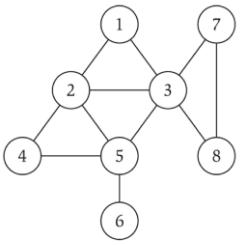
Nama : Rizky Anugerah
NPM : 140810180049
Tugas 6

Pendahuluan

Graf Tak Berarah (Undirected
Graf) (Undirected) graph:
G=(V,E)

- V = sekumpulan node (vertex, simpul, titik, sudut)
- E = sekumpulan edge (garis, tepi)
- Menangkap hubungan berpasangan antar objek.
- Parameter ukuran Graf: n = |V|, m=|E|



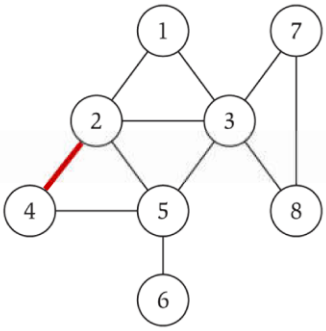
V = {1,2,3,4,5,6,7,8}
E = {(1,2), (1,3), (2,3), (2,4), (2,5), (3,5),(3,7), (3,8), (4,5), (5,6), (7,8)}
n = 8
M = 11

Dalam pemrograman, Graf dapat direpresentasikan dengan ***adjacency matrix*** dan ***adjacency list***

Representasi Graf dengan Adjacency Matrix

Adjacency Matrix: n-ke-n matriks dengan $A_{uv} = 1$ jika (u,v) adalah sebuah garis

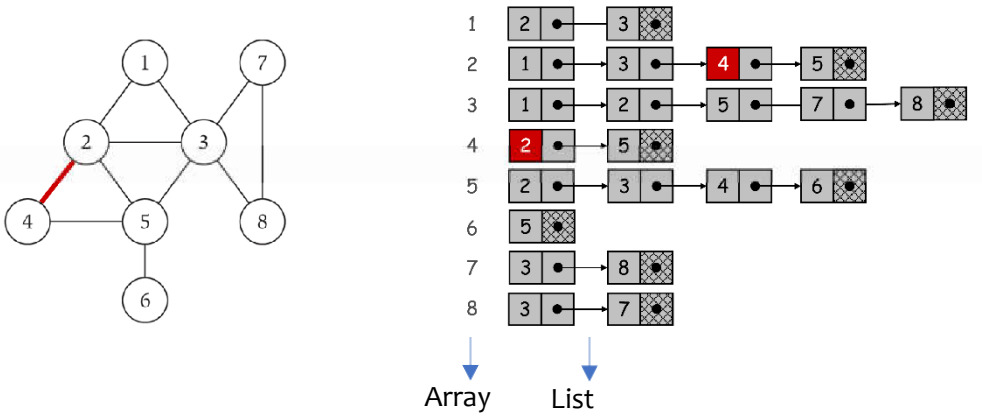
- Dua representasi dari setiap sisi
- Ruang berukuran sebesar n^2
- Memeriksa apakah (u, v) edge membutuhkan waktu Θ(1)
- Mengidentifikasi semua tepi membutuhkan Θ (n^2) waktu



	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Representasi Graf dengan Adjacency List Adjacency List: node diindeks sebagai list

- Dua representasi untuk setiap sisi
- Ukuran ruang m + n
- Memeriksa apakah (u, v) edge membutuhkan O (deg (u)). Degree = jumlah tetangga u.
- Mengidentifikasi semua tepi membutuhkan Θ(m + n).

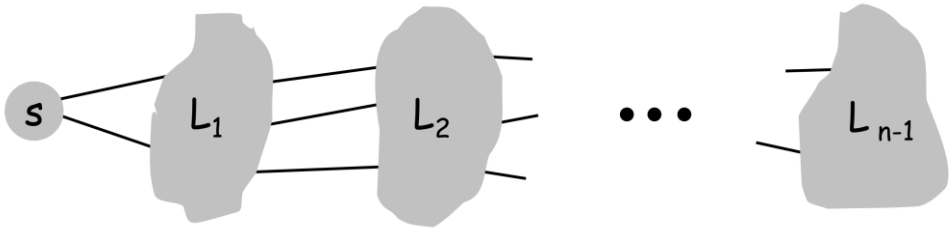


Breadth First Search

Intuisi BFS. Menjelajahi alur keluar dari s ke semua arah yang mungkin, tambahkan node satu "layer" sekaligus.

