Hongsup OH
GTID:903232384

1.Intro

I use Breath First Search (BFS) method to calculate distance from one node to the node where heist may occur. And then, I calculate the heist-closeness centrality of u, based on the distance d(u,v) calculated by BFS. The equation is:

$$C_{HC}(H,u) = \frac{1}{\Sigma_{u\in H} d(u,v)}$$

Where, H is heist nodes set

2.Breth First Search (BFS)

BFS uses queue instead of stack which is used for DFS. And it is frequently used for graph, to get path or distance. The procedure for BFS is following steps; first, it searches the edges with depth 1, and second it searches the edges with depth 2, and then depth 3. If all nodes are visited or searched, then it will be terminated.

3.BFS Pseudo Code

In this part, I will show how to calculate the shortest path distance based on BFS. First I will explain BFS code and then shortest path code.

3.1.BFS Pseudo Code (python code)

This code will show the path from start point to goal point.

```
def bfs_paths (graph, start, goal):
    queue = [(start, [start])]
    while queue:
            (vertex, path) = queue.pop(0)
              for n in graph[vertex] – set(path):
                  if n == goal:
                     yield path + [n]
                  else:
                     queue.append((n,path+[n]))
```

where, graph = G (V, E), start is initial node, goal is destination.

3.2.Shortest Path Pseudo Code (python code)

```
def shortest_path (graph, start, goal):
    if start == goal:
       return 0
    try:
       path = next (bfs_paths(graph, start, goal)
       return len (path) – 1
    except StopIteration:
       return None
```

This code use bfs_paths(graph, start, goal) to compute the shortest path. Above code define that if start equals goal node, then distance becomes zero. This is because the problem says distance between same node equal zero; d(u, u) = 0.

Hongsup OH
GTID:903232384

3.3.Expected time and space complexity

The time complexity:

$$O(|V| + |E|)$$

Where, $|V|$ is the number of vertices and $|E|$ is the number of edges

Time complex is sum of the number of vertex and edges, when every vertex and edge could be explored in the worst case.

The space complexity:

$$O(|V|)$$

When the number of vertices in the graph is known ahead of time, and additional data structures are used to determine which vertices have already been added to the queue.

3.Heist-Closeness Centrality pseudo code

```
def compute (g, h):
    #part1 code
    Graph_i = {}
    for v in range (g.n):
        a = str (v)
        vert_edg = set ([str(v) for v in g.vertex_edges(v)])
        Graph_i[a] = vert_edg
    #part2 code
    Result = [ ]
    for i in range (g.n):
        dis = lambda x: shortest_path(Graph_i,str(i),str(x))
        lists = list(map(dis,h))
        total = sum(lists)
        inv = 1./total
        g.vdata[i] = inv
    return g.vdata
```

where, g is graph and h is heist node data

Part1 Code is re-organize graph file. An I name it as Graph_i. The form of Graph_i is:

$$\text{Graph\_i} = \{v_i, \{e_{i1}, e_{i2,} \dots, e_{ik}\}\}$$

Where first term is i'th node's vertex and second term is i'th node's edge, which is exactly node data linked with i'th node. I expect time and space complexity is $O(|V|)$

Part2 Code is code for calculating the heist-closeness centrality of u. In this part, code compute distance from each node to the place where heist may happen. And then calculate closeness centrality. This code depends on the number of vertices, the number of edges and the number

of heist node. So I expect time and space complexity is $O([|V| + |E|] \times |H|)$. Where V is the number of vertices, E is the number of edges and H is the number of heist node. Thus, I expect total code's time complexity is:

$$O([|V| + |E|] \times |H|) + O(|V|) \approx O([|V| + |E|] \times |H|)$$

In the problem, the number of vertices is n, the number of edges is m, and the number of heist node is k, thus time complexity is:

$$O((n + m) * k)$$

But I think it is the worst case for BFS, which is expected to visit or search all nodes. Thus, I want to think about following situation.

$$O(n * k)$$

The reason why I think about this situation is that the number of node (n) and the number of heist node (k) strongly affect my code.

