이산수학

팀프로젝트 최종보고서

Project Team

10 Team

Date

2021-12-09

Team Information

기술경영학과 201912150 김현아

컴퓨터공학부 201911278 정경은

컴퓨터공학부 202111335 이다민

컴퓨터공학부 202111350 이준혁

컴퓨터공학부 202111352 이지현

1. 그래프 탐방 알고리즘

1.1. 문제 정의

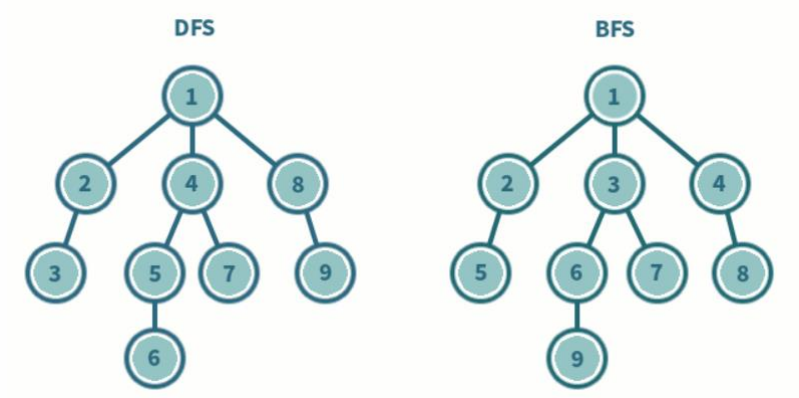
그래프 정보를 인접행렬로 표현하고, 깊이 우선 탐색과 너비 우선 탐색 알고리즘을 구현하여 입력 파일로 주어지는 모든 그래프에 대하여 탐색 알고리즘을 수행한다. 수행한 후에는 그래프의 각 정점이 방문되는 경로를 출력한다.

※ 깊이 우선 탐색 알고리즘 (DFS)

시작 정점에서 인접한 정점 중에서 방문하지 않은 정점 w를 방문하고, w에서 다시 인접 정점 중 방문하지 않은 정점 u를 방문한다. 이 과정을 반복하다가 어떤 정점 u에 인접한 모든 정점들을 이미 방문한 경우가 발생하면, 이전에 마지막으로 방문한 정점으로 되돌아가서 위의 과정을 반복한다. 모든 정점을 방문한 후에는 탐색을 종료한다. BFS와 다르게 DFS는 탐색을 한 뒤 이전 정점으로 돌아온다.

※ 너비 우선 탐색 알고리즘 (BFS)

시작 정점에서 인접한 모든 정점을 차례로 방문한다. 더 이상 방문할 정점이 없는 경우 시작 정점에 인접한 정점을 기준으로 해당 정점과 인접한 모든 정점을 차례로 방문한다. 이를 계속 반복하면서 모든 정점들을 방문한 후 탐색을 종료한다. BFS는 DFS와 다르게 처음에 방문한 정점과 인접한 정점들을 차례로 방문한다.



1.2. 시스템 환경 및 사용 언어

Window10 64bit 이클립스 Java 버전 8

1.3. 그래프 표현 기법: 인접 행렬

인접행렬은 그래프의 각 정점의 연결 관계를 이차원 배열로 나타내는 방식이다. 인접행렬 adj[][]에 대하서, 정점 i에서 정점 j로 가는 간선이 있으면 adj[i][j]가 1이고, 정점 i에서 정점 j로 가는 간선이 없으면 adj[i][j]가 0이다.

1.4. 주요 변수 및 함수에 대한 설명

※ input 파일을 입력 받는 main 함수는 2번에서 서술한다.

① 깊이 우선 탐색 알고리즘 (DFS)

```
/* 깊이 우선 탐색 알고리즘 구현 함수 */
public static void dfs(int start, int n, boolean[] visited, int [][] graphMatrix) {

    /* 방문한 정점 확인 및 출력 */
    visited[start-1] = true;
    System.out.print(start);

     for(int i=0; i< visited.length; i++) {
        if(visited[i]==false) {
            System.out.print(" - ");
            break;
        }
     }

     /* 시작 정점에서 인접한 정점 중 방문하지 않은 정점에 대하여 탐색 실행 */
     for(int j=0; j<n; j++){
         if(graphMatrix[start-1][j]==1 && visited[j]==false){
             dfs(j+1, n, visited, graphMatrix);
         }
     }
}
```

dfs의 매개변수는 총 4개로 설정하였다.

