Twitter Network Analysis

Team 2

21300461 우상흠 21400183 김조운 21700097 김민주
21700428 Sergio Eduardo Alfaro Lopez 21700643 전영원

INDEX

1. Processing Text File
2. Problem_1
3. Problem_2
4. Problem_3
5. Problem_4
6. Problem_5

1. Processing Text File

- **Problem analysis**: We were really concerned about what kind of data structure we should use to manage this huge amount of data. Because we were convinced that this process was the biggest factor in reducing complexity in solving future problems. However, while we were working on the algorithms of problems, we found that one data structure alone could not solve these problems efficiently.

Therefore, we concluded that it is right to apply the optimal data structure for each problem without applying the same data structure to all problems in this process.

- **Data Processing process**:

Adjacency List → Problem 1(It helped find the follow of each user.) and 5
Adjacency Matrix → Problem 2, 3, and 4
Array → Problem 1(finding follow, follower)

2. Problem_1

- **Problem analysis** : The problem was to get the number of people the user follows and the followers who follow the user. At first, we had to approach this problem too loosely because we thought that every user had follow at least one user. However, through trial and error, we realized that the idea was wrong and there was a user who did not follow anyone.
- **Used data structure**: Array and Adjacency linked list
- **Variables Used** :

* vertex and edge is the number of users and the number of follows, respectively.
int fwcount[vertex] = { 0 }; // store each user's follow number
int fecount[vertex] = { 0 }; // store each user's follower number

- **Solution :**

Follow - When creating an adjacency list, first add the node for that user, and then add that user's follow to the next node. In this process, add 1 to the array called "fwcount".
Follower - An array "user_list" containing 5128 unique user IDs was created and used for matrix creation. And in index corresponding to user_list, the number of followings and followers of the user is recorded in "fwcount" and "fecount" respectively.

- **Result** :

```
id:201499968, #n of follow:5, #n of followee:5
id:201882918, #n of follow:1, #n of followee:0
id:202037047, #n of follow:2, #n of followee:1
id:202186639, #n of follow:2, #n of followee:2
id:202500496, #n of follow:1, #n of followee:3
id:202881242, #n of follow:4, #n of followee:0
id:203226736, #n of follow:7, #n of followee:7
id:203770783, #n of follow:2, #n of followee:0
id:203783262, #n of follow:2, #n of followee:3
id:203942956, #n of follow:10, #n of followee:2
id:204317520, #n of follow:144, #n of followee:152
id:204422008, #n of follow:3, #n of followee:0
id:204424857, #n of follow:1, #n of followee:1
```

3. Problem_2

- **Problem analysis** : Find the maximum distance user X to user Y, and who is that.
- **Process of finding solution** : Since problem_2 is must know the distance of each other. So make the *distance_adjacency matrix*, that is store the distance. First, user X and user Y is direct connected, that pair's distance is "1". Then, X and Y is undirect connected, that pair's distance is sum of "X to Z", "Z to Y".
- **Used algorithms** : Floyd–Warshall algorithm
- **Variables Used** :

int distance_adjmatrix[vertex][vertex] = { 0 }; // store the distance of each pair

- **Solution :**

Step1. When make the adjacency, check the directed connection of each pair. And if follow of followed give the distance "1"

Step2. Use the "Floyd-Warshall algorithm" update the distance_adjmatrix[vertex][vertex].

- This algorithm has three loops. The first loop is passing point. Called "Z". And the second loop is start point. Called "X". The last loop is destination point. Called "Y".
- In the second loop, we check the "X == Z" and "distance of X to Z". If "X != Z" and "distance of X to Z >= 1", go to next loop. If not go to next "X".
- In the third loop, we check the "X == Y", "distance of Z to Y", and "distance of X to Y". If "X != Y" and "distance of Z to Y >= 1", and "distance of X to Y == 0", update the distance_adjmatrix[X][Y] = distance_adjmatrix[X][Z] + distance_adjmatrix[Z][Y]. Else if "X != Y" and "distance of Z to Y >= 1", and "distance of X to Y > 1", update the distance_adjmatrix[X][Y]'s minimum distance. The other is leave its original state

Step3. Use the nested loop, accesses all elements in distance_adjmatrix[vertex][vertex]. And then find the maximum number.

