

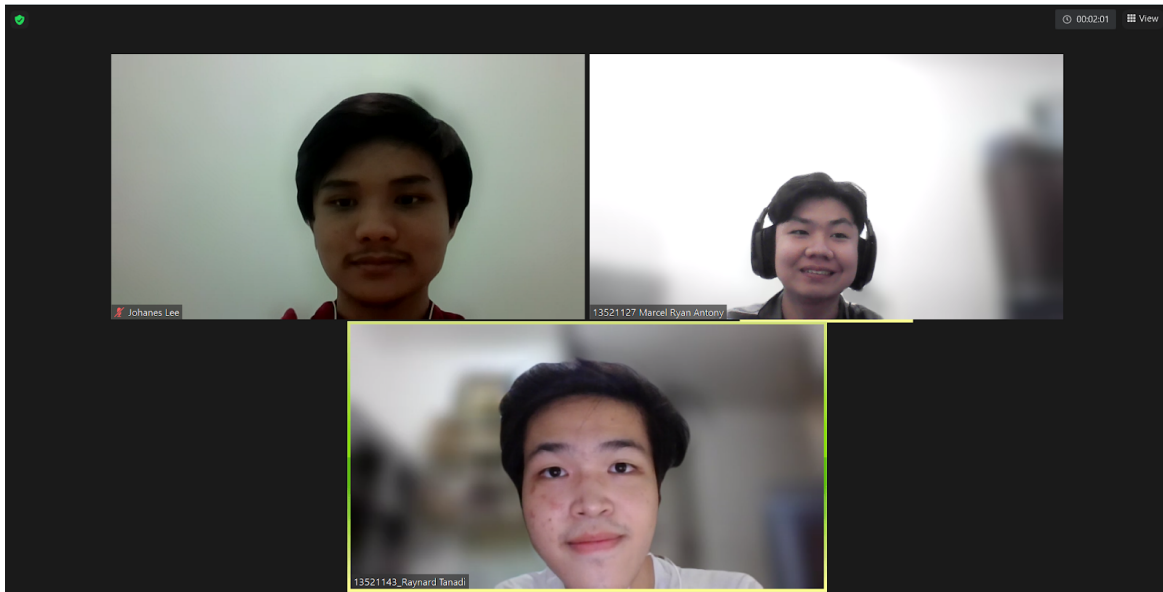
Tugas Besar 2 IF2211 Strategi Algoritma
Semester II tahun 2022/2023

Aplikasi Algoritma BFS dan DFS dalam Menyelesaikan Persoalan Maze Treasure Hunt

Disusun Oleh:

Kelompok 26 Euforia

Marcel Ryan Antony	13521127
Raynard Tanadi	13521143
Johanes Lee	13521148



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

Bab 1: Deskripsi Tugas

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



Gambar 1.1 Labirin di Bawah Krusty Krab

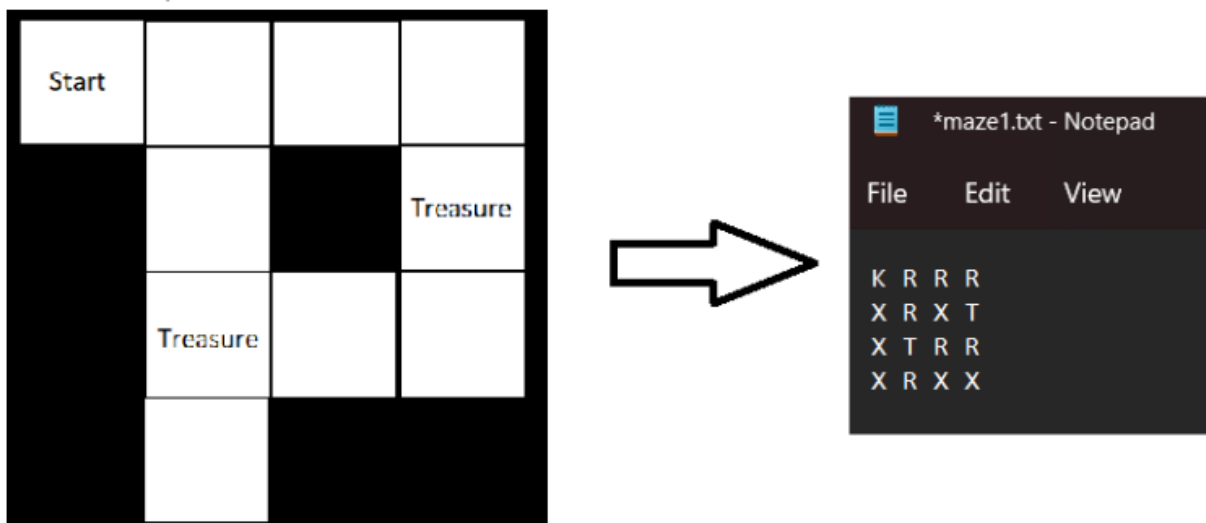
Setelah berpikir cukup lama, Tuan Krabs tiba-tiba mengingat bahwa ketika Ia berada pada kelas Strategi Algoritma-nya dulu, Ia ingat bahwa Ia dulu mempelajari algoritma BFS dan DFS sehingga Tuan Krabs menjadi yakin bahwa persoalan ini dapat diselesaikan menggunakan kedua algoritma tersebut. Akan tetapi, 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.

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

Contoh file input :

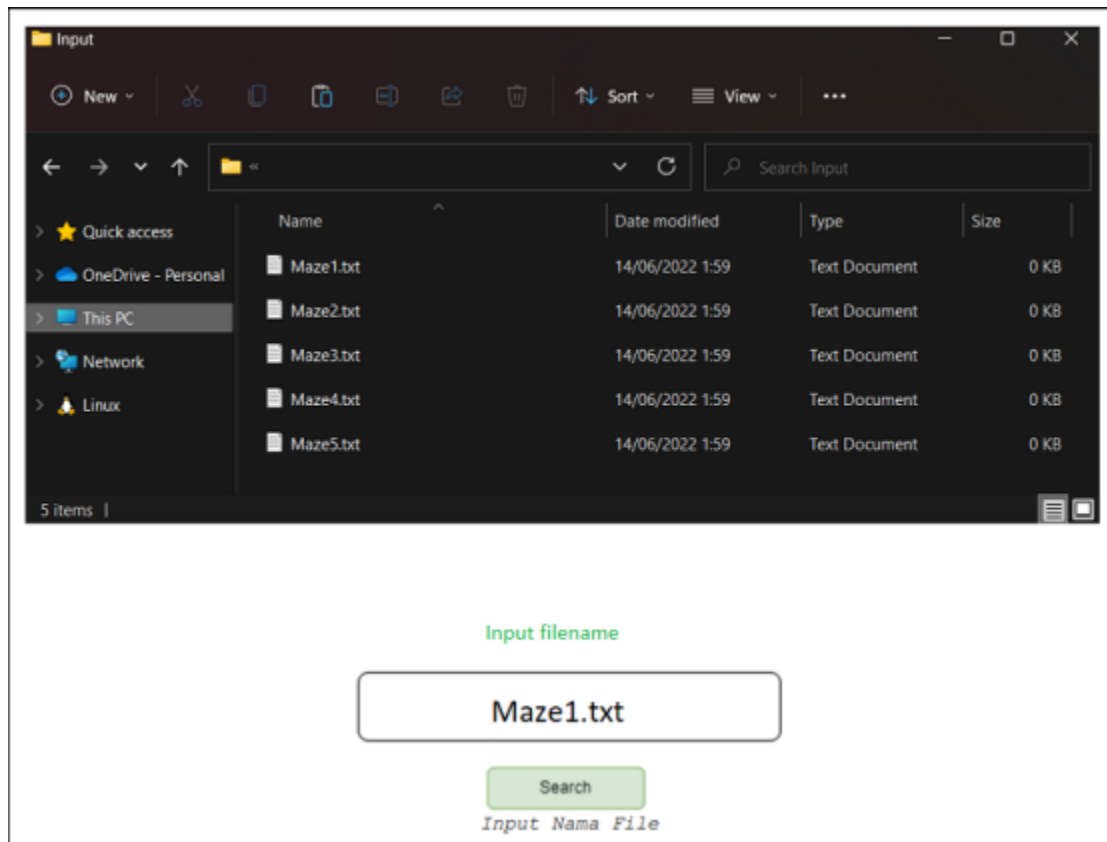


Gambar 1.2 Ilustrasi input file maze

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

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

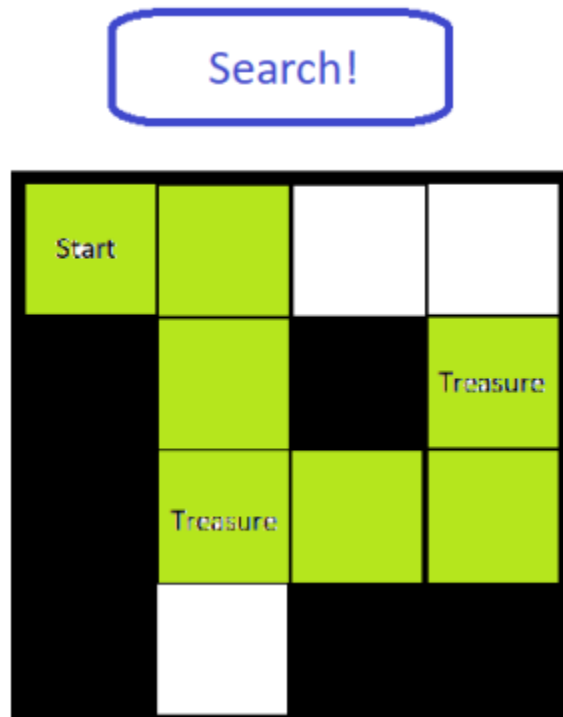
Contoh input aplikasi :



Gambar 1.3 Contoh input program

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.

Contoh output Aplikasi :



Gambar 1.4 Contoh output program untuk gambar 2

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.

Spesifikasi Program:

Aplikasi yang akan dibangun dibuat berbasis GUI. Berikut ini adalah contoh tampilan dari aplikasi GUI yang akan dibangun

Treasure Hunt Solver

Input

Filename

Algoritma
☒ BFS
☐ DFS

Output

Start			
			Treasure
	Treasure		

Route:
Nodes :

Steps:
Execution Time :

Gambar 1.5 Tampilan Program Sebelum dicari solusinya

Treasure Hunt Solver

Input

Filename

Algoritma
☒ BFS
☐ DFS

Output

Start			
			Treasure
	Treasure		

Route: R - D - D - R - R - U
Nodes : 11

Steps: 6
Execution Time : 850 ms

Gambar 1.6 Tampilan Program setelah dicari solusinya

Catatan: Tampilan diatas hanya berupa contoh layout dari aplikasi saja, untuk design layout aplikasi dibebaskan dengan syarat mengandung seluruh input dan output yang terdapat pada spesifikasi.

