가톨릭대학교 THE CATHOLIC UNIVERSITY OF KOREA Computer SCIENCE

알고리즘 설계 - 201921725 안성현

홀수 사이클

<2022/05/30>

문제 소개 및 접근법

1-1] 홀수 사이클 문제

① 문제

홀수 사이클은 길이가 홀수인 사이클, 에지의 개수가 홀수인 사이클을 말한다. 입력으로 주어진 단순 방향 그래프에 홀수 사이클이 존재하는지 아닌지를 판별하는 효율적인 프로그램을 작성하라. 첫째 줄에 방향 그래프 G의 정점과 에지 개수를 나타내는 양의 정수 n, m이 입력된다. 단, n<=100,000이고 m<=1,000,000이다. 방향 그래프의 정점은 1부터 n까지 번호가 매겨져 있다. 둘째 줄부터 마지막 줄까지 m개의 줄에는 한 줄에 하나씩 G의 방향 에지 (u,v)를 나타내는 두 양의 정수 u,v가 입력된다.

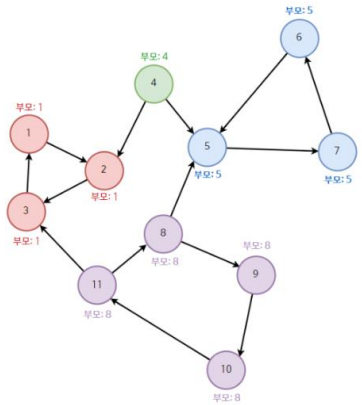
(예시1) 3 3 2 3 2 1 1 3 -> -1 **(예시2)** 3 4 2 1 2 3 1 3 3 2 -> 1

② 접근법 (소스 간략 설명)

▶ (아래 방법은 간단한 풀이 방법이지만 효율적이지 않다)

1. 방향 그래프 G에서 SCC(강한 연결 성분)를 찾는다. SCC는 Tajan 알고리즘을 이용해서 구할 수 있다.

2. SCC 각각을 무방향 그래프로 간주한다. 예를 들어 아래 그림은 4개의 SCC를 가진 방향 그래프이다. SCC 안쪽의 방향 에지를 무방향 에지로 교체한다. 결국 4개의 무방향 그래프가 생성된다.



3. 2번에서 구한 무방향 그래프가 모두 이분 그래프(Bipartite)이면 방향 그래프 G는 홀수 사이클이 아니다. 그렇지 않고, 한 개라도 이분 그래프(Bipartite)가 아니라면 G는 홀수 사이클이다. 이분 그래프를 판정할 때는 DFS를 이용해서 두 가지 색으로 정점 채색을 하고 인접한 정점이 같은 색인지 확인하면 된다.

이분 그래프(Bipartite Graph): 정점의 집합을 둘로 분할하여, 각 집합에 속한 정점끼리는 서로 인접하지 않도록 분할할 수 있는 그래프

이 문제는 이분 그래프와 홀수 사이클의 관계로 해결하는 문제이다.
무방향 그래프가 이분 그래프라는 것과 홀수 사이클이 존재하지 않는다는 것은 동치이다.
방향 그래프에서 문제를 풀 때는 SCC 각각을 무방향 그래프로 간주하고 풀면 된다.

