2022-1 Algorithm Homework 3 보고서

서어서문학과 2017-16849 이윤경

1. 실행 환경 및 실행 방식

vs code를 통해 코드를 구현하였고, Python 3.9.6 버전을 이용하였다. 윈도우 환경에서 테스트를 진행하였으며, 터미널에서 hw3.py가 존재하는 디렉토리로 이동 후, command line argument를 이용하여 실행 가능하다. 구체적으로,

adj_mat의 경우 python ./hw3.py input/input.txt input/adj_mat.txt adj_mat,

adj_arr의 경우 python ./hw3.py input/input.txt input/adj_arr.txt adj_arr

adj_list의 경우 python ./hw3.py input/input.txt input/adj_list.txt adj_list

를 통해 실행 가능하다.

input.txt는 hw3.py가 존재하는 디렉토리의 하위 디렉토리인 input에 존재하여야 한다. 결과는 각 명령어에 따라 adj_mat.txt, adj_arr.txt, adj_list.txt에 scc와 실행 시간이 출력된다. 출력되는 실행 시간은 각 자료구조를 만들고 정렬하는 시간을 제외한 DFS, rDFS의 시간만을 측정하였다.

2. 알고리즘 구현

Strongly connected component를 구하는 알고리즘은 다음 순서로 구현하였다.

① 그래프 G에 대해 DFS(G)를 수행해 각 정점의 완료 시간(ftime)을 계산한다;

② G의 모든 간선이 반대로 구성된 새로운 그래프 trans_G를 만든다;

③ 완료 시간이 가장 늦은 정점(ftime의 마지막 원소)부터 시작하여 rDFS(trans_G)을 수행한다;

④ rDFS(trans_G)을 수행하는 과정에서 분리된 부분 그래프들(scc)을 리턴한다

matrix, array, list에 따라 각 클래스를 생성하여 클래스 내에서 scc를 구하는 작업을 하도록 구현하였다. 각 클래스 내부에 getSCC 함수가 있어, 이 함수에 scc를 구하는 알고리즘을 정의하였다. 먼저 모든 vertex를 훑으며 DFS를 수행하고 완료 시간이 빠른 순서대로 ftime에 담는다. transpose된 graph가 없을 경우 transpose를 하며 ftime이 가장 늦은 원소부터 rDFS를 수행하고 rDFS가 끝날 때마다 scclist에 있는 원소들을 result에 string으로 변환하여 추가한다. 이 result를 lexicographic order로 정렬하여 output에 출력한다. 세부적인 알고리즘은 다음과 같다.

(1) graph 만들기 및 transpose

세 가지 명령어 모두 공통으로, 각 클래스를 생성한 후 input.txt를 한 줄씩 읽으면서 vertex와 edge를 구현하였다. readline()을 통해 입력을 읽으며 공백을 기준으로 split하고 map함수를 이용하여 int형으로 바꾸었다. 각 줄에서 첫 번째 숫자는 edge의 개수를 의미하므로 numedge라는 변수에 저장하여, 각 자료구조에 맞게 구현하였다. 세부사항은 다음과 같다.

1) matrix

각 line에서 numedge만큼 루프를 돌며 adj_mat (Adj_matrix 클래스)의 make_matrix 함수를 수행한다. make_matrix 함수는 효율을 위하여 matrix와 이를 뒤집은 trans_matrix를 한꺼번에 생성한다. v^2 크기의 matrix를 모두 0으로 초기화한 후, edge가 있는 곳에 matrix[v][e], trans_matrix[e][v] 의 값을 1로 설정한다. matrix의 경우 v^2만큼의 공간이 필요하므로 다른 자료구조와 다르게 v=1을 index 0으로 사용하여 구현하였다. 이후 scc를 구할 때 값에 +1을 해주어 결과 출력에 문제가 생기지 않도록 하였다.

2) list

list는 linked list를 이용하기 때문에, linked list의 원소인 Node class를 생성하였다. Node는 data와 next의 값을 변수로 가진다. 그리고 각 vertex의 edge들을 Node로 만들어 이를 담을 Node_list class를 Adj_list class 내부에 생성하였다. Node_list는 각 vertex를 head로 하며, 이번 과제에서 linked list의 다른 기능은 필요하지 않기 때문에 linked list의 맨 끝에 원소를 추가하는 append만을 구현하였다. 그리고 이 Node_list들을 담을 전체 entire_list와 trans_list를 Adj_list class 변수로 선언하였다. 초기에 이 list는 각 vertex의 수만큼 Node_list(v)의 값을 원소로 가지고 있는 상태이다. 입력을 한 줄씩 읽으며 각 vertex가 가진 edge를 Node_list에 append한다. trans_list도 vertex와 edge만 바꾸어 초기에 만들어 두었다.

