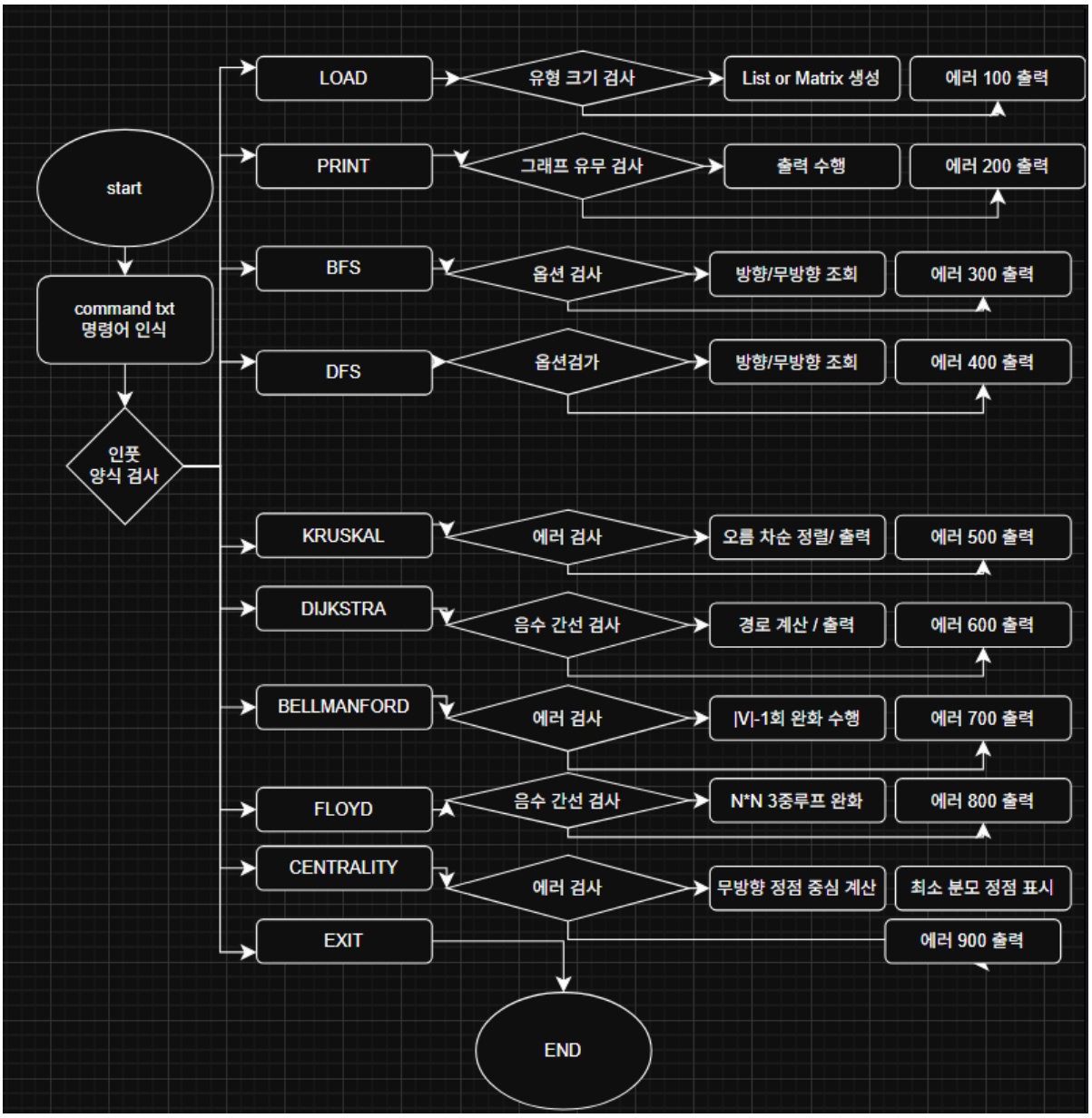
데이터 구조 설계/ 실습

2025_Kwangwoon_DS_Project_2
2021202003 강준우

Introduction

해당 프로잭트는 관리 프로그램으로 인접 리스트 및 인접 행렬 파일을 기반하여 그래프를 표현하고 관리한다.
그래프는 방향 무방향 그래프등의 데이터셋을 기반으로 자료를 컨트롤 할 수 있게된다.
해딩 프로잭트에서는 DFS BFS를 기반으로한 탐색 및 탐욕 알고리즘을 통한 최소 신장 트리가 수현되어 있으며 간선의 양음수에 따라 최단 경로를 계산하게 된다. 연산 내용은 커맨드 텍스틀 파일을 기반으로 수행하며 각 구현단계에 따라 100~900까지의 에러 코드를 반환한다.

