LAPORAN

PRAKTIKUM KECERDASAN BUATAN

Oleh :

Nama	**:**	**Rizqillah**
		Sharhan Anhar
		Mesti
Kelas	**:**	**TI 2C**
Dosen Pembimbing	**:**	**Muhammad Arhami, S.Si, M.Kom**

PROGRAM STUDI TEKNIK INFORMATIKA

JURUSAN TEKNOLOGI INFORMASI DAN KOMPUTER

POLITEKNIK NEGERI LHOKSEUMAWE

2021

LEMBAR PENGESAHAN

No. Praktikum	:	01/P.KB/2C/TI/2021
Judul Praktikum	:	BFS dan DFS
Tanggal Praktikum	:	22 Maret 2021
Tanggal Pengumpulan Laporan	:	29 Maret 2021
Nama Praktikan / NIM	:	▪ Rizqillah / 1957301020 ▪ Sharhan Anhar / 1957301071 ▪ Mesti / 1957301014
Kelas	:	TI 2C
Nilai	:	

Buketrata, 29 Maret 2021

Dosen Pengampu,

Muhammad Arhami, S.Si, M.Kom
NIP. 19741029 200003 1 001

DAFTAR ISI

DAFTAR GAMBAR

BAB I
PENDAHULUAN

1.1 Latar Belakang

Dalam beberapa tahun terakhir *Artificial Intelligence* (AI) atau kecerdasan buatan telah menjadi sesuatu yang berpengaruh dalam industri *game application*. Hampir semua jenis dan tipe *game application* sekarang membutuhkan AI untuk membuat komputer seolah-olah tampak cerdas. Program pertama yang dibuat AI adalah *game board playing*. Sejarah teori *game application* dimulai dari tahun 1950 ketika komputer mulai dapat diprogram. *Game application board* pertama yang menggunakan AI adalah catur. Pencetus teori game dalam AI adalah Konard Zuse yang merupakan penemu pertama komputer yang dapat diprogram dengan bahasa pemrograman pertama, Claude Shannon yang merupakan penemu teori informasi dan Norbert Wiener yang merupakan pencipta teori kontrol modern. Sejak saat itu mulai ada kemajuan dalam standar bermain game terutama game yang melibatkan papan main, sampai-sampai komputer dapat mengalahkan manusia dalam permainan papan seperti catur, reversi (othello), dan game aplikasi yang bersifat versus lainnya.

Game playing pada komputer juga saat ini banyak digunakan oleh beberapa kalangan sebagai media untuk melatih daya pikir dan strategi mereka dalam mengalahkan lawan mainnya. Sebagaimana diketahui bahwa kunci dari kecerdasan 2 lawan main pada suatu *game playing* adalah kedalaman memperhitungkan kemungkinan-kemungkinan langkah yang ada pada suatu permainan dan setelah itu akan dapat menentukan langkah mana yang paling menguntungkan agar kemenangan dapat diperoleh. Hal tersebut juga menjadi pembeda dari *players* yang memainkan *game board* tersebut.

Bagi komputer perbedaan perhitungan kemungkinan langkah tersebut dibedakan dengan algoritma-algoritma yang telah dikembangkan. Hingga saat ini sudah banyak metode dan algoritma yang dikembangkan untuk permasalahan ini khususnya dalam *Artificial Intelligence* umumnya. Salah satu contoh algoritma yang akan kami bahas disini yaitu beberapa metode pencarian yang digunakan pada *searching* berupa metode *Breadth-First Search* (BFS) dan metode *Depth-First Search* (DFS). Perbedaan metode tersebut yaitu penelusuran yang dilakukan pada setiap node yang terbentuk dalam bentuk tree. *Breadth-First Search* (BFS) menelusuri berdasarkan lebar dari suatu tree, sedangkan *Depth-First Search* (DFS) menelusuri berdasarkan kedalaman suatu *tree*.

1.2 Tujuan Praktikum

Adapun tujuan dari penelitian ini di antaranya yaitu :

1. Memahami penggunaan metode search berupa BFS dan DFS
2. Bisa menggunakan metode search BFS atau DFS didalam program sehari-hari
3. Mengenal lebih jauh mengenai teknik *searching* BFS dan DFS
4. Dapat mengimplementasikan penggunaan metode search tersebut dengan baik
5. Mempelajari lebih lanjut mengenai metode-metode *searching* lainnya.
6. Mengetahui perbedaan metode search BFS dan DFS

1.3 Manfaat Praktikum

Adapun manfaat dari melakukan praktikum mengenai metode *searching* DFS(*Depth-First Search*) dan BFS(*Breatdh-First Search*) adalah sebagai berikut :

1. Sebagai dasar bagi penggunaan metode-metode pencarian dalam suatu aplikasi.
2. Membandingkan kelebihan dan kekurangan metode BFS dan DFS.
3. Meningkatkan daya kreatifitas khususnya dalam kepekaan berfikir untuk menganalisa kearah yang lebih baik.
4. Mengasah daya tangkap visual dengan indikasi mengasah kemampuan dalam melakukan *searching*.
5. Mengetahui kebutuhan akan kecerdasan buatan pada setiap program atau aplikasi komputer pada umumnya dan aplikasi game playing pada khususnya.

BAB II
TINJAUAN PUSTAKA

2.1 Breadth-First Search (BFS)

Strategi pencarian grafik tanpa informasi di mana setiap tingkat dicari sebelum pergi ke tingkat yang lebih dalam berikutnya. Strategi ini menjamin untuk menemukan jalur terpendek ke node yang dicari terlebih dahulu. setiap jalur ke solusi yang panjangnya n akan ditemukan saat pencarian mencapai kedalaman n, dan ini dijamin sebelum setiap node dengan kedalaman <n dicari.

Algoritma BFS (Breadth First Search) adalah salah satu algoritma yang digunakan untuk pencarian jalur. Algoritma ini adalah salah satu algoritma pencarian jalur sederhana, dimana pencarian dimulai dari titik awal, kemudian dilanjutkan ke semua cabang titik tersebut secara terurut. Jika titik tujuan belum ditemukan, maka perhitungan akan diulang lagi ke masing-masing titik cabang dari masing-masing titik, sampai titik tujuan tersebut ditemukan.

BFS juga adalah algoritma yang melakukan pencarian secara melebar yang mengunjungi simpul secara *preorder* yaitu mengunjungi suatu simpul kemudian mengunjungi semua simpul yang bertetangga dengan simpul tersebut terlebih dahulu. Selanjutnya, simpul yang belum dikunjungi dan bertetangga dengan simpul-simpul yang tadi dikunjungi, demikian seterusnya.

