2023 겨울학기 연구참여 보고서

Graph Partitioning : Connectivity

2024.01.02. ~ 2024.02.08. (6주)

Logistics Lab (물류 연구실)

지도교수: 김병인 교수님

지도선배: 이승엽 선배님

참여학생: 신정호

목차

1 Introduction

기존 과제에서는 다중 차량 경로 생성 문제(VRP)를 여러 개의 외판원 문제(TSP)로 바꾸어 풀기 위해서 Balanced Graph Partition을 이용한다. Graph Partition이 원활하게 이루어지기 위해서는 주어진 graph data에서 연결성이 보장되어야 한다. 하지만, raw data 및 sample data에서는 이 연결성이 보장되지 않아, 퀄리티 높은 Graph Partition을 수행할 수 없다는 문제점이 존재한다.

이를 해결하기 위해 본 연구에서는 무방향그래프(Undirected Graph)와 방향그래프(Directed Graph), 각각에서 연결성이 보장되도록 데이터를 전처리할 수 있는 프로그램을 구성하고자 한다. 합리적인 시간 내에 연결성을 확인하는 것이 최우선 목표로 설정하여, 각 데이터 형식에 맞춘 적절한 알고리즘을 선정할 수 있도록 다양한 분석을 진행할 계획이다.

2 연구과정

2.1 사용 데이터

본 연구에서 사용할 data는 edgeid, fnode, tnode, ftcostsec, tfcostsec, edgelengthcm, oneway, cross, edgetype, emnix, eminy, emaxx, emaxy, edgevertices, onesidepickup에 대한 정보가 tab으로 구분(tab-delimited data)되어있다. Figure 1은 사용데이터를 엑셀 프로그램을 통해 시각화한 것이다.

edgeid	fnode	tnode	ftcostsec	tfcostsec	edgelengthcm	oneway	cross	edgetype	emnix	eminy	emaxx	emaxy	edgevertic	onesidepickup
a949fa68-	00004342-	c9eb97b2	6.5	6	9006	1	0	6	-123.53	48.3812	-123.529	48.38188	-123.5292	1
ee3c5e5b	00004342-	e8f92d0a-	2.9	5.8	3053	1	0	7	-123.538	48.38179	-123.538	48.38203	-123.5378	1
bd000aa9	00004342-	eba143d2	3.9	3.8	4087	1	0	6	-123.538	48.38184	-123.537	48.38203	-123.5378	1
e0a7618b	a6f9c4e5-	00004342	1.7	1.6	2464	1	0	6	-123.529	48.38188	-123.529	48.38206	-123.5290	1
00004342	00004342-	00004342	56.2	60	98324	1	0	6	-123.547	48.38203	-123.538	48.3883	-123.5471	1
d1e2cd2d	00004342-	8dfe8baf-	34.4	34.4	21024	1	0	7	-123.538	48.38203	-123.536	48.38362	-123.5378	1
a284670d	00004342-	a6f9c4e5-	18.5	17.9	28280	1	0	6	-123.529	48.38206	-123.527	48.38418	-123.5269	1
b422015e	b37fb25f-	00004342	42.8	42.8	34439	1	0	7	-123.523	48.38241	-123.519	48.38317	-123.5192	1
00004342	00004342-	00004342	31.1	26.1	26789	1	0	6	-123.527	48.38293	-123.523	48.38407	-123.5234	1
00004342	00004342-	00004342	8.8	8.8	6853	1	0	7	-123.523	48.38293	-123.523	48.38347	-123.5230	1

Figure 1 사용 데이터

한편, ftcostsec와 tfcostsec는 fromnode와 tonode 사이를 이동하는데 걸리는 시간을 의미하는데, 이 cost가 999999일 경우에는 해당 방향으로 이동할 수 없다는 의미를 가진다.