- Define the variable int max, x, y.
- Accesses all elements in distance_adjmatrix[vertex][vertex], and compare the max.
- If max < distance_adjmatrix[X][Y] element, update the max = distance_adjmatrix[X][Y], x = X, y = Y.

Step4. Print the max, user_list[x], user_list[y].

- **Result** :

```
[sol2] Maximum distance: 25, with 17461932 and 131364612
```

4. Problem_3

- **Problem analysis** : For each user id, it will be possible to consider every single one of them as a vertex. Moreover, the edge between vertex is connected when one of two follow another. Therefore, the question, for whether all vertexes are connected by an edge, can be raised.

- **Process of finding solution** : Since problem_3 is not affected by the direction of follow; therefore, the undirected_adjacency matrix is decided to be used. First, find the group which contains two user id that has a following relation. Then, expand the following relationship by looking for undirect_adjmatrix. But it had a problem because we have to consider 5128 vertexes, but in this process, checking which vertex already is visited is very hard. It was important to find an algorithm that makes it easy to check visit. And we decided to use the connected component algorithm.

- **Used Algorithms** : Connected Component, DFS, graph, undirected_adjacency matrix

- **Variables Used** : int DFSvisit[vertex] = { 0 } // visit =1 , not visit =0,
int cntCompoenet =0 // the number of partition = the number of start vertex

4. Problem_3

- **Solution :**

Step1. When DFS function is performed, all vertex that can be reached from the start vertex becomes one component.

- The DFS function's parameter is the index of the start vertex and looks for all vertex that can be reached from there.
- Therefore, when entering the DFS function, make DFSvisit[index] =1 corresponding to the index in the start vertex.
- Next, in the repeat sentence 'for(int i=0); i<vertex; i++)', if this two condition is satisfied, DFSvisit[i]=0 (has not yet been visited again), and undirected_adjmatrix[start][i] == 1(there is a relation of follow), calling the DFS function again like recursive function(parameter = i).
- Because all vertex are reviewed within the for() statement and find DFS again in connected vertex, all vertex that is connected to first start vertex can be founded.
- In addition, all vertex starting points in the DFS function is recorded as visiting DFSvisti[]= 1.

Step2. Then select a vertex that is not visited, and all vertex that can be reached from that vertex becomes another component.

- In this case, for example, since all vertex connected with the 0th starting vertex is in the visit=1 state through step1(due to the DFS function), the vertex not related to 0^{th} vertex becomes a another start vertex.

Step3. This process can be repeated until all 5128 vertex visits have been completed. Also, all the connection components present on the graph can be found.

- If you provide cntComponent++ when the start point vertex occurs before calling the DFS function, the cntComponent is the number of partitions.

- **Result** :

```
[sol3] every does not be connect with another one
because number of partition : 244
```

5. Problem_4

- **Problem analysis**: If each user id is considered a vertex, the edge between vertex is connected when one of two follow another. Therefore, the question is whether all vertexes are connected by an edge.
- **Process of finding a solution**: Since problem_4 is affected by the direction of follow, the adjacency matrix is decided to be used. This problem is pretty similar to problem_3. But the difference relies on an only direction of follow, because Problem 3 is not affected by the direction. Therefore, most solution for problem_4 is similar to problem_3 solution. But the matrix which will be used for problem 4 has to different.
- **Used algorithms**: Connected Component, DFS, graph, adjacency matrix.
- **Variables Used** :

Int mutualmatirx[vertex][vertex] = { 0 }; // mutually follow = 1, not =0
int DFSvisit[vertex] = { 0 } // visit =1 , not =0
int cntCompoenet =0 // the number of partition = the number of start vertex

- **Solution :**

Step1. Make a mutualmatrix by using adjmatrix. There are two cases that make a mutualmatrix[][] =1. First, if the index of x and y of mutualmatrix[][] are the same. Second, if adjmatrix[a][b] ==1 and adjmatrix[b][a] ==1.
(In Step2, only small thing is different from the solution for problem3, so this shows differently only on the part for the specific process)
Step2. When DFS function is performed, all vertex that can be reached from the start vertex becomes one component.

