

Pengaplikasian Algoritma BFS dan DFS dalam Menyelesaikan Persoalan Maze Treasure Hunt

Diajukan sebagai pemenuhan tugas besar II.



Oleh:

1. 13521111 - Tabitha Permalla
2. 13521116 - Juan Christopher Santoso
3. 13521162 - Antonio Natthan Krishna

Dosen Pengampu :

Dr. Ir. Rinaldi Munir, M.T

Dr. Nur Ulfa Mauldievi, S.T, M.Sc

Ir. Rila Mandala, M.Eng, Ph.D

IF2211 - Strategi Algoritma

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

2022

DAFTAR ISI

DAFTAR ISI	2
DAFTAR TABEL	3
DAFTAR GAMBAR	3
BAB 1 DESKRIPSI MASALAH	4
BAB 2 TEORI DASAR	5
2.1. Traversal Graf	5
2.2. Breadth First Search (BFS)	5
2.3. Depth First Search (DFS)	7
2.4. Traveling Salesman Problem (TSP)	9
2.5. C# Desktop Application Development	9
2.6. Graphical User Interface (GUI)	10
BAB 3 ALGORITMA DFS DAN BFS	11
3.1. Algoritma penyelesaian secara umum (general solving algorithm)	11
3.1.1. Simpul (nodes)	11
3.1.2. Struktur data penampung solusi	11
3.1.3. Prioritas arah pencarian	11
3.1.4. Fungsi solusi	12
3.2. Algoritma BFS (Breadth First Search)	12
3.3. Algoritma DFS (Depth First Search)	13
3.4. Implementasi Algoritma Pada Kasus Lain	15
BAB 4 ANALISIS PEMECAHAN MASALAH	17
4.1. Implementasi Program	17
4.2. Struktur Data	36
4.3. Tata Cara Penggunaan Program	38
4.4. Hasil Pengujian	42
4.5. Analisis Desain Solusi	56
BAB 5 KESIMPULAN DAN SARAN	58
5.1. Kesimpulan	58
5.2. Saran	58
5.3. Refleksi	58
5.4. Tanggapan	59
DAFTAR PUSTAKA	60
LAMPIRAN	61

DAFTAR TABEL

Tabel 2.1 Karakteristik metode BFS	6
Tabel 2.2 Karakteristik metode DFS	8
Tabel 4.1 Daftar kelas dan atribut kelas	37

DAFTAR GAMBAR

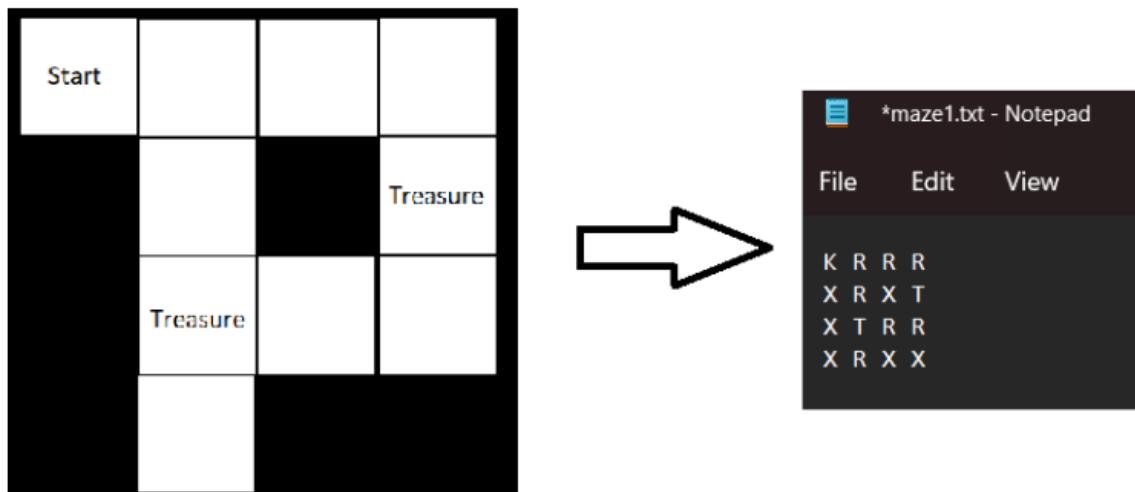
Gambar 1.1 Ilustrasi input maze	4
Gambar 2.1 Ilustrasi pemecahan solusi menggunakan metode BFS	6
Gambar 2.2 Ilustrasi pemecahan solusi menggunakan metode BFS	6
Gambar 2.3 Ilustrasi pemecahan solusi menggunakan metode DFS	7
Gambar 2.4 Ilustrasi pemecahan solusi menggunakan metode DFS	7
Gambar 2.5 Ilustrasi solusi atas permasalahan TSP	9
Gambar 4.1 Struktur data program	37

BAB 1

DESKRIPSI MASALAH

Dalam tugas besar ini, program yang dibuat merupakan 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



Gambar 1.1 Ilustrasi input maze

Dengan memanfaatkan algoritma *Breadth First Search* (BFS) dan *Depth First Search* (DFS), program 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*. Rute yang diperoleh dengan algoritma BFS dan DFS dapat berbeda dan banyak langkah yang dibutuhkan pun menjadi berbeda. Prioritas arah simpul yang dibangkitkan dibebaskan, semisal LRUD (*left right up down*). Tidak ada pergerakan secara diagonal. Program juga perlu memiliki kapabilitas 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, dengan pemilihan warna dibebaskan kepada pengembang program.

BAB 2

TEORI DASAR

2.1. Traversal Graf

Algoritma traversal graf adalah algoritma yang menelusuri/mengunjungi simpul-simpul pada graf dengan cara yang sistematik. Graf yang ditelusuri ini diasumsikan adalah graf terhubung. Apabila graf umumnya digunakan untuk merepresentasikan persoalan, maka algoritma traversal graf digunakan dalam pencarian solusi. Secara umum, algoritma pencarian solusi menggunakan traversal graf dapat dibagi menjadi dua: *uninformed/blind search* atau *informed search*.

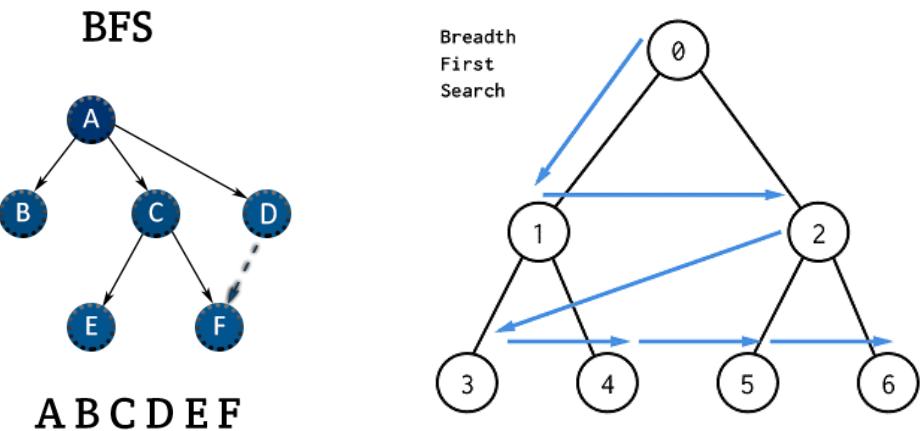
1. *Uninformed/blind search* adalah pencarian solusi yang dilakukan tanpa adanya solusi tambahan. Beberapa contoh dari *uninformed search* adalah BFS, DFS, *Depth Limited Search*, *Iterative Deepening Search*, dan *Uniform Cost Search*.
2. Sedangkan *informed search* adalah pencarian yang berbasis heuristik. Contohnya adalah *Best First Search* dan *A**.

Dalam proses pencarian solusi traversal graf, terdapat dua pendekatan representasi graf, yaitu:

1. Graf statis. Graf statis adalah graf yang sudah terbentuk sebelum proses pencarian dilakukan. Graf statis umumnya direpresentasikan sebagai struktur data.
2. Graf dinamis. Graf dinamis adalah graf yang baru terbentuk saat proses pencarian dilakukan. Graf dinamis tidak tersedia sebelum pencarian dan dibangun selama pencarian solusi.

2.2. Breadth First Search (BFS)

Breadth First Search (BFS) adalah metode pencarian yang dapat dikatakan sebagai metode pencarian melebar. Metode ini disebut sebagai pencarian melebar karena dalam setiap iterasi pencarian, dilakukan pengecekan terlebih dahulu untuk semua simpul yang bertetangga dengan simpul yang sedang ditempati. Dengan kata lain, metode ini akan melakukan pengecekan pada semua simpul yang berada pada level yang sama sebelum melakukan pengecekan kepada simpul yang memiliki level lebih rendah.



Gambar 2.1 dan 2.2 Ilustrasi pemecahan solusi menggunakan metode BFS

Metode BFS merupakan metode pencarian yang berbasis kepada simpul dari suatu graf. Dalam penemuan solusi yang diinginkan, metode BFS menggunakan struktur data *Queue* yang bersandar pada prinsip *First In First Out*. Maka dari itu, simpul-simpul yang berada pada level yang sama akan dilakukan pengecekan terlebih dahulu karena simpul-simpul tersebut terlebih dahulu dimasukkan ke dalam *Queue* penampung. Setelah semua simpul pada level yang sama selesai dilakukan pengecekan, program akan turun ke simpul pada level satu tingkat di bawah simpul sebelumnya, yang pertama kali dimasukkan ke dalam *Queue* penampung.

Karakteristik dari metode BFS dapat dijabarkan menggunakan tabel berikut:

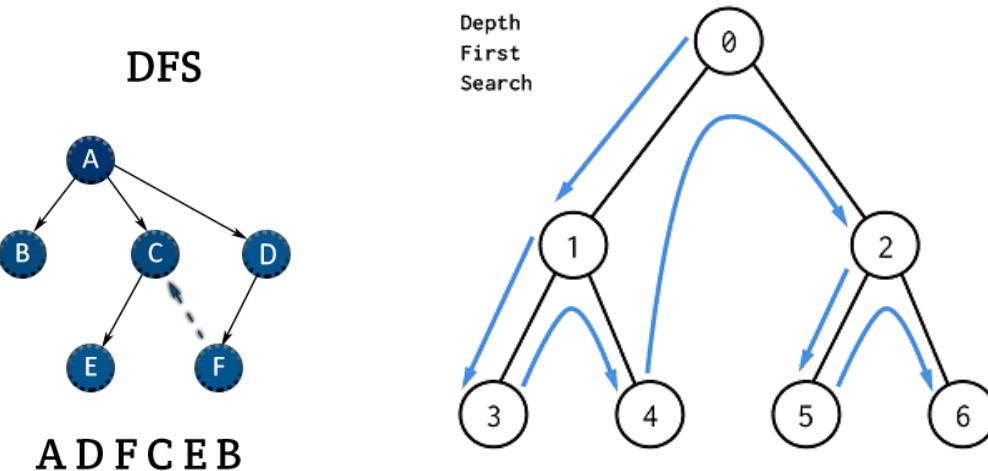
Tabel 2.1 Karakteristik metode BFS

No	Aspek	Keterangan
1.	Struktur data	BFS menggunakan struktur data <i>Queue</i> .
2.	Prinsip	Simpul yang pertama kali ditemui selama pencarian menjadi simpul yang pertama kali keluar dari daftar penampung simpul, <i>First In First Out</i> (FIFO).
3.	Kondisi pemakaian	BFS cocok untuk penyelesaian permasalahan dimana posisi target berdekatan dengan posisi sumber.
4.	Ruang memori (<i>memory space</i>) yang dipakai	BFS membutuhkan ruang memori yang besar.
5.	Kecepatan algoritma	BFS memiliki kecepatan algoritma yang rendah.

6.	Kecocokan dalam aplikasi pohon keputusan (<i>decision tree</i>)	BFS tidak cocok diaplikasikan dalam pohon keputusan. Hal ini dikarenakan BFS perlu melakukan pengecekan pada semua keputusan yang mungkin pada level tertentu sebelum masuk ke keputusan berikutnya.
7.	Optimalitas dalam pencarian rute terpendek (<i>shortest path</i>)	Metode BFS selalu menghasilkan solusi terpendek dalam penyelesaian masalah. Dengan begitu, BFS dianggap optimal dalam menyelesaikan permasalahan pencarian rute terpendek.

2.3. Depth First Search (DFS)

Depth First Search (DFS) adalah metode pencarian yang disebut sebagai pencarian mendalam. Metode ini disebut sebagai pencarian mendalam karena dalam setiap iterasi pencarian, dilakukan pengecekan terhadap tetangga dari suatu simpul selama hal tersebut memungkinkan. Dengan kata lain, metode ini akan terus melakukan pengecekan pada simpul yang berada pada level yang lebih rendah hingga berada pada level paling rendah, atau tidak ada jalan lain yang bisa dilalui, sebelum melakukan pengecekan kepada simpul lain di level yang sama.



Gambar 2.3 dan 2.4 Ilustrasi pemecahan solusi menggunakan metode DFS

Metode DFS merupakan metode pencarian yang berbasis sisi dari suatu graf. Dalam penemuan solusi yang diinginkan, metode DFS menggunakan struktur data *Stack* yang bersandar pada prinsip *Last In First Out*. Maka dari itu, simpul yang bertetanggaan dengan simpul yang sedang dilakukan pengecekan (*current node*) akan dicek terlebih dahulu, mengingat simpul tersebut adalah simpul terakhir yang

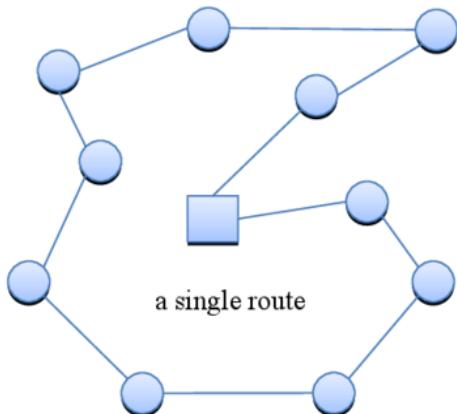
dimasukkan ke dalam *Stack* penampung. Setelah semua kemungkinan jalur di bawah dari suatu simpul dilakukan pengecekan, pengecekan akan berpindah kepada simpul lain yang berada pada level yang sama dari simpul tersebut.

Tabel 2.2 Karakteristik metode DFS

No	Aspek	Keterangan
1.	Struktur data	DFS menggunakan struktur data <i>Stack</i> .
2.	Prinsip	Simpul yang terakhir kali ditemui selama pencarian menjadi simpul yang pertama kali keluar dari daftar penampung simpul, <i>Last In First Out</i> (LIFO).
3.	Kondisi pemakaian	DFS cocok untuk penyelesaian permasalahan dimana posisi target jauh dari posisi sumber.
4.	Ruang memori (<i>memory space</i>) yang dipakai	DFS tidak membutuhkan ruang memori yang besar.
5.	Kecepatan algoritma	DFS memiliki kecepatan algoritma yang tinggi.
6.	Kecocokan dalam aplikasi pohon keputusan (<i>decision tree</i>)	DFS cocok diaplikasikan dalam pohon keputusan. Hal ini dikarenakan DFS akan dengan cepat menemukan keputusan mengingat metode ini akan terus melakukan pengecekan pada simpul selanjutnya.
7.	Optimalitas dalam pencarian rute terpendek (<i>shortest path</i>)	Metode DFS tidak selalu menghasilkan solusi terpendek dari suatu permasalahan, mengingat metode DFS akan terus melakukan pengecekan pada suatu jalur selama hal tersebut memungkinkan. Dengan begitu, DFS tidak optimal dalam menemukan rute terpendek.

2.4. Traveling Salesman Problem (TSP)

Permasalahan *Traveling Salesman Problem* (TSP), merupakan permasalahan yang berbasis pada penggunaan traversal graf. Permasalahan ini berlandaskan kepada pertanyaan yaitu,



“Diberikan suatu graf yang merupakan daftar dari semua kota sebagai simpul beserta jarak yang menghubungkan antar kota sebagai sisinya. Bagaimana cara menentukan lintasan terpendek yang dapat ditempuh seseorang untuk mengunjungi seluruh kota tepat sekali dan kembali ke tempat semula?”

Gambar 2.5 Ilustrasi solusi atas permasalahan TSP

Mengingat setiap simpul hanya dapat dilalui sekali dan simpul terakhir yang dikunjungi merupakan simpul mula-mula, graf yang dipakai dalam permasalahan TSP perlu memenuhi syarat graf yang memiliki sirkuit Hamilton.

Permasalahan TSP dapat diselesaikan baik menggunakan metode BFS maupun DFS. Dalam penyelesaiannya menggunakan metode BFS, rute pertama yang memenuhi syarat solusi merupakan solusi dari permasalahan. Di sisi lain, bila menggunakan metode DFS, diperlukan perbandingan dengan rute lain yang juga memenuhi syarat solusi sampai ditemukan rute solusi dengan jalur terpendek.

2.5. C# Desktop Application Development

C# adalah bahasa pemrograman yang dikembangkan oleh Microsoft dan digunakan secara luas dalam pembangunan berbagai aplikasi desktop. Terdapat sejumlah fitur kunci dari pengembangan aplikasi desktop menggunakan C#.

Pertama-tama, C# adalah bahasa yang berorientasi objek. Hal ini berarti C# dirancang untuk bekerja dengan objek-objek yang merepresentasikan entitas-entitas dan konsep-konsep dalam dunia nyata. Pendekatan ini memperbolehkan pengembang untuk menciptakan aplikasi yang kompleks dengan modularitas dan *reusability* yang tinggi.

Selanjutnya, karena C# men-support berbagai *framework*, termasuk *Windows Forms* dan *WPF*, C# menjadi bahasa pemrograman yang sesuai dalam pembangunan aplikasi berbasis GUI (*graphical user interface*). *Framework* seperti *Windows Forms* dan

WPF menyediakan serangkaian *controls* dan *tools* yang kaya untuk membangun antarmuka yang intuitif dan juga menarik.

