실습과제 9.1

[과제 설명]

그래프(graph)는 정점(vertex)과 간선(edge)로 구성된다. 정점은 연결의 대상이 되는 개체/위치이고, 간선은 정점 사이를 연결한다(간선은 직선이 아니어도 된다).

그래프에는 크게 무방향 그래프(undirected graph)와 방향 그래프(directed graph, digraph)로 나눌 수 있다. 이번 실습에서 구현하는 무방향 그래프는 간선의 방향이 없는 그래프이다.

그래프는 정방행렬을 이용한 인접행렬 기반으로 표현한다. 두 정점 사이에 간선이 존재하면 1이고, 없다면 0이다. 이때 무방향 그래프이기 때문에, 대칭행렬이 된다. 또, 자기 자신과는 간선이 없다고 보기 때문에 항상 주대각원소는 0이 된다.

인접 리스트 기반으로 그래프를 표현하게 되면, 간선이 있으면 새로운 노드를 만들어서 연결한다. 이때 노드의 순서는 무관하기 때문에 새로운 간선이 생기게 되면, 맨 처음에 붙이면 된다.

각 정점의 차수는, 인접 행렬에서 행에 있는 모든 원소의 합을 구하면 된다.

그래프의 경우 각 정점이 연결된 간선의 규칙이 없기 때문에 탐색을 하기 위해서는 별도의 알고리즘을 사용할 필요가 있는데, 크게 2가지 방법이 있다.

첫 번째는 깊이 우선 탐색(DFS: Depth First Search)로, 한 방향으로 갈 수 있을 때까지 가는 것이다. 두 번째는 너비 우선 탐색(BFS: Breath First Search)로, 연결된 정점을 모두 방문하는 방식이다.

이번에 구현할 깊이 우선 탐색은 스택을 이용해 구현한다. 스택은 경로 정보 추적을 위해 사용한다. 더이상 갈 수 없을 때, 갔던 길을 되돌아가서 가지 않은 정점이 있으면 그 정점으로 이동하기 위해서이다. 배열은 방문 정보를 기록한다. 방문했던 정점이면 1을 대입한다.

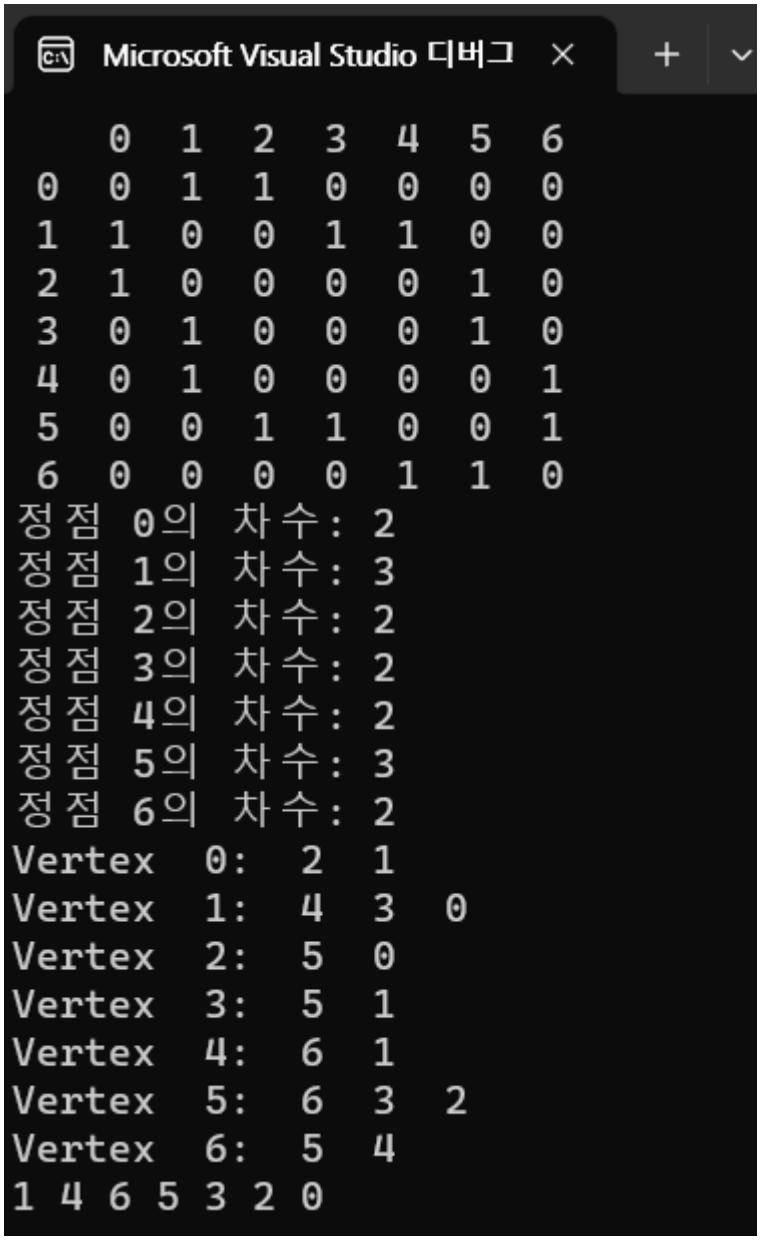
[코드]