연결성 확인에는 edge과 node를 구분하기 위한 각자의 ID, 연결성을 확인하기 위한 node 사이의 이동시간에 대한 정보만을 필요로 한다. 이에 본 프로그램에서는 edgeid, fnode, tnode, ftcostsec, tfcostsec의 5가지의 data만을 사용한다.

2.2 Undirected Graph

Undirected Graph는 각 edge가 방향을 가지지 않는 그래프를 의미한다. 이 그래프에서는 노드 간의 연결 관계가 양방향으로 존재한다. 이는 노드 A에서 노드 B로 이동할 수 있다면, 반대로도 노드 B에서 노드 A로 이동할 수 있는 것을 의미한다.

기존 데이터에는 방향성이 존재한다. 하지만, 우선 주어진 문제를 간단하게 해결해보기 위해 모든 edge에 방향성이 없다고 가정한 상태에서 프로그램을 구성해보기로 했다. 이에 Undirected Graph로 가정한 프로그램에서는 edgeid, fnode, tnode의 정보만 활용하여 프로그램이 작동하도록 구성했다.

2.2.1 Connected Component

Connected Component는 그래프 내에서 모든 노드가 서로 연결되어 있는 부분을 의미한다. 어떤 그래프에 속하는 subgraph 중에서 그래프 내의 임의의 두 노드 사이에 경로가 존재하는 경우 이를 Connected Component라고 한다. 다시 말해, Connected Component는 그래프 내의 모든 노드가 서로 연결되어 있는 subgraph를 의미한다. 이는 Undirected Graph에서 "연결성"을 확인하는 데 주요 사용되는 개념이다.

만약 그래프가 단일 연결 요소(single connected component)로 이루어져 있다면, 이는 전체 그래프가 한 개의 Connected Component로 이루어져 있다는 것을 의미한다. 그러나 그래프가 여러 개의 부분 그래프로 나뉘어져 있다면, 각 부분 그래프는 독립된 Connected Component로 간주된다.

본 연구에서는 edge가 어떤 Connected Component에 속하는지를 1-based indexing으로 구분하는 것을 통해 연결성을 확인하고자 한다.

2.2.2 BFS

Undirected Graph에서 Connected Component를 찾아 indexing 할 수 있는 가장 쉬운 방법은 바로 BFS(Breadth-First Search, 너비 우선 탐색)를 활용하는 것이다. BFS는 그래프나 트리를 탐색하는 데 사용되는 알고리즘 중 하나로, 시작 노드로부터 시작하여 인접한 모든 노드를 먼저 탐색한 후, 그 다음 레벨의 노드들을 탐색한다. 이 알고리즘은 큐나 재귀함수를 사용하여 구현되기 때문에 만약 서로 다른 edge나 node가 연결되어 있다면 한 번에 탐색이 되기 때문에 같은 Connected Component를 구할 수 있고, 그 과정에서 indexing을 처리할 수 있다.

본 연구에서는 먼저 재귀함수로 BFS를 구현해보았으나, 자료의 크기가 너무 커서 재귀함수 호출의 한계를 넘어가는 문제가 발생했다. 이에 BFS의 진행을 queue를 활용하도록 수정하여 프로그램이 멈추는 것을 방지했다.

2.3 Directed Graph

Directed Graph는 각 edge가 방향을 가지는 그래프를 의미한다. 즉, Directed Graph는 노드들 사이의 관계가 방향을 가지며, 각 edge는 출발점과 도착점이 명확하게 정의된다. 이때, Directed Graph에서는 기존의 edge와는 구별이 되도록 두 node 사이의 연결을 arc라고 한다. Directed Graph에서 각 arc는 일반적으로 화살표로 표현되며, 화살표의 방향은 해당 arc를 따라 이동할 수 있는 방향을 나타낸다. 예를 들어, 도로망을 나타내는 그래프에서 Directed Graph는 각 도로가 특정 방향으로만 이동 가능한 경우를 나타낼 수 있다.

