EXPERIMENT 12

- ASHISH KUMAR
- 2K18/SE/041

AIM:- Write a program for generating control flow graph and DD path graph and generate graph matrix.

THEORY:-

Control Flow Graph

In computer science, a control-flow graph (CFG) is a representation, using graph notation, of all paths that might be traversed through a program during its execution. In a control flow graph each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow. There are, in most presentations, two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves.

DD Path Graph

The Decision to Decision (DD) path graph is an extension of a program graph. It is widely known as DD path graph. There may be many nodes in a program graph which are in a sequence. When we enter into the first node of the sequence, we can exit only from the last node of that sequence. In DD path graph, such nodes which are in a sequence are combined into a block and are represented by a single node. Hence, the DD path graph is a directed graph in which nodes are sequences of statements and edges are control flow amongst the nodes. All programs have an entry and an exit and the corresponding program graph has a source node and a destination node. Similarly, the DD path graph also has a source node and a destination node.

Graph matrix

A graph matrix is a square matrix with one row and one column for every node of the graph. The size of the matrix (number of rows and number of columns) is the number of nodes of the graph. Graph matrix is the tabular representation of a program graph.

CODE (for generating CFG):-

```
#include<iostream>

#include<fstream>

#include<vector>

#include<queue>

#include<stack>

using namespace std;

// constants

const int NOTYPE = 0 , TYPE_REGULAR = 1, TYPE_IF = 2, TYPE_ELSE = 3,
TYPE_IF_CLOSE = 4, TYPE_ELSE_CLOSE = 5;

const string IF_STRING = "if(", ELSE_STRING = "else" , CLOSE_STRING = "}";

class Node

{

public:

    int num;

    static int node_count;

// list of ptrs to next nodes

    vector<Node*> next;

    static Node* create_new_node() {

        Node* new_node = new Node();

        new_node->num = Node::node_count++;

        return new_node;

    }

};
```

```
int Node::node_count = 1;

class Stack_Node {

public:

    Node* ptr;

    int type;

    Stack_Node() {

        this->ptr = NULL;

        this->type = NOTYPE;

    }

    Stack_Node(Node* ptr, int type) {

        this->ptr = ptr;

        this->type = type;

    }

};

stack<Stack_Node> mStack;

class Graph {

public:

    Node* head;

    void printGraph() {

        queue<Node*> q;

        q.push(head);

        while (!q.empty()) {

            Node* travel_ptr = q.front();

            q.pop();

            cout << travel_ptr->num << " -> ";
            for (std::vector<Node*>::iterator i = travel_ptr->next.begin(); i != travel_ptr->next.end();
                ++i)
            {
                cout << (*i)->num << " ";
                q.push((*i));
            }
           cout<<endl;
        }
        cout<<"END";
    }
};
void process_line(string line, Graph* g) {
    Node* current_node = NULL;
// if the line is not empty create a new node for this line
    if (line.find_first_not_of(' ') != string::npos)
    {
        current_node = Node::create_new_node();
    }
    if (g->head == NULL)
    {
        g->head = current_node;
        Stack_Node new_node(current_node, TYPE_REGULAR);
        mStack.push(new_node);
        return;
    }
```