Jika graf berbentuk pohon berakar, maka semua simpul pada aras *d* dikunjungi lebih dahulu sebelum simpul-simpul pada aras *d*+1. Algoritma ini memerlukan sebuah antrian *q* untuk menyimpan simpul yang telah dikunjungi. Simpul-simpul ini diperlukan sebagai acuan untuk mengunjungi simpul-simpul yang bertetanggaan dengannya. Tiap simpul yang telah dikunjungi masuk ke dalam antrian hanya satu kali. Algoritma ini juga membutuhkan table Boolean untuk menyimpan simpul yang te lah dikunjungi sehingga tidak ada simpul yang dikunjungi lebih dari satu kali.

Permainan penyusunan blok bisa juga dikatakan sebagai Blocks World Architecture, yaitu sebuah permainan yang mengajak pemainnya untuk berfikir secara logika dalam menyelesaikan permainan ini, misalnya pemain diharuskan untuk menyusun blok angka atau huruf sampai terurut berdasarkan susunan yang telah diacak sebelumnya. Permainan ini bisa dikategorikan sebagai permainan puzzle, yang mana pada keadaan awal blok huruf disusun secara acak untuk beberapa huruf misalnya 6 huruf (A,F,C,E,B,D) pada tiga tempat, kemudian blok huruf tersebut harus disusun kembali dengan menggunakan tiga tempat yang ada

sampai didapatkan susunan blok huruf yang tersusun dari huruf A sampai F pada sebuah tempat. Untuk menyelesaikan masalah penyusunan blok diperlukan suatu metode atau algoritma pencarian solusi. Salah satunya algoritma pencarian yang dapat digunakan adalah algoritma Breadth First Search, karena algoritma ini dapat melakukan pencarian solusi dari keadaan awal yang disediakan berdasarkan parameter-parameter yang ditentukan dari masalah penyusunan blok.

Algoritma BFS juga merupakan salah satu algoritma pencarian yang menguji tiap link pada sebuah halaman sebelum memproses kehalaman berikutnya. Jadi, algoritma ini menelusuri tiap link pada halaman pertama dan kemudian menelusuri tiap link pada halaman kedua pada link pertama dan begitu seterusnya sampai tiap level pada link telah dikunjungi.

2.2 Depth-First Search (DFS)

Depth-First Search (DFS) adalah metode untuk menjelajahi pohon data atau grafik. Dalam DFS, pencarian akan masuk sedalam mungkin ke satu jalur kemudian kembali dan mencoba masuk ke jalur yang lain. Metode DFS seperti berjalan melalui labirin dengan menggunakan jagung sebagai penanda jalan kembali.

Algoritma DFS (Depth First Search) adalah salah satu algoritma yang digunakan untuk pencarian jalur. Algoritma ini mirip dengan Algoritma BFS (Breadth First Search) yang sudah dijelaskan sebelumnya. Jika Algoritma BFS (Breadth First Search) melakukan perhitungan secara terurut dari urutan pertama sampai urutan terakhir, maka algoritma ini melakukan kebalikannya, yaitu melakukan perhitungan secara terurut dari urutan terakhir. Setelah menghabiskan semua kemungkinan dari titik terakhir, barulah mundur ke titik-titik sebelumnya sampai pada titik pertama.

Algoritma DFS, pencarian dalam algoritma ini dimulai dari node yang paling kiri sampai node yang ada pada setiap level selesai dilalui. Jika pada level yang paling dalam solusi yang diharapkan belum diperoleh, maka pencarian akan dilanjutkan pada node yang berada disebelah kanan node awal. Node yang berada di kiri dapat dihapus dari memori agar memori yang digunakan menjadi lebih sedikit. Jika pada level yang paling dalam belum juga ditemukan solusi dari pencarian, maka pencarian akan dilanjutkan pada level sebelumnya. Demikian seterusnya proses akan terus berlangsung sampai ditemukannya solusi. Jika pada level dan node dari algoritma pencarian ini menemukan solusi, maka proses backtracking (penelusuran untuk mendapatkan jalur yang diinginkan) tidak diperlukan.

Penerapan algoritma depth first search pada sistem pencarian dokumen sangat dibutuhkan oleh pihak-pihak terkait yang membutuhkan data di dalam dokumen tersebut. Pencarian kembali terhadap data di dalam dokumen menggunakan Algoritma Depth First Search menjadi fokus dalam penelitian. Penelitian ini bertujuan untuk menghasilkan aplikasi pencarian yang relevan terhadap kata kunci pencarian yang diharapkan peningkatan efisien waktu bagi pengguna. Metode pengembangan sistem yang digunakan adalah Waterfall yang terdiri dari beberapa tahapan diantaranya, Analysis, Design, Code, Testing, dan Maintenance. Dengan menggunakan alat bantu flowchart untuk membuat rancangan sistem yang akan dibangun.

Teori graf merupakan pokok bahasan yang sudah tua usianya namun memiliki banyak terapan sampai saat ini. Menurut catatan sejarah, masalah jembatan Königsberg adalah masalah yang pertama kali menggunakan graf, dimana permasalahan jembatan Königsberg tersebut menuntut suatu pemodelan dalam suatu graf. Suatu graf direpresentasi secara grafik dengan menggambar titik untuk setiap simpul, dan menggambar garis antara dua simpul apabila mereka dihubungkan oleh suatu sisi. Graf tentunya memiliki banyak jenis. Suatu graf khusus yang disebut pohon juga merupakan salah satu contoh jenis dari graf.

Dengan struktur data apapun, algoritma umum yang akan ditulis untuk pertama kali adalah suatu algoritma yang menelusuri struktur datanya, atau istilah lainnya, algoritma traversal. Untuk graf, terdapat dua metode standar traversal untuk menelusuri graf itu sendiri, yaitu *Depth First Search* dan *Breadth First Search*. Dan nampaknya, dua traversal ini, umumnya *Depth First Search* lebih banyak, dapat dimanfaatkan dan digunakan untuk menyelesaikan variasi masalah pada graf ketika menelusuri graf itu sendiri.

Kelebihan dari algoritma Depth First Search yaitu pemakaian memori hanya sedikit untuk menyimpan node-node pada lintasan yang aktif dan menemukan solusi tanpa harus menguji lebih banyak dalam ruang keadaan. Algoritma Depth First Search dapat menemukan solusi secara cepat apabila solusi berada pada tingkat paling dalam dan paling kiri. Kekurangan dari algoritma Depth First Search yaitu kemungkinan tidak ditemukannya solusi yang diharapkan dan hanya menemukan satu solusi pada setiap pencarian.