소스 코드와 실행 결과

2-1] 소스 코드와 실행 결과

① 코드

```
#include <iostream>
#include <vector>
#include <stack>
#include <algorithm>
#define MAX_VERTICES 20000
#define INIT_VALUE -1
#define RED 1
#define BLUE 2

using namespace std;


/*
   그래프 - 인접 행렬 버전
*/

typedef struct GraphType {
        int n; // 정점의 개수
        int adj_mat[MAX_VERTICES][MAX_VERTICES];
}GraphType;


// 그래프 초기화
void init(GraphType* g) {
        int r, c;
        g->n = 0;
        for (r = 0; r < MAX_VERTICES; r++) {
                for (c = 0; c < MAX_VERTICES; c++)
                        g->adj_mat[r][c] = 0;
        }
}

// 정점 삽입 연산
void insert_vertex(GraphType* g, int v) {
        if ((g->n) + 1 > MAX_VERTICES) {
                fprintf(stderr, "그래프: 정점의 개수 초과");
                return;
        }
        g->n++;
}

// 간선 삽입 연산, v를 u의 인접 리스트에 삽입한다.
void insert_edge(GraphType* g, int start, int end) {
```

```
        if (start >= g->n || end >= g->n) {
                fprintf(stderr, "그래프: 정점 번호 오류");
                return;
        }
        g->adj_mat[start][end] = 1;
}


/*
   그래프 - 인접 리스트 버전
*/

typedef struct GraphNode {
        int vertex;
        struct GraphNode* link;
}GraphNode;

typedef struct GraphType2 {
        int n; // 정점의 개수
        GraphNode* adj_list[MAX_VERTICES];
}GraphType2;


// 그래프 초기화
void init2(GraphType2* g) {
        int v;
        g->n = 0;
        for (v = 0; v < MAX_VERTICES; v++)
                g->adj_list[v] = NULL;
}

// 정점 삽입 연산
void insert_vertex2(GraphType2* g, int v) {
        if ((g->n) + 1 > MAX_VERTICES) {
                fprintf(stderr, "그래프: 정점의 개수 초과");
                return;
        }
        g->n++;
}

// 간선 삽입 연산, v를 u의 인접 리스트에 삽입한다.
void insert_edge2(GraphType2* g, int u, int v) {
        GraphNode* node;
        if (u >= g->n || v >= g->n) {
                fprintf(stderr, "그래프: 정점 번호 오류");
                return;
        }
        if (g->adj_list[u] == NULL) { // insert_first
                node = (GraphNode*)malloc(sizeof(GraphNode));
                node->vertex = v;
                node->link = g->adj_list[u];
                g->adj_list[u] = node;
        }
```