Flowchart



메니저 파일에서 총괄 관리를 하고 각 기능은 파일을 호출하여 사용한다.

Algorithm

getAdjacentEdges·getAdjacentEdgesDirect 인터페이스를 통해 인접 정점을 조회하게 된다.

너비 우선 탐색은 큐를 기반으로 큐에 해당 노드 기준 인접 노드를 순서대로 넣고 반복한다. 옵션 O일 때는 getAdjacentEdgesDirect로 방향 그래프의 out-degree를 X일 때는 getAdjacentEdges로 양방향 인접 정점을 가져와 방문 순서를 기록한다.

깊이 우선 탐색은 인접 맵을 재귀를 기반으로 리프로 가기전까지 계속 다음 방문하지 않은 깊은 노드로 이동한다.

O 옵션은 방향 간선만 X 옵션은 양방향 간선을 사용해 탐색 순서를 결과 로그에 출력한다.

탐욕 방식은 먼저 무방향 관점에서 각 정점에 대해 getAdjacentEdges를 호출해 간선을 모두 수집하고 키 기준으로 가장 작은 가중치만 남긴 뒤 가중치 오름차순으로 정렬한다. 유니온 파인드 알고리즘을 기반으로 사이클이 생기지 않도록 관리한다. 유니온 파인드 알고리즘은 노드가 연결될 때 헤드를 계속 한 노드로 지정하여 두 노드가 사이클이 있는지 없는지 확인한다. 해드가 서로를 바라보고 있다면 사이클을 의마한다. MST 인접 리스트를 쌓으면서 총 비용을 기반으로 최종적으로 비용을 반환한다.

다익스트라는 실행 전에 모든 정점에 대해 getAdjacentEdgesDirect를 이용해 음수 가중치 간선이 존재하는지 검사해 하나라도 있으면 알고리즘을 바로 실패로 처리한다. 음수 간선이 없을 때만 거리 배열을 무한대로 초기화한 뒤 시작 정점 거리를 0으로 초기화둔다. 아직 확정되지 않은 정점 중 최소 거리를 갖는 정점을 반복적으로 선택해 그 정점에서 나가는 간선을 관리한다. 이전 정점 배열로 경로를 기록하고 최종적으로 각 정점까지의 거리를 바탕으로 이전 체인을 기반으로 경로 와 거리 형식으로 출력한다.

벨만 포드는 O일 때는 getAdjacentEdgesDirect X 일 때는 getAdjacentEdges로 모든 간선을 리스트에 담는다. 시작 정점의 거리를 0으로 나머지를 무한대로 둔 뒤 |V|-1회에 걸쳐 모든 간선에 대해 기존보다 더 짧은 경우를 찾아 갱신한다. 그 후 한 번 더 전체 간선을 검사해 여전히 완화가 가능하면 음수 사이클이 있다고 판단해 실패를 반환한다. 사이클이 없고 도착 정점 거리가 무한대가 아니면 prev를 역추적해 경로를 구성하고 최종 비용을 함께 출력한다. 실패시 x 를 반환한다.

플로이드는 내부 헬퍼인 buildFloydMatrix에서 정점 수 N에 대해 N×N 거리 행렬을 무한대로 초기화하고 행렬값이 같은 부분은 0으로 초기화한다. 각 정점에 대해 인접 정점 맵을 얻어 직접 간선 가중치를 dist[i][j]에 반영한다. 이후 모든 중간 정점 k에 대해 모든 i j 쌍에 대해 기존의 길이가 신규 발견한 거리보다 길면 갱신하는 3중 루프를 수행해 모든 정점 쌍 최단 거리를 계산한다.