Terakhir, C# adalah bahasa yang terintegrasi dengan *framework* .NET. *Framework* .NET sendiri adalah serangkaian *library* dan *tools* yang menyediakan cakupan fungsional yang luas untuk pengembangan aplikasi desktop. Integrasi C# dengan .NET mengizinkan pengembang aplikasi untuk mengambil keuntungan dari suatu lingkungan atau ‘ekosistem’ *tools* dan *libraries* yang besar dan *well-established*. Berkaitan dengan hal ini, dengan diluncurkannya .NET Core, C# menjadi pilihan yang cukup populer untuk pengembangan aplikasi desktop *cross-platform*.

2.6. *Graphical User Interface (GUI)*

Graphical User Interface (GUI) adalah suatu sistem komponen-komponen visual interaktif dari suatu perangkat lunak. Suatu GUI menampilkan objek-objek yang memberikan informasi dan juga merepresentasikan aksi-aksi yang dapat dilakukan pengguna. Umumnya, objek-objek ini berubah warna, ukuran, dan/atau visibilitas ketika seorang pengguna berinteraksi dengannya.

Suatu GUI mengandung berbagai objek GUI, seperti *icon*, *cursor*, *button*, dan lain sebagainya. Elemen-elemen grafis ini terkadang dilengkapi dengan suara ataupun efek-efek visual. Menggunakan objek-objek ini, pengguna dapat menggunakan komputer atau perangkat lunak tanpa harus mengetahui perintah-perintah di baliknya.

Dengan berbagai elemen yang ada, GUI dibuat *se-user-friendly* mungkin. GUI yang didesain dengan baik haruslah intuitif dan mudah untuk digunakan, dengan labeling objek yang jelas dan konsisten. Desain GUI penting karena dapat berdampak pada pengalaman pengguna.

BAB 3

ALGORITMA DFS DAN BFS

3.1. Algoritma penyelesaian secara umum (*general solving algorithm*)

3.1.1. Simpul (*nodes*)

Dalam setiap iterasi yang dilakukan selama pencarian solusi, akan dibuat simpul sebagai penanda jalur yang telah atau akan dilewati. Dalam hal ini simpul yang digunakan dalam program, dibuat menggunakan *class* bernama *Nodes*.

Class Nodes memiliki 3 atribut yaitu *Parent* yang merupakan *class Nodes* (menunjuk pada simpul orang tua), *Node* yang merupakan *class Point* (menunjuk pada simpul petak yang sedang ditempati), dan *Children* yang merupakan array of *Nodes* (menunjuk pada simpul-simpul anak yang dapat dikunjungi).

Pada saat penciptaan *class*, nilai dari *Parent* dan setiap elemen dari *Children* akan di-set sebagai *null*. Ketika penciptaan simpul anak barulah nilai *Parent* pada simpul anak, dan nilai elemen dari *Children* pada simpul orangtua di-set secara berhubungan.

Class Nodes ini digunakan untuk menyimpan semua simpul yang telah dikunjungi selama pencarian solusi, termasuk juga runtutan jalur sehingga simpul tersebut dapat dikunjungi. *Class* ini juga yang digunakan untuk menghitung jumlah *nodes* yang perlu ditampilkan pada GUI.

3.1.2. Struktur data penampung solusi

Runtutan jalur solusi ditampung dalam *Stack of Point*. Simpul yang disimpan pada *Stack* tersebut adalah deretan simpul mulai dari simpul lokasi mula-mula hingga simpul lokasi yang sedang ditempati. Penambahan simpul baru pada jalur yang sedang dilalui dilakukan dengan prosedur *Push* pada *Stack* tersebut. Di sisi lain, pelaksanaan *backtracking* dilakukan dengan fungsi *Pop* pada *Stack* tersebut.

Cara penyimpanan *Stack* penampung solusi tentunya berbeda dalam implementasi BFS dan DFS. Dalam metode BFS, *Stack of Point* tersebut disimpan dalam sebuah *Queue* sehingga dibentuklah *Queue of <Stack of Point>*. Di sisi lain, pada metode DFS, dapat dibentuk *Stack of <Stack of Point>* untuk menyimpan jalur-jalur selama pencarian dilakukan.

3.1.3. Prioritas arah pencarian

Prioritas arah dalam pencarian solusi secara berturut-turut dari prioritas tertinggi adalah ULRD (*Up, Left, Right, Down*). Urutan prioritas inilah yang menjadi landasan dari rute pencarian solusi yang dikembangkan.

Namun, terdapat beberapa *adjustment* terhadap urutan prioritas tersebut. Urutan prioritas arah gerak dapat berubah bergantung kepada kondisi rute selama pencarian berlangsung. Hal ini akan dijelaskan lebih lanjut pada bagian penjelasan metode BFS dan DFS.

3.1.4. Fungsi solusi

Pada setiap iterasi selama pencarian berlangsung, program akan selalu melakukan pengecekan mengenai apakah rute yang telah dilewati sejauh ini sudah dapat dianggap sebagai sebuah solusi.

Dalam pembacaan *class Map* dari *file txt*, setiap petak *treasure* disimpan dalam sebuah *array of Point*. Array penyimpan lokasi *treasure* ini yang menjadi tolak ukur mengenai apakah sebuah rute sudah bisa dianggap sebagai sebuah solusi.

Sebuah rute dianggap sebagai sebuah solusi apabila setiap *Point* yang disimpan pada array penyimpan lokasi *treasure* telah dikandung pada rute yang dilewati (di dalam *Stack of Point*). Untuk menyelesaikan permasalahan TSP, syarat fungsi solusi yang diterapkan ditambah dengan simpul yang dikunjungi terakhir kali haruslah simpul yang sama dengan simpul mula-mula.

3.2. Algoritma BFS (*Breadth First Search*)

3.2.1. Mapping persoalan

Terdapat dua komponen utama pembentuk suatu graf, yaitu simpul (*vertex*) dan sisi (*edge*). Dalam penyelesaian persoalan *maze*, elemen-elemen *maze* dipetakan sebagai berikut:

1. Simpul : jalur pada *maze* (kumpulan grid yang sudah ditelusuri)
2. Sisi : langkah pergerakan (atas, kiri, kanan, bawah)

Dalam penyelesaian persoalan ini, digunakan representasi graf dinamis. Sesuai dengan penjelasan dalam sub-bab sebelumnya, urutan simpul yang dibangkitkan ditentukan menurut prioritas arah pencarian, yaitu ULRD.

3.2.2. Algoritma

Langkah-langkah penyelesaian persoalan *maze* dengan algoritma BFS adalah sebagai berikut:

1. Simpul pertama (simpul akar) adalah *stack* yang berisi grid K, titik mulai permainan.
2. Simpul-simpul selanjutnya dibangkitkan sesuai dengan prioritas ULRD, dengan menambahkan grid atas, kiri, kanan, atau bawah dari grid K yang mungkin untuk dikunjungi (bukan grid X) ke *stack* sebelumnya. *Stack* yang terbentuk kemudian ditambahkan ke dalam suatu *queue*. (Perlu diperhatikan bahwa tiap-tiap grid atas, kiri, kanan, dan bawah ditambahkan ke *stack* pertama, dan tiap-tiap *stack* ditambahkan ke *queue*).

3. Selanjutnya dilakukan pencarian solusi untuk tiap-tiap anak dari *stack* yang pertama, yaitu *stack-stack* yang terbentuk pada langkah (2) dan yang telah disimpan dalam *queue*. Pencarian solusi dilakukan sesuai dengan urutan pada *queue*.
4. Apabila *stack* yang diperiksa (sebut saja *stack A*) bukan merupakan solusi (belum semua grid *treasure* dilewati), maka *stack A* dikeluarkan dari *queue*. Di saat yang sama, dilakukan pembangkitan simpul-simpul baru seperti di langkah (2), tapi dengan *stack A* sebagai *stack* dasarnya. Simpul atau *stack* yang dibangkitkan ini ditambahkan ke akhir *queue*.
5. Pemeriksaan *stack* dan pembangkitan simpul terus dilakukan sampai ditemukan jalur yang telah melewati semua *treasure*.
6. Apabila ingin menyelesaikan *maze* dengan TSP, maka setelah semua *treasure* ditemukan, dilakukan BFS lagi untuk kembali ke titik mulai.

3.2.3. Evaluasi Teknik Pencarian

3.2.3.1. Completeness

Selama *maze* mungkin untuk diselesaikan, *maze* pasti dapat diselesaikan dengan algoritma BFS di atas.

3.2.3.2. Optimality

Algoritma BFS pasti akan menghasilkan solusi dengan langkah paling sedikit.

3.2.3.3. Time Complexity

$O(b^d)$, dengan $b = 4$, dan d adalah jumlah langkah.

3.2.3.4. Space Complexity

$O(b^d)$, dengan $b = 4$, dan d adalah jumlah langkah. Dalam hal ini, kompleksitas ruang dinilai kurang baik, karena simpul yang dibangkitkan menjadi sangat banyak.

3.3. Algoritma DFS (*Depth First Search*)

3.3.1. Mapping persoalan

Pemetaan persoalan masih sama dengan pemetaan yang digunakan dalam algoritma BFS, hanya saja algoritma pemecahan solusinya yang berbeda.

Terdapat dua komponen utama pembentuk suatu graf, yaitu simpul (*vertex*) dan sisi (*edge*). Dalam penyelesaian persoalan *maze*, elemen-elemen *maze* dipetakan sebagai berikut:

3. Simpul : jalur pada *maze* (kumpulan grid yang sudah ditelusuri)
4. Sisi : langkah pergerakan (atas, kiri, kanan, bawah)

Dalam penyelesaian persoalan ini, digunakan representasi graf dinamis. Sesuai dengan penjelasan dalam sub-bab sebelumnya, urutan simpul yang dibangkitkan ditentukan menurut prioritas arah pencarian, yaitu ULRD.

3.3.2. Algoritma

Langkah-langkah penyelesaian persoalan *maze* dengan algoritma DFS adalah sebagai berikut:

1. Algoritma DFS yang digunakan pada program mengandung dua jenis algoritma DFS. Algoritma pertama adalah algoritma penyelesaian DFS dimana selama pencarian solusi, program tidak boleh kembali ke petak di rute yang telah dilewati. Algoritma berikutnya adalah algoritma metode DFS dimana program dapat kembali ke petak yang telah dilewati dengan syarat tertentu.
2. Untuk setiap kali program dijalankan, program akan mencoba untuk mencari solusi menggunakan algoritma pertama. Namun, bila tidak ada solusi yang didapatkan atau algoritma tersebut mengembalikan *stack* kosong, maka algoritma kedua akan dijalankan. Untuk pemecahan masalah TSP, algoritma pertama akan *di-skip* mengingat TSP harus kembali ke petak yang telah dilewati (petak mula-mula).
3. Kedua algoritma yang digunakan sama-sama menggunakan rekursif untuk menyelesaikan permasalahan. Dengan begitu, untuk setiap langkah yang diambil oleh program, fungsi yang sama akan dipanggil kembali, tetapi dengan parameter fungsi yang berbeda.
4. Prioritas dalam pemilihan arah pergerakan secara berturut-turut dari prioritas tertinggi adalah ULRD. Namun, untuk algoritma kedua, terdapat *adjustment* pada prioritas tersebut. Mengingat pada algoritma kedua program dapat kembali ke petak yang telah dilewati, digunakan konsep *visitedCount* untuk setiap petak pada map. Semakin besar *visitedCount* sebuah petak maka semakin rendah nilai prioritas petak tersebut dalam pemilihan arah pergerakan. Bila terdapat lebih dari 1 petak yang memiliki nilai *visitedCount* yang sama, maka urutan prioritas awal akan digunakan kembali. Konsep ini dilakukan untuk mencegah program kembali ke petak semula bila tidak dibutuhkan.
5. Program akan selesai dijalankan bila ditemukan sebuah *stack of Point* yang mengandung semua lokasi dari *treasure* yang ada pada *Map*. Hal ini ditandai dengan dikembalikannya *stack of Point* tersebut kepada program utama. Untuk pemecahan persoalan TSP, fungsi solusi yang digunakan akan ditambah dengan dilakukannya pengecekan terhadap kesamaan petak posisi terakhir dengan petak posisi mula-mula.

3.3.3. Evaluasi Teknik Pencarian

3.3.3.1. Completeness

Selama *maze* mungkin untuk diselesaikan, *maze* pasti dapat diselesaikan dengan algoritma DFS di atas.

3.3.3.2. Optimality

Algoritma DFS tidak selalu akan menghasilkan solusi dengan langkah paling sedikit.

3.3.3.3. Time Complexity

$O(b^m)$, dengan $b = 4$, dan m adalah jumlah langkah maksimum. Dalam hal ini, kompleksitas waktu kurang baik, khususnya karena nilai m tidak menentu dan dapat menjadi sangat besar.

3.3.3.4. Space Complexity

$O(bm)$, dengan $b = 4$, dan m adalah jumlah langkah maksimum. Dalam hal ini, kompleksitas ruangnya jauh lebih baik dari BFS.

3.4. Implementasi Algoritma Pada Kasus Lain

3.4.1. Social Networking Sites

Social Networking Sites merupakan fitur rekomendasi pada sebuah aplikasi dimana akan terdapat daftar akun rekomendasi dari social network yang dimiliki oleh pengguna. Misalkan pada LinkedIn, pengguna dapat diberikan rekomendasi connection baru yang mana daftar rekomendasi tersebut didapatkan dari network connection yang telah pengguna tersebut bangun. Umumnya, daftar rekomendasi merupakan koneksi dari daftar orang yang telah menjadi koneksi dari pengguna tersebut. Pencarian koneksi dari koneksi ini biasanya menggunakan algoritma BFS dan DFS dengan batasan depth pencarian (misalkan hingga depth 3).

3.4.2. GPS Navigation System

Mirip seperti persoalan *maze* yang menjadi tugas besar kali ini, GPS Navigation System yang kita gunakan sehari hari melalui aplikasi seperti Google Maps, Apple Maps, dan Maze, menggunakan algoritma BFS dan DFS untuk mencari jarak terdekat dari dua buah titik. Persoalan optimasi ini dapat diselesaikan dengan menggunakan algoritma DFS dan BFS. Algoritma DFS dan BFS membantu penelusuran dan mencari nilai efektif tanpa harus melakukan Exhaustive Search karena terdapat banyak sekali titik yang terdapat pada GPS Navigation System.

3.4.3. Web Exploration

Halaman *web* dapat digambarkan menjadi suatu graf berarah. Simpul menyatakan halaman *web* dan sisi menyatakan *link* menuju suatu halaman *web*. *Link* yang berada pada suatu *web* dapat mengarahkan *user* menuju suatu laman *web* yang baru. Dengan begitu, penjelajahan *web* dapat dilakukan baik menggunakan BFS ataupun DFS, dimulai dari *page* awal menelusuri setiap *link* yang ada pada *website* tersebut sampai ditemukan *web page* yang tidak mengandung *link*.

3.4.4. *File Exploration*

Memiliki persoalan yang serupa dengan *web*, implementasi dari DFS dan BFS dapat digunakan untuk melakukan penelusuran *folder*. Dalam hal ini, *local disk* dapat dianggap sebagai sebuah *root* yang mengandung beberapa *folders* yang dapat disimpulkan sebagai simpul (*nodes*). Penelusuran baik menggunakan BFS dan DFS akan terus dilakukan sampai ditemukan *file* yang diinginkan.

BAB 4

ANALISIS PEMECAHAN MASALAH

4.1. Implementasi Program

<i>namespace Class</i>
Class Map
Atribut:
row : int col : int nTreasure : int treasureLocs : array of Point startLoc : Point curLoc : Point buffer : array of (array of Tile) valid : bool
Method:
{ Constructor } Map() this.row <- 0 this.col <- 0 this.nTreasure <- 0 this.treasureLocs <- new Point[] {} this.startLoc <- new Point() this.curLoc <- new Point() this.buffer <- new Tile[0,0] {} this.valid <- true { Setter Getter } procedure setRow(input int r) row <- r procedure setCol(input int c) col <- c function getRow() -> int -> row function getCol() -> int -> col procedure setCurLoc(input Point p) this.curLoc <- p

```

function getCurLoc() -> Point
-> this.curLoc

function getStartLoc() -> Point
-> this.startLoc

function getnTreasure() -> int
-> this.nTreasure

function getValueAtCoordinate(int r, int c) -> char
-> this.buffer[r,c].getValue()

function getValueAtCoordinate(Point p) -> char
-> this.buffer[p.getRow(), p.getCol()].getValue()

procedure setValueAtCoordinate(input Point p, input char c)
this.buffer[p.getRow(), p.getCol()].setValue(c)

procedure setVCAtCoordinate(input Point p, input int n)
this.buffer[p.getRow(), p.getCol()].setVisitedCount(n)

function getVCAtCoordinate(int r, int c) -> int
-> this.buffer[r, c].getVisitedCount()

function getVCAtCoordinate(Point p) -> int
-> this.buffer[p.getRow(), p.getCol()].getVisitedCount()

procedure increaseVCAtCoordinate(input Point p)
this.buffer[p.getRow(), p.getCol()].increaseVisitedCount()

function getTreasureLocations() -> array of Point
-> this.treasureLocs

procedure setValidTrue()
this.valid <- true

procedure setValidFalse()
this.valid <- false

function getValid() -> bool
-> this.valid

{ Print and Display }
procedure displayMap()
i traversal [0...this.row]
    output("[ ")
    j traversal [0...this.col]
        if (j = this.col-1) then
            output(this.buffer[i,j].getValue())
        else

```

```

        output(this.buffer[i,j].getValue())
        output(' ')
    output(" ]")
    output()

{ Reading File }
procedure IdentifyFile(input string textField)
    nCol <- 0
    nRow <- 0
    lines <- File.ReadAllLines(textFile)
    foreach (string line in lines)
        if (nCol = 0) then
            foreach(char c in line)
                if (c != ' ')
                    nCol <- nCol + 1
                if (c != 'K' and c != 'X' and c!= 'R' and
c != 'T') then
                    setValidFalse()
                nRow <- nRow + 1
            this.setCol(nCol)
            this.setRow(nRow)

procedure ReadFile()
    output("Enter File Name: ")
    fileName <- Console.ReadLine()
    fileName <- fileName + ".txt"
    path <- Directory.GetCurrentDirectory()
    fullPath <- path + "/test/" +fileName

    IdentifyFile(fullPath)
    if (getValid()) then
        this.buffer <- new Tile[this.row, this.col]
        i traversal [0...row]
            j traversal [0...col]
                this.buffer[i, j] <- new Tile()

    lines <- File.ReadAllLines(fullPath)

    nCol <- 0
    nRow <- 0
    foreach (string line in lines)
        nCol <- 0
        foreach (char c in line)
            if (c != ' ') then
                this.buffer[nRow, nCol].setValue(c)
            if (c = 'T')
                this.nTreasure <- this.nTreasure + 1
                addTreasureLocation(nRow, nCol)
            else if (c = 'K')
                this.curLoc <- new Point(nRow, nCol)
            nCol++