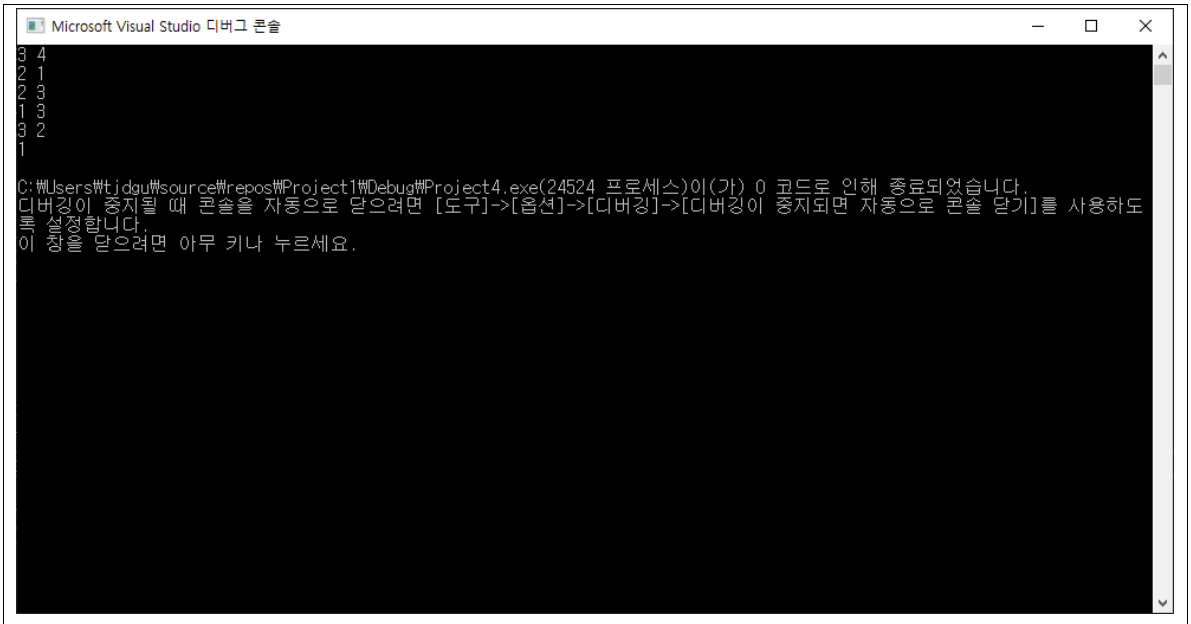
```
        else { //insert_last
                node = g->adj_list[u];
                while (node->link != NULL)
                        node = node->link;
                node->link = (GraphNode*)malloc(sizeof(GraphNode));
                node->link->vertex = v;
                node->link->link = NULL;
        }
}


/*
   알고리즘1 - SCC
*/

int id, VID[MAX_VERTICES];
bool finished[MAX_VERTICES];
vector<vector<int>> scc_list;
stack<int> s;

// DFS for SCC
int DFS1(GraphType* g, int v) {
        // 스택에 저장
        s.push(v);

        // 정점에 고유의 ID 부여
        VID[v] = ++id;

        // 부모를 자신 ID로 설정
        int parent = VID[v];

        // DFS로 SCC 구성
        for (int w = 0; w < g->n; w++) {
                if (g->adj_mat[v][w]) {
                        int next = w;
                        // 방문하지 않은 정점
                        if (!VID[next])
                                parent = min(parent, DFS1(g, next));
                        // 처리중인 정점
                        else if (!finished[next])
                                parent = min(parent, VID[next]);
                }
        }

        // 부모가 자신인 경우
        if (parent == VID[v]) {
                vector<int> scc;
                int top_v = INIT_VALUE;

                // 스택에서 빼고 scc배열에 저장
                while (top_v != v) {
                        top_v = s.top();
                        s.pop();
```

```
                        scc.push_back(top_v);
                        finished[top_v] = true;
                }
                // scc를 scc_list에 저장
                scc_list.push_back(scc);
        }

        return parent;
}

/*
   알고리즘2 - Bipartite Graph
*/

// DFS for Coloring
int* visited;
void DFS2(GraphType2* g, int v) {
        // 처음 정점은 빨간 색으로 칠함
        if (!visited[v])
                visited[v] = RED;

        for (GraphNode* w = g->adj_list[v]; w; w = w->link) {
                // 아직 방문하지 않은 정점이면 현재 정점과 반대 색으로 칠함
                if (!visited[w->vertex]) {
                        if (visited[v] == RED)
                                visited[w->vertex] = BLUE;
                        else if (visited[v] == BLUE)
                                visited[w->vertex] = RED;
                        DFS2(g, w->vertex);
                }
        }
}

// 이분 그래프 체크
bool Bipartite(GraphType2* g) {
        // 모든 정점들을 순회하면서 인접한 정점과의 색이 같은지 확인
        for (int i = 0; i < g->n; i++) {
                GraphNode* p = g->adj_list[i];
                int curr = i;
                while (p != NULL) {
                        int next = p->vertex;
                        // check bipartite
                        if (visited[curr] == visited[next])
                                return false;
                        // update p
                        p = p->link;
                }
        }
        return true;
}

// find OddCycle
int oddCycle(GraphType* g) {
```