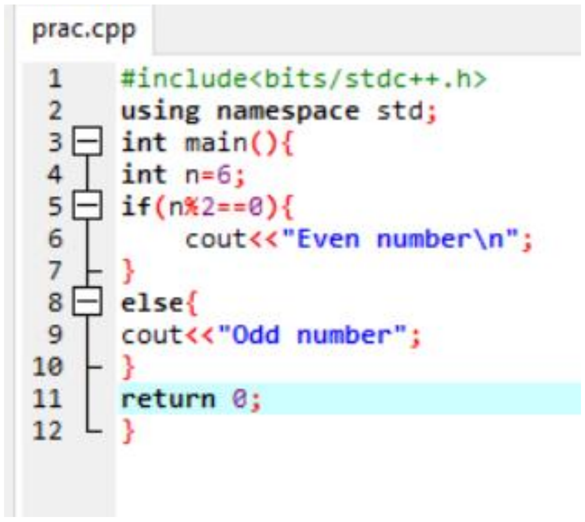
```
// ----------------------------- ATTACHMENT CODE ---------------------------------------------

    /*

    Check if the top of the stack is an 'ELSE_CLOSE' node , if so then attach the current node to
the else

    tail and pop the 'ELSE CLOSE AND ELSE' nodes from the stack. Then attach the IF CLOSE
to the current node

    and pop the IF CLOSE AND IF nodes from the stack */

    if (mStack.top().type == TYPE_ELSE_CLOSE)

    {

// Attach to the else node

        mStack.top().ptr->next.push_back(current_node);

// Pop ELSE CLOSE

        mStack.pop();

// Pop ELSE

        mStack.pop();

// Attach IF_CLOSE

        mStack.top().ptr->next.push_back(current_node);

// Pop IF_CLOSE

        mStack.pop();

// Pop IF

        mStack.pop();

    }

    else {

// if the current node is not an else node Attach like normal

        if (line.find(ELSE_STRING) == string::npos) {

            mStack.top().ptr->next.push_back(current_node);
        }
    }
// ---------------------------------- PROCESSING CODE -------------------------------------
// if line contains 'if' then
// we push the if node to the stack
    if (line.find(IF_STRING) != string::npos) {
// Push the if node to the stack
        Stack_Node new_node(current_node, TYPE_IF);
        mStack.push(new_node);
    }
// if the line contains 'else' we push the
// else node to the stack
    else if (line.find(ELSE_STRING) != string::npos) {
// Attach the current node to the last if
        Stack_Node node_if_close = mStack.top();
        mStack.pop();
        mStack.top().ptr->next.push_back(current_node);
        mStack.push(node_if_close);
        Stack_Node new_node(current_node, TYPE_ELSE);
        mStack.push(new_node);
    }
// if the line contains a '}' we check if it is a IF_CLOSE or an
// ELSE_CLOSE
    else if (line.find(CLOSE_STRING) != string::npos) {
// Check if the top of the stack is a regular node
```

```
    if (mStack.top().type == TYPE_REGULAR) {
        mStack.pop();
        if (!mStack.empty() && mStack.top().type == TYPE_IF) {
            Stack_Node new_node(current_node, TYPE_IF_CLOSE);
            mStack.push(new_node);
        } else if (!mStack.empty() && mStack.top().type == TYPE_ELSE) {
            Stack_Node new_node(current_node, TYPE_ELSE_CLOSE);
            mStack.push(new_node);
        } else {
            Stack_Node new_node(current_node , TYPE_REGULAR);
            mStack.push(new_node);
        }
    } else {
        if (mStack.top().type == TYPE_IF)
        {
            Stack_Node new_node(current_node, TYPE_IF_CLOSE);
            mStack.push(new_node);
        } else if (mStack.top().type == TYPE_ELSE) {
            Stack_Node new_node(current_node, TYPE_ELSE_CLOSE);
            mStack.push(new_node);
        }
    }
}
```

```
// if the line is a regular line we see if the top is a regular line and pop it and push this line
// else if the top node is IF_TYPE or ELSE_TYPE we simply push it
    else {
        if (mStack.top().type == TYPE_REGULAR) {
            mStack.pop();
            Stack_Node new_node(current_node, TYPE_REGULAR);
            mStack.push(new_node);
        }
        else if (mStack.top().type == TYPE_IF || mStack.top().type == TYPE_ELSE) {
            Stack_Node new_node(current_node, TYPE_REGULAR);
            mStack.push(new_node);
        }
    }
}
int main(int argc, char const *argv[])
{
    char filename[100];
    string line;
// input the filename
    cout << "Enter the filename : ";
    cin >> filename;
    Graph* g = new Graph();
    ifstream file(filename);
```

```
  if (file.is_open())
    {
// start reading the file
        while (getline(file, line))
        {
            process_line(line, g);
        }
    }
    g->printGraph();
    cout<<"\nControl Flow Graph(CFG) for the above file's code:\n\n";
    return 0;
}
```

