langkah ketiga hingga tidak ada lagi *treasure* yang tersisa.
5. Langkah – langkah yang dibutuhkan untuk memperoleh seluruh *treasure* yang ada pun dapat diperoleh setelah iterasi telah selesai.

Berikut adalah langkah-langkah pemecahan masalah dengan algoritma TSP-BFS:

1. Inisialisasi seluruh petak peta menjadi sebuah *node* yang memiliki keterhubungan sesuai pada peta.
2. Terapkan algoritma BFS untuk mencari *path* menuju seluruh *treasure* yang ada.
3. Tetapkan petak *treasure* terakhir menjadi *node* awal dimulainya pencarian langkah kembali menuju *start node*. *Enqueue node* tersebut ke dalam *queue*.
4. *Dequeue node* paling awal dari *queue* dan cek apakah *node* tersebut merupakan *start node*.
  - Apabila bukan, *enqueue node-node* tetangganya dengan prioritas mulai dari tetanggan kanan, bawah, kiri, atas. Pastikan bahwa *node* tetangga masih dalam status *unvisited*. Tandai pula *node* yang sedang diperiksa sebagai *visited*. Catat pula langkah-langkah yang telah dilalui dalam mencapai *node* tetangga tersebut.
  - Apabila iya, tambahkan *sequence* langkah yang telah dilalui ke pencatatan langkah solusi yang telah diperoleh sebelumnya dari algoritma BFS untuk memperoleh seluruh *treasure*.

5. Langkah – langkah yang dibutuhkan untuk memperoleh seluruh *treasure* yang ada serta kembali ke petak awal pun dapat diperoleh setelah iterasi telah selesai.

### 3.2 Mapping Persoalan

Berikut *Mapping* persoalan pada DFS dan TSP-DFS.

1. *GraphNode* merepresentasikan setiap *node* dari graf. Atribut *visited* digunakan untuk menghitung berapa kali *node* tersebut dikunjungi, atribut *treasure* digunakan untuk menandakan apakah *node* adalah *treasure node* atau tidak. Setiap *node* juga menyimpan informasi dari tetangganya, yaitu tetangga kanan, bawah, kiri, dan atas. Tetangga bernilai *null* jika tidak ada.
2. Graf pada *maze* direpresentasikan dengan list of *GraphNode* yang merupakan sebuah *linked list*. *GraphNode* yang termasuk ke dalam graf adalah *node-node* yang merupakan representasi dari *tile* yang ada pada map yang bisa dikunjungi ('K', 'R', 'T').
3. *Stack of Route* digunakan untuk merepresentasikan urutan pemeriksaan *node*. Urutan pemeriksaan yang digunakan adalah LIFO(*Last In First Out*). *Stack* ini adalah bentuk implementasi dari algoritma DFS yang digunakan. Dengan *stack* ini juga algoritma DFS yang digunakan bisa melakukan *backtracking* saat sudah buntu.

Bentuk *Mapping* persoalan pada BFS dan TSP-BFS.

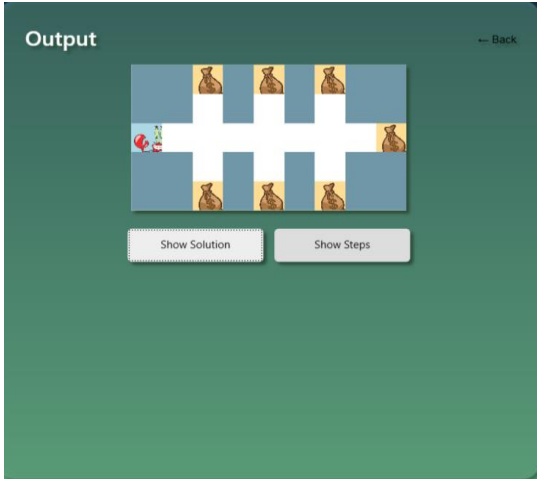
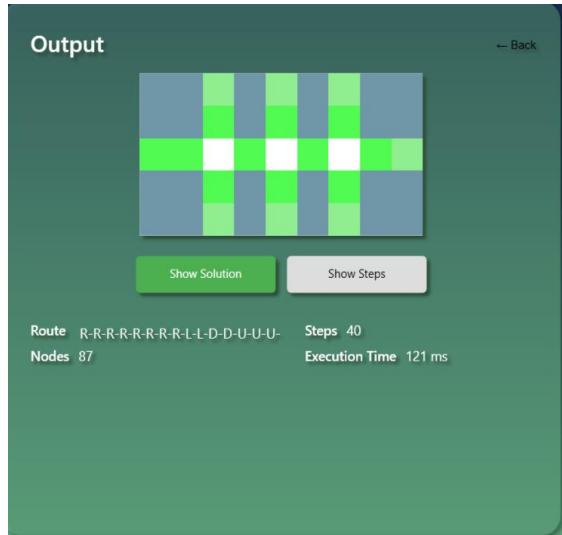
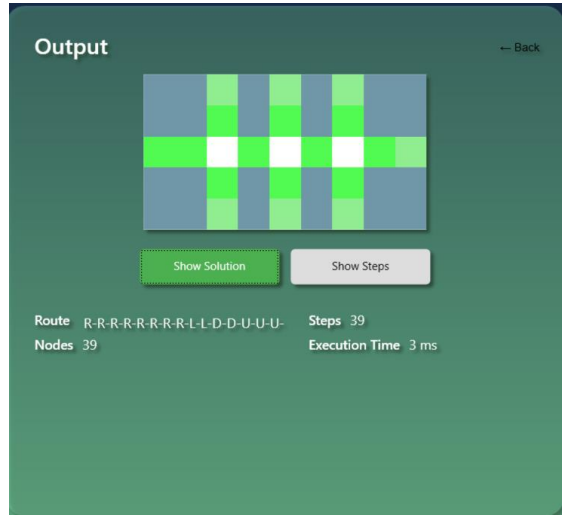
1. *GraphNode* merepresentasikan setiap *node* dari graf. Atribut *visited* digunakan untuk menghitung berapa kali *node* tersebut dikunjungi, atribut *treasure* digunakan untuk menandakan apakah *node* adalah *treasure node* atau tidak. Setiap *node* juga menyimpan informasi dari tetangganya, yaitu tetangga kanan, bawah, kiri, dan atas. Tetangga bernilai *null* jika tidak ada.
2. *ElementQueue* merupakan kelas bagi objek yang menjadi elemen-elemen dari *queue* pada algoritma BFS. *ElementQueue* mengandung atribut *route* yang memiliki kelas *Route* dan juga *visitedNodes* yang merupakan objek dengan kelas *Matrix*. Setiap percabangan yang ada pada *path* akan membentuk *ElementQueue* baru, yang berarti setiap percabangan/variasi *path* memiliki *route* dan *visitedNodes* yang terlepas dari satu sama lain. Pencatatan *visitedNodes* tidak dilakukan secara global selama iterasi karena persoalan ini mengimplikasikan bahwa langkah-langkah menuju tujuan

merupakan bagian dari solusi, sehingga pencatatan visitedNodes harus dilakukan untuk masing-masing percabangan.

