HW3 Strongly Connected Components

2016-17097 유준열

0. 환경 및 실행 방법

프로그램은 Python 3.6.8. 가상 환경에서 작성하였으며 외부 라이브러리인 NumPy를 사용한다. 프로그램 실행 방법은 다음과 같다.

```
(Terminal)
python3 ./hw3.py $input_path $output_path $type[adj_mat/adj_list/adj_arr]

(Result)
Terminal 에 소요된 시간이 표시되며, 지정한 output path 에 수행 결과가 출력된다.
```

1. Graph & Strongly Connected Components (SCC)

Graph는 정점 (vertex) 간 관계를 모델링하는 자료구조이다. Graph는 간선의 방향성 유무에 따라 유향 / 무향으로 나누어지며, vertex를 순차적으로 방문하는 방법 중 하나로 DFS를 갖는다. Graph는 일반적으로 인접 행렬 (Adjacency matrix) / 인접 리스트 (list) / 인접 배열 (array) 로 표현된다.

SCC는 주어진 유향그래프에서 서로에게 도달하는 경로가 있는 정점의 집합이다. 모든 정점은 기본적으로 자기 자신과 SCC 관계이며, 서로 다른 두 SCC가 하나의 정점을 공유한다면 두 SCC는 한 개의 SCC로 통합 (merge)된다. SCC를 찾는 알고리즘은 DFS를 활용하며, 다음과 같이 표현된다.

```
SCC(G)
{
        1. Graph G에 DFS를 수행하고, 각 vertex의 finish time을 기록한다
        2. G의 전치행렬인 G^R을 만든다
        3. G^R에 대해 1. 에서 구한 finish time의 역순으로 DFS를 수행한다.
        4. 3. 에서 만들어진 각 트리를 SCC로 return한다.
}
```

2. Implementation

Graph의 서로 다른 표현을 class로 정의하였다. 각 class는 원소를 더하는 addEntity() 와 graph를 전치하는 transpose() 함수를 갖는다. 각 class에 대한 DFS는 서로 다른 dfs_$type[matrix / list / array]() 함수를 이용해 구현하였으나, 세부 로직은 거의 동일하다. SCC 알고리즘 중 finish time의 기록은 스택을 이용하여, 임의의 정점에서 더이상 방문 되지 않은 정점이 없을 때 스택에 삽입하는 것으로 구현했다.

class AdjMatrix :

- matrix: NumPy로 구현된 인접 행렬. 정점 간 유향 간선이 존재할 때 1의 값을 갖는다.
- addEntity(): 정점 간 유향 간선이 존재함을 표시한다.
- transpose(): matrix에 NumPy의 transpose()를 수행한 결과를 return 한다.

def dfs_matrix() {

기점이 되는 정점과 타 정점의 연결관계를 모두 순차적으로 접근하며 인접 정점을 찾는다.
만약 인접 정점이 방문 되지 않았다면, 해당 정점을 기점으로 DFS를 재귀 호출한다.
스택에 기점 정점을 삽입한다.

}

class AdjList :

- list: 정점의 개수만큼 초기화 되어 있는 리스트로, 각 인덱스는 해당 정점에서 파생하는 간선의 도착 정점들을 연결한 리스트의 head이다.
- addEntity(): 정점의 연결 리스트에 정렬된 위치에 도착 정점을 삽입한다.
- transpose(): 새로운 AdjList 자료구조를 만들고, 각 정점의 연결 리스트를 순회하며 관계를 역전해 새 자료구조에 넣어 return 한다.

def dfs_list() {

기점이 되는 정점의 인접 정점을 순차적으로 접근한다.
만약 인접 정점이 방문 되었다면, loop를 종료하지 않고 다음 인접 정점으로 넘어간다.
만약 인접 정점이 방문 되지 않았다면, 해당 정점을 기점으로 DFS를 재귀 호출한다.
스택에 기점 정점을 삽입한다.