Centrality는 무방향 기준 전체 쌍 최단 거리 행렬이 필요하므로 먼저 플로이드를 X 옵션으로 호출해 dist[u][v]를 얻은 뒤 각 정점 v에 대해 자신을 제외한 모든 정점 u에 대해 dist[u][v]를 합산하되 하나라도 무한대 혹은 발견 못한 간선이 있으면 x로 표시한다. 모든 u로부터 도달 가능한 정점 v에 대해서만 가장 작은 정점을 찾아 Most Central 표시를 붙인다. 수식은 분모는 거리 u_dist[u][v]의 합으로 분자를 N-1기반으로 수행한다.

Result Screen

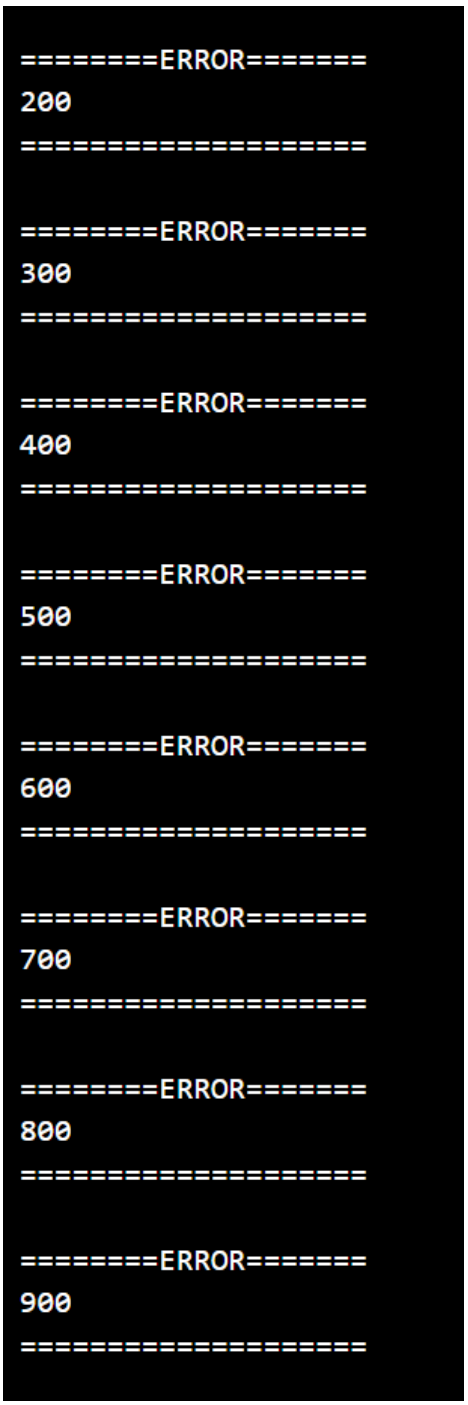
```
PRINT
BFS O 0
DFS O 0
KRUSKAL
DIJKSTRA O 0
BELLMANFORD O 0 6
FLOYD O
CENTRALITY
LOAD graph_L.txt
PRINT
BFS O 0
BFS X 0
DFS O 0
DFS X 0
KRUSKAL
DIJKSTRA O 0
DIJKSTRA X 0
BELLMANFORD O 0 6
BELLMANFORD X 0 6
FLOYD O
FLOYD X
CENTRALITY
LOAD graph_M.txt
PRINT
BFS O 0
BFS X 0
DFS O 0
DFS X 0
KRUSKAL
DIJKSTRA O 0
DIJKSTRA X 0
BELLMANFORD O 0 6
BELLMANFORD X 0 6
FLOYD O
FLOYD X
CENTRALITY
```

수행한다.

```
L
7
0
1 6
2 2
1
3 5
2
1 7
4 3
5 8
3
6 3
4
3 4
5
6 1
6
4 10
```

```
M
7
0 6 2 0 0 0 0
0 0 0 5 0 0 0
0 7 0 0 3 8 0
0 0 0 0 0 0 3
0 0 0 4 0 0 0
0 0 0 0 0 0 1
0 0 0 0 10 0 0
```

순서대로 Graph_L Graph_M파일이다.