BAB III
LANGKAH-LANGKAH PRAKTIKUM

3.1 Alat Dan Bahan

a. PC atau Laptop
b. Aplikasi Dev Cpp, VisualStudio Code, Python, NetBeans IDE, Java Jdk/Jre
c. Ekstensi Python di Software VisualStudio Code

3.2 Rangkaian Percobaan dan Hasil Percobaan

Adapun rangkaian percobaan yang telah dilakukan adalah sebagai berikut :

3.2.1 Breadth-First Search

Dengan Bahasa Pemrograman :

a. Python

Program :

```
# Python3 Program to print BFS traversal
# from a given source vertex. BFS(int s)
# traverses vertices reachable from s.
from collections import defaultdict

# This class represents a directed graph
# using adjacency list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # function to print a BFS of graph
    def BFS(self, s):

        # Mark all the vertices as not visited
        visited = [False] * (max(self.graph) + 1)
```

```
        # Create a queue for BFS
        queue = []

        # Mark the source node as
        # visited and enqueue it
        queue.append(s)
        visited[s] = True

        while queue:
            # Dequeue a vertex from
            # queue and print it
            s = queue.pop(0)
            print (s, end = " ")

            # Get all adjacent vertices of the
            # dequeue vertex s. if a adjacent
            # has not been visited, then mark it
            # visited and enqueue it
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True
# Driver code
# Create a graph given in
# the above diagram
g = Graph ()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print ("Following is Breadth First Traversal"
                " (starting from vertex 2)")
g.BFS(2)
```

Gambar 1. Program BFS Bahasa Python

Hasil :

```
Following is Breadth First Traversal (starting from ver
tex 2)
2 0 3 1
```

Gambar 2. Hasil BFS Bahasa Python

b. C++

Program :

```
// Program to print BFS traversal from a given
// source vertex. BFS(int s) traverses vertices
// reachable from s.
#include<iostream>
#include<list>
using namespace std;

// This class represents a directed graph using
// adjacency list representation
class Graph{
  int V;  // No. of vertices
  // Pointer to an array containing adjacency
  // lists
  list<int> *adj;
public:
  Graph(int V); // Constructor

  // Function to add an edge to graph
  void addEdge(int v, int w);

  // prints BFS traversal from a given source s
  void BFS(int s);
};

Graph::Graph(int V){
  this->V = V;
  adj = new list<int>[V];
}

void Graph::addEdge(int v, int w){
  adj[v].push_back(w);  // Add w to v's list
}

void Graph::BFS(int s){
  // Mark all the vertices as not visited
  bool *visited = new bool[V];
  for(int i = 0; i < V; i++)
    visited[i] = false;

  // Create a queue for BFS
  list<int> queue;
```

```
  // Mark the current node as visited and enqueue it
  visited[s] = true;
  queue.push_back(s);

  // 'i' will be used to get all adjacent
  // vertices of a vertex
  list<int>::iterator i;
  while(!queue.empty()){
    // Dequeue a vertex from queue and print it
    s = queue.front();
    cout << s << " ";
    queue.pop_front();

    // Get all adjacent vertices of the dequeued
    // vertex s. if a adjacent has not been visited,
    // then mark it visited and enqueue it
    for(i = adj[s].begin(); i != adj[s].end(); ++i){
      if(!visited[*i]){
        visited[*i] = true;
        queue.push_back(*i);
      }
    }
  }
}
// Driver program to test methods of graph class
int main(){
  // Create a graph given in the above diagram
  Graph g(4);
  g.addEdge(0, 1);
  g.addEdge(0, 2);
  g.addEdge(1, 2);
  g.addEdge(2, 0);
  g.addEdge(2, 3);
  g.addEdge(3, 3);

  cout << "Following is Breadth First Traversal "
     << " (starting from vertex 2) \n";
  g.BFS(2);
  return 0;
}
```

Gambar 3. Program BFS Bahasa C++

Hasil :

```
Following is Breadth First Traversal  (starting from vertex 2)
2 0 3 1
--------------------------------
Process exited after 0.0932 seconds with return value 0
Press any key to continue . . .
```

Gambar 4. Hasil BFS Bahasa C++

c. Java

Program :

```java
package artificialinteligence;
// Java program to print BFS traversal from a givern source vertex.
// BFS(int s) traverses vertices reachable from s.
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph {
    private int V;  // No. of vertices
    private LinkedList<Integer> adj[];  // Adjacency Lists

    // Constructor
    Graph(int v) {
        V = v;
        adj = new LinkedList[v];
        for (int i = 0; i < v; ++i) {
            adj[i] = new LinkedList();
        }
    }

    // Function to add an edge into the graph
    void addEdge(int v, int w) {
        adj[v].add(w);
    }

    // prints BFS traversal from a given source s
    void BFS(int s) {
        // Mark all the vertices as not visited(By default
        // set as false)
        boolean visited[] = new boolean[V];

        // Create a queue for BFS
        LinkedList<Integer> queue = new LinkedList<Integer>();
```

```
        // Mark the current node as visited and enqueue it
        visited[s] = true;
        queue.add(s);

        while (queue.size() != 0) {
            // Dequeue a vertex from queue and print it
            s = queue.poll();
            System.out.print(s + " ");

            // Get all adjacent vertices of the dequeued vertex s
            // If a adjacent has not been visited, the mark it
            // visited and enqueue it
            Iterator<Integer> i = adj[s].listIterator();
            while (i.hasNext()) {
                int n = i.next();
                if (!visited[n]) {
                    visited[n] = true;
                    queue.add(n);
                }
            }
        }
    }
}

public class BFS {

    // Driver method to
    public static void main(String[] args) {
        Graph g = new Graph(4);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        System.out.println("Following is Breadth First Traversal "
                + "(starting from vertex 2");
        g.BFS(2);
    }
}
```

Gambar 5. Program BFS Bahasa Java

Hasil :

```
Following is Breadth First Traversal (starting from vertex 2
2 0 3 1 BUILD SUCCESSFUL (total time: 1 second)
```

Gambar 6. Hasil BFS Bahasa Java

3.2.2 Depth-First Search

Dengan Bahasa Pemrograman :

d. Python

Program :

```python
# Python3 program to print DFS traversal
# from a given given graph
from collections import defaultdict

# This class represents a directed graph using
# adjacency list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict (list)

    # function to add an edge to graph
    def addEdge(self, u, v):
        self.graph[u].append(v)

    # A function used by DFS
    def DFSUtil (self, v, visited):

        # Mark the current node as visited
        # and print it
        visited.add(v)
        print(v, end=' ')

        # Recur for all the vertices
        # adjacent to this vertex
        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil (neighbour, visited)
```

```
    # The function to do DFS traversal. It uses
    # recursive DFSUtil ()
    def DFS(self, v):

        # Create a set to store visited vertices
        visited = set()

        # Call the recursive helper function
        # to print DFS traversal
        self.DFSUtil (v, visited)

# Driver code


# Create a graph given
# in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print("Following is DFS from (starting from vertex 2)")
g.DFS(2)
```

