DSA Lab
Name: Siddhi Parekh
Reg No.: 221071047
Batch: C
SY Comps

Experiment No.: 10

Aim:
Create a graph and check whether the created graph has a cycle or not.

Theory:
To find cycles in a directed graph we can use the Depth First Traversal (DFS) technique. It is based on the idea that there is a cycle in a graph only if there is a back edge [i.e., a node points to one of its ancestors] present in the graph.
To detect a back edge, we need to keep track of the nodes visited till now and the nodes that are in the current recursion stack [i.e., the current path that we are visiting]. If during recursion, we reach a node that is already in the recursion stack, there is a cycle present in the graph.

• Nodes: The graph consists of 5 nodes labeled A, B, C, D, and E. Nodes are represented as blue circles.
• Edges: The edges, represented as black lines, connect the nodes in the following way:
  ◦ A is connected to B and C
  ◦ B is connected to D
  ◦ C is connected to D and E
  ◦ E is connected to D
• Cycle: The graph has a cycle. This means that it is possible to start at one node and traverse the graph in such a way that you end up back at the starting node without revisiting any node. In this case, you could start at node A, move to B, then D, then E, and finally back to A to form a cycle.

Algorithm:

1. Create a visited array to keep track of visited nodes.
2. For each unvisited node in the graph:
    a. Do DFS traversal from the current node.
    b. During the traversal, if you encounter an adjacent node that is already visited and is not the parent of the current node, then a cycle exists.
3. If a cycle is detected during any DFS traversal, the graph contains a cycle.
4. If all nodes are visited and no cycles are detected, the graph does not contain a cycle.

Example:

1. Recursion Stack and Visited Array:

   Let's consider the graph created in the main function with edges (0-1), (0-2), (0-3), (1-2), and (3-4). Here's how the recursion stack and visited array change as the DFS progresses:

   - Start DFS at vertex 0: Recursion Stack = [0], Visited = [true, false, false, false, false]
   - Visit vertex 1 from vertex 0: Recursion Stack = [0, 1], Visited = [true, true, false, false, false]

     As 0 is already visited and is a parent of 1 no cycle is found.
   - Visit vertex 2 from vertex 1: Recursion Stack = [0, 1, 2], Visited = [true, true, true, false, false]

     As 1 is already visited and is a parent of 2 no cycle is found.
     Vertex 2 has an adjacent vertex 0, but 0 not the parent of 2 in the DFS tree, a cycle is detected. So we return from the recursive calls.

CONCLUSION:

We can successfully detect cycles in an undirected graph using DFS in linear time complexity of O(V + E), where V is the number of vertices and E is the number of edges in the graph. The code also uses a simple data structure of linked lists to represent the graph as an adjacency list. The code can be tested with different graphs by changing the number of vertices, the edges, and the starting vertex in the main function.

