Written by Tianwei Mo

Q1

1.1
d = 9/15 = 0.6

1.2
Edge_array = [2, 3, 4, 1, 3, 1, 2, 4, 5, 6, 1, 3, 5, 3, 4, 6, 3, 5]
Vertex_array = [0, 3, 5, 10, 13, 16, 18]

1.3
CSR is the best data structure in this case. Because CSR is best for saving storage space and processing neighbor scanning, which is needed in PageRank. It is worst for updating, while we do not have this concern with dealing a static graph. Adjacency matrix is good to use to represent a small dynamic graph, because it is easy to update. Adjacency list is good to use when the graph is large and dynamic since its performance is moderate to do any operation.

Q2

2.1
a. Wrong. A possible DFS traversal could be a-d-c-b-e-f.
b. Wrong. a is a two hop neighbor of c, while e is a one hop neighbor of c. BFS traversal must visit any two hop neighbor after visiting all one hop neighbor.
c. Correct. A possible DFS traversal could be d-a-b-c-e-f.
d. Correct. e is a two hop neighbor of b, while d is a one hop neighbor of c. BFS traversal must visit any two hop neighbor after visiting all one hop neighbor.

2.2
The initialization requires O(n) time.
In the while loop, to pop and maintain Q needs O(log(n)) time with a min heap, while it only need O(1) to do with a Fibonacci heap. So the time complexity is O(n)
The worst case to update d[u] is to update all the edges in the graph, which cost O(log(n)) time to do once with a min heap and O(1) with a Fibonacci heap, and do O(m) times in all. So the time complexity is O(m).
In all, the time complexity is O(m).

Q3

3.1

	A	B	C	D	E	F	G	H
A	1	1	1	1	1	1	1	1
B	0	1	0	1	0	1	1	1
C	0	0	1	1	1	1	1	1
D	0	0	0	1	0	1	1	1
E	0	0	0	0	1	0	1	1
F	0	0	0	0	0	1	1	1
G	0	0	0	0	0	0	1	1
H	0	0	0	0	0	0	0	1

3.2
Node index: B: 1, E: 2, H: 3, G: 4, F: 5, D: 6, C: 7, A: 8
Construct interval: B: [1, 1], E: [2, 2], H: [3, 3], G: [4, 4], F: [3, 5], D: [3, 6], C: [2, 7], A: [1, 8]
Compress: (C, F): no effect.
There are 8 intervals in the final compression scheme.

Q4

4.1
Yes. Node v1, v2, v3, v4 and v5 compose a 4-core.

4.2
Yes. Node v1, v2, v3, v4 and v5 compose a 5-truss.

4.3
Core(v1) = 3. v1 have at least 3 neighbor with core >= 3.

Q5

5.1
PR(D) = (1-0.85)/7 + 0.85*(1/7/2+1/7/2) = 1/7

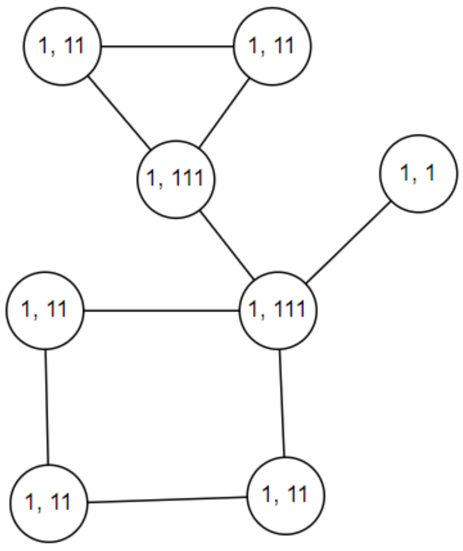
5.2

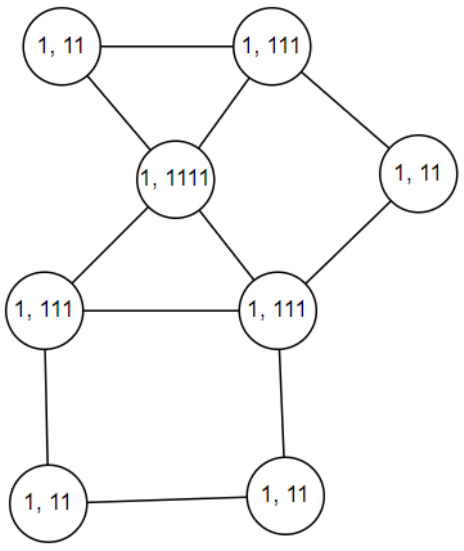
If the combiner is used, X would send 2 messages to Y, one to node C and one to node D. Y would send 2 messages to Z, one to E and one to F. In all, 4 messages are generated. If the combiner is not used, X would send 3 messages to Y, since 3 arrows points from X to Y. Y would send 3 messages to Z, since 3 arrows points from Y to Z. in all, 6 messages are generated.