```

```

        nRow++
        this.startLoc.copyPoint(this.curLoc)
    else
        output("Map Reading Failed. Invalid Map Detected.")

{ Other Methods }
procedure changeCurLoc(input char c)
    if (c = 'L') then
        this.curLoc.goLeft()
    else if (c = 'R') then
        this.curLoc.goRight()
    else if (c = 'U') then
        this.curLoc.goUp()
    else if (c = 'D') then
        this.curLoc.goDown()

procedure addTreasureLocation(input int r, input int c)
    treasure <- new Point(r,c)
    temp <- (Point[])this.treasureLocs.Clone()
    this.treasureLocs <- new Point[temp.Length+1]
    i traversal [0...this.treasureLocs.Length]
    if (i = this.treasureLocs.Length -1)
        this.treasureLocs[i] <- treasure
    else
        this.treasureLocs[i] <- temp[i]

procedure getInfo()
    if (getValid())
        displayMap()
        output("Row: " +getRow())
        output("Col: " +getCol())
        output("Treasure Amount: " +getnTreasure())
        output("Treasure Locations: ")
        displayTreasureLocations()
        output("Starting Location: ")
        this.startLoc.displayPoint()

void resetMap()
    if (getValid())
        this.curLoc.copyPoint(this.startLoc)
        i traversal [0...row]
        j traversal [0...col]
        setVCAtCoordinate(new Point(i, j), 0)

```

Class Nodes

Atribut:

parent : Nodes
node : Point

```
children : array of Nodes
```

Method:

```
{ Constructor }
Nodes()
    this.node <- new Point()
    this.parent <- null
    this.children <- new Nodes[4] {null, null, null, null}

Nodes(Point p)
    this.node <- new Point(p)
    this.parent <- null
    this.children <- new Nodes[4] {null, null, null, null}

{ Setter Getter }
procedure setNode(input Point p)
    this.node.copyPoint(p)

procedure setParent(input Nodes n)
    this.parent <- n

procedure setUpChild(input Point p)
    this.children[0] <- new Nodes()
    this.children[0].setNode(p)
    this.children[0].setParent(this)

procedure setLeftChild(input Point p)
    this.children[1] <- new Nodes()
    this.children[1].setNode(p)
    this.children[1].setParent(this)

procedure setRightChild(input Point p)
    this.children[2] <- new Nodes()
    this.children[2].setNode(p)
    this.children[2].setParent(this)

procedure setDownChild(input Point p)
    this.children[3] <- new Nodes()
    this.children[3].setNode(p)
    this.children[3].setParent(this)

function getParent() -> Nodes
    -> this.parent

function getNode() -> Point
    -> this.node

function getChildren() -> array of Nodes
    -> this.children
```

```

function setUpChild() -> Nodes
    -> this.children[0]

function getLeftChild() -> Nodes
    -> this.children[1]

function getRightChild() -> Nodes
    -> this.children[2]

function getDownChild() -> Nodes
    -> this.children[3]

{ Print and Display }
procedure displayRoutes(input int h, input int depth)
    j traversal [0...h*depth]
        output(' ')
    getNode().displayPoint()
    foreach(Nodes childN in getChildren())
        if (childN != null) then
            childN.displayRoutes(h, depth+1)

{ Other Methods }
function getNChild() -> int
    count <- 0
    foreach(Nodes n in this.children)
        if (n != null)
            count <- count + 1
    -> count

function getNodesAmount() -> int
    count <- 1
    foreach(Nodes newN in children)
        if(newN != null)
            count <- count + newN.getNodesAmount()
    -> count

```

Class Point

Atribut:

row : int
 col : int

Method:

```

{ Constructor }
Point ()
    row <- 0
    col <- 0

```

```

Point(int r, int c)
    row <- r
    col <- c

Point(Point p)
    this.row <- p.getRow()
    this.col <- p.getCol()

{ Setter Getter }
function getRow() -> int
    -> row

function getCol() -> int
    -> col

procedure setRow(input int r)
    this.row <- r

procedure setCol(input int c)
    this.col <- c

{ Print and Display }
procedure displayPoint()
    output("(" +this.row +", "+this.col+ ")")

{ Override Object Properties }
function override Equals(object obj) -> bool
    if (obj = null or GetType() != obj.GetType()) then
        -> false
    p <- (Point) obj
    -> (getRow() = p.getRow() and getCol() = p.getCol())

function override GetHashCode() -> int
    -> (row,col).GetHashCode()

{ Other Methods }
procedure copyPoint(input Point p)
    this.row <- p.getRow()
    this.col <- p.getCol()

procedure goLeft()
    this.col--

procedure goRight()
    this.col++

procedure goUp()
    this.row--

procedure goDown()

```

```

        this.row++

function isLeftOf(Point p) -> bool
    -> (this.getRow() = p.getRow() and this.getCol() =
p.getCol()-1)

function isRightOf(Point p) -> bool
    -> (this.getRow() = p.getRow() and this.getCol() =
p.getCol()+1)

function isUpOf(Point p) -> bool
    -> (this.getRow() = p.getRow()-1 and this.getCol() =
p.getCol())

function isDownOf(Point p) -> bool
    -> (this.getRow() = p.getRow()+1 and this.getCol() =
p.getCol())

```

Class Tile

Atribut:

```

value : char
visitedCount : int

```

Method:

```

{ Constructor }
Tile()
    value <- '.'
    visitedCount <- 0

{ Setter Getter }
procedure setValue(input char c)
    value <- c

procedure setVisitedCount(input int vc)
    visitedCount <- vc

function getValue() -> char
    -> value

function getVisitedCount() -> int
    -> visitedCount

{ Other Methods }
procedure increaseVisitedCount()
    visitedCount <- visitedCount + 1

```

namespace Method

Class Solver

Atribut:

```
nodes : int
steps : int
solRoutes : array of char
solPaths : array of Point
routeNodes : Nodes
watch : Stopwatch
listCurLoc : array of Point
```

Method:

```
{ Constructor }
Solver()
    this.nodes <- 0
    this.steps <- 0
    this.solRoutes <- new char[] {}
    this.solPaths <- new Point[] {}
    this.routeNodes <- new Nodes()
    this.watch <- new Stopwatch()
    this.listCurLoc <- new Point[] {}

{ Setter Getter }
procedure setNodes (input int n)
    this.nodes <- n

procedure setSteps (input int s)
    this.steps <- s

function getNodes () -> int
    -> this.nodes

function getSteps () -> int
    -> this.steps

function getSolRoutes () -> array of char
    -> this.solRoutes

function getSolPaths () -> array of Point
    -> this.solPaths

function getRoute () -> Nodes
    -> this.routeNodes

function getListCurLoc () -> array of Point
    -> this.listCurLoc
```

```

procedure setListCurLoc(input Point[] newListCurLoc)
    this.listCurLoc = newListCurLoc;

{ Print and Display }
procedure displaySolutionRoutes()
    output("(")
    i traversal [0...this.solRoutes.Length]
        if (i = this.solRoutes.Length-1) then
            output(this.solRoutes[i])
        else
            output(this.solRoutes[i]+", ")
    output(")")

procedure displaySolutionPaths()
    if (solPaths.Length != 0) then
        i traversal [0...this.solPaths.Length]
            if (i % 5 = 0) then
                output("(")
            if (i = this.solPaths.Length - 1 and i % 5 != 4)
then
                this.solPaths[i].displayPoint()
                output(")")
            else
                if (i % 5 != 4) then
                    this.solPaths[i].displayPoint()
                    output(" -> ")
                else
                    this.solPaths[i].displayPoint()
                    output(")")
            else
                output("No solution is stored.")

{ Other Methods }
procedure startTime()
    this.watch.Start()

procedure stopTime()
    this.watch.Stop()

function getExecutionTime() -> long
    -> this.watch.ElapsedMilliseconds

procedure getInfo(input bool displayNodes)
    output("Execution Time: " + getExecutionTime() + " ms")
    output("Total Nodes: " + getNodeCount())
    output("Total Steps: " + getStepCount())
    output("Solution Route: ")
    displaySolutionRoutes()
    output("Solution Paths: ")
    displaySolutionPaths()

```

```

if (displayNodes)then
    output("Constructed Nodes: ")
    routeNodes.displayRoutes(2, 0)

function insertLastRoutes(char[] routes, char c) -> array of char
    temp <- new char[routes.Length+1]
    i traversal [0...temp.Length]
        if(i = temp.Length-1) then
            temp[i] <- c
        else
            temp[i] <- routes[i]
    -> temp

function deleteFirstRoutes(char[] routes) -> array of char
    temp <- (char[])routes.Clone()
    routes <- new char[temp.Length-1]
    i traversal [0...routes.Length]
        routes[i] <- temp[i+1]
    -> routes

function insertLastPaths(Point[] paths, Point p) -> array of Point
    temp <- new Point[paths.Length+1]
    i traversal [0...temp.Length]
        if (i = temp.Length-1)
            temp[i] <- p
        else
            temp[i] <- paths[i]
    -> temp

function deleteFirstPaths(Point[] paths) -> array of Point
    temp <- (Point[])paths.Clone()
    paths <- new Point[temp.Length-1]
    i traversal [0...paths.Length]
        paths[i] <- temp[i+1]
    -> paths

procedure convertPathsToRoutes()
    i traversal [0...this.solPaths.Length-1]
        if (this.solPaths[i+1].isUpOf(this.solPaths[i])) then
            this.solRoutes <-
    this.insertLastRoutes(this.solRoutes, 'U')
        else if
    (this.solPaths[i+1].isDownOf(this.solPaths[i])) then
            this.solRoutes <-
    this.insertLastRoutes(this.solRoutes, 'D')
        else if
    (this.solPaths[i+1].isLeftOf(this.solPaths[i])) then
            this.solRoutes <-
    this.insertLastRoutes(this.solRoutes, 'L')

```

```

        else
if(this.solPaths[i+1].isRightOf(this.solPaths[i])) then
    this.solRoutes <-
this.insertLastRoutes(this.solRoutes, 'R')

procedure copySolutionPathsDFS(input Stack<Point> paths)
    this.solPaths <- new Point[paths.Count]
    i traversal [0...solPaths.Length-1]
        Point top <- paths.Pop()
        this.solPaths[i] <- new Point(top)
    i traversal [0...solPaths.Length]
        paths.Push(solPaths[i])

procedure copySolutionPathsBFS(input Stack<Point> paths)
    this.solPaths <- new Point[paths.Count]
    i traversal [0...solPaths.Length-1]
        Point top <- paths.Pop()
        this.solPaths[i] <- new Point(top)

```

Class BFS

Atribut:

TSP : bool

Method:

```

{ Constructor }
BFS()
    this.TSP <- false

{ Setter Getter }
function getTSP() -> bool
    -> this.TSP

procedure setTSP(input bool tsp)
    this.TSP <- tsp

{ Other Methods }
function isAllTreasureTaken(Stack<Point> path, Point[] tLoc) -> bool
    i traversal [0...tLoc.Length]
        if (not path.Contains(tLoc[i])) then
            -> false
        -> true

function solve(Nodes n, Map m) -> Stack of Point
    q <- new Queue<Stack<Point>>()
    nQueue <- new Queue<Nodes>()
    s <- new Stack<Point>()

```

```

s.Push(m.getStartLoc())
q.Enqueue(s)
n.setNode(m.getStartLoc())
nQueue.Enqueue(n)
while (q.Count > 0 and not isAllTreasureTaken(q.Peek(), m.getTreasureLocations())) do
    n <- nQueue.Dequeue()
    temp <- q.Dequeue()
    m.setCurLoc(temp.Peek())
    cl <- m.getCurLoc()
    m.increaseVCATCoordinate(cl)
    noOtherPath <- true
    i traversal [0...3]
        if (noOtherPath) then
            if (i = 2 and noOtherPath) then
                -> new Stack<Point>()
            if (cl.getRow() != 0 and
m.getValueAtCoordinate(cl.getRow()-1, cl.getCol()) != 'X') then
                { check Up }
                newLoc <- new Point(cl)
                newLoc.goUp()
                if (not temp.Contains(newLoc) or (i = 1
and noOtherPath)) then
                    temp2 <- new Stack<Point>(temp)
                    temp1 <- new Stack<Point>(temp2)
                    temp1.Push(newLoc)
                    n.setUpChild(newLoc)
                    nQueue.Enqueue(n.setUpChild())
                    q.Enqueue(temp1)
                    noOtherPath <- false

                if (cl.getCol() != 0 and
m.getValueAtCoordinate(cl.getRow(), cl.getCol()-1) != 'X')
then
                    { check Left }
                    newLoc <- new Point(cl)
                    newLoc.goLeft()
                    if (not temp.Contains(newLoc) or (i = 1
and noOtherPath)) then
                        temp2 <- new Stack<Point>(temp)
                        temp1 <- new Stack<Point>(temp2)
                        temp1.Push(newLoc)
                        n.setLeftChild(newLoc)
                        nQueue.Enqueue(n.getLeftChild())
                        q.Enqueue(temp1)
                        noOtherPath <- false

                if (cl.getCol() != m.getCol()-1 and
m.getValueAtCoordinate(cl.getRow(), cl.getCol()+1) != 'X')
then
                    { check Right }

```

```

        newLoc <- new Point(cl)
        newLoc.goRight()
        if (not temp.Contains(newLoc) or (i = 1
and noOtherPath)) then
            temp2 <- new Stack<Point>(temp)
            temp1 <- new Stack<Point>(temp2)
            temp1.Push(newLoc)
            n.setRightChild(newLoc)
            nQueue.Enqueue(n.getRightChild())
            q.Enqueue(temp1)
            noOtherPath <- false

            if (cl.getRow() != m.getRow()-1 and
m.getValueAtCoordinate(cl.getRow()+1, cl.getCol()) != 'X')
then
                { check Down }
                newLoc <- new Point(cl)
                newLoc.goDown()
                if (not temp.Contains(newLoc) or (i = 1
and noOtherPath)) then
                    temp2 <- new Stack<Point>(temp)
                    temp1 <- new Stack<Point>(temp2)
                    temp1.Push(newLoc)
                    n.setDownChild(newLoc)
                    nQueue.Enqueue(n.getDownChild())
                    q.Enqueue(temp1)
                    noOtherPath <- false

if (TSP)
    n <- nQueue.Dequeue()
    while(nQueue.Count > 0) do
        nQueue.Dequeue()
        nQueue.Enqueue(n)
        sol <- q.Dequeue()
        while (q.Count > 0) do
            q.Dequeue()
            q.Enqueue(sol)
        while(not q.Peek().Peek().Equals(m.getStartLoc())) do
            n <- nQueue.Dequeue()
            temp <- q.Dequeue()
            m.setCurLoc(temp.Peek())
            cl <- m.getCurLoc()
            if (cl.getRow() != 0 and
m.getValueAtCoordinate(cl.getRow()-1, cl.getCol()) != 'X') then
                { check Up }
                newLoc <- new Point(cl)
                newLoc.goUp()
                temp2 <- new Stack<Point>(temp)
                temp1 <- new Stack<Point>(temp2)
                n.setUpChild(newLoc)
                nQueue.Enqueue(n)
                temp1.Push(newLoc)

```

```

        q.Enqueue(temp1)

        if (cl.getCol() != 0 and
m.getValueAtCoordinate(cl.getRow(), cl.getCol()-1) != 'X') then
            { check Left }
            newLoc <- new Point(cl)
            newLoc.goLeft()
            temp2 <- new Stack<Point>(temp)
            temp1 <- new Stack<Point>(temp2)
            n.setLeftChild(newLoc)
            nQueue.Enqueue(n)
            temp1.Push(newLoc)
            q.Enqueue(temp1)

        if (cl.getCol() != m.getCol() - 1 and
m.getValueAtCoordinate(cl.getRow(), cl.getCol()+1) != 'X') then
            { check Right }
            newLoc <- new Point(cl)
            newLoc.goRight()
            temp2 <- new Stack<Point>(temp)
            temp1 <- new Stack<Point>(temp2)
            n.setRightChild(newLoc)
            nQueue.Enqueue(n)
            temp1.Push(newLoc)
            q.Enqueue(temp1)

        if (cl.getRow() != m.getRow() - 1 and
m.getValueAtCoordinate(cl.getRow()+1, cl.getCol()) != 'X') then
            { check Down }
            newLoc <- new Point(cl)
            newLoc.goDown()
            temp2 <- new Stack<Point>(temp)
            temp1 <- new Stack<Point>(temp2)
            n.setDownChild(newLoc)
            nQueue.Enqueue(n)
            temp1.Push(newLoc)
            q.Enqueue(temp1)
        if (q.Count > 0) then
            -> q.Dequeue()
        -> new Stack<Point>()

procedure getSolution(input Map m)
    if (m.isValid()) then
        startTime()
        Stack<Point> solution <- solve(routeNodes, m)
        stopTime()

        this.nodes <- routeNodes.getNodesAmount()
        this.steps <- solution.Count
        copySolutionPathsBFS(solution)
        convertPathsToRoutes()

```

```

        else
            output("BFS method cannot be done since the map is
invalid.")

Class DFS

Atribut:

TSP : bool

Method:

{ Constructor }
DFS()
    this.TSP <- false

{ Setter Getter }
function getTSP() -> bool
    -> this.TSP

procedure setTSP(input bool tsp)
    this.TSP <- tsp

{ Other Methods }
function isPathAlreadyTaken(Stack<Point> paths, Point p) ->bool
    -> paths.Contains(p)

function sortPriorityCoordinate(Point[] p, Map m) -> array of
Point
    { using insertion sort }
    if (p.Length > 1) then
        i traversal[1...p.Length]
        temp <- new Point(p[i])
        j <- i - 1
        while (m.getVCAtCoordinate(temp) <
m.getVCAtCoordinate(p[j]) and j > 0) do
            p[j + 1].copyPoint(p[j])
            j <- j - 1

            if (m.getVCAtCoordinate(temp) >=
m.getVCAtCoordinate(p[j])) then
                p[j + 1].copyPoint(temp)
            else
                p[j + 1].copyPoint(p[j])
                p[j].copyPoint(temp)
    -> p

function getPriorityCoordinates(Map m) -> array of Point
    coordinates <- new Point[] {}