4.Expected time complexity and real running time

Four different size data are given; simple, rmat-100-200, rmat-1000-10000, rmat-2000-100000. Then size of data and expected time complexity is:

	simple	rmat-100-200	rmat-1000-10000	rmat-2000-100000
n	11	100	1000	2000
m	13	1000	10022	99989
k	5	20	50	20
$O((n + m) * k)$	$O(120)$	$O(22000)$	$O(220440)$	$O(2039780)$
$O(n * k)$	$O(55)$	$O(2000)$	$O(50000)$	$O(4000)$

<Chart1>

Based on above chart, for the worst case, the running time is simple < rmat-100-200 < rmat-1000-10000< rmat-2000-100000. But for the case ignoring edge's effect, the running time is simple < rmat-100-200 < rmat-2000-100000 < rmat-1000-10000. For the first case, rmat-2000-100000's running time is the longest. But for the second case, rmat-1000-10000's running time is the longest.

Now, I show my results for real running time.

	simple	rmat-100-200	rmat-1000-10000	rmat-2000-100000
Time (sec)	0.000559	0.135936	347.347	185.936

<Chart2>

Hongsup OH
GTID:903232384

The result is same with $O(n * k)$ situation. I think Data does not have the worst situation, such as visiting all nodes. Most of case, the distance between u and v is 1 or 2, which means all nodes never need to be searched. Where u is belonging to node data, and v is belonging to heist node data.

5.Conclusion for the time complexity

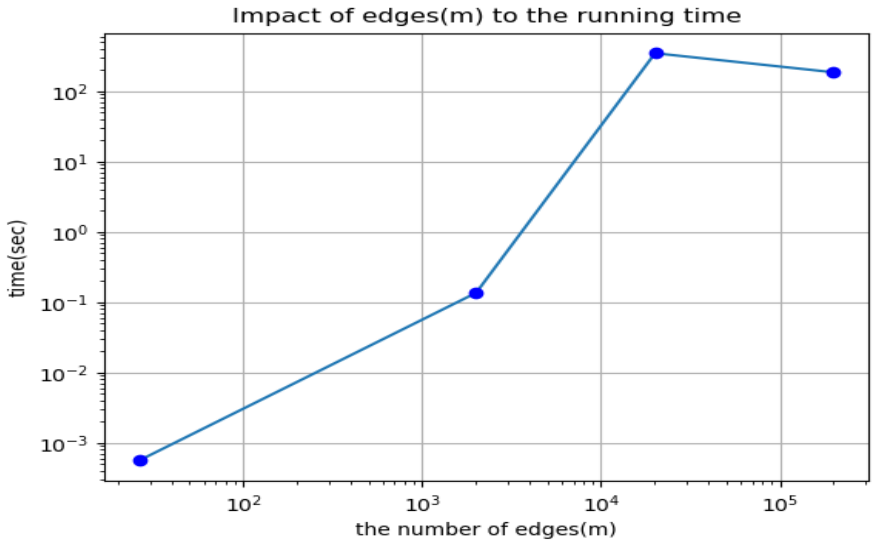
5.1 Complexity based on theory

Based on the theory, the number of vertices (n), the number of edge (m), and the number of heist node (k) are affect the time complexity. This is because BFS's time complexity is O(n+m) and closeness centrality can be calculated by between the shortest distance measured by BFS and the number of heist O(k). Thus, time complexity can be expected as O(k(n+m)).

5.2 Complexity of this problem

However, the number of edge (m) does not affect greatly in my situation (I never mean the number of edge (m) is not important). Instead, the number of heist node (k) strongly affect the running time. This is because I only need to considering edges' data between v_n and v_k, where v_n is belong to graph's vertices and v_k is belong to heist data. For example, shortest_path (graph, start, goal) is my code to determine the shortest path between start and goal points. In the function, start point consider all nodes in the graph data (total n nodes), but goal point only considers the points belonging to heist data. Therefore, the edges data between each nodes and heist point is important. Thus, running time is similar with O(k*n).