Algoritma BFS

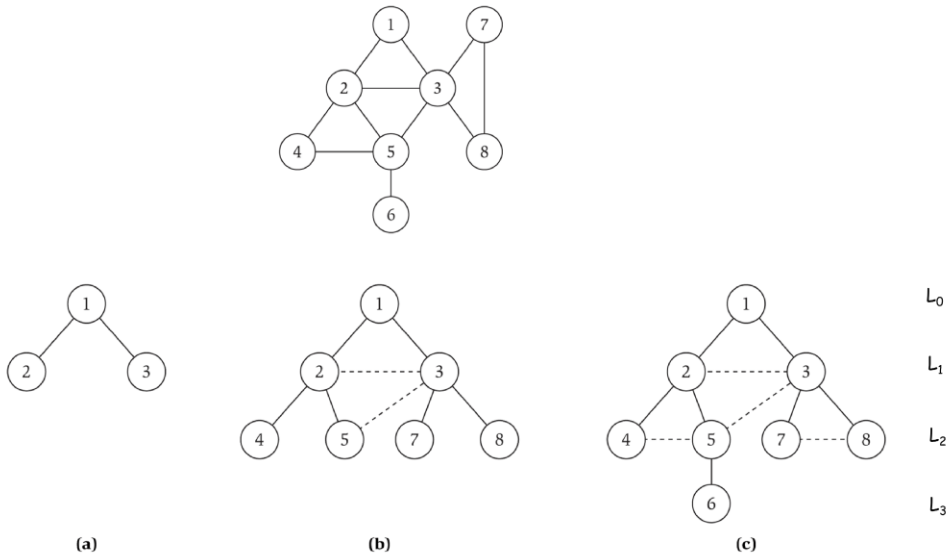
- $L_0 = \{s\}$
- $L_1 =$ *semua tetangga dari* L_0
- $L_2 =$ *semua node yang tidak termasuk ke dalam* L_0 atau L_1, dan yang mempunyai edge ke sebuah node di L_1
- $Li + 1$ = semua node yang bukan milik layer sebelumnya, dan yang memiliki edge ke node di Li



Gambar 1. Ilustrasi algoritma BFS

Teorema 1.0

Untuk setiap i, Li terdiri dari semua node pada jarak tepat ke i dari s. Ada path dari s ke t jika t muncul di beberapa layer.



Gambar 2. Ilustrasi pembentukan tree BFS dari undirected Graf

Implementasi BFS dalam Koding Program

- Adjacency list adalah representasi struktur data paling ideal untuk BFS
- Algoritma memeriksa setiap ujung yang meninggalkan node satu per satu. Ketika kita memindai edge yang meninggalkan u dan mencapai edge(u, v), kita perlu tahu apakah node v telah ditemukan sebelumnya oleh pencarian.
- Untuk menyederhanakan ini, kita maintain array yang ditemukan dengan panjang n dan mengatur Discovered [v] = true segera setelah pencarian kita pertama kali melihat v. Algoritma BFS membangun lapisan node L_1, L_2, ..., di mana L_i adalah set node pada jarak *i* dari sumber *s*.
- Untuk mengelola node dalam layer *Li*, kami memiliki daftar *L[i]* untuk setiap $i = 0,1,2, \ldots$.

```
BFS(s):
  Set Discovered[s] = true and Discovered[v] = false for all other v
  Initialize L[0] to consist of the single element s
  Set the layer counter i = 0
  Set the current BFS tree T = ∅
  While L[i] is not empty
    Initialize an empty list L[i + 1]
    For each node u ∈ L[i]
      Consider each edge (u, v) incident to u
      If Discovered[v] = false then
        Set Discovered[v] = true
        Add edge (u, v) to the tree T
        Add v to the list L[i + 1]
      Endif
    Endfor
    Increment the layer counter i by one
  Endwhile
```

Depth First Search

Algoritma BFS muncul, khususnya, sebagai cara tertentu mengurutkan node yang kita kunjungi — dalam lapisan berurutan, berdasarkan pada jarak node lain dari s. Metode alami lain untuk menemukan node yang dapat dijangkau dari s adalah pendekatan yang mungkin Anda dilakukan jika grafik G benar-benar sebuah labirin dari kamar yang saling berhubungan dan kita berjalan-jalan di dalamnya.