의도적으로 로드하기전 수행해 에러를 확인한 모습이다.

```
========LOAD========
Success
====================

========PRINT========
[0] -> (1,6) -> (2,2)
[1] -> (3,5)
[2] -> (1,7) -> (4,3) -> (5,8)
[3] -> (6,3)
[4] -> (3,4)
[5] -> (6,1)
[6] -> (4,10)
======================

========BFS========
Directed Graph BFS
Start: 0
0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6
======================

========BFS========
Undirected Graph BFS
Start: 0
0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6
======================

========DFS========
Directed Graph DFS
Start: 0
0 -> 1 -> 3 -> 6 -> 4 -> 2 -> 5
======================

========DFS========
Undirected Graph DFS
Start: 0
0 -> 1 -> 2 -> 4 -> 3 -> 6 -> 5
======================
```

```
========KRUSKAL========
[0] 2(2)
[1] 3(5)
[2] 0(2) 4(3)
[3] 1(5) 4(4) 6(3)
[4] 2(3) 3(4)
[5] 6(1)
[6] 3(3) 5(1)
Cost: 18
======================

========DIJKSTRA========
Directed Graph Dijkstra
Start: 0
[0] 0 (0)
[1] 0 -> 1 (6)
[2] 0 -> 2 (2)
[3] 0 -> 2 -> 4 -> 3 (9)
[4] 0 -> 2 -> 4 (5)
[5] 0 -> 2 -> 5 (10)
[6] 0 -> 2 -> 5 -> 6 (11)
======================

========DIJKSTRA========
Undirected Graph Dijkstra
Start: 0
[0] 0 (0)
[1] 0 -> 1 (6)
[2] 0 -> 2 (2)
[3] 0 -> 2 -> 4 -> 3 (9)
[4] 0 -> 2 -> 4 (5)
[5] 0 -> 2 -> 5 (10)
[6] 0 -> 2 -> 5 -> 6 (11)
======================

========BELLMANFORD========
Directed Graph Bellman-Ford
0 -> 2 -> 5 -> 6
Cost: 11
======================
```

```
========FLOYD========
Directed Graph Floyd
    [0] [1] [2] [3] [4] [5] [6]
[0] 0 6 2 9 5 10 11
[1] x 0 x 5 18 x 8
[2] x 7 0 7 3 8 9
[3] x x x 0 13 x 3
[4] x x x 4 0 x 7
[5] x x x 15 11 0 1
[6] x x x 14 10 x 0
======================

========FLOYD========
Undirected Graph Floyd
    [0] [1] [2] [3] [4] [5] [6]
[0] 0 6 2 9 5 10 11
[1] 6 0 7 5 9 9 8
[2] 2 7 0 7 3 8 9
[3] 9 5 7 0 4 4 3
[4] 5 9 3 4 0 8 7
[5] 10 9 8 4 8 0 1
[6] 11 8 9 3 7 1 0
======================

========CENTRALITY========
[0] 6/43
[1] 6/44
[2] 6/36
[3] 6/32 <- Most Central
[4] 6/36
[5] 6/40
[6] 6/39
======================
```

LOAD
graph_L.txt를 인접 리스트 그래프로 로드

PRINT
각 정점의 출력이 graph_L.txt에 정의된 간선과 정확히 일치한다.
정점별로 오름차순으로 간선이 나열되므로 이후 BFS/DFS 등에서 “작은 번호 우선 탐색”이 기대대로 동작할 수 있는 상태다.

BFS O 0
시작 정점 0에서 방향 그래프 기준으로 탐색했을 때 0 1 2 3 4 5 6 순서로 모든 정점이 한 번씩 방문되므로 0에서 모든 정점이 도달 가능하고 BFS 구현이 레벨 순서 와 오름차순 방문 규칙을 지킨다.
특히 1과 2가 0의 직후에 이후 3·4·5·6이 그 다음 레벨에서 방문되는 구조가 그래프의 간선 구성과 정확히 대응한다.

BFS X 0
무방향 모드에서도 방문 순서가 동일하게 0 1 2 3 4 5 6으로 나오는 것은 이 그래프가 방향을 무시했을 때도 하나의 연결 컴포넌트를 이루고 있고 BFS 구현이 getAdjacentEdges를 통해 양방향 간선을 일관되게 처리함을 보여준다.