변수명	type	설명
start	int	시작 정점
n	int	정점의 개수
visited	boolean[]	방문한 정점 확인을 위한 배열. 처음에는 모든 요소가 false로 되어있고, 방문하면 true로 변경된다.
graphMatrix	int[][]	txt파일에서 받아들인 그래프의 인접행렬

정점은 1에서 시작하는 반면, 배열의 인덱스는 0에서 시작하기 때문에 visited 인덱스에는 start-1이 필요하다. 방문한 정점이므로 visited의 해당 인덱스를 true로 변경한다. 다음 코드는 방문한 정점을 출력하는 구문이다. 깊이 우선 탐색의 결과이기도 하다.

첫번째 for문은 모든 정점을 방문하기까지 -를 출력하는 출력형식 구문이다. visited의 참 거짓 여부를 이용하여 방문하지 않은 정점이 존재하면(false인 경우) -를 출력한다. 두번째 for문은 dfs를 재귀적으로 호출한다. 시작 정점에서 인접한 정점(graphMatrix[start-1][j] ==1) 중 방문하지 않은 정점(visited[j]==false)에 대하여 탐색을 실행한다.

```
/*모든 정점끼리 연결되지 않은 그래프의 경우 방문하지 않은 정점에 대해 탐색 실행 */
for(int i=0; i < n; i++){
    if(!visited[i]){
        dfs(i+1, n, visited, graphMatrix);
    }
}
System.out.println();
```

만일 정점끼리 끊어진 그래프가 있을 경우를 위한 for문이다. 모든 정점을 방문했는지 확인하여 방문하지 않은 정점이 있으면 다시 dfs를 수행한다. 이 코드를 통해 서로 연결되지 않은 한 그래프에서도 탐색이 원활하게 진행되었다.

② 깊이 우선 탐색 알고리즘 (BFS)

```
/* 너비 우선 탐색 알고리즘 구현 함수 */
public static void bfs(int[][] graphMatrix, int v, int n) {

   System.out.println("너비 우선 탐색");
   boolean[] visit = new boolean[n];
   Queue<Integer> q = new LinkedList();
   q.offer(v);
   visit[v-1]=true;
```

매개변수로 인접행렬(graphMatrix)과 탐색을 시작 정점(v), 노드의 개수(n)를 받았다.

변수명	type	설명
visit	Boolean[]	방문한 노드를 표시하기 위해 선언하였다. 방문하지 않으면 false, 방문하였으면 true로 바꿔주었다.
q	Queue<Integer>	방문한 노드를 순서대로 저장하기 위해 선언하였다.

```
/* 방문한 정점을 순서대로 저장한 q가 비어있기 전까지 탐색 실행 */
label: while(!q.isEmpty()) {
   v=q.poll();
   System.out.print(v);
   for(int i=0;i<n;i++) {
      if(visit[i]==true||graphMatrix[v-1][i]==0)
         continue;
      q.offer(i+1);
      visit[i]=true;
   }
```

방문한 정점은 q에서 poll해주기 때문에 q가 비어있다는 것은 연결된 모든 정점을 방문했다는 것을 나타낸다. 따라서 q가 비어 있지 않을 때 while문을 반복하여 탐색을 진행하였다. 방문한 정점을 poll해서 v에 저장한다. 순서를 결과로 보여주기 위해 v를 출력한다.

for 반복문을 통해 n개의 노드를 탐색한다. 해당 노드가 이미 visit이 true로 되어있거나(q에 저장된 노드), v와 인접하지 않다면(graphMatrix[v-1][i]==0) continue하여 for문을 반복하도록 한다.

만약 해당 노드가 q에 저장되어 있지 않고 v와 인접하다면 해당 노드를 q에 추가한다.

```
if(!q.isEmpty()) {
   System.out.print(" - ");
}

/* 방문하지 않은 정점 추가 */
else {
 for(int i=0;i<n;i++) {
    if(visit[i]==false) {
       System.out.print(" - ");
       q.offer(i+1);
       visit[i] = true;
       continue label;
    }
 }
}
```

q가 비어있지 않다면 탐색할 노드가 남아있는 것이므로 -를 출력하고 while문을 반복한다. q가 비어있는 경우는 모든 정점의 탐색을 끝낸 경우이거나, 연결되어 있지 않은 노드가 존재하는 경우이다.