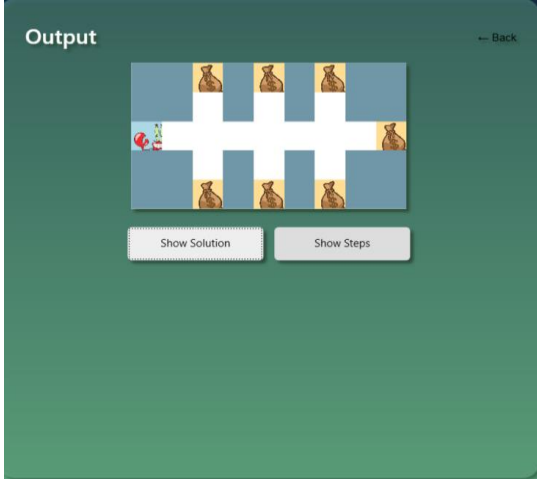
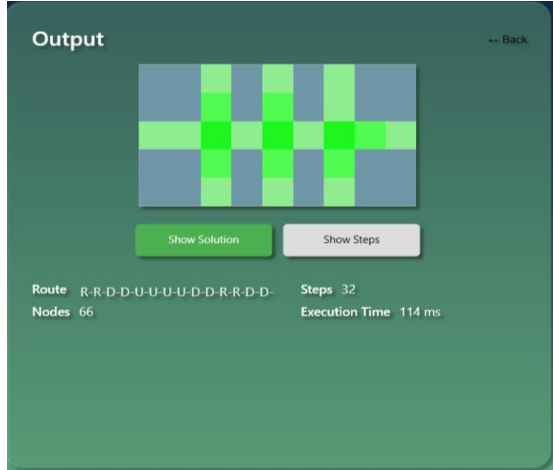
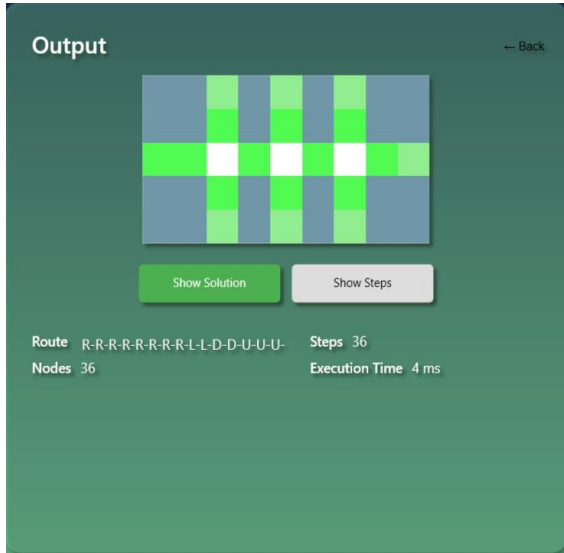
3. *Queue of ElementQueue* digunakan untuk merepresentasikan *container dengan* urutan pemeriksaan *node* dengan aturan FIFO (*First In First Out*).

### 3.3 Contoh Ilustrasi Kasus Lain

#### Test 3 (Dengan TSP)

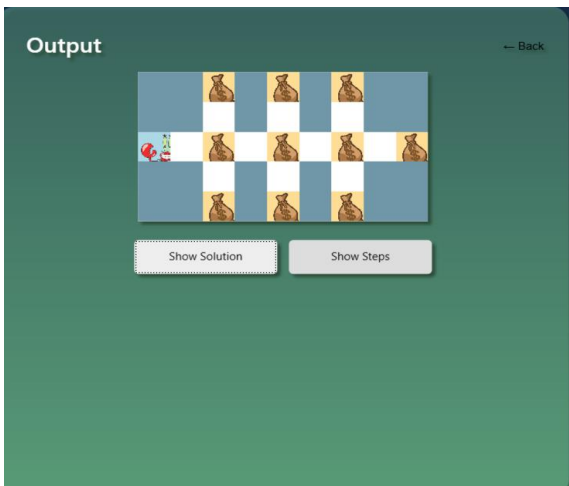
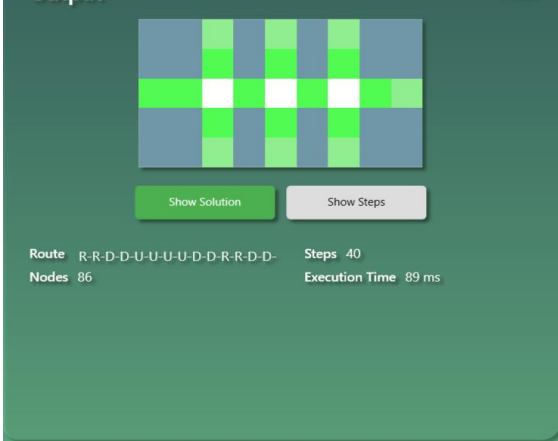
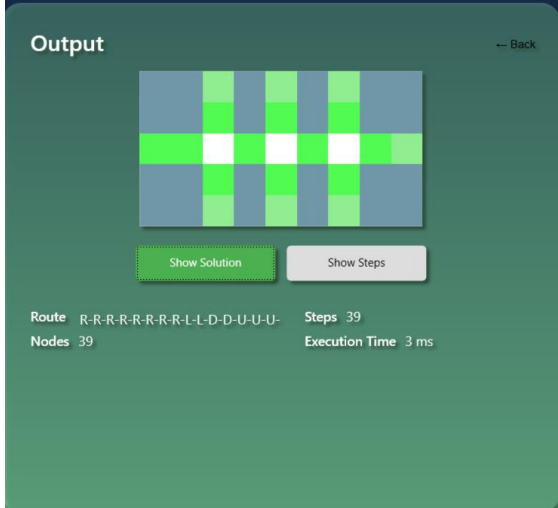
Input	Hasil
	<b>BFS</b>
	
	<b>DFS</b>
	

**Test 3 (Tanpa TSP)**

Input	Hasil
	<b>BFS</b>
	 <p>Route R-R-D-D-U-U-U-U-D-D-R-R-D-D- Nodes 66 Steps 32 Execution Time 114 ms</p>
	<b>DFS</b>
	 <p>Route R-R-R-R-R-R-R-L-L-D-D-U-U-U- Nodes 36 Steps 36 Execution Time 4 ms</p>

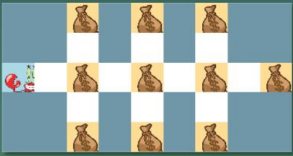

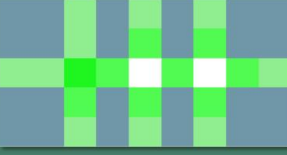
**Test 4 (Dengan TSP)**

Input	Hasil
	<b>BFS</b>

	 <p>Route R-R-D-D-U-U-U-U-D-D-R-R-D-D- Steps 40 Nodes 86 Execution Time 89 ms</p>
	<p><b>DFS</b></p>  <p>Route R-R-R-R-R-R-R-L-L-D-D-U-U-U- Steps 39 Nodes 39 Execution Time 3 ms</p>