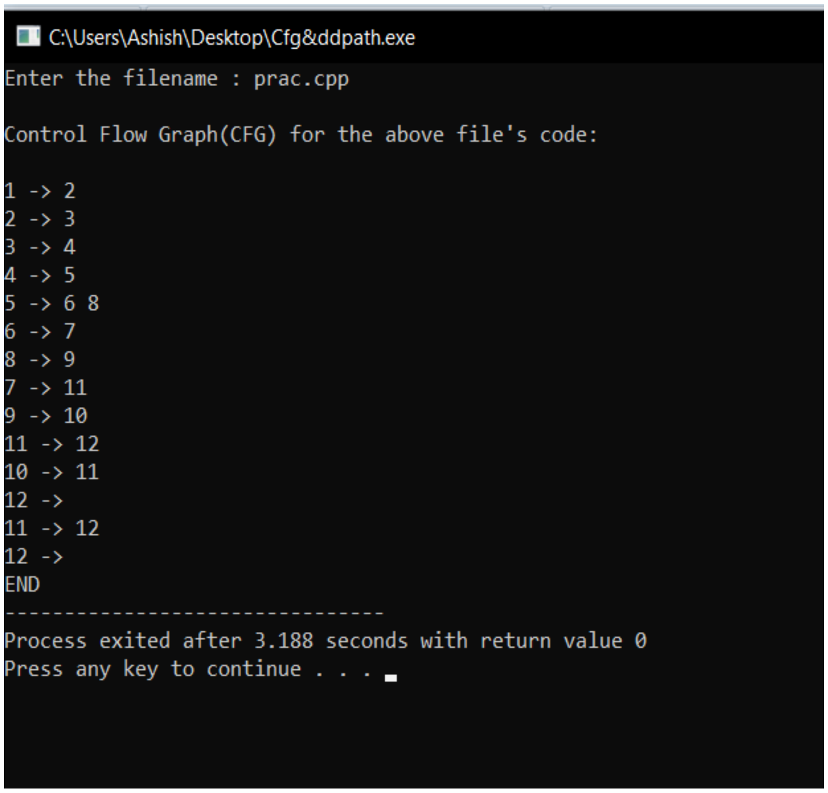
File code:-



prac.cpp

```
#include<bits/stdc++.h>
using namespace std;
int main(){
int n=6;
if(n%2==0){
	cout<<"Even number\n";
}
else{
cout<<"Odd number";
}
return 0;
}
```

OUTPUT:-



```
C:\Users\Ashish\Desktop\Cfg&ddpath.exe
Enter the filename : prac.cpp

Control Flow Graph(CFG) for the above file's code:

1 -> 2
2 -> 3
3 -> 4
4 -> 5
5 -> 6 8
6 -> 7
8 -> 9
7 -> 11
9 -> 10
11 -> 12
10 -> 11
12 ->
11 -> 12
12 ->
END
--------------------------------
Process exited after 3.188 seconds with return value 0
Press any key to continue . . .
```

Conlusion: We successfully generated control flow graph and DD path graph in this experiment. Since the control flow through programs is determined by the decisions, for example, if - then-else-constructs, based on the data and the conditions in such constructs, it is promising to keep in graphical representations of programs only the decisions and the control flow between them and thus defining a reduction of control flow graphs that preserves the branching structure. Decision graphs form an abstraction from control flow graphs that display only the decisions, for example, if-then-else-constructs and the paths between decisions to the programmer.

CODE (for generating graph matrix):-

```
#include <fstream>

#include <iostream>

#include <vector>

using namespace std;

class vertex{

 public:

 int in_deg;

 vector<int> e_list;

 int line_num;

 vertex(){

 in_deg = 0;

 }

 vertex(const vertex & v){

 in_deg = v.in_deg;

 for (int i=0;i<v.e_list.size();i++)

 e_list.push_back(v.e_list[i]);

 line_num = v.line_num;

 }

};

struct graph{

 int num_vertex;

 vector<vertex> v_list;

 void insert(const vertex & v){

 vertex v2 = v;
 num_vertex ++;
 v2.line_num = num_vertex;
 v_list.push_back(v2);
 }
 void erase(int i){
 num_vertex--;
 v_list.erase(v_list.begin() + i);
 }
 graph(const graph & cfg){
 num_vertex = cfg.num_vertex;
 for (int i=0;i<num_vertex;i++)
 v_list.push_back(cfg.v_list[i]);
 }
 graph(){
 num_vertex = 0;
 }
};
int buildDDHelper(graph & dd, int index, int & deleted){
 int i,prev_val;
 for (i=index;i<dd.num_vertex;){
 if (dd.v_list[i].in_deg == 1 &&
 dd.v_list[i].e_list.size() == 1){
 prev_val = i;
 i++;
 while (dd.v_list[i].in_deg == 1 && dd.v_list[i].e_list.size() == 1){
 int val = dd.v_list[i].e_list[0] - deleted - 2;
 dd.erase(i);
 deleted++;
 i = val;
 }
 dd.v_list[prev_val].e_list[0] = i + deleted + 1;
 }
 if (dd.v_list[i].e_list.size() > 1){
 buildDDHelper(dd,dd.v_list[i].e_list[0] - deleted - 1, deleted);
 i = buildDDHelper(dd,dd.v_list[i].e_list[1] - deleted - 1,deleted);
 }
 else
 break;
 }
 return i;
}
void buildDD(graph & dd){
 int deleted = 0;
 for (int i=0;i<dd.num_vertex;){
 if (dd.v_list[i].in_deg == 1 && dd.v_list[i].e_list.size() == 1){
 i++;
 while (dd.v_list[i].in_deg == 1 && dd.v_list[i].e_list.size() == 1){
 dd.erase(i);
 deleted ++;
 }
 dd.v_list[i-1].e_list[0] = dd.v_list[i].line_num;
 }
 if (dd.v_list[i].e_list.size() > 1){
 buildDDHelper(dd,dd.v_list[i].e_list[0] - deleted - 1,deleted);
 i = buildDDHelper(dd,dd.v_list[i].e_list[1] - deleted - 1,deleted);
 }
 else
 i++;
 }
}
void buildCFG(graph & cfg, ifstream & fin){
 char ar[100];
 do{
 fin.getline(ar,100);
 string str(ar);
 int pos = str.find_first_not_of(' ');
 if (pos == string::npos)
 continue;
 int prev_pos = cfg.num_vertex - 1;
 if (str.substr(pos,2) == "if"){
 vertex v;
 v.in_deg = 1;
 cfg.insert(v);
 p1:
```