3) array

array의 경우 클래스가 생성될 때 vertex를 담는 vertex_array와 edge를 담는 edge_array를 변수로 생성한다. vertex_array에는 각 vertex가 갖는 edge의 수를 누적합하여 담았다. 입력은 vertex가 1부터 n까지 차례대로 들어오기 때문에 바로 전 vertex의 값을 더해주는 것으로 충분하다. 입력을 한 줄씩 읽으며 num_edge가 0이 아닐 경우 num_edge를 제외한 나머지, 즉 edge들을 edge_array에 extend하여 저장하였다. vertex i의 edge들은 edge_array에 vertex_array[i-1]+1부터 vertex_array[i]까지의 index에 저장되어 있는 원소를 의미한다.

이런 방식으로 array를 구현하면 나머지 두 개의 케이스와 달리 transpose 함수를 통해 그래프를 반대로 만들어야 한다. 그래서 transpose 함수를 구현하였다. 먼저 각 vertex의 수만큼 for loop를 돌며 trans_vertex(기존의 edge) array의 해당 index에 1씩을 추가하며 trans_edge의 개수를 세고, edge와 vertex를 뒤집은 값을 temp_edge 리스트에 추가한다. 이 과정이 끝나면 다시 for loop를 돌며 기존의 make_array와 같은 방식으로 trans_vertex와 trans_edge를 만든다. for loop를 두 번씩 도는 이유는 기존의 입력과 달리 transpose하는 과정에서는 trans_vertex의 크기 순서대로 입력이 들어오지 않기 때문이다.

(2) DFS, rDFS

DFS와 rDFS는 유사한 구조로 구현하였으며, 처음에는 재귀를 사용하여 구현하였으나, 입력의 크기(vertex, edge의 수)가 커질수록 프로그램이 멈추는 현상이 발생하여 stack을 사용하여 iterative하게 구현하는 방식으로 수정하였다. 세 class 모두 전체적인 작동 방식은 동일하며 vertex의 edge를 찾는 과정만 차이가 있다.

DFS는 모든 원소가 False로 초기화된 visited 리스트와 ftime을 계산하기 위해 필요한 index, ftime 리스트, stack을 활용한다. 먼저 DFS를 시작한 원소의 visited 값을 True로 바꾸고, stack에 추가한다. stack에 존재하는 원소가 없을 때까지 while문이 실행되며, stack에서 pop한 node의 edge값이 아직 방문되지 않았으면 stack에 이 값을 추가하여 DFS를 계속한다. 이 때 node에서 더 이상 방문할 값이 없으면 DFS가 완료된 것이므로 ftime에 추가한다. root node부터 ftime에 추가하면 완료된 시간으로 정렬되지 않기 때문에, while문 전에 ftime의 길이를 계산하여 하나의 DFS가 완료될 때마다 index를 update하고, 가장 먼저 완료된 원소를 그 index에 insert하는 방식으로 구현하였다.

rDFS도 DFS의 과정은 stack을 활용하여 동일하지만, scc list를 구하는 것이 목적이기 때문에 ftime을 계산할 것 없이 한 번의 rDFS가 진행될 때마다 scc list를 result에 추가하고 scc list를 초기화하여 scc를 구하였다.

edge를 찾는 과정은, adj_mat의 경우 전체 vertex를 훑으며 matrix[i][j]값이 1일 때 j값을 stack에 추가하였다. adj_list의 경우 stack에서 pop한 node를 head로 갖는 링크드 리스트에서 next값이 null일 때까지 while문을 반복하며 stack에 next의 data값을 차례로 추가하였다. adj_arr의 경우 vertex[i-1]+1과 vertex[i] 사이의 값을 index로 하는 edge_array의 값들을 stack에 추가하였다.