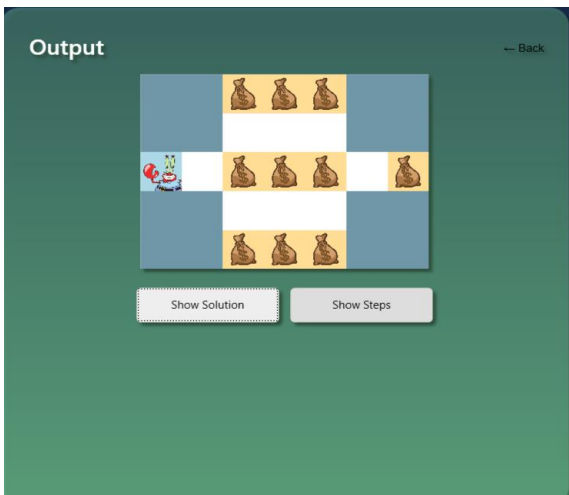
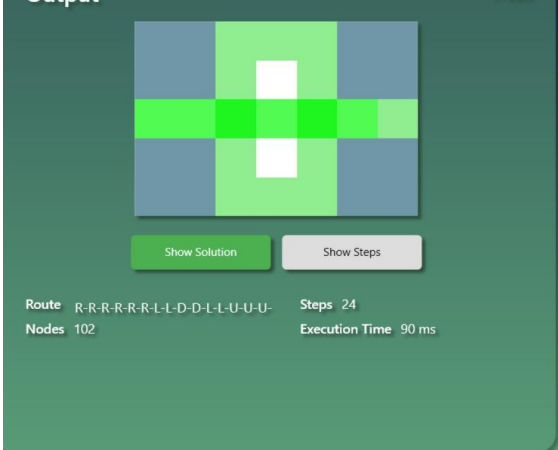

**Test 4 (Tanpa TSP)**

Input	Hasil
	<b>BFS</b>

<p>Output <span>← Back</span></p>  <p>Show Solution Show Steps</p>	<p>Output <span>← Back</span></p>  <p>Show Solution Show Steps</p> <p>Route R-R-R-R-R-R-R-L-L-D-D-U-U-U- Steps 36 Nodes 78 Execution Time 114 ms</p>
	<p><b>DFS</b></p> <p>Output <span>← Back</span></p>  <p>Show Solution Show Steps</p> <p>Route R-R-R-R-R-R-R-L-L-D-D-U-U-U- Steps 35 Nodes 35 Execution Time 3 ms</p>



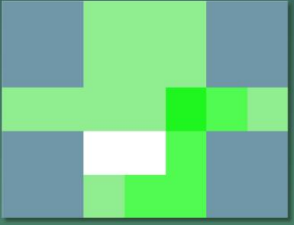
**Test 5 (Dengan TSP)**

Input	Hasil
	<b>BFS</b>

	 <p>Route R-R-R-R-R-L-L-D-D-L-L-U-U-U- Nodes 102 Steps 24 Execution Time 90 ms</p>
	<p><b>DFS</b></p>  <p>Route R-R-R-R-R-L-L-D-D-L-L-R-R-U-L Nodes 32 Steps 32 Execution Time 9 ms</p>

**Test 5 (Tanpa TSP)**

Input	Hasil
	<b>BFS</b>

<p><b>Output</b> <span>← Back</span></p>  <p><span>Show Solution</span> <span>Show Steps</span></p>	<p><b>Output</b> <span>← Back</span></p>  <p><span>Show Solution</span> <span>Show Steps</span></p> <p>Route R-R-R-R-R-L-L-D-D-L-L-U-U- Steps 18 Nodes 47 Execution Time 80 ms</p>
	<p><b>DFS</b></p> <p><b>Output</b> <span>← Back</span></p>  <p><span>Show Solution</span> <span>Show Steps</span></p> <p>Route R-R-R-R-R-L-L-D-D-L-L-R-R-U-L Steps 25 Nodes 25 Execution Time 0 ms</p>

## BAB IV

### IMPLEMENTASI DAN PENGUJIAN

#### 4.1 Repositori Github

[https://github.com/Mehmed13/Tubes2\\_goldRush](https://github.com/Mehmed13/Tubes2_goldRush)

#### 4.2 Implementasi dalam *Pseudocode*



```

{Main Program}
{DFS Algorithm}
procedure cekTetangga(input node:GraphNode, path: list of char, nodePath: listOfGraphNode,
remainingTreasure: int)
    KAMUS LOKAL
    ALGORITMA
    if (tetangga atas != null) then
        if(tetangga atas is not visited) then
            inisialisasi route dengan path, nodePath, remainingTreasure yang
            dipass
            push stack
    if (tetangga kiri != null) then
        if(tetangga kiri is not visited) then
            inisialisasi route dengan path, nodePath, remainingTreasure yang
            dipass
            push stack
    if (tetangga bawah != null) then
        if(tetangga bawah is not visited) then
            inisialisasi route dengan path, nodePath, remainingTreasure yang
            dipass
            push stack
    if (tetangga kanan != null) then
        if(tetangga kanan is not visited) then
            inisialisasi route dengan path, nodePath, remainingTreasure yang
            dipass
            push stack

procedure runDFSAlgorithm()
    KAMUS LOKAL
    backtracking, stopbacktracking, multiplevisited: bool
    remainingTreasures: int

    ALGORITMA
    inisialisasi route pertama dan variabel control
    push route awal ke dalam stack
    cekTetangga(node, path, nodePath, remaining treasure)
    while (remainingTreasure != 0) do
        akses top stack
        if (node sudah pernah dikunjungi and not multiplevisited) then
            pop stack
            if (backtracking) then
                tambahkan path ke node sekarang
            else

```

```

        if (stopbacktracking) then
            tambahkan path ke node sekarang
            backtracking <- false
            stopbacktracking <- false
            multiplevisited <- true
    else
        if (backtracking) then
            pop stack
            wariskan route selain node ke top stack selanjutnya
        else
            {Evaluasi current node}
            multiplevisited <- false
            if (currentNode is treasure and belum pernah visited) then
                kurangi remaining treasure
                if (remaining treasure = 0) then
                    break
                if (currentNode memiliki tetangga yang bisa visited) then
                    cekTetangga(node,path,nodePath,remaining treasure)
            else
                wariskan route selain node ke top stack selanjutnya
    {remainingTreasure = 0}
    update path, nodepath, numvisited, dan time execution

```

{TSP-DFS Algorithm}

{Prekondisi: sudah dilakukan DFS terlebih dahulu}

procedure runTSPDFSAlgorithm()

**KAMUS LOKAL**

backtracking, backToStart, stopbacktracking, multiplevisited: bool  
remainingTreasures: int

**ALGORITMA**

```

inisialisasi route pertama dan variabel control
push route awal ke dalam stack
cekTetangga(node, path, nodePath, remaining treasure)
while (!backToStart) do
    akses top stack
    if (startNode) then
        backToStart <- true
    else
        if (node sudah pernah dikunjungi and not multiplevisited) then
            pop stack
            if (backtracking) then
                tambahkan path ke node sekarang

```

```

        else
            if (stopbacktracking) then
                tambahkan path ke node sekarang
                backtracking <- false
                stopbacktracking <- false
                multiplevisited <- true
        else
            if (backtracking) then
                pop stack
                wariskan route selain node ke top stack selanjutnya
            else
                {Evaluasi current node}
                multiplevisited <- false
                if (currentNode memiliki tetangga yang bisa visited) then
                    cekTetangga(node,path,nodePath,remaining treasure)
                else
                    wariskan route selain node ke top stack selanjutnya
    {backToStart}
    update path, nodepath, numvisited, dan time execution