- in the repeat sentence 'for(int i=0); i<vertex; i++)', if this two condition is satisfied, DFSvisit[i]=0 and multualmatrix[start][i] == 1, calling the DFS function again like recursive function.

(Step 3 & Step 4 is exactly the same as the solution for problem3. Therefore, delete specific process)
Step3. Then select a vertex that is not visited, and all vertex that can be reached from that vertex becomes another component.
Step4. This process can be repeated until all 5128 vertex visits have been completed. Also, all the connection components present on the graph can be found.

- **Result** :

```
[sol4] number of friend partition : 2646
```

6. Problem_5

- **Problem analysis**: The problem is finding the 20 most influential users through “Random Surfer”. The random surfer starts randomly at a specific user node. With a 90% probability, surfer moves to one of specific user that the user follow. And when the probability is 10% or when the current node is not following anyone, surfer moves to a random node.
- **Process of finding a solution**: Since problem_4 is affected by the direction of follow, the adjacency matrix is decided to be used. This problem is pretty similar to problem_3. But the difference relies on an only direction of follow, because Problem 3 is not affected by the direction. Therefore, most solution for problem_4 is similar to problem_3 solution. But the matrix which will be used for problem 4 has to different.
- **Used method :** Threshold, Black box testing
- **Result** :

```
[sol5] Pagerank with threshold 20000:
rank 1 - user813286 percentage: 0.014714
rank 2 - user7861312 percentage: 0.014235
rank 3 - user18713254 percentage: 0.007734
rank 4 - user59804598 percentage: 0.004073
rank 5 - user22027186 percentage: 0.003948
rank 6 - user14413075 percentage: 0.003889
rank 7 - user16129920 percentage: 0.003815
rank 8 - user21158690 percentage: 0.003263
rank 9 - user73381700 percentage: 0.003116
rank 10 - user20291791 percentage: 0.003051
rank 11 - user26281970 percentage: 0.002948
rank 12 - user20240002 percentage: 0.002914
rank 13 - user31572616 percentage: 0.002810
rank 14 - user31067157 percentage: 0.002768
rank 15 - user111164068 percentage: 0.002689
rank 16 - user17092592 percentage: 0.002664
rank 17 - user13058232 percentage: 0.002541
rank 18 - user113420831 percentage: 0.002531
rank 19 - user14269220 percentage: 0.002500
rank 20 - user8163442 percentage: 0.002406
```

6. Problem_5

- **Variables Used:**

List *adjlist[vertex] = { 0 }; // In each list, the user id node appears first, and then the user's follow id appears.
int usercount[vertex] = { 0 }; // When random surfer visit each user, count++ is performed.
int userflag[vertex] = { 0 }; // To prevent duplicate users from entering the top 20
int rank[20] = { 0 }; // 20 most influential user id is recorded.

- **Solution:**

Step1. We finished implementing the behavior of random surfer according to a given problem.
Step2. Applied the threshold method as a way to find 20 influential users.

- Method for applying the Threshold are as follows. If the "rank_index" variable is less than 20, the random surfer continues to move.
- Whenever Random Surfer moves to a specific user node, the user count is increased through the "usercount" array. After this process, check if the count of the user is greater than the threshold and the value of the user in the "userflag" array is 0. If this condition is correct, record this user id in the rank array and "rank_index" increases by 1.

Step3. But the problem was to find the most optimal threshold.

- We tried to use binary search algorithm to compare the 20 user IDs of the previous and current in the 1 to 50000 range and select that threshold as the optimal threshold if all 20 are the same. Thinking about many other methods, we concluded that there is no more efficient method than the black box testing technique.
- In order to intuitively view the changes using the black box technique, a ratio value was generated by dividing the total number that surfer moved to each user.
- We observed a change in the ratio of fixed 10 user ids while executing a certain threshold five times. At first, the change was so large that we increased the threshold by 5000 in each subsequent attempt. And we gradually decreased the threshold increase whenever the change amount decreased. Finally, we found that the threshold of 25000 is the most optimal threshold.

- **Total Time taken to solve the problem** : Window - 4min 20secs, Mac OS - 3min 30secs