후자인 경우, visit 배열 중에서 false로 되어있는 부분이 존재하므로 for문을 이용하여 탐색하여 visit에 false가 존재한다면 해당 노드를 q에 추가하고 true로 바꿔준다. label을 이용하여 while문을 반복하도록 한다.

만약 visit의 모든 요소가 true를 나타내고 q도 비어있다면 모든 정점을 방문하여 탐색을 끝낸 것이므로 bfs 탐색을 종료한다.

1.5 다양한 그래프 형태에 대한 수행 결과

파일 이름은 input.txt로 고정하였고, 각기 다른 그래프를 input.txt에 대입하여 실행시켰다.

그래프는 총 5가지이며 결과는 다음과 같다.

① 모든 정점이 연결되어 있는 한 개의 그래프

```
8
1 2 3
2 1 4 5
3 1 6 7
4 2 8
5 2 8
6 3 8
7 3 8
8 4 5 6 7
```

```
그래프 [1]
----------------------------
깊이우선탐색
1 - 2 - 4 - 8 - 5 - 6 - 3 - 7
너비 우선 탐색
1 - 2 - 3 - 4 - 5 - 6 - 7 - 8
----------------------------
```

② 모든 정점이 연결되어 있는 3개의 그래프

```
3
1 2 3
2 1 3
3 1 2
2
1 2
2 1
4
1 2 3
2 4
3 2
4
```

```
그래프 [1]
----------------------------
깊이우선탐색
1 - 2 - 3
너비 우선 탐색
1 - 2 - 3
----------------------------

그래프 [2]
----------------------------
깊이우선탐색
1 - 2
너비 우선 탐색
1 - 2
----------------------------

그래프 [3]
----------------------------
깊이우선탐색
1 - 2 - 4 - 3
너비 우선 탐색
1 - 2 - 3 - 4
----------------------------
```

③ 모든 정점이 연결되지 않은 한 개의 그래프

```
5
1 2 3
2 1
3 1
4 5
5 4
```

```
그래프 [1]
----------------------------
깊이우선탐색
1 - 2 - 3 - 4 - 5
너비 우선 탐색
1 - 2 - 3 - 4 - 5
----------------------------
```

④ 모든 정점이 연결된 그래프 한 개와 모든 정점이 연결되지 않은 그래프 한 개

```
8
1 2 3
2 1 4 5
3 1 6 7
4 2 8
5 2 8
6 3 8
7 3 8
8 4 5 6 7
5
1 2 3
2 1
3 1
4 5
5 4
```

```
그래프 [1]
----------------------------
깊이우선탐색
1 - 2 - 4 - 8 - 5 - 6 - 3 - 7
너비 우선 탐색
1 - 2 - 3 - 4 - 5 - 6 - 7 - 8
----------------------------
그래프 [2]
----------------------------
깊이우선탐색
1 - 2 - 3 - 4 - 5
너비 우선 탐색
1 - 2 - 3 - 4 - 5
----------------------------
```

⑤ 모든 정점이 연결된 복잡한 그래프

```
16
1 2 3
2 1 4
3 1 4
4 2 3 5 6 11 12
5 4 6 7
6 4 5
7 5 8
8 7
9 10 11
10 9 11
11 4 9 13
12 4 13
13 12 14
14 13 15 16
15 16
16 15
그래프 [1]
----------------------------
깊이우선탐색
1 - 2 - 4 - 3 - 5 - 6 - 7 - 8 - 11 - 9 - 10 - 13 - 12 - 14 - 15 - 16
너비 우선 탐색
1 - 2 - 3 - 4 - 5 - 6 - 11 - 12 - 7 - 9 - 13 - 8 - 10 - 14 - 15 - 16
----------------------------
```

2 최단 거리 구하기 알고리즘 구현

2.1. 문제 정의

다익스트라(Dijkstra) 알고리즘을 이용하며, 입력 파일로 주어지는 모든 그래프에 대하여 시작점(정점 1)으로부터 각 정점까지의 최단 경로 및 거리를 출력한다.