```

```

{BFS                                                                    Algorithm}
procedure runBFSAlgorithm(input row: int, input col: int)
KAMUS LOKAL
    remainingTreasure, numOfNodesVisited : int
    found : bool
    headElement, newElementQueue : ElementQueue
    visitedNodeSequence : List of GraphNode

ALGORITMA
    startTime()
    {Inisialisasi start node dan remainingTreasure}

    while (remainingTreasure != 0 ) do
        {Inisialisasi first queue element}
        enqueue(newElementQueue)
        found <-- false
        while (!found) do
            headElement <-- dequeue()
            if (headElement = treasure) then
                found <-- true

```

```

{Append path ke atribut final path}
if (remainingTreasure = 1) then
visitedNodeSequence <-- visitedNodeSequence + headElement's last node
numOfNodesVisited <-- numOfNodesVisited + 1
repeat
{tambahkan 1 untuk jumlah visit setiap node di nodePath}
until (jumlah visit setiap node sudah ditambahkan)
else
repeat
{ tambahkan 1 untuk jumlah visit setiap node di nodePath}
until (jumlah visit setiap node kecuali node terakhir sudah ditambahkan)
remainingTreasure <-- remainingTreasure - 1
{clear queue}
else
visitedNodeSequence <-- visitedNodeSequence + headElement's last node
numOfNodesVisited <-- numOfNodesVisited + 1
{set headElement menjadi visited pada matrix}

{ LAKUKAN CODE DI BAWAH UNTUK SETIAP TETANGGA DENGAN URUTAN KANAN, BAWAH, KIRI, ATAS }
if (!found and tetangga!=NULL and tetangga belum visited) then
{Initialize newElementQueue}
{add char path sesuai dengan posisi tetangga}
enqueue(newElementQueue)

stoptime()
execution time <-- stop time - start time

{TSP-BFS Algorithm}
procedure runTSPBFSAlgorithm(input row: int, input col: int)
KAMUS LOKAL
numOfNodesVisited : int
found : bool
headElement, newElementQueue : ElementQueue
visitedNodeSequence : List of GraphNode

ALGORITMA
startTime()
{Inisialisasi start node}
RunBFSAlgorithm(row, col)
{Inisialisasi first queue element}
enqueue(newElementQueue)
found <-- false
while (!found) do

```

```

headElement <-- dequeue()
if (headElement = startNode) then
found <-- true
{Append path ke atribut final path}
if (remainingTreasure = 1) then
visitedNodeSequence <-- visitedNodeSequence + headElement's last node
numOfNodesVisited <-- numOfNodesVisited + 1
repeat
{tambahkan 1 untuk jumlah visit setiap node di nodePath}
until (jumlah visit setiap node sudah ditambahkan)
else
repeat
{ tambahkan 1 untuk jumlah visit setiap node di nodePath}
until (jumlah visit setiap node kecuali node terakhir sudah ditambahkan)
{clear queue}
else
visitedNodeSequence <-- visitedNodeSequence + headElement's last node
numOfNodesVisited <-- numOfNodesVisited + 1
{set headElement menjadi visited pada matrix}

{ LAKUKAN CODE DI BAWAH UNTUK SETIAP TETANGGA DENGAN URUTAN KANAN, BAWAH, KIRI, ATAS }
if (!found and tetangga!=NULL and tetangga belum visited) then
{Initialize newElementQueue}
{add char path sesuai dengan posisi tetangga}
enqueue(newElementQueue)

stoptime()
execution time <-- stop time - start time

```

### 4.3 Struktur Data Program dan Spesifikasi Program

Algoritma pada 4.2 diimplementasikan dengan bahasa pemrograman C# yang memanfaatkan *framework* .NET yang diintegrasikan dengan WPF untuk GUI. Kode untuk GUI dan program utama terdapat pada folder src. Sedangkan, struktur data yang digunakan terdapat pada folder lib. Struktur data yang digunakan antara lain:

- BFS.cs

BFS merupakan *class* yang digunakan untuk melakukan *graph traversal* dengan algoritma Breadth First Search (BFS). *Class* ini menyimpan atribut finalPath (rangkaian aksi yang merupakan solusi untuk mendapatkan semua *treasure*), numOfTreasures (jumlah *treasure* yang perlu dicari), numOfNodesVisited (jumlah simpul yang dikunjungi), executionTime

(lama waktu pemecahan masalah), *visitedNodeSequence* (urutan simpul yang diperiksa), *queue* (queue simpul-simpul yang akan diperiksa). Dalam *class* ini, diimplementasikan method *runBFSAlgorithm* yang menerima parameter jumlah baris dan kolom pada maze yang diperiksa untuk mencari solusi dengan algoritma BFS. Terdapat juga *runTSPBFSAlgorithm* untuk mencari solusi *path* untuk menemukan semua *treasure* kemudian kembali ke titik awal.

- *Coordinate.cs*

*Coordinate* merupakan *class* untuk menyimpan posisi *node* dalam matriks *maze*. Koordinat memiliki atribut *x* untuk indeks baris, dan *y* untuk indeks kolom.

- *DFS.cs*

*DFS* merupakan *class* yang digunakan untuk melakukan graph traversal dengan algoritma Depth First Search (DFS). *Class* ini menyimpan atribut *path* (rangkaian aksi yang merupakan solusi untuk mendapatkan semua treasure), *numOfTreasures* (jumlah treasure yang perlu dicari), *numOfNodesVisited* (jumlah simpul yang dikunjungi), *executionTime* (lama waktu pemecahan masalah), *graph* (graf yang menggambarkan keterhubungan simpul-simpul dalam permasalahan), *visitedNodeSequence* (urutan simpul yang diperiksa), *stack* (stack simpul-simpul yang akan diperiksa). Dalam *class* ini, diimplementasikan method *runDFSAlgorithm* untuk mencari solusi dengan algoritma DFS.

- *ElementQueue.cs*

*ElementQueue* merupakan *class* untuk elemen-elemen queue pada queue simpul yang akan dicek pada algoritma BFS. *Class* ini menyimpan atribut *route* yang merupakan objek dengan kelas *Route* dan juga *visitedNodes* yang merupakan objek dengan kelas *Matrix*. *route* dan *visitedNodes* dibuat berpasangan karena setiap elemen dari queue akan berisi 1 percabangan dari *path* (setiap percabangan akan menambah element dari queue) dan setiap cabang memiliki *list of visited nodes* yang berbeda dan khusus untuk cabang itu sendiri. *route* akan berperan dalam menyimpan *path* (List of char), *nodePath* (List of *GraphNode*), *node* (*GraphNode*). *visitedNodes* akan berperan dalam menyimpan koordinat-koordinat *node* yang sudah dikunjungi sebelumnya pada *path* pasangannya yang disimpan dalam bentuk Matriks dengan elemen biner.

- *GraphNode.cs*

*GraphNode* merupakan *class* yang merepresentasikan simpul pada *graph*. *Class* ini memiliki atribut *position* (posisi *node* pada matriks *maze*), *visited* (*integer* berapakah sebuah

*node* telah dikunjungi), *intersection*(boolean apakah *node* adalah simpang atau bukan), *right*, *down*, *left*, *up* (tetangga dari *node* pada arah yang bersesuaian).