DFS O 0
방향 DFS 결과 0 → 1 → 3 → 6 → 4 → 2 → 5는 항상 가능한 가장 작은 정점으로 재귀적으로 내려가는 구현(0→1→3→6→4 이후 아직 방문 안 된 2→5)을 정확히 반영한다. 이 순서는 코드에서 인접 정점 맵을 오름차순으로 순회하면서 방문 배열로 중복 방문을 막는 재귀 DF이다.
DFS X 0
무방향 DFS 결과 0 → 1 → 2 → 4 → 3 → 6 → 5는 양방향 간선까지 포함해 "가능한 한 깊게" 내려가는 과정에서 0-1-2-4-3-6-5 순으로 트리를 형성했음을 보여준다.
방향 모드와 방문 순서가 달라진 것은 getAdjacentEdges가 간선을 모두 반환하기 때문에 무방향 관점 DFS가 실제 무방향 그래프 구조를 반영하고 있다는 의미다.

KRUSKAL
KRUSKAL 결과 MST 인접 리스트와 cost 18은 무방향으로 봤을 때 이 그래프의 최소 신장 트리가 6개의 간선으로 구성된다는 손계산과 정확히 일치한다.
출력된 구조가 모든 정점을 포함하면서 사이클이 없고 코스트 합도 18이므로 간선 수집·중복 제거·정렬·유니온-파인드 로직이 모두 제대로 동작한 상태다.

DIJKSTRA O 0
방향 Dijkstra 결과에서 각 정점까지의 최단 거리와 경로(예: 0→2→4→3의 거리 9 0→2→5→6의 거리 11)는 실제 경로 비용 합과 일치하므로 거리 배열·이전 정점 배열·최소 거리 정점 선택 로직이 올바르다.

DIJKSTRA X 0
무방향 Dijkstra에서도 거리와 경로가 방향 모드와 완전히 동일하게 나오는 것은 이 그래프가 방향을 양방향으로 취급해도 최단 경로 구조가 변하지 않는 형태라는 점과 getAdjacentEdges 기반 relax 로직이 문제 없이 돌아간다는 것을 동시에 보여준다.
즉 내부 표현(List/Matrix)와 방향 옵션을 달리해도 Dijkstra의 알고리즘 핵심 부분은 일관되게 동작하고 있다.

BELLMANFORD O 0 6
Bellman-Ford 결과 0 → 2 → 5 → 6 Cost 11은 Dijkstra가 구한 0→6 최단 경로와 동일하고 추가 완화에서 더 짧은 경로가 나타나지 않았다는 의미로 음수 사이클이 없음을 확인한다.
도달 불가능일 때 x를 찍도록 구현했는데 이 케이스에서는 유한 비용이 출력되므로 edge list 구성·반복 완화·음수 사이클 검사까지 정상이다.

BELLMANFORD X 0 6 (Undirected)
무방향 모드에서도 동일 경로와 비용이 나오는 것은 getAdjacentEdges로 구성한 간선 리스트가 방향을 양방향으로만 확장할 뿐 최단 경로 구조는 그대로 유지된다는 것을 의미한다.

다시 말해 Bellman-Ford가 방향/무방향 옵션에 따라 입력 간선 집합만 변경하고 나머지 로직은 동일하게 잘 돌아가고 있다는 점을 확인한 단계다.

FLOYD O
방향 Floyd 결과 행렬에서 0행(0→*)이 Dijkstra(O)의 결과 거리(0 6 2 9 5 10 11)와 정확히 일치하므로 초기 행렬 구성과 3중 루프 동작이 Dijkstra와 상호 검증된다.
어떤 대각 원소도 음수가 아니므로 음수 사이클이 없는 그래프라는 판단도 코드와 수학적으로 일치한다.

FLOYD X
무방향 Floyd 결과 행렬이 대칭이고 각 행의 값이 무방향 Dijkstra로 계산한 거리와 일치하므로 getAdjacentEdges 기반 거리 초기화와 k를 통한 완화 과정이 무방향 그래프에서도 제대로 구현되었음을 보여준다.
이 행렬은 이후 Centrality 계산에 그대로 사용되기 때문에 여기서 값이 맞다는 것은 중앙성 계산의 입력이 신뢰 가능하다는 의미다.