Spesifikasi GUI:

1. **Masukan program** adalah file maze treasure hunt tersebut atau nama filenya.
2. Program dapat menampilkan visualisasi dari input file maze dalam bentuk grid dan pewarnaan sesuai deskripsi tugas.
3. Program memiliki toggle untuk menggunakan alternatif algoritma BFS ataupun DFS.
4. Program memiliki tombol search yang dapat mengeksekusi pencarian rute dengan algoritma yang bersesuaian, kemudian memberikan warna kepada rute solusi output.
5. **Luaran program** adalah banyaknya node (grid) yang diperiksa, banyaknya langkah, rute solusinya, dan waktu eksekusi algoritma.
6. **(Bonus)** Program dapat menampilkan progress pencarian grid dengan algoritma yang bersesuaian. Hal tersebut dilakukan dengan memberikan slider / input box untuk menerima durasi jeda tiap step, kemudian memberikan warna kuning untuk tiap grid yang sudah diperiksa dan biru untuk grid yang sedang diperiksa.
7. **(Bonus)** Program membuat toggle tambahan untuk persoalan TSP. Jadi apabila toggle dinyalakan, rute solusi yang diperoleh juga harus kembali ke titik awal setelah menemukan segala harta karunnya (Tetap dengan algoritma BFS atau DFS).
8. GUI dapat dibuat **sekreatif** mungkin asalkan memuat 5 (7 jika mengerjakan bonus) spesifikasi di atas.

Program yang dibuat harus memenuhi spesifikasi wajib sebagai berikut:

- 1) Buatlah program dalam bahasa **C#** untuk mengimplementasi Treasure Hunt Solver sehingga diperoleh output yang diinginkan. Penelusuran harus memanfaatkan algoritma **BFS** dan **DFS**.
- 2) Awalnya program menerima file atau nama file maze treasure hunt.
- 3) Apabila filename tersebut ada, Program akan melakukan validasi dari file input tersebut. Validasi dilakukan dengan memeriksa apakah tiap komponen input hanya berupa K, T, R, X.

Apabila validasi gagal, program akan memunculkan pesan bahwa file tidak valid. Apabila validasi berhasil, program akan menampilkan **visualisasi** awal dari maze treasure hunt.

4) Pengguna memilih algoritma yang digunakan menggunakan toggle yang tersedia.

5) Program kemudian dapat menampilkan **visualisasi** akhir dari maze (dengan pewarnaan rute solusi).

6) Program menampilkan luaran berupa durasi eksekusi, rute solusi, banyaknya langkah, serta banyaknya node yang diperiksa.

Proses visualisasi ini boleh memanfaatkan pustaka atau kakas yang tersedia. Sebagai referensi, salah satu kakas yang tersedia untuk memvisualisasikan matrix dalam bentuk grid adalah **DataGridView**. Berikut adalah panduan singkat terkait penggunaannya <http://csharp.net-informations.com/datagridview/csharp-datagridview-tutorial.htm>

7) Mahasiswa **tidak diperkenankan** untuk melihat atau menyalin library lain yang mungkin tersedia bebas terkait dengan pemanfaatan BFS dan DFS. Akan tetapi, untuk algoritma lain diperbolehkan menggunakan library jika ada.

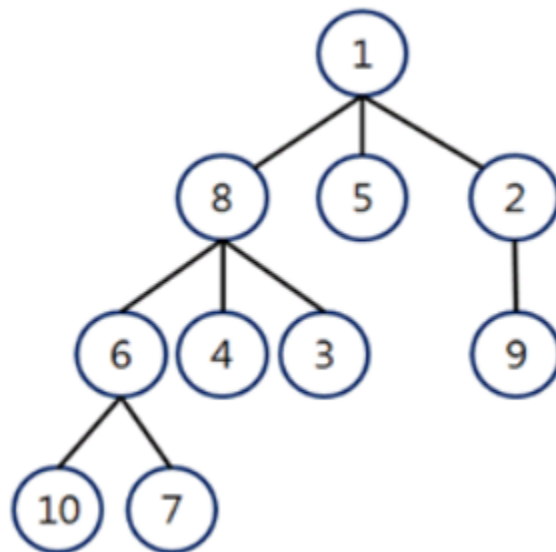
Bab 2: Landasan Teori

2.1. *Graph Traversal*

Traversal graf atau *graph traversal* adalah proses mengunjungi setiap simpul pada graf dengan sistematis. Algoritma traversal graf diklasifikasikan berdasarkan cara algoritma tersebut menentukan urutan simpul yang dikunjungi terlebih dahulu pada sebuah graf. Dengan demikian, dapat diperoleh dua klasifikasi umum, yaitu *Breadth First Search* (BFS) atau pencarian melebar dan *Depth First Search* (DFS) atau pencarian mendalam.

2.2. *Breadth First Search*

Breadth First Search memulai pencarian dari simpul akar dan kemudian mengunjungi semua simpul yang berada pada tingkat kedalaman (*level*) yang sama sebelum melanjutkan pencarian pada simpul dengan tingkat kedalaman selanjutnya. Berdasarkan graf pada gambar 2.2.1, pencarian akan dimulai pada simpul 1 dan kemudian dilanjutkan ke simpul 8, 5, dan 2 terlebih dahulu sebelum mengunjungi simpul di bawahnya karena ketiga simpul tersebut memiliki kedalaman yang lebih kecil dibandingkan simpul lainnya yang belum dikunjungi. Algoritma ini dapat juga diimplementasikan ke pencarian di dalam graf yang bukan berbentuk *tree*, dengan kedalaman menyatakan banyak sisi minimum yang perlu dilalui untuk sampai ke simpul tertentu dari simpul awal. Implementasi algoritma BFS umumnya berkaitan dengan struktur data *queue* yang memiliki properti FIFO (*First In First Out*) untuk menentukan urutan simpul yang dikunjungi.



Gambar 2.2.1 Graf contoh

2.3. *Depth First Search*

Depth First Search atau pencarian mendalam memprioritaskan urutan pencarian pada salah satu simpul tetangga terlebih dahulu sebelum melakukan pencarian untuk simpul tetangga lainnya.

Apabila tidak ada lagi simpul tetangga yang dapat diperiksa, pencarian akan kembali ke simpul sebelumnya hingga ditemukan simpul dengan tetangga yang belum dikunjungi. Sebagai contoh, dengan menggunakan graf yang sama pada Gambar 2.2.1, pencarian akan dimulai dari simpul 1 dan dilanjutkan ke simpul 8, kemudian simpul 6 dan 10. Setelah tiba di simpul 10, proses pencarian kembali ke simpul 6 dan memeriksa simpul ke 7 sebagai tetangga yang belum diperiksa. Proses pencarian akan terus dilanjutkan hingga semua simpul sudah dikunjungi. Algoritma DFS memungkinkan seringnya terjadi proses runut-balik (*backtrack*) akibat tidak ada simpul tetangga yang belum dikunjungi pada simpul tertentu. Implementasi algoritma DFS umumnya berkaitan dengan struktur data *stack* yang memiliki properti FILO (*First In Last Out*) untuk menentukan urutan simpul yang dikunjungi.

2.4. C# Desktop Application Development

C# adalah bahasa pemrograman yang cukup populer dalam membuat aplikasi desktop pada platform Microsoft Windows. *Desktop Application* adalah sebuah perangkat lunak yang dapat dipasang dan dioperasikan secara lokal pada komputer atau laptop. IDE Microsoft Visual Studio umum digunakan dalam pengembangan sebuah *desktop application* dalam bahasa pemrograman C#.

Berikut adalah beberapa langkah pengembangan *desktop application* menggunakan Microsoft Visual Studio dalam bahasa pemrograman C# dengan memanfaatkan *Windows Forms App Framework*.

1. Buat *project* baru dengan langkah-langkah berikut.
 - a. Buka aplikasi Microsoft Visual Studio
 - b. Pada menu awal pilih *Create a new project*
 - c. Pilih *Windows Forms App* atau *Windows Forms App (.NET Framework)*
 - d. Isi data-data yang diperlukan untuk membuat *project* baru seperti *project name* dan lokasi *project*.
2. Setelah *project* baru berhasil dibuat, Microsoft Visual Studio akan secara otomatis membuat *form* baru
3. Pada *form* tersebut dapat ditambahkan berbagai komponen yang ingin digunakan pada aplikasi desktop menggunakan menu *Toolbox*
4. Klik dua kali file *Form1.cs[Design]* untuk melihat kode inisialisasi berbagai komponen *form*
5. Klik dua kali komponen tertentu yang ditampilkan pada *form* untuk membuka kode *Form1.cs* yang berisi berbagai fungsi untuk mengatasi *event* interaksi pengguna
6. Tekan tombol *start* pada menu atas Visual Studio (mode *debug* ataupun *release*) untuk melakukan kompilasi dan menjalankan program yang dibuat.

Bab 3: Aplikasi Strategi *BFS* dan *DFS*

3.1. Langkah Pemecahan Masalah

Permasalahan utama pada tugas besar 2 strategi algoritma ini adalah pencarian *treasure* pada *maze*. Pencarian akan dimulai dari kotak *start* dan solusi didapatkan jika diperoleh rute sehingga seluruh *treasure* telah ditemukan. Dalam pemecahan masalah ini, digunakan algoritma pencarian BFS dan DFS.

Implementasi algoritma pencarian BFS dapat menggunakan metode iteratif dengan memanfaatkan struktur data *queue*. Prioritas pencarian dilakukan dengan urutan prioritas arah atas, kanan, bawah, dan kiri (*up, right, down, left*). Algoritma pencarian BFS yang digunakan dibagi menjadi dua jenis, yaitu memperbolehkan *multiple visits* ataupun tidak. Pada jenis pertama, terdapat sedikit modifikasi dari BFS pada umumnya karena dibutuhkan rute atau *path* yang jelas untuk mendapatkan seluruh *treasure*. Pada algoritma ini, pencarian dimulai dengan pencarian BFS pada umumnya. Namun, apabila ditemukan *treasure* baru, algoritma akan me-reset ulang *queue* pada saat itu dan simpul-simpul sebelumnya sebelumnya. Dengan demikian, semua simpul dapat dikunjungi kembali dan *treasure* yang baru ditemukan dijadikan sebagai *start* baru untuk melakukan algoritma BFS ke *treasure* lainnya. Pendekatan ini akan mendapatkan seluruh *treasure* yang ada dengan rute yang jelas.

Jenis kedua untuk pendekatan BFS tidak mengizinkan suatu simpul dikunjungi dua kali dalam rute solusi. Akibatnya, pendekatan ini perlu melakukan pencarian terpisah untuk setiap cabang rute yang ditemukan sehingga pencarian cabang tertentu bersifat independen dengan pencarian cabang lainnya. Pendekatan ini menyimpan catatan simpul-simpul yang dikunjungi secara independen untuk masing-masing cabang rute yang mungkin. Namun, urutan pencarian rute tetap mengikuti prinsip BFS sehingga rute dengan panjang yang lebih tinggi tidak dapat diperiksa sebelum semua rute dengan panjang yang lebih rendah selesai diperiksa. Pencarian berhenti ketika rute yang sedang diperiksa ternyata dapat memperoleh seluruh *treasure* pada *maze*.