메인 함수와 header 파일, 출력 함수는 교안에서 미리 구현된 것을 사용했으므로 보고서에는 생략하고 구현된 함수 세부 내용만 첨부한다.

```
//정점별 차수 출력
void ADJ_Degree(int adj_mat[][MAX_VERTICES], int n) {
  for (int i = 0; i < n; i++) {
    int degree = 0;
    for (int j = 0; j < n; j++) {
      degree += adj_mat[i][j];
    }
    //인접행렬에서 행의 모든 원소의 합인 정점의 차수를 구한다.
    printf("정점 %d의 차수: %d\n", i, degree);
  }
}

//인접 행렬→ 인접 리스트 변환
void ADJ_Insert(G_Node** List, int i, int j) {
  G_Node* newNode = (G_Node*)malloc(sizeof(G_Node));
  newNode->vertex = j;
  newNode->link = List[i];
  List[i] = newNode;
  //간선이 존재하면 연결 리스트의 노드로 추가한다.
}

//인접 행렬 변환
void ADJ_Mat2List(int adj_mat[][MAX_VERTICES], int n, G_Node** List) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (adj_mat[i][j] == 1)
        ADJ_Insert(List, i, j);
    }
  }
  //간선이 존재하면 1을 대입
  //무방향 그래프기 때문에 대칭 행렬이 된다.
}

//깊이 우선 탐색 - 반복문
void Graph_DFS(G_Node** List, int v) {
  G_Node* iter = (G_Node*)malloc(sizeof(G_Node));
  ArrStack<int> stack;
  //지나왔던 길을 저장하기 위한 스택

  int current = v;
  int visted[MAX_VERTICES] = { 0, };
  //방문 여부를 저장하기 위한 배열
  visted[current] = 1;
  printf("%d ", current);

  do {
    bool vFlag = false; //방문 여부 저장
    if (visted[current] == 0) { //방문한 적이 없는 경우
      printf("%d ", current);
      visted[current] = 1;
      vFlag = true;
    }
    else { //방문한 적이 있는 경우
      iter = List[current];
      while (iter != NULL) {
        if (visted[iter->vertex] == 0) {
```

```
          stack.Stack_Push(current);
          current = iter->vertex;
          vFlag = true;
          break;
        }
        iter = iter->link;
      }
    }
    if (vFlag == false) { //방문한 적이 있는데 vFlag가 false인 경우
      if (stack.Stack_IsEmpty()) //스택이 빈 경우
        break;
      else //스택이 비지 않았다면 다시 돌아간다.
        current = stack.Stack_Pop();
    }
  } while (!stack.Stack_IsEmpty());
  //스택이 비어있지 않다면 계속 반복

  puts("");
}

//깊이 우선 탐색 - 재귀 함수
void Graph_DFS_Recursive(G_Node** List, int v) {
  static int visted[MAX_VERTICES] = { 0, };
  //최초 1회만 실행되게 static 변수로 선언

  G_Node* iter = (G_Node*)malloc(sizeof(G_Node));
  iter = List[v];
  visted[v] = 1;
  printf("%d ", v);

  while (iter != NULL) {
    int temp = iter->vertex;
    if (visted[temp] == 0) Graph_DFS_Recursive(List, temp);
    //방문한 적이 없다면 재귀 호출
    iter = iter->link;
  }
}
```

[실행 결과]



[과제에 대한 고찰]

깊이 우선 탐색을 구현할 때, 이렇게 하면 되돌아와야 하는 길이 많다면 너비 우선 탐색이 더 효율적이지 않을까 라는 생각이 들었다. 한 방향으로만 계속 있을 가능성도 있기 때문에, 실생활의 많은 데이터가 있을 때는 비효율적일 수도 있다고 생각했다. 대신, 연결되어있는 정점이 한 정점에 많지 않다면 너비 우선 탐색보다 깊이 우선 탐색이 더 효율적일 때도 많을 것 같다. 미리 데이터를 볼 수 있고, 판단할 수 있는 정도라면 때에 따라서 깊이 우선 탐색과 너비 우선 탐색을 선택해서 쓰는 것이 좋을 것 같다.