Gambar 7. Program DFS Bahasa Python

Hasil :

```
Following is DFS from (starting from vertex 2)
2 0 1 3
```

Gambar 8. Hasil DFS Bahasa Python

e. C++

Program :

```
// C++ program to print DFS traversal from
// a given vertex in a given graph
#include <bits/stdc++.h>
using namespace std;
// Graph class represents a directed graph
// using adjacency list representation
class Graph{
  public:
```

```
    map<int, bool> visited;
    map<int, list<int> > adj;
    // function to add an edge to graph
    void addEdge(int v, int w);
    // DFS traversal of the vertices
    // reachable from v
    void DFS(int v);
};

void Graph::addEdge(int v, int w){
  adj[v].push_back(w);  // Add w to v's list.
}

void Graph::DFS(int v){
  // Mark the current node as visited and
  // print it
  visited[v] = true;
  cout << v << " ";

  // Recur for all the vertices adjacent
  // to this vertex
  list<int>::iterator i;
  for(i = adj[v].begin(); i != adj[v].end(); ++i)
    if(!visited[*i])
      DFS(*i);
}

// Driver code
int main(){
  // Create a graph given in the above diagram
  Graph g;
  g.addEdge(0, 1);
  g.addEdge(0, 9);
  g.addEdge(1, 2);
  g.addEdge(2, 0);
  g.addEdge(2, 3);
  g.addEdge(9, 3);

  cout << "Following is Depth First Traversal "
     << " (starting from vertex 2) \n";
  g.DFS(2);
  return 0;
}
```

Gambar 9. Program DFS Bahasa C++

Hasil :

```
Following is Depth First Traversal  (starting from vertex 2)
2 0 1 9 3
--------------------------------
Process exited after 0.08938 seconds with return value 0
Press any key to continue . . .
```

Gambar 10. Hasil DFS Bahasa C++

f. Java

Program :

```
package artificialinteligence;

// Java program to print DFS
// traversal from a given given
// graph
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class GraphDFS {

    private int V;  // No. of vertices

    // Array of lists for
    // Adjacency List Represention
    private LinkedList<Integer> adj[];

    // Constructor
    @SuppressWarnings("Unchecked")
    GraphDFS(int v) {
        V = v;
        adj = new LinkedList[v];

        for (int i = 0; i < v; ++i) {
            adj[i] = new LinkedList();
        }

    }
```

```
    // Function to add an edge into the graph
    void addEdge(int v, int w) {
        adj[v].add(w);  // Add w to v's list.
    }

    // A function used by DFS
    void DFSUtil(int v, boolean visited[]) {

        // Mark the current node as visited and print it
        visited[v] = true;
        System.out.print(v + " ");

        // Recur for all the vertices adjacent to this
        // vertex
        Iterator<Integer> i = adj[v].listIterator();

        while (i.hasNext()) {
            int n = i.next();

            if (!visited[n]) {
                DFSUtil(n, visited);
            }

        }
    }

    // The function to do DFS traversal.
    // It uses recursive
    // DFSUtil()
    void DFS(int v) {

        // Mark all the vertices as
        // not visited(set as
        // false by default in java)
        boolean visited[] = new boolean[V];

        // call the recursive helper
        // function to print DFS
        // traversal
        DFSUtil(v, visited);
    }
}
```

```java
public class DFS {

    // Driver Code
    public static void main(String[] args) {
        GraphDFS g = new GraphDFS(4);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        System.out.println("Following is Depth First Traversal "
                + "(starting from vertex 2");
        g.DFS(2);
    }
}
```

Gambar 11. Program DFS Bahasa Java

Hasil :

```
Following is Depth First Traversal (starting from vertex 2
2 0 1 3 BUILD SUCCESSFUL (total time: 0 seconds)
```

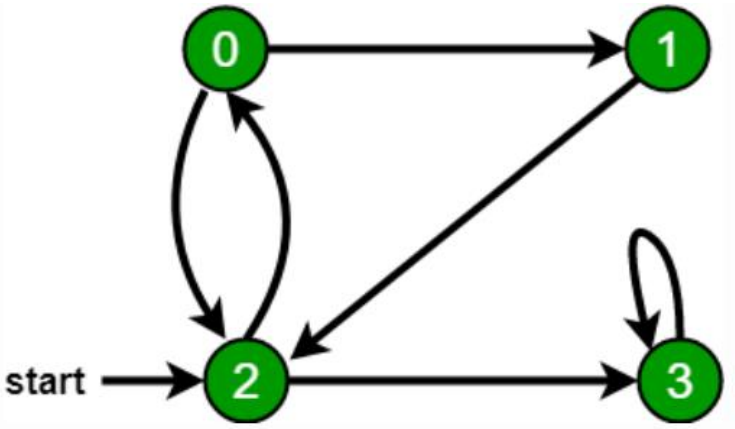
Gambar 12. Hasil DFS Bahasa Java

BAB IV
PEMBAHASAN DAN ANALISA HASIL

4.1 Breadth-First Search (BFS)

Breadth First Search untuk grafik mirip dengan Breadth First Traversal dari sebuah pohon. Satu-satunya tangkapan di sini adalah, tidak seperti pohon, grafik mungkin berisi siklus, jadi ini mungkin datang ke simpul yang sama lagi. Untuk menghindari pemrosesan node lebih dari sekali, maka disini menggunakan array yang dikunjungi boolean. Untuk mempermudah, diasumsikan bahwa semua simpul dapat dijangkau dari simpul awal.

Sebagai contoh, pada graf berikut, program memulai traversal dari simpul 2. Ketika program sampai pada simpul 0, program mencari semua simpul yang berdekatan dengannya. 2 juga merupakan simpul yang berdekatan dari 0. Jika program tidak menandai simpul yang telah dikunjungi, maka 2 akan diproses lagi dan ini akan menjadi proses yang tidak mengakhiri. Breadth First Traversal dari grafik berikut adalah 2, 0, 3, 1.



Gambar 13. Penggambaran dari grafik 2, 0, 3, 1

Dibawah ini adalah contoh dari hasil implementasi grafik diatas. Penerapannya menggunakan representasi daftar ketetanggaan grafik. Penampung daftar digunakan untuk menyimpan daftar node yang berdekatan dan antrian node yang diperlukan untuk traversal BFS.