Implementasi algoritma pencarian DFS juga menggunakan metode iteratif dan memanfaatkan struktur data *stack*. Prioritas pencarian dilakukan dengan urutan prioritas arah atas, kanan, bawah, dan kiri (*up, right, down, left*). Algoritma pencarian DFS juga dibagi menjadi dua jenis, yaitu memperbolehkan *multiple visits* ataupun tidak. Pada jenis *multiple visits*, pencarian dilakukan menggunakan algoritma DFS pada umumnya. Apabila tidak ada simpul tetangga yang belum dikunjungi pada simpul tertentu, proses pencarian akan mundur ke simpul-simpul sebelumnya hingga mendapatkan simpul yang memiliki tetangga tersebut. Proses mundur ini juga dimasukkan dalam rute akhir solusi. Pencarian akan terus dilakukan hingga seluruh *treasure* ditemukan.

Jenis kedua untuk pendekatan DFS memanfaatkan algoritma *backtrack* untuk mengeliminasi rute yang menyebabkan pencarian berhenti akibat tidak ada simpul tetangga yang dapat dikunjungi tanpa kembali ke simpul sebelumnya. Rute yang dieliminasi tidak akan dimasukkan di dalam rute solusi. Pencarian akan terus dilakukan dengan prinsip DFS dan *backtrack* hingga ditemukan suatu rute yang memperoleh seluruh *treasure* tanpa melakukan pengunjungan simpul lebih dari sekali.

Rute solusi disimpan dalam bentuk urutan posisi. Setelah mendapatkan rute solusi, dapat diperoleh juga *node count*, *step count*, *time execution*, dan rute dalam bentuk urutan karakter U, R, L, dan D (U menandakan *Up*, R menandakan *Right*, L menandakan *Left*, dan D menandakan *Down*). *Node count* dihitung dari banyak *grid* yang pernah diperiksa pada saat proses pencarian (maksimal sebanyak *grid* yang dapat dikunjungi, yaitu banyak karakter selain X pada peta). *Step count* dihitung berdasarkan panjang rute karakter arah yang dihasilkan.

Permasalahan lainnya adalah diperlukannya GUI atau *Graphical User Interface* untuk program ini. Untuk penyelesaian permasalahan ini, digunakan WinForms yang dilengkapi dengan .NET *framework* pada Visual Studio. Pembuatan GUI didasarkan pada spesifikasi yang sudah ada (termasuk bonus visualisasi *progress* pencarian) dan dilengkapi dengan beberapa hal yang berkaitan dengan kreativitas.

Terakhir, terdapat permasalahan bonus berupa TSP sehingga rute solusi yang diperoleh juga harus kembali ke titik awal setelah menemukan seluruh *treasure*. Untuk penyelesaian permasalahan bonus ini, akan ditambahkan algoritma TSP pada kedua macam algoritma pencarian BFS dan kedua macam algoritma pencarian DFS sesuai dengan algoritma masing-masing. Penambahan algoritma ini sebenarnya cukup dengan membuat titik asal sebagai simpul yang dicari setelah seluruh *treasure* ditemukan. Semua jenis algoritma yang digunakan dalam pencarian *treasure* akan tetap dapat diaplikasikan dalam pencarian rute kembali ke titik awal.

3.2. Pemetaan Persoalan menjadi Elemen Algoritma BFS dan DFS

3.2.1. Breadth First Search

Algoritma *breadth first search* yang digunakan untuk memecahkan permasalahan ini menggunakan konsep pencarian solusi dari graf statis (memperbolehkan *multiple visits*) maupun dinamis (tidak memperbolehkan *multiple visits*). Pencarian selesai apabila semua *treasure* telah ditemukan. Dalam algoritma BFS ini digunakan struktur data *queue* yang berisi urutan *node-node* yang akan dikunjungi serta simpul tetangga asalnya (digunakan untuk memperoleh informasi urutan rute). Selain itu, pada permasalahan ini, digunakan juga sebuah matriks *Tuple<bool, Tuple<int, int>> _checkmap[[]]* yang digunakan untuk mengecek apakah *node* tersebut telah dikunjungi serta posisi *node* sebelumnya yang dikunjungi sebelumnya. (Matriks tersebut juga digunakan dalam mendapatkan rute solusi akhir.)

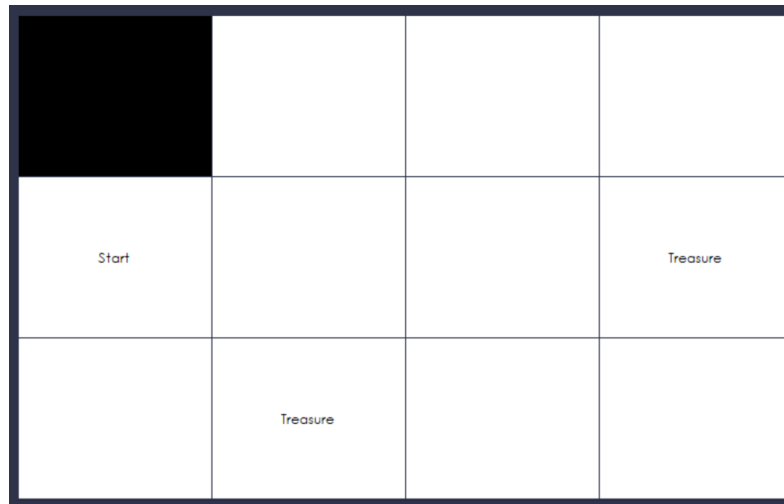
Dalam pencarian *treasure* setiap tetangga dari *node* yang sedang diperiksa akan dimasukkan ke dalam *queue*. Selain itu, untuk setiap *node* yang diperiksa akan di-set matriks *_checkmap[[]]* pada posisi tersebut menjadi *true* untuk elemen pertama dari *tuple* dan nilai posisi *node* sebelumnya untuk elemen kedua dari *tuple* tersebut. Iterasi akan terus dilakukan sampai semua *treasure* telah ditemukan atau dipastikan tidak ditemukan solusi.

3.2.2. Depth First Search

Algoritma *depth first search* yang digunakan untuk memecahkan permasalahan ini juga menggunakan konsep pencarian solusi graf statis (memperbolehkan *multiple visits*) maupun dinamis (tidak memperbolehkan *multiple visits*). Dalam algoritma DFS ini digunakan juga matriks *_checkmap* untuk kegunaan yang sama seperti pada algoritma BFS. Namun, berbeda dengan algoritma BFS,

digunakan struktur data *stack* untuk menyimpan urutan *node-node* yang akan dikunjungi. Iterasi juga akan terus dilakukan hingga ditemukan solusi akhir ataupun dipastikan tidak ada solusi untuk konfigurasi peta tertentu.

3.3. Ilustrasi Kasus Persoalan



Gambar 3.3.1 Contoh Peta Labirin

Gambar 3.3.1 merepresentasikan persoalan mencari jalur untuk mendapatkan semua *treasure* yang ada pada labirin. Pada contoh ini, kotak hitam merepresentasikan dinding, kotak *start* merepresentasikan titik awal, kotak *treasure* merepresentasikan harta karun, serta kotak putih kosong merepresentasikan kotak yang dapat dilewati. Peta yang dijadikan contoh sengaja dibuat demikian untuk menunjukkan kemungkinan hasil yang berbeda dengan pendekatan algoritma berbeda.

Terlepas dari urutan arah pencarian *up right down left* (URDL), pendekatan BFS untuk *multiple visits* yang telah dijelaskan akan selalu memperoleh jalan menuju *treasure* terdekat dari titik tertentu (jalur akhir untuk mendapatkan keseluruhan *treasure* tidak dijamin optimal). Setelah mendapatkan salah satu *treasure*, akan dilakukan pencarian jalan menuju *treasure* lainnya yang terdekat dari titik saat itu. Langkah ini akan diulang hingga dicapai tujuan (mendapatkan seluruh *treasure* dan dapat kembali lagi ke titik awal jika pencarian sedang berada dalam mode TSP). Pada Gambar 3.3.2, terlihat bahwa kotak $(1, 1)$ dikunjungi sebanyak dua kali (indeks baris dan kolom dimulai dari 0).

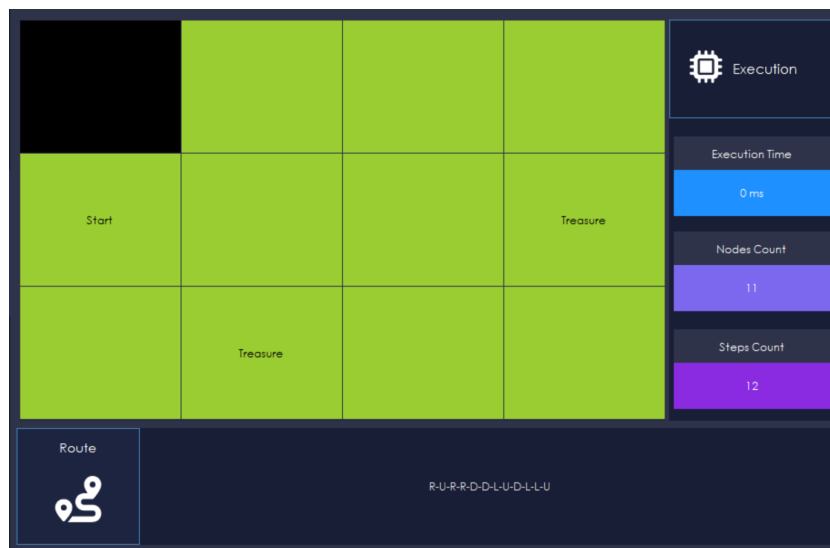
Gambar 3.3.2 Hasil Pencarian BFS untuk *Multiple Visits*Gambar 3.3.3 Hasil Pencarian BFS untuk *Multiple Visits* dan TSP

Berbeda dengan BFS di atas, pendekatan DFS untuk *multiple visits* dipengaruhi oleh prioritas arah yang dipilih. Dengan prioritas URDL, pencarian akan bergerak ke kanan terlebih dahulu (tidak dapat bergerak ke atas akibat terhalang dinding) dan kemudian menelusuri kotak pada baris teratas. Setelah tiba pada kotak $(0, 3)$, pencarian akan memprioritaskan arah bawah dibandingkan kiri (arah atas dan kanan tidak dapat ditelusuri lagi). Setelah bergerak kiri dari kotak $(2, 3)$ ke kotak $(2, 2)$, diprioritaskan lagi arah atas akibat kotak $(1, 2)$ belum pernah dikunjungi. Pada kasus tersebut, pencarian mundur satu langkah akibat sudah tidak ada lagi kotak yang belum dikunjungi setelah sampai di titik tersebut. Pencarian dilanjutkan hingga diperoleh *treasure* kedua, yaitu kotak di sebelah kiri yang belum dikunjungi. Untuk mode TSP, DFS akan dilanjutkan seperti pencarian DFS pada

umumnya hingga memperoleh jalan menuju titik awal dengan menelusuri kotak yang belum dikunjungi (dalam kasus ini adalah kotak di sebelah kiri *treasure* kedua yang ditemukan).