CENTRALITY
Centrality 출력에서 각 정점의 값이 표현되고 정점 3이 6/32 ← Most Central로 표시된 것은 모든 정점에서 3까지의 거리 합이 최소라는 Floyd 결과와 일치한다.

```
========LOAD========
Success
====================

========PRINT========
    [0] [1] [2] [3] [4] [5] [6]
[0] 0 6 2 0 0 0 0
[1] 0 0 0 5 0 0 0
[2] 0 7 0 0 3 8 0
[3] 0 0 0 0 0 0 3
[4] 0 0 0 4 0 0 0
[5] 0 0 0 0 0 0 1
[6] 0 0 0 0 10 0 0
========================

========BFS========
Directed Graph BFS
Start: 0
0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6
========================

========BFS========
Undirected Graph BFS
Start: 0
0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6
========================

========DFS========
Directed Graph DFS
Start: 0
0 -> 1 -> 3 -> 6 -> 4 -> 2 -> 5
========================

========DFS========
Undirected Graph DFS
Start: 0
0 -> 1 -> 2 -> 4 -> 3 -> 6 -> 5
========================
```

```
========KRUSKAL========
[0] 2(2)
[1] 3(5)
[2] 0(2) 4(3)
[3] 1(5) 4(4) 6(3)
[4] 2(3) 3(4)
[5] 6(1)
[6] 3(3) 5(1)
Cost: 18
========================

========DIJKSTRA========
Directed Graph Dijkstra
Start: 0
[0] 0 (0)
[1] 0 -> 1 (6)
[2] 0 -> 2 (2)
[3] 0 -> 2 -> 4 -> 3 (9)
[4] 0 -> 2 -> 4 (5)
[5] 0 -> 2 -> 5 (10)
[6] 0 -> 2 -> 5 -> 6 (11)
========================

========DIJKSTRA========
Undirected Graph Dijkstra
Start: 0
[0] 0 (0)
[1] 0 -> 1 (6)
[2] 0 -> 2 (2)
[3] 0 -> 2 -> 4 -> 3 (9)
[4] 0 -> 2 -> 4 (5)
[5] 0 -> 2 -> 5 (10)
[6] 0 -> 2 -> 5 -> 6 (11)
========================

========BELLMANFORD========
Directed Graph Bellman-Ford
0 -> 2 -> 5 -> 6
Cost: 11
========================
```

```
========FLOYD========
Directed Graph Floyd
    [0] [1] [2] [3] [4] [5] [6]
[0] 0 6 2 9 5 10 11
[1] x 0 x 5 18 x 8
[2] x 7 0 7 3 8 9
[3] x x x 0 13 x 3
[4] x x x 4 0 x 7
[5] x x x 15 11 0 1
[6] x x x 14 10 x 0
=======================

========FLOYD========
Undirected Graph Floyd
    [0] [1] [2] [3] [4] [5] [6]
[0] 0 6 2 9 5 10 11
[1] 6 0 7 5 9 9 8
[2] 2 7 0 7 3 8 9
[3] 9 5 7 0 4 4 3
[4] 5 9 3 4 0 8 7
[5] 10 9 8 4 8 0 1
[6] 11 8 9 3 7 1 0
=======================

========CENTRALITY========
[0] 6/43
[1] 6/44
[2] 6/36
[3] 6/32 <- Most Central
[4] 6/36
[5] 6/40
[6] 6/39
=======================

========EXIT========
Success
====================
```

LOAD
두 번째 LOAD는 graph_M.txt를 인접 행렬 그래프로 로드

PRINT
출력된 행렬은 graph_L.txt에서 읽은 간선들을 행렬 형태로 옮긴 것과 정확히 일치하므로 ListGraph와 MatrixGraph가 동일 구조의 방향 그래프를 표현하고 있음을 확인할 수 있다.

BFS O 0
방향 BFS 결과가 앞선 리스트 버전과 동일하게 0 1 2 3 4 5 6으로 나오는 것은 인접 행렬에서 getAdjacentEdgesDirect로 꺼낸 out-degree 정보가 리스트 버전과 완전히 동일하다는 의미다.
즉 내부 저장 구조만 다를 뿐 BFS의 논리와 방문 순서는 완전히 일치한다.

BFS X 0
무방향 BFS에서도 역시 0 1 2 3 4 5 6 순서가 동일하게 나와 행렬 기반 getAdjacentEdges가 리스트 기반과 동일한 무방향 인접 집합을 구성한다는 것을 보여준다.

이로써 BFS에 대해 ListGraph/MatrixGraph 간 표현 독립성이 검증된다.