```

```

cl <- m.getCurLoc()

if (cl.getRow() != 0 and
m.getValueAtCoordinate(cl.getRow() - 1, cl.getCol()) != 'X')
then
    { check Up }
    newCl <- new Point(cl)
    newCl.goUp()
    coordinates <- insertLastPaths(coordinates, newCl)

    if (cl.getCol() != 0 and
m.getValueAtCoordinate(cl.getRow(), cl.getCol() - 1) != 'X')
then

        { check Left }
        newCl <- new Point(cl)
        newCl.goLeft()
        coordinates <- insertLastPaths(coordinates, newCl)

        if (cl.getCol() != m.getCol() - 1 and
m.getValueAtCoordinate(cl.getRow(), cl.getCol()+1) != 'X')
then
            { check Right }
            newCl <- new Point(cl)
            newCl.goRight()
            coordinates <- insertLastPaths(coordinates, newCl)

        if (cl.getRow() != m.getRow() - 1 and
m.getValueAtCoordinate(cl.getRow()+1, cl.getCol()) != 'X') then
            { check Down }
            newCl <- new Point(cl)
            newCl.goDown()
            coordinates <- insertLastPaths(coordinates, newCl)

coordinates <- sortPriorityCoordinate(coordinates, m)
-> coordinates

function setNewNodes(Point cl, Point newCl, Nodes n) -> Nodes
if (newCl.isUpOf(cl)) then
    n.setUpChild(newCl)
    -> n.getUpChild()
else if (newCl.isLeftOf(cl)) then
    n.setLeftChild(newCl)
    -> n.getLeftChild()
else if (newCl.isRightOf(cl)) then
    n.setRightChild(newCl)
    -> n.getRightChild()
else if (newCl.isDownOf(cl)) then
    n.setDownChild(newCl)
    -> n.getDownChild()
-> n

```

```

function solve(Nodes n, Map m, Stack<Point> p) -> Stack of Point
    cl <- m.getCurLoc()
    p.Push(cl)
    m.increaseVCAtCoordinate(cl)

    if (this.TSP) then
        objectives <- isAllTreasureTaken(p,
m.getTreasureLocations()) and p.Peek().Equals(m.getStartLoc())
    else
        objectives <- isAllTreasureTaken(p,
m.getTreasureLocations())

    if (objectives) then
        -> p
    iteration <- 0
    availableCoordinates <- getPriorityCoordinates(m)
    while (iteration < availableCoordinates.Length) do
        m.setCurLoc(availableCoordinates[iteration])
        newNodes <- setNewNodes(cl,
availableCoordinates[iteration], n)
        if (isAllTreasureTaken(solve(newNodes, m, p),
m.getTreasureLocations())) then
            -> p
        iteration++

    { Backtrack }
    Point dump <- p.Pop()
    -> new Stack<Point>()

function solveOneWay(Nodes n, Map m, Stack<Point> p) -> Stack of Point
    cl <- m.getCurLoc()
    n.setNode(cl)
    p.Push(n.getNode())
    if (isAllTreasureTaken(p, m.getTreasureLocations())) then
        -> p

    if (cl.getRow() != 0 and
m.getValueAtCoordinate(cl.getRow() - 1, cl.getCol()) != 'X')
then
    { check Up }
    newCl <- new Point(cl)
    newCl.goUp()
    if (not isPathAlreadyTaken(p, newCl)) then
        m.setCurLoc(newCl)
        n.setUpChild(newCl)
        if (isAllTreasureTaken(solveOneWay(n.setUpChild(),
m, p), m.getTreasureLocations())) then
            -> p

```

```

        if (cl.getCol() != 0 and
m.getValueAtCoordinate(cl.getRow(), cl.getCol() - 1) !<= 'X')
then

        { check Left }
        newCl <- new Point(cl)
        newCl.goLeft()
        if (not isPathAlreadyTaken(p, newCl)) then
            m.setCurLoc(newCl)
            n.setLeftChild(newCl)

if(isAllTreasureTaken(solveOneWay(n.getLeftChild(), m, p),
m.getTreasureLocations())) then
    -> p

        if (cl.getCol() != m.getCol() - 1 and
m.getValueAtCoordinate(cl.getRow(), cl.getCol()+1) != 'X') then
            { check Right }
            newCl <- new Point(cl)
            newCl.goRight()
            if (not isPathAlreadyTaken(p, newCl))
                m.setCurLoc(newCl)
                n.setRightChild(newCl)
                if (isAllTreasureTaken(solveOneWay
(n.getRightChild(), m, p), m.getTreasureLocations())))
                    then
                        -> p

        if (cl.getRow() != m.getRow() - 1 and
m.getValueAtCoordinate(cl.getRow()+1, cl.getCol()) != 'X') then
            { check Down }
            newCl <- new Point(cl)
            newCl.goDown()
            if (not isPathAlreadyTaken(p, newCl)) then
                m.setCurLoc(newCl)
                n.setDownChild(newCl)
                if (isAllTreasureTaken(solveOneWay
(n.getDownChild(), m, p), m.getTreasureLocations())))
                    then
                        -> p

        { Backtrack }
        dump <- p.Pop()
        -> new Stack<Point>()

function isAllTreasureTaken(Stack<Point> p, Point[] tLoc) ->
bool
    i traversal [0...tLoc.Length]
        if (not p.Contains(tLoc[i])) then
            -> false
        -> true

procedure getSolution(input Map m)

```

```

if (m.getValid()) then
    p <- new Stack<Point>()
    solution <- new Stack<Point>()
    if (not this.TSP) then
        startTime()
        solution <- solveOneWay(routeNodes, m, p)
        stopTime()

    if (this.TSP or solution.Count = 0)
        m.resetMap()
        routeNodes <- new Nodes()
        startTime()
        solution <- solve(routeNodes, m, p)
        stopTime()

    this.steps <- solution.Count
    this.nodes <- routeNodes.getNodesAmount()
    copySolutionPathsDFS(solution)
    convertPathsToRoutes()
else
    output("DFS method cannot be done since the map is
invalid.")

```

4.2. Struktur Data

```
.vs
Properties
assets
bin
doc
src
    class
        Map.cs
        Point.cs
        Route.cs
        Tile.cs
    method
        BFS.cs
        DFS.cs
        Solver.cs
    obj
        MainWindow.xaml.cs
        App.config
        App.xaml
        MainWindow.xaml
        TreasureMaze.csproj
    test
.gitignore
README.md
```

Gambar 4.1 Struktur data program

Tabel 4.1 Daftar kelas dan atribut kelas

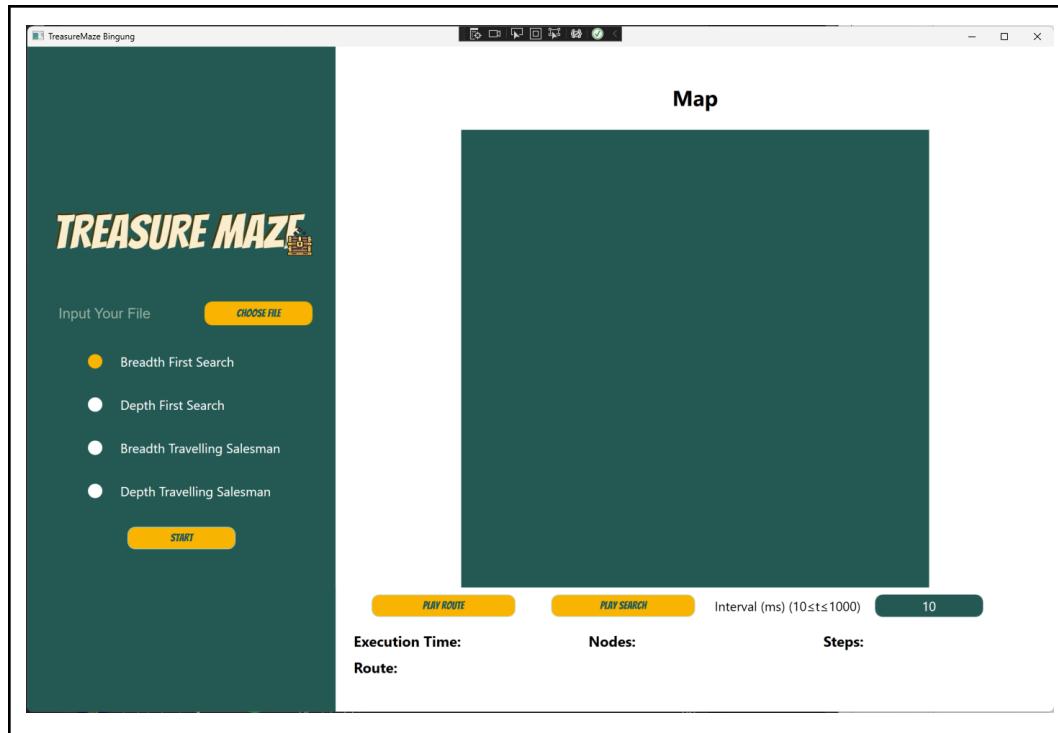
Namespace	Class	Attribute	Kegunaan
Class	Map	int row	menyimpan jumlah baris dari Map
		int col	menyimpan jumlah kolom dari Map
		int nTreasure	menyimpan jumlah <i>treasure</i>

			yang ada pada Map
		Point[] treasureLocs	menyimpan lokasi dari seluruh <i>treasure</i> pada Map
		Point startLoc	menyimpan lokasi simpul mula-mula
		Point curLoc	menyimpan lokasi simpul yang sedang ditempati
		Tile[,] buffer	menyimpan matrix dari setiap petak pada Map
		bool valid	menyimpan nilai valid atau tidaknya sebuah Map
	Nodes	Nodes parent	menyimpan simpul orang tua
		Point node	menyimpan nilai dari simpul tertentu
		Nodes[] children	menyimpan daftar dari simpul anak
	Point	int row	menyimpan lokasi baris dari <i>class Point</i>
		int col	menyimpan lokasi kolom dari <i>class Point</i>
	Tile	char value	menyimpan nilai simbol dari sebuah petak
		int visitedCount	menyimpan jumlah frekuensi petak tersebut dikunjungi
Method	Solver	int nodes	menyimpan jumlah simpul yang dibentuk selama pencarian solusi
		int steps	menyimpan jumlah langkah yang diambil dalam menempuh rute solusi
		char[] solRoutes	menyimpan runtutan arah langkah dari rute solusi
		Point[] solPaths	menyimpan runtutan petak yang dilewati pada rute solusi
		Nodes routeNodes	menyimpan simpul-simpul

			yang dibentuk selama pencarian solusi dilakukan
		Stopwatch watch	menyimpan nilai waktu mulai dan waktu berhenti pada saat pencarian solusi
		Point[] listCurLoc	menyimpan urutan semua <i>current location</i> yang pernah dikunjungi selama pencarian solusi berlangsung
	BFS	bool TSP	menyimpan tanda perlu atau tidaknya pemecahan permasalahan TSP dilakukan
	DFS	bool TSP	
TreasureMaze	MainWindow	String mode	menyimpan mode metode penyelesaian permasalahan
		Map map	menyimpan <i>class Map</i> yang akan ditampilkan
		Point[] solPaths	<i>reference</i> ke Solver.solPaths
		double side	menyimpan nilai panjang sisi <i>grid</i>
		string rootPath	menyimpan <i>path</i> untuk <i>directory</i> yang dipakai
		Point[] listCurLoc	<i>reference</i> ke Solver.listCurLoc

4.3. Tata Cara Penggunaan Program

a. Antarmuka



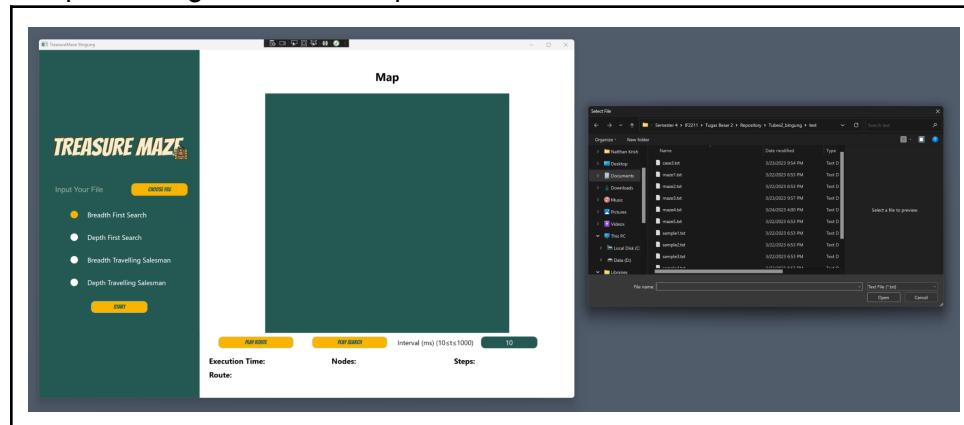
Antarmuka yang terdapat pada perangkat lunak yang dibangun adalah sebagai berikut,

No	Nama	Tipe	Deskripsi
1	Tombol Choose File	Button	Membangkitkan event pemilihan file
2	Tombol Start	Button	Memproses map apabila valid
3	Tombol Play Route	Button	Melakukan visualisasi pada Map Buffer untuk solusi algoritma tertentu
4	Tombol Play Search	Button	Melakukan visualisasi pada Map Buffer untuk proses pencarian algoritma tertentu
5	Buffer Interval	TextBox	Menerima input, validasi dan memberikan informasi interval waktu pada visualisasi pada Map Buffer
6	Map Buffer	Grid	Buffer visualisasi peta
7	Tombol Pilihan (BFS, DFS,	Button + Label	Pemilihan algoritma yang ingin digunakan

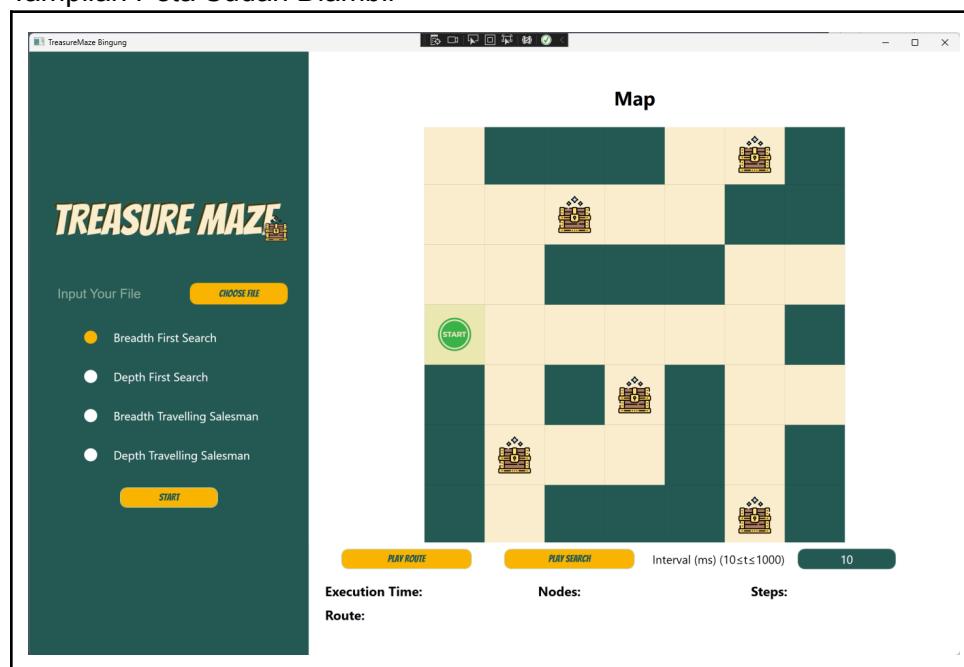
	BFSTSP, DFSTSP)		
8	Results Buffer	Label	Menampilkan hasil pencarian rute dari algoritma yang dipilih