Gambar 3.3.3 Hasil Pencarian DFS untuk *Multiple Visits*



Gambar 3.3.4 Hasil Pencarian DFS untuk *Multiple Visits* dan TSP

Jalur untuk mendapatkan keseluruhan *treasure* yang lebih baik tentu dapat diperoleh jika kotak tertentu hanya diperbolehkan untuk dilalui maksimal sekali (kecuali titik awal dalam mode TSP). Namun, pencarian jalur ini membutuhkan tambahan pendekatan algoritma *backtrack*, baik pada BFS maupun DFS. Selain itu, pendekatan BFS dan DFS untuk kasus ini juga tetap dapat menghasilkan jalur yang berbeda. Pendekatan gabungan BFS dan *backtrack* selalu mendapatkan solusi optimal sedangkan pendekatan DFS tidak dapat dijamin keoptimalannya. Kedua pendekatan ini juga menyebabkan beberapa persoalan tidak dapat diselesaikan walaupun dapat diselesaikan jika memperbolehkan *multiple visits*.



Gambar 3.3.5 Hasil Pencarian BFS tanpa *Multiple Visits*



Gambar 3.3.6 Hasil Pencarian BFS tanpa *Multiple Visits* dalam Mode TSP

Gambar 3.3.7 Hasil Pencarian DFS tanpa *Multiple Visits*Gambar 3.3.8 Hasil Pencarian DFS tanpa *Multiple Visits* dalam Mode TSP

Bab 4: Analisis Pemecahan Masalah

4.1. Implementasi Algoritma

Pendekatan BFS**Atribut Kelas BFSState**

```

nReset                : integer
multipleVisitPath     : ArrayList
fromPosition          : Tuple<integer, integer>
nowPosition           : Tuple<integer, integer>
stop                  : bool
position              : Tuple<integer, integer>
_queue               : Queue
treasureFound         : bool
memo                  : Tuple<bool, Tuple<integer, integer>>[, ]
nodeCount             : integer
foundTreasureCount    : integer
TPosition            : ArrayList
defaultCheckValue     : Tuple<bool, Tuple<integer, integer>>z
row                   : integer
col                   : integer
treasureCount         : integer
sequentialMode        : bool
tspMode               : bool
foundAll              : bool

```

```

// StateInfo menyimpan atribut stepCount, foundTreasureCount,
// memo sebagai matriks yang mencatat grid yang telah dikunjungi,
// prevPosition untuk posisi sebelum posisi saat ini,
// serta row dan col peta
_shortestPathQueue    : Tuple<Tuple<integer, integer>, StateInfo>

```

```

Function GetMapElmt(input currentPosition : Tuple<integer, integer>)
    → string
{ Mengembalikan string karakter pada posisi tertentu dalam peta. }

```

```

Function GetCheckMap(input currentPosition : Tuple<integer, integer>)
    → Tuple<boolean, Tuple<integer, integer>>
{ Mengembalikan status pemeriksaan suatu posisi pada peta.
  Elemen pertama tuple merupakan status sudah dikunjungi.
}

```

Elemen kedua tuple merupakan posisi kotak asal sebelum mengunjungi kotak saat ini.}

Function GetCheckMap(**input** currentPosition : Tuple<integer, integer>, **input** memo : Tuple<boolean, Tuple<integer, integer>>[,]))

→ Tuple<boolean, Tuple<integer, integer>>

{ Mengembalikan status pemeriksaan suatu posisi pada memo.

Elemen pertama tuple merupakan status sudah dikunjungi.

Elemen kedua tuple merupakan posisi kotak asal sebelum mengunjungi kotak saat ini.}

Procedure SetCheckMap(**input** currentPosition : Tuple<integer, integer>, **input** checkStatus : Tuple<boolean, Tuple<integer, integer>>)

{ mengubah atribut stop menjadi **true** dan memperbarui nilai **stepCount** }

Function IsValid(**input** currentPosition : Tuple<integer, integer>)

→ boolean

{ mengembalikan true jika currentPosition merupakan kotak yang dapat dikunjungi serta belum pernah dikunjungi pada peta. }

Function IsValid(**input** currentPosition : Tuple<integer, integer>, **input** memo : Tuple<boolean, Tuple<integer, integer>>[,])

→ boolean

{ mengembalikan true jika currentPosition merupakan kotak yang dapat dikunjungi serta belum pernah dikunjungi pada memo. }

Function IsVisitedTreasure(**input** currentPosition : Tuple<integer, integer>)

→ boolean

{ mengembalikan true jika treasure pada currentPosition sudah pernah dikunjungi.}

Procedure addPath(**input** nextPath : ArrayList)

{ Aksi untuk menggabungkan path baru dengan path yang sudah ada }

nReset++;

KAMUS LOKAL

idx : integer

ALGORITMA

```
idx ← 0;
```

```
for each pathElement in nextPath do
```

```
    if (nReset > 1 and idx > 0) or (nReset <= 1) then
```

```
        Insert(multipleVisitPath, pathElement);
```

```
    idx++;
```

```
Function GetCurrentPath(input fromPosition : Tuple<integer, integer>)
```

```
    → ArrayList
```

```
{ mengembalikan ArrayList yang berisi path dari fromPosition sampai currentPosition}
```

KAMUS LOKAL

```
path : ArrayList
```

```
visitedFromPosition : bool
```

```
tempPosition : Tuple<integer, integer>
```

ALGORITMA

```
visitedFromPosition ← false;
```

```
tempPosition ← position;
```

```
while tempPosition.Item1 != fromPosition.Item1 or tempPosition.Item2 !=  
fromPosition.Item2 do
```

```
    if tempPosition.Item1 != -1 and tempPosition.Item2 != -1 then
```

```
        Insert(path, tempPosition);
```

```
    if tempPosition = fromPosition then
```

```
        if visitedFromPosition then
```

```
            tempPosition = fromPosition;
```

```
        else
```

```
            visitedFromPosition = true;
```

```
    else
```

```
        tempPosition = GetCheckMap(tempPosition).Item2;
```

```
if tempPosition.Item1 != -1 and tempPosition.Item2 != -1 then
```

```
    Insert(path, tempPosition);
```

```
Reverse(path);
```

```
return path;
```

Procedure ShortestPathMove()

{ Berpindah pencarian satu langkah sesuai kaidah BFS dan pendekatan
 backtrack }

KAMUS LOKAL

temp : Tuple<Tuple<integer, integer>, StateInfo>
newPosition : Tuple<integer, integer>

ALGORITMA

// jika sudah ditemukan solusi atau dipastikan tidak ada solusi

if stop **then return**

// ambil elemen terdepan

temp ← _shortestPathQueue.Dequeue()

// buang semua elemen terdepan yang tidak valid untuk dikunjungi

while not IsValid(temp.Item1, temp.Item2.memo) **do**

if not _shortestPathQueue.Empty() **then**

 temp ← _shortestPathQueue.Dequeue()

else

 stop ← **true**

return

// kunjungi titik terdepan pada queue

newPosition ← temp.Item1

// set status titik baru sudah dikunjungi

temp.Item2.memo[newPosition.Item1, newPosition.Item2]

 ← (**true**, temp.Item2.prevPosition)

// set posisi asal yang baru untuk cabang pencarian ini

temp.Item2.prevPosition ← newPosition

position ← newPosition

// jika grid baru belum pernah dikunjungi dalam keseluruhan proses

if (**not** totalMemo[newPosition.Item1, newPosition.Item2]) **then**

```
nodeCount ← nodeCount + 1
totalMemo[newPosition.Item1, newPosition.Item2] ← true

// set matriks pemeriksaan sesuai dengan cabang pencarian saat ini
_checkMap ← temp.Item2.memo

temp.Item2.stepCount++

// sequential mode memastikan jumlah step diperbarui tiap pencarian
if sequentialMode then updateStepCount()

// set atribut foundTreasureCount agar sesuai dengan cabang pencarian
// saat ini
foundTreasureCount ← temp.Item2.foundTreasureCount;

// Jika menemukan treasure yang belum pernah dikunjungi
if (GetMapElmt(position) = "T") then
    // tambah banyak treasure yang sudah ditemukan
    temp.Item2.foundTreasureCount ← temp.Item2.foundTreasureCount + 1
    foundTreasureCount ← foundTreasureCount + 1

// jika sudah menemukan semua treasure
if (temp.Item2.foundTreasureCount ← treasureCount) then
    foundAll ← true

    // belum selesai jika mode TSP
    // set titik awal menjadi belum dikunjungi
    if (tspMode) then
        temp.Item2.memo[initialPosition.Item1, initialPosition.Item2]
            ← defaultCheckValue

    // solusi diperoleh jika bukan mode TSP
    else
        Terminate()
        return

// jika sudah menemukan semua treasure dan kembali ke titik awal
```

```

else if (tspMode and GetMapElmt(position) = "K" and foundAll) then
    Terminate()
    return

// push tetangga yang mungkin dikunjungi jika pencarian belum selesai
for i traversal [3..0]
    _shortestPathQueue.Enqueue(
        ((position.Item1 + direction[i].Item1,
          position.Item2 + direction[i].Item2),
         new StateInfo(temp.Item2))

Procedure Move()
{ Berpindah pencarian satu langkah sesuai kaidah BFS }

KAMUS LOKAL
top                                : Tuple<Tuple<integer, integer>, Tuple<integer,
integer>>
newPosition                        : Tuple<integer, integer>
tempPathBFS                       : ArrayList
temp                              : Tuple<integer, integer>

ALGORITMA

// sudah ditemukan solusi atau dapat dipastikan solusi tidak ada
if stop then return

if not allowMultipleVisits then
    ShortestPathMove()
    return

// buang lokasi tujuan pada queue yang tidak dapat dikunjungi
while _queue.Count > 0 and not IsValid(_queue.Peek().Item1) do
    _queue.Dequeue()

if _queue.Count = 0 then
    stop ← true;

```

```
return

top ← _queue.Peek()
newPosition ← _queue.Dequeue().Item1
treasureFound ← false;

// set nilai _checkMap pada posisi baru menjadi sudah dikunjungi
// serta titik asalnya merupakan titik saat ini
SetCheckMap(newPosition, (true, position))
position ← newPosition

// jika grid baru belum pernah dikunjungi dalam keseluruhan proses
if (not totalMemo[newPosition.Item1, newPosition.Item2]) then
    nodeCount ← nodeCount + 1
    totalMemo[newPosition.Item1, newPosition.Item2] ← true

if sequentialMode then updateStepCount()

if (foundAll and GetMapElmt(position) = "K") then
    fromPosition ← nowPosition;
    tempPathBFS ← GetCurrentPath(fromPosition);
    addPath(tempPathBFS);

// Jika menemukan treasure yang belum pernah dikunjungi
if (GetMapElmt(position) = "T" and not isVisitedTreasure(position)) then
    // tambah banyak treasure yang sudah ditemukan
    foundTreasureCount ← foundTreasureCount + 1
    treasureFound ← true

//Menambah posisi treasure baru ke TPosition
Insert(TPosition, position);

//Mengubah asal dan tujuan untuk path
if foundTreasureCount = 1 then
    fromPosition ← defaultCheckValue.Item2;
    nowPosition ← position;
else
    fromPosition ← nowPosition;
```



```

        nowPosition ← position;

//Menggabungkan Path
tempPathBFS ← GetCurrentPath(fromPosition);
If foundTreasureCount > 0 then
    addPath(tempPathBFS);

// Mereset semua cell kecuali initial menjadi default
for i traversal [0..row-1]
    for j traversal [0..col-1]
        if (i != position.Item1 or j != position.Item2) and (i != 0 or j != 0) then
            temp ← (i,j);
            setCheckMap(temp, defaultCheckValue);
_queue.Clear();

// jika sudah menemukan semua treasure
if (foundTreasureCount = treasureCount) then
    foundAll ← true

    // belum selesai jika mode TSP
    // set titik awal menjadi belum dikunjungi
    if (tspMode) then SetCheckMap(initialPosition, defaultCheckValue)

    // solusi diperoleh jika bukan mode TSP
    else
        Terminate()
        return

// jika sudah menemukan semua treasure dan kembali ke titik awal
else if (tspMode and GetMapElmt(position) = "K" and foundAll) then
    Terminate()
    return

// push tetangga yang mungkin dikunjungi jika pencarian belum selesai
for i traversal [3..0]
    _queue.Enqueue(
        ((position.Item1 + direction[i].Item1,