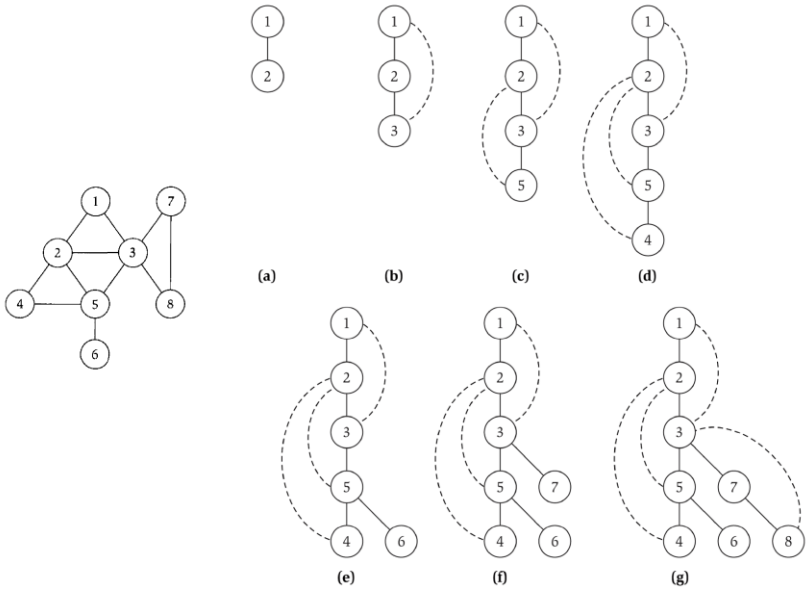
Kita akan mulai dari s dan mencoba edge pertama yang mengarah ke ke node v. Kita kemudian akan mengikuti edge pertama yang mengarah keluar dari v, dan melanjutkan dengan cara ini sampai kita mencapai "jalan buntu" —sebuah node di mana Anda sudah menjelajahi semua tetangganya. Kita kemudian akan mundur sampai kita mencapai node dengan tetangga yang belum dijelajahi, dan melanjutkan dari sana. Kita menyebutnya Depth-first search (DFS), karena ini mengeksplorasi G dengan masuk sedalam mungkin dan hanya mundur jika diperlukan.

DFS juga merupakan implementasi khusus dari algoritma component-growing generik yang dijelaskan sebelumnya. Kita dapat memulai DFS dari titik awal mana pun tetapi mempertahankan pengetahuan global tentang node yang telah dieksplorasi.

```
DFS(u):
  Mark u as "Explored" and add u to R
  For each edge (u, v) incident to u
    If v is not marked "Explored" then
      Recursively invoke DFS(v)
    Endif
  Endfor
```

Untuk menerapkan ini pada problem konektivitas s-t, kita cukup mendeklarasikan semua node pada awalnya untuk tidak dieksplorasi, dan memanggil DFS (s).

Ada beberapa kesamaan dan beberapa perbedaan mendasar antara DFS dan BFS. Kesamaan didasarkan pada fakta bahwa mereka berdua membangun komponen terhubung yang mengandung s, dan bahwa mereka mencapai tingkat efisiensi yang serupa secara kualitatif. Sementara DFS akhirnya mengunjungi set node yang sama persis seperti BFS, ia biasanya melakukannya dalam urutan yang sangat berbeda; menyelidiki jalan panjang, berpotensi menjadi sangat jauh dari s, sebelum membuat cadangan untuk mencoba lebih dekat node yang belum dijelajahi.



Gambar 3. Ilustrasi pembentukan tree DFS dari undirected graph

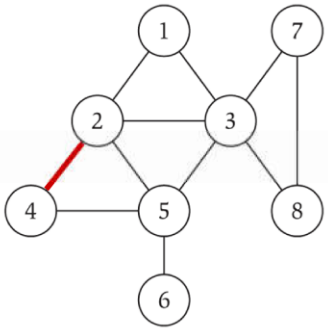
Implementasi BFS dalam Koding Program

Implementasi DFS paling ideal adalah dengan menggunakan stack. Adapun algoritma DFS dengan stack adalah sebagai berikut:

```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
       Set Explored[u] = true
       For each edge (u, v) incident to u
         Add v to the stack S
       Endfor
    Endif
  Endwhile
```

Tugas Anda

1. Dengan menggunakan *undirected graph* dan *adjacency matrix* berikut, buatlah koding programmnya menggunakan bahasa C++.



