Algorithm hw3 Report

2013-11425 이창영

1. 구현환경, 실행방법

java를 사용하여 구현하였다.
압축을 해제한 뒤
컴파일 : javac *.java
실행 : java SCC
을 통해 실행할 수 있다.
실행 완료된 모습은 아래와 같다. (실행시간 단위 : ms)

```
C:₩Users₩CY₩Desktop₩test>javac *.java
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

C:₩Users₩CY₩Desktop₩test>java SCC
3
2 2 3
0
1 1
adjacency matrix result
1 3
2
adjacency matrix executing time: 00

adjacency list result
1 3
2
adjacency list executing time: 01

adjacency array result
1 3
2
adjacency array executing time : 00
```

2. 구현내용, 방법

1) SCC 구하는 알고리즘
a. 임의의 노드에서 DFS를 수행한다.
b. 아직 방문하지 않은 노드가 있다면 그 노드에서 다시 DFS를 수행하는 것을 반복하여 모든 정점을 stack에 담도록 한다.
c. 원래그래프의 역방향 그래프에 대해 이전에 저장한 stack의 top 노드에서부터 DFS를 수행한다. 이 때 탐색되는 정점들을 묶으면 하나의 SCC가 된다.
d. stack이 비어있을 때까지 c를 반복하며 pop 하는데 이미 stack의 top 노드가 방문한 노드이면 pop만 해준다.

2) adjacency matrix
* 1~n 의 노드를 0~n-1의 노드라고 생각하고 구현한 뒤 마지막에 +1 해서 출력하였다.
a. 주어진 인풋을 가지고 n*n matrix를 구성하였다. 이때 역방향 그래프를 담고 있는 matrix도 동시에 구성하였다.
b. matrix를 구현하는 방식은 a->b 엣지가 있다면 m[a-1][b-1] 에 1을 넣어주는 방식이다.
c. 두 matrix를 가지고 1)에 작성한 SCC 구하는 과정을 적용하여 SCC를 모두 찾는다.
d. 정렬한 뒤 출력한다.

3) adjacency list
* 1~n의 노드를 0~n-1의 노드라고 생각하고 구현한 뒤 마지막에 +1 해서 출력하였다.
a. 주어진 인풋을 가지고 list배열을 구성하였다. 이때 역방향 그래프를 담고 있는 list배열도 동시에 구성하였다.
b. list 배열의 크기는 노드의 개수이고, a가 갈 수 있는 노드를 list[a-1]에 매다는 방식으로 구현하였다.
c. 두 list 배열을 가지고 1)에 작성한 SCC 구하는 과정을 적용하여 SCC를 모두 찾는다.
d. 정렬한 뒤 출력한다.

4) adjacency array
a. 주어진 인풋을 가지고 이차원 배열을 구성하였다. 이 때 역방향 그래프를 담고 있는 배열도 동시에 구성하였다.
b. 이차원 배열의 크기는 (노드개수+1) * (노드개수+1)이고 a->b 엣지가 있다면 array[a]의 가장 앞 빈 쪽에 b를 넣었다. 그리고 array[x][0] 자리에는 노드 x가 몇 개의 outgoing edge를 가지고 있는지 저장하였다.
c. 두 이차원 배열을 가지고 1)에 작성한 SCC 구하는 과정을 적용하여 SCC를 모두 찾는다.
d. 정렬한 뒤 출력한다.

3. 결과 분석, 결론
* 자료구조를 구성하는 시간, 결과를 정렬하고 출력하는 시간은 빼고 측정하였다.
1) sparse와 dense 비교
1000개의 노드로 구성하였고 sparse는 노드당 0~2개의 edge 갖게 구성하였고 dense는 모든 노드가 연결되도록 구성하였다. 결과는 아래와 같다.

단위(ms)	matrix	list	array
sparse	139	5	92
dense	191	1509	202

matrix의 경우에는 sparse인 경우에도 모든 다른 노드에 대해 연결되어있는지 아닌지 확인하게 되므로 아무리 sparse해도 일정 시간 이상 줄어들지 않는다.
반면에 list와 array는 연결되어있는 노드만 바로 파악할 수 있으므로 sparse하냐 dense하냐에 따라 그 차이가 명확하게 나게 된다. list의 경우 java에서 제공하는 LinkedList를 사용하였는데 get(i)를 통한 index접근이 overhead가 큰데 빈번하게 발생해서 그런지 dense해질수록 실행시간이 크게 늘어나는 것 같다. 이론적으로 array가 matrix에비해 더 빠르게 작동해야하는데 array를 구현할 때 비효율성이 있었는지 dense할 때 array가 좀 느려지는 것을 볼

수 있다. 이를 해결하기 위해선 어떤 노드에 대해 갈 수 있는 노드를 모두 살펴보는 부분을 최적화할 필요가 있을 것 같다.

2) 노드 개수에 따라 비교
각각의 노드가 전체 노드의 1/3로 갈 수 있도록 일정하게 테스트케이스를 만들어서 테스트하였다.

	matrix	list	array
100개	5	3	2
500개	83	64	43
1000개	192	248	142

위에서 작성하였듯이 list는 개수가 적을 때는 빠르게 작동하였지만 edge가 많아지면 어느순간부터 수행시간이 크게 증가하게 된다. array와 matrix는 그에 비해 overhead가 작아서 list에 비해 수행시간이 크게 증가하지는 않는다.