※ 다익스트라 알고리즘

음의 가중치가 없는 그래프에서, 시작점으로부터 다른 모든 정점까지의 최단 거리를 구하는 알고리즘이다. 시작점을 제외한 점들 중에서 시작점으로부터 이르는 거리가 가장 짧은 점들의 거리를 방문한다. 이 때 방문한 점은 최단 거리가 확정되며, 그 후로는 변하지 않는다. 시작점으로부터의 거리는 하나의 배열(D)에 저장되어 있으며, 시작점으로부터의 거리가 최솟값인 점을 방문하고, 매번 방문하지 않은 점들에 대하여 '기존 최단 거리'와 '이번에 방문한 점을 경유하는 거리'를 비교하여 최솟값이 배열에 저장된다. 즉, 배열의 값들을 갱신하거나 유지한다. 예를 들어, 이번에 방문한 점을 A, 아직 방문하지 않은 점들 중 한 점을 B, 점 B와 A사이 거리를 C[A][B]라고 하면, D[B]와 D[A]+C[A][B] 중 작은 값이 D[B]에 저장된다. 모든 정점를 방문할 때까지 이 과정이 반복된 후, 배열 D에 저장된 값이 시작점으로부터 각 정점까지 이르는 최단 거리이다.

2.2. 시스템 환경 및 사용 언어

MacOS BigSur 11.5.2 / 이클립스 / Java (버전 8)

2.3. 그래프 표현 기법: 인접 행렬

그래프 정보는 인접 행렬로 표현한다. n개의 정점을 가지는 그래프에 대하여, n*n int형 2차원 배열을 선언하고, 입력에서 주어지는 각 정점 별 인접 노드와 인접 노드까지의 가중치 값을 배열에 저장한다. (배열 graphMatrix)

- 거리(가중치) 값 표현

-> 자기 자신: 0

-> 인접한 정점: C[A][B] (점 A에서부터 B까지 거리)

-> 인접하지 않은 정점: 무한대 (프로그램에서는 int 범위 내의 최댓값으로 표현)

2.4. 주요 변수 및 함수 설명

① main 함수

```
public static void main(String[] args) throws IOException{
    int n=0, count=1;
    String str;

    /* 파일 읽기 */
    File file = new File("./input2.txt");
    if(file.exists()) {
        BufferedReader bufRead = new BufferedReader(new FileReader(file));
        while ((str = bufRead.readLine()) != null) {
            System.out.printf("그래프 [%d]\n", count);

            n = Integer.parseInt(str);

            int[][] graphMatrix = new int[n][n];
```

파일 객체 생성 후 "input2.txt" 파일을 연다. 해당 파일이 존재하는 경우 파일을 읽고, 존재하지 않는다면 파일을 찾지 못했다는 문구를 출력한 후 프로그램을 종료(else문에 기재)한다. 파일의 내용을 읽을 때는 BufferedReader를 사용하며, 이 때문에 발생하는 예외를 처리해주기 위하여 IOException를 throws 해준다.

input 파일의 모든 내용은 readLine() 함수를 사용하여 줄 단위로 받으며, 받아올 수 있는 줄이 존재하는 동안 while문이 실행된다. while문 조건식의 str에는 각 그래프의 첫 줄, 즉, 해당 그래프의 총 노드 수가 저장된다. 이 때, str 변수는 string 타입이므로, int형으로 변환하여 변수 n에 저장해준다. int형 변수 n은 프로그램 내에서, 한 그래프의 총 노드 개수를 의미한다.

인접 행렬을 표현하기 위하여 n*n int형 2차원 배열 graphMatrix을 선언 및 초기화한다.

```
for (int k = 0; k < n; k++) {
    String[] splitStr = bufRead.readLine().split(" ");
    int num = Integer.parseInt(splitStr[0]);

    for(int l=0; l<(splitStr.length/2); l++){
        int j = Integer.parseInt(splitStr[l*2+1]);
        int weight = Integer.parseInt(splitStr[(l+1)*2]);

        /* 가중치 값 저장 */
        graphMatrix[num-1][j-1] = weight;
    }

    for(int l=0; l<n; l++){
        if(graphMatrix[num-1][l] == 0){
            if(l == num-1){
                continue;
            }
            graphMatrix[num-1][l] = Integer.MAX_VALUE;
        }
    }
}
```

input 파일의 그래프 정보는 해당 그래프의 총 정점 개수만큼의 줄로 이루어져 있으므로, for문으로 총 n번, 줄 단위로 파일을 읽는다. 각 정점 별로, 인접 정점과 인접 정점까지의 가중치 값들이 공백으로 구분되어 있으므로, split() 함수를 사용하여 정보를 String 배열 splitStr에 저장한다.