2.3.2 Strong Connected Component

Undirected Graph에서는 Connected Component를 통해 연결성을 정의했다. 그러나 Directed Graph에서는 Connected Component를 통해 연결성을 정의하게 되면 모호한 부분이 존재하게 된다. 예를 들어, A에서 B로 향하는 arc가 있다고 하면, A 입장에서 B는 Connected Component지만, B 입장에서는 A가 Connected Component가 아니게 되는 문제가 발생하게 된다.

이에 Directed Graph에서는 Strong Connected Component라는 개념을 정의한다. Strong Connected Component는 Directed Graph에서의 연결성을 나타내는 개념으로, 그래프 내에서 강한 연결성을 가지는 subgraph를 의미한다. 구체적으로 Strong Connected Component 내의 모든 노드들은 서로에게 도달할 수 있으며, 한 Strong Connected Component 내의 임의의 두 노드 사이에는 방향성을 따라 경로가 존재하는 경우 같은 Strong Connected Component에 속하게 된다. 만약 Strong Connected Component 내의 한 노드에서 다른 노드로 이동할 수 있다면, 그 반대의 방향으로도 이동할 수 있다는 특징을 가지기 때문에 연결성을 정의하는데 있어서 모호한 부분이 존재하지 않는다.

본 연구에서는 edge data를 받아서 사용하기 때문에 연결성에서도 edge의 연결성을 확인해야 한다. 그러나, 위에서 정의한 Strong Connected Component는 node를 기반으로 한 연결성을 의미한다. 이 때문에 node 사이에 위치한 edge의 연결성을 어떻게 정의해야하는지에 대한 문제가 발생한다. 이를 해결하기 위해 edge를 virtual node로 선언하여 일종의 node로 판단하는 방식을 통해 edge의 연결성을 확인한다.

2.3.3 Kosaraju Algorithm

Kosaraju Algorithm은 Directed Graph에서 Strong Connected Component를 찾는 알고리즘 중 하나로, 각 node 사이의 강한 연결성을 식별하는 데 사용된다. 이 알고리즘은 두 번의 DFS(Depth-First Search, 깊이 우선 탐색)을 통해 이루어진다. 구체적인 진행 방식은 아래와 같다.

먼저, 원본 그래프의 각 노드에 대해 DFS를 수행한다. 이때, 방문한 노드들을 역순으로 stack에 넣는다. 이후에는 기존의 그래프에서 arc의 방향을 반대로 바꾼, 역방향 그래프를 구성한다. 마지막으로 역방향 그래프에서 stack에 담긴 노드 순서대로 다시 DFS를 수행한다. 여기서 한 번의 DFS 실행에서 방문한 노드들은 같은 Strong Connected Component에 속하는 요소라고 판단할 수 있다.

이렇게 Kosaraju Algorithm은 두 번의 DFS를 통해 Directed Graph의 Strong Connected Component를 효과적으로 찾아내는 데 사용된다.

3 연구결과

위에서 설명한 알고리즘을 활용한 결과, 연결성을 만족하지 않는 데이터를 제거하는 전처리 과정을 효과적으로 수행할 수 있었다. Figure2 는 기존의 데이터를 시각화한 것이다. Figure3 는 기존의 데이터를 연결 요소에 따라 다른 색으로 시각화한 것이다. Figure4 는 기존의 데이터에서 가장 큰 연결 요소만을 남기도록 전처리 한 후, 시각화 한 것이다.

