Algorithm HW3

Strongly Connected Components

2013-11381
강지원

● 구현 환경 및 실행

구현은 java 언어를 사용하였으며, "1.8.0_171" 버전의 환경에서 구현하였다. 소스 파일 구성은 < AdjacencyMatrix.java, AdjacencyList.java, AdjacencyArray.java, SCC.java > 로 총 4개의 java 파일로 구현하였다. 아래의 명령어를 통해서 컴파일 및 실행 할 수 있다. 제출은 컴파일이 된 상태로 제출 했기 때문에, 컴파일은 실행하지 않고 'hw3' 폴더에서 실행만 하면 된다.

Compile : `javac *.java -d .`

Execute : `java alg_hw3.SCC`

위의 명령어를 통해서 실행하면, 아래와 같이 실행된다. 가장 먼저 인접 행렬을 통해 찾은 SCC와 수행 시간을 출력한다. 다음 인접 리스트와 인접 행렬을 통해 구한 SCC와 수행 시간이 차례로 출력된다.

```
jiwon@jiwon-VirtualBox:~/alg_hw/hw3$ javac *.java -d .
jiwon@jiwon-VirtualBox:~/alg_hw/hw3$ java alg_hw3.SCC
3
2 2 3
0
1 1
1 3
2
Adjacency Matrix time : 1
1 3
2
Adjacency List time : 0
1 3
2
Adjacency Array time : 0
```

● 구현할 내용

주어진 그래프에서 어떠한 노드 u에서 다른 노드 v로 향하는 경로가 존재하고, 그 반대인 v에서 u로 향하는 경로도 존재하는 'Strongly Connected Components'를 찾는 알고리즘을 구현하였다. 이때, 그래프를 표현하는 방식으로 인접 행렬, 인접 리스트, 인접 배열 세 가지로 구현하였고, 각각 표현 방식의 시간을 측정하였다.

● 구현 방법

java 언어의 "1.8.0_171" 버전의 환경에서 구현하였다. 각각의 그래프를 표현할 인접 행렬과 인접 리스트, 인접 배열을 클래스를 활용해서 구현하였고, SCC 알고리즘 또한 각각의 클래스에 메소드로 구현하였다. 이 세 클래스를 활용해서 인풋 파일에 대한 처리, 출력 등을 하는 'SCC.java' 파일을 메인으로 구현하였다.

SCC 알고리즘은 '코라사주 알고리즘'을 이용해서 구현하였다. 코라사주 알고리즘은 다음과 같다.

① 그래프 G에 대해 DFS를 수행하여 각 정점의 완료 시간을 계산한다.

② G의 방향을 뒤집은 역그래프를 만든다.

③ 2번에서 만든 역그래프를 1번의 완료 시간의 역순으로 DFS를 수행한다.

④ 3에서 만들어진 분리된 트리가 Strongly Conneted Component 이다.

1. Adjacency Matrix

인접 행렬은 2차원 배열을 이용해서 구현하였다. 구현의 편의성을 위해서 입력된 정점의 개수에 하나를 더해 구현해서 0번 요소는 사용하지 않았다. Matrix[x][y]일 때, x에서 y로 직접 가는 길이 있는 경우 1, 그렇지 않은 경우 0으로 표현하였다. 역 그래프는 전치 행렬을 구하는 방식과 똑같이 x와 y의 위치를 변경해서 구현하였다.

2. Adjacency List

인접 리스트는 자바의 'Array List'를 이용해서 구현하였다. Integer를 저장하는 Array List 하나와 이 Array List를 저장하는 Array List를 사용하였다. 0번 위치의 Array List는 비워두고, 1번 위치부터 각 정점에서 직접 가는 길이 있는 정점을 저

장하였다. 따라서, list.get(x)에 {1, 2, 3}이 나온다면 x에서 1, 2, 3번 정점으로 직접 가는 길이 존재하는 것이다. 역 그래프는 위와 같은 경우, 1, 2, 3의 에서 x로 가는 길이 존재한다는 방식으로 구현하였다.

3. Adjacency Array

인접 배열 또한 행렬과 비슷하게 2차원 배열을 이용해서 구현하였다. 구현의 편의성을 위해서 미리 (N+1) * (N+1)개의 배열을 선언하였다. 0행은 사용하지 않는 배열이다. 따라서 각 행의 번호가 정점을 나타내게 된다. 각 행의 0열은 각 정점이 가지고 있는 직접 다른 정점으로 갈 수 있는 길의 개수이다. 1번 정점에서 2, 3으로 직접 가는 길이 존재하면 1행의 배열은 [2, 2, 3]이 된다. 역 그래프는 직접 갈 수 있는 길을 0으로 초기화 해놓고 시작하였다. 앞의 1행의 예를 들어 설명하면, 2가 입력 될 때, 2행의 초기화 해놓은 갈 수 있는 길에 1을 더하고 해당 번호에 1을 추가하여 구현하였다.

● 실험 및 결과

1. Sparse / Dense Case

입력 되는 그래프가 sparse한 경우와 dense한 경우를 비교하기 위해서 정점의 개수가 1000개인 그래프를 두 개 만들어서 실험하였다. 이때 sparse한 경우는 각 정점 당 0~2개의 엣지를 가지고, dense한 경우는 각 정점 당 999개의 엣지를 가진다. 실험은 각각 3회씩 실행하였고, 수행 시간의 평균을 기록하였다. 결과는 아래와 같다.

	Sparse	**Dense**	**(단위 : ms)**
인접 행렬 :	40.0	51.3	
인접 리스트 :	4.3	151.7	
인접 배열 :	9.3	30.7	

인접 행렬의 경우 두 경우 모두 큰 차이가 없다. 그래프의 연결 관계와 상관 없이 모든 정점과의 관계를 검사하기 때문인 것으로 예상된다. 인접 리스트의 경우에는 정점의 연결 개수가 늘어나는 만큼 검사를 해야 하는 횟수가 증가하기 때문에 수행 시간의 큰 차이가 나타난다. 인접 행렬의 경우도 비슷한 이유로 차이를 나타낸다. 하지만 인접 리스트의 경우보다 큰 차이가 없는데, 이유는 리스트

에서 다음 요소로 넘어 갈 때 호출 하는 등의 오버헤드가 없기 때문인 것으로 예상된다.

2. 그래프의 정점 개수

그래프의 정점 개수 증가에 따른 수행 시간을 측정하기 위해서 정점의 개수가 100개, 500개, 1000개인 그래프를 만들어서 실험하였다. 각 정점에서 나가는 엣지의 수는 전체 정점의 수의 1/3로 설정하여 만들었다. 각 그래프 별로 3번의 실험을 하였고, 수행 시간의 평균을 기록하였다. 결과는 아래와 같다.

	100	**500**	**1000**	**(단위 : ms)**
인접 행렬 :	2.7	25.0	50.0	
인접 리스트 :	2.0	28.0	63.0	
인접 배열 :	1.0	16.0	20.0	

전체적으로 인접 배열, 인접 행렬, 인접리스트 순서로 빠른 수행 시간을 보인다. 인접 리스트의 경우 DFS 시 리스트의 다음 요소로 넘어가는 오버헤드가 커서 수행 시간이 길어진 것으로 예상된다. 인접 행렬과 인접 배열의 경우, 행렬은 정점 사이의 연결 유무에 상관없이 모든 정점 간의 관계를 확인 하기 때문에 연결 되어 있는 정점만 확인하는 인접 배열에 비해서 수행 시간이 길어진 것으로 예상된다.