```

```

        position.Item2 + direction[i].Item2),
    position)

```

Procedure AutoComplete()

{ Menyelesaikan persoalan labirin dengan pendekatan BFS }

while not stop **then**

 Move()

Pendekatan DFS

Atribut Kelas DFSState

```

position          : Tuple<integer, integer>
stop              : boolean
allowMultipleVisits : boolean
tspMode          : boolean
foundAll         : boolean
map              : array[] of (array[] of string)
_checkMap        : array[] of (array [] of
                        Tuple<boolean, Tuple<integer, integer>>)
multipleVisitPath : array[] of Tuple<integer, integer>
stepCount        : integer
foundTreasureCount : integer
treasureCount     : integer
defaultCheckValue : Tuple<boolean, Tuple<integer, integer>>
direction        : array[] of Tuple<integer, integer>
_stack           : Stack<Tuple<Tuple<integer, integer>,
                        Tuple<integer, integer>>>

```

Function GetMapElmt(**input** currentPosition : Tuple<integer, integer>)
→ string

{ Mengembalikan string karakter pada posisi tertentu dalam peta }

Function GetCheckMap(**input** currentPosition : Tuple<integer, integer>)
→ Tuple<boolean, Tuple<integer, integer>>

{ Mengembalikan status pemeriksaan suatu posisi pada peta.
Elemen pertama tuple merupakan status sudah dikunjungi.

```

    Elemen kedua tuple merupakan posisi kotak asal sebelum mengunjungi
    kotak saat ini.
}

Function GetCheckMap(input currentPosition : Tuple<integer, integer>)
    → Tuple<boolean, Tuple<integer, integer>>
{ mengubah atribut stop menjadi true dan memperbarui nilai stepCount }

Procedure SetCheckMap(input currentPosition : Tuple<integer, integer>,
    input checkStatus : Tuple<boolean,
    Tuple<integer, integer>>)

{ mengubah atribut stop menjadi true dan memperbarui nilai stepCount }

Procedure StackEmptyAction()
{ Aksi ketika tidak ada kotak yang dapat dikunjungi }
// jika diperbolehkan mengunjungi kotak > sekali, dalam mode TSP
// serta seluruh harta karun telah ditemukan
if allowMultipleVisits and tspMode and foundAll then

    // jika belum sampai ke titik awal
    if GetMapElmt(position) ≠ "K" then
        position ← GetCheckMap(position).Item2

        // Menambahkan posisi saat ini sebagai jalur solusi
        Insert(multipleVisitPath, position)

    // solusi sudah ditemukan
    else Terminate()

Function IsVisitable(input currentPosition : Tuple<integer, integer>)
    → boolean
{ mengembalikan true jika currentPosition merupakan kotak yang dapat
dikunjungi }

Function IsValid(input currentPosition : Tuple<integer, integer>)

```

→ **boolean**

{ mengembalikan true jika currentPosition merupakan kotak yang dapat dikunjungi serta belum pernah dikunjungi }

Procedure BackTrack()

{ Kembali dari arah sebelumnya sebanyak satu langkah jika
allowMultipleVisit = true, atau kembali ke arah sebelumnya hingga
kembali ke titik asal dari titik tujuan pada elemen stack teratas }

Procedure UpdateStackCount()

{ Memperbarui banyak step untuk rute sementara yang telah ditelusuri,
yaitu sebesar panjang rute dari titik awal hingga titik saat ini }

Procedure Move()

{ Berpindah pencarian satu langkah sesuai kaidah DFS }

KAMUS LOKAL

newPosition : Tuple<integer, integer>
i : integer

ALGORITMA

// sudah ditemukan solusi atau dapat dipastikan solusi tidak ada

if stop **then return**

if _stack.Empty() **then**

 StackEmptyAction()

return

repeat

 // buang lokasi tujuan pada stack yang tidak dapat dikunjungi

while _stack.Count > 0 **and not** IsVisitable(_stack.Top.Item1) **do**

 _stack.Pop();

if _stack.Empty() **then**

 StackEmptyAction()

Return

```

// jika posisi saat ini bukan titik asal dari titik tujuan
// elemen stack teratas
// (titik asal adalah titik yang menyebabkan suatu titik lain
// dimasukkan ke dalam stack sebagai kandidat penelusuran)
if not _stack.Top.Item2 = position then
    BackTrack()
    // membatasi mundur 1 langkah untuk mode multiple visits
    if allowMultipleVisits then
        // memperbarui step count setiap 1 perpindahan untuk mode
        // sequential
        if sequentialMode then UpdateStepCount()
        return

// jika top stack sudah dikunjungi atau tidak dapat dikunjungi
else if not IsValid(_stack.Top.Item1) then
    _stack.Pop()
    if _stack.Empty() then
        StackEmptyAction()
        return

until _stack.Top.Item2 = position and IsValid(_stack.Top.Item1)

// posisi baru pada top stack
newPosition ← _stack.Top.Item1

// set nilai _checkMap pada posisi baru menjadi sudah dikunjungi
// serta titik asalnya merupakan titik saat ini
SetCheckMap(newPosition, (true, position))
position ← newPosition

// jika grid baru belum pernah dikunjungi dalam keseluruhan proses
if (not totalMemo[newPosition.Item1, newPosition.Item2]) then
    nodeCount ← nodeCount + 1
    totalMemo[newPosition.Item1, newPosition.Item2] ← true

// tambah setiap titik yang dikunjungi jika mode multiple visits

```

```

if allowMultipleVisits then Insert(multipleVisitPath, newPosition)

if sequentialMode then updateStepCount()

// Jika menemukan treasure yang belum pernah dikunjungi
if (GetMapElmt(position) = "T") then
    // tambah banyak treasure yang sudah ditemukan
    foundTreasureCount ← foundTreasureCount + 1

    // jika sudah menemukan semua treasure
    if (foundTreasureCount = treasureCount) then
        foundAll ← true

        // belum selesai jika mode TSP
        // set titik awal menjadi belum dikunjungi
        if (tspMode) then SetCheckMap(initialPosition, defaultCheckValue)

        // solusi diperoleh jika bukan mode TSP
        else
            Terminate()
            return

// jika sudah menemukan semua treasure dan kembali ke titik awal
else if (tspMode and GetMapElmt(position) = "K" and foundAll) then
    Terminate()
    return

// push tetangga yang mungkin dikunjungi jika pencarian belum selesai
for i traversal [0..3]
    _stack.Push(
        ((position.Item1 + direction[i].Item1,
          position.Item2 + direction[i].Item2),
         position)

Procedure AutoComplete()
{ Menyelesaikan persoalan labirin dengan pendekatan DFS }

```

```

while not stop then
    Move ()

```

4.2. Penjelasan Struktur Data

4.2.1. Kelas MazeState

Tabel 4.2.1.1 Atribut dan deskripsi kelas MazeState

Atribut	Deskripsi
Tuple<int, int> position	Atribut ini menyimpan posisi saat ini
string[][] map	Atribut ini digunakan untuk menyimpan peta labirin (berisi string karakter K, R, T, atau X untuk setiap posisi)
int nodeCount	Atribut ini menyimpan jumlah <i>node</i> yang pernah ditelusuri selama proses pencarian
int stepCount	Atribut ini menyimpan jumlah langkah dalam rute sementara saat ini dari titik awal
int row	Jumlah baris peta
int col	Jumlah kolom peta
bool foundAll	Atribut ini merupakan penanda semua <i>treasure</i> telah ditemukan
bool tspMode	Atribut ini merupakan penanda mode TSP untuk pencarian
bool allowMultipleVisits	Atribut ini merupakan penanda <i>node</i> boleh dikunjungi lebih dari sekali
bool stop	Atribut ini merupakan penanda pencarian berhenti (bernilai <i>true</i> jika sudah ditemukan solusi atau dipastikan tidak ada solusi)
int treasureCount	Atribut ini digunakan untuk mencatat jumlah <i>treasure</i> yang ada pada <i>maze</i>
int foundTreasureCount	Atribut ini digunakan untuk mencatat jumlah <i>treasure</i> yang telah ditemukan untuk rute sementara saat ini
bool sequentialMode	Atribut ini merupakan penanda perlunya dilakukan <i>update</i> nilai jumlah <i>step</i> pada rute sementara untuk setiap <i>move</i>
bool treasureFound	Atribut ini merupakan penanda adanya <i>treasure</i> yang ditemukan pada <i>move</i> saat ini

bool[][] totalMemo	Atribut ini menyimpan status suatu <i>grid</i> sudah pernah ditelusuri dalam keseluruhan proses pencarian yang pernah dilakukan
Tuple<int, int> initialPosition	Atribut ini menyimpan posisi awal pencarian
Tuple<bool, Tuple<int, int>> [,] _checkMap	Atribut ini menyimpan status sebuah <i>grid</i> telah dikunjungi untuk rute sementara saat ini beserta <i>grid</i> asal sebelum menelusuri <i>grid</i> ini.
ArrayList multipleVisitPath	Atribut ini menyimpan rute solusi khusus untuk mode <i>multiple visits</i>
Tuple<bool, Tuple<int, int>> defaultCheckValue	Atribut ini menyimpan <i>default value</i> dari atribut <i>_checkmap</i> (nilai yang menunjukkan <i>grid</i> belum dikunjungi untuk rute saat ini)
Tuple<int, int>[] directions	Atribut yang menyimpan <i>tuple</i> perpindahan arah atas, kanan, bawah, dan kiri
Dictionary<(int, int), string> directionMap	Atribut ini melakukan konversi <i>tuple</i> perpindahan ke <i>string</i> terkait (U, R, D, atau L)

