

**Tugas Besar 2 IF2211 Strategi Algoritma
Semester II Tahun 2022/2023
Pengaplikasian Algoritma BFS dan DFS dalam Menyelesaikan
Persoalan Maze Treasure Hunt**



Disusun oleh
13521074 Eugene Yap Jin Quan
13521123 William Nixon
13521155 Kandida Edgina Gunawan

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

DAFTAR ISI

DAFTAR ISI	2
BAB 1	4
DESKRIPSI TUGAS	4
BAB 2	7
LANDASAN TEORI	7
2.1. Algoritma Pencarian dan Penelusuran pada Graf	7
2.1.1. Graph Traversal	7
2.1.2. Breadth First Search (BFS)	7
2.1.3. Depth First Search (DFS)	8
2.2. C# Desktop Application Development	9
2.3. WPF (Windows Presentation Foundation)	10
BAB 3	12
ANALISIS PEMECAHAN MASALAH	12
3.1. Langkah-Langkah Pemecahan Masalah	12
3.2. Mapping Persoalan Menjadi Elemen-Element Algoritma BFS dan DFS	12
3.2.1. Identifikasi Elemen Penyelesaian Masalah pada Implementasi Algoritma BFS/DFS	12
3.2.2. Perancangan Elemen Graf untuk Persoalan BFS/DFS pada Kelas Maze	13
3.2.3. Perancangan Penyimpanan Informasi Umum Terkait Pencarian Solusi pada Kelas Abstrak Player	13
3.2.4. Perancangan Penyimpanan Informasi Tambahan pada Kelas dan Algoritma BFS	15
3.2.5. Perancangan Penyimpanan Informasi Tambahan pada Kelas dan Algoritma DFS	16
3.3. Ilustrasi Kasus Penyelesaian Menggunakan DFS dan BFS	17
BAB 4	20
IMPLEMENTASI DAN PENGUJIAN	20
4.1. Implementasi Algoritma Pencarian Solusi	20
4.1.1. Pseudocode Algoritma BFS (Breadth First Search)	20
4.1.2. Pseudocode Algoritma DFS (Depth First Search)	22
4.1.3. Pseudocode Flow Program Utama (Direpresentasikan GUI)	25
4.2. Struktur Data dan Spesifikasi Program	25
4.2.1. Penggunaan Struktur Data pada Program	25
4.2.2. Spesifikasi Program	28
4.3. Tata Cara Penggunaan Program	29
4.4. Hasil Pengujian dan Analisis	32
4.4.1. Pengujian untuk test case sample-1.txt	32
4.4.2. Pengujian untuk test case sample-2.txt	34

Laporan Tugas Besar II
IF2211 Strategi Algoritma

4.4.3. Pengujian untuk test case sample-3.txt	35
4.4.4. Pengujian untuk test case sample-4.txt	36
4.4.5. Pengujian untuk test case sample-5.txt	38
4.4.6. Pengujian Pruning	40
4.4.7. Pengujian Kasus Lain	42
4.5. Analisis Desain Solusi Algoritma BFS dan DFS	42
BAB 5	44
KESIMPULAN DAN SARAN	44
5.1. Kesimpulan	44
5.2. Saran	44
5.3. Refleksi	44
5.4. Tanggapan Terkait Tugas Besar	44
DAFTAR PUSTAKA	45
LAMPIRAN	46

BAB 1

DESKRIPSI TUGAS

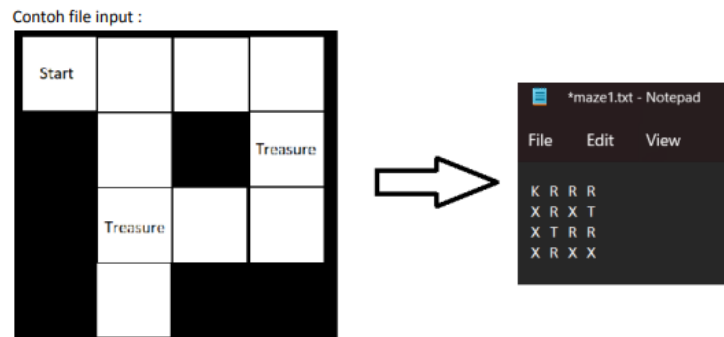
Dalam tugas besar yang kedua ini, sebuah aplikasi dengan GUI sederhana dibuat untuk mengimplementasikan BFS dan DFS untuk mendapatkan rute yang memperoleh seluruh *treasure* dari suatu peta masukan dari pengguna. Program dapat menerima dan membaca *file input* txt yang berisi konfigurasi peta berbentuk segiempat dengan tiap bloknya berada pada *state* tertentu. Berikut ini keterangan dari tiap *state* dari blok yang dimuat pada peta

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 file maze*

Dengan menggunakan algoritma *Breadth First Search (BFS)* dan *Depth First Search (DFS)*, dapat ditelusuri blok-blok yang dikunjungi yang akan membentuk rute solusi yang merupakan rute yang memperoleh semua *treasure* di pada *maze*. Pergerakan hanya dapat dilakukan dalam arah *Left*, *Right*, dan *Down* dengan skala prioritasnya ditentukan oleh kelompok masing-masing. File txt yang dimasukkan pengguna serta hasil pencarian rute solusi nantinya akan ditampilkan pada program yang dibuat.

Daftar *file input* akan dimasukkan ke dalam sebuah folder *test* pada *repository* program. Letak folder tersebut sejajar dengan folder *src* dan *doc*. Aplikasi dapat menerima *input maze* melalui textfield (mengetikkan langsung nama file) atau dengan input file langsung.

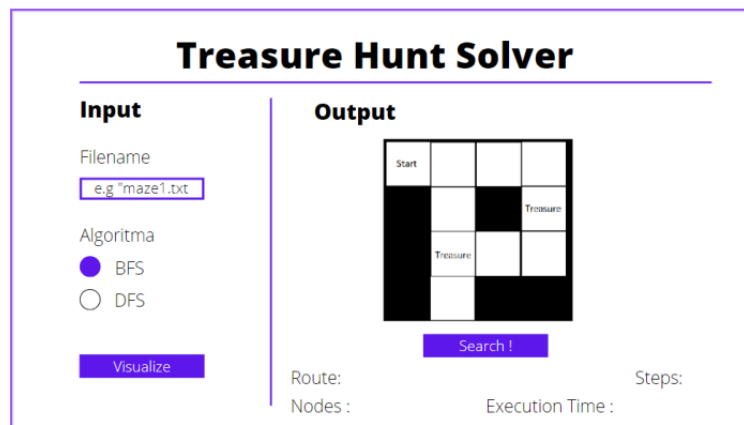
Setelah melakukan pembacaan *input*, program akan memvisualisasikan *maze* masukan yang kosong (belum memiliki solusi), kemudian ketika pengguna menekan tombol solve, program akan menjalankan algoritma DFS dan BFS, serta program akan memberi warna pada rute solusi.

Spesifikasi GUI:

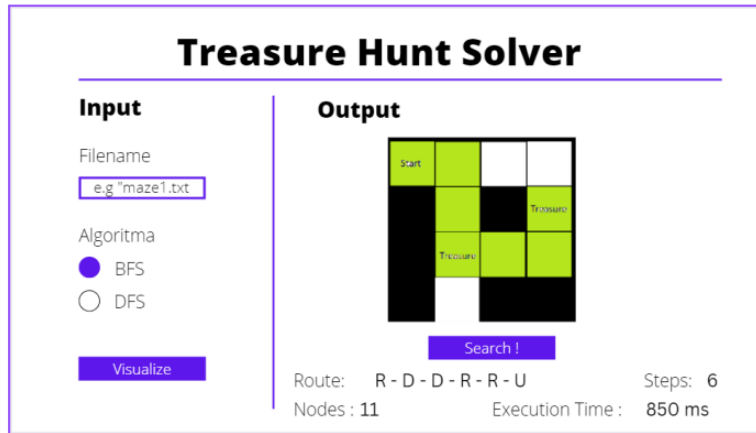
1. Masukan program adalah file maze treasure hunt tersebut atau nama filenya.
2. Program dapat menampilkan visualisasi dari input file maze dalam bentuk grid dan pewarnaan sesuai deskripsi tugas.
3. Program memiliki toggle untuk menggunakan alternatif algoritma BFS ataupun DFS.
4. Program memiliki tombol 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. 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. 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).

Spesifikasi wajib lainnya yang harus dipenuhi adalah sebagai berikut:

1. Program dibuat dengan bahasa C#
2. Apabila filename input dari *textfield* 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.



Gambar 1.2. Tampilan Program Sebelum Pencarian Solusi



Gambar 1.3. Tampilan Program Setelah Pencarian Solusi

BAB 2

LANDASAN TEORI

2.1. Algoritma Pencarian dan Penelusuran pada Graf

2.1.1. *Graph Traversal*

Graf biasanya digunakan untuk merepresentasikan suatu persoalan. Traversal graf memiliki arti pencarian solusi dari suatu persoalan yang dimodelkan dalam bentuk graf. Algoritma traversal graf adalah sekumpulan instruksi atau langkah-langkah sistematis untuk mengunjungi simpul-simpul dari suatu graf. Algoritma traversal graf secara garis besar dapat dibagi menjadi dua, yaitu pencarian melebar (*Breadth First Search / BFS*) dan pencarian mendalam (*Depth First Search/DFS*).

Dalam proses pencarian solusi, terdapat dua pendekatan representasi graf dalam proses pencarian:

a. Graf statis

Graf statis adalah graf yang sudah terbentuk sebelum proses pencarian dilakukan. Graf direpresentasikan sebagai suatu struktur data.

b. Graf dinamis

Graf dinamis adalah graf yang terbentuk saat proses pencarian dilakukan. Graf awalnya tidak tersedia sebelum pencarian, graf dibangun selama pencarian solusi.

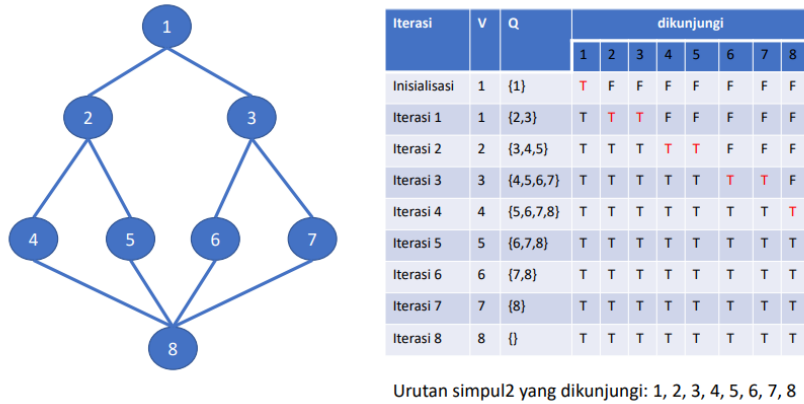
Pohon dinamis dapat disebut juga sebagai pohon ruang status. Simpul dari suatu pohon dinamis merupakan *problem state* dengan akar dari pohon sebagai *initial state* dan daun dari pohon sebagai *goal state*. Cabang dari pohon menggambarkan operator atau langkah dalam persoalan. Selain itu terdapat pula ruang status yang merupakan himpunan semua simpul dan ruang solusi yang merupakan himpunan status solusi. Solusi dari pencarian pada graf dinamis merupakan *path* ke status solusi.