splitStr[0]은 해당 줄이 정보를 알려주는 정점(int num)이고, 배열의 나머지 값들은 모두 이와 인접한 정점과 인접한 정점까지의 거리이다. 정점과 가중치 값을 구분하여 저장하기 위하여 for문 조건식의 l 값으로 인덱스 값을 조정(1 이상의 인덱스 중 홀수 -> 인접 정점, 짝수 -> 인접 정점까지의 거리)한다. 각 행렬의 인덱스는 0부터 시작하므로 가중치 값을 배열에 저장할 때, 인덱스를 각 (정점(숫자)-1)로 한다.

인접하지 않은 정점들에 대해서는, 거리(가중치)를 무한대(Integer 범위 내 가장 큰 값)로 저장한다.

```
        /* 다익스트라 알고리즘 구현 함수 호출 */
        Dijkstra(graphMatrix, n);
        count++;
    }
    bufRead.close();
}else {
    System.out.println("No file found");
}
```

Dijkstra함수를 호출한다. 이 때, 인자 값으로 인접 행렬로 표현한 그래프 정보인 graphMatrix와 정점 개수 n을 넘겨준다.

더 이상 읽을 내용이 없으면 while문이 종료되며, BufferedReader를 close해주고 프로그램은

종료된다.

② Dijkstra 함수 (다익스트라 알고리즘 구현 함수)

```
/* 시작점 : 1 */
public static void Dijkstra(int [][] graphMatrix, int n){
    boolean[] visitedNode = new boolean[n];
    int [] distance = new int[n];
    String [] route = new String[n];

    distance = graphMatrix[0];
    visitedNode[0] = true;

    Arrays.fill(route, "1");
```

시작점(정점 1)으로부터 각 정점까지의 최단 거리를 담아줄 int형 distance 배열과, 방문 여부를 표시할 boolean형 visitedNode 배열, 시작점으로부터 각 정점에 이르는 최단 경로를 담아줄 String형 route 배열을 선언 및 초기화해준다.

초기 distance 배열은 Dijkstra 함수가 인자로 받은 graphMatrix 배열의 1행과 같다.

문제에서 시작점은 항상 1이다. 따라서, 자기 자신인 정점 1에 대한 방문 여부를 true로 저장해주고, 모든 정점들에 대하여 경로의 첫 시작이 1로 동일하므로 route 배열을 1로 일괄 채워준다.

```
for (int i=0; i<n-1; i++){
    int minDistance = Integer.MAX_VALUE;
    int minIndex = 0;

    /* distance 배열에서 시작점으로 부터 최소 거리를 가지는 정점 추출 */
    for(int j=0; j<n; j++){
        if(!visitedNode[j] && distance[j] != Integer.MAX_VALUE){
            if(distance[j] < minDistance) {
                minDistance = distance[j];
                minIndex = j;
            }
        }
    }

    /* 확정된 정점에 대하여 마지막 경로 지정 */
    route[minIndex] += " - " + (minIndex+1);


    /* 해당 정점 방문 표기 */
    visitedNode[minIndex] = true;
```

시작점을 제외한 모든 정점들에 대해서 시작점으로부터의 최단 거리 및 경로를 구해야 하기 때문에, n-1번 반복한다. (바깥 for문)

아직 방문하지 않은 정점들 중에서, 시작점으로부터의 최단 거리 중 최솟값을 저장하기 위한 int형 변수 minDistance를 선언하고, 무한대(Integer값의 범위 내 가장 큰 값)로 초기화해준다. 모든 값들을 비교한 후, minDistance의 값을 최단 거리로 가지는 정점이 int 형 변수 minIndex에 저장된다.

아직 방문하지 않았고, 시작점으로부터 경로가 있는 정점(distance[j] != Interger.MAX_VALUE)들에 대하여, 최단 거리(distance 배열의 값)를 비교하여 최솟값을 minDistance에, 해당 정점을 minIndex에 저장한다.