Tabel 4.2.1.2 Method dan deskripsi kelas MazeState

Method	Deskripsi
MazeState(string[][] map, bool tspMode, bool sequentialMode, bool allowMultipleVisits)	Constructor untuk kelas MazeState
DefaultConfig()	Method yang digunakan untuk inisialisasi atribut-atribut yang tersedia pada kelas MazeState
bool IsValid(Tuple<int, int> target)	Mengembalikan <i>true</i> jika posisi target <i>valid</i> untuk dikunjungi selanjutnya
updateStepCount()	Memperbarui stepCount berdasarkan rute sementara saat ini
Terminate()	Method yang digunakan untuk memberhentikan proses pencarian setelah menemukan solusi
string GetMapElmt(Tuple<int, int> p)	Method yang digunakan untuk mengambil elemen peta <i>maze</i> pada posisi p
Tuple<bool, Tuple<int, int>> GetCheckMap(Tuple<int, int> p)	Method yang digunakan untuk mengambil elemen <i>_checkmap</i> pada posisi p
SetCheckMap(Tuple<int, int> p,	Method yang digunakan untuk meng- <i>set</i> elemen

<code>Tuple<bool, Tuple<int,int>> val)</code>	pada atribut <code>_checkmap</code> pada posisi <code>p</code>
<code>bool [,] GetVisited()</code>	Method yang digunakan untuk mengambil matriks <i>visited</i> (<i>node-node</i> yang telah dikunjungi untuk rute sementara saat ini)
<code>ArrayList GetCurrentRoute()</code>	Method yang digunakan untuk mengambil string rute sementara saat ini
<code>GetCurrentPath()</code>	Method yang digunakan untuk mengambil rute sementara saat ini yang berupa urutan posisi
<code>Move()</code>	Method yang digunakan untuk bergerak berdasarkan algoritma BFS atau DFS
<code>AutoComplete()</code>	Method yang digunakan untuk melakukan iterasi hingga ditemukan solusi
<code>Reset()</code>	Method yang digunakan untuk <i>reset</i> semua atribut kelas sehingga kembali ke posisi pencarian awal

4.2.2. Kelas DFSSState

Tabel 4.2.2.1 Atribut dan deskripsi kelas DFSSState

Atribut	Deskripsi
<code>Stack _stack</code>	Atribut yang menyimpan kandidat <i>node</i> yang akan dikunjungi

Tabel 4.2.2.2 Method dan deskripsi kelas DFSSState

Method	Deskripsi
<code>IsVisitable(Tuple<int, int> target)</code>	Method yang digunakan untuk mengecek apakah <i>grid</i> pada posisi target dapat dikunjungi (tidak mempedulikan status sudah dikunjungi)
<code>Backtrack()</code>	Method yang digunakan untuk runut-balik pada proses pencarian DFS
<code>StackEmptyAction()</code>	Method untuk menjalankan aksi yang harus dilakukan apabila atribut <code>_stack</code> kosong (seluruh <i>node</i> sudah pernah ditelusuri)

4.2.3. Kelas BFSSState

Tabel 4.2.3.1 Atribut dan deskripsi kelas BFSSState

Atribut	Deskripsi
<code>Queue _queue</code>	Atribut yang menyimpan kandidat <i>node</i> yang akan

	dikunjungi
Queue shortestPathQueue	Atribut yang menyimpan kandidat <i>node</i> yang akan dikunjungi dalam mode tidak <i>multiple visits</i>
Tuple <int, int> fromPosition	Atribut yang digunakan untuk menyimpan posisi awal pencarian sebelum pencarian saat ini
Tuple <int, int> nowPosition	Atribut yang digunakan untuk menyimpan posisi awal pencarian saat ini
ArrayList TPosition	Atribut yang digunakan untuk menyimpan posisi dari setiap <i>treasure</i> yang telah ditemukan
Int nReset	Atribut yang digunakan untuk menyimpan banyak program melakukan <i>reset</i> map setelah menemukan <i>treasure</i>
Class StateInfo	Kelas yang menyimpan informasi tentang cabang pencarian BFS tertentu untuk mode tidak <i>multiple visits</i>

Tabel 4.2.3.2 Method dan deskripsi kelas BFSState

Method	Deskripsi
ShortestPathMove()	Method untuk melakukan pencarian dengan algoritma BFS untuk mode tidak <i>multiple visits</i>
isVisitedTreasure(Tuple <int, int> target)	Method untuk memeriksa apakah <i>treasure</i> pada posisi <i>target</i> telah dikunjungi atau belum
addPath(ArrayList nextPath)	Method untuk menambahkan <i>path</i> ke rute akhir

4.2.4. Kelas StateInfo

Tabel 4.2.3.2 Method dan deskripsi kelas BFSState

Atribut	Deskripsi
Int stepCount	Atribut yang digunakan untuk menyimpan jumlah step pencarian untuk cabang rute tertentu
Int foundTreasureCount	Atribut yang digunakan untuk menyimpan jumlah <i>treasure</i> yang telah ditemukan untuk cabang rute tertentu
Tuple<bool, Tuple<int,int>>[,] memo	Atribut yang menyimpan status <i>grid</i> sudah dikunjungi untuk cabang rute tertentu
Tuple<int, int> prevPosition	Atribut untuk menyimpan catatan posisi terdahulu pada cabang rute tertentu
Int row	Jumlah baris pada <i>maze</i>

Int col	Jumlah kolom pada <i>maze</i>
---------	-------------------------------

Tabel 4.2.3.2 Method dan deskripsi kelas BFSState

Method	Deskripsi
getMemo(int row, int col)	Method untuk mengambil elemen pada baris row dan kolom j atribut <i>memo</i>
getMemo()	Method untuk mengembalikan matriks <i>memo</i>
getMemo(Tuple<bool, Tuple<int, int>>[,] other)	Method untuk mengembalikan matriks <i>copy</i> dari memo <i>other</i>
StateInfo(int row, int col)	Constructor Kelas StateInfo
StateInfo(StateInfo other)	Copy-Constructor Kelas StateInfo

4.3. Tata Cara Penggunaan Sistem



Gambar 4.3.1 Tampilan Sistem

Sistem menerima masukan *file* sebagai konfigurasi peta labirin yang akan dicari solusinya. Masukan dapat berupa nama file pada *text box* ataupun menggunakan *file explorer* dengan menekan ikon *file* di kanan *text box* tersebut. Jika masukan nama file menggunakan *text box*, *file* akan dicari menggunakan *path* yang relatif terhadap folder *test* sistem. Masukan menggunakan *file explorer* akan membuat sistem mencari *file* tersebut berdasarkan letak *file* sebenarnya. Perlu diperhatikan bahwa

sistem secara otomatis mendeteksi *file extension* (semua karakter setelah tanda titik) pada nama *file* dan menggantinya menjadi format *txt*.

Pengguna dapat langsung melakukan visualisasi peta jika *file* sudah dimasukkan. Kesalahan nama ataupun isi *file* akan menyebabkan sistem menampilkan pesan *error* di layar. Terdapat beberapa kesalahan *file* yang membuat sistem gagal melakukan visualisasi:

1. *file* tidak ditemukan;
2. *file* kosong;
3. ada spasi pada akhir baris dalam *file*;
4. panjang baris tidak seragam;
5. karakter tidak dipisah spasi;
6. *file* mengandung karakter selain K, T, R, X (tidak *case sensitive*)
7. *file* tidak memiliki karakter K
8. *file* mengandung lebih dari satu karakter K

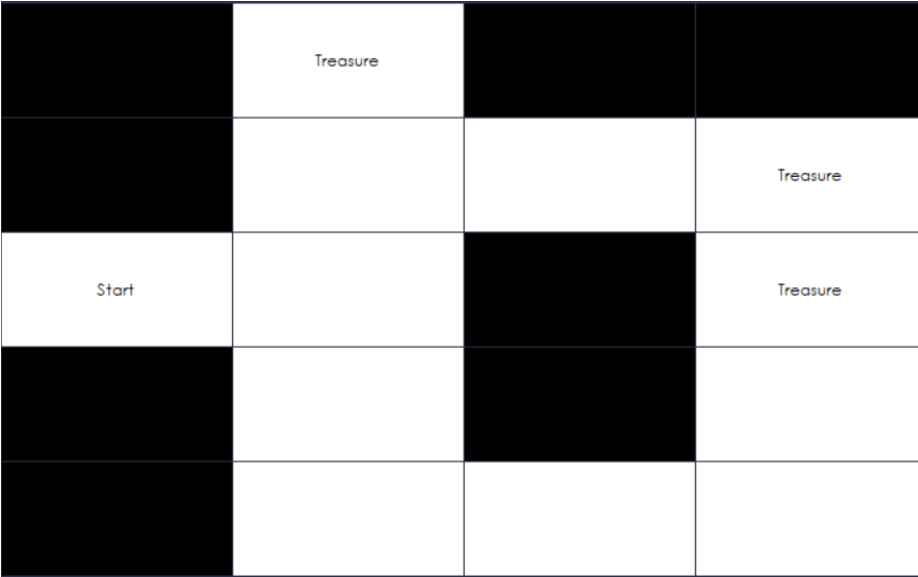

Walaupun visualisasi dapat dilakukan setelah memasukkan *file*, pencarian solusi mewajibkan pengguna setidaknya memilih algoritma BFS atau DFS (pengguna hanya dapat memilih salah satu). Untuk masing-masing algoritma tersebut, terdapat juga pilihan untuk memperbolehkan ataupun tidak memperbolehkan rute solusi untuk melewati grid yang sama lebih dari sekali. Selain itu, pengguna juga dapat memilih mode TSP sehingga solusi yang dihasilkan merupakan rute untuk mendapatkan seluruh *treasure* dan kemudian kembali ke titik awal. Jika salah satu algoritma BFS atau DFS sudah dipilih, pengguna dapat menekan tombol *solve* untuk mendapatkan solusi pencarian.

Tombol *show progress* digunakan untuk menunjukkan proses pencarian berdasarkan algoritma yang dipilih. Jika tombol tersebut ditekan, akan muncul kotak masukan waktu jeda tampilan dalam menampilkan *progress* pencarian tersebut (dalam milisekon). Masukan waktu ini dapat diubah selama proses pencarian untuk mempercepat ataupun memperlambat proses yang ditampilkan. Tombol *show progress* perlu ditekan sebelum menekan *solve*. (Masukan waktu 0 ms akan menyebabkan proses tidak memberikan jeda antar perpindahan kecuali pada algoritma BFS untuk *no multiple visits* yang memiliki jeda waktu independen terhadap masukan tersebut. Jeda waktu independen ini ditambahkan untuk kejelasan proses dan bukan karena masalah performa).