2.1.2. *Breadth First Search (BFS)*

a. Breadth First Search (BFS) pada graf statis

Berikut ini merupakan contoh algoritma pencarian BFS:

1. Mengunjungi simpul 1
2. Mengunjungi semua simpul yang bertetangga dengan simpul 1 terlebih dahulu
3. Mengunjungi simpul yang belum dikunjungi dan bertetangga dengan simpul-simpul yang tadi dikunjungi, demikian seterusnya sampai ditemukan solusi yang diharapkan dari suatu persoalan

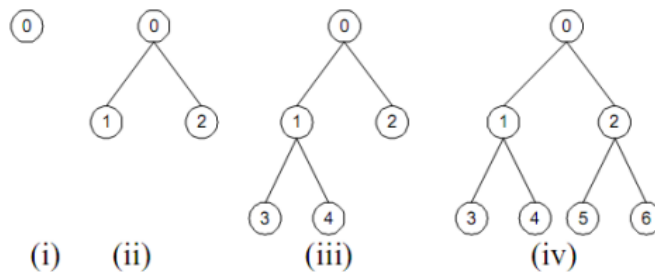


Gambar 2.1. Ilustrasi dari Breadth First Search

b. Breadth First Search (BFS) pada graf dinamis

Pencarian solusi dengan BFS ataupun DFS pada graf dinamis sama dengan pembentukan pohon dinamis itu sendiri. Setiap simpul yang terbentuk nantinya akan diperiksa apakah solusi telah tercapai atau tidak. Jika solusi telah tercapai, pencarian dapat dihentikan.

Pembangkitan pohon ruang status pada BFS dapat dilihat sebagai berikut:



Gambar 2.2. Ilustrasi Pembentukan Pohon Ruang Status dengan BFS

- Inisialisasi dengan status awal sebagai akar, lalu tambahkan simpul anaknya, dan seterusnya
- Semua simpul pada level d harus dibangkitkan sebelum membangkitkan simpul pada level d+1

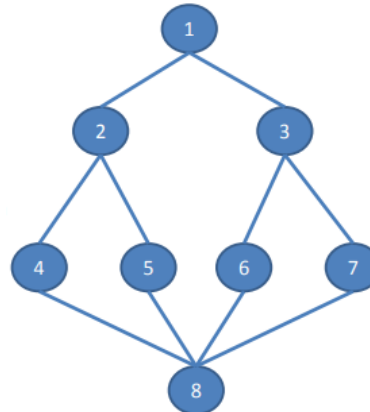
2.1.3. Depth First Search (DFS)

a. Depth First Search (DFS) pada graf statis

Berikut ini merupakan contoh algoritma pencarian dengan DFS:

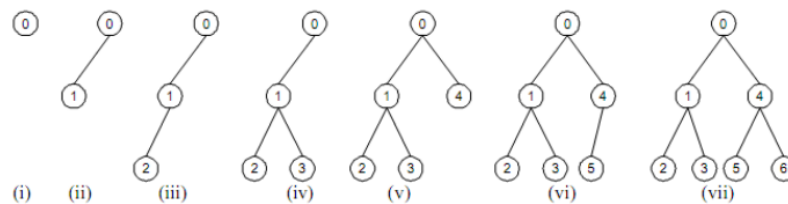
- Mengunjungi simpul 1
- Mengunjungi simpul 2 yang bertetangga dengan simpul 1

3. Ulangi DFS mulai dari simpul 2
4. Ketika mencapai simpul 8 yang mana semua simpul yang bertetangga dengannya telah dikunjungi, lakukan pencarian runut balik ke simpul terakhir yang dikunjungi sebelumnya dan mempunyai simpul 5 yang belum dikunjungi.
5. Pencarian berakhir bila tidak ada lagi simpul yang belum dikunjungi yang dapat dicapai dari simpul yang telah dikunjungi.



Gambar 2.3. Contoh Graf sebagai Demonstrasi DFS

b. Depth First Search (DFS) pada graf dinamis



Gambar 2.4.. Ilustrasi Pembentukan Pohon Ruang Status dengan DFS

- a) Inisialisasi dengan status awal sebagai akar, lalu tambahkan simpul yang bertetangga dengannya, dan seterusnya
- b) Apabila tidak terdapat lagi simpul yang bertetangga dengannya, lakukan runut balik lalu tambahkan simpul yang bertetangga dengannya.

2.2. C# Desktop Application Development

C# Desktop Application Development adalah pembuatan aplikasi perangkat lunak dengan menggunakan bahasa pemrograman C# yang dapat beroperasi di komputer tanpa adanya sambungan dengan internet. Pembuatan aplikasi desktop dengan bahasa C# dapat dilakukan dengan beberapa *development tools*, salah satunya adalah dengan menggunakan *Microsoft Visual Studio*. *Visual Studio* merupakan salah satu *Integrated Development Environment (IDE)* yang digunakan untuk membuat suatu aplikasi dengan mudah dan cepat. Pengembangan aplikasi *desktop* ini juga dipermudah dengan menggunakan WPF.

Berikut ini adalah langkah-langkah umum yang dapat diikuti untuk mengembangkan suatu *desktop application* pada *Visual Studio*:

1. Membuka aplikasi Visual Studio dan pilih Create new project
2. Pada jendela *Create a new project*, pilih *Windows Presentation Foundation (.Net Framework)*
3. Masukkan nama project yang dibuat, serta klik tombol *Create*
4. Memilih menu *Toolbox* untuk membuka jendela *Toolbox fly-out*
5. Pada *Toolbox fly-out*, dapat dipilih komponen-komponen yang diinginkan untuk aplikasi *desktop* yang sedang dibangun. Kita juga dapat memodifikasi komponen dengan mengubah properti pada bagian *properties*.
6. Untuk menjalankan aplikasi yang telah dibuat, tekan tombol *Start* pada *menu*.

2.3. WPF (Windows Presentation Foundation)

WPF (Windows Presentation Foundation) adalah kerangka kerja (framework) aplikasi yang dikembangkan oleh Microsoft untuk membangun aplikasi desktop pada sistem operasi Windows. WPF memungkinkan pengembang untuk membuat antarmuka pengguna (user interface) yang modern, interaktif, dan visual yang dapat menampilkan berbagai jenis konten multimedia seperti teks, gambar, video, dan animasi.

WPF didasarkan pada bahasa pemrograman C# atau VB.NET dan menggunakan platform .NET Framework. Kerangka kerja ini memiliki fitur yang kuat seperti pengaturan tata letak (layout), animasi, data binding, pengujian unit, dan integrasi dengan teknologi web. WPF menggunakan XML-based markup language yang disebut XAML (eXtensible Application Markup Language) untuk membuat antarmuka pengguna.

XAML memungkinkan pengembang untuk membuat antarmuka pengguna secara deklaratif, yang berarti antarmuka pengguna dapat didefinisikan secara terpisah dari kode aplikasi. Dalam hal ini, WPF menyediakan kemudahan dalam mengembangkan aplikasi desktop dengan antarmuka pengguna yang menarik dan mudah digunakan bagi pengguna.

2.4. Code Behind

Code-behind pada WPF merujuk pada kode pemrograman yang terletak di belakang file XAML yang digunakan untuk mendefinisikan antarmuka pengguna. File XAML hanya digunakan untuk mendefinisikan tampilan antarmuka pengguna, sedangkan code-behind digunakan untuk menambahkan logika aplikasi yang diperlukan untuk mengontrol perilaku antarmuka pengguna.

Code-behind pada WPF biasanya ditulis dalam bahasa pemrograman C# atau VB.NET dan dapat diakses melalui file .cs atau .vb yang terkait dengan file XAML. Kode ini dapat mengakses semua elemen antarmuka pengguna yang didefinisikan dalam file XAML dan memodifikasinya sesuai kebutuhan.

Code-behind dapat digunakan untuk menambahkan berbagai jenis logika aplikasi, termasuk logika untuk mengatur perilaku kontrol antarmuka pengguna, mengelola data, menjalankan kode pada peristiwa tertentu, dan berinteraksi dengan API sistem operasi Windows. Code-behind juga dapat digunakan untuk membuat fungsi dan kelas khusus yang dapat digunakan di seluruh aplikasi.

BAB 3

ANALISIS PEMECAHAN MASALAH

3.1. Langkah-Langkah Pemecahan Masalah

Langkah pertama yang diambil untuk mendesain solusi yang akan memecahkan masalah *Maze Treasure Hunt* ini adalah identifikasi submasalah agar dapat diselesaikan secara modular. Dua persoalan utama adalah algoritma pencarian rute untuk mendapatkan semua *treasure* dari *maze* dengan algoritma BFS (*Breadth First Search*) dan dengan algoritma DFS (*Depth First Search*). Persoalan berikutnya adalah pembacaan dan validasi peta konfigurasi *maze* dari masukan pengguna berupa *file* teks. Selain itu, persoalan lain yang juga perlu diselesaikan adalah memvisualisasikan hasil pencarian rute dari dua algoritma tersebut dengan menggunakan GUI (*graphical user interface*). Adapun persoalan bonus yang dikerjakan, yaitu visualisasi progres pencarian grid dan juga pencarian rute TSP *maze* dengan menggunakan algoritma DFS ataupun BFS.

3.2. Mapping Persoalan Menjadi Elemen-Elemen Algoritma BFS dan DFS

Pada persoalan ini, algoritma BFS dan DFS yang diterapkan merupakan algoritma pencarian solusi dari graf dinamis. Dalam kasus ini, graf persoalan berwujud labirin atau *maze*, dan solusi merupakan rute yang melewati setiap titik khusus (*treasure*). Berdasarkan persoalan tersebut, berikut ini merupakan identifikasi elemen-elemen penting yang terdapat pada algoritma BFS dan DFS.

3.2.1. Identifikasi Elemen Penyelesaian Masalah pada Implementasi Algoritma BFS/DFS

Berdasarkan analisis kami, kami menemukan beberapa elemen penting yang digunakan dalam proses pencarian rute solusi. Elemen-elemen tersebut adalah sebagai berikut.

- *Initial state*: titik atau simpul pertama pada *maze* (disimbolkan dengan 'K').
- *Goal state*: titik yang terakhir dikunjungi pada sebuah rute yang berhasil melewati semua *treasure* pada *maze* (khusus TSP, titik tersebut adalah titik awal *maze*).
- *Operator*: penambahan titik tetangga (urutan prioritas terpilih: LRUD) pada antrian pencarian dengan syarat belum ditelusuri pada status rute. Pada kasus khusus jalan buntu, syarat penambahan titik dimodifikasi untuk menambahkan titik yang mengeluarkan posisi pemain dari jalan buntu (*backtrack*).
- *State space*: himpunan rute yang disimpan pada setiap langkah eksplorasi titik *maze*.
- *Solution space*: himpunan rute yang mampu mencapai *goal state*.