- **Matrix.cs**

*Matrix* merupakan *class* yang merepresentasikan *matrix* dengan jumlah baris *row*, jumlah kolom *col*, dan elemen-elemen berupa *integer*.

- **Route.cs**

*Route* merupakan *class* yang merepresentasikan *state* dari suatu proses pemeriksaan *node*. *Class* ini memiliki atribut *path* yang merepresentasikan *path* dari *node* awal menuju *node* yang sedang diperiksa. Atribut *nodePath* merepresentasikan urutan *node* yang membentuk *path*. Atribut *node* yang merepresentasikan *node* yang sedang diperiksa, dan atribut *remaining treasure* yang merepresentasikan banyak *treasure* yang belum dikunjungi.

- **TSP.cs**

*TSP* merupakan *class* untuk menyelesaikan permasalahan pencarian *treasure* dan pencarian *path* untuk kembali ke posisi awal setelah mengumpulkan semua *treasure*. *Class* ini dikhususkan untuk pemecahan TSP dengan DFS. Atribut-attribut pada *class* ini, yaitu *path* (rangkaian aksi yang merupakan solusi untuk mendapatkan semua *treasure*), *numOfTreasures* (jumlah *treasure* yang perlu dicari), *numOfNodesVisited* (jumlah simpul yang dikunjungi), *executionTime* (lama waktu pemecahan masalah), *graph*(graf yang menggambarkan keterhubungan simpul-simpul dalam permasalahan), *visitedNodeSequence* (urutan simpul yang diperiksa), *stack* (stack simpul-simpul yang akan diperiksa). Dalam *class* ini, diimplementasikan *method* *runTSPDFSAlgorithm* yang mencari *path* untuk kembali ke titik awal.

- **Utility.cs**

*Utility.cs* merupakan *class* dengan *method-method* yang dibutuhkan untuk membantu pengembangan program. Dalam *utility*, diimplementasikan *method* untuk konversi dari file *.txt* ke *matrix* dan juga dari *matrix* ke *Graph*.

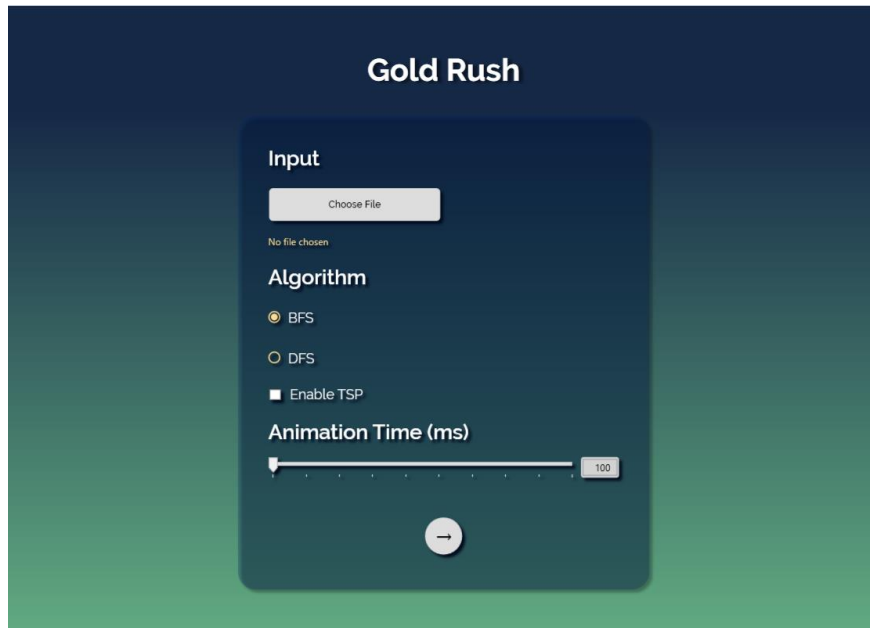
#### 4.4 Tata Cara Penggunaan Program

Program harus dijalankan pada sistem yang memiliki .NET CORE SDK versi 6.0 dan sistem operasi Windows 7.0 ke atas. Berikut adalah tata cara memulai program *GoldRush*:

1. Lakukan *git clone* repositori dari program ini dengan mengetik `git clone https://github.com/Mehmed13/Tubes2\_goldRush.git` pada terminal.
2. Buka folder *bin*.

3. Klik file GoldRush.exe atau ketik `./GoldRush.exe` pada terminal.

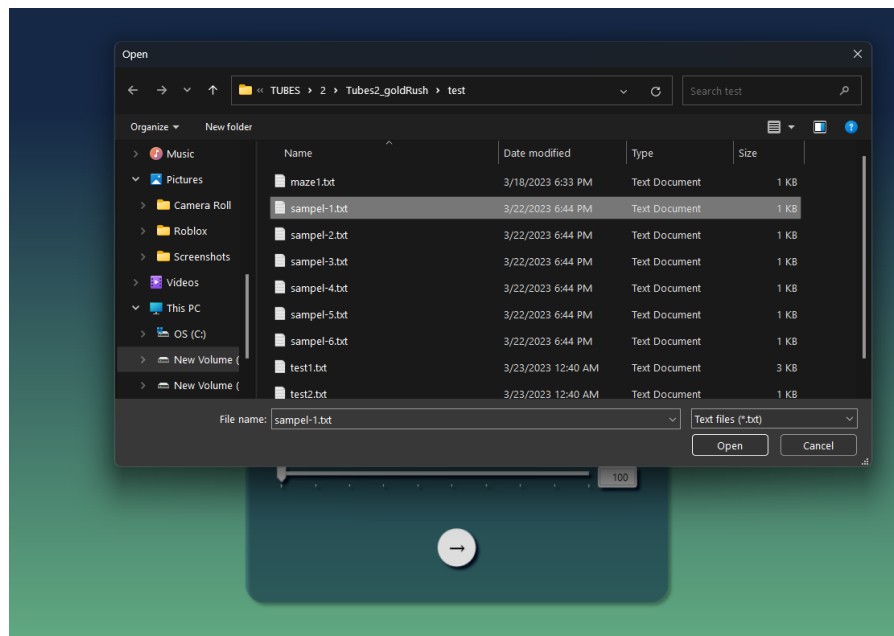
Berikut adalah tampilan program setelah di-run :



Gambar 5. Antarmuka GoldRush

Berikut adalah tata cara penggunaan program:

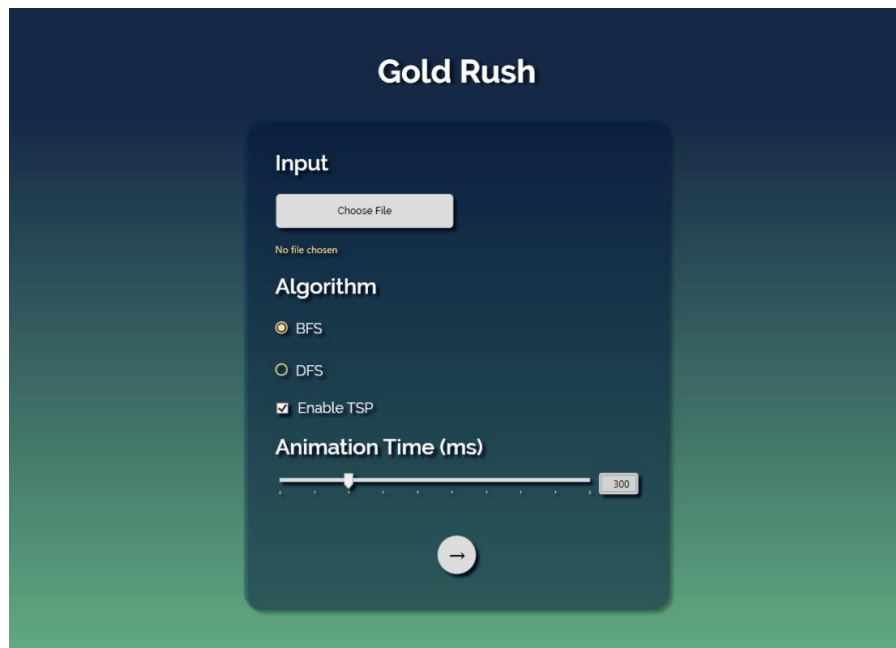
1. Klik “Choose File” lalu pilih file yang akan digunakan sebagai konfigurasi *maze*. File harus berupa file .txt dengan format yang sesuai dengan Bab 1.



Gambar 6. Pemilihan file .txt

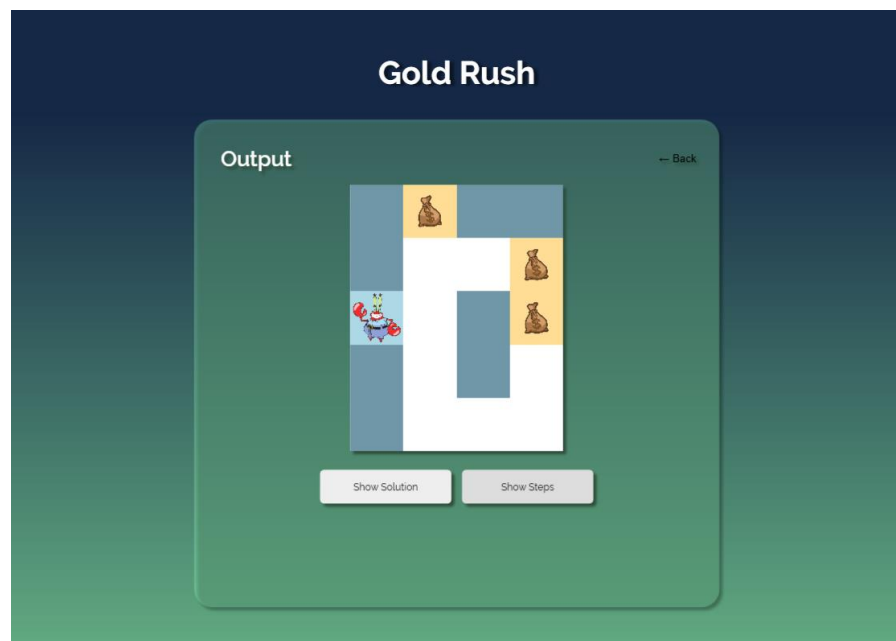


- Pilih algoritma yang ingin digunakan (BFS/DFS). Centang TSP jika ingin menyelesaikan permasalahan TSP. Geser *numeric slider* untuk memilih waktu animasi (interval antar-step)



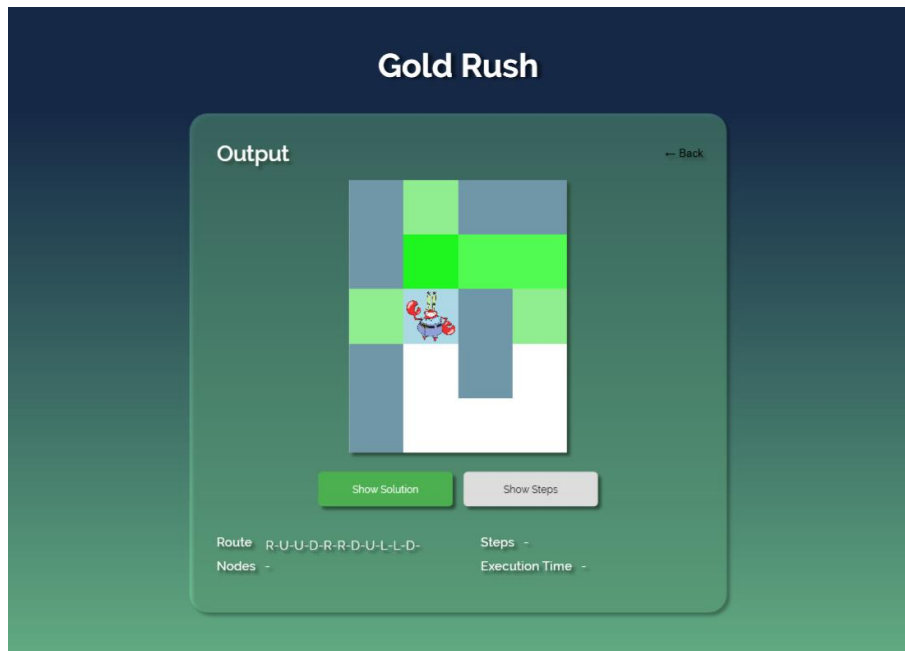
Gambar 7. Pemilihan algoritma

- Tekan tombol  $\rightarrow$  untuk mem-visualisasi *maze*.



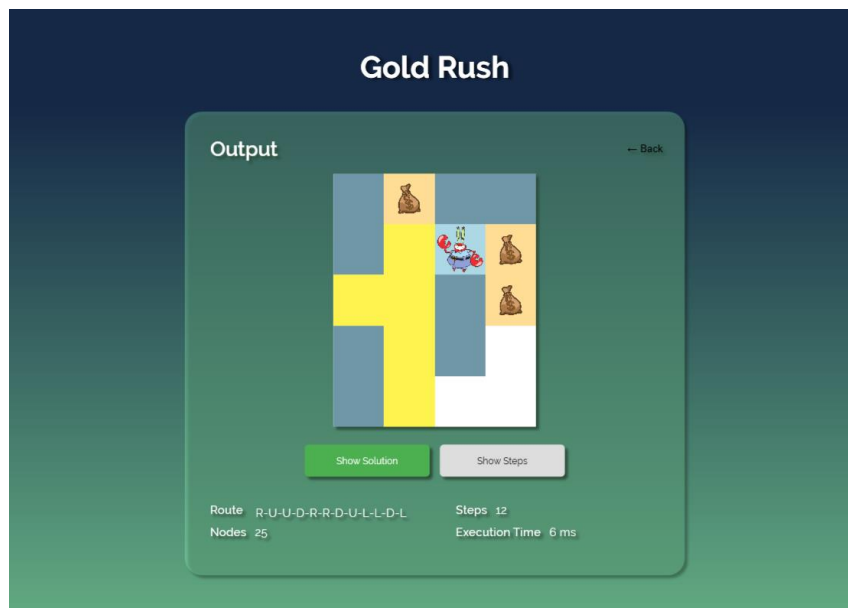
Gambar 8. Hasil visualisasi maze

- Tekan toggle “Show Solution” untuk menampilkan animasi solusi.



