Assignment 6

Name: Diptangshu Dey

Roll No: 20CS8018

Code:

```c
// Name - Diptangshu Dey
// Roll No.- 20CS8018

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 20
#define E 80

typedef struct _node
{
    int vno;
    struct _node *next;
} node;

typedef node **graph;

/* Add the directed edge (u,v) to the graph G. Return 1 on sucess,
   0 on failure (if the edge already exists in G). */
int insertDirectedEdge(graph G, int u, int v)
{
    node *p, *q;

    /* Locate insertion position. The neighbors are kept sorted, although
       this is not very necessary.*/
    p = G[u];
    while (p->next)
    {
        if (p->next->vno == v)
            return 0; /* Edge already exists */
        if (p->next->vno > v)
            break;   /* Insertion location found */
        p = p->next; /* Continue search */
    }
    /* Insert the new neighbor after (*p) */
    q = (node *)malloc(sizeof(node));
    q->vno = v;
    q->next = p->next;
    p->next = q;
    return 1;
}
```

```c
/* Insert the undirected edge (u,v) to G. This essentially makes two
   directed edge insertions: (u,v) and (v,u). */
int insertEdge(graph G, int u, int v)
{
    int c;

    c = insertDirectedEdge(G, u, v);
    c += insertDirectedEdge(G, v, u);
    if (c == 0)
        return 0;
    if (c == 2)
        return 1;
    fprintf(stderr, "*** Error in insertEdge()\n");
    return -1;
}

/* Create a random graph with n nodes and e edges. The returned graph
   need not be connected. */
graph createGraph(int n, int e)
{
    graph G;
    int u, v, m;

    if (e > n * (n - 1) / 2)
    {
        fprintf(stderr, "*** Too many edges in createGraph\n");
        exit(1);
    }

    /* Create n list headers */
    G = (node **)malloc(n * sizeof(node *));

    /* Initialize each list to a linked list of one node. A dummy node is
       always maintained at the beginning of each list. This simplifies
       the edge-insertion procedure.*/
    for (u = 0; u < n; ++u)
    {
        G[u] = (node *)malloc(sizeof(node));
        G[u]->next = NULL;
    }
    m = 0; /* m stores the count of edges added to G */
    while (m < e)
    {
        u = rand() % n;
        v = rand() % n; /* Randomly choose vertices */
        if (u == v)
            continue;             /* Loops are not allowed */
        m += insertEdge(G, u, v); /* Successful insertion increments m */
    }
    return G;
}

/* Function to print the neighbors of each node */
void printGraph(graph G, int n)
{
    int u;
    node *p;
```

```
    printf("+++ n = %d\n+++ Neighbor list:\n", n);
    for (u = 0; u < n; ++u)
    {
        printf("%5d :", u);
        p = G[u]->next; /* Skip the initial dummy node */
        while (p)
        { /* Standard linked-list traversal */
            printf("%3d", p->vno);
            p = p->next;
        }
        printf("\n");
    }
}
int dfsB(graph G, int u, int *dfsLevel)
{
    node *p;
    int v, c;

    printf("%3d", u); /* DFS listing */
    c = 0;            /* c stores the number of backward and forward edges from u
*/
    p = G[u]->next;   /* Skip the dummy node */
    while (p)
    {
        v = p->vno;
        if (dfsLevel[v] == -1)
        {                                   /* unvisited vertex */
            dfsLevel[v] = dfsLevel[u] + 1; /* v is a child of u in the DFS tree
*/
            c += dfsB(G, v, dfsLevel);     /* Recursive call */
        }
        else
        {                                              /* (u,v) is either a
backward or a forward edge */
            if ((dfsLevel[u] - dfsLevel[v]) % 2 == 0) /* odd cycle detected */
                return 1;
        }
        p = p->next;
    }
    return c;
}

/* The wrapper function for checking whether G is bipartite */
int isBipartite(graph G, int n)
{
    int *dfsLevel, u, c;

    printf("+++ Running DFS\n  ");
    dfsLevel = (int *)malloc(n * sizeof(int));
    for (u = 0; u < n; ++u)
        dfsLevel[u] = -1; /* Initialize all levels to -1 */

    /* G is not necessarily connected, so multiple DFS may be needed.
       Every time, we start a new DFS from the vertex u. */
    u = 0;
    c = 0;
    while (u < n)
    {
```

```
        if (dfsLevel[u] == -1)
        {                                /* unvisited vertex located */
            dfsLevel[u] = 0;             /* DFS with u as root */
            c += dfsB(G, u, dfsLevel); /* Accumulate in c the number of backward
                                          and forward edges leading to odd-length
                                          cycles */
        }
        ++u;
    }
    printf("\n");
    free(dfsLevel);
    return (c == 0); /* G is bipartite if and only if it contains no cycles
                        of odd length */
}

int dfsC(graph G, int u, int *dfsLevel, int *dfsNumber, int *dfsLow, int cnt, int
atRoot)
{
    node *p;
    int v, nc;

    /* Initialize dfsnumber and dfslow for u to the current cnt value */
    dfsLow[u] = dfsNumber[u] = cnt++;

    /* nc is used to store the number of children of u in the DFS tree.
       Needed only if the flag atRoot is true. */
    nc = 0;

    p = G[u]->next; /* Discard the dummy node at the beginning */
    while (p)
    {
        v = p->vno;
        if (dfsLevel[v] == -1)
        { /* unvisited vertex located */
            dfsLevel[v] = dfsLevel[u] + 1;
            ++nc;                                                    /* v will be
child of u in DFS tree */
            cnt = dfsC(G, v, dfsLevel, dfsNumber, dfsLow, cnt, 0); /* Recursive
call */
            if (dfsLow[v] < dfsLow[u])
                dfsLow[u] = dfsLow[v]; /* Minimum over child nodes */

            if ((!atRoot) && (dfsLow[v] >= dfsNumber[u]))
                printf("%5d disconnects %d\n", u, v);
        }
        else if (dfsLevel[u] >= dfsLevel[v] + 2)
        { /* Back edge */
            /* Now (u,v) is a back edge. From u, there is an escape route
               to v without using the DFS tree edges. Record this in
               low[u]. */
            if (dfsNumber[v] < dfsLow[u])
                dfsLow[u] = dfsNumber[v];
        }
        p = p->next;
    }

    if ((atRoot) && (nc > 1))
        printf("%5d has %d children\n", u, nc);
```

```
    return cnt;
}

/* The wrapper function for locating all cut vertices in G */
void findCutVertices(graph G, int n)
{
    int *dfsLevel, *dfsNumber, *dfsLow, cnt, u;

    dfsLevel = (int *)malloc(n * sizeof(int));
    dfsNumber = (int *)malloc(n * sizeof(int));
    dfsLow = (int *)malloc(n * sizeof(int));

    /* Initialize levels and numbers to -1 (implying unvisited) */
    for (u = 0; u < n; ++u)
        dfsLevel[u] = dfsNumber[u] = -1;

    printf("+++ The cut vertices of G are:\n");
    u = cnt = 0;
    while (u < n)
    { /* Run multiple DFS */
        if (dfsLevel[u] == -1)
        {
            /* u is the root for this DFS instance */
            dfsLevel[u] = 0;
            cnt = dfsC(G, u, dfsLevel, dfsNumber, dfsLow, cnt, 1);
        }
        ++u;
    }
    free(dfsLevel);
    free(dfsNumber);
    free(dfsLow);
}

int main(int argc, char *argv[])
{
    int n, e;
    graph G;
    srand((unsigned int)time(NULL));
    n = (argc >= 3) ? atoi(argv[1]) : N;
    e = (argc >= 3) ? atoi(argv[2]) : E;
    G = createGraph(n, e);
    printGraph(G, n);
    printf("The graph is%sbipartite\n", isBipartite(G, n) ? " " : " not ");
    findCutVertices(G, n);
    exit(0);
}
```