Implementasi elemen-elemen tersebut memanfaatkan elemen-elemen penyimpanan informasi yang dijelaskan pada bagian berikutnya.

3.2.2. Perancangan Elemen Graf untuk Persoalan BFS/DFS pada Kelas *Maze*

A screenshot of a code editor showing the implementation of the Maze class. The code is in Java and includes private fields for a map matrix, dimensions, start point, and treasure set, followed by a comment indicating further implementation.

```
1 public class Maze
2 {
3     private List<List<char>> _mapMatrix;
4     private int _nRows;
5     private int _nCols;
6     private Point _mazeStart;
7     private List<Point> _treasureSet;
8
9     // ....implementasi Maze
```

Gambar 3.2.1. Hasil Perancangan Elemen Kelas *Player*

Kelas *Maze* adalah kelas yang diimplementasikan untuk membuat objek *maze* atau peta dari permainan. Kelas ini berperan sebagai cetak biru dari graf *maze* persoalan BFS/DFS. Informasi-informasi dari *Maze* yang krusial untuk persoalan ini adalah informasi susunan *maze* itu sendiri yang disimpan sebagai matriks *char*, informasi jumlah baris dan jumlah kolom dari *maze* (*integer*), informasi titik awal *maze* (*Point*), dan informasi jumlah titik *treasure* yang disimpan secara implisit dalam *List of Point*. Informasi susunan *maze* dan ukuran *maze* digunakan dalam validasi pembangkitan *node* tetangga. Informasi titik awal *maze* digunakan dalam memulai pencarian solusi serta sebagai titik tujuan dari TSP. Adapun informasi jumlah titik *treasure* yang digunakan dalam pengecekan solusi rute.

3.2.3. Perancangan Penyimpanan Informasi Umum Terkait Pencarian Solusi pada Kelas Abstrak *Player*

A screenshot of a code editor showing the implementation of the abstract Player class. The code includes various protected and private fields for state, configuration, and logging, followed by a comment indicating further implementation of methods.

```
1 public abstract class Player
2 {
3     /* Properties & State Fields */
4     protected List<Point> _exploredNodes;
5     protected Maze _mazeMap;
6     public List<string> _playerDirectionState;
7     public List<string> _numSteps;
8     public List<string> _numNodes;
9     public long _recordedSearchTime;
10    protected bool _isGoalFinished;
11    protected bool _isTspStarted;
12    protected bool _isTspFinished;
13
14    /* Config Fields */
15    protected bool _branchPruningEnabled;
16    protected bool _tspEnabled;
17
18    /* Logger-Utility Fields */
19    private string _filename;
20    private List<int[,]> _mazeStateLog;
21    private string _solutionRoute;
22    private int[, ] _mazeStateTemplate;
23
24    // ....implementasi method Player
```

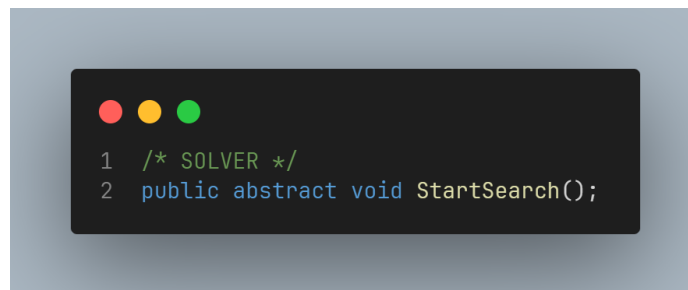
Gambar 3.2.2. Hasil Perancangan Elemen Kelas *Player*

Pada perancangan kelas abstrak *Player*, kami menetapkan bahwa objek-objek dari kelas ini perlu menyimpan beberapa informasi terkait pencarian solusi rute pada *maze*. Terkait pencarian solusi, objek memerlukan penyimpanan informasi peta *maze* yang akan ditelusuri, informasi ekspansi rute pada setiap langkah, penanda boolean terkait tercapainya solusi rute, penanda boolean terkait tercapainya solusi TSP, dan informasi waktu komputasi solusi. Terkait konfigurasi, kami menambahkan penanda boolean terkait aktifnya fitur tambahan, yaitu fitur TSP dan fitur *pruning*.

Apabila fitur TSP diaktifkan, pencarian solusi akan dilanjutkan dengan pencarian rute balik. Di sisi lain, fitur *pruning*, yang secara default aktif, akan memangkas sebuah cabang pencarian dan tidak melakukan *backtracking* apabila tidak terjadi penambahan jumlah *treasure* sejak persimpangan terdekat.

Terdapat juga informasi lain yang disimpan pada *Player*, yaitu nama *file*, status eksplorasi, template informasi status, dan solusi. Keempat informasi ini digunakan untuk berkomunikasi dengan antarmuka program terkait pewarnaan *maze*. Informasi ini disampaikan melalui media *log* berupa *file* teks.

Field dan *method* yang didefinisikan pada kelas abstrak ini kemudian digunakan sebagai cetak dasar dari kelas *BFSPlayer* dan *DFSPlayer*. Adapun *method* abstrak *StartSearch* yang berfungsi untuk memulai pencarian solusi. *Method* khusus ini diimplementasikan pada kelas turunannya.



```

1  /* SOLVER */
2  public abstract void StartSearch();

```

Gambar 3.2.3. *Method* Abstrak pada Kelas *Player*

3.2.4. Perancangan Penyimpanan Informasi Tambahan pada Kelas dan Algoritma BFS



```
1 public class BFSPlayer : Player
2 {
3     /* INNER CLASS: BFSPoint */
4     public class BFSPoint
5     {
6         private Point _currentPosition;
7         private string _movementSteps;
8         private int _treasureCount;
9         private int _branchTreasureGain;
10        private int _backtrackFlag;
11        private List<Point> _pointSteps;
12        private List<Point> _tspSteps;
13        // ....implementasi BFSPoint
14    }
15    // ....implementasi BFSPlayer
16 }
```

Gambar 3.2.4. Implementasi *Inner Class* dalam Kelas *BFSPlayer*

Pada perancangan *BFSPlayer*, kami menambahkan sebuah kelas objek bernama *BFSPoint* yang merupakan penambahan terhadap *Point* biasa. Alasan penambahan

BFSPoint bertugas menyimpan beberapa informasi, seperti posisi *node* saat ini yang bertipe *Point*, urutan langkah-langkah yang telah dilakukan untuk sampai ke posisi *node* saat ini (bertipe *string*), banyaknya *treasure* yang telah diakses dari titik awal sampai ke titik saat ini (bertipe *integer*), serta banyaknya *branch treasure*, yaitu banyak *treasure* yang didapatkan sejak titik cabang terdekat (bertipe *integer*). Kami juga menyimpan *flag* yang menyatakan banyaknya langkah *backtrack* yang telah dilakukan (bertipe *integer*). Informasi *branch treasure* digunakan untuk penentuan keputusan untuk mencatat langkah *backtrack* dalam solusi, sedangkan informasi *flag backtrack* beserta urutan langkah dilakukan untuk mendapatkan langkah *backtrack* selanjutnya (dengan cara *string slicing* untuk mendapatkan langkah sebelum *backtrack*).

Untuk membangkitkan *node* tetangga yang dapat dikunjungi, kami mencatat semua *node* yang telah dikunjungi pada tahap pencarian. Untuk mencatat *node-node* apa saja yang sudah dikunjungi, digunakan *List* yang berisi *Point* yang akan menyimpan posisi *node-node* yang sudah dikunjungi. Terdapat dua buah *List* bertipe *Point* yang berturut-turut menyimpan posisi *node-node* yang telah dikunjungi pada tahap BFS sebagai *point steps* dan menyimpan posisi *node-node* yang telah dikunjungi pada tahap TSP sebagai *tsp steps*.

Dalam implementasi pencarian solusi BFS, objek-objek *BFSPoint* dibangkitkan dan disimpan dalam *Queue of BFSPoint* untuk mempertahankan urutan pencarian BFS. Implementasi

solusi ini dilakukan pada *method IterateBFS* dan *StartSearch* yang mengimplementasi *method* abstrak dari *Player*.



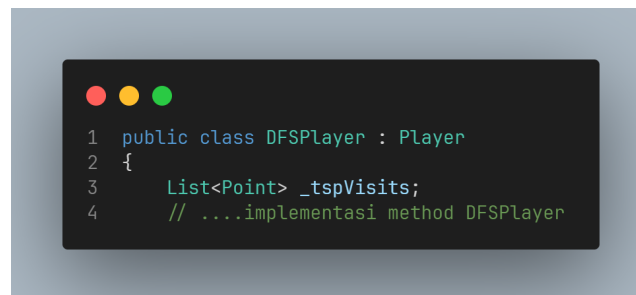
```

1  /* BFS Solution Methods */
2  public override void StartSearch()
3  {
4      Stopwatch searchTimer = new Stopwatch();
5      searchTimer.Start();
6      IterateBFS(this._mazeMap.StartPoint);
7      searchTimer.Stop();
8      this._recordedSearchTime = searchTimer.ElapsedMilliseconds;
9  }
10 }
11
12 public void IterateBFS(Point startNode)
13 {
14     BFSPoint mazeStart = new BFSPoint(this._mazeMap.StartPoint, "", 0, 0, 0);
15     mazeStart.AddSelfAsStep();
16     Queue<BFSPoint> searchQueue = new Queue<BFSPoint>();
17
18     searchQueue.Enqueue(mazeStart);
19
20     while (/* kondisi BFS */)
21     {
22         // ....algoritma BFS
23     }
24 }

```

Gambar 3.2.5. *Method* Pencarian Solusi BFS

3.2.5. Perancangan Penyimpanan Informasi Tambahan pada Kelas dan Algoritma DFS



```

1  public class DFSPlayer : Player
2  {
3      List<Point> _tspVisits;
4      // ....implementasi method DFSPlayer

```

Gambar 3.2.6. Field Tambahan pada *DFSPlayer*

Pada implementasi kelas *DFSPlayer*, kami menambahkan satu buah *field* di atas *field* dari kelas *Player* untuk menyimpan semua *node* yang telah dikunjungi pada tahap TSP. Hal ini bertujuan untuk membedakan fungsi pembangkitan *node* valid pada tahap DFS, yang menggunakan informasi *node* yang telah dikunjungi pada kelas dasar *Player*. Penyimpanan informasi ini menggunakan *List of Point*.