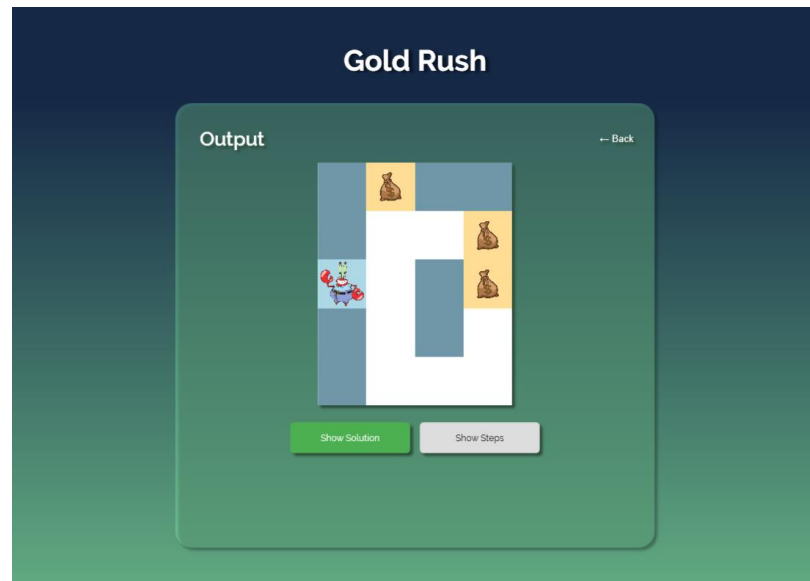
Gambar 9. Solusi

5. Toggle “Show Solution” dapat diklik lagi untuk me-reset maze.
6. Tekan tombol “Show Steps” untuk menampilkan tahap pengecekan *node*.



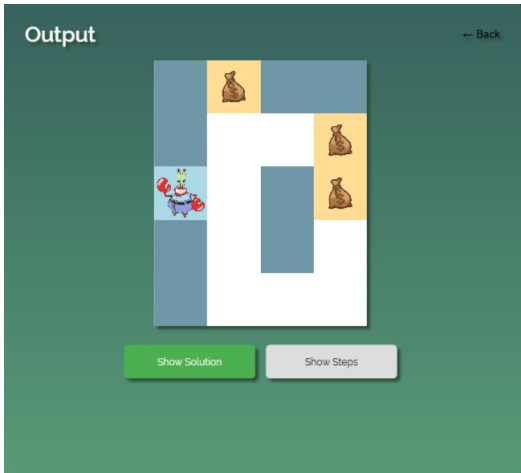
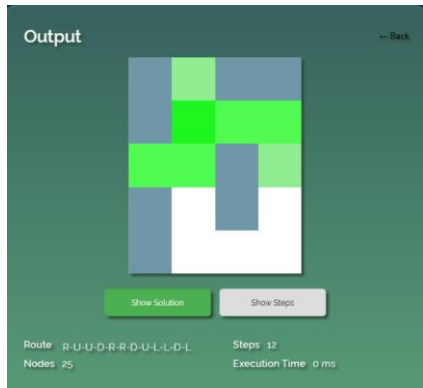
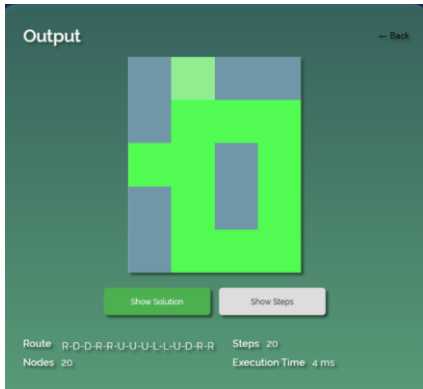
Gambar 10. Step pencarian solusi

7. Tekan tombol “Back” untuk mengganti konfigurasi



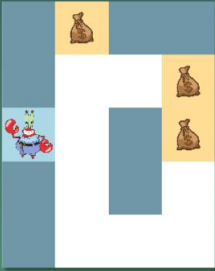


*Gambar 11. Tombol back*

**4.5 Analisis dan Pengujian****Kasus 1 (Dengan TSP)**



Input	Hasil
	<b>BFS</b>
	
	<b>DFS</b>
	

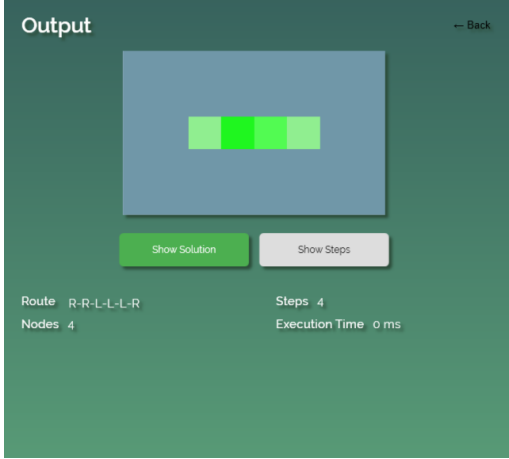
**Kasus 1 (Tanpa TSP)**

Input	Hasil
	<b>BFS</b>

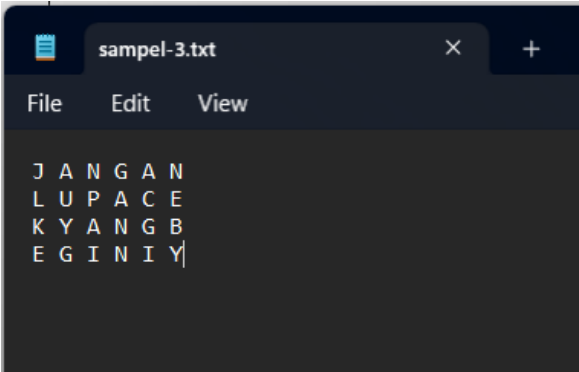
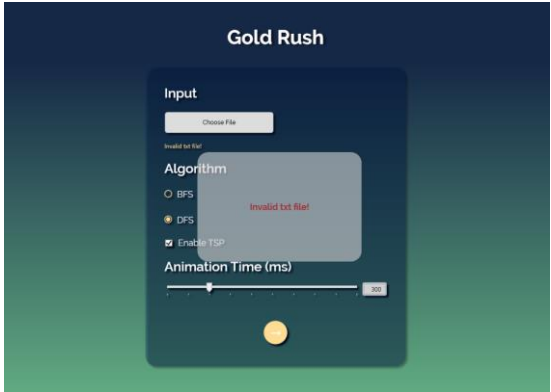
<p>Output</p>  <p>Show Solution Show Steps</p>	<p>Output</p>  <p>Show Solution Show Steps</p> <p>Route R-U-U-D-R-R-D Nodes 12</p> <p>Steps 7 Execution Time 0 ms</p>
	<p>DFS</p> <p>Output</p>  <p>Show Solution Show Steps</p> <p>Route R-D-D-R-R-U-U-L-L-U Nodes 12</p> <p>Steps 12 Execution Time 0 ms</p>

**Kasus 2 (Tanpa TSP)**

<p>Input</p> <p>Output</p>  <p>Show Solution Show Steps</p>	<p>Hasil</p> <p>BFS</p> <p>Output</p>  <p>Show Solution Show Steps</p> <p>Route L-R-R-L-L Nodes 9</p> <p>Steps 6 Execution Time 0 ms</p> <p>DFS</p>
--	---