Perhatikan bahwa kode di sebelumnya hanya melintasi simpul yang dapat dijangkau dari simpul sumber tertentu. Semua simpul mungkin tidak dapat dijangkau dari simpul yang diberikan (contoh: graf Terputus). Untuk mencetak semua simpul, kita dapat memodifikasi fungsi BFS untuk melakukan traversal mulai dari semua simpul satu per satu

4.1.1 Python

Hasil :

```
Following is Breadth First Traversal (starting from ver
tex 2)
2 0 3 1
```

Gambar 14. Hasil Pencarian menggunakan BFS pada Python

Analisis :

```
# Python3 Program to print BFS traversal
# from a given source vertex. BFS(int s)
# traverses vertices reachable from s.
from collections import defaultdict
# This class represents a directed graph
# using adjacency list representation
class Graph:
```

➔ Pertama import library defaultdict dari collections, kemudian membuat class Graph untuk merepresentasikan graf berarah menggunakan adjacency list representation.

```
    # Constructor
    def __init__(self):
```

➔ Kemudian buat konstruktor yang berisi fungsi __init__ yang berparameter self untuk menyimpan graf.

```
        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)
```

➔ Kemudian buat fungsi addEdge yang berparameter self, u, v untuk menambah sisi pada graf.

```
    # function to print a BFS of graph
    def BFS(self, s):
        # Mark all the vertices as not visited
        visited = [False] * (max(self.graph) + 1)
        # Create a queue for BFS
        queue = []
```

➔ Kemudian buat fungsi untuk memprint BFS dari graf dan tandai semua titik yang tidak dikunjungi (visited = [False] * (max(self.graph) + 1)) dan buat antrian untuk BFS (queue = []).

```
        # Mark the source node as
        # visited and enqueue it
        queue.append(s)
        visited[s] = True
```

➔ Tandai sumber node sebagai “telah dikunjungi” dan masukkan kedalam antrian.

```
        while queue:
            # Dequeue a vertex from
            # queue and print it
            s = queue.pop(0)
            print (s, end = " ")
```

➔ Kemudian lakukan perulangan while untuk menngambil data dari antrian dan kemudian diprint.

```
            # Get all adjacent vertices of the
            # dequeue vertex s. if a adjacent
            # has not been visited, then mark it
            # visited and enqueue it
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True
```

➔ Kemudian lakukan perulangan yg bernilai dari graph di index s.

```
# Driver code
# Create a graph given in
# the above diagram
g = Graph ()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print ("Following is Breadth First Traversal"
                " (starting from vertex 2)")

g.BFS(2)
```

➔ Terakhir membuat graf yang diberikan pada diagram dan cetak hasilnya dilayar.

4.1.2 C++

Hasil :

```
Following is Breadth First Traversal  (starting from vertex 2)
2 0 3 1
--------------------------------
Process exited after 0.0932 seconds with return value 0
Press any key to continue . . .
```

Gambar 15. Hasil Pencarian menggunakan BFS pada C++

Analisis :

```
// Program to print BFS traversal from a given
// source vertex. BFS(int s) traverses vertices
// reachable from s.
#include<iostream>
#include<list>
```
➔ Deklarasi library iostream dan list
```
using namespace std;
```
➔ Deklarasi bahwa akan menggunakan namespace.

```
// This class represents a directed graph using
// adjacency list representation
class Graph{
```
➔ Membuat class bernama Graph

```
  int V;  // No. of vertices
```
➔ Deklarasi variabel V bertipe data int

```
  // Pointer to an array containing adjacency
  // lists
  list<int> *adj;
```
➔ Deklarasi pointer bernama adj yang bertipe int melalui fungsi library list

```
public:
```
➔ Deklarasi fungsi public dari class

```
  Graph(int V); // Constructor
```
➔ Deklarasi constructor dari class Graph dengan mengisi nilai parameter dari V

```
  // Function to add an edge to graph
  void addEdge(int v, int w);
```
➔ Deklarasi fungsi addEdge yang bertipe void dengan menerima parameter variabel v dan w bertipe data int

```
  // prints BFS traversal from a given source s
  void BFS(int s);
```
➔ Deklarasi fungsi BFS bertipe void yang menerima parameter variabel s bertipe data int

```
};
Graph::Graph(int V){
```
➔ Deklarasi fungsi Graph yang menerima parameter variabel V

```
  this->V = V;
```
➔ Mengisi nilai variabel V global dengan nilai V variabel local

```
  adj = new list<int>[V];
```
➔ Mengisi nilai variabel adj dengan nilai list baru dari list yang berisi nilai V local

```
}
void Graph::addEdge(int v, int w){
```
➔ Deklarasi fungsi addEdge bertipe void dari class Graph, yang menerima parameter v dan w bertipe data int

```
  adj[v].push_back(w);  // Add w to v's list
```
➔ Menambahkan nilai w kedalam variabel adj yang berada di nilai v

```
}

void Graph::BFS(int s){
```
➔ Deklarasi fungsi BFS bertipe void dari class Graph, yang menerima parameter variabel s bertipe int

```
  // Mark all the vertices as not visited
  bool *visited = new bool[V];
  for(int i = 0; i < V; i++)
    visited[i] = false;
```
➔ Membuat seluruh variabel visited bertipe bool yang diisi nilai dari bool yang berada di nilai V. Kemudian membuat nilai visited menjadi false semua

```
  // Create a queue for BFS
  list<int> queue;
```
➔ Membuat variabel queue dari library list yang bertipe data int

```
  // Mark the current node as visited and enqueue it
  visited[s] = true;
```
➔ Membuat nilai visited pada index ke s bernilai true

```
queue.push_back(s);
```
➔ Melakukan push_back nilai pada variabel queue dengan nilai dari variabel s

```
// 'i' will be used to get all adjacent
// vertices of a vertex
list<int>::iterator i;
```
➔ Membuat iterator yang bernama i dari list yang bertipe data int

```
while(!queue.empty()){
```
➔ Melakukan while selama nilai queue tidak kosong maka akan dijalankan statement didalamnya

```
  // Dequeue a vertex from queue and print it
  s = queue.front();
```
➔ Mengisi nilai variabel s dengan nilai dari queue yang berada di depan

```
  cout << s << " ";
  queue.pop_front();
```
➔ Mencetak nilai s kelayar beserta spasi. Dan melakukan pengambilan beserta penghapusan nilai queue yang berada di depan

```
  // Get all adjacent vertices of the dequeued
  // vertex s. if a adjacent has not been visited,
  // then mark it visited and enqueue it
  for(i = adj[s].begin(); i != adj[s].end(); ++i){
```
➔ Melakukan perulangan for dengan nilai i diisi dengan nilai dari variabel adj pada index s dari awal. Dan melakukan pengecekan apakah i tidak sama dengan dengan nilai adj pada index s yang berada di akhir. Jika iya maka akan melakukan statement didalam for

```
    if(!visited[*i]){
```
➔ Mengecek apakah nilai visited yang berada di index i bernilai tidak true yang artinya bernilai false, dan nilai visited sebelumnya telah diisi dengan false, maka akan dijalankan statement didalam if tersebut

```
      visited[*i] = true;
```
➔ Membuat nilai visited pada index i menjadi true

```
      queue.push_back(*i);
```
➔ Menambah nilai queue dibagian belakang dengan nilai i

```
      }
    }
  }
}

// Driver program to test methods of graph class
int main(){
  // Create a graph given in the above diagram
  Graph g(4);
```