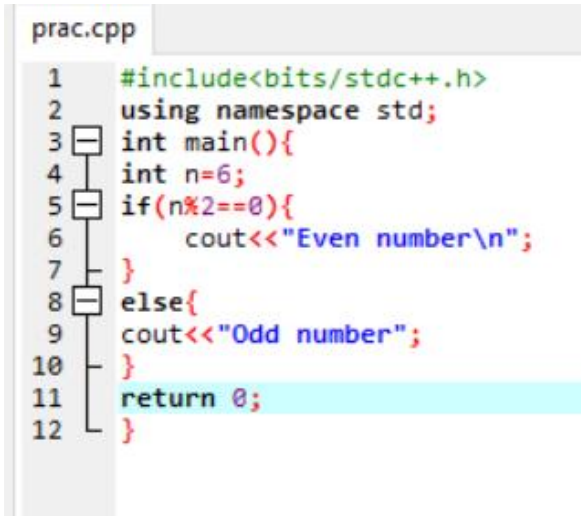
```
cfg.v_list[prev_pos].e_list.push_back(cfg.num_vertex);
buildCFG(cfg,fin);
int prev_pos2 = cfg.num_vertex - 1;
do{
fin.getline(ar,100);
string str(ar);
int pos = str.find_first_not_of(' ');
if (pos == string::npos)
continue;
if (str.substr(pos,4) == "else"){
vertex v;
v.in_deg = 1;
cfg.insert(v);
cfg.v_list[prev_pos + 1].e_list.push_back(cfg.num_vertex);
buildCFG(cfg,fin);
do{
fin.getline(ar,100);
string str(ar);
int pos = str.find_first_not_of(' ');
if (pos == string::npos)
continue;
vertex v2;
v2.in_deg = 2;
cfg.insert(v2);
cfg.v_list[cfg.num_vertex -2].e_list.push_back(cfg.num_vertex);
cfg.v_list[prev_pos2].e_list.push_back(cfg.num_vertex);
if (str[pos] == '}')
return;
break;
}
while (true);
}
else{
vertex v2;
v2.in_deg = 2;
cfg.insert(v2);
cfg.v_list[cfg.num_vertex -2].e_list.push_back(cfg.num_vertex);
cfg.v_list[prev_pos2 - 2].e_list.push_back(cfg.num_vertex);
if (str.substr(pos,2) == "if")
goto p1;
else if (str[pos] == '}')
return;
}
break;
}
while (true);
}
else{
vertex v;
v.in_deg = 1;
```

```
 cfg.insert(v);
 cfg.v_list[prev_pos].e_list.push_back(cfg.num_vertex);
 if (str[pos] == '}')
 break;
 }
 }while (true);
}
void printGraph(const graph & cfg){
 for (int i=0;i<cfg.num_vertex;i++){
 cout << cfg.v_list[i].line_num << ": Indegree: "<<cfg.v_list[i].in_deg<<" Edge List-> ";
 int size_edge = cfg.v_list[i].e_list.size();
 for (int j=0;j<size_edge;j++)
 cout << cfg.v_list[i].e_list[j] << " ";
 cout << endl;
 }
}
void graph_matrix(const graph & cfg){
 int size = cfg.num_vertex;
 int i,j,k,num;
 int ar1[size];
 int arr[size][size];
 for(i=0;i<size;i++){
 ar1[i] = cfg.v_list[i].line_num;
 }
```

```
for(i=0;i<size;i++){
for(j=0;j<size;j++){
arr[i][j]=0;
}
}
for(i=0;i<size;i++){
int size_edge = cfg.v_list[i].e_list.size();
for (int j=0;j<size_edge;j++){
num= cfg.v_list[i].e_list[j];
for(k=0;k<size;k++){
if(ar1[k]==num){
arr[i][k]=1;
break;
}
}
}
}
cout<<" ";
for(i=0;i<size;i++){
cout<<" "<<i;
 }
 int count=2;
printf("\n");
for(i=0;i<size;i++){
cout<<i<<" ";
```