b. Tampilan

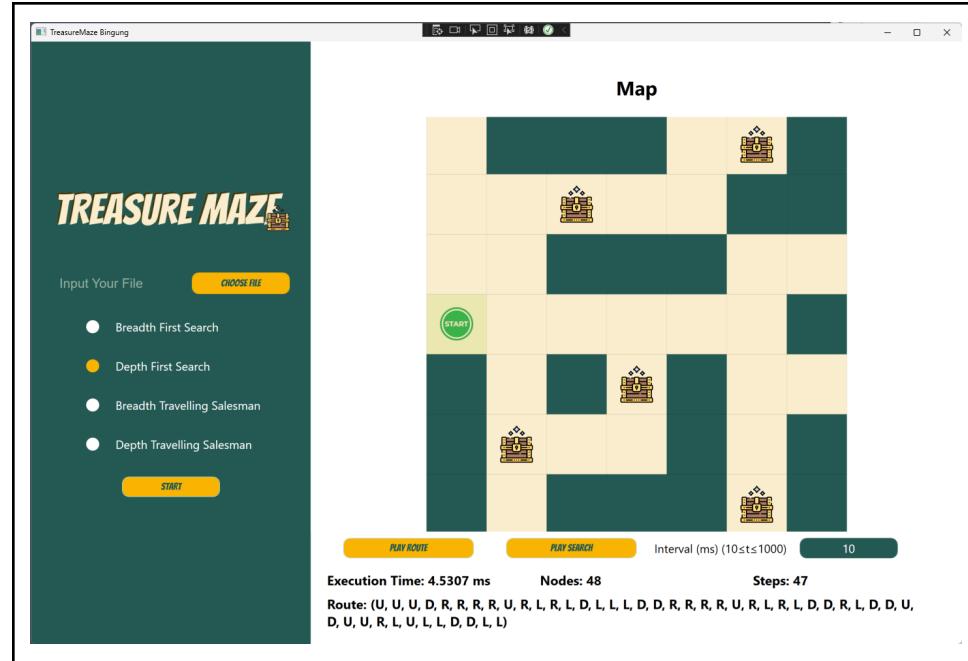
1. Tampilan Pengambilan File Input



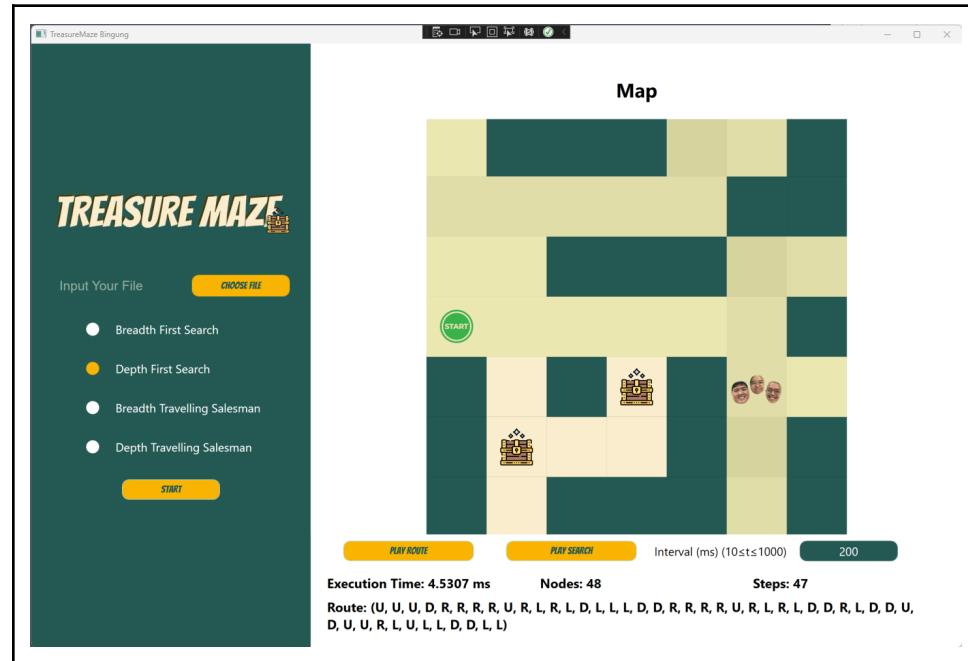
2. Tampilan Peta Sudah Diambil



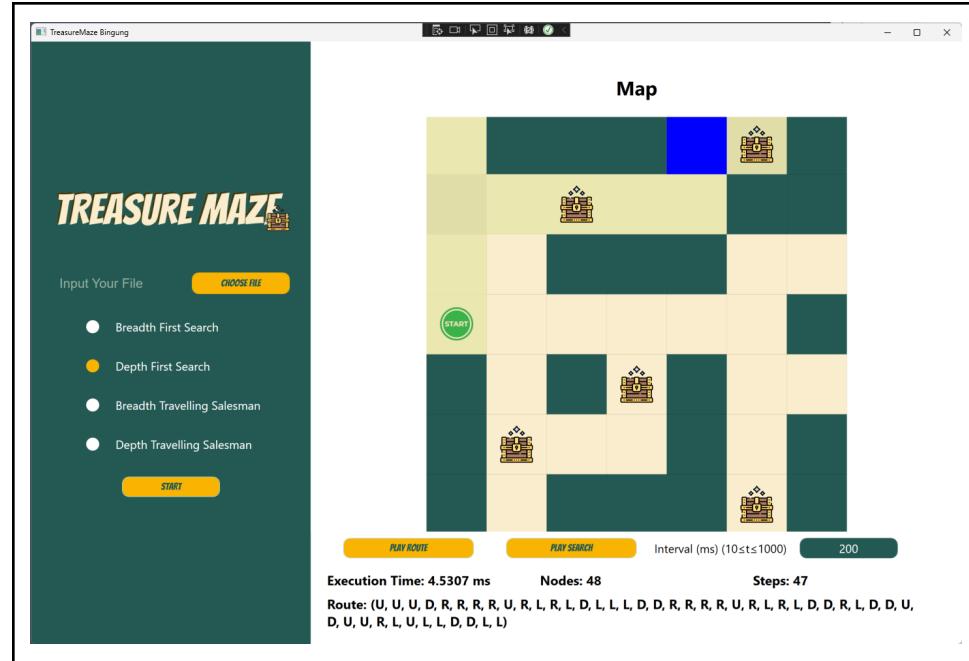
3. Tampilan Hasil



4. Tampilan Visualisasi Solusi



5. Tampilan Visualisasi Pencarian



4.4. Hasil Pengujian

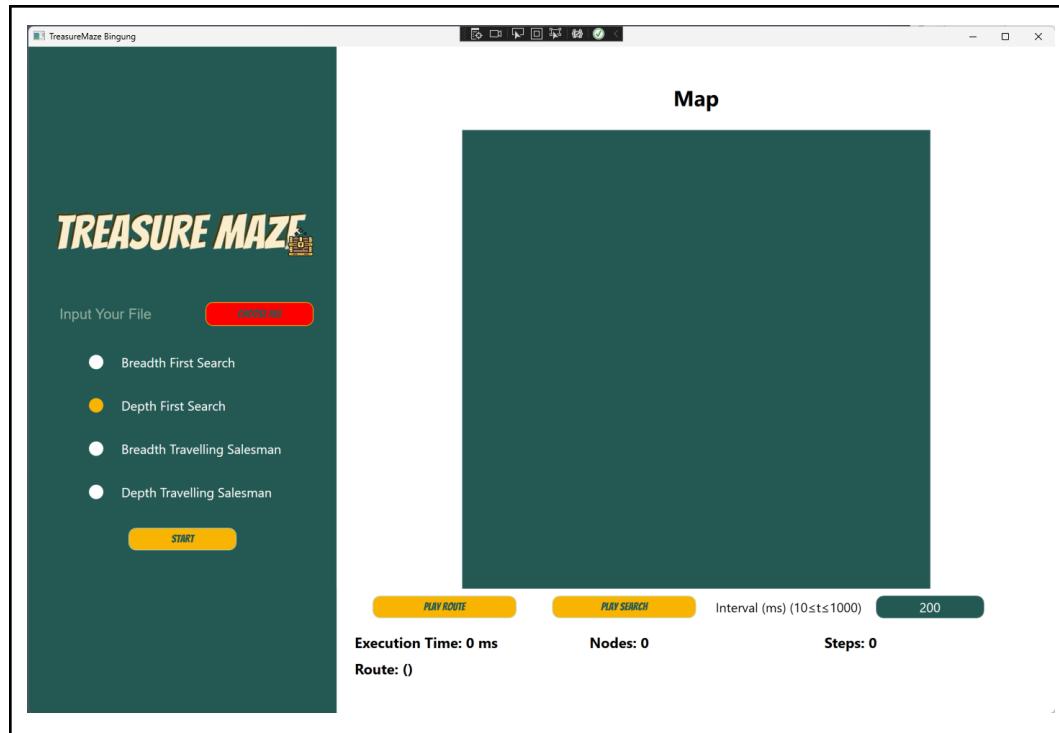
a. Pengujian 1

1. Deskripsi peta

J A N G A N
L U P A C E
K Y A N G B
E G I N I Y

Peta Tidak Valid

2. Hasil Pengujian



3. Penjelasan Hasil Pengujian

Button Choose File berubah warna menjadi merah setelah button Start ditekan. Hal ini disebabkan ketidaksesuaian dengan spesifikasi peta

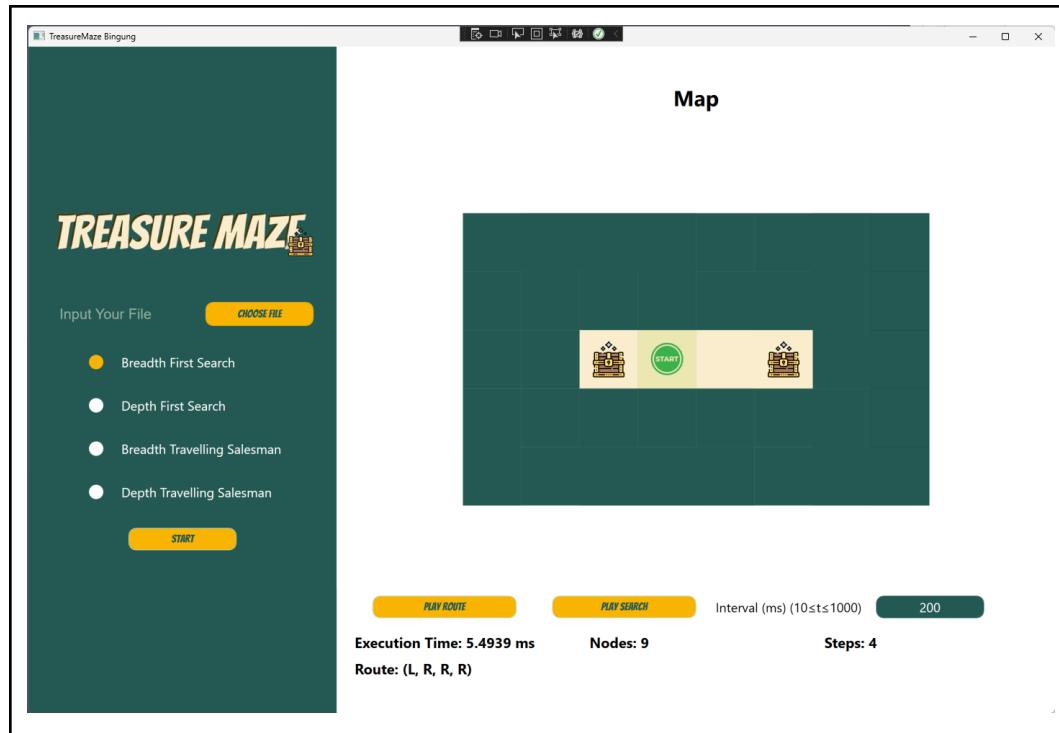
b. Pengujian 2

1. Deskripsi peta

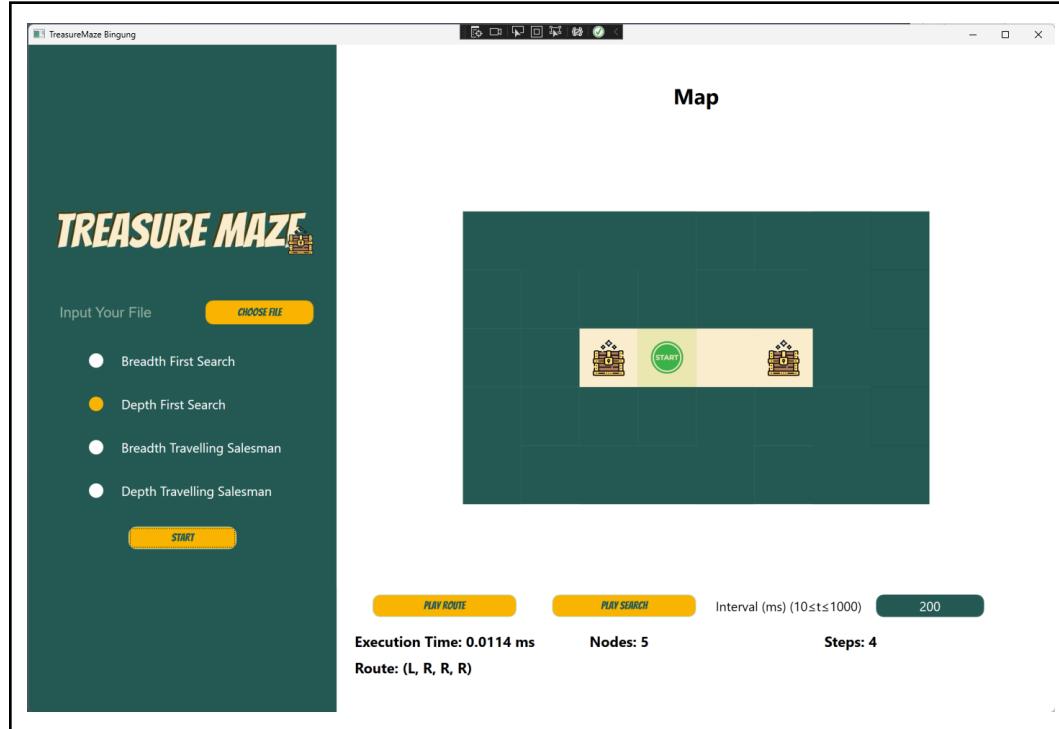
X	X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X	X
X	X	T	K	R	T	X	X	X
X	X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X	X