➔ Membuat objek bernama g dari class Graph yang telah dibuat diatas, dan mengisi nilai dari constructor class Graph dengan 4

```
  g.addEdge(0, 1);
  g.addEdge(0, 2);
  g.addEdge(1, 2);
  g.addEdge(2, 0);
  g.addEdge(2, 3);
  g.addEdge(3, 3);
```

➔ Memanggil fungsi addEdge dari class Graph melalui objek yang telah dibuat, dan mengisi nilai parameter pada tiap-tiap parameter yang diminta oleh method addEdge

```
  cout << "Following is Breadth First Traversal "
     << " (starting from vertex 2) \n";
```

➔ Mencetak kata-kata ke layar

```
  g.BFS(2);
```

➔ Memanggil fungsi BFS dan mengisi nilai parameter dengan 2

```
  return 0;
}
```

4.1.3 Java

Hasil :

```
Following is Breadth First Traversal (starting from vertex 2
2 0 3 1 BUILD SUCCESSFUL (total time: 1 second)
```

Gambar 16. Hasil Pencarian menggunakan BFS pada Java

Analisis :

```
package artificialinteligence;
// Java program to print BFS traversal from a givern source vertex.
// BFS(int s) traverses vertices reachable from s.
import java.io.*;
import java.util.*;
```

```
// This class represents a directed graph using adjacency list
// representation
class Graph {
    private int V;  // No. of vertices
    private LinkedList<Integer> adj[];  // Adjacency Lists
```

➔ Membuat class graph

➔ Deklarasi varibel V dengan tipe data integer, nilai variabel bersifat private hanya dapat diakses oleh class graph

➔ membuat linked list dengan variabel adj array yang bertipe data int

```
    // Constructor
    Graph(int v) {
        V = v;
        adj = new LinkedList[v];
        for (int i = 0; i < v; ++i) {
            adj[i] = new LinkedList();
        }
    }
```

➔ Membuat Construktor yang bertujuan untuk memeberi nilai awal pada class graph dengan parameter v, kemudian nilainya disimpan pada varibel V pada class graph

```
    // Function to add an edge into the graph
    void addEdge(int v, int w) {
        adj[v].add(w);
    }
```

➔ Membuat method addEdge dengan parameter variabel v dan variabel w untuk menambahkan sisi pada graph

➔ Menambahkan nilai pada edge[v] dari nilai variabel w

```
    // prints BFS traversal from a given source s
    void BFS(int s) {
        // Mark all the vertices as not visited(By default
        // set as false)
        boolean visited[] = new boolean[V];
```

➔ Membuat method BFS dengan parameter s

➔ mencetak BFS traversal dari sumber

➔ Menandai semua simpul sebagai “tidak dikunjungi” (Secara default set sebagai false)

```
        // Create a queue for BFS
        LinkedList<Integer> queue = new LinkedList<Integer>();
```

➔ Membuat sebuah queue untuk BFS

```
        // Mark the current node as visited and enqueue it
        visited[s] = true;
        queue.add(s);
```

➔ Menandai sumber dengan " telah dikunjungi" lalu memasukkan nilai kedalam antrian

```
        while (queue.size() != 0) {
            // Dequeue a vertex from queue and print it
            s = queue.poll();
            System.out.print(s + " ");
```

➔ Melakukan perulangan while untuk mengambil data dari antrin, lalu di print.

```
            // Get all adjacent vertices of the dequeued vertex s
            // If a adjacent has not been visited, the mark it
            // visited and enqueue it
            Iterator<Integer> i = adj[s].listIterator();
            while (i.hasNext()) {
                int n = i.next();
                if (!visited[n]) {
                    visited[n] = true;
                    queue.add(n);
                }
            }
        }
    }
}
```

➔ Dapatkan semua simpul yang berdekatan dari vertex s yang diurai, Jika berdekatan belum dikunjungi, tandai mengunjungi dan mengantri

```
public class BFS {

    // Driver method to
    public static void main(String[] args) {
        Graph g = new Graph(4);
```

➔ Membuat objek dari class Graph, dan mengisi nilai parameter construct 4

```
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);
```

```
        System.out.println("Following is Breadth First Traversal "
                + "(starting from vertex 2");
        g.BFS(2);
    }
}
```

➔ Membuat fungsi main dan membuat objek graph untuk memanggil method –method yang telah dideklarasi sebelumnya dengan membuat graph yang telah diberikan oleh diagram diatas

4.2 Depth-First Search (DFS)

Depth First Search untuk grafik mirip dengan Depth First Traversal dari sebuah pohon. Satu-satunya tangkapan di sini adalah, tidak seperti pohon, grafik mungkin berisi siklus, node dapat dikunjungi dua kali. Untuk menghindari pemrosesan node lebih dari sekali, gunakan array yang dikunjungi sebagai boolean.

Pendekatan pencarian kedalaman pertama adalah algoritma untuk melintasi atau mencari struktur data pohon atau grafik. Algoritma dimulai pada node root (memilih beberapa node arbitrer sebagai node root dalam kasus grafik) dan mengeksplorasi sejauh mungkin di setiap cabang sebelum melakukan backtracking. Jadi ide dasarnya adalah memulai dari root atau sembarang node dan menandai node tersebut dan pindah ke node yang berdekatan yang tidak ditandai dan melanjutkan loop ini sampai tidak ada node yang berdekatan yang tidak ditandai. Kemudian lacak kembali dan periksa node lain yang tidak ditandai dan lintasi. Terakhir, cetak node di jalur tersebut.

Algoritma :

1. Buat fungsi rekursif yang mengambil indeks node dan array yang dikunjungi.
2. Tandai node saat ini sebagai telah dikunjungi dan cetak node tersebut.
3. Lintasi semua node yang berdekatan dan tidak ditandai dan panggil fungsi rekursif dengan indeks node yang berdekatan.