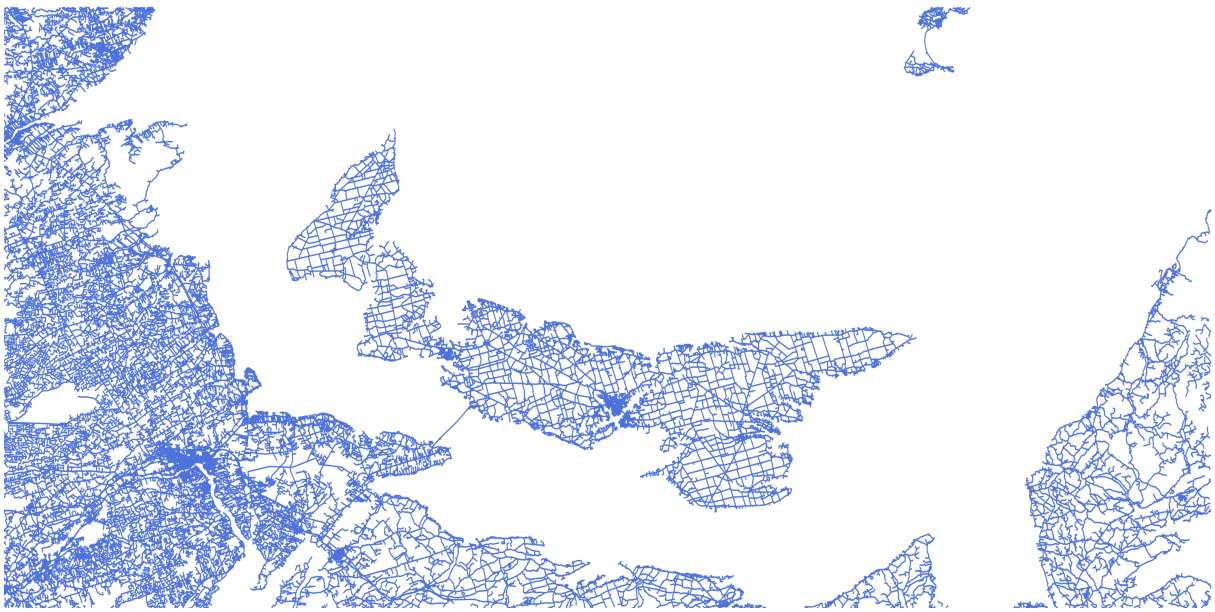

Figure 2 기존 데이터

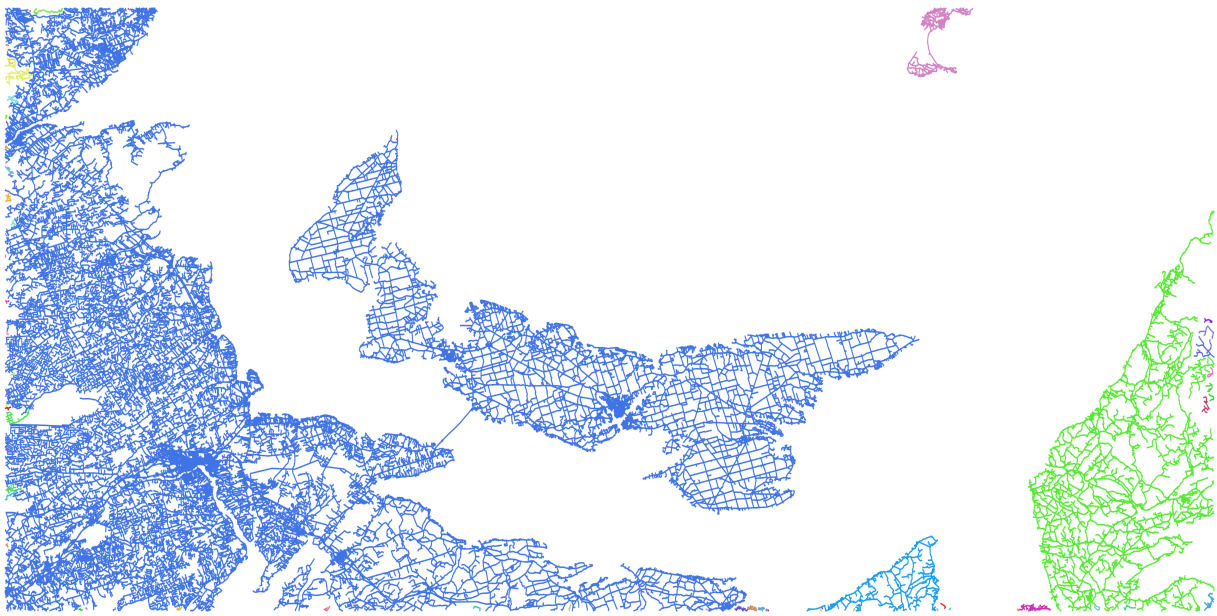

Figure 3 기존 데이터의 연결 요소

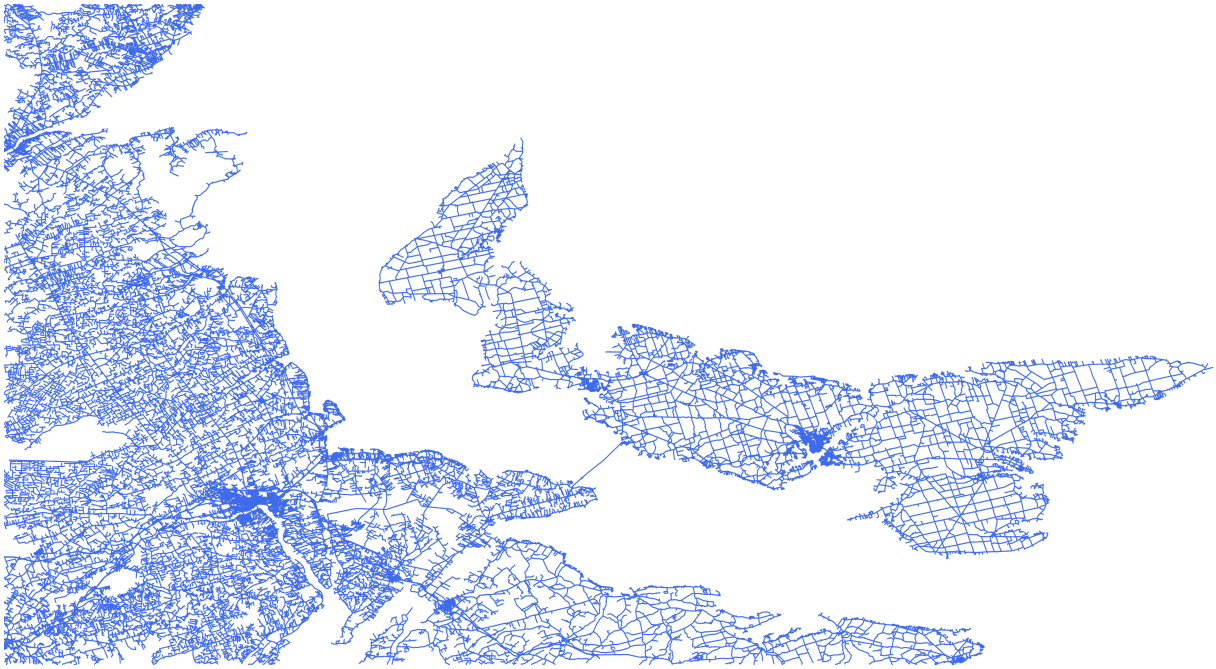

Figure 4 전처리 이후 데이터

전체적인 프로그램의 실행 시간은 아래와 같다. Figure 5 는 Undirected Graph 에서 알고리즘 소모 시간이다. Figure 6 은 Directed Graph 에서 알고리즘 소모 시간이다.