안쪽 for문이 종료되었을 때, minIndex의 정점은 최단 거리가 확정되며, 앞으로 변하지 않는다. 또한, 시작점으로부터 minIndex 정점에 이르는 최단 경로 또한 확정된다. 따라서, 최솟값의 최단 거리를 가지는 정점으로 확정되기 전까지 갱신 및 유지되었던 route 배열의 값에 자기 자신(정점 = 인덱스 값 + 1)으로의 경로를 추가하여 해당 정점의 최단 경로를 마무리 짓는다. (route 배열의 값 갱신 및 유지는 바로 아래에서 기술한다.)

또, minIndex 정점의 방문 여부를 표시해준다.

```
/* distance 배열의 각 원소(최단 거리) 유지 및 갱신 */
for(int j=0; j<n; j++){
    if(!visitedNode[j] && graphMatrix[minIndex][j] != Integer.MAX_VALUE){
        if(distance[j] > (distance[minIndex] + graphMatrix[minIndex][j])){
            distance[j] = distance[minIndex] + graphMatrix[minIndex][j];
            route[j] = route[minIndex];
        }
    }
}
```

아직 방문하지 않았고, minIndex와 인접한 정점들에 대하여, 기존 distance 배열에 저장된 값과 minIndex 정점을 경유한 거리 중 작은 값을 distance 배열에 저장한다. 즉, 점들의 최단 거리를 갱신 및 유지한다. 만약, 거리가 갱신될 경우, 시작점으로부터 정점까지의 경로 또한 수정되므로, route 배열에 route[minIndex] 값을 저장한다. 이 때, 자기 자신을 제외하고 경유하는 점 minIndex까지의 경로만 저장하는 이유는, 그 다음 과정에서 해당 정점의 최단 거리가 확정되지 않으면서, 최단 거리 및 경로가 다시 수정될 수 있기 때문이다. 최단 경로를 마무리 짓는 파트에서 표현된 route 배열의 값 갱신 및 유지가 이것을 의미한다.

```
/* 시작점으로부터 경로가 존재하지 않는 정점에 대하여, 경로 표기 */
for(int i=0; i<n; i++){
    if(distance[i] == Integer.MAX_VALUE){
        route[i] = "경로가 존재하지 않음";
    }
}
```

모든 정점을 방문하고 나면 for문이 종료된다. 이 때, distance 배열의 값들이 시작점으로부터 각 정점들에 이르는 최단 거리를 의미하고, route에는 최단 경로가 문자열로 저장되어있다. 이 때, 시작점으로부터 이르는 경로가 존재하지 않아, 거리 값이 무한대(프로그램 상 integer 범위 내 최댓값)로 유지되는 정점에 대하여, 경로가 존재하지 않음을 route 배열에 저장한다.

```
System.out.println("---------------------------------------");

/* (결과) 최단 경로 및 거리 출력 */
System.out.println("시작점 : 1");
for(int i=1; i<n; i++){
    System.out.printf("정점 [%d] : %s, 길이 : %d\n", i+1, route[i], distance[i]);
}

System.out.println("=======================================");
System.out.println();
```

시작점으로부터 모든 정점들에 이르는 최단 경로와 거리를 출력한다.

2.5. 다양한 그래프 형태에 대한 수행 결과

① 1개의 그래프(방향 그래프)

```
5
1 2 30 3 10
2 4 20
3
4 5 10
5
```

```
그래프 [1]
---------------------------------------
시작점 : 1
정점 [2] : 1 - 2, 길이 : 30
정점 [3] : 1 - 3, 길이 : 10
정점 [4] : 1 - 2 - 4, 길이 : 50
정점 [5] : 1 - 2 - 4 - 5, 길이 : 60
=======================================
```

② 1개의 그래프(무방향 그래프)

```
5
1 2 3 3 6 4 8 5 7
2 1 3 3 2 4 4 5 8
3 1 6 2 2 4 5 5 5
4 1 8 2 4 3 5 5 2
5 1 7 2 8 3 5 4 2
```

```
그래프 [1]
---------------------------------------
시작점 : 1
정점 [2] : 1 - 2, 길이 : 3
정점 [3] : 1 - 2 - 3, 길이 : 5
정점 [4] : 1 - 2 - 4, 길이 : 7
정점 [5] : 1 - 5, 길이 : 7
=======================================
```