	
--	--

**Kasus 3**

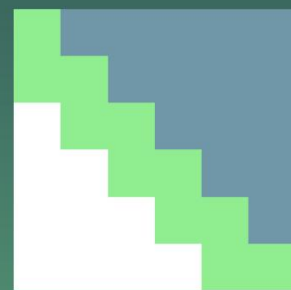
Input	Hasil
	

**Kasus 4 (Dengan TSP)**

Input	Hasil
	<b>BFS</b>

[illegible]

### Kasus 4 (Tanpa TSP)

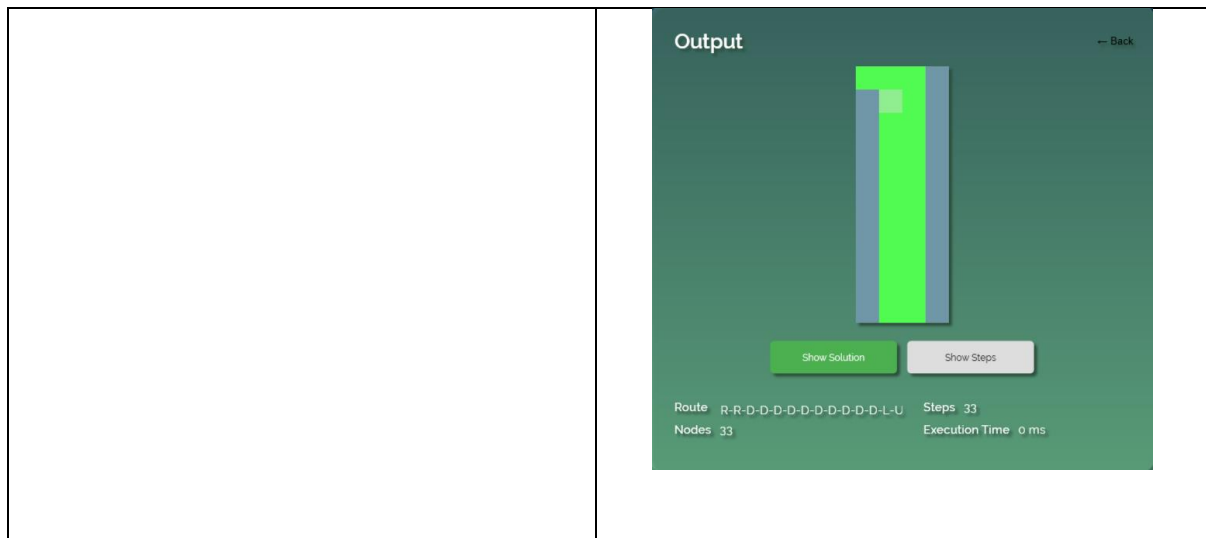
Input	Hasil
	<div data-bbox="793 1473 1375 2018"> <h3>BFS</h3> <div> <div>Output</div> <div>  </div> <div> <div>Show Solution</div> <div>Show Steps</div> </div> <div> <div>Route D-R-D-R-D-R-D-R-D-R</div> <div>Steps 10</div> </div> <div> <div>Nodes 155</div> <div>Execution Time 0 ms</div> </div> </div> </div>

<div><div>Output</div><div></div><div><div>Show Solution</div><div>Show Steps</div></div></div>	<div>← Back</div>
	<div><div>DFS</div><div><div>Output</div><div></div><div><div>Show Solution</div><div>Show Steps</div></div><div><div>Route D-R-D-R-D-R-D-R-D-R</div><div>Steps 11</div><div>Nodes 11</div><div>Execution Time 0 ms</div></div></div></div>

**Kasus 5 (Dengan TSP)**

<div>Input</div>	<div>Hasil</div>
<div><div>Output</div><div></div><div><div>Show Solution</div><div>Show Steps</div></div></div>	<div>DFS</div>
	<div><div>Output</div><div></div><div><div>Show Solution</div><div>Show Steps</div></div><div><div>Route R-D-D-D-D-D-D-D-D-U-U-</div><div>Steps 22</div><div>Nodes 100</div><div>Execution Time 39 ms</div></div></div>
	<div>DFS</div>





### Analisis:

Dari kasus-kasus di atas, kita bisa melihat perbedaan signifikan antara BFS dan DFS, yaitu pada jumlah *node* yang dicek dan pada jumlah *step* pada solusi. BFS melakukan pengecekan *node* jauh lebih banyak dibandingkan DFS. Selain itu, jumlah *step* pada solusi BFS lebih sedikit dibandingkan DFS. Artinya, BFS lebih banyak melakukan pengecekan *meaningless* dibandingkan DFS karena BFS selalu mengecek semua kemungkinan pada “kedalaman” tertentu sebelum melanjutkan ke kedalaman selanjutnya. Namun, BFS menghasilkan *step* yang lebih sedikit karena alasan yang sama. Sebaliknya, DFS tidak melakukan pengecekan *node* terlalu banyak karena akan mengecek sebuah kemungkinan dan berlanjut terus ke kedalaman berikutnya sampai ditemukan solusi. Hal ini menyebabkan DFS memiliki *step* yang lebih banyak dibandingkan BFS. BFS juga menghasilkan jumlah *step* minimum.

Dari hasil eksperimen di atas juga terlihat bahwa BFS dan DFS merupakan algoritma yang efisien dalam menyelesaikan permasalahan pencarian *treasure* dalam *maze* karena waktu eksekusi yang cepat untuk berbagai varian permasalahan.

## BAB V

### KESIMPULAN DAN SARAN

Dari tugas besar IF2211 Strategi Algoritma ini, kami telah berhasil membuat bot program “GoldRush” yang dapat digunakan untuk menyelesaikan permasalahan pencarian *path* dalam *maze* dengan algoritma BFS dan DFS. Kami juga telah berhasil memanfaatkan C# Development Framework (dalam hal ini .NET dan WPF) dengan baik.

Algoritma DFS dan BFS terbukti dapat menyelesaikan permasalahan secara *time-efficient*. Namun, pemilihan algoritma DFS dan BFS dapat didasarkan oleh keinginan pengguna. Jika pengguna mengharapkan solusi merupakan jumlah *step* minimum, maka algoritma BFS menunjukkan performa yang lebih baik. Jika pengguna mengharapkan untuk meminimasi kalkulasi, maka algoritma DFS lebih baik.

Saran pengembangan untuk tugas besar ini adalah:

1. Lebih komunikatif agar ide-ide yang dimiliki dapat dipahami oleh seluruh anggota tim dan terealisasi.
2. Kesepakatan struktur data dalam perancangan sebaiknya dibahas lebih baik sebelum implementasi dilaksanakan.

## DAFTAR PUSTAKA

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

terakhir diakses 22 Maret 2023, 19.36

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

terakhir diakses 22 Maret 2023, 19.37

<https://docs.google.com/document/d/11AUaI6PsZK089rcWfaTYRgMwNwe3wvDbGdIdJrLatEU/edit>, terakhir diakses 23 Maret 2023, 21.41

## REPOSITORY

[https://github.com/Mehmed13/Tubes2\\_goldRush](https://github.com/Mehmed13/Tubes2_goldRush)

## YOUTUBE VIDEO

<https://youtu.be/QEeqFvwLxAU>