- Input Data

	Edges (개)	Nodes (개)
Data1	15960	13518
Data2	20050	17465
Data3	36399	28290
Data4	102933	91658
Data5	332673	281127

- 소모 시간

ReadFile	Clustering	WriteFile	Total
431 ms	**10 ms**	**164 ms**	**605 ms**
550 ms	**14 ms**	**226 ms**	**790 ms**
895 ms	**24 ms**	**449 ms**	**1368 ms**
2913 ms	**96 ms**	**1383 ms**	**4392 ms**
9166 ms	**381 ms**	**3554 ms**	**13101 ms**

Figure 5 Undirected Graph 알고리즘 소모 시간

• Input Data

	Arcs (개)	Nodes (개)
Data1	29268	13518
Data2	39027	17465
Data3	66430	28290
Data4	201346	91658
Data5	608322	281127

• 소모 시간

ReadFile	Clustering	WriteFile	Total
528 ms	**219 ms**	242 ms	**989 ms**
619 ms	**255 ms**	346 ms	**1220 ms**
1121 ms	**574 ms**	533 ms	**2228 ms**
2717 ms	**1679 ms**	1650 ms	**6046 ms**
8624 ms	**5202 ms**	4082 ms	**17908 ms**

Figure 6 Directed Graph 알고리즘 소모 시간

4 결론 및 후기

이번 연구를 통해 짧은 시간에 graph data가 연결성을 만족하도록 하는 전처리 프로그램을 제작할 수 있었다. 다음에는 METIS라는 graph partitioning 라이브러리를 학습하고, 주어진 data 및 balace 조건 (load balance, visually attractive, Overlap)에 맞도록 프로그램에 적용하는 활동을 이어서 하고 싶다.

이전부터 가지고 있던 대학원에서의 연구 및 공부에 대한 궁금증을 이번 연구참여를 통해 해소할 수 있었습니다. 고등학교와 대학교에서 수동적으로 지식을 습득하는 것에서 벗어나, 주어진 실생활 문제를 해결하기 위해 능동적으로 학습하는 과정에서 대학원 생활에 대한 흥미를 느낄 수 있었습니다.

지금까지는 교과목에서 제공하는 과제를 해결하기 위해 조건이 주어진 환경에서 프로그래밍을 진행하는 것에 익숙해져 있었습니다. 하지만 이번 연구 참여를 통해 입출력 파일의 형식, 알고리즘, 자료구조 등의 선택을 유동적으로 내가 선택해서 진행하는 경험을 할 수 있었습니다. 이러한 경험을 통해 알고리즘과 자료구조의 상호보완적 관계에 대해 명확히 이해할 수 있었습니다.

연구참여 기간 동안 다양한 경험을 할 수 있어서 좋았습니다. QGIS 라는 도로 정보 시각화 도구를 활용했던 것, 랩 미팅과 세미나, 또 연구실 선배님들과의 상담과 같은 경험을 통해 6 주라는 연구참여 기간동안 의미있는 시간을 보낼 수 있었던 것 같습니다.

첫 번째 연구참여를 훌륭한 교수님과 선배님 밑에서 관심있는 주제로 진행할 수 있었던 것이 매우 행운이었다고 생각합니다. 6 주라는 긴 시간동안 원활하게 연구참여를 진행할 수 있도록 많은 도움을 주신 김병인 교수님, 이승엽 사수님, 또 다른 물류 연구실 선배님들에게 감사의 말씀을 드리고 싶습니다.

Appendix (code)

Connectivity 에 따른 Graph partitioning 에 사용된 코드의 주요 함수는 아래와 같다.