Peta Valid Sederhana

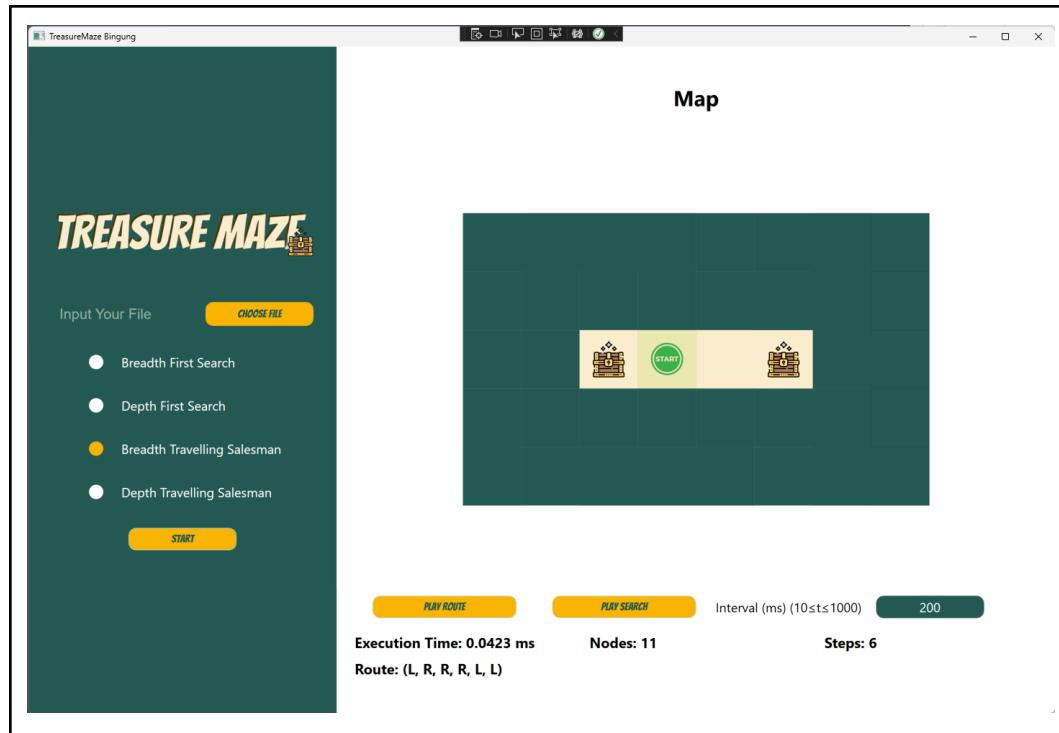
2. Hasil Pengujian BFS



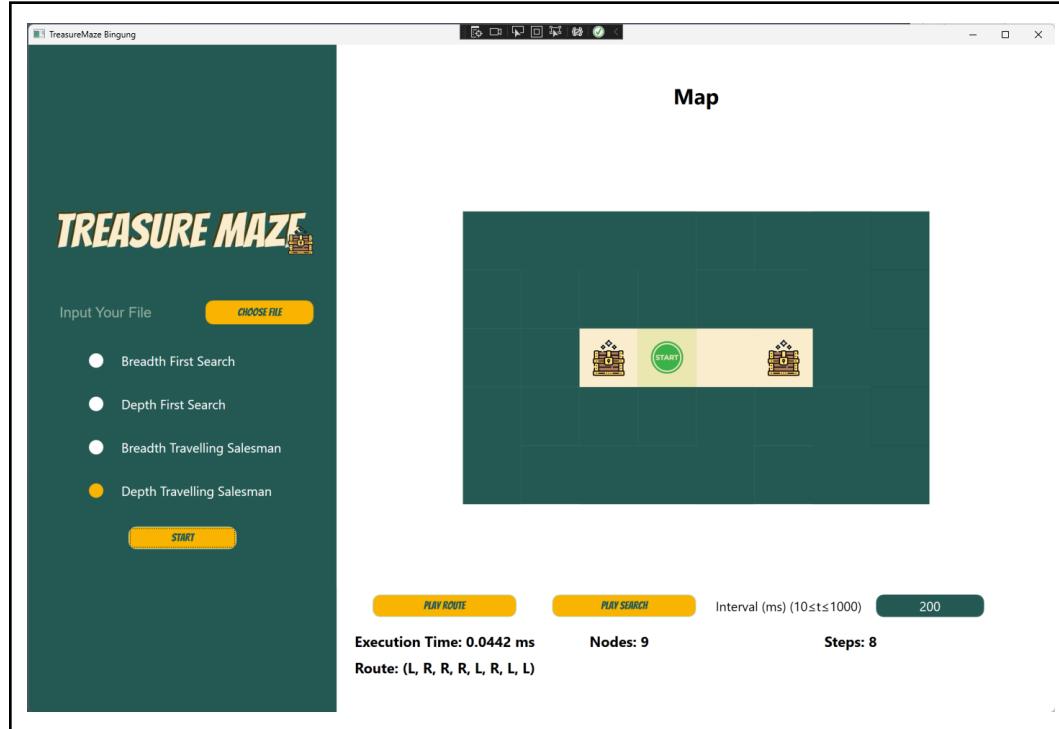
3. Hasil Pengujian DFS



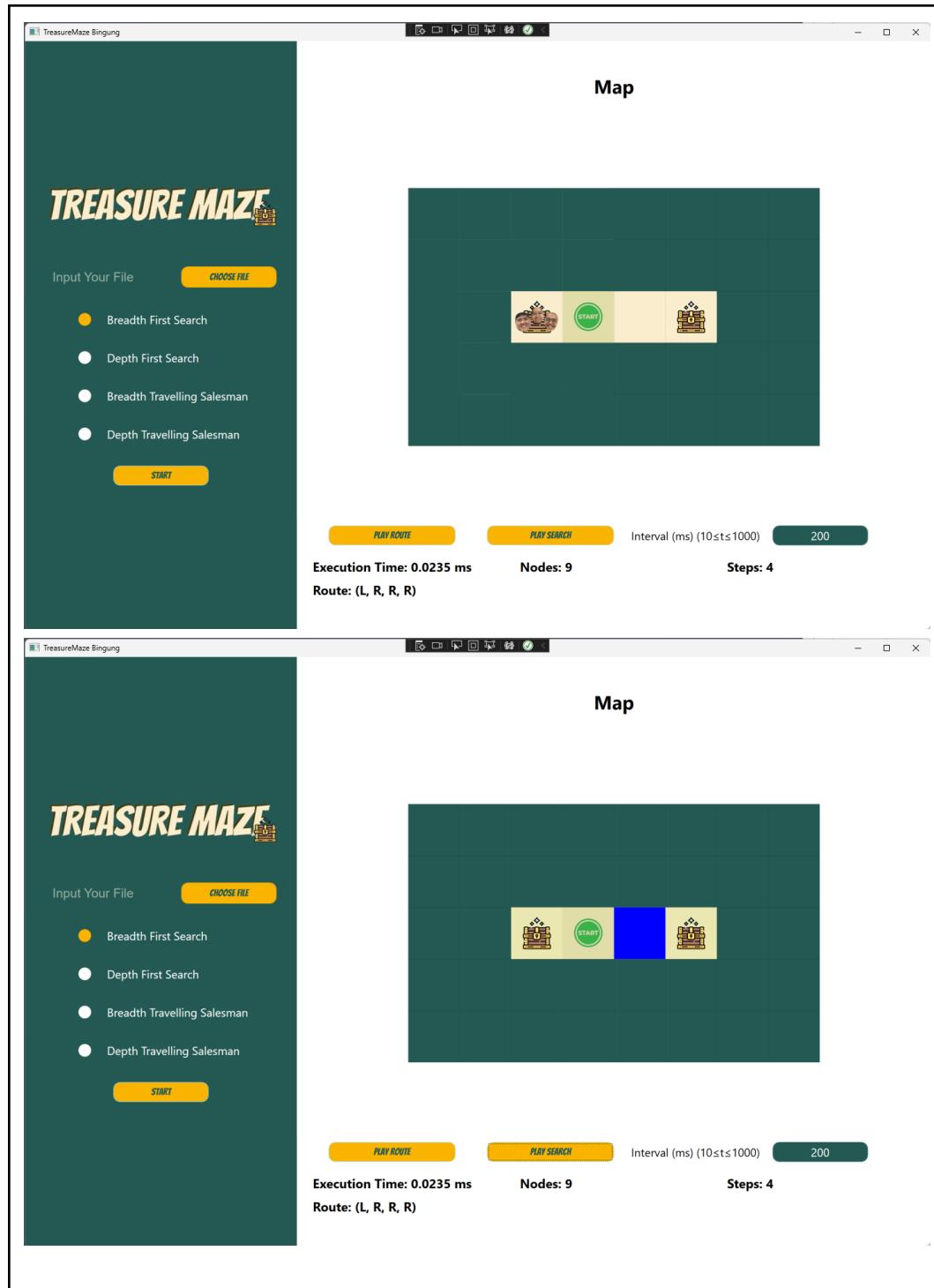
4. Hasil Pengujian BFS Travelling



5. Hasil Pengujian DFS Travelling



6. Beberapa Tampilan Visualisasi

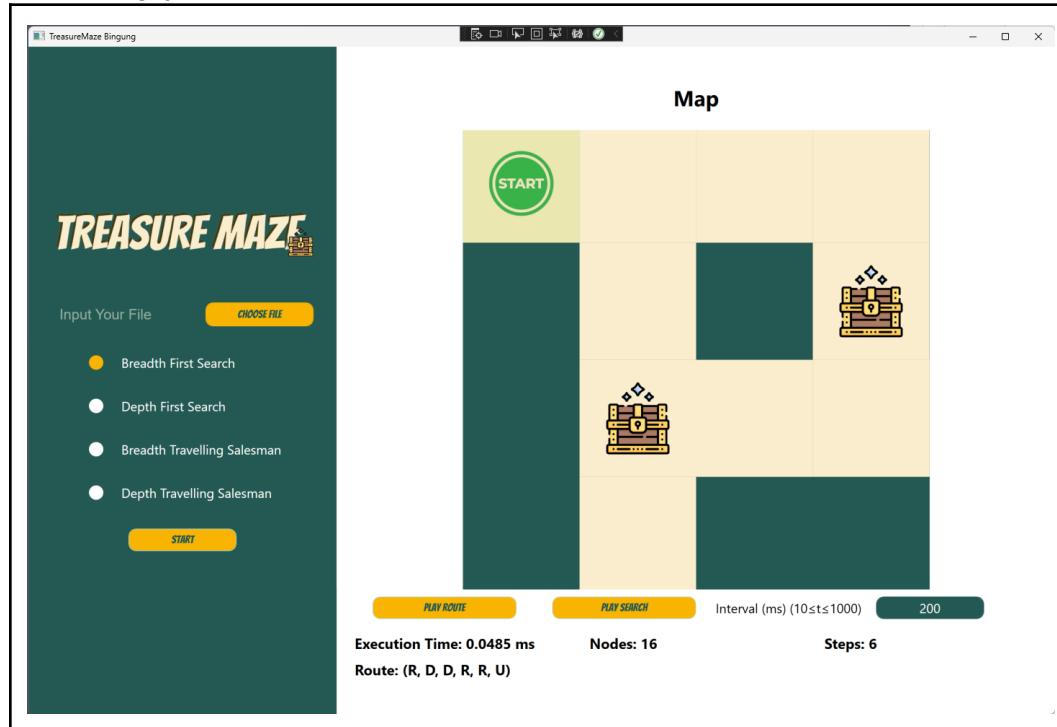


- c. Pengujian 3
1. Deskripsi peta

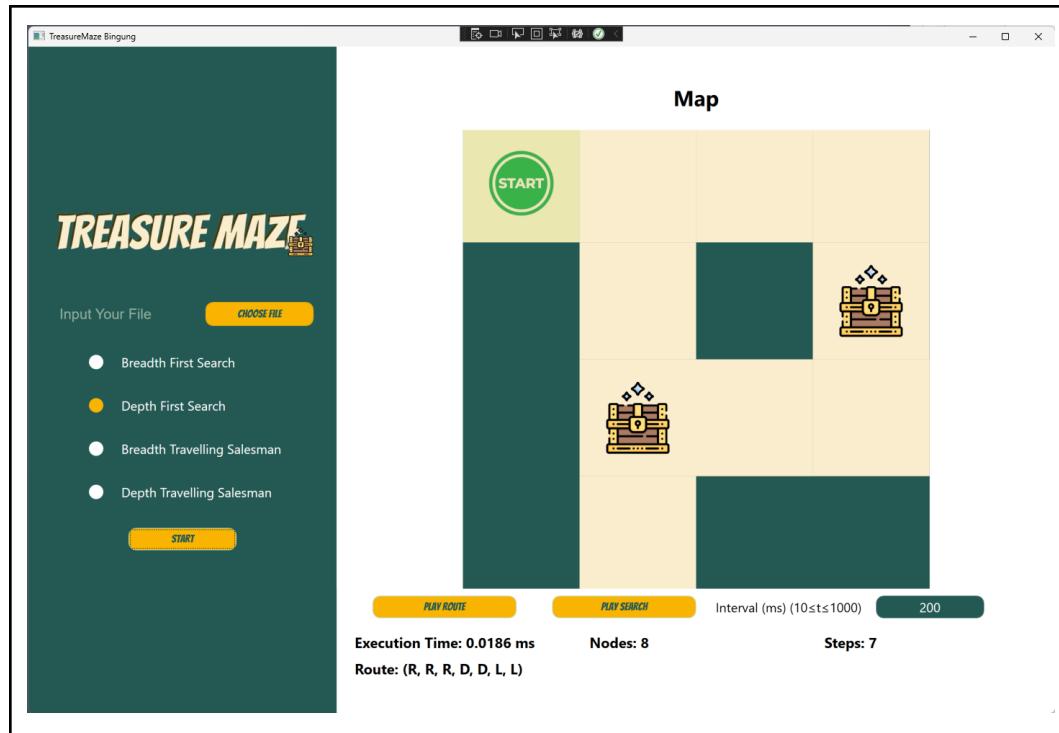
```
K R R R  
X R X T  
X T R R  
X R X X
```

