자료구조 프로젝트 Report

2012120366 김민정

1. System environment & interface

1.1 System environment

파이썬 2.7.11을 윈도우 환경에서 사용하였습니다.

1.2 System interface

```
0. Read data files
1. display statistics
2. Top 5 most tweeted words
3. Top 5 most tweeted users
4. Find users who tweeted a word
5. Find all people who are friends of the above users
6. Delete all mentions of a word
7. Delete all users who mentioned a word
8. Find strongly connected components
9. Find shortest path from a given user
99. Quit
Select Menu:
```

```
Select Menu: 50
Not available number

Select Menu:
```

Init.py 코드를 실행을 하면 다음과 같은 메뉴화면이 뜨고 user는 메뉴를 선택할 수 있습니다. 99번을 선택할 때 까지 시스템은 계속 실행이 됩니다. 또한 유효하지 않은 메뉴번호를 선택할 시, 위의 그림에서와 같이 'Not available number' 가 뜨면서 새로 메뉴를 선택할 수 있습니다.

사용자가 메뉴를 선택하면 init.py 는 선택된 메뉴에 해당하는 함수를 menu.py 에서 호출합니다. 그리고 중복되어 쓰이는 function이나 길이가 좀 길어지는 function들 같은 경우에는 functions.py 라는 파일을 또 만들어서 menu.py에서 호출 하도록 하였습니다.

2. What data structure you chose and why?

파일을 읽고 바로 dictionary 형태의 hash table로 자료를 저장하였습니다. 그 이유는 time complexity 가 작았기 때문입니다. 구체적으로 Dictionary를 만드는 데는 O(n), 그를 이용한 search, insert, delete 를 구현하는 데 있어서는 O(1)를 달성할 수 있었습니다.

```
# Make data structure: Hashtable
# read txt file
u = readtxt('user.txt')
f = readtxt('friend.txt')
w = readtxt('word.txt')

# Construct Hashtable(Dictionary)(key=user, value=friend)
user_friendship = make_dic(f, 3, 1)
# Construct Hashtable(Dictionary)(key=user, value=word)
tweet_dic= make_dic(w, 4, 2)
# Construct word_user_dict(key=word, value=user)
word_user_dic = make_word_user_dic(w, 4, 2)
```

또한, 데이터들이 저장된 dictionary 들을 global variable 로 설정함으로 써

```python
# Construct Hashtable(Dictionary)(key=user, value=friend or word)
def make_dic(data, num_line, space):
    dic = defaultdict(list)
    # append values
    for i in range(0, len(data), num_line):
        key = data[i].split('\n')[0]
        value = data[i+space].split('\n')[0]
        dic[key].append(value)
    return dic

# Construct word_user_dict(key=word, value=user)
def make_word_user_dic(data, num_line, space):
    dic = defaultdict(list)
    # append values
    for i in range(0, len(data), num_line):
        key = data[i+space].split('\n')[0]
        value = data[i].split('\n')[0]
        dic[key].append(value)
    return dic

def make_user_list(data, num_line):
    all_users = []
    for i in range(0, len(data), num_line):
        user = data[i].split('\n')[0]
        all_users.append(user)
    return all_users
```

메뉴를 통해 user나 단어를 삭제하면 전체 dataset에도 그대로 적용되고 다른 메뉴를 호출할 때도 적용된 data에서 실행되게 하였습니다.

8번과 9번의 graph algorithms에 대하여는, dictionary의 형태로 저장된 data를 연결리스트 형태로 바꿔줌으로써 8번과 9번 각각에 대하여 DFS를 통한 strongly connected components를 찾는 task, dijsktra algorithm을 이용해 shortest path를 찾는 task를 수행하였습니다.

3. What is your expected performance?

기본적으로 전체적으로 hash table인 dictionary를 사용했기 때문에 바로 위의 그림에서 보다시피 만드는 것은 O(n), 그리고 search, insert, delete를 수행하는 데 있어 O(1)이 걸렸습니다. 하지만 주어진 task의 특성에 따라 부가적으로 수행해야 할 일이 생겨 task마다의 time complexity는 달랐습니다. 또한 graph algorithm task를 수행하는데 있어서는 linked list 형태로 바꾸는데 있어 더 많은 time complexity를 희생했습니다.

3.1 Display statistics

```python
def stat(dic):
    num_list = []
    for key in dic:
        size = len(dic[key])
        num_list.append(size)
    avg = np.mean(num_list)
    minimum = min(num_list)
    maximum = max(num_list)
    return avg, minimum, maximum
```

평균, max, min을 계산하기 위해서 key 값에 해당하는 value size에 대해 평균과 maximum, mininum을 수행했기 때문에 총 O(N)이 걸렸습니다.

3.2 Find the Top 5 most tweeted words/users

```
def top5(dic):
    key_list = []
    for key in dic.keys():
        key_list.append(key)
    # Extract the number of values and sort and get the indices
    count_list = extract_num_value(dic)
    index_list = range(len(count_list))
    index_list.sort(key=lambda n:count_list[n])
    index_list.reverse()
    # Find the top 5 words
    top5 = extract_top5(key_list, index_list)
    print top5
```

```
def extract_num_value(dic):
    value_list = []
    for key in dic.keys():
        v = len(dic[key])
        value_list.append(v)
    return value_list

def extract_top5(key_list, index_list):
    top5_list = []
    for i in range(5):
        j = int(index_list[i])
        top5_list.append(key_list[j])
    return top5_list
```

Extract_num_value 함수에서 각 key에 해당하는 value size (the number of users and words for each)을 계산하고, sorting을 하여 가장 큰 수 5 개의 해당하는 index의 key 값을 반환하는 구조로 진행하였습니다. 총 time complexity 는 O(N) 입니다.