}

class AdjArray :

- pos_list: 정점의 개수만큼 초기화 되어 있는 리스트로, 각 인덱스는 해당 정점에서 파생하는 간선의 개수를 갖는다. 이를 이용해 임의의 정점에 대해 array 중 어느 partition이 해당 정점의 인접 정점을 포함하고 있는지 알 수 있다.
- array: 인접 정점들의 리스트로, 만약 정점 i가 pos_list에서 n, 정점 i+1이 n+k 값을 갖는다면 정점 i+1의 인접 정점들은 array[n+1 ... n+k]에 해당한다.
- addEntity(): array의 알맞은 partition에 도착 정점을 삽입하고 pos_list를 업데이트한다.
- transpose(): 새로운 AdjArray 자료구조를 만들고, array의 각 partition을 순회하며 연결관계를 역전해 새 자료구조에 넣어 return 한다.

def dfs_list() {

기점이 되는 정점의 인접 정점 partition을 찾아 순차적으로 접근한다.
만약 인접 정점이 방문 되지 않았다면, 해당 정점을 기점으로 DFS를 재귀 호출한다.
스택에 기점 정점을 삽입한다.

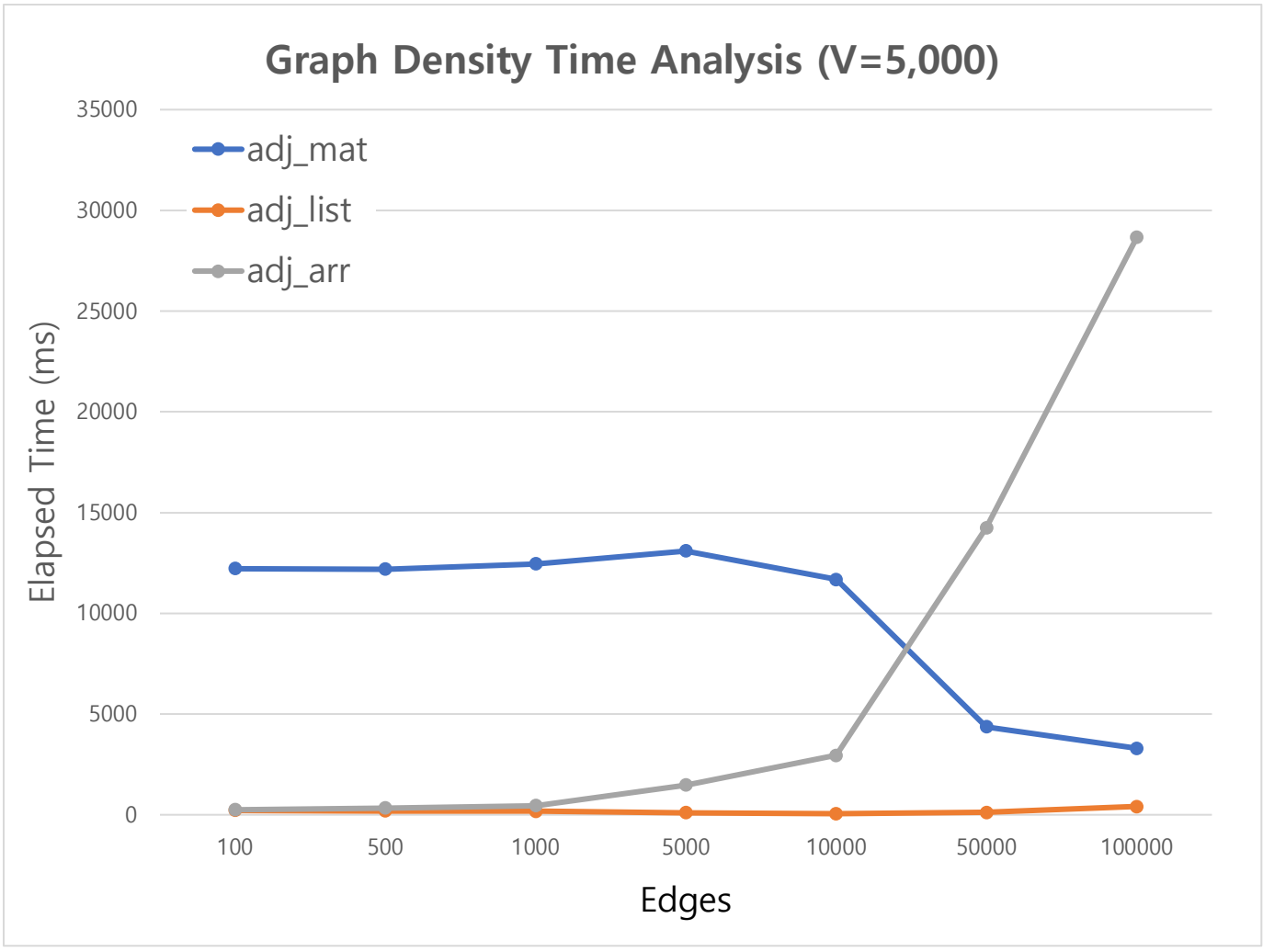
}

3. Experiment

A. Graph 밀도에 따른 수행 시간 분석

정점의 개수를 고정하였을 때 graph의 밀도는 간선의 개수에 비례함을 이용하여 실험을 설계하였다. 정점은 5000개로 고정하였고, 간선의 개수는 100에서 100000까지 변화시키며 소요 시간을 측정하였다. 소요 시간의 측정은 자료 구조를 구성하는 시간을 제외하고, SCC를 수행하는 시간만을 계산하였다. 유의해야 할 점은, 간선의 개수가 50000 이상이 되었을 때부턴 모든 정점이 하나의 SCC에 해당했다는 점이다. 그러므로 실험 결과의 해석은 간선 개수가 [100...10000] 범위일 때에 한정하여 진행했다.