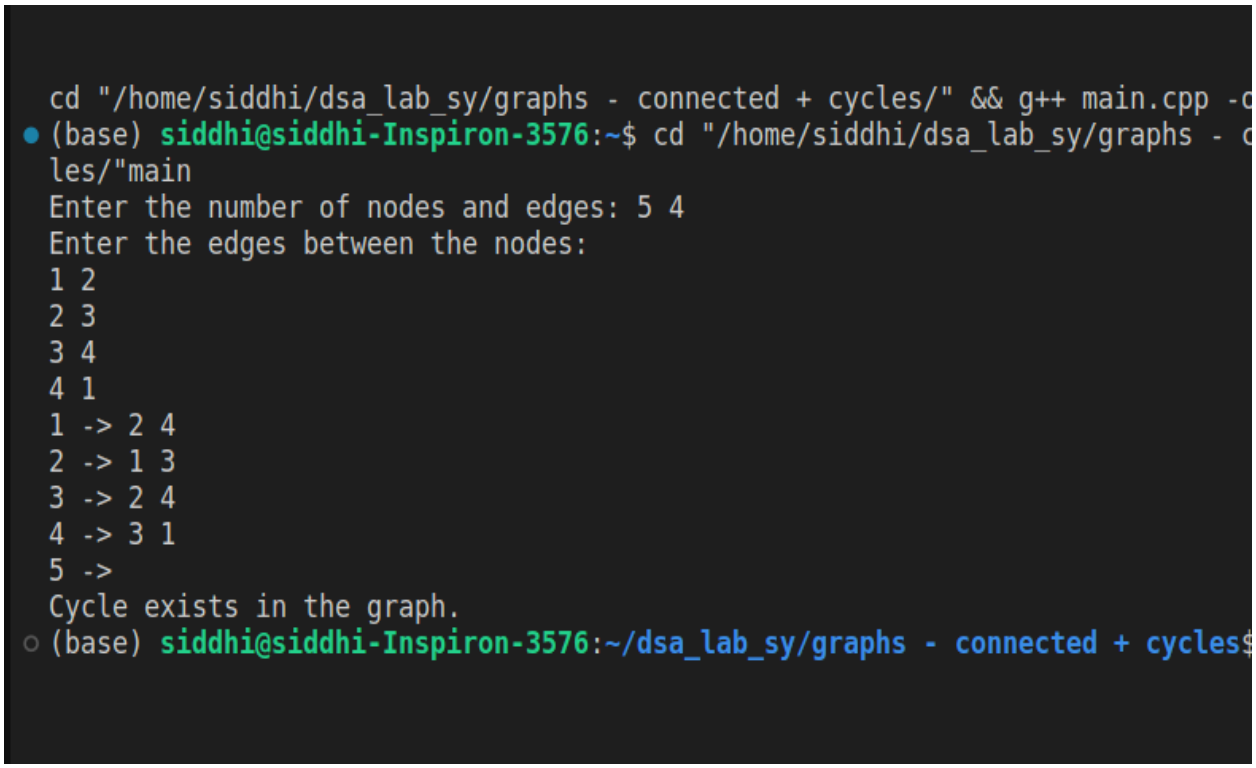
CODE:

```
#include <iostream>
#include <vector>

using namespace std;

void createList(vector<int> adjList[], int e)
{
	cout << "Enter the edges between the nodes:" << endl;
	for (int i = 1; i <= e; i++)
	{
	int u, v;
	cin >> u >> v;
	adjList[v].push_back(u);
	adjList[u].push_back(v);
	}
}

void display(vector<int> adj[], int n)
{
	for (int i = 1; i <= n; i++)
	{
	cout << i << " -> ";

	for (auto x : adj[i])
	{
```

```
        cout << x << " ";
        }

        cout << endl;
        }
}

bool isCycleUtil(vector<int> adj[], int v, vector<bool> &visited, int parent)
{
        visited[v] = true;

        for (auto i : adj[v])
        {
        if (!visited[i])
        {
        if (isCycleUtil(adj, i, visited, v))
                return true;
        }
        else if (i != parent)
        {
        // If the adjacent node is visited and not the parent of the current node,
        // then a cycle exists.
        return true;
        }
        }

        return false;
}

bool isCycle(vector<int> adj[], int n)
{
        vector<bool> visited(n + 1, false);

        for (int i = 1; i <= n; i++)
        {
        if (!visited[i])
        {
```

```
        if (isCycleUtil(adj, i, visited, -1))
                return true;
        }
        }

        return false;
}

int main()
{
        int n, e;
        cout << "Enter the number of nodes and edges: ";
        cin >> n >> e;

        vector<int> adjList[n + 1];

        createList(adjList, e);
        display(adjList, n);

        if (isCycle(adjList, n))
        {
        cout << "Cycle exists in the graph." << endl;
        }
        else
        {
        cout << "No cycle in the graph." << endl;
        }
        return 0;
}
```

OUTPUT:


cd "/home/siddhi/dsa_lab_sy/graphs - connected + cycles/" && g++ main.cpp -o
(base) siddhi@siddhi-Inspiron-3576:~$ cd "/home/siddhi/dsa_lab_sy/graphs - c
les/"main
Enter the number of nodes and edges: 5 4
Enter the edges between the nodes:
1 2
2 3
3 4
4 1
1 -> 2 4
2 -> 1 3
3 -> 2 4
4 -> 3 1
5 ->
Cycle exists in the graph.
(base) siddhi@siddhi-Inspiron-3576:~/dsa_lab_sy/graphs - connected + cycles$