Undirect Graph (BFS)

```
int Graph::Clustering() {
    int maxNodeEdgeCount = 0;
    int maxClusterCount = 0;
    int clusterCount = 1;

    // Iterate through each node in the graph
    for (auto& entry : nodeMap) {
        Node* startNode = entry.second;
        int nodeCount = 0;
        int edgeCount = 0;

        if (startNode->visited)
            continue;

        startNode->visited = true;
        startNode->index = clusterCount;

        if (startNode->isEdge)
            edgeCount++;
        else
            nodeCount++;

        queue<Node*> q;
        q.push(startNode);

        // BFS traversal
        while (!q.empty()) {
            Node* currentNode = q.front();
            q.pop();

            // Process each adjacent edge
            for (Node* adjNode : currentNode->adjNodes) {
                if (!adjNode->visited) {
                    adjNode->visited = true;
```

```
                    adjNode->index = clusterCount;

                    if (adjNode->isEdge)
                        edgeCount++;
                    else
                        nodeCount++;

                    // Add the unvisited adjacent node to the queue
                    q.push(adjNode);
                }
            }
        }

        // Update max counts if necessary
        if (maxNodeEdgeCount < nodeCount + edgeCount) {
            maxNodeEdgeCount = nodeCount + edgeCount;
            maxClusterCount = clusterCount;
        }

        // Store node and edge counts
        nodeEdgeCount.push_back(make_pair(nodeCount, edgeCount));

        // If BFS traversal occurred, increment the cluster count
        clusterCount++;
    }

    return maxClusterCount;
}
```

Direct Graph (Kosaraju Algorithm)

```
void Graph::DFS(Node* startNode, stack<string>& s) {
    stack<string> DFSStack;
    DFSStack.push(startNode->ID);

    while (!DFSStack.empty()) {
        Node* currentNode = nodeMap[DFSStack.top()];
        DFSStack.pop();
```

```
        if (!currentNode->visited) {
            currentNode->visited = true;
            s.push(currentNode->ID);

            for (Node* adjNode : currentNode->adjNodes) {
                if (!adjNode->visited) {
                    DFSStack.push(adjNode->ID);
                }
            }
        }
    }
}

pair<int, int> Graph::DFSUtil(Node* startNode, int clusterCount) {
    stack<string> nodeStack;
    nodeStack.push(startNode->ID);

    int nodeCount = 0;
    int arcCount = 0;

    while (!nodeStack.empty()) {
        Node* currentNode = nodeMap[nodeStack.top()];
        nodeStack.pop();

        if (!currentNode->visited) {
            currentNode->visited = true;
            currentNode->index = clusterCount;

            if (currentNode->isEdge)
                arcCount += currentNode->isEdge;
            else
                nodeCount++;

            for (Node* adjNode : currentNode->reverse_adjNodes) {
                if (!adjNode->visited) {
                    nodeStack.push(adjNode->ID);
                }
            }
        }
```

```
    }
    return make_pair(nodeCount, arcCount);
}

void Graph::getTranspose() {

    // Copy edges with reversed direction
    for (auto& entry : nodeMap) {
        Node* originalNode = entry.second;
        for(Node* adjNode : originalNode->adjNodes)
            adjNode->reverse_adjNodes.push_back(originalNode);
    }
}

int Graph::Kosaraju() {
    stack<string> stack;

    // First DFS to fill the stack
    for (auto& entry : nodeMap) {
        Node* node = entry.second;
        if (!node->visited && node->isEdge)
            DFS(node, stack);
    }

    // Reverse the graph
    getTranspose();

    // Reset visited flags
    for (auto& entry : nodeMap) {
        entry.second->visited = false;
    }

    // Second DFS to find SCCs
    int maxNodeEdgeCount = 0;
    int maxClusterCount = 0;
    int clusterCount = 1;
    while (!stack.empty()) {
        string currentNodeID = stack.top();
        stack.pop();
```

```
            Node* currentNode = nodeMap[currentNodeID];

            if (!currentNode->visited && currentNode->isEdge) {
                pair<int, int> count = DFSUtil(currentNode, clusterCount);

                // Update max counts if necessary
                if (maxNodeEdgeCount < count.first + count.second) {
                    maxNodeEdgeCount = count.first + count.second;
                    maxClusterCount = clusterCount;
                }

                // Store node and edge counts
                nodeArcCount.push_back(count);
                clusterCount++;
            }
        }
        return maxClusterCount;
}
```