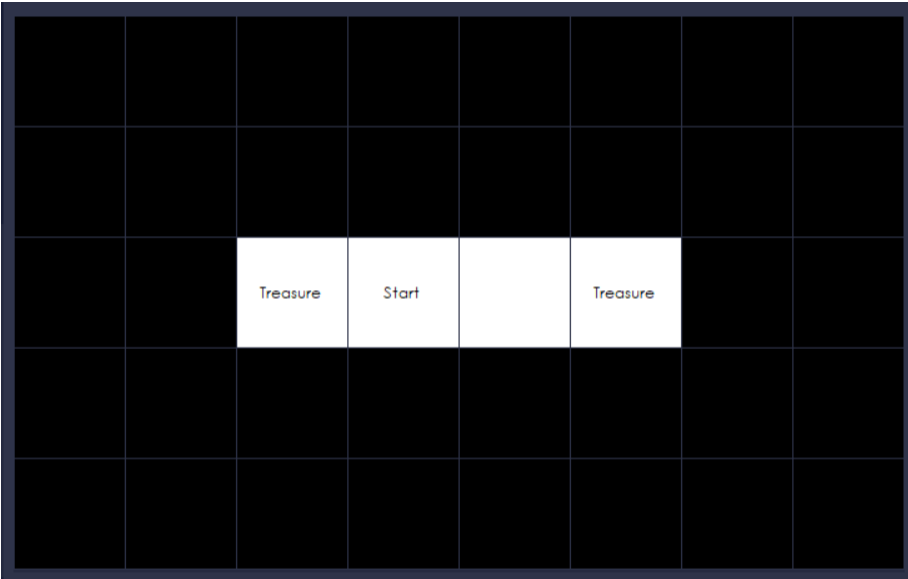
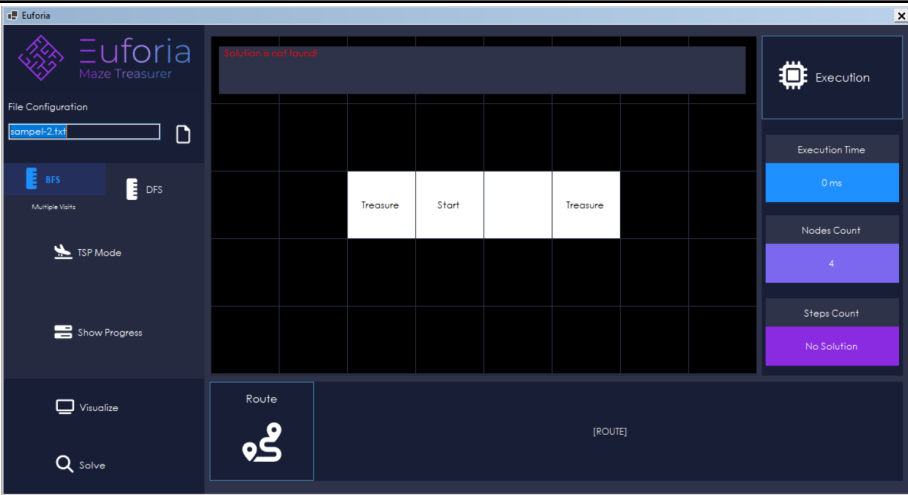
Apabila sistem sudah menemukan solusi, sistem akan menampilkan informasi eksekusi dan pencarian pada kanan dan bawah tampilan. Pada kanan tampilan ditunjukkan waktu eksekusi pencarian (termasuk waktu yang dibutuhkan untuk menampilkan *progress* jika dipilih), jumlah *node/grid* yang diperiksa selama pencarian (*grid* yang ditelusuri lebih dari sekali akan tetap dihitung sebagai 1 *node/grid*), serta jumlah langkah yang diperlukan dalam solusi akhir. Bagian bawah layar menampilkan rute solusi akhir berupa arah perjalanan rute tersebut menggunakan simbol U, R, D, dan L untuk *up*, *right*, *down*, *left*.


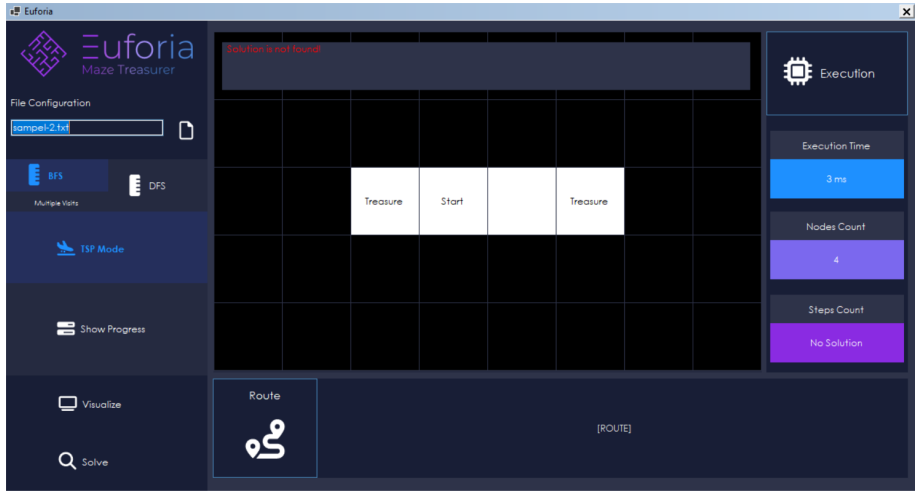
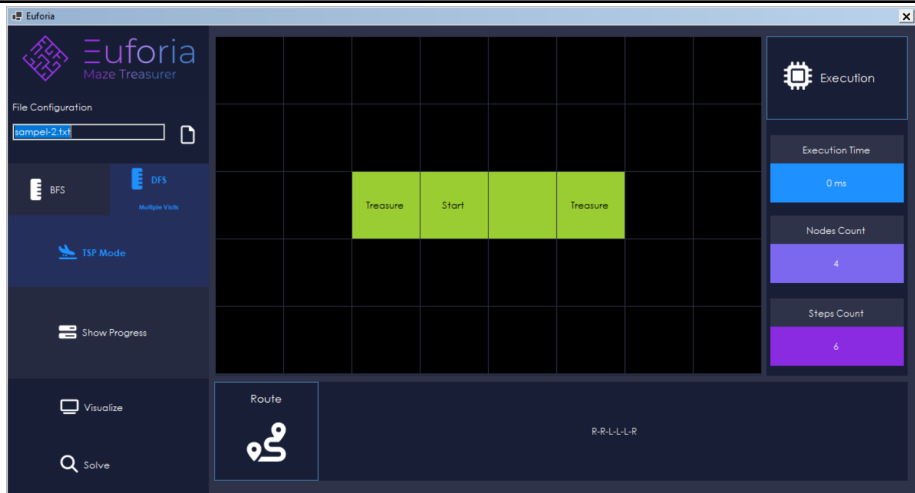
4.4. Hasil Pengujian

No	Input/ Output	Gambar
.		

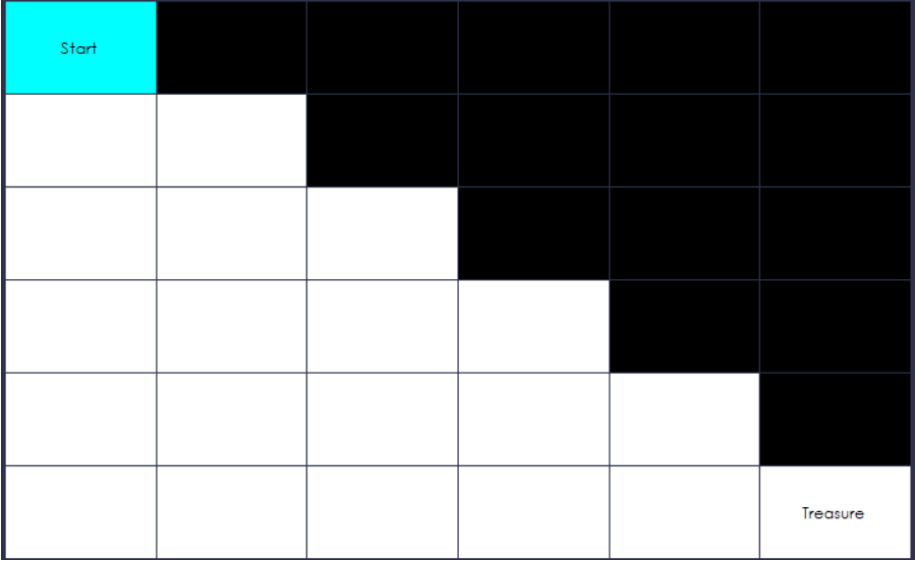

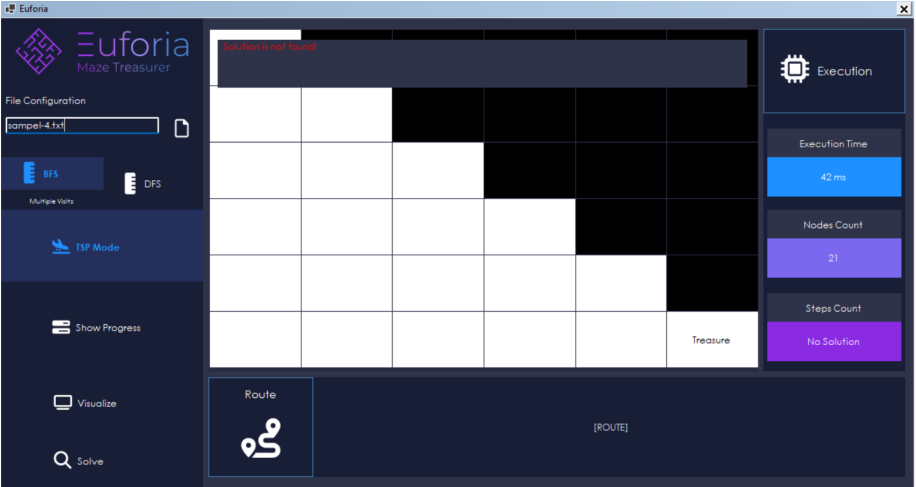
1	<i>Input sampel-1.txt</i>	<pre> X T X X X R R T K R X T X R X R X R R R </pre>			
	<i>Output Visualisasi</i>				
	<i>Output BFS - Multiple visits - Non TSP</i>				

<p><i>Output DFS - Non Multiple visits - Non TSP</i></p>	
<p><i>Output BFS - Multiple visits - TSP</i></p>	
<p><i>Output DFS - Non Multiple visits - TSP</i></p>	

2	<i>Input sampel-2.txt</i>	<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>
	<i>Output Visualisasi</i>	
	<i>Output BFS - Non Multiple visits - Non TSP</i>	