Dalam implementasi pencarian solusi DFS, informasi terkait pencarian rute disimpan sebagai argumen dari pemanggilan fungsi pencarian solusi. Implementasi solusi ini dilakukan pada *method RecurseDFS* dan *StartSearch* yang mengimplementasi *method* abstrak dari *Player*. Dalam kasus ini dapat dilihat bahwa, berbeda dengan *BFSPlayer*, *DFSPlayer* tidak

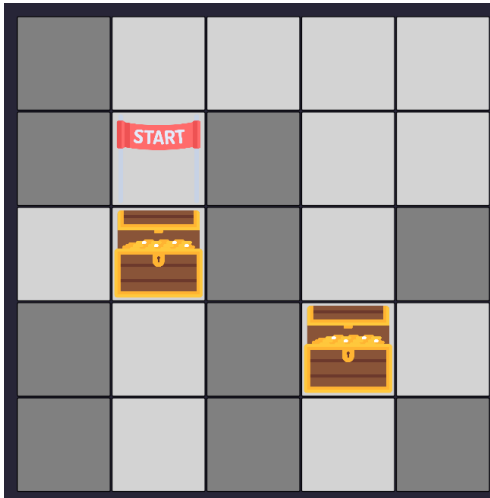
membutuhkan spesialisasi *Point* yang menyimpan informasi rute dan *treasure*. Hal ini karena informasi ini secara implisit disimpan pada *call stack* pemanggilan fungsi rekursif pencarian solusi.

```
1  /* DFS Solution Methods */
2  public override void StartSearch()
3  {
4      Stopwatch searchTimer = new Stopwatch();
5      searchTimer.Start();
6      RecurseDFS(this._mazeMap.StartPoint, 0, 0, "", "");
7      if (this.IsTspEnabled)
8      {
9          // ...solve TSP using DFS
10     }
11     searchTimer.Stop();
12     this._recordedSearchTime = searchTimer.ElapsedMilliseconds;
13 }
14
15 public void RecurseDFS(Point currentNode, int treasureCount, int treasureGain, string routeTaken, string backtrackRoute)
16 {
17     // ....algoritma DFS
18 }
```

Gambar 3.2.7. *Method* Pencarian Solusi DFS

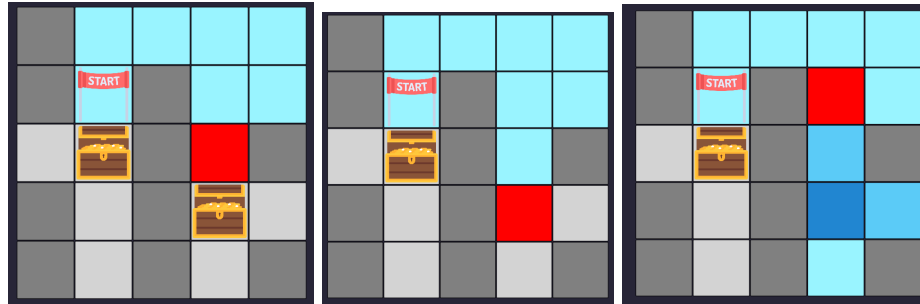
3.3. Ilustrasi Kasus Penyelesaian Menggunakan DFS dan BFS

Misalkan peta *maze* pencarian adalah sebagai berikut.



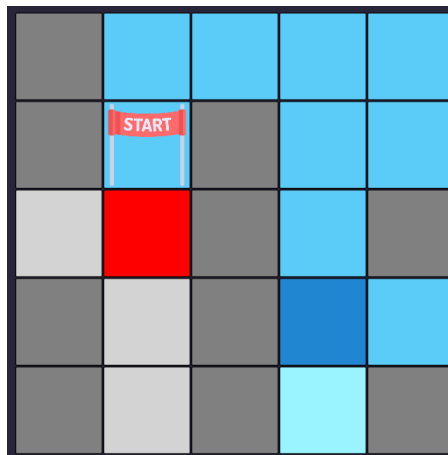
Gambar 3.3.1 Ilustrasi *Maze* Uji Coba Algoritma

Dengan urutan pembangkitan status LRUD, algoritma DFS akan melakukan pencarian terhadap *treasure* di sisi kanan terlebih dahulu (melalui jalur atas). Ketika mendapatkan *treasure* pada sebelah kanan, pencarian dihadapkan dengan dua buah simpul, yaitu simpul di bawahnya dan di kanannya. Karena keduanya merupakan jalan buntu dan pemain sudah mengambil *treasure*, pemain akan dimasukkan ke dalam mode *backtrack* karena Goal jumlah *treasure* belum tercapai.



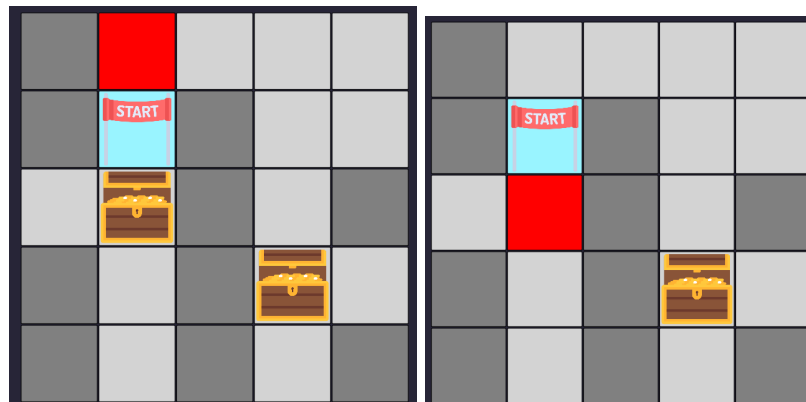
Gambar 3.3.2. Ilustrasi Uji Kasus DFS

Setelah *backtrack*, pencari solusi akan mencari simpul yang belum dikunjungi terdekat. Dalam kasus ini, simpul tersebut adalah simpul Goal. Solusi yang diperoleh adalah rute.



Gambar 3.3.3. Ilustrasi Solusi Uji Kasus DFS

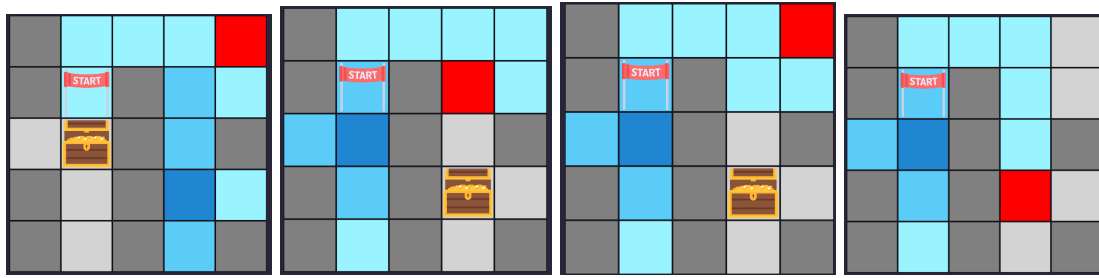
Untuk algoritma BFS, giliran pencarian dilakukan secara bergantian sesuai urutan pembangkitan simpul. Giliran ini sangat berbeda jika dibandingkan algoritma DFS.



Gambar 3.3.4. Ilustrasi Dua Langkah Pertama Uji Kasus BFS

Kedua langkah ini kemudian menjalankan pencarian masing-masing yang tidak saling mempengaruhi informasi simpul yang telah dikunjungi (akibat pemisahan informasi pencatatan

simpul pada setiap *BFSPoint*). Pencarian ini dilakukan oleh masing-masing percabangan rute hingga tercapai solusi.



Gambar 3.3.5. Ilustrasi Empat Langkah Terakhir Uji Kasus BFS

BAB 4

IMPLEMENTASI DAN PENGUJIAN

4.1. Implementasi Algoritma Pencarian Solusi

Berikut merupakan implementasi dari algoritma BFS maupun DFS. Kedua algoritma memiliki prioritas pemilihan rute yaitu Left, Right, Up, lalu Down.

4.1.1. Pseudocode Algoritma BFS (Breadth First Search)

```
IterateBFS {proses utama dari algoritma BFS}
{beberapa kondisi percabangan dan perulangan dapat berubah sesuai toggle yang
aktif (tampak pada implikasi (if..then..else true))}
```

```
procedure IterateBFS(startNode: Point)
{Melakukan Iterasi pada BFS pada maze, dimulai dari sebuah node Start Node}
{I.S. currentNode, treasureCount, routeTaken, backtrackRoute terdefinisi}
{F.S. Dihasilkan sebuah list of string yang berisi path yang ditempuh player}
```

KAMUS LOKAL

```
treasureCount, treasureGain, branchGain, treasureGain: int
nextDirection : char
Neighbors, validNeighbors : List of Point
nextRoute, rRoute : string
nextPoint : Point
```

ALGORITMA

```
mazeStart <- alloc(BFSPoint)
mazeStart.AddSelfAsStep()
searchQueue <- alloc(Queue<BFSPoint>())
searchQueue.Enqueue(mazeStart)
```

```
{cek apakah queue tidak kosong}
{jika TSP belum dijalankan, maka cek Goal belum tercapai}
{jika TSP dijalankan, maka cek TSP belum ditandai selesai}
```

```
while (searchQueue.Any() and
(if(not this._isTspStarted) then not(this._isGoalFinished) else true) and
(if this._isTspStarted then not(this._isTspFinished) else true)
) do
currentNode <- searchQueue.Dequeue();
this.BackupDirectionState(currentNode.Steps);
this.AddExploredNode(currentNode.Point);
if (mazeTile(currentNode.Point) = 'T' and not
currentNode.HasExploredBfsStep(currentNode.Point)) then
currentNode.TreasureCount <-
currentNode.TreasureCount + 1

currentNode.BranchGain <-
currentNode.TreasureCount + 1
```

```
{mark this point as visited}
currentNode.AddSelfAsStep()
if (this._isTspStarted) then
    currentNode.AddSelfAsTspStep()

{ =====GOAL CHECK===== }
{ check for goal treasure count when not in TSP mode }
if (currentNode.TreasureCount =
    this._mazeMap.TreasureCount and not
    this._isTspStarted
) then
    this._isGoalFinished = true
    if (this.IsTspEnabled) then {reset Queue, redo BFS for TSP}
        searchQueue.Clear()
        searchQueue.Enqueue(currentNode)
        this._isTspStarted <- true
        this.DeleteLastState()
        continue
    else
        break
{TSP Goal Check: check if current point is startpoint}
else if (this._isGoalFinished and this._isTspStarted and
currentNode.Point =this._mazeMap.StartPoint) then
    this._isTspFinished <- true
    break

{ BFS ROUTINE: goal (bfs or tsp) is unsatisfied }
{ get valid neighbors }
neighbors <-this._mazeMap.GetNeighbors(currentNode.Point)
validNeighbors <- alloc(List<Point>())
I iterate[0..neighbors.Count]
    if (
        {fungsi pembangkit status}
        (this._mazeMap.IsWalkable(neighbors[i])
        and
        (if(not this._isTspStarted) then
            not(currentNode.HasExploredBfsStep(neighbors[i])) else true)
        and
        (if this._isTspStarted then
            not(currentNode.HasExploredTspStep(neighbors[i])) else true)
        ) then
        validNeighbors.Add(neighbors[i])

I iterate[0..validNeighbors.Count]
    nextDirection <-
    Maze.GetDirectionBetween(currentNode.Point,
    validNeighbors[i])
    if (validNeighbors.Count > 1) then
        nextBranchGain <- 0
    else
        nextBranchGain <- currentNode.BranchGain
```

```

        route <- currentNode.Steps + nextDirection

        n <- alloc(BFSPoint(validNeighbors[i], route,
        currentNode.TreasureCount, currentNode.PointSteps,
        nextBranchGain, 0)
        searchQueue.Enqueue(n)

        {backtrack if no available neighbour}
        if (validNeighbors.Count = 0
and
        (if this.IsBranchPruningEnabled then currentNode.BranchGain != 0 else
        true)
and
        not this._isTspStarted
        ) then
            backStep <- currentNode.NextBackStep
            nextPoint <- Maze.GetNextPoint(currentNode.Point,
            backStep)
            nextSearchPoint <- alloc(BFSPoint(nextPoint,
            currentNode.Steps + backStep,
            currentNode.TreasureCount, currentNode.PointSteps,
            currentNode.BranchGain, currentNode.BacktrackFlag))
            nextSearchPoint.IncrementBacktrackFlag()

            searchQueue.Enqueue(nextSearchPoint)