③ 2개의 그래프(방향 그래프)

```
8
1 2 15 6 2
2 3 17
3 4 8 5 10
4 5 3
5
6 2 10 7 9 8 14
7 8 10
8 5 17
7
1 2 20 4 13 6 8
2 3 7 4 12 5 6
3 5 14
4 5 10 7 9
5
6 3 11
7 6 10
```

```
그래프 [1]
---------------------------------------
시작점 : 1
정점 [2] : 1 - 6 - 2, 길이 : 12
정점 [3] : 1 - 6 - 2 - 3, 길이 : 29
정점 [4] : 1 - 6 - 2 - 3 - 4, 길이 : 37
정점 [5] : 1 - 6 - 8 - 5, 길이 : 33
정점 [6] : 1 - 6, 길이 : 2
정점 [7] : 1 - 6 - 7, 길이 : 11
정점 [8] : 1 - 6 - 8, 길이 : 16
=======================================

그래프 [2]
---------------------------------------
시작점 : 1
정점 [2] : 1 - 2, 길이 : 20
정점 [3] : 1 - 6 - 3, 길이 : 19
정점 [4] : 1 - 4, 길이 : 13
정점 [5] : 1 - 4 - 5, 길이 : 23
정점 [6] : 1 - 6, 길이 : 8
정점 [7] : 1 - 4 - 7, 길이 : 22
=======================================
```

④ 3개의 그래프(방향 그래프)

- 시작점으로부터 경로가 존재하지 않는 경우의 그래프 포함

```
7
1 3 10 7 5
2 5 11 7 7
3 2 8 4 12
4 6 6
5 3 2 6 13
6 2 3
7 4 4 6 9
6
1 2 10 3 1
2 4 4
3 2 5 4 16 6 23
4 5 7 6 10
5 6 2
6
5
1 3 6 4 3
2 1 3
3 4 2
4 2 1 3 1
5 2 4 4 2
```

```
그래프 [1]
--------------------------------------
시작점 : 1
정점 [2] : 1 - 7 - 6 - 2, 길이 : 17
정점 [3] : 1 - 3, 길이 : 10
정점 [4] : 1 - 7 - 4, 길이 : 9
정점 [5] : 1 - 7 - 6 - 2 - 5, 길이 : 28
정점 [6] : 1 - 7 - 6, 길이 : 14
정점 [7] : 1 - 7, 길이 : 5
======================================

그래프 [2]
--------------------------------------
시작점 : 1
정점 [2] : 1 - 3 - 2, 길이 : 6
정점 [3] : 1 - 3, 길이 : 1
정점 [4] : 1 - 3 - 2 - 4, 길이 : 10
정점 [5] : 1 - 3 - 2 - 4 - 5, 길이 : 17
정점 [6] : 1 - 3 - 2 - 4 - 5 - 6, 길이 : 19
======================================

그래프 [3]
--------------------------------------
시작점 : 1
정점 [2] : 1 - 4 - 2, 길이 : 4
정점 [3] : 1 - 4 - 3, 길이 : 4
정점 [4] : 1 - 4, 길이 : 3
정점 [5] : 경로가 존재하지 않음, 길이 : 2147483647
======================================
```

⑤ 4개의 그래프(무방향 그래프)

```
7
1 2 7 5 3 6 10
2 1 7 3 4 4 10 5 2 6 6
3 2 4 4 2
4 2 10 3 2 6 9 7 4
5 1 3 2 2 7 5
6 1 10 2 6 4 9
7 4 4 5 5
4
1 2 3 3 6 4 7
2 1 3 3 1
3 1 6 2 1 4 1
4 1 7 3 1
6
1 2 2 3 5 4 1
2 1 2 3 3 4 2
3 1 5 2 3 4 3 5 1 6 5
4 1 1 2 2 3 3 5 1
5 3 1 4 1 6 2
6 3 5 5 2
```

```
--------------------------------------
시작점 : 1
정점 [2] : 1 - 5 - 2, 길이 : 5
정점 [3] : 1 - 5 - 2 - 3, 길이 : 9
정점 [4] : 1 - 5 - 2 - 3 - 4, 길이 : 11
정점 [5] : 1 - 5, 길이 : 3
정점 [6] : 1 - 6, 길이 : 10
정점 [7] : 1 - 5 - 7, 길이 : 8
======================================

그래프 [2]
--------------------------------------
시작점 : 1
정점 [2] : 1 - 2, 길이 : 3
정점 [3] : 1 - 2 - 3, 길이 : 4
정점 [4] : 1 - 2 - 3 - 4, 길이 : 5
======================================

그래프 [3]
--------------------------------------
시작점 : 1
정점 [2] : 1 - 2, 길이 : 2
정점 [3] : 1 - 4 - 5 - 3, 길이 : 3
정점 [4] : 1 - 4, 길이 : 1
정점 [5] : 1 - 4 - 5, 길이 : 2
정점 [6] : 1 - 4 - 5 - 6, 길이 : 4
======================================
```