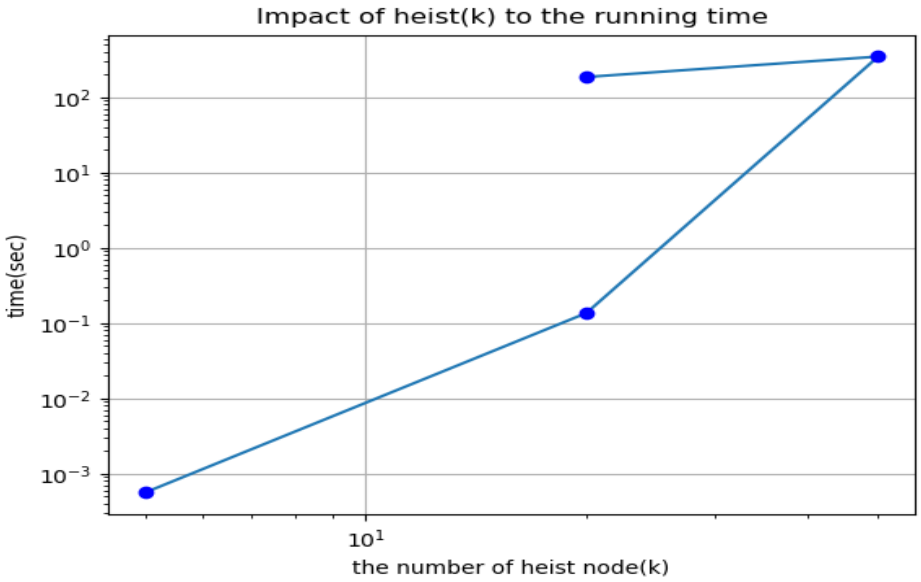
Then I will show three types of graph. The first is between the number of edges and the running time, second is between the number of heist nodes and the running time, and last is between the number of nodes in graph and the running time. Node (n), edge (k), and heist node (k) are from <Chart1> and time(sec) is from <Chart2>.



<Figure1. Edges(m) vs time(sec)>

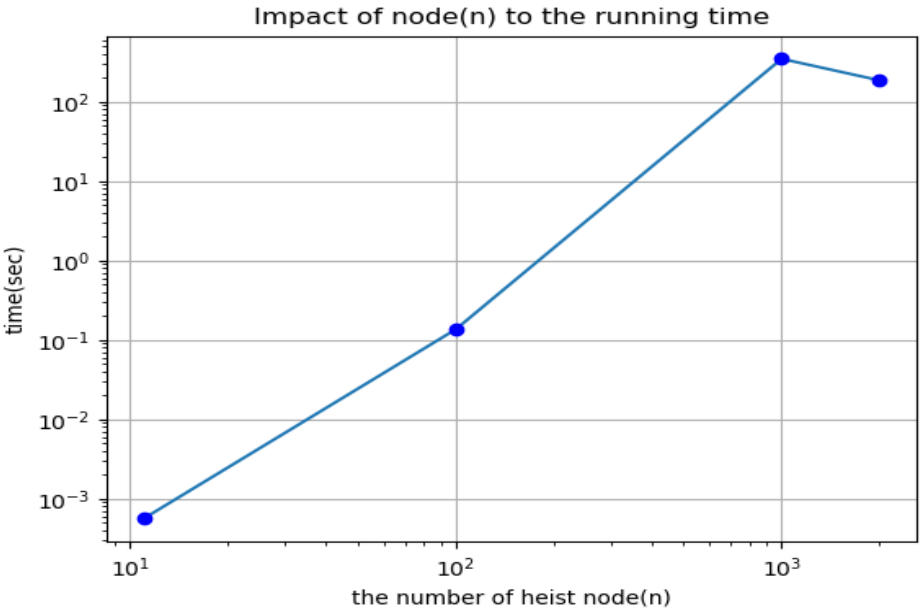
Based on Figure1, the running time increases as the number of edges increases. But the running time of rmat-2000-100000 is shorter than rmat-1000-10000. I think this is influenced by the number of heist node (k).

Hongsup OH
GTID:903232384



<Figure2. Heist(k) vs time(sec)>

Based on Figure2, the running time increase as the number of heist node(k) increases. Thus, rmat-1000-10000's running time (k=50) is longer than rmat-2000-100000 (k=20). Here, running time for rmat-100-200 is shorter than that of rmat-2000-100000, although their k is equal (k=20). This is because the number of edge(m) and node(n) for rmat-2000-100000 is much larger than that of rmat-100-200.



<Figure3. Nodes(n) vs time(sec)>

Figure3's trend is almost same with that of Figure1.

Thus, I think the number of heist node (k)'s impact for the running time is the strongest. And the number of node (n)'s impact is stronger than that of the number of edges (m).

Hongsup OH
GTID:903232384

6.Result

I must find the vertex with the highest heist closeness centrality, to help Batgirl. I find the maximum values by following code:

```
ans = compute(g,v)
Cc_index = [ ]
for i in range (len (ans)):
    if ans[i] == max(ans):
        Cc_index.append(i)
```

	simple	rmat-100-200	rmat-1000-10000	rmat-2000-100000
Best vertex	2, 7	0, 3	0	0, 250

For the simple case, v_2, v_7 are the best position, for rmat-100-200 case, v_0, v_3 are the best position, for rmat-1000-10000 v_0 is the best position, and for rmat-2000-100000 case, v_0, v_{250} are the best position.