DFS O 0
방향 DFS 결과 0 → 1 → 3 → 6 → 4 → 2 → 5가 리스트 버전과 완전히 같으므로 인접 행렬에서 읽어온 인접 정점들의 정렬·순회 순서도 동일하게 유지되고 있다.
재귀 DFS 로직이 저장 방식에 의존하지 않고 Graph 인터페이스만 사용한다.

DFS X 0
무방향 DFS에서도 0 → 1 → 2 → 4 → 3 → 6 → 5 순서가 그대로 반복되는 것은 행렬 기반 무방향 인접 집합이 리스트 기반과 동일한 탐색 트리를 만들어 낸다는 뜻이다.
결과적으로 DFS 역시 두 표현 간 구현 차이로 인한 부작용 없이 일관성을 유지한다.

KRUSKA
행렬에서 추출한 무방향 간선으로 구성된 MST 인접 리스트와 cost 18이 리스트 버전의 결과와 완전히 동일하므로 행렬 기반 getAdjacentEdges로 수집한 간선 집합이 리스트 기반 것과 논리적으로 일치한다.

DIJKSTRA O/X 0
방향·무방향 Dijkstra에서 각 정점의 경로와 비용이 리스트 버전과 한 글자도 다르지 않으므로 행렬 기반 relax 연산이 리스트 기반과 동일한 그래프를 보고 있다는 점이 확인된다.
특히 0→6 경로가 항상 0→2→5→6 비용 11로 유지된다.

BELLMANFORD O/X 0 6
Bellman-Ford에서도 두 모드 모두 0 → 2 → 5 → 6 Cost 11이 동일하게 출력되어 리스트 버전과 일치하므로 edge list 구성 역시 행렬·리스트 구현 사이에 차이가 없다는 것을 의미한다.
이는 “음수 사이클 검출·도달 불가 처리·경로 재구성” 전 과정을 통틀어 내부 표현에 따른 오차가 없음을 보여준다.

FLOYD O/X
Floyd 결과 행렬이 리스트 버전과 값까지 완전히 동일하므로 모든 세부 구현이 문제 없다는 것을 알 수 있다.
0행과 각 행의 대칭성 그리고 대각 원소가 0인 점까지 동일하기 때문에 두 구현이 같은 알고리즘을 정확히 실행하고 있다는 사실을 재확인한다.

CENTRALITY
Centrality 결과 역시 모든 정점의 값과 3번 정점만 Most Central이라는 결론이 리스트 버전과 완전히 같으므로 무방향 Floyd 결과를 기반으로 한 중심성 계산이 표현 방식과 상관없이 동일하게 수행된다.
따라서 Centrality까지 포함한 전체 파이프라인이 ListGraph MatrixGraph 양쪽 모두에서 일관되게 동작하고 있다는 최종 검증이 된다.

EXIT

마지막으로 EXIT 명령이 Success로 종료되었다는 것은 명령 파싱 루프가 중간에 크래시 없이 끝까지 내려온 뒤 정상적으로 프로그램을 종료했다는 의미다.
이 시점까지의 로그에 에러 코드가 없으므로(에러 테스트를 할 때를 제외하면) 전체 실행 흐름이 설계 의도대로 마무리된 상태라고 볼 수 있다.

```
==11989== Memcheck, a memory error detector
==11989== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==11989== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==11989== Command: ./run
==11989==
==11989==
==11989== HEAP SUMMARY:
==11989==     in use at exit: 0 bytes in 0 blocks
==11989==   total heap usage: 743 allocs, 743 frees, 140,157 bytes allocated
==11989==
==11989== All heap blocks were freed -- no leaks are possible
==11989==
==11989== For lists of detected and suppressed errors, rerun with: -s
==11989== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

마무리로 메모리 누수에 대해서 확인하였다.

Consideration

이번 과제를 통해서 여러 노드 데이터 인접 데이터 등등 관리 기법들을 수행해볼 수 있었다. 이전에는 음수까지 관리 가능한 알고리즘만 알고 있으면 되는것이 아닐까? 라는 생각을 했고 항상 무조건적으로 좋은 코딩 방법이 있지 않을까? 했지만 이번 과제를 하면서 간선이 음수냐 양수냐 뿐만 아니라 다양한 0 x 조건에 따른 분기등에 대해서 관리할 수 있었습니다. 감사합니다.