Output:

```
❯ ./a.out
+++ n = 20
+++ Neighbor list:
    0 :  1  3  5  7  8  9 12 13 14 16 18 19
    1 :  0  2  5  6  9 10 12 13 14 15 16 18
    2 :  1  3  5 11 13 15
    3 :  0  2  7  8 10 11 13 15 16 17 18 19
    4 :  5  7 11 12 14
    5 :  0  1  2  4 10 18
    6 :  1  7 11 14 15 16 19
    7 :  0  3  4  6  8 13 15 17
    8 :  0  3  7  9 12 16
    9 :  0  1  8 15 17
   10 :  1  3  5 15 19
   11 :  2  3  4  6 12 13 17
   12 :  0  1  4  8 11 13 14 16 17 19
   13 :  0  1  2  3  7 11 12 14 16 18 19
   14 :  0  1  4  6 12 13 16 17
   15 :  1  2  3  6  7  9 10 16 17
   16 :  0  1  3  6  8 12 13 14 15 19
   17 :  3  7  9 11 12 14 15 18
   18 :  0  1  3  5 13 17
   19 :  0  3  6 10 12 13 16
+++ Running DFS
    0  1  2  3  7  8 10  5 15 19 11 13  4  6  9 12 14 16 17 18
The graph is not bipartite
+++ The cut vertices of G are:
```