Q6

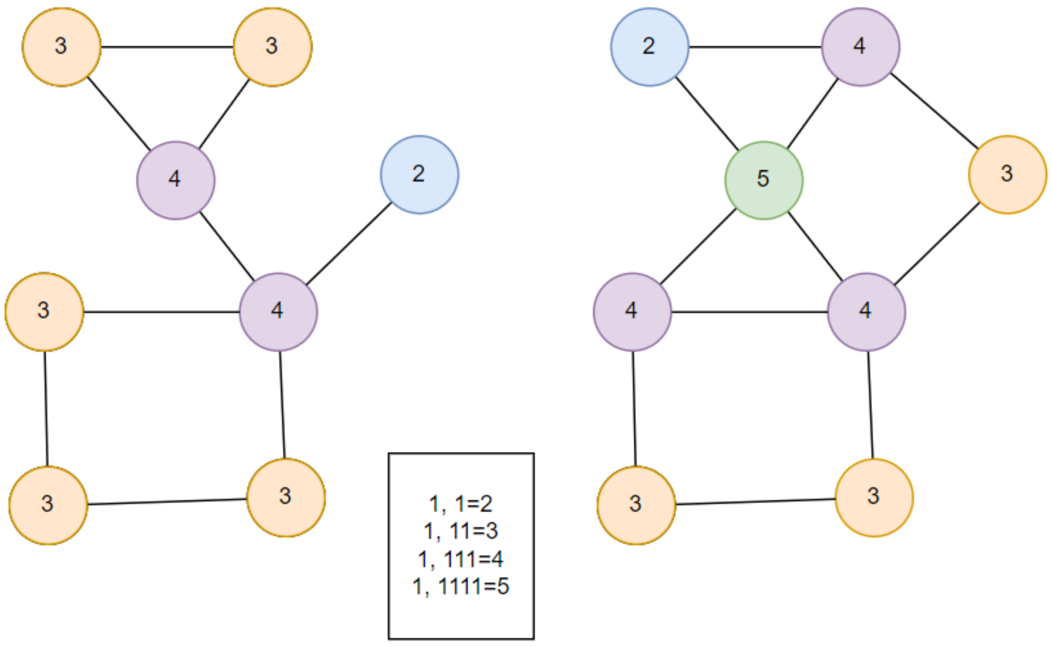
6.1
Betweenness of B = {(2+0+2+0+1+2+1)/(7*6) = 8/42 = 4/21 if consider duplicate shortest paths, (1+0+0+0+0+1+0)/(7*6) = 2/42 = 1/21 if consider unique shortest paths}
Closeness centrality of B = 1/(1+2+1+2+1+1+2) = 1/10
Betweenness of C = {(3+0+1+3+0+2+3)/(7*6) = 12/42 = 1/3 if consider duplicate shortest paths, (2+0+0+2+0+1+2)/(7*6) = 7/42 = 1/6 if consider unique shortest paths}
Closeness centrality of C = 1/(1+2+2+1+2+1+1) = 1/10