Peta Valid Persegi

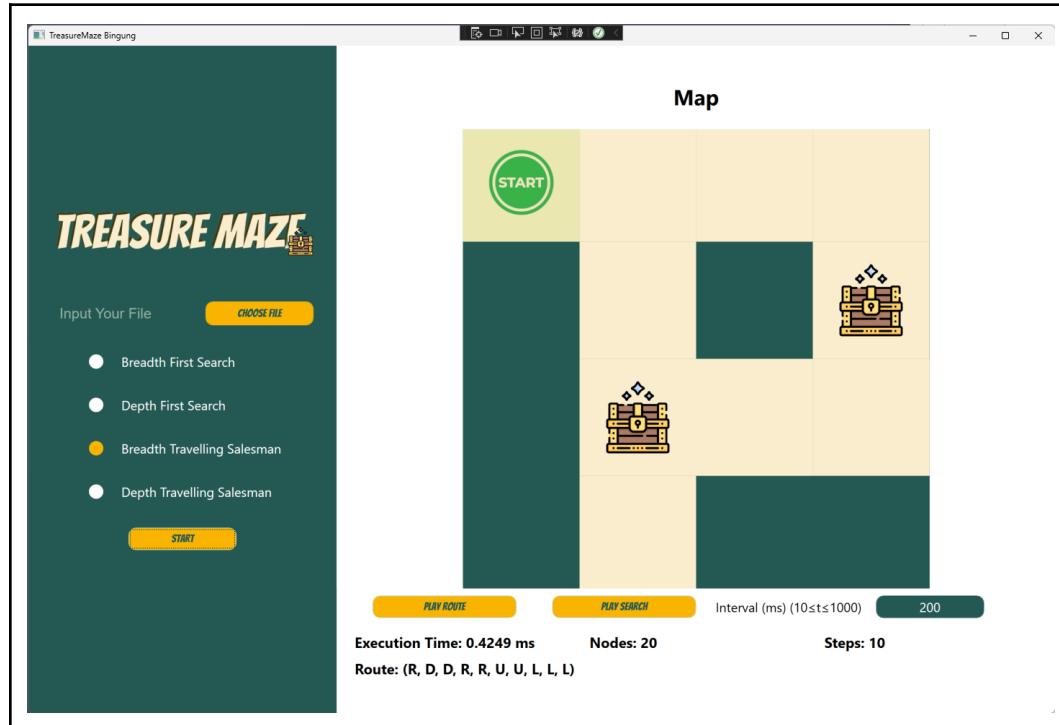
2. Hasil Pengujian BFS



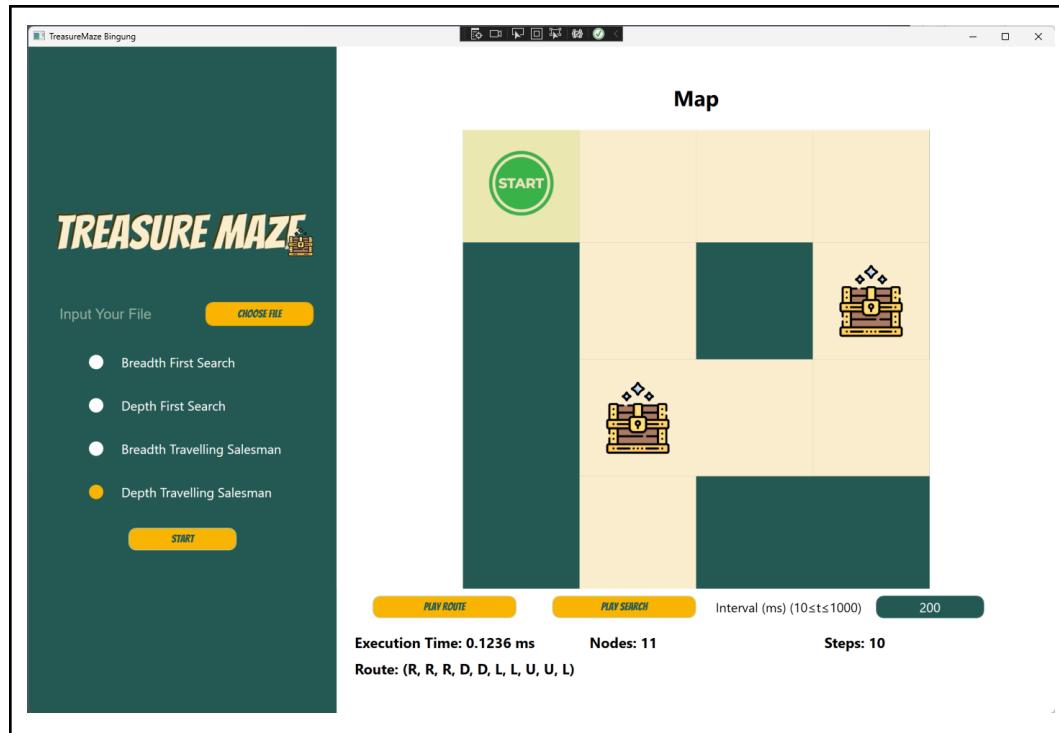
3. Hasil Pengujian DFS



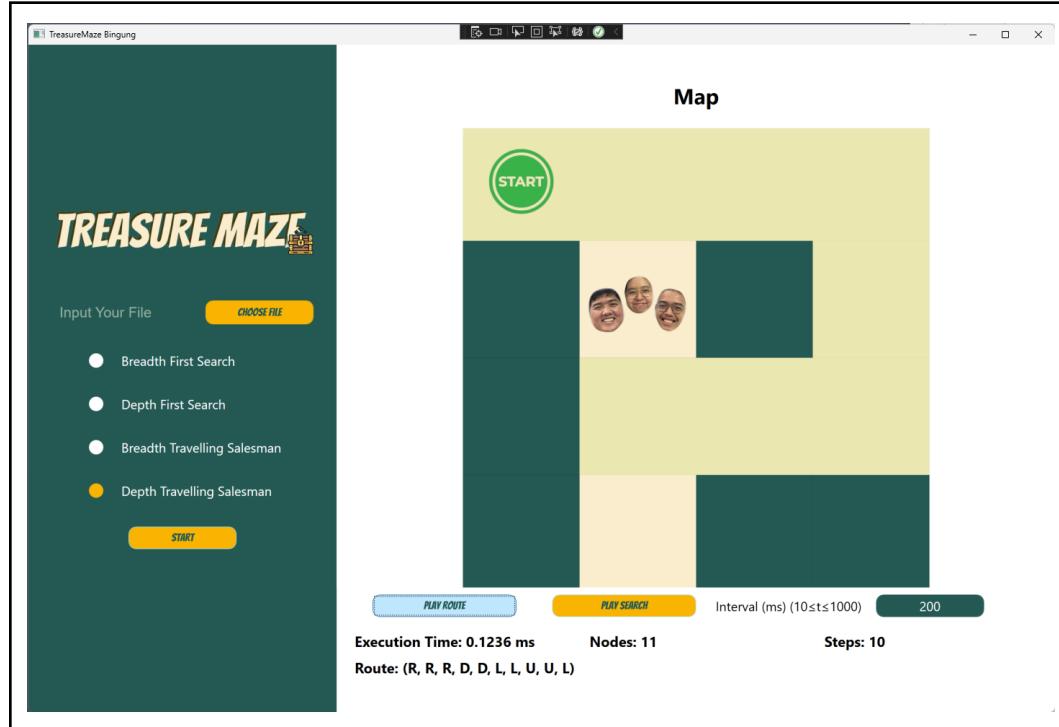
4. Hasil Pengujian BFS Travelling

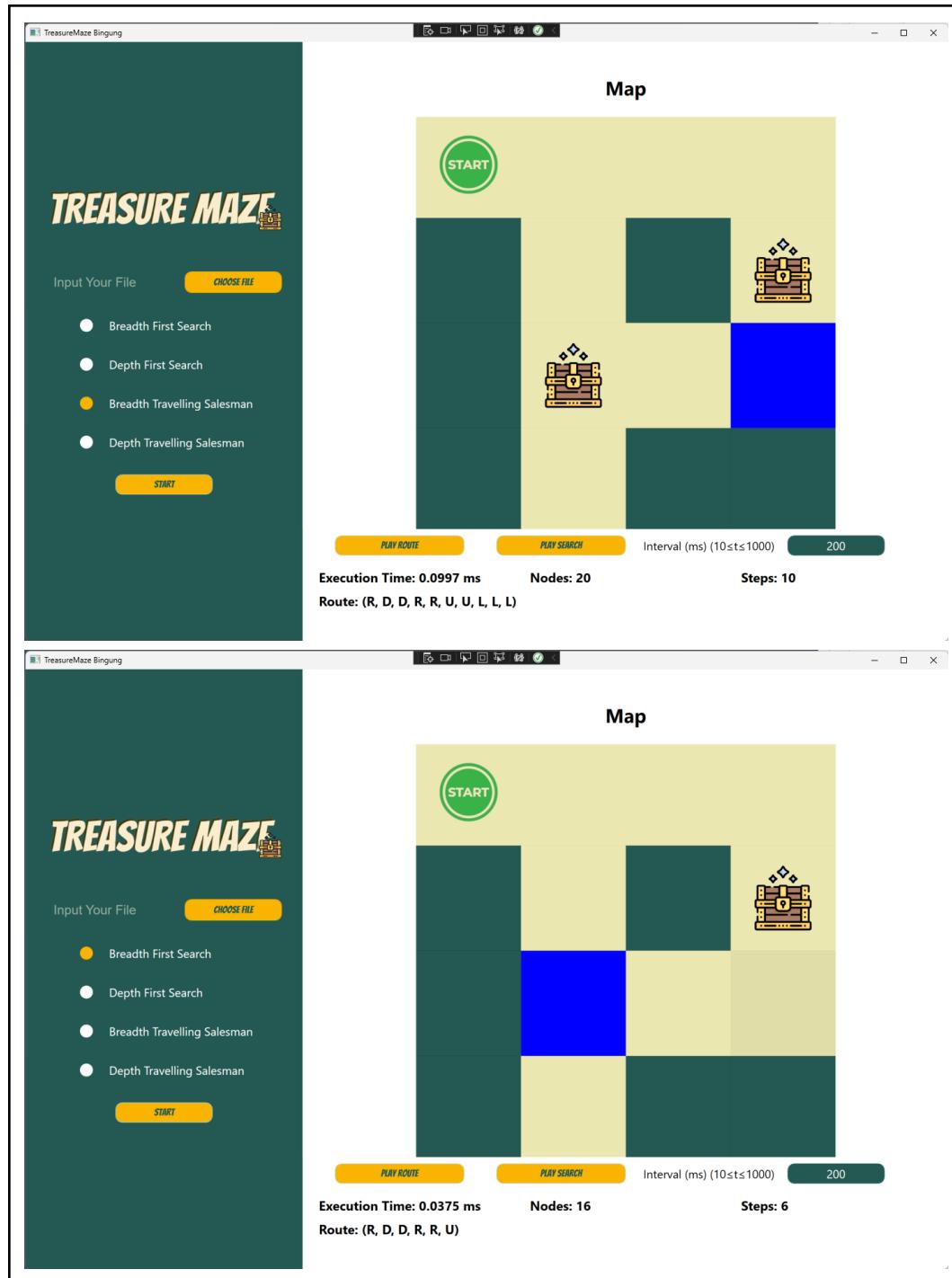


5. Hasil Pengujian DFS Travelling



6. Beberapa Tampilan Visualisasi



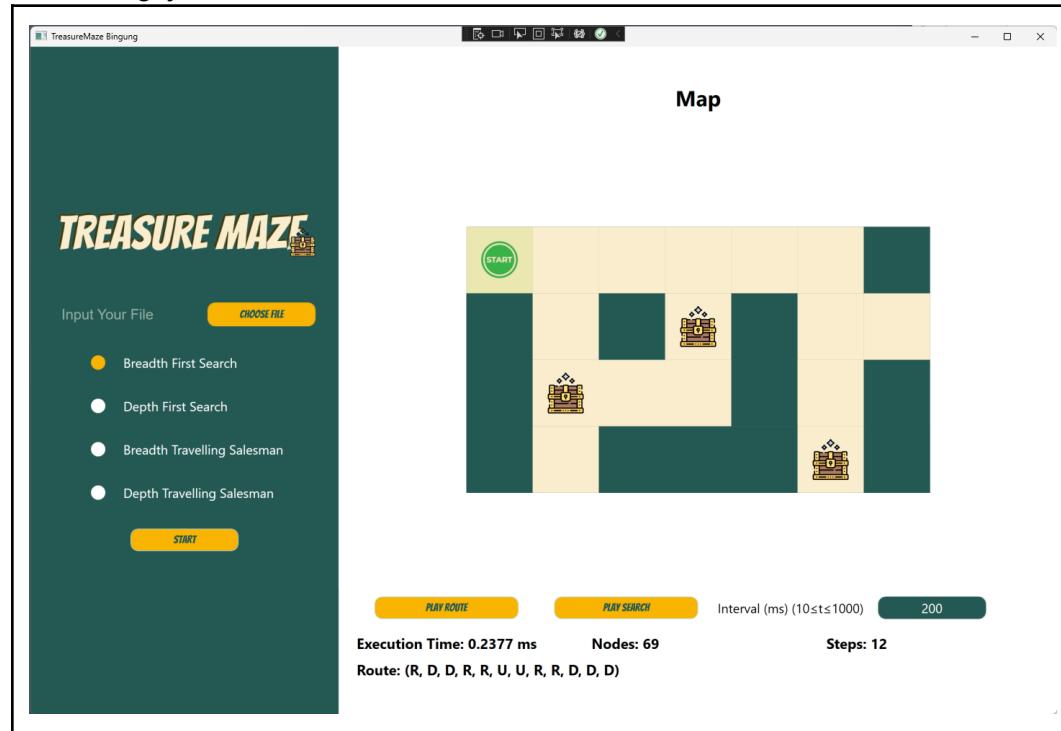


d. Pengujian 4

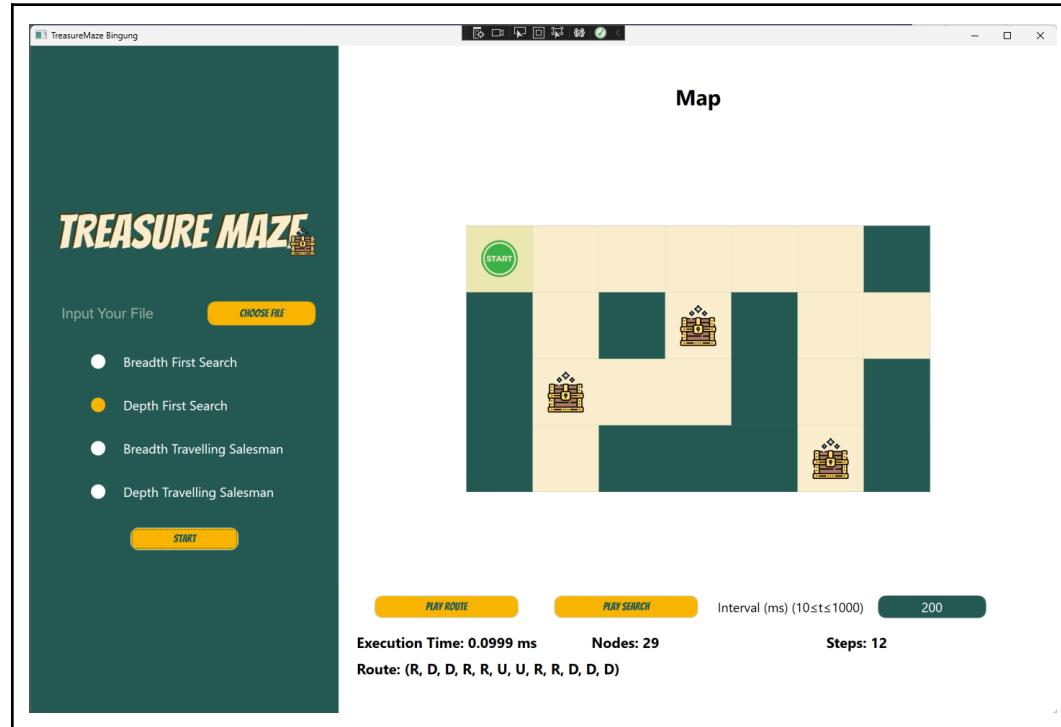
1. Deskripsi peta

Peta Valid Persegi Panjang

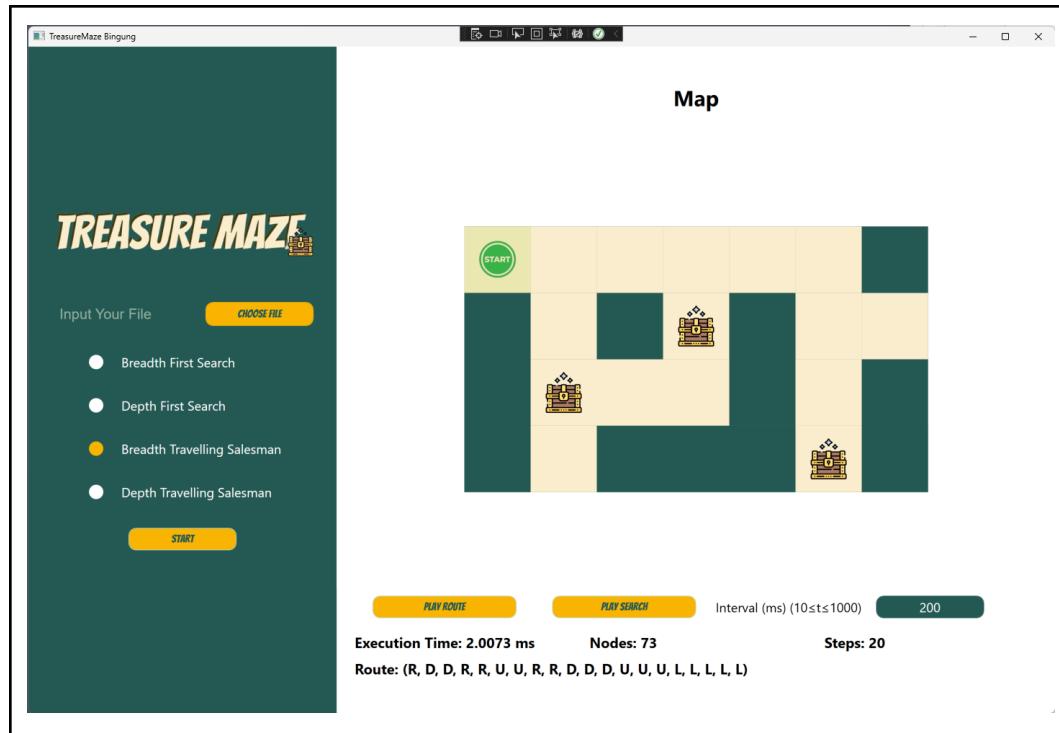
2. Hasil Pengujian BFS



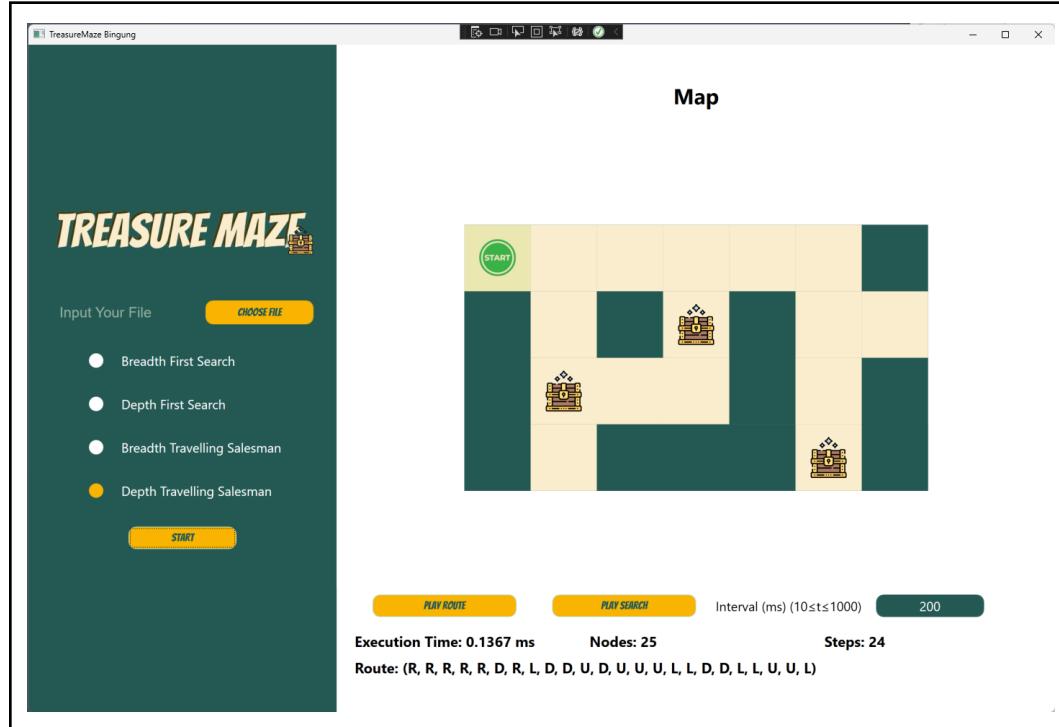
3. Hasil Pengujian DFS



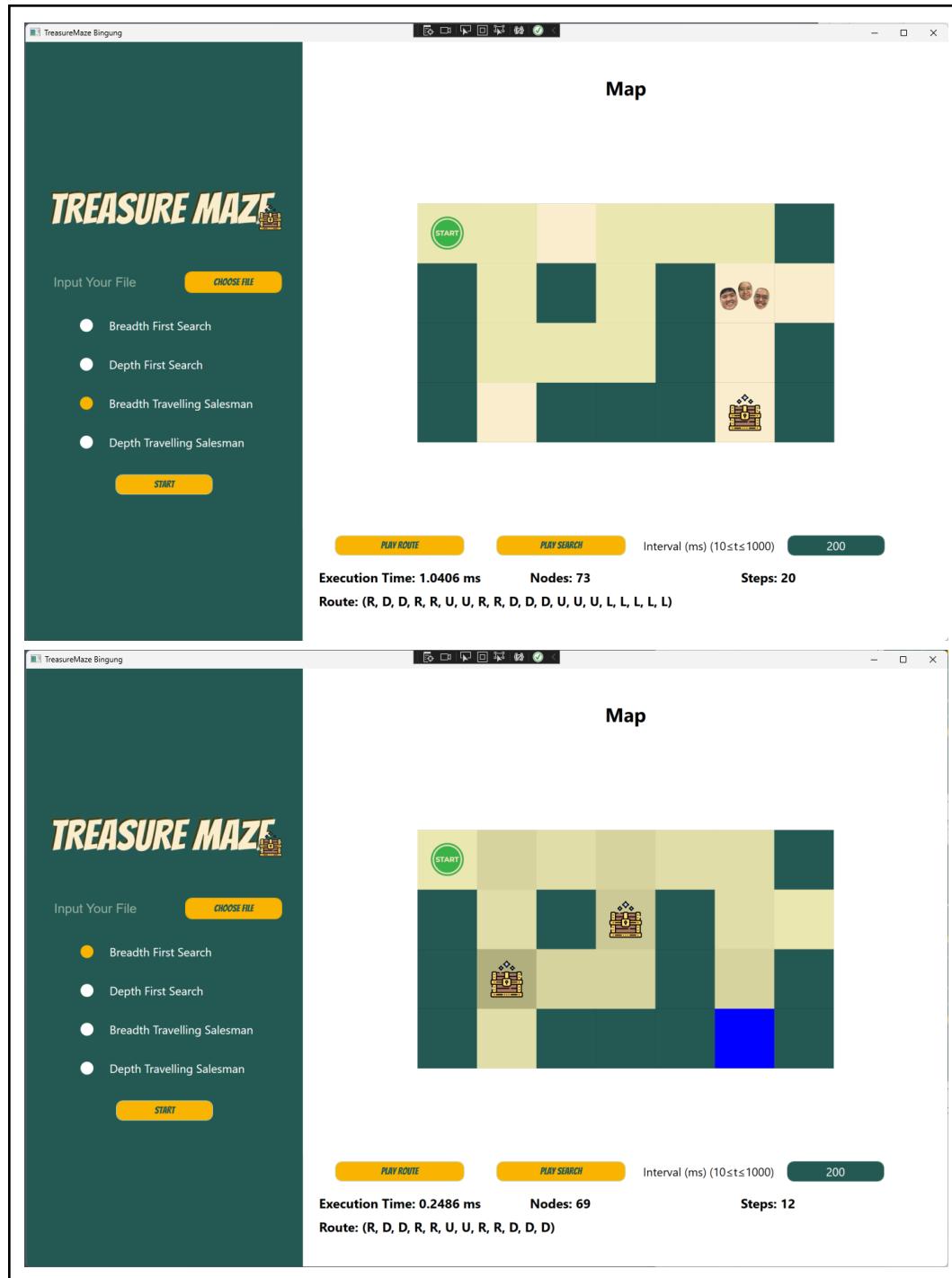
4. Hasil Pengujian BFS Travelling

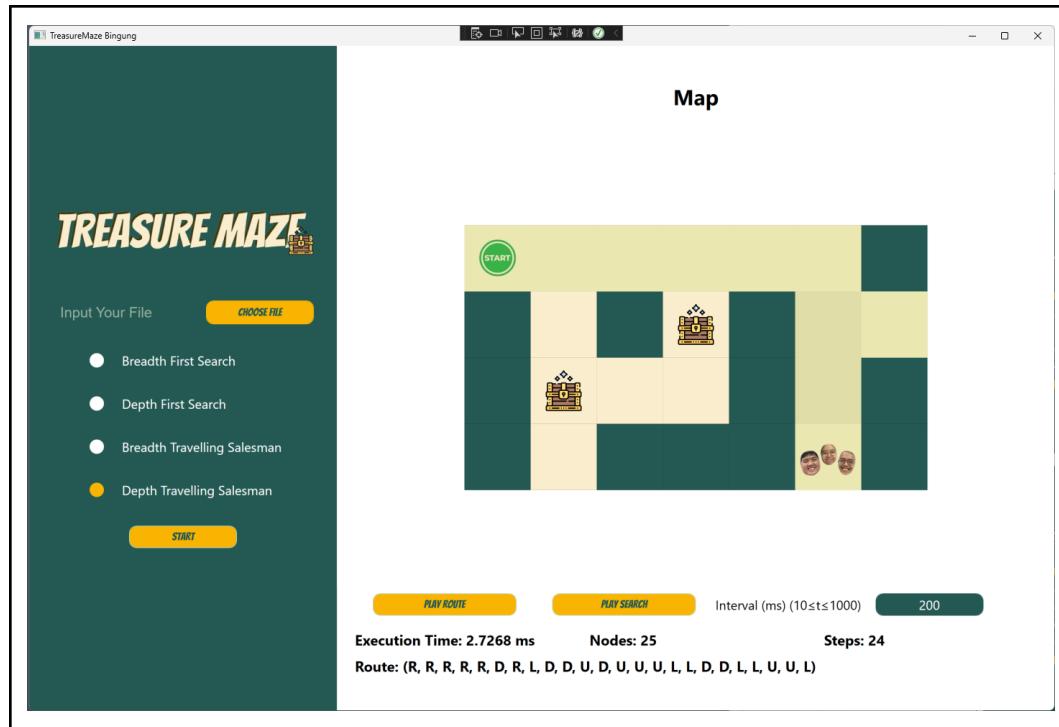


5. Hasil Pengujian DFS Travelling



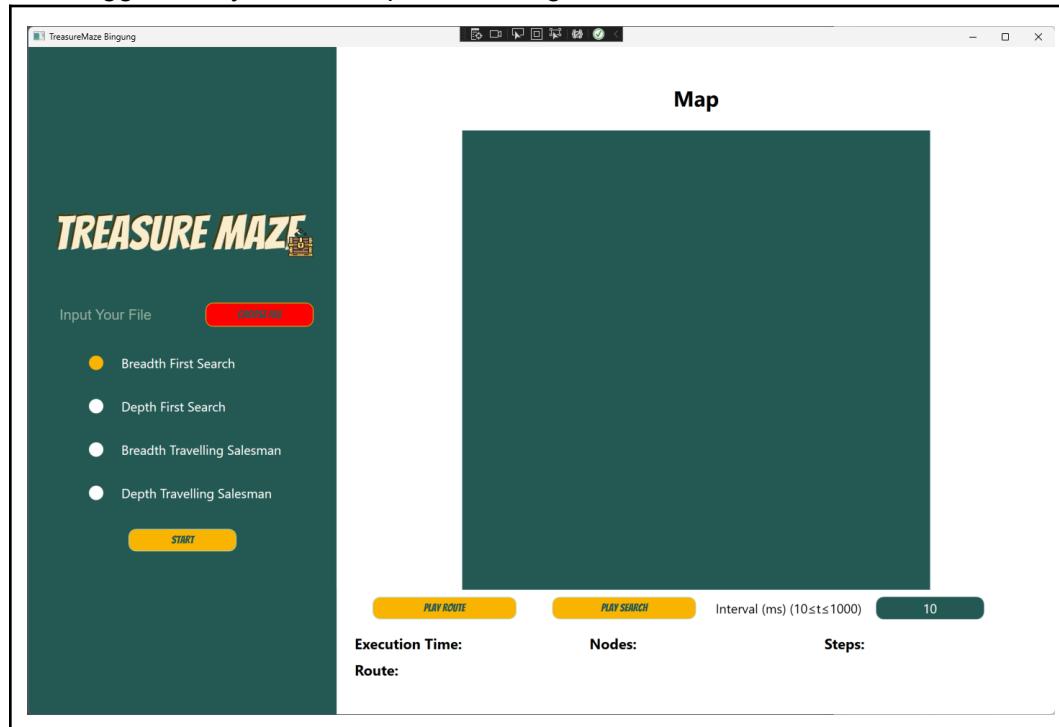
6. Beberapa Tampilan Visualisasi



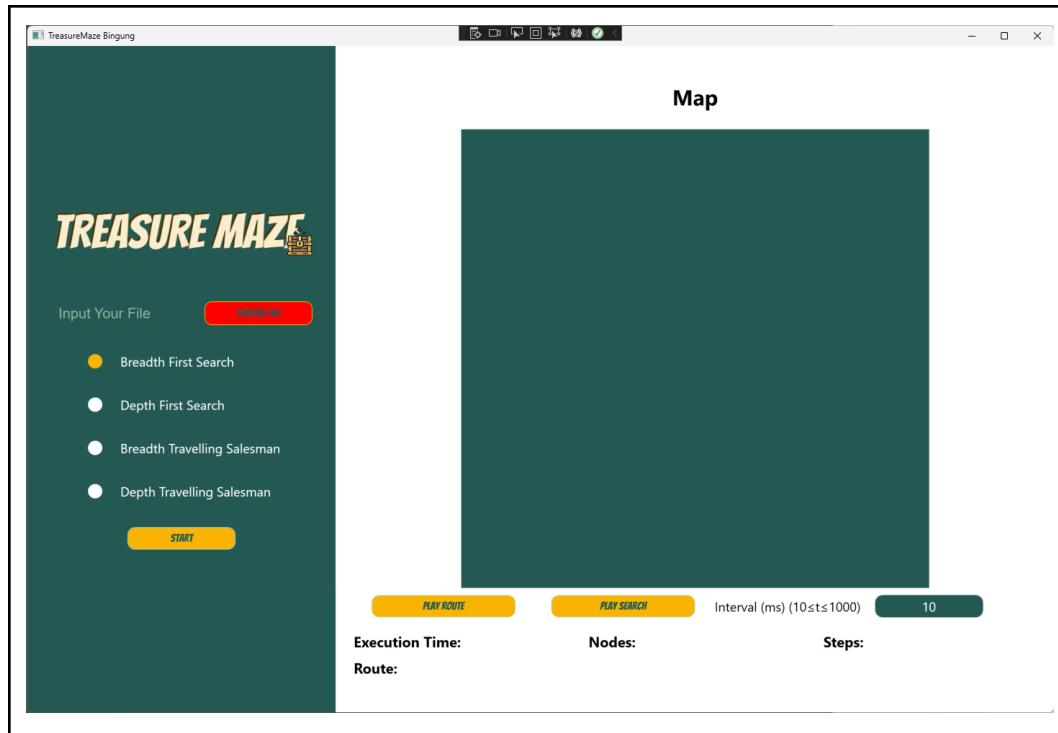


e. Pengujian 5 (Lainnya)

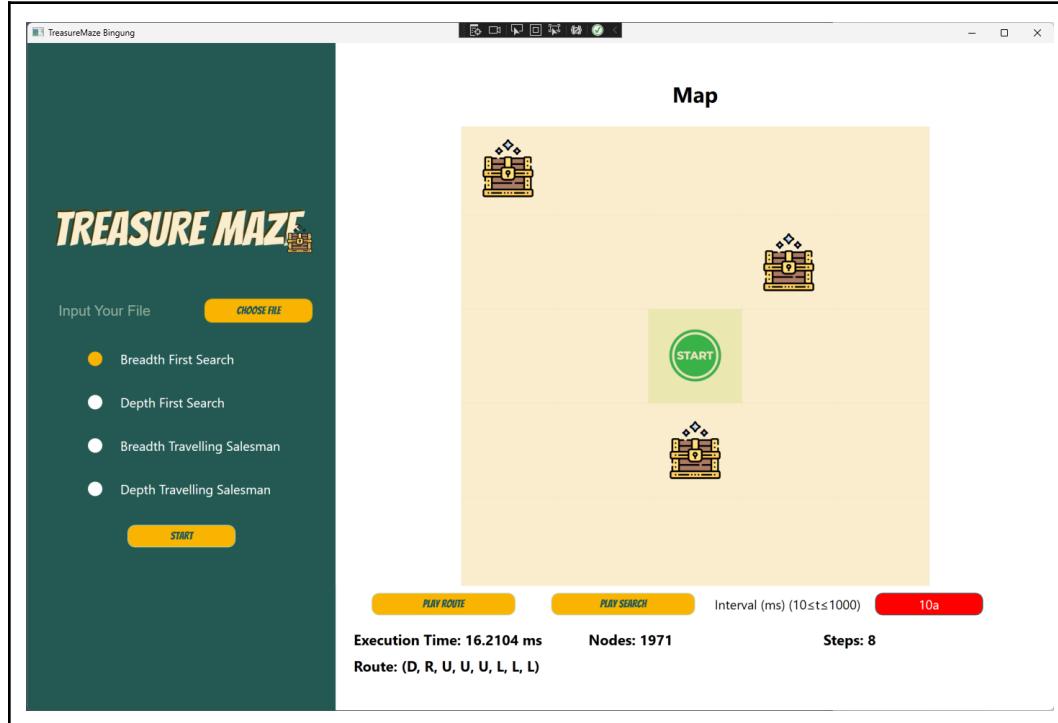
1. Pemanggilan Play Route Tanpa Peta Yang Valid



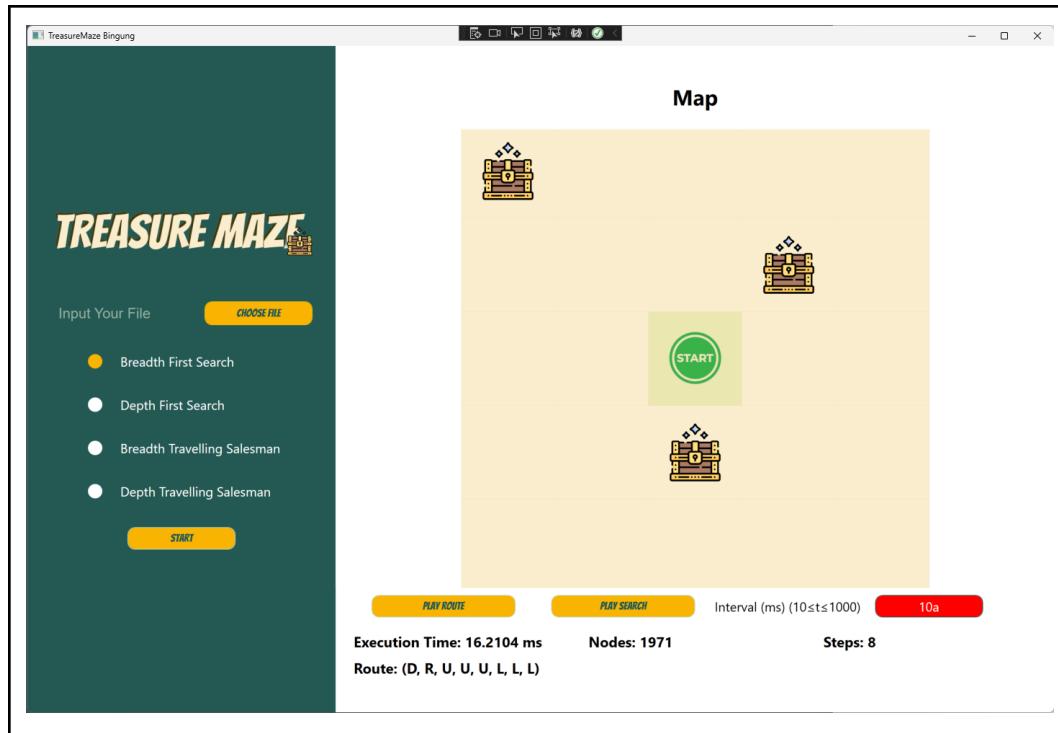
2. Pemanggilan Play Search Tanpa Peta Yang Valid



3. Pemanggilan Play Route Tanpa Interval Yang Valid



4. Pemanggilan Play Search Tanpa Interval Yang Valid



4.5. Analisis Desain Solusi

Analisis desain solusi yang ditampilkan di bawah ini merupakan hasil analisis dari setiap pengujian yang telah ditampilkan pada sub bab sebelumnya.

Pengujian ke-	Analisis
1	Pada pengujian pertama, peta yang dimasukkan tidaklah valid. Dengan begitu, tidak bisa dijadikan acuan untuk menentukan metode apa yang lebih baik dalam menyelesaikan permasalahan yang diberikan.
2	Pada pengujian kedua, peta yang digunakan relatif kecil dan sederhana. Dengan begitu, tidak banyak perbedaan yang ditampilkan baik menggunakan metode BFS maupun DFS. Namun, dapat dilihat bahwa penggunaan metode BFS membutuhkan jumlah <i>nodes</i> yang lebih banyak. Ditambah lagi, waktu eksekusi yang dibutuhkan BFS lebih lama dibandingkan DFS. Maka dari itu, untuk kasus sederhana seperti ini, penggunaan BFS ataupun DFS menghasilkan hasil dengan margin yang relatif kecil.
3	Pada pengujian ini, perbedaan hasil antara BFS dan DFS mulai terlihat. Dapat diketahui bahwa metode BFS masih membutuhkan jumlah <i>nodes</i> yang lebih banyak dibandingkan metode DFS. Namun, jumlah langkah yang didapatkan oleh metode DFS lebih

	banyak dibandingkan metode BFS. Hal ini membuktikan bahwa metode DFS tidak selalu menghasilkan jarak terpendek. Namun, mengingat perbedaan jumlah langkah yang sedikit, dalam hal ini metode DFS masih lebih baik karena mampu menemukan solusi dalam waktu yang lebih singkat.
4	Pada pengujian ini, BFS dan DFS menghasilkan rute yang sama. Hanya saja, BFS memerlukan jumlah <i>nodes</i> yang jauh lebih banyak daripada DFS. Hal ini mengindikasikan bahwa DFS jauh lebih efisien dari segi kompleksitas ruang. Namun, ada sedikit perbedaan rute pada pengujian TSP. Algoritma BFS menghasilkan rute yang lebih pendek dibandingkan DFS. Dengan begitu, dalam hal ini penggunaan metode BFS dinilai lebih mangkus dalam menjawab permasalahan yang diberikan.
5	Dalam pengujian ini, tidak ada algoritma BFS maupun DFS yang dibandingkan. Namun, pengujian ini lebih mengarah kepada skenario alternatif yang dapat ditemui <i>user</i> selama penggunaan program, beserta perbedaan dari kedua skenario tersebut.

BAB 5

KESIMPULAN DAN SARAN

5.1. Kesimpulan

Dalam tugas besar ini, kami mengembangkan suatu aplikasi desktop berbasis GUI yang menyelesaikan persoalan *maze*. Penyelesaian persoalan *maze* diselesaikan dengan traversal graf, secara khusus dengan menggunakan algoritma BFS dan DFS. Dalam pengerjaan tugas besar ini, kami belajar untuk memetakan persoalan ke dalam representasi graf serta menyelesaikan persoalan tersebut dengan algoritma BFS dan DFS. Selain itu, kami juga belajar untuk mengembangkan GUI yang baik untuk program yang telah kami buat. Kami juga menemukan bahwa algoritma BFS dan DFS memiliki kelebihan dan kekurangannya masing-masing dalam pemecahan persoalan *maze*. Tautan yang berisi *source code* tugas ini serta video penjelasan kami dapat dilihat pada bagian Lampiran.

5.2. Saran

Saran-saran untuk kelompok kami dalam pengerjaan Tugas Besar 2 IF2211 Strategi Algoritma Semester 2 Tahun Ajaran 2022/2023 adalah sebagai berikut:

1. Perlunya membiasakan diri melakukan pengembangan program dengan menggunakan berbagai IDE, sehingga dapat menggunakan IDE paling sesuai dengan proyek yang sedang dikerjakan.
2. Perlunya bijak dalam beraktivitas dalam masa pengerjaan tugas besar, sehingga tidak berakibat pada pengerjaan tugas besar yang terganggu. Misalnya, tidak bermain *paintball* saat masa pengerjaan tugas besar.

5.3. Refleksi

Dalam tugas besar ini, kami belajar untuk memetakan persoalan ke dalam struktur yang lebih sistematis, sehingga mempermudah penyelesaian persoalan. Kami juga belajar bahwa algoritma yang berbeda dapat memberikan hasil yang berbeda juga untuk persoalan yang sama. Selain itu, tiap-tiap algoritma memiliki kelebihan dan kekurangannya masing-masing. Terdapat *trade-offs* untuk tiap kelebihan yang ada. Kami juga belajar untuk mengembangkan GUI yang baik dan sesuai. Selain itu, kami belajar untuk membiasakan diri dengan berbagai IDE yang berbeda. Sebelumnya, kami hanya terbiasa menggunakan *VSCode*. Namun, dalam pengerjaan tugas besar ini, kami belajar untuk menggunakan *Visual Studio*. Terakhir, kami belajar untuk mempergunakan waktu kami dengan bijaksana dan juga belajar menjaga komunikasi dalam bekerja sama.

5.4. Tanggapan

Menurut kami, tugas besar kali ini cukup menyenangkan untuk dikerjakan. Hanya saja, kami harus lebih banyak melakukan eksplorasi dalam penggerjaan tugas besar ini, khususnya dalam membangun GUI yang baik. Eksplorasi ini cukup memakan waktu dan tenaga. Namun, terlepas dari itu semua, kami mempelajari cukup banyak hal.

DAFTAR PUSTAKA

Computer Hope. (2021, December 4). *What is a GUI (Graphical User Interface)?*. Retrieved March 23, 2023, from <https://www.computerhope.com/jargon/g/gui.htm>

GeeksforGeeks. (n.d.-a). *Difference between BFS and DFS*. Retrieved March 22, 2023, from <https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/>

GeeksforGeeks. (2022, June 6). *Travelling Salesman Problem implementation using BackTracking*. <https://www.geeksforgeeks.org/travelling-salesman-problem-implementation-using-backtracking/>

Microsoft Learn. (2023, February 13). *A tour of the C# language*. Retrieved March 22, 2023, from <https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/>

Munir, R. (n.d.-a). *Breadth/ Depth First Search (BFS/ DFS) Bagian 1*. Retrieved March 22, 2023, from <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf>

Munir, R. (n.d.-b). *Breadth/ Depth First Search (BFS/ DFS) Bagian 2*. Retrieved March 22, 2023, from <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf>

tutorialspoint. (n.d.-b). *Difference between BFS and DFS*. Retrieved March 22, 2023, from <https://www.tutorialspoint.com/difference-between-bfs-and-dfs>

LAMPIRAN

Link repository GitHub :

https://github.com/Gulilil/Tubes2_bingung

Link video YouTube:

<https://youtu.be/uSijep5Uec>

TABEL CHECKLIST SPEK

Poin	Ya	Tidak
1. Program berhasil dikompilasi tanpa ada kesalahan.	✓	
2. Program berhasil running.	✓	
3. Program dapat menerima dan melakukan validasi masukan (<i>input</i>).	✓	
4. Program berhasil menemukan penyelesaian program menggunakan metode BFS dan DFS dan menampilkan hasil solusi tersebut sebagai <i>output</i> .	✓	
5. Rute keluaran (<i>output</i>) yang dihasilkan program sudah benar (mengandung semua lokasi <i>treasure</i> yang ada pada <i>map</i> yang telah dibaca oleh program).	✓	
6. Program berhasil menampilkan GUI dengan spesifikasi minimal GUI yang telah ditentukan.	✓	
7. Bonus 1 (Implementasi proses pencarian <i>grid</i>) dikerjakan	✓	
8. Bonus 2 (Implementasi penyelesaian TSP) dikerjakan	✓	
9. Bonus 3 (Video YouTube) dikerjakan	✓	

PEMBAGIAN TUGAS

NIM	Nama	Tugas
13521111	Tabitha Permallia	BFS, Bonus TSP, dan Laporan
13521116	Juan Christopher Santoso	DFS, Bonus TSP, dan Laporan
13521162	Antonia Natthan Krishna	GUI, Bonus proses pencarian <i>grid</i> , dan Laporan