**TUGAS PRAKTIKUM**

**ANALISIS ALGORITMA**

**MODUL 6**



**Disusun Oleh:**

Agnes Hata 140810170011

Rifaa’ Zalfaa’ Fakhriyyah 140810170031

Fahmi Auliya 140810170041

Firmansyah Yanuar 140810170051

**PROGRAM STUDI S1 TEKNIK INFORMATIKA**

**FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM**

**UNIVERSITAS PADJADJARAN**

**2019**

1. Matrix Adjacency

#include<iostream>

using namespace std;

int main(){

    int matriks[8][8] = {

    {0,1,1,0,0,0,0,0},

    {1,0,1,1,1,0,0,0},

    {1,1,0,0,1,0,1,1},

    {0,1,0,1,1,0,0,0},

    {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}

    };

    for(int i=0;i<8;i++){

        for(int j=0; j<8;j++){

            cout<<matriks[i][j]<<" ";

        }

        cout<<endl;

    }

}

Screenshot



1. List Adjacency

/\*

\* C++ Program to Implement Adjacency List

\*/

#include <iostream>

#include <cstdlib>

using namespace std;

/\*

\* Adjacency List Node

\*/

struct AdjListNode

{

int dest;

struct AdjListNode\* next;

};

/\*

\* Adjacency List

\*/

struct AdjList

{

struct AdjListNode \*head;

};

/\*

\* Class Graph

\*/

class Graph

{

private:

int V;

struct AdjList\* array;

public:

Graph(int V)

{

this->V = V;

array = new AdjList [V];

for (int i = 0; i < V; ++i)

array[i].head = NULL;

}

/\*

\* Creating New Adjacency List Node

\*/

AdjListNode\* newAdjListNode(int dest)

{

AdjListNode\* newNode = new AdjListNode;

newNode->dest = dest;

newNode->next = NULL;

return newNode;

}

/\*

\* Adding Edge to Graph

\*/

void addEdge(int src, int dest)

{

AdjListNode\* newNode = newAdjListNode(dest);

newNode->next = array[src].head;

array[src].head = newNode;

newNode = newAdjListNode(src);

newNode->next = array[dest].head;

array[dest].head = newNode;

}

/\*

\* Print the graph

\*/

void printGraph()

{

int v;

for (v = 0; v < V; ++v)

{

AdjListNode\* pCrawl = array[v].head;

cout<<"\n Adjacency list of vertex "<<v<<"\n head ";

while (pCrawl)

{

cout<<"-> "<<pCrawl->dest;

pCrawl = pCrawl->next;

}

cout<<endl;

}

}

};

/\*

\* Main

\*/

int main()

{

int pilihan,a,b,n;

cout<<"Banyak node : ";cin>>n;

Graph gh(n);

for(; ;)

{

cout<<"\nMenu\n"

<<"1. Tambah edge\n"

<<"2. Print Edge\n"

<<"0. Exit\n\n"

<<"Pilihan : ";cin>>pilihan;

switch (pilihan)

{

case 1:

cout<<"\nedge(a,b)\n"

<<"Input a : ";cin>>a;

cout<<"Input b : ";cin>>b;

gh.addEdge(a,b);

continue;

case 2:

gh.printGraph();

continue;

case 0:

return 0;

break;

default:

continue;

}

}

// Graph gh(5);

// gh.addEdge(0, 1);

// gh.addEdge(0, 4);

// gh.addEdge(1, 2);

// gh.addEdge(1, 3);

// gh.addEdge(1, 4);

// gh.addEdge(2, 3);

// gh.addEdge(3, 4);

// print the adjacency list representation of the above graph

// gh.printGraph();

return 0;

}

Screenshot

1. BFS

// 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(8);

g.addEdge(1, 2);

g.addEdge(1, 3);

    g.addEdge(2, 4);

   g.addEdge(2, 5);

   g.addEdge(2, 3);

   g.addEdge(3, 7);

   g.addEdge(3, 8);

   g.addEdge(4, 5);

   g.addEdge(5, 3);

   g.addEdge(5, 6);

   g.addEdge(7, 8);

    cout << "Following is Breadth First Traversal "

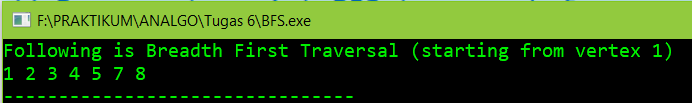
        << "(starting from vertex 1) \n";

    g.BFS(1);

    return 0;

}

Screenshot



Analisis :

* BFS merupakan metode pencarian secara melebar sehingga mengunjungi node dari kiri ke kanan di level yang sama. Apabila semua node pada suatu level sudah dikunjungi semua, maka akan berpindah ke level selanjutnya. Dalam worst case BFS harus mempertimbangkan semua jalur (path) untuk semua node yang mungkin, maka nilai kompleksitas waktu dari BFS adalah O( |V| + |E| ).
* Karena Big-O dari BFS adalah O(V+E) dimana V itu jumlah vertex dan E itu adalah jumlah edges maka Big-O = O(n) dimana n = v+e
* Maka dari itu Big-Ө nya adalah Ө(n).

1. DFS

// C++ program to print DFS traversal from

// a given vertex in a given graph

#include<iostream>

#include<list>

using namespace std;

// Graph 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;

// A recursive function used by DFS

void DFSUtil(int v, bool visited[]);

public:

Graph(int V); // Constructor

// 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);

};

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::DFSUtil(int v, bool visited[])

{

// 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])

DFSUtil(\*i, visited);

}

// DFS traversal of the vertices reachable from v.

// It uses recursive DFSUtil()

void Graph::DFS(int v)

{

// Mark all the vertices as not visited

bool \*visited = new bool[V];

for (int i = 0; i < V; i++)

visited[i] = false;

// Call the recursive helper function

// to print DFS traversal

DFSUtil(v, visited);

}

int main()

{

// Create a graph given in the above diagram

Graph g(8);

g.addEdge(1, 2);

g.addEdge(1, 3);

    g.addEdge(2, 4);

   g.addEdge(2, 5);

   g.addEdge(2, 3);

   g.addEdge(3, 7);

   g.addEdge(3, 8);

   g.addEdge(4, 5);

   g.addEdge(5, 3);

   g.addEdge(5, 6);

   g.addEdge(7, 8);

cout << "Following is Depth First Traversal"

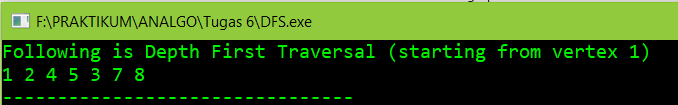
" (starting from vertex 1) \n";

g.DFS(1);

return 0;

}

Screenshot



Analisis :

* DFS merupakan metode pencarian mendalam, yang mengunjungi semua node dari yang terkiri lalu geser ke kanan hingga semua node dikunjungi. Kompleksitas ruang algoritma DFS adalah O(bm), karena kita hanya hanya perlu menyimpan satu buah lintasan tunggal dari akar sampai daun, ditambah dengan simpul-simpul saudara kandungnya yang belum dikembangkan.
* Big O

kompleksitas total DFS () adalah

(V + E).

O(n)

Dengan V = Jumlah Verteks

Dan E = Jumlah Edges