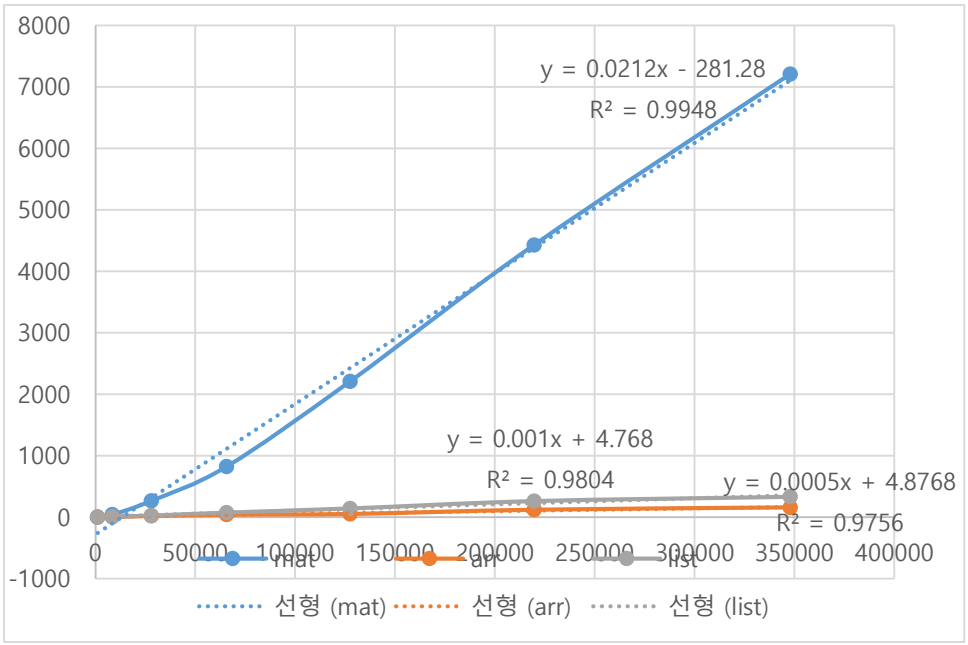
3. 결과 분석

결과 분석은 vertex의 크기에 따른 비교와, vertex의 수가 같더라도 얼마나 sparse하고 dense한지의 여부에 따라 실행 시간이 어떻게 차이가 나는지를 비교하였다. 측정된 시간은 getSCC 함수에서 vertex의 수만큼 루프를 돌며 DFS와 rDFS를 하는 과정만을 측정하여, 실제 실행시간과는 차이가 있다. 특히 list의 경우 e가 커질수록 각 vertex가 갖는 링크드 리스트의 길이가 길어져 graph를 만드는 데 시간이 오래 걸린다는 점을 감안해야 한다. 이는 수업에서 말한 것처럼 dense한 경우에 리스트를 만드는 데 필요한 오버헤드가 커졌기 때문임을 알 수 있다. matrix도 graph를 만드는 데에 v^2만큼의 시간과 공간이 필요하므로 상당한 시간이 소요된다.

(1) V의 크기에 따른 비교

DFS와 rDFS의 시간 복잡도는 $\Theta(V+E)$이다. 이를 증명하기 위하여 V와 E의 개수를 변화시키며 검증을 진행하였다. E가 과하게 sparse/dense하여 오버헤드가 영향을 주는 경우를 배제하기 위하여 $E=v^{1.5}$, 즉 각 v가 $v^{0.5}$개의 edge를 가질 때의 시간을 측정하였다.

v	100	400	900	1600	2500	3600	4900
e	1000	8000	27000	64000	125000	216000	343000
v+e	**1100**	**8400**	**27900**	**65600**	**127500**	**219600**	**347900**
mat	2	40	264	822	2209	4429	7210
arr	1	3	23	43	53	121	158
list	1	7	24	73	141	262	332



이 결과를 토대로 V+E를 x축, 실행 시간(ms)를 y축으로 하여 그래프를 만들어 선형 추세선을 추가하였을 때, R^2의 값이 1에 가까워 모두 $\Theta(V+E)$를 만족함을 확인할 수 있었다. 또한 각 자료구조의 시간을 비교하였을 때, 인접 배열이 가장 빠르고 그 다음이 인접 리스트, 인접 행렬이 가장 느림을 확인할 수 있었다. 이는 인접 행렬의 경우 V^2 크기의 matrix를 만드는 시간을 제외하여도, 각각의 vertex가 edge를 가지는지를 v의 크기만큼 loop를 돌며 비교해야 하기 때문에 시간이 오래 걸리는 것으로 판단하였다. 인접 리스트는 vertex가 갖고 있는 edge만을 node로 가지고 있기 때문에 상대적으로 모든 node를 훑는 데 드는 시간이 적게 든다. 인접 배열은 이 보다 더 효율적인 방식으로, 배열의 index만으로도 edge를 알아낼 수 있기 때문에 리스트보다 접근이 좀 더 빠르다.