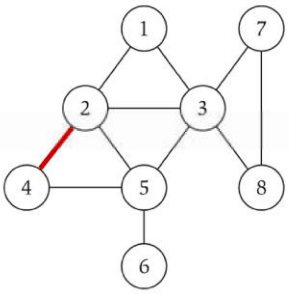
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

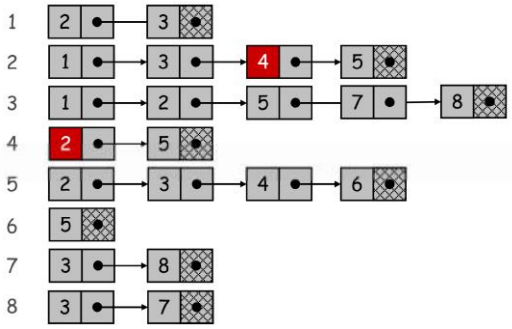
```
/*
Nama    : Rizky Anugerah
NPM     : 140810180049
Kelas   : A
Program : Adjacency Matrix
*/

#include <iostream>
using namespace std;

int vertArr[20][20];
int count = 0;

void printMatrix(int v)
{
    int i, j;
    for (i = 0; i < v; i++)
    {
        for (j = 0; j < v; j++)
        {
            cout << vertArr[i][j] << " ";
        }
        cout << endl;
    }
}

void addEdge(int u, int v)
{
    vertArr[u][v] = 1;
    vertArr[v][u] = 1;
}

int main()
{
    int v = 8;      // verticle = 8
```

```
    addEdge(0, 1);
    addEdge(0, 2);

    addEdge(1, 0);
    addEdge(1, 2);
    addEdge(1, 3);
    addEdge(1, 4);

    addEdge(2, 0);
    addEdge(2, 1);
    addEdge(2, 4);
    addEdge(2, 6);
    addEdge(2, 7);

    addEdge(3, 1);
    addEdge(3, 4);

    addEdge(4, 1);
    addEdge(4, 2);
    addEdge(4, 3);
    addEdge(4, 5);

    addEdge(5, 4);

    addEdge(6, 2);
    addEdge(6, 7);

    addEdge(7, 2);
    addEdge(7, 6);

    printMatrix(v);
}
```

```
0 1 1 0 0 0 0 0
1 0 1 1 1 0 0 0
1 1 0 0 1 0 1 1
0 1 0 0 1 0 0 0
0 1 1 1 0 1 0 0
0 0 0 0 1 0 0 0
0 0 1 0 0 0 0 1
0 0 1 0 0 0 1 0
```

2. Dengan menggunakan *undirected graph* dan representasi *adjacency list*, buatlah koding programmnya menggunakan bahasa C++.



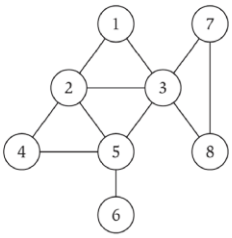


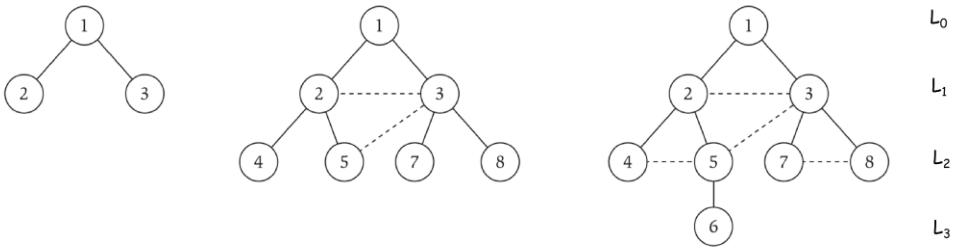

```
/*
Nama    : Rizky Anugerah
NPM     : 140810180049
Kelas   : A
Program : Adjacency List
*/

#include <iostream>
#include <list>
using namespace std;

struct Graph
{
    int vertex;
    list<int> *edge;
};
Graph G;

void graph(Graph &G, int vertex)
{
    G.vertex = vertex;
    G.edge = new list<int>[vertex];
}

void addEdge(Graph &G, int i, int j)
{
    G.edge[i].push_back(j);
}

void printList(Graph G)
{
    for (int i = 0; i < G.vertex; ++i)
    {
        cout << "\n Vertex " << i+1 << "\n head";
        for (auto x : G.edge[i])
            cout << " -> " << x+1;
```

```
        cout << endl;
    }
}

int main()
{
    graph(G, 8);

    addEdge(G, 0, 1);
    addEdge(G, 0, 2);

    addEdge(G, 1, 0);
    addEdge(G, 1, 2);
    addEdge(G, 1, 3);
    addEdge(G, 1, 4);

    addEdge(G, 2, 0);
    addEdge(G, 2, 1);
    addEdge(G, 2, 4);
    addEdge(G, 2, 6);
    addEdge(G, 2, 7);

    addEdge(G, 3, 1);
    addEdge(G, 3, 4);

    addEdge(G, 4, 1);
    addEdge(G, 4, 2);
    addEdge(G, 4, 3);
    addEdge(G, 4, 5);

    addEdge(G, 5, 4);

    addEdge(G, 6, 2);
    addEdge(G, 6, 7);

    addEdge(G, 7, 2);
    addEdge(G, 7, 6);

    printList(G);
}
```

```
Vertex 1
head -> 2 -> 3

Vertex 2
head -> 1 -> 3 -> 4 -> 5

Vertex 3
head -> 1 -> 2 -> 5 -> 7 -> 8

Vertex 4
head -> 2 -> 5

Vertex 5
head -> 2 -> 3 -> 4 -> 6

Vertex 6
head -> 5

Vertex 7
head -> 3 -> 8

Vertex 8
head -> 3 -> 7
```