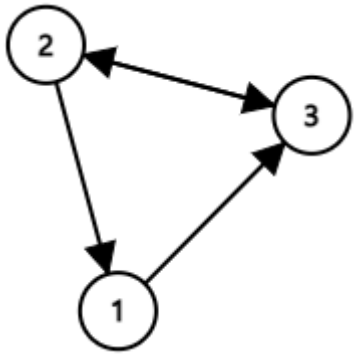
```
        // scc 별 Bipartite 찾기
        for (int i = 0; i < scc_list.size(); i++) {

                // make small graph
                GraphType2* sg;
                sg = (GraphType2*)malloc(sizeof(GraphType2));
                init2(sg);

                // scc
                vector<int> list = scc_list[i];

                // init visited
                visited = new int[list.size()]{};

                // add vertex
                for (int j = 0; j < list.size(); j++)
                        insert_vertex2(sg, j);

                // add edge
                for (int k = 0; k < list.size(); k++) {
                        for (int h = k + 1; h < list.size(); h++) {
                                int val1 = list[k];
                                int val2 = list[h];

                                if (g->adj_mat[val1][val2] || g->adj_mat[val2][val1]) {
                                        insert_edge2(sg, k, h);
                                        insert_edge2(sg, h, k);
                                }
                        }
                }

                // find Bipartite
                DFS2(sg, 0);

                bool output = Bipartite(sg);
                // Bipartite -> no odd cycle
                if (output)
                        continue;
                // no Bipartite -> odd cycle
                else
                        return true;

                // delete
                free(sg);
                delete[] visited;
        }

        // 모든 scc가 Bipartite
        return -1;
}

int main(void) {
        GraphType* g;
```

```
        g = (GraphType*)malloc(sizeof(GraphType));
        init(g);

        // 정점, 에지 개수 입력
        int n, m;
        cin >> n >> m;

        // 정점 삽입
        for (int i = 0; i < n; i++)
                insert_vertex(g, i);

        // 에지 삽입
        for (int i = 0; i < m; i++) {
                int u, v; // vertex(u) -> vertex(v)
                cin >> u >> v;
                // 정점이 0부터 시작하므로 1씩 제거
                u = u - 1;
                v = v - 1;
                insert_edge(g, u, v);
        }

        // SCC 찾기
        for (int i = 0; i < n; i++) {
                if (VID[i] == 0)
                        DFS1(g, i);
        }

        // OddCycle 찾기
        int rst = oddCycle(g);
        printf("%d₩n", rst);

        free(g);
        return 0;
}
```

② Visual Studio 2017 실행 결과 (1)

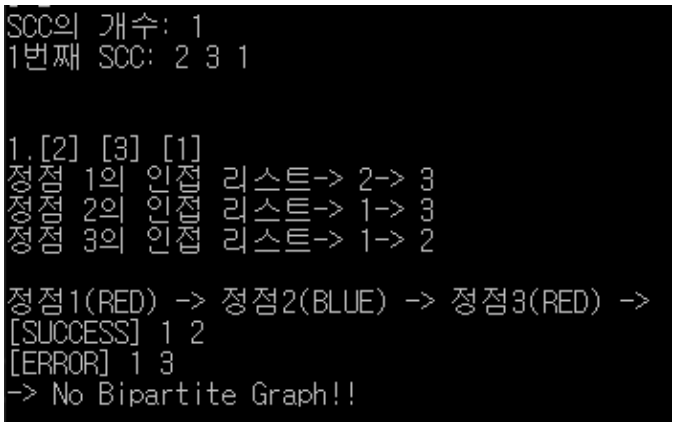


입력: 3 4(정점 개수, 에지 개수) 2 1 2 3 1 3 3 2(에지)