실험 결과 graph의 밀도에 따라 각 자료구조의 효율성이 명확히 대비됨을 확인할 수 있다. 간선 개수가 적어 graph의 밀도가 낮을 때에는 기점 정점과 모든 정점 간 연결 관계를 확인하는 인접 행렬은 비효율적인 반면, 인접 정점들만 확인하는 인접 리스트와 인접 배열은 효율적임이 나타났다. 그러나 graph의 밀도가 높아짐에 따라 인접 행렬은 소요 시간은 상대적으로 큰 변동이 없거나 감소하였지만, 인접 배열의 경우 소요 시간이 증가하였다. 실험 결과 상 인접 리스트가 가장 높은 효율성을 가지며, 인접 행렬과 인접 배열의 경우 graph 밀도가 효율성에 영향이 크다는 것을 발견할 수 있었다.



B. Vertex 개수에 따른 수행 시간 분석

밀도는 고정되었다고 가정할 때 정점 개수의 변화에 따른 소요 시간을 측정하는 실험을 설계하였다. 밀도를 고정시키기 위하여 정점과 간선의 비율이 고정되도록 간선의 개수 역시 정점에 따라 조정하였다. Graph는 간선의 개수 E가 정점의 개수 V와 같다면 한 정점에서 기원하는 간선이 한 개이므로 매우 sparse 하고, E가 V^2일 때 가장 dense하다. 그러므로 실험에서는 두 비율의 중간 지점인 $E = V^2$을 고정 밀도로 설정했다.

실험 결과 정점의 개수가 [100...900] 범위에 있을 때에는 세 자료구조의 성능 차이가 크지 않았지만, 해당 범위 이상으로 정점 개수가 늘어났을 때 인접 배열의 효율성이 크게 떨어졌다. 그 이유에 대해 정확한 답을 찾기는 어려웠지만, Python의 기본 배열은 dynamic array라는 점에서 인접 배열의 크기가 커졌을 때 memory operation 상에서 지역성을 활용하지 못하거나 cache miss가 빈번히 발생하는 등의 문제가 있었을 것으로 추정할 수 있다.