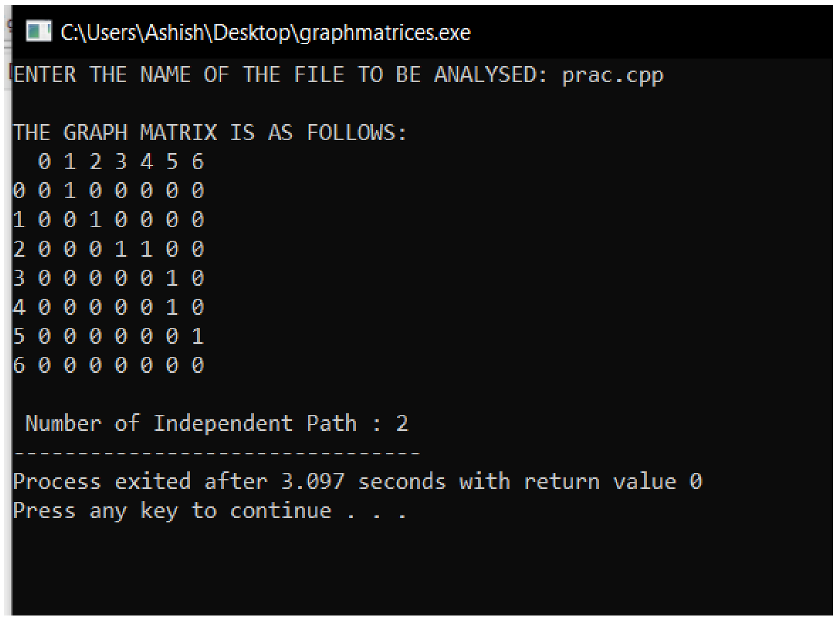
```
 for(j=0;j<size;j++){
 printf("%d ",arr[i][j]);
 if(arr[i][j]==1)
 count++;
 }
 count--;
 printf("\n");
 }
 printf("\n Number of Independent Path : %d",count);
}

int main(){
 char file_name[100];
 cout << "ENTER THE NAME OF THE FILE TO BE ANALYSED: ";
 cin.getline(file_name,100);
 ifstream fin(file_name);
 if (!fin){
 cout << "FILE NOT FOUND!\n";
 }
 else{
 char ar[100];
 bool terminate = false;
 while (!terminate){
 fin.getline(ar,100 - 1);
 string str(ar);
int pos = str.find_first_not_of(' ');
if (str[pos] == '#')
continue;
if (str.find("main()") != string::npos){
graph cfg;
vertex v;
cfg.insert(v);
buildCFG(cfg,fin);
cout << "\nTHE GRAPH MATRIX IS AS FOLLOWS: \n";
graph dd(cfg);
buildDD(dd);
graph_matrix(dd);
terminate = true;
}
}
}
return 0;
}
```

File code:-



```
prac.cpp
#include<bits/stdc++.h>
using namespace std;
int main(){
int n=6;
if(n%2==0){
	cout<<"Even number\n";
}
else{
cout<<"Odd number";
}
return 0;
}
```

OUTPUT:-



```
C:\Users\Ashish\Desktop\graphmatrices.exe
ENTER THE NAME OF THE FILE TO BE ANALYSED: prac.cpp

THE GRAPH MATRIX IS AS FOLLOWS:
  0 1 2 3 4 5 6
0 0 1 0 0 0 0 0
1 0 0 1 0 0 0 0
2 0 0 0 1 1 0 0
3 0 0 0 0 0 1 0
4 0 0 0 0 0 1 0
5 0 0 0 0 0 0 1
6 0 0 0 0 0 0 0

 Number of Independent Path : 2
--------------------------------
Process exited after 3.097 seconds with return value 0
Press any key to continue . . .
```

Conlusion: We successfully generated graph matrix in this experiment by converting a flow graph into matrix and each node corresponds to particular row & column, by applying step by step process to get desired paths with its calculated values.