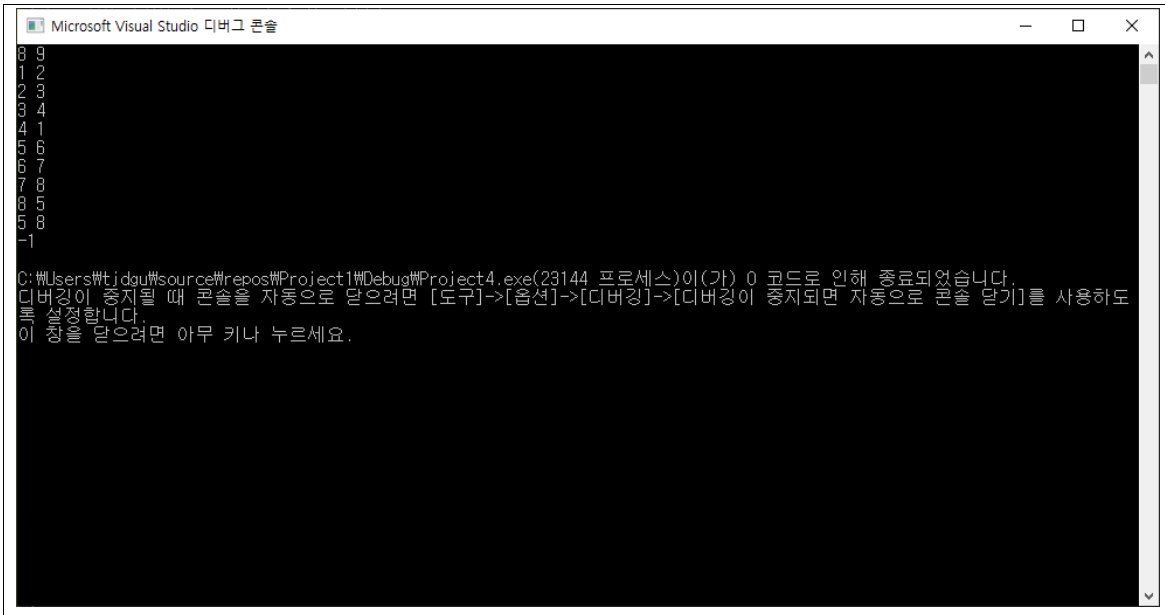


출력: 1 (홀수 사이클)

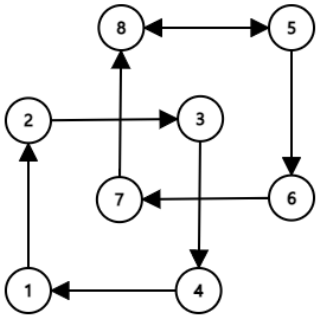
분석: SCC 1개, 무방향 그래프가 이분 그래프가 아님 -> 홀수 사이클



② Visual Studio 2017 실행 결과 (2)



입력: 8 9(정점 개수, 에지 개수) 1 2 2 3 3 4 4 1 5 6 6 7 7 8 8 5 5 8(에지)



출력: -1 (홀수 사이클이 아님)

분석: SCC 2개, 모든 무방향 그래프가 이분 그래프 -> 홀수 사이클이 아님

```
1.[4] [3] [2] [1]
정점 1의 인접 리스트-> 2-> 4
정점 2의 인접 리스트-> 1-> 3
정점 3의 인접 리스트-> 2-> 4
정점 4의 인접 리스트-> 1-> 3

정점1(RED) -> 정점2(BLUE) -> 정점3(RED) -> 정점4(BLUE) ->
[SUCCESS] 1 2
[SUCCESS] 1 4
[SUCCESS] 2 1
[SUCCESS] 2 3
[SUCCESS] 3 2
[SUCCESS] 3 4
[SUCCESS] 4 1
[SUCCESS] 4 3
-> Bipartite Graph!!
```

```
2.[8] [7] [6] [5]
정점 1의 인접 리스트-> 2-> 4
정점 2의 인접 리스트-> 1-> 3
정점 3의 인접 리스트-> 2-> 4
정점 4의 인접 리스트-> 1-> 3

정점1(RED) -> 정점2(BLUE) -> 정점3(RED) -> 정점4(BLUE) ->
[SUCCESS] 1 2
[SUCCESS] 1 4
[SUCCESS] 2 1
[SUCCESS] 2 3
[SUCCESS] 3 2
[SUCCESS] 3 4
[SUCCESS] 4 1
[SUCCESS] 4 3
-> Bipartite Graph!!
```

(1) 무방향 그래프 -> 이분 그래프

(2) 무방향 그래프 -> 이분 그래프

모든 무방향 그래프가 이분 그래프이니 홀수 사이클가 아님(-1)을 반환