6.2
Aggregate colors

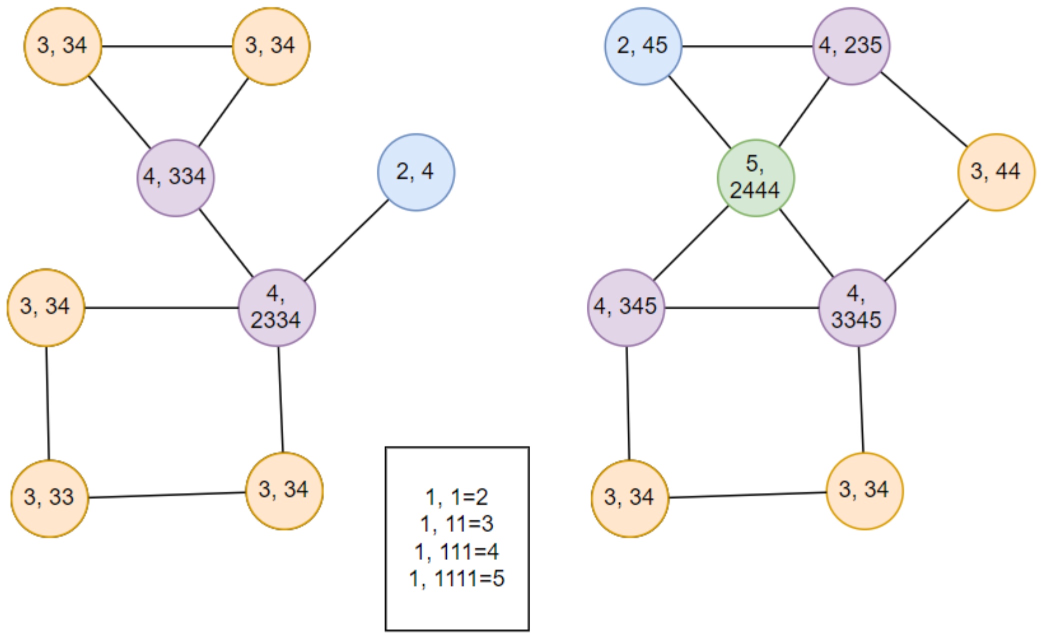




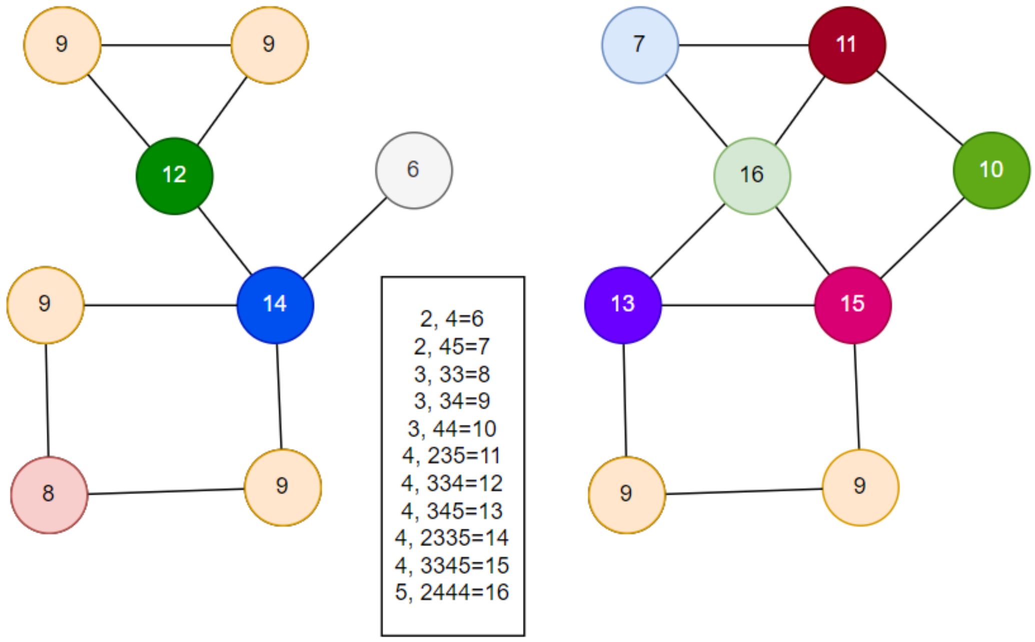
Hash colors



Aggregate colors



Hash colors



2, 4=6
2, 45=7
3, 33=8
3, 34=9
3, 44=10
4, 235=11
4, 334=12
4, 345=13
4, 2335=14
4, 3345=15
5, 2444=16

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
Phi(G4) = [8, 1, 5, 2, 0, 1, 0, 1, 4, 0, 0, 1, 0, 1, 0, 0]
Phi(G5) = [8, 1, 3, 3, 1, 0, 1, 0, 2, 1, 1, 0, 1, 0, 1, 1]
K = 94

Q7

7.3
GraphSAGE perform better.

Q8

8.1
C

8.2
a. True
b. True
c. True
d. True

8.3

a. True
b. True
c. True
d. True

8.4
We multiply H^l and B_l^t because we want to consider not only neighbors' information, but also the information of itself. We multiply D^-1 and A because this is a normalization. In this way, node with different number of neighbors can have similar scale.