(2) E의 개수와 밀도에 따른 비교

이 경우는 V의 개수를 1000으로 고정하고, E의 개수를 1000부터 500,000까지 증가시키며 시간을 비교하여 보였다. 다양한 case를 위해 V의 개수를 500으로 정하고, E의 개수를 5000에서 250,000(완전그래프)까지 증가시키며 비교한 결과도 첨부한다.

1) V = 1000

E	1000	5000	10000	50000	100000	200000	500000
mat	247	296	345	382	326	414	451
arr	7	7	9	16	44	71	196
list	5	9	16	47	94	181	632

2) V = 500

E	5000	10000	50000	100000	250000
mat	61	60	73	118	123
arr	3	2	21	44	113
list	17	17	48	125	262

두 결과에서 확인할 수 있듯이 E가 점점 커져 밀도가 높아질수록 matrix의 실행 시간보다 list의 실행 시간이 커짐을 알 수 있다. 이는 간선이 많은 경우 리스트에서 차례대로 원소를 훑어야 하기 때문에 오버헤드가 증가하는 것이다. 반면 matrix는 간선의 밀도가 낮을수록 실행시간이 다른 자료구조에 비해 오래 걸리고, 밀도가 높아질수록 오버헤드가 낮아짐을 알 수 있다.

4. example running

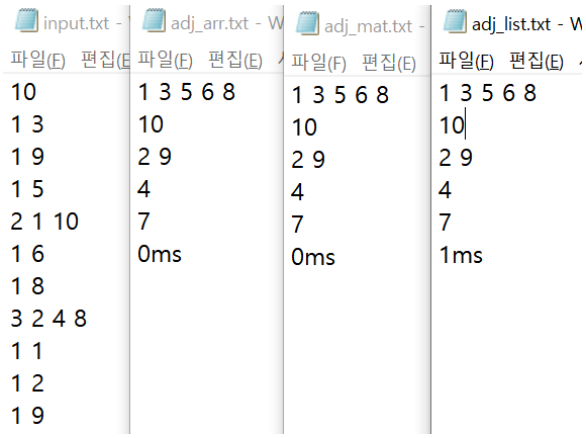
각 testcase는 testcase.py로 v와 e의 개수에 따라 케이스를 만들어 실험을 진행하였다.

```
import random
input = open("input/input.txt", "w")

n = 100
v = 2
input.write(str(n)+ "₩n")

for i in range(n):
    input.write(str(v)+" ")
    check=[ ]
    for j in range(v):
        num = random.randint(1, n)
        while num in check:
            num = random.randint(1, n)
        check.append(num)
        input.write(str(num)+ " ")
    input.write("₩n")
```

첨부한 코드와 같이 v의 범위에서 random으로 정수를 생성하여 각 vertex의 edge로 추가하는 과정이다.

```
> python testcase.py
> python ./hw3.py input/input.txt input/adj_mat.txt adj_mat
> python ./hw3.py input/input.txt input/adj_list.txt adj_list
> python ./hw3.py input/input.txt input/adj_arr.txt adj_arr
```

다음의 코드를 활용하여 example running을 진행하였고 결과는 다음과 같다.



input.txt
```
10
1 3
1 9
1 5
2 1 10
1 6
1 8
3 2 4 8
1 1
1 2
1 9
```

adj_arr.txt
```
1 3 5 6 8
10
2 9
4
7
0ms
```

adj_mat.txt
```
1 3 5 6 8
10
2 9
4
7
0ms
```

adj_list.txt
```
1 3 5 6 8
10
2 9
4
7
1ms
```

시각적으로 표현하기 위해 v의 크기를 줄여서 첨부하였다. 입력이 좀 더 큰 경우는 제출

파일에 첨부할 것이다.