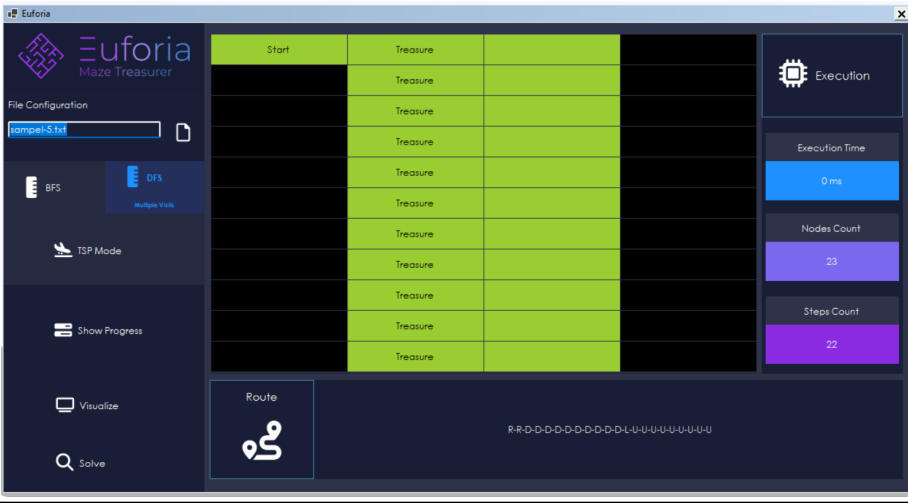
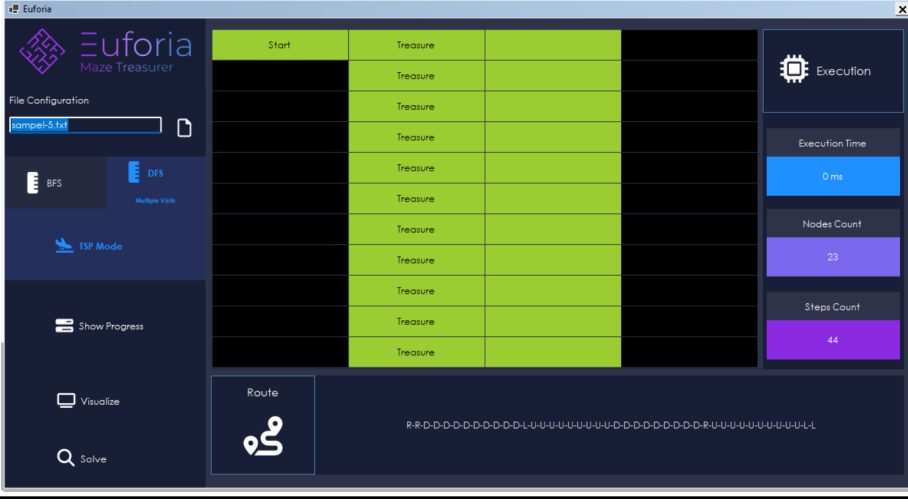
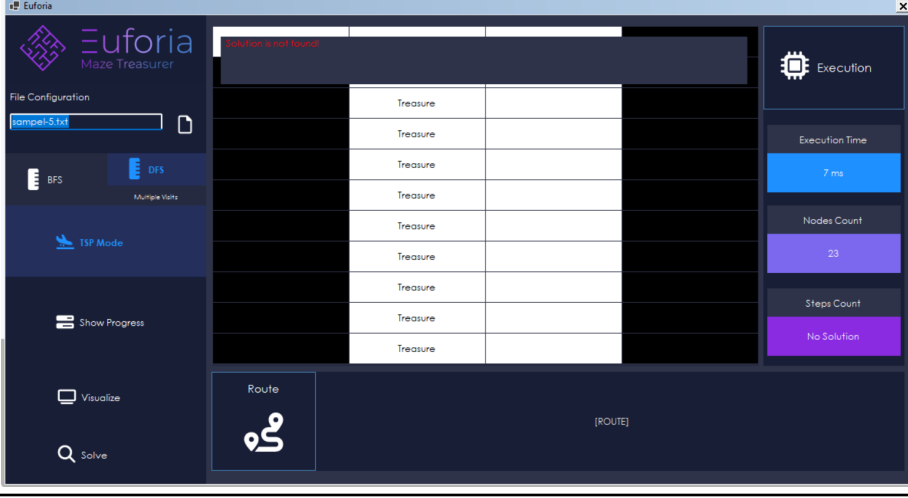
	<i>Output DFS - Multiple visits - Non TSP</i>	
	<i>Output BFS - Non Multiple visits - TSP</i>	
	<i>Output DFS - Multiple visits - TSP</i>	
3	<i>Input sampel-3.txt</i>	<p>J A N G A N L U P A C E K Y A N G B E G I N I Y</p>

	<i>Output Visualisasi</i>	
	<i>Output</i>	
4	<i>Input sampel4-txt</i>	<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>

	<p><i>Output Visualisasi</i></p>	
	<p><i>Output BFS - Non Multiple visits - Non TSP</i></p>	
	<p><i>Output BFS - Non Multiple visits - TSP</i></p>	

[illegible]

5	<i>Output Visualisasi</i>	
	<i>Output BFS - Non Multiple visits - Non TSP</i>	
	<i>Output BFS - Multiple visits - TSP</i>	

<p><i>Output DFS - Multiple visits - Non TSP</i></p>	 <p>Execution Time: 0 ms Nodes Count: 23 Steps Count: 22</p>
<p><i>Output DFS - Multiple visits - TSP</i></p>	 <p>Execution Time: 0 ms Nodes Count: 23 Steps Count: 44</p>
<p><i>Output DFS - Non Multiple visits - TSP</i></p>	 <p>Execution Time: 7 ms Nodes Count: 23 Steps Count: No Solution</p>

4.5. Analisis Desain Solusi

Pada mode yang mengizinkan *multiple visit*, algoritma BFS maupun DFS tidak menjamin diperoleh solusi optimal. Selain itu, solusi salah satu algoritma tidak lebih baik dari algoritma lainnya. Namun, dalam mode TSP, algoritma BFS lebih umum menghasilkan rute yang *straight forward* karena hanya perlu mencari rute terpendek (tetap menggunakan BFS) dari *treasure* yang terakhir ditemukan dan tidak mempedulikan apakah suatu *grid* sudah dilewati pada saat pencarian sebelumnya dalam menemukan *treasure*. Algoritma DFS yang mengizinkan *multiple visit* pada mode TSP akan tetap mencari terlebih dahulu rute yang belum pernah dilalui untuk kembali ke titik awal. Dalam kasus ini, algoritma DFS memiliki kasus terburuk melakukan penelusuran terhadap seluruh *grid* sebelum kemudian melakukan *backtrack* sesuai jalur kedatangan hingga kembali ke titik awal.

Pada mode yang tidak mengizinkan *multiple visit*, kedua algoritma BFS dan DFS mengimplementasikan algoritma *backtrack*. Perbedaanannya adalah algoritma BFS dalam hal ini perlu menyimpan catatan pengunjungan *grid* untuk masing-masing cabang pencarian sehingga masing-masing cabang tidak mempedulikan rute cabang lainnya. Algoritma BFS dengan pendekatan ini terlihat mirip dengan *Iterative Deepening-Search* tetapi tidak melakukan pengulangan pencarian dari awal. Karena menyimpan *array* 2 dimensi untuk masing-masing cabang pencarian, algoritma BFS ini lebih boros memori dibandingkan algoritma DFS yang mengimplementasikan *backtrack*. Namun, efisiensi waktu algoritma DFS ini tidak lebih baik dari algoritma BFS. Walaupun menggunakan memori yang lebih besar, BFS yang mengimplementasikan *backtrack* menjamin solusi paling optimal jika ditemukan solusi (termasuk juga dalam mode TSP) dibandingkan semua algoritma lainnya karena pencarian dimulai dari rute dengan panjang terpendek. Namun, kedua algoritma pada mode yang tidak mengizinkan *multiple visit* tidak menjamin didapatkan solusi untuk setiap konfigurasi peta labirin. Selain itu, waktu eksekusi kedua algoritma dalam mode ini dapat menjadi jauh lebih besar dibandingkan mode yang mengizinkan *multiple visit*.

Secara umum, algoritma BFS lebih natural digunakan apabila ditujukan untuk mendapatkan solusi yang setidaknya mendekati solusi optimal. Hal ini disebabkan jarak antar

node/grid yang sama sehingga *behaviour* algoritma BFS yang melakukan pencarian ke tetangga-tetangga terdekat terlebih dahulu dapat membuat hasil yang diperoleh mendekati rute optimal (atau bahkan dijamin optimal untuk konfigurasi peta yang memenuhi kriteria tertentu). Hal ini juga ditunjukkan dari hasil analisis untuk mode *multiple visit* ataupun tidak.

Bab 5: Kesimpulan, Saran, Refleksi, dan Tanggapan

5.1. Kesimpulan

Algoritma *breadth first search* dan *depth first search* dapat digunakan untuk menyelesaikan persoalan *maze treasure hunt*. Algoritma *breadth first search* melakukan penelusuran dengan melakukan pencarian yang mengutamakan grid dengan tingkat kedalaman / *level* yang sama terlebih dahulu. Grid yang memiliki tingkat kedalaman yang sama adalah grid-grid yang bertetangga dengan grid yang sudah dikunjungi. Sebaliknya, algoritma *depth first search* melakukan penelusuran grid secara mendalam, yaitu mengutamakan penelusuran rute melalui salah satu tetangga terlebih dahulu.

Aplikasi desktop dapat dibuat untuk menunjukkan dan memvisualisasi hasil penelusuran algoritma yang sudah dibuat. Windows Form App Framework dapat digunakan untuk mempermudah pengembang dalam membuat aplikasi desktop. Aplikasi desktop ini juga memudahkan pengguna dalam berinteraksi dengan sistem dibandingkan menggunakan *Command Line Interface*.

5.2. Saran

Berdasarkan proses pengerjaan tugas besar ini, kelompok memiliki saran untuk tim asisten berupa peningkatan kejelasan yang diberikan di spesifikasi tugas. Kejelasan ini seharusnya diupayakan secara lengkap diletakkan pada dokumen spesifikasi dan bukan bergantung pada QNA. Dengan demikian, tidak ada perbedaan asumsi untuk setiap kelompok kecuali untuk kasus-kasus kecil.

5.3. Refleksi

Kelompok merasa tugas besar ini telah menjadi salah satu media untuk melakukan eksplorasi lebih lanjut mengenai penerapan algoritma *Breadth First Search* dan *Depth First Search*. Selain itu, kelompok memiliki kesempatan untuk mengimplementasikan konsep OOP yang telah dipelajari pada mata kuliah Pemrograman Berorientasi Objek. Kelompok juga berkesempatan untuk meningkatkan kemampuan kerja sama tim, mulai dari pembagian tugas yang merata, *testing* program, dan lainnya.

5.4. Tanggapan

Kelompok merasa sangat terbantu karena dari tugas besar ini kelompok memiliki kesempatan melakukan eksplorasi lebih lanjut terhadap bahasa pemrograman C#, terutama eksplorasi pembuatan GUI dengan C#. Selain itu, kelompok juga dapat berkreasi dalam perancangan algoritma pencarian maupun aplikasi desktop yang dibuat.

Daftar Pustaka

- Munir, Rinaldi. 2023. Breadth First Search (BFS) dan Depth First Search (DFS) (Bagian 1).
Homepage Rinaldi Munir Sekolah Teknik Elektro dan Informatika (STEI) ITB.
<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf>
- Munir, Rinaldi. 2023. Breadth First Search (BFS) dan Depth First Search (DFS) (Bagian 2).
Homepage Rinaldi Munir Sekolah Teknik Elektro dan Informatika (STEI) ITB.
<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf>
- Raps MK. 2023. "C# DataGridView Tutorial". Net-informations.com
<http://csharp.net-informations.com/datagridview/csharp-datagridview-tutorial.htm>

LAMPIRAN

Link Github : https://github.com/MarcelRyan/Tubes2_Euforia

Link Video : <https://youtu.be/L7jeQ3MNCgQ>