3.3 Fine users / all people who are friends of the user who tweete a word

```
def find_users(dic):
    # Get the word to be searched
    word = raw_input("type the word tweeted:")
    # Find the users who tweeted word
    user_list = dic[word]
    print user_list
```

```
def find_friend_user(word_user_dic, user_friendship):
    # Get the word to be searched
    word = raw_input("type the word tweeted:")
    # Get the list of users who tweeted word
    user_list = word_user_dic[word]
    # Make the list of users' friend
    friend_list = []
    for i in range(len(user_list)):
        friends = []
        friends = user_friendship.get(user_list[i])
        friend_list += friends
    print friend_list
```

해당 단어를 tweet한 user를 찾는 task에 대해서는 target word를 사용한 user list를 직접 받아 O(1)이 걸렸으며, 그 유저들의 친구 목록을 search하는 데 있어서는 user들의 친구 리스르틀 다시 받아 새로 list를 만들었습니다. 위 과정에서는 O(|V|)가, (이 때는 |V|는 해당 단어를 tweet 한 유저의 수), 들었습니다.

3.4 Delete all mentions of a word / all users who mentioned a word

```
def delete_meantions(tweet_dic, word_user_dic):
    # Get the word to be searched
    word = raw_input("type the word tweeted:")
    # Delete the word from tweet_dic
    user_list = word_user_dic[word]
    for user in user_list:
        try:
            tweet_dic[user].remove(word)
        except ValueError:
            pass
    # Delete the word from word_user_dic
    try:
        del word_user_dic[word]
    except KeyError:
        pass
    return tweet_dic, word_user_dic
```

```
def delete_user(user_friendship, tweet_dic, word_user_dic):
    # Get the word to be searched
    word = raw_input("type the tweeted word:")
    user_list = word_user_dic[word]
    for user in user_list:
        try:
            del user_friendship[user]
            del tweet_dic[user]
        except KeyError:
            pass
        try:
            word_user_dic[word].remove(user)
        except ValueError:
            pass
    return user_friendship, tweet_dic, word_user_dic
```

첫번째로 입력된 단어를 지우는 task에서는 단어를 우선 받고, 그 단어가 들어 있는 dictionary, 즉 user_friendship dictionary를 제외한 모든 dictionary에서 해당 단어를 지웠습니다. 이 과정에서 user가 key이고 단어가 value 인 경우 모든 user에 대해 지워야 하기 때문에 O(|V|), (이 때 |V|는 user의 수), 그리고 단어가 key 인 경우에는 O(1)이 걸렸습니다.

두번째로, 입력된 단어를 tweet 한 user를 지우는 task 에서는 user가 들어있는 모든 dictionary에서 user를 삭제하였습니다. 각각의 user를 지우는 데에는 O(1), 총 해당 하는 단어를 언급하는 user를 지우는 데에는 O(|V|), (이 때 |V|는 user의 수)가 소요 되었습니다.

3.5 Find scc/shortest path

```
def find_scc(user_friendship):
    make_print_scc(user_friendship)
    return 0

def find_shortest_path(user_friendship):
    userID = raw_input("type the user number:")
    shortest_path(user_friendship, userID)
    return 0
```

위의 task에 대해서는 각각 DFS와 Dijkstra 알고리즘을 사용하여 수행했습니다. 그 과정에서 dictionary를 연결 리스트, adjacency list로 바꿔야 했고, 1) 모든 user가 Vertex class를 가져야 했고, 2)친구 관계(edge)를 이어주는 과정에서도 친구 역시 dictionary value가 아닌

```
def make_print_scc(dic):
    vertices = []
    for key in dic.keys():
        vertex = DFSVertex(key)
        vertices.append(vertex)

    DFS = DepthFirstSearch()
    DFS.set_vertices(vertices)

    for vertex in vertices:
        friends = dic[vertex.name]
        for friend in friends:
            for node in vertices:
                if friend == node.name:
                    vertex.add(node)
    DFS.scc()

def shortest_path(dic, userID):
    g = Dijkstra()
    for key in dic.keys():
        user = g.add_vertex(key)
        if user.name == userID:
            user.d = 0
    for vertex in g.vertices:
        friends = dic[vertex.name]
        w = len(dic[vertex.name])
        for friend in friends:
            for node in g.vertices:
                if friend == node.name:
                    vertex.add(node, w)
    g.shortest_path()
```

vertex class를 가져야 한다는 점, 3) 그리고 해당 key의 dictionary value (맞는 친구관계)만 add 해야 한다는 점에서 많은 time complexity 가 소요 되었습니다.

만약 위의 menu를 통해 삭제 될 user를 고려하지 않는 다면 데이터를 바로 읽고 연결리스트를 만듬으로 써 O(N)이 걸리지만, 위의 menu들을 고려한 빠른 hash table 형태의 자료구조를 연결리스트로 만들다 보니 time complexity가 높아진 단점 (O(N^3))이 발생하였습니다.

하지만 다행히 하나의 for loop은 해당하는 vertex와 같은 값을 가지는 key의 value만을 대상으로 for loop이 돌기 때문에 최악의 상황보다는 적은 computation cost가 소요 될것이라 예상 됩니다.

4. How would you improve the system in the future?

시스템에서 가장 큰 단점이었던 Hash table에서 연결리스트로 변환하는 과정에서 improvement 할 여지가 있다고 생각이 됩니다. Hash table를 이용함으로써 search, insert 그리고 delete에 있어서는 다른 어떤 data structure 보다 빠른 성능을 보였지만, 그래프 알고리즘의 측면에 있어서는 취약한 모습을 보였습니다. 따라서 애초부터 vertex class 를 가진 노드를 hash table 형태로 저장하는 등의 방법을 통해서 효과적으로 성능을 높일 수 있을 것 같습니다.