```

StartSearch {memulai pencarian BFS, tidak memasukkan kode pencatatan waktu}

procedure StartSearch()
 {starts search for solution using BFS}

ALGORITMA
 IterateBFS(this._mazeMap.StartPoint)

4.1.2. Pseudocode Algoritma DFS (Depth First Search)

RecurseDFS {proses rekursif penyelesaian DFS}
 {beberapa kondisi percabangan dan perulangan dapat berubah sesuai toggle yang aktif (tampak pada implikasi (if..then..else true))}

procedure RecurseDFS(currentNode : Point, treasureCount : integer, routeTaken : string, backtrackRoute : string)
 {Prosedur yang akan melakukan pencarian solusi DFS}
 {I.S. currentNode, treasureCount, routeTaken, backtrackRoute terdefinisi}
 {F.S. Dibuat sebuah List of String yang menyimpan path yang dilalui untuk mengumpulkan semua treasure}

KAMUS LOKAL
 treasureCount, treasureGain, branchGain, treasureGain: integer
 nextDirection : char
 Neighbors, validNeighbors : List of Point

Laporan Tugas Besar II

IF2211 Strategi Algoritma

```
nextRoute, rRoute : string
nextPoint : Point

ALGORITMA
  if(
    (if(not this._isTspStarted) then not(this._isGoalFinished) else true) and
    (if this._isTspStarted then not(this._isTspFinished) else true)
  ) then

    this.BackupDirectionState(routeTaken)
    if(this._mazeMap.GetMazeTile(currentNode) = 'T') then
      if(not this.IsNodeExplored(currentNode)) then
        treasureCount <- treasureCount + 1
        treasureGain <- treasureGain + 1
      this.AddExploredNode(currentNode)

    if(this._isTspStarted) then
      this._tspVisits.Add(currentNode)
    if((not this._isGoalFinished) and treasureCount =
      this._mazeMap.TreasureCount and not this._isTspStarted)
    ) then
      this._isGoalFinished <- true
      return

    else if(this._isGoalFinished and this._isTspStarted
      and currentNode = this._mazeMap.StartPoint
    ) then
      this._isTspFinished <- true
      return

    { DFS ROUTINE}
    else
      neighbors <- this._mazeMap.GetNeighbors(currentNode)
      validNeighbors <- alloc(List<Point>)
      I iterate[0..neighbors.Count]
        if(
          this._mazeMap.IsWalkable(neighbors[i])
          and (if(not this._isTspStarted) then not(this.IsNodeExplored(neighbors[i])) else true)
          and (if(this._isTspStarted) then not(this._tspVisits.Contains(neighbors[i])) else true)
        ) then
          validNeighbors.Add(neighbors[i])

      I iterate[0..validNeighbors.Count]
        nextDirection <-
          Maze.GetDirectionBetween(currentNode,
            validNeighbors[i])
        nextRoute <- routeTaken + nextDirection
        if(validNeighbors.Count > 1) then
          branchGain <- 0
        Else
          branchGain <- treasureGain
        ReurseDFS(validNeighbors[i], treasureCount,
          branchGain, nextRoute, "")
```

```

    { no valid neighbor, do backtracking }
    if(
        validNeighbors.Count = 0 and
        (if this.IsBranchPruningEnabled then treasureGain != 0 else true)
        and not(this._isTspStarted)
    ) then
        if(backtrackRoute != "") then
            rRoute <- backtrackRoute
        else
            rRoute <-
                Player.GenerateBacktrackRoute(routeTaken)
        nextPoint <- Maze.GetNextPoint(currentNode,
            rRoute[0])

        if(rRoute.Length > 1) then
            RecurseDFS(nextPoint, treasureCount,
                treasureGain, routeTaken + rRoute[0],
                rRoute.Substring(1))
        else
            RecurseDFS(nextPoint, treasureCount,
                treasureGain, routeTaken + rRoute[0], "")

```

StartSearch {memulai pencarian DFS, tidak memasukkan kode pencatatan waktu}

procedure StartSearch()

{starts search for solution using DFS}

KAMUS LOKAL

route : string
 newStart : Point

ALGORITMA

```

    { DO DFS }
    RecurseDFS(this._mazeMap.StartPoint, 0, 0, "", "")
    if (this.IsTspEnabled) then {redo DFS for TSP}
        { TSP SETUP }
        route <- this.GetStateBackup(this.BackupCount - 1)
        Point newStart <- Maze.GetNextPoint(this._mazeMap.StartPoint, route)
        this._isTspStarted <- true
        DeleteLastState()

        { DO TSP }
        RecurseDFS(newStart, this._mazeMap.TreasureCount, 0, route, "")

```

4.1.3. Pseudocode Flow Program Utama (Direpresentasikan GUI)

Procedure Main()


```
KAMUS LOKAL
    _maze : Maze
    _searchMode : string
    _krustyKrab : Player

ALGORITMA

fileInput <- PromptFilePath()
_searchMode, _tspToggle, _pruningToggle <- PromptSearchMode()
_maze = new Maze("", _fileInput)

If _searchMode == "DFS" then
    krustyKrab = new DFSPlayer(_maze, _tspToggle)
else
    krustyKrab = new BFSPlayer(_maze, _tspToggle)

If _pruningToggle == false then
    krustyKrab.setBranchPruning(_pruningToggle)

krustyKrab.startSearch()
krustyKrab.displaySearch()
```

4.2. Struktur Data dan Spesifikasi Program

4.2.1. Penggunaan Struktur Data pada Program

Karena bekerja dengan menggunakan bahasa pemrograman C# yang berparadigma *object oriented*, kami menyusun kelas objek-objek bersesuaian yang berguna untuk menyelesaikan masalah pencarian rute ini. Kelas objek yang dibuat di antaranya adalah kelas *Maze*, *Player*, *BFSPlayer*, *DFSPlayer*, dan *GameException*. *Maze* merepresentasikan peta *maze* dan bertugas dalam pembacaan dan validasi *file* peta. Selanjutnya, kelas abstrak *Player* bertugas untuk menyimpan informasi dari solusi, pencarian rute, state, dan konfigurasi mode pencarian. Kelas ini diturunkan oleh kelas *BFSPlayer* dan kelas *DFSPlayer*, yang masing-masing memiliki tugas untuk melakukan pencarian rute secara BFS dan DFS, serta melakukan TSP. Terdapat pula kelas *GameException* yang bertugas untuk *exception handling* pembacaan peta dari program.

Selanjutnya adalah bagian GUI dari program. Implementasi dari GUI menggunakan *Windows Presentation Foundation* (WPF). Pada bagian GUI terdapat kelas *Element*. Kelas ini digunakan untuk merepresentasikan elemen pewarnaan visualisasi *maze* dan solusi.

Secara singkat, berikut merupakan beberapa atribut struktur data yang dimanfaatkan oleh objek-objek program.

```

/ ** State variables, untuk menentukan mode program dan juga
informasi yang akan ditampilkan pada program */

```

```

    int _stateViewIndex = 1;
    public string _searchMode = "DFS";
    bool _tspToggle = false;
    bool _pruningToggle = false;
    string _totalTime;
    int _nGridRows = 1;
    int _nGridCols = 2;
    int _autoSpeed = 0;
    int SPEED_STATE = 9;
    public string _configFileName = "";

```

```

/* Struktur Data Elemen GUI */

```

```

    string _color;
    char _type;

```

```

/** Information variables state yang disimpan pada program
utama untuk memecahkan masalah*/

```

```

    Maze? _maze;
    List<Element>? _board;
    List<List<Element>> _states;
    List<string> _steps;
    List<string> _numNodes;
    List<string> _numSteps;

```

```

/* Properties, State, dan Config Fields yang disimpan pada
objek Player*/

```

```

    List<Point> _exploredNodes;
    Maze _mazeMap;
    List<string> _playerDirectionState;
    List<string> _numSteps;
    List<string> _numNodes;
    long _recordedSearchTime;
    bool _isGoalFinished;
    bool _isTspStarted;
    bool _isTspFinished;

    bool _branchPruningEnabled;
    bool _tspEnabled;

```

```
/* Logger-Utility Fields yang digunakan untuk komunikasi antara
GUI dan pemecahan masalah (Player)*/
/* sisi Player */
string _filename;
List<int[,]> _mazeStateLog;
string _solutionRoute;
int[,] _mazeStateTemplate;

/* sisi GUI */
List<List<Element>> _logBoard;
string mapConfig;
```

```
/* Struktur Data pada DFS */
List<Point> _tspVisits;
callstack
```

```
/* Struktur Data pada BFS */
BFSPoint {
    Point _currentPosition;
    string _movementSteps;
    int _treasureCount;
    int _branchTreasureGain;
    int _backtrackFlag;
    List<Point> _pointSteps;
    List<Point> _tspSteps;
}

Queue<BFSPoint> searchQueue;
```

```
/* Struktur data graf maze */
List<List<char>> _mapMatrix;
int _nRows;
int _nCols;
Point _mazeStart;
List<Point> _treasureSet;
```

```
/* Struktur data lainnya */
Exception
CallStack
```

4.2.2. Spesifikasi Program

Program yang diimplementasikan menggunakan bahasa C# dan memiliki GUI berbasis WPF. Program mengadaptasi algoritma *Breadth First Search (BFS)* dan *Depth First Search (DFS)*, untuk menyelesaikan persoalan graf dinamis berupa *maze*. Selain itu, permasalahan ini

menggunakan *maze* khusus yang memiliki beberapa titik yang perlu dikunjungi dalam solusi akhir. Pada program ini, pembangkitan status menggunakan urutan prioritas kiri, kanan, atas, dan bawah (LRUD).

Konfigurasi peta *maze* konfigurasi dilakukan melalui GUI program. Pada program, pengguna dapat memasukkan *path* menuju file teks yang memuat konfigurasi, baik menggunakan *relative path* atau *absolute path*. Pengguna juga dapat memasukkan *file* melalui *File Explorer*. Program kemudian akan melakukan validasi terhadap konfigurasi peta. Beberapa syarat yang harus dipenuhi peta adalah sebagai berikut.

- Simbol yang dapat digunakan adalah sebagai berikut
 - K : Krusty Krab (Start Point), hanya satu per peta
 - T : *Treasure*, minimal satu buah pada peta konfigurasi, dan setiap *treasure* dapat diakses dari titik awal
 - R : Lintasan
 - X : Tembok
- Karakter pada konfigurasi peta harus disusun sebagai matriks, dengan setiap karakter dipisahkan oleh spasi, dan setiap baris dipisahkan oleh *newline*. Matriks juga harus bersifat lengkap.