4.2.1 Python

Hasil :

```
Following is DFS from (starting from vertex 2)
2 0 1 3
```

Gambar 17. Hasil Pencarian menggunakan DFS pada Python

Analisis :

```
# Python3 program to print DFS traversal
```

```
# from a given given graph
from collections import defaultdict
# This class represents a directed graph using
# adjacency list representation
class Graph:
```

➔ Pertama import library defaultdict dari collections, kemudian membuat class Graph untuk merepresentasikan graf berarah menggunakan adjacency list representation.

```
    # Constructor
    def __init__(self):
```

➔ Kemudian buat konstruktor yang berisi fungsi __init__ yang berparameter self, yang mana parameter self tersebut menyimpan graf.

```
        # default dictionary to store graph
        self.graph = defaultdict (list)

    # function to add an edge to graph
    def addEdge(self, u, v):
        self.graph[u].append(v)
```

➔ Kemudian buat fungsi addEdge yang berparameter self, u, v untuk menambah sisi pada graf.

```
    # A function used by DFS
    def DFSUtil (self, v, visited):
```

➔ Kemudian buat fungsi DFSUtil, yang mana fungsi ini digunakan oleh DFS yang parameternya berisi self, v, visited.

```
        # Mark the current node as visited
        # and print it
        visited.add(v)
        print(v, end=' ')
```

➔ Kemudian tuliskan visited.add(v) untuk menandai node yang sedang dikunjungi dan kemudian dicetak dilayar (print(v, end= ‘ ‘)).

```
        # Recur for all the vertices
        # adjacent to this vertex
        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil (neighbour, visited)
```

➔ Kemudian membuat perulangan untuk semua titik yang berdekatan dengan vertex.

```
    # The function to do DFS traversal. It uses
    # recursive DFSUtil ()
    def DFS(self, v):

        # Create a set to store visited vertices
        visited = set()

        # Call the recursive helper function
        # to print DFS traversal
        self.DFSUtil (v, visited)
```

➔ Kemudian buat fungsi DFS berparameter self, v dan membuat variable yang menyimpan data titik yang sudah dikunjungi dan dipanggil fungsi recursive untuk memprint DFS traversal.

```
# Driver code
# Create a graph given
# in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print("Following is DFS from (starting from vertex 2)")
g.DFS(2)
```

➔ Terakhir membuat graf yang diberikan pada diagram dan cetak hasilnya dilayar.

4.2.2 C++

Hasil :

```
Following is Depth First Traversal  (starting from vertex 2)
2 0 1 9 3
--------------------------------
Process exited after 0.08938 seconds with return value 0
Press any key to continue . . .
```

Gambar 18. Hasil Pencarian menggunakan DFS pada C++

Analisis :

```
// C++ program to print DFS traversal from
// a given vertex in a given graph
#include <bits/stdc++.h>
```

➔ Deklarasi library stdc++ yang berada dalam directory bits

```
using namespace std;
```
➔ Deklarasi bahwa akan menggunakan namespace yang bernama std, namespace ini berfungsi agar nantinya bisa melakukan cout dan cin

```
// Graph class represents a directed graph
// using adjacency list representation
class Graph{
```
➔ Deklarasi class bernama Graph

```
  public:
    map<int, bool> visited;
```
➔ Deklarasi variabel bernama visited dengan tipe map, yang memiliki tipe data int dan bool

```
    map<int, list<int> > adj;
```
➔ Membuat variabel adj dari map yang bertipe int, dan memiliki tipe dari list yang bertipe data int juga

```
    // function to add an edge to graph
    void addEdge(int v, int w);
```
➔ Membuat fungsi untuk menambahkan edge kedalam graph, melalu fungsi addEdge yang memiliki parameter variabel v dan w bertipe int

```
    // DFS traversal of the vertices
    // reachable from v
    void DFS(int v);
```
➔ Membuat fungsi DFS yang menerima nilai parameter v bertipe data int

```
};
void Graph::addEdge(int v, int w){
```
➔ Membuat fungsi dari class Graph yang bernama addEdge yang menerima parameter v dan w bertipe data int

```
  adj[v].push_back(w);  // Add w to v's list.
```
➔ Menambahkan data pada bagian akhir dari adj di index ke v, dengan nilai yang ditambahkan dari variabel w

```
}

void Graph::DFS(int v){
```
➔ Membuat fungsi dari class Graph bernama DFS yang menerima parameter variabel v bertipe int

```
  // Mark the current node as visited and
  // print it
  visited[v] = true;
  cout << v << " ";
```
➔ Membuat nilai visited pada index v bernilai true. Kemudian mencetak nilai v kelayar beserta spasi

```
  // Recur for all the vertices adjacent
  // to this vertex
  list<int>::iterator i;
```
➔ Membuat variabel untuk iterator bernama i dari list yang bertipe data int

```
  for(i = adj[v].begin(); i != adj[v].end(); ++i)
```
➔ Melakukan perulangan for dengan nilai i sama dengan nilai variabel adj yang pertama. Dan mengecek apakah nilai i tidak sama dengan nilai variabel adj yang terakhir, jika iya maka akan melakukan statement didalam for

```
    if(!visited[*i])
      DFS(*i);
```
➔ Mengecek apakah nilai variabel visited pada index i tidak true, dalam artian apakah false. Jika iya maka akan memanggil fungsi DFS dengan mengisi nilai parameter dari variabel i

```
}

// Driver code
int main(){
  // Create a graph given in the above diagram
  Graph g;
```
➔ Membuat objek dari class Graph bernama g

```
  g.addEdge(0, 1);
  g.addEdge(0, 9);
  g.addEdge(1, 2);
  g.addEdge(2, 0);
  g.addEdge(2, 3);
  g.addEdge(9, 3);
```
➔ Memanggil fungsi addEdge melalui objek yang telah dibuat. Dan mengisi nilai dari parameter fungsi addEdge

```
  cout << "Following is Depth First Traversal "
     << " (starting from vertex 2) \n";
```
➔ Mencetak kalimat di layar menggunakan cout

```
  g.DFS(2);
```
➔ Memanggil fungsi DFS melalui objek g. Dan mengisi nilai parameter dengan 2

```
  return 0;
}
```

4.2.3 Java

Hasil :

```
Following is Depth First Traversal (starting from vertex 2
2 0 1 3 BUILD SUCCESSFUL (total time: 0 seconds)
```

Gambar 19. Hasil Pencarian menggunakan DFS pada Java

Analisis :

```
package artificialinteligence;

// Java program to print DFS
// traversal from a given given
// graph
import java.io.*;
import java.util.*;
```
➔ Mengimport java.io dan java.util untuk memanggil data inputan dan agar user juga bisa menginputkan data dari keyboard

```
// This class represents a directed graph using adjacency list
// representation
class GraphDFS {
```
➔ Membuat class graphDFS

```
    private int V;  // No. of vertices
    // Array of lists for
    // Adjacency List Represention
    private LinkedList<Integer> adj[];
```
➔ Deklarasi variabel V dengan tipe data int dan membuat linked list dengan variabel adj array yang bertipe data int, tetapi hanya dapat diakses oleh class graphDFS.

```
    // Constructor
    @SuppressWarnings("Unchecked")
```
➔ Membuat operasi bisa menjadi unchecked di IDE

```
    GraphDFS(int v) {
        V = v;
        adj = new LinkedList[v];
```

```
        for (int i = 0; i < v; ++i) {
            adj[i] = new LinkedList();
        }
    }
```