3. Buatlah program Breadth First Search dari algoritma BFS yang telah diberikan. Kemudian uji coba program Anda dengan menginputkan *undirected graph* sehingga menghasilkan tree BFS. Hitung dan berikan secara asimptotik berapa kompleksitas waktunya dalam Big-Θ!






```
/*
Nama    : Rizky Anugerah
NPM     : 140810180049
Kelas   : A
Program : BFS
*/

#include <iostream>
#include <list>
using namespace std;

struct Graph
{
    int vertex;
    list<int> *edge;
};
Graph G;

void graph(Graph &G, int vertex)
{
    G.vertex = vertex;
    G.edge = new list<int>[vertex];
}

void addEdge(Graph &G, int i, int j)
{
    G.edge[i].push_back(j);
}

void BFS(Graph G, int vertex)
{
```

```
    bool *visited = new bool[G.vertex];
    for (int i = 0; i < G.vertex; i++)
        visited[i] = false;

    list<int> queue;
    visited[vertex] = true;
    queue.push_back(vertex);

    while (!queue.empty())
    {
        vertex = queue.front();
        cout << vertex+1 << " ";
        queue.pop_front();

        for (list<int>::iterator i = G.edge[vertex].begin(); i != G.edge[verte
x].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}

int main()
{
    graph(G, 8);

    addEdge(G, 0, 1);
    addEdge(G, 0, 2);

    addEdge(G, 1, 0);
    addEdge(G, 1, 2);
    addEdge(G, 1, 3);
    addEdge(G, 1, 4);

    addEdge(G, 2, 0);
    addEdge(G, 2, 1);
    addEdge(G, 2, 4);
    addEdge(G, 2, 6);
    addEdge(G, 2, 7);

    addEdge(G, 3, 1);
    addEdge(G, 3, 4);

    addEdge(G, 4, 1);
```

```
    addEdge(G, 4, 2);
    addEdge(G, 4, 3);
    addEdge(G, 4, 5);

    addEdge(G, 5, 4);

    addEdge(G, 6, 2);
    addEdge(G, 6, 7);

    addEdge(G, 7, 2);
    addEdge(G, 7, 6);

    cout << "Breadth First Search\n";
    BFS(G, 0);
}
```

```
Breadth First Search
1 2 3 4 5 7 8 6
```

Kompleksitas waktu asimptotik

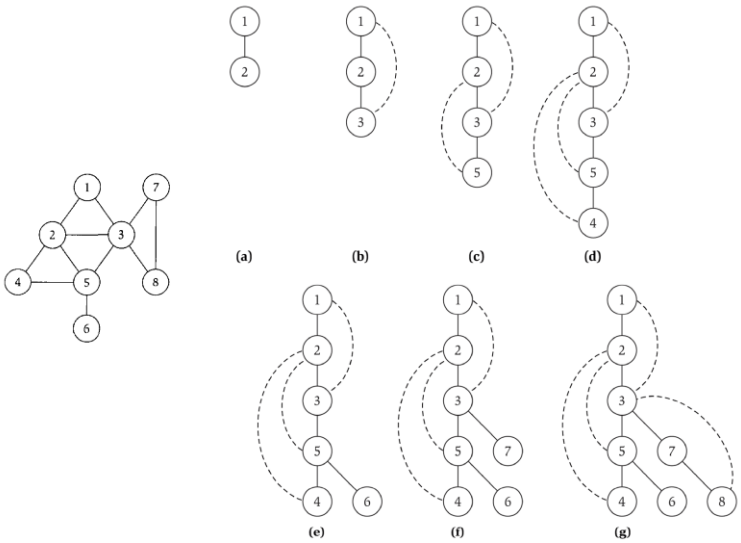
Breadth first search adalah algoritma yang melakukan pencarian secara melebar yang mengunjungi simpul secara preorder yaitu mengunjungi suatu simpul kemudian mengunjungi semua simpul yang bertetangga dengan simpul tersebut terlebih dahulu. Jika terjadi worst case, maka:

Dik:
V = Jumlah Vertex
E = Jumlah Edge

vertex yang belum dikunjungi = O(V)
vertex awal yang sudah dikunjungi lalu masukan ke queue = O(1)
cetak vertex = O(V)
kunjungi setiap vertex yang belum dikunjungi dan masuk ke queue : O(E)

T(n) = O(V) + O(1) + O(V) + O(E)
= O(maks(V,1)) + O(V) + O(E)
= O(V) + O (V) + O(E)
= O(maks(V,V)) + O(E) = O(V) + O(E)
= **O(|V| + |E|)**

4. Buatlah program Depth First Search dari algoritma DFS yang telah diberikan. Kemudian uji coba program Anda dengan menginputkan *undirected graph* sehingga menghasilkan tree DFS. Hitung dan berikan secara asimptotik berapa kompleksitas waktunya dalam Big-Θ!




```
/*
Nama    : Rizky Anugerah
NPM     : 140810180049
Kelas   : A
Program : DFS
*/

#include <iostream>
#include <list>
using namespace std;

struct Graph
{
    int vertex;
    list<int> *edge;
};
Graph G;

void graph(Graph &G, int vertex)
{
    G.vertex = vertex;
    G.edge = new list<int>[vertex];
}

void addEdge(Graph &G, int i, int j)
{
    G.edge[i].push_back(j);
}

void DFSPrint(int vertex, bool visited[])
{
    visited[vertex] = true;
```

```
    cout << vertex+1 << " ";

    for (list<int>::iterator i = G.edge[vertex].begin(); i != G.edge[vertex].e
nd(); ++i)
        if (!visited[*i])
            DFSPrint(*i, visited);
}

void DFS(Graph G, int vertex)
{
    bool *visited = new bool[G.vertex];
    for (int i = 0; i < G.vertex; i++)
        visited[i] = false;

    for (int i = 0; i < G.vertex; i++)
        if (visited[i] == false)
            DFSPrint(i, visited);
}

int main()
{
    graph(G, 8);

    addEdge(G, 0, 1);
    addEdge(G, 0, 2);

    addEdge(G, 1, 0);
    addEdge(G, 1, 2);
    addEdge(G, 1, 3);
    addEdge(G, 1, 4);

    addEdge(G, 2, 0);
    addEdge(G, 2, 1);
    addEdge(G, 2, 4);
    addEdge(G, 2, 6);
    addEdge(G, 2, 7);

    addEdge(G, 3, 1);
    addEdge(G, 3, 4);

    addEdge(G, 4, 1);
    addEdge(G, 4, 2);
    addEdge(G, 4, 3);
    addEdge(G, 4, 5);

    addEdge(G, 5, 4);

    addEdge(G, 6, 2);
```

```
    addEdge(G, 6, 7);

    addEdge(G, 7, 2);
    addEdge(G, 7, 6);

    cout << "Depth First Search\n";
    DFS(G, 0);
}
```

```
Depth First Search
1 2 3 5 4 6 7 8
```

Kompleksitas waktu asimptotik

Algoritma DFS (Depth First Search) adalah algoritma melakukan perhitungan secara terurut dari urutan terakhir. Setelah menghabiskan semua kemungkinan dari titik terakhir, barulah mundur ke titik-titik sebelumnya sampai pada titik pertama. Untuk kompleksitas waktunya:

Dik:

V = Jumlah Vertex

E = Jumlah Edge

Kunjungi vertex awal lalu cetak = O(1)

Rekursif untuk semua vertex = T(E/1)

Menandai semua vertex yang belum dikunjungi = O(V)

Rekursif untuk mencetak DFS = T(V/1)

T(n) = O(1) + T(E/1) + O(V) + T(V/1)
= O(1) + O(E) + O(V) + O(V)
= O(maks(1,E)) + O(V) + O(V)
= O(E) + O (V) + O(V)
= O(maks(V,V)) + O(E)
= O(V) + O(E)
= **O(|V| + |E|)**