Apabila valid, *file* txt yang dimasukkan pengguna dapat divisualisasikan pada GUI. Setelah itu, pengguna dapat melakukan pencarian menggunakan mode yang disediakan. Pilihan mode yang disediakan adalah sebagai berikut.

- Toggle DFS/BFS: memilih algoritma pencarian yang digunakan.
- Toggle TSP: memastikan bahwa rute solusi mendapatkan semua *treasure* dilanjutkan dengan rute kembali ke titik asal.
- Toggle Pruning: mengubah mekanisme pruning yang terjadi apabila pemain memasuki jalan buntu tanpa mendapatkan *treasure* baru.

Setelah menjalankan pencarian solusi, GUI akan menampilkan banyaknya node (grid) yang diperiksa, banyaknya langkah, rute solusinya, dan waktu eksekusi algoritma. Untuk melihat tahapan pencarian, pengguna dapat memainkan animasi tahapan atau menggunakan tombol untuk melihat masing-masing state. Mekanisme komunikasi antara pencari solusi dan GUI adalah dengan menggunakan *log*. Pada setiap pencarian, hasil dari eksekusi algoritma dituliskan ke dalam sebuah *file log*, yang akan dibaca oleh GUI yang dibuat dengan menggunakan *Windows Presentation Foundation* (WPF). Pada *log file* tersebut, tertulis angka angka yang merepresentasikan banyak sebuah grid telah dikunjungi. Setiap titik akan diproses menjadi *Element* yang dimasukkan ke dalam list of Element. List of element tersebut akan dijadikan sebagai sumber dari *data binding* dari sebuah uniform grid (matriks) yang terdapat di GUI.

4.3. Tata Cara Penggunaan Program

Untuk menjalankan program ini, pengguna harus menggunakan sistem operasi *Windows* dan memasang *dotnet* dan *MSBuild*. Setelah itu, pengguna harus melakukan clone pada repository dengan melakukan perintah:

1. `git clone https://github.com/williamnixon20/Tubes2_zainali.git` pada *command line*.

Berikutnya, pengguna dapat melakukan *build* dengan menjalankan berikut pada *root* hasil clone

2. `msbuild src/Tubes2_zainali.csproj /p:OutputPath=..\bin\ /p:Configuration=Release`

dan kemudian mengeksekusi *build* tersebut dari bin dengan perintah

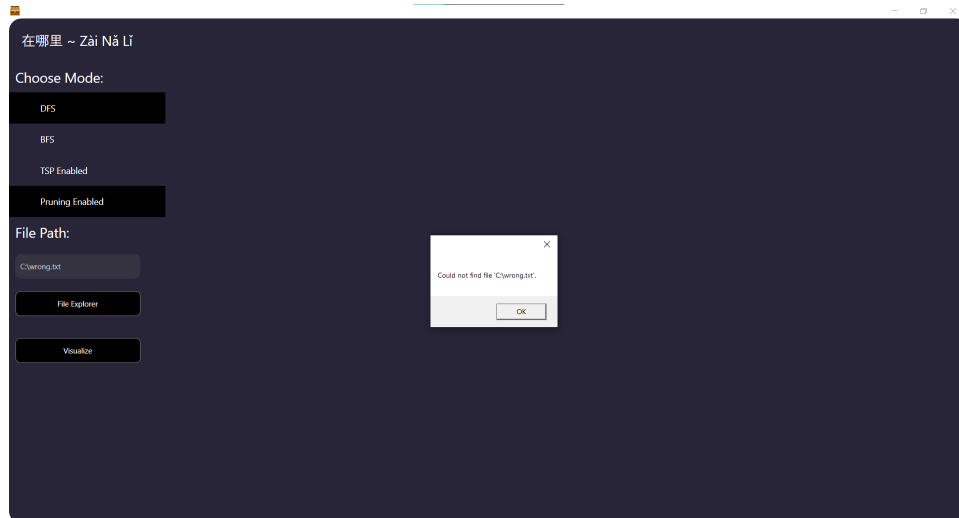
3. `..\bin\Tubes2_zainali.exe`

Berikut ini merupakan tampilan awal dari program saat baru saja dijalankan



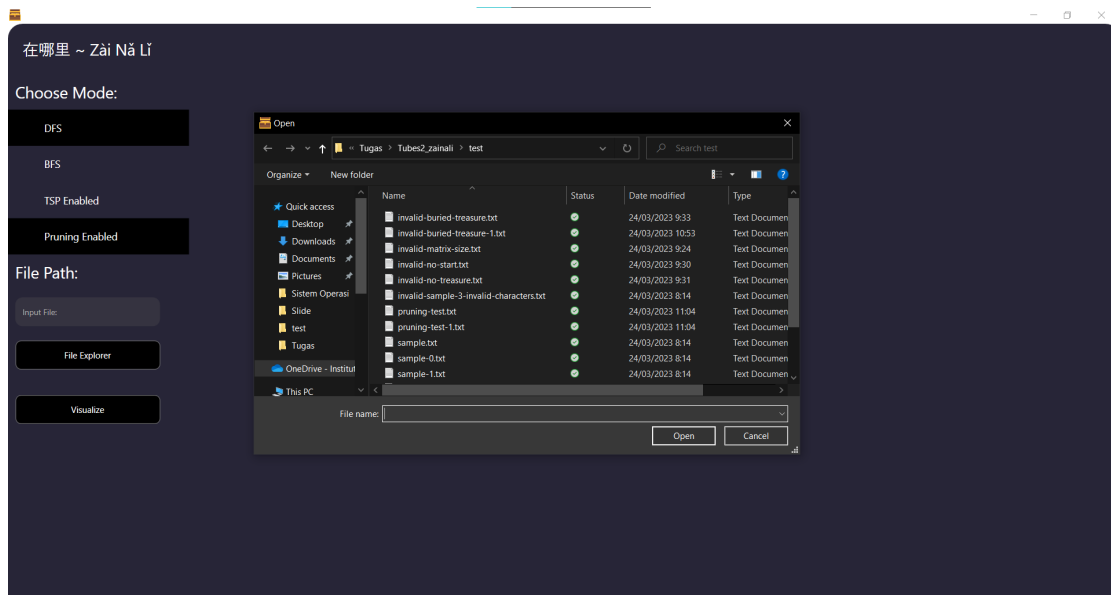
Gambar 4.3.1. Tampilan Awal Program

Untuk memasukkan peta *maze*, pengguna dapat langsung mengetikkan nama *file* yang berada di folder *test* pada *textfield* menggunakan penulisan *path* relatif terhadap *root*. Pengguna dapat pula menekan tombol *File Explorer* untuk langsung memilih *file maze* yang ingin dimasukkan dari *directory* kita. Setelah memilih *file maze* yang akan digunakan, pengguna dapat menekan tombol *Visualize* untuk memvisualisasikan *grid* dari *maze file* yang telah dimasukkan. Apabila nama *file* yang dimasukkan pada *textfield* ternyata tidak tersedia pada folder *test*, akan muncul pesan kesalahan yang menyatakan *file* tersebut tidak ada.



Gambar 4.3.3 Tampilan Program saat Memunculkan Peringatan File yang Tidak Tersedia

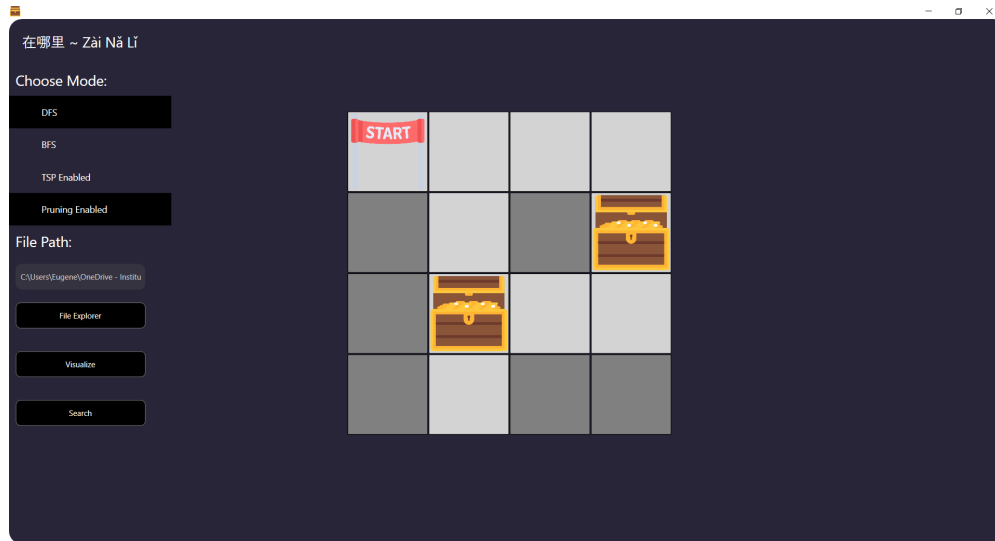
Pengguna juga harus memastikan bahwa isi *file* yang dipilih menggunakan karakter yang tepat. Selain itu, syarat peta yang dianggap valid adalah: memiliki hanya satu titik awal, memiliki setidaknya satu *treasure*, dan setiap *treasure* dapat diakses dari titik awal dalam beberapa langkah.



Gambar 4.3.2. Tampilan Program saat Memasukkan Input dari File Explorer

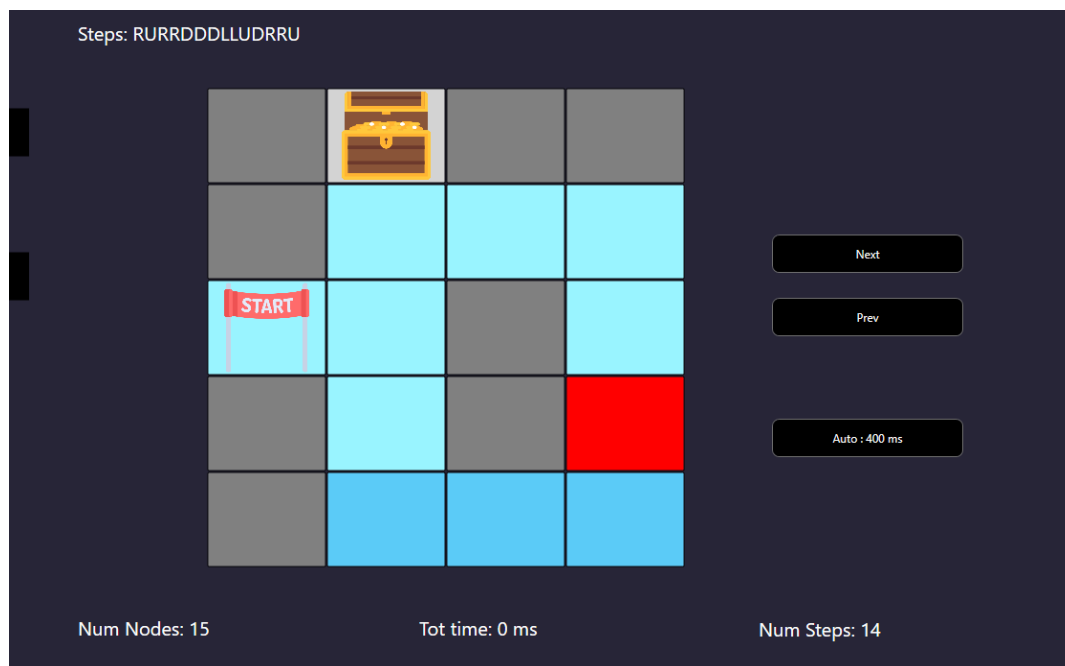
Laporan Tugas Besar II

IF2211 Strategi Algoritma



Gambar 4.3.3. Tampilan Program saat Memvisualisasikan *Maze Grid*

Pengguna dapat memilih mode penyelesaian solusi yang diinginkan, yaitu BFS maupun DFS. Apabila pengguna menginginkan agar rute solusi yang dihasilkan dapat kembali lagi ke posisi awal (start), pengguna dapat mengaktifkan mode TSP selain memilih tipe algoritma BFS ataupun DFS. Setelah visualisasi, akan muncul tombol search yang memungkinkan pengguna untuk memulai pencarian dengan setting mode yang telah dipilih sebelumnya.