➔ Membuat Contructor class GraphDFS dengan parameter v, bertujuan untuk memeberi nilai awal pada class graph dengan parameter v, kemudian nilainya disimpan pada varibel V pada class graph

```
    // Function to add an edge into the graph
    void addEdge(int v, int w) {
        adj[v].add(w);  // Add w to v's list.
    }
```

➔ Membuat method addEdge dengan parameter variabel v dan variabel w untuk menambahkan sisi pada graph

➔ Menambahkan nilai pada edge[v] dari nilai variabel w

```
    // A function used by DFS
    void DFSUtil(int v, boolean visited[]) {
```

➔ Membuat method DFSUtil dengan parameter v, visited Boolean), dimana fungsi ini digunakan oleh DFS

```
        // Mark the current node as visited and print it
        visited[v] = true;
        System.out.print(v + " ");
```

➔ Tuliskan visited[v] untuk menandai node yang telah dikunjungi, kemudian tampilkan kelayar

```
        // Recur for all the vertices adjacent to this
        // vertex
        Iterator<Integer> i = adj[v].listIterator();

        while (i.hasNext()) {
            int n = i.next();

            if (!visited[n]) {
                DFSUtil(n, visited);
            }

        }
    }
```

➔ Membuat sebuah pperulangan while untuk mengecek kondisi, apakah objek iterator masih mempunyai nilai pada selanjutnya atau tidak. Jika ya, maka lanjutkan iterator dan simpan kedalam variabel n.

➔ Jika tidak maka cetak visited disimpan pada variabel n

```
    // The function to do DFS traversal.
    // It uses recursive
    // DFSUtil()
    void DFS(int v) {
```
➔ Membuat method DFS berparameter int v

```
        // Mark all the vertices as
        // not visited(set as
        // false by default in java)
        boolean visited[] = new boolean[V];
```
➔ Secara default, java akan membuat variabel boolean yang tidak memiliki menjadi false

```
        // call the recursive helper
        // function to print DFS
        // traversal
        DFSUtil(v, visited);
```
➔ Memanggil pembantu rekursif untuk mencetak DFS Traversal

```
    }
}

public class DFS {
    // Driver Code
    public static void main(String[] args) {
```
➔ Membuat class main agar dapat membuat sebuah objek baru graphDFS dengan nilai 4 untuk dapat di pemanggil method – method yang ada pada class graph dan program dieksekusi

```
        GraphDFS g = new GraphDFS(4);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);
```
➔ Membuat diagram untuk graph

```
        System.out.println("Following is Depth First Traversal "
                + "(starting from vertex 2");
        g.DFS(2);
    }
}
```
➔ Menampilkan hasil eksekusi program ke layar

BAB V
KESIMPULAN DAN SARAN

4.3 Kesimpulan

Pada teori graf, terdapat algoritma standar yang praktis dan umum untuk penelusuran (traversal) pada graf, yaitu Depth First Search dan Breadth First Search, implementasinya cukup sederhana, namun walaupun sederhana, banyak digunakan dalam menyelesaikan masalah-masalah pada graf. Depth First Search dan Breadth First Search memiliki langkah algoritma yang berbeda, namun dalam implementasinya, kedua algoritma ini saling berhubungan, dan memiliki kemiripan. Depth First Search dan Breadth First Search digunakan dalam berbagai banyak aplikasi, dan salah satunya merupakan permasalahan dalam kehidupan sehari-hari. Contohnya, penggunaan Depth First Search dalam menemukan solusi dari Maze.

Algoritma Breadth First Search dan Depth First Search berhasil diterapkan dalam aplikasi melakukan perbandingan berdasarkan jumlah langkah dan lama waktu pencarian.

4.4 Saran

Saran untuk pengembangan lebih lanjut dari metode-metode algoritma pencarian Breadth First Search (BFS) dan Depth First Search yang lebih jauh lagi. Dan adapun saran mengenai metode-metode searching lainnya, seperti pencarian Heuristik Search yaitu berupa Hill Climbing, Generate & Test, A*, Best First Search(BFS), dan lain sebagainya.

DAFTAR PUSTAKA

[1] Nilsson, N.J. *Principles of Artificial Intelligence*. Tioga Pub. Co., 1982.

[2] Pip. "Algoritma BFS(Breadth First Search)". 12 Oktober 2015. https://piptools.net/algoritma-bfs-breadth-first-search/.

[3] Pinkerton Brian.1994."Finding What People Want: Experiences with the WebCrawler".

[4] Cynthia Kustanto, Ratna Mutia, Pocut Viqarunnisa. "Penerapan Algoritma Breadth First Search dan Depth First Search pada FTP Search Engine for ITB Network. 17 Mei 2005. ITB.

[5] Jufri. "Implementasi Algoritma Breadth First Search (BFS) Pada Masalah Penyusunan Blok". 16 Agustus 2019. https://ejurnal.dipanegara.ac.id/index.php/sisiti/article/view/95-102.

[6] Budi Prasetyo, Maulidia Rahmah Hidayah. "Penggunaan Metode Breadth First Search dan Depth First Search pada Strategi Game Kamen Rider Decade. https://journal.unnes.ac.id/nju/index.php/sji/article/view/4022.

[7] Doddy Teguh Yuwono, Abdul Fadlil, Sunardi. "Perbandingan Algoritma BFS dan DFS sebagai Focused Crawler". Universitas Ahmad Dahlan. 6 Desember 2016.

[8] Pip. "Algoritma DFS(Depth First Search". 14 Oktober 2015. https://piptools.net/algoritma-dfs-depth-first-search/.

[9] Ulty. "Metode Depth First Search". 03 Februari 2020. https://lancangkuning.com/post/15012/metode-depth-first-search.html.

[10] Siti Lailiyah, Amelia Yusnita, Twom Ali Panotogomo. "Penerapan Algoritma Depth First Search pada Sistem Pencarian Dokumen". https://jurnal.poltekba.ac.id/index.php/prosiding/article/view/395.

[11] Hafid Inggiantowi. "Perbandingan Algoritma Penelusuran Depth First Search dan Breadth First Search pada Graf serta Aplikasinya". Institut Teknologi Bandung.

[12] Emil Glorio Masala, Immanuela P. Saputro, Rinaldo T.B. Turang. "Perbandingan Algoritma Breadth First Search Dan Depth First Search Pada Game Mummy Maze Deluxe". Universitas Katolik De La Salle Manado; Kombos.