3. 토의사항

깊이 우선 탐색 함수 호출 위치를 지정하는 과정에 어려움을 겪었다. 매개변수를 바꿔가며 수정하는 과정에서, 함수의 호출 위치가 잘못되어 인접행렬이 제대로 인식되지 않는 문제가 발생했다는 것을 알 수 있었다. 따라서, 함수 호출 위치를 파일을 읽어들이는 while문 안으로 수정하였고, 문제를 해결할 수 있었다.

모든 정점이 연결되지 않은 그래프에 관하여 깊이 우선 탐색 알고리즘을 구현하는 과정에서 어려움이 있었다. 연결이 끊겨 방문하지 않은 정점에 대해 어떻게 구현해야 할지 고민하였고, 이 과정에서 dfs 함수를 재귀로 구현하기도, stack으로 구현하기도 하였다. 결론적으로 재귀를 쓰면서 main함수에서 다시 방문하지 않은 정점에 대해 dfs 함수를 실행하는 것으로 해결하였다.

너비 우선 탐색에서는 깊이 우선 탐색과 달리 Queue를 이용해서 해야 했다. 따라서 Queue의 작동 원리와 너비 우선 탐색의 탐색 방법에 대한 정확한 이해가 필요했다. 또한 연결되지 않은 정점에 대해 탐색하는 경우, 어떻게 Queue에 정점을 저장해야 하는지 고민하였다. 이 경우에는, 이미 Queue에 저장되어 있는 정점을 방문한 후 방문하지 않은 정점이 남아있는지 확인하는 방법으로 해결하였다.

시작 정점으로부터 각 정점에 이르는 '최단 경로'를 갱신해 나가는 방법이 떠오르지 않았다. 최단 거리(distance) 중 최솟값을 가지는 정점을 선택하고 나면, 해당 정점에 대하여 최단 거리가 다시는 변경되지 않는다는 점에서 아이디어를 얻었다. 최단 경로 또한, 최단 거리가 갱신될 때 그에 따라 당시 거리 및 경로가 확정된 정점을 경유하도록 경로가 바뀌고, 최단 거리가 유지되면 기존 경로를 유지한다. 따라서, 정점이 선택되어 방문 여부가 표시됨과 동시에, 해당 정점에 대한 최단 경로에 그 정점을 포함시켜 마무리되도록 하고, 그 후 최단 거리를 갱신 및 유지하는 과정에서, 거리가 갱신되는 경우에만 선택된 정점을 경유하도록 route 배열을 갱신하도록 하였다.

또, 여러 그래프 형태에 대하여 프로그램을 실행시켜보던 중 한 정점에 대해서 무한대 길이가 나오는 결과를 보게 되었다. 처음에 코드를 잘못 작성한 줄 알았으나, 수기로 직접 풀어본 결과 그 정점에 대한 경로가 존재하지 않아 발생하는 문제였다. 결과를 출력하기 전, 존재하지 않는 경로에 대해 예외 처리를 해주는 것으로 해결하였다.

4. 기여도

김현아(20%): DFS 함수 구현, 보고서 1번 수정 및 총 정리

정경은(20%): 다익스트라 함수 구현, 보고서 2번 수정 및 총 정리

이다민(20%): 다익스트라 함수 구현, 보고서 2번 초안 작성

이준혁(20%): DFS 함수 구현, 보고서 1번 초안 작성

이지현(20%): BFS 함수 구현, 보고서 1번 초안 작성