Gambar 4.3.4. Tampilan GUI saat memvisualisasikan hasil pencarian

Setelah proses pencarian selesai, maka pengguna dapat memulai melakukan visualisasi menggunakan tombol-tombol yang muncul. Selain itu, juga akan ditampilkan informasi tambahan mengenai status dari pencarian rute seperti steps, banyak node, waktu eksekusi, dan banyaknya steps. Tombol *Next* dan *Prev* dapat digunakan pengguna untuk mengubah transisi ke state berikut ataupun sebelumnya. Tombol *Auto* dapat digunakan untuk melakukan transisi setiap interval waktu tertentu, yang dapat ditekan kembali untuk mengubah interval tiap state menjadi bervariasi.

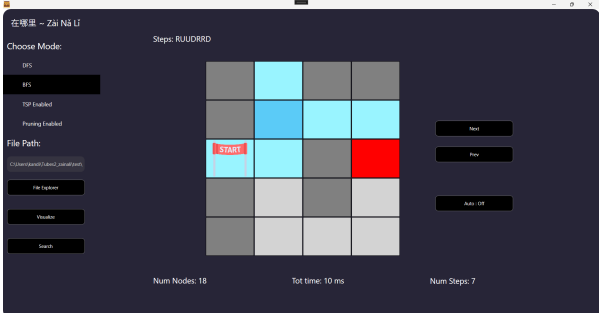
4.4. Hasil Pengujian dan Analisis

4.4.1. Pengujian untuk *test case sample-1.txt*

- a. Isi sample-1.txt

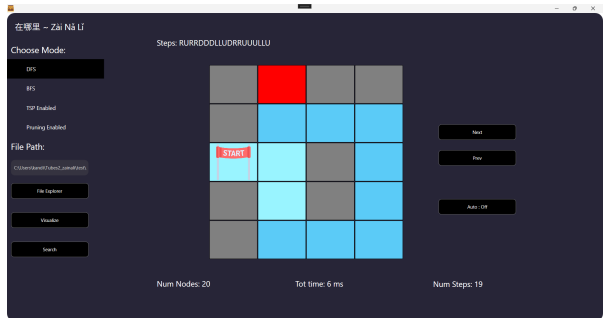
```
X T X X
X R R T
K R X T
X R X R
X R R R
```

- b. Rute solusi dengan algoritma BFS

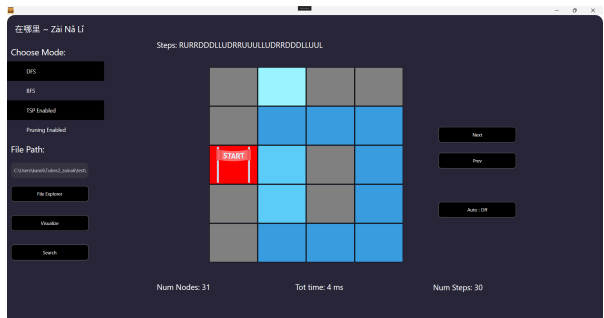
Gambar	Keterangan
	<p>Algoritma berhasil mendapatkan jalur terpendek secara cukup efektif</p>

Laporan Tugas Besar II
IF2211 Strategi Algoritma

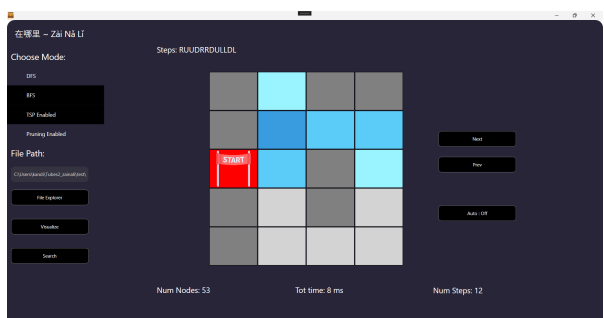
c. Rute solusi dengan algoritma DFS

Gambar	Keterangan
	<p>Algoritma kurang efektif karena harus melakukan backtracking, DFS melakukan pencarian ke kanan terlebih dahulu.</p>

d. Rute solusi dengan algoritma DFS (TSP)

Gambar	Keterangan
	<p>Algoritma kurang efektif karena harus melakukan backtracking, DFS melakukan pencarian ke kanan terlebih dahulu.</p>

e. Rute solusi dengan algoritma BFS (TSP)

Gambar	Keterangan
	<p>Algoritma berhasil mendapatkan jalur terpendek secara cukup efektif</p>

4.4.2. Pengujian untuk *test case sample-2.txt*

a. Isi sample-2.txt

```

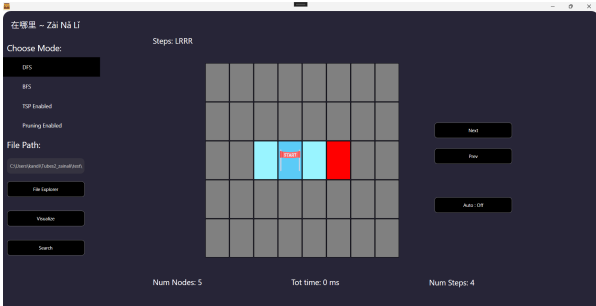
X X X X X X X X
X X X X X X X X
X X T K R T X X
X X X X X X X X
X X X X X X X X

```

b. Rute solusi dengan algoritma BFS

Gambar	Keterangan
	<p>Algoritma berhasil mendapat rute dengan cukup efektif.</p>

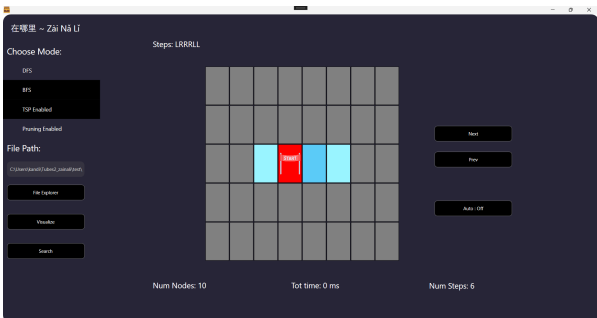
c. Rute solusi dengan algoritma DFS

Gambar	Keterangan
	<p>Algoritma berhasil mendapat rute dengan cukup efektif.</p>

d. Rute solusi dengan algoritma DFS (TSP)

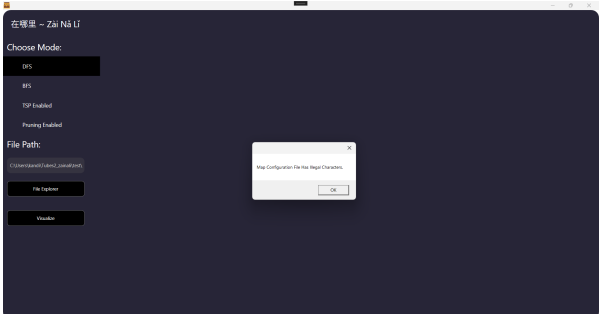
Gambar	Keterangan
	<p>Algoritma berhasil mendapat rute dengan cukup efektif.</p>

e. Rute solusi dengan algoritma BFS (TSP)

Gambar	Keterangan
	<p>Algoritma berhasil mendapat rute dengan cukup efektif.</p>

4.4.3. Pengujian untuk *test case sample-3.txt*

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

Gambar	Keterangan
	File text memiliki karakter invalid

4.4.4. Pengujian untuk *test case sample-4.txt*

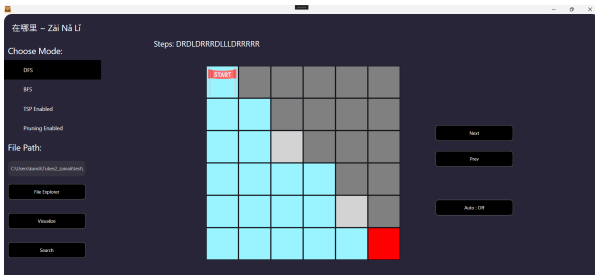
a. Isi sample-4.txt

```
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
```

b. Rute solusi dengan algoritma BFS

Gambar	Keterangan
	Algoritma berhasil mendapat rute dengan cukup efektif.

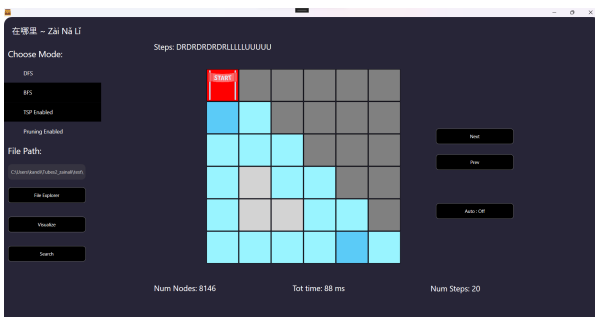
c. Rute solusi dengan algoritma DFS

Gambar	Keterangan
	<p>Algoritma kurang efektif karena memiliki prioritas mengeksplorasi kiri dan kanan terlebih dahulu.</p>

d. Rute solusi dengan algoritma DFS (TSP)

<h2>Gambar</h2>	<h2>Keterangan</h2>
	<p>Walaupun rute kurang efisien, algoritma cukup efektif di dalam waktu eksekusi di banding dengan BFS.</p>

e. Rute solusi dengan algoritma BFS (TSP)


Gambar	Keterangan
 <p>The screenshot shows a web-based application for a 3x3 grid search. The grid has a red start cell at (0,0) and a blue goal cell at (2,2). The path taken is indicated by light blue cells. The interface includes a 'Choose Mode' dropdown menu with 'DFS' selected, a 'Steps' counter showing '0R0R0R0R0RLLLLLLLLUUUUU', and a 'File Path' input field. At the bottom, statistics are displayed: 'Num Nodes: 8146', 'Tot time: 88 ms', and 'Num Steps: 20'.</p>	<p>Walaupun rute efisien, algoritma membutuhkan waktu yang cukup lama karena mengeksplor banyak sekali state.</p>

4.4.5. Pengujian untuk *test case sample-5.txt*


a. Isi sample-5.txt

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


b. Rute solusi dengan algoritma BFS

Gambar	Keterangan
	Algoritma cukup efektif.


c. Rute solusi dengan algoritma DFS

Gambar	Keterangan
 <p>Steps: RRDLDRDLDRDLDRDLDRDL</p> <p>Num Nodes: 22 Tot time: 5 ms Num Steps: 21</p>	<p>Algoritma kurang efektif karena memiliki prioritas mengeksplorasi node kiri kanan terlebih dahulu.</p>

d. Rute solusi dengan algoritma DFS (TSP)

Gambar	Keterangan
	<p>Algoritma cukup efektif, walaupun rute tidak optimal akan tetapi dapat diselesaikan dengan mengunjungi jumlah node yang cukup sedikit.</p>

e. Rute solusi dengan algoritma BFS (TSP)

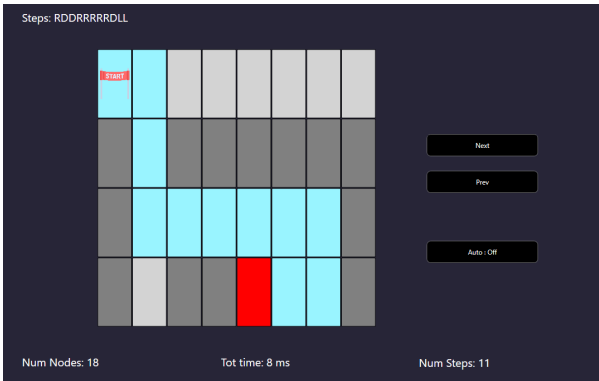
Gambar	Keterangan
	Algoritma berjalan dengan sangat lambat karena ada terlalu banyak pembangkitan rute yang dilakukan. Algoritma BFS dengan TSP tidak cocok untuk diterapkan pada kasus ini.

4.4.6. Pengujian Pruning

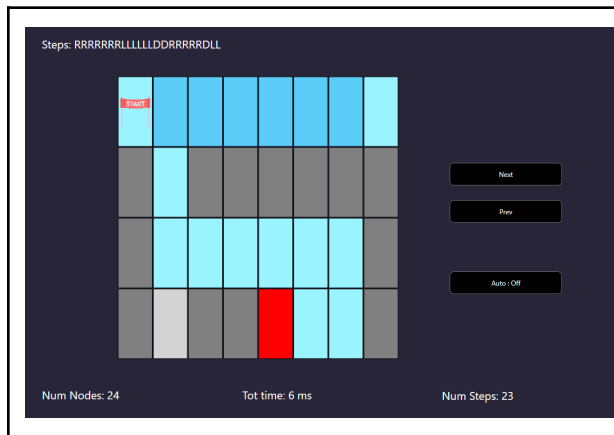
a. Konfigurasi

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

b. DFS dengan Pruning

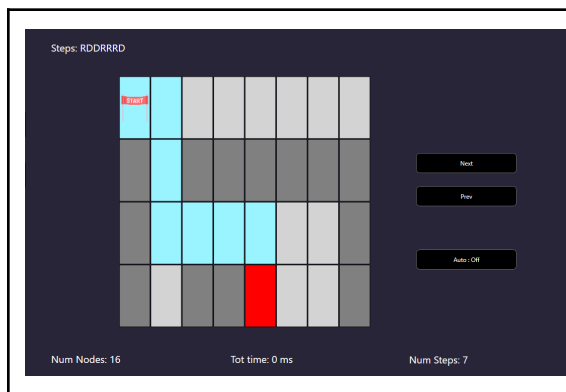
	Dapat dilihat bahwa dengan pruning, rute yang melakukan backtrack terlalu jauh akan dieliminasi. Hal ini meminimalisir angka step dan jumlah node yang dikunjungi di dalam program.
---	---

c. DFS tanpa Pruning



Tanpa pruning, tidak akan dilakukan apapun pada rute yang melakukan backtracking. Hal ini membuat step di dalam program menjadi kurang efisien.

d. BFS tanpa Pruning



Tanpa pruning, tidak akan dilakukan apapun pada rute yang melakukan backtracking. Hal ini membuat step di dalam program menjadi kurang efisien.

e. BFS dengan Pruning



Dapat dilihat bahwa dengan pruning, rute yang melakukan backtrack terlalu jauh akan dieliminasi. Hal ini meminimalisir angka step dan jumlah node yang dikunjungi di dalam program.

4.4.7. Pengujian Kasus Lain

a. Konfigurasi



b. Hasil

Hasil pengujian secara singkat dapat dilihat pada bagian 3.1.

4.5. Analisis Desain Solusi Algoritma BFS dan DFS

Melalui pengamatan dari rute hasil solusi algoritma BFS maupun DFS yang diterapkan, kami mengamati beberapa kelebihan dan juga kekurangan dari algoritma BFS maupun DFS hasil implementasi. Secara general, pada peta yang padat dan kecil, algoritma BFS berkemungkinan memiliki waktu eksekusi lebih baik. Akan tetapi pada peta yang besar, algoritma DFS berpotensi memiliki waktu eksekusi yang lebih cepat.

Algoritma BFS hasil implementasi memiliki beberapa kelebihan, yaitu dapat menemukan rute terpendek antara dua titik pada peta, karena BFS mencari rute yang paling dekat terlebih dahulu sebelum mencari rute yang lebih jauh. Selain itu, algoritma BFS juga memastikan pencarian dilakukan secara sistematis dengan mengeksplorasi seluruh simpul pada level yang sama terlebih dahulu sebelum melakukan pencarian ke level selanjutnya. Hal ini menyebabkan algoritma BFS untuk memiliki performa yang cukup baik pada peta yang kecil, karena tidak harus sering melakukan backtracking dalam eksplorasi simpul. BFS baik dalam mengeksplorasi map yang kecil dan node yang sedikit, dengan treasure yang terletak dekat dari titik asal.

Walaupun demikian, algoritma BFS memiliki beberapa kekurangan. Kekurangan tersebut adalah diperlukannya memori yang lebih banyak dibandingkan dengan DFS karena perlu menyimpan seluruh simpul yang dikunjungi di setiap level. Selain itu, pada kasus di mana peta sangat besar dan harta karun berada di ujung map (jauh dari titik asal), BFS mungkin tidak efektif karena membutuhkan waktu yang lama untuk menjelajahi seluruh simpul.

Algoritma DFS memiliki kelebihan utama dalam menjelajahi peta yang memiliki banyak simpul akan tetapi hanya memiliki kedalaman yang sedikit. Hal ini dilakukan untuk meminimalisir proses backtracking yang dilakukan algoritma. Algoritma ini juga memerlukan memori yang lebih sedikit dibandingkan BFS karena hanya menyimpan simpul terakhir yang dikunjungi. Algoritma ini kami rasa cocok untuk menjelajahi map besar dengan banyak node dengan harta karun yang berada jauh dari titik asal.

Kekurangan algoritma DFS adalah tidak dapat menjamin menemukan rute terpendek karena mungkin melakukan proses backtracking beberapa kali. Pencarian jalur tidak sistematis sehingga terkadang dapat melewati simpul yang seharusnya dijelajahi. Selain itu, algoritma ini juga mudah terjebak dalam lingkaran pada peta yang memiliki loop, sehingga dapat menghabiskan waktu dan memori yang tidak efisien.

Secara umum, ketika pencarian peta diharapkan untuk menghasilkan rute yang terpendek, kami merekomendasikan untuk menggunakan algoritma BFS. Akan tetapi jika syarat tersebut tidak diperlukan dan waktu eksekusi lebih penting, kami merekomendasikan algoritma BFS untuk peta yang kecil dan padat, dan algoritma DFS untuk peta yang relatif lebih besar. Ketika ingin menggunakan algoritma TSP, kami merekomendasikan menggunakan DFS karena BFS relatif memiliki waktu eksekusi yang cukup lama.

BAB 5

KESIMPULAN DAN SARAN

5.1. Kesimpulan

Dalam tugas besar ini, kami berhasil membuat sebuah aplikasi dengan GUI sederhana yang dapat mengimplementasikan algoritma BFS dan DFS untuk mencari rute terpendek untuk mendapatkan seluruh harta karun pada peta masukan. Kami juga berhasil mengimplementasikan berbagai bonus seperti visualisasi progres pencarian di dalam algoritma pencarian tersebut, dan juga mengimplementasikan fitur TSP sederhana.

Analisis dari kedua algoritma tersebut menunjukkan bahwa algoritma BFS memiliki beberapa kelebihan seperti menemukan rute terpendek antara titik-titik, dan juga baik di dalam mengeksplorasi peta yang relatif padat. Akan tetapi, algoritma BFS memerlukan waktu dan juga memori yang cukup besar ketika mengeksplorasi peta yang memiliki peta yang besar. Algoritma DFS memiliki beberapa kelebihan yaitu memerlukan lebih sedikit memori daripada BFS, dan juga relatif lebih cepat di dalam mengeksplorasi peta yang besar. Akan tetapi, DFS tidak dapat menjamin memberikan rute yang terpendek dan memiliki potensi untuk terjebak di dalam loop ketika mengunjungi kembali titik yang sudah dikunjungi.

5.2. Saran

Tugas ini dapat dikembangkan kembali dengan menerapkan berbagai algoritma pencarian lanjutan yang melibatkan heuristik, seperti pencarian A*, atau yang menghasilkan solusi lebih optimal. Fitur *“Draw your own map”* lewat GUI juga mungkin dapat dikembangkan kepada pembaca yang tertarik untuk mengembangkan lebih lanjut aplikasi ini.

5.3. Refleksi

Tugas besar ini memberikan pengalaman berharga bagi kami untuk mengembangkan aplikasi dengan paradigma OOP dan juga memperkenalkan kami pada bahasa pemrograman baru, yaitu C#. Tugas besar ini juga memperkuat pengetahuan kami atas berbagai algoritma pencarian dan memperkaya pengetahuan kami dalam pemrograman. Kami juga belajar beberapa prinsip pemrograman *native* aplikasi GUI berbasis *Windows*, dan diperkenalkan kepada berbagai konsep di dalam WPF seperti Code Behind, Data Binding, XAML, dan lain sebagainya.

5.4. Tanggapan Terkait Tugas Besar

“Tugas besar ini mengajarkan saya cara mencari ruang kelas di GKUB 👍” - Eugene

“Makasih karena tidak menaruh deadline di tanggal merah 😊” - William

“Berkat tubes ini, saya bisa merasakan rasanya jadi dora :)” - Kandida

DAFTAR PUSTAKA

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

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

LAMPIRAN

Link repository : [williamnixon20/Tubes2_zainali \(github.com\)](https://github.com/williamnixon20/Tubes2_zainali)

Video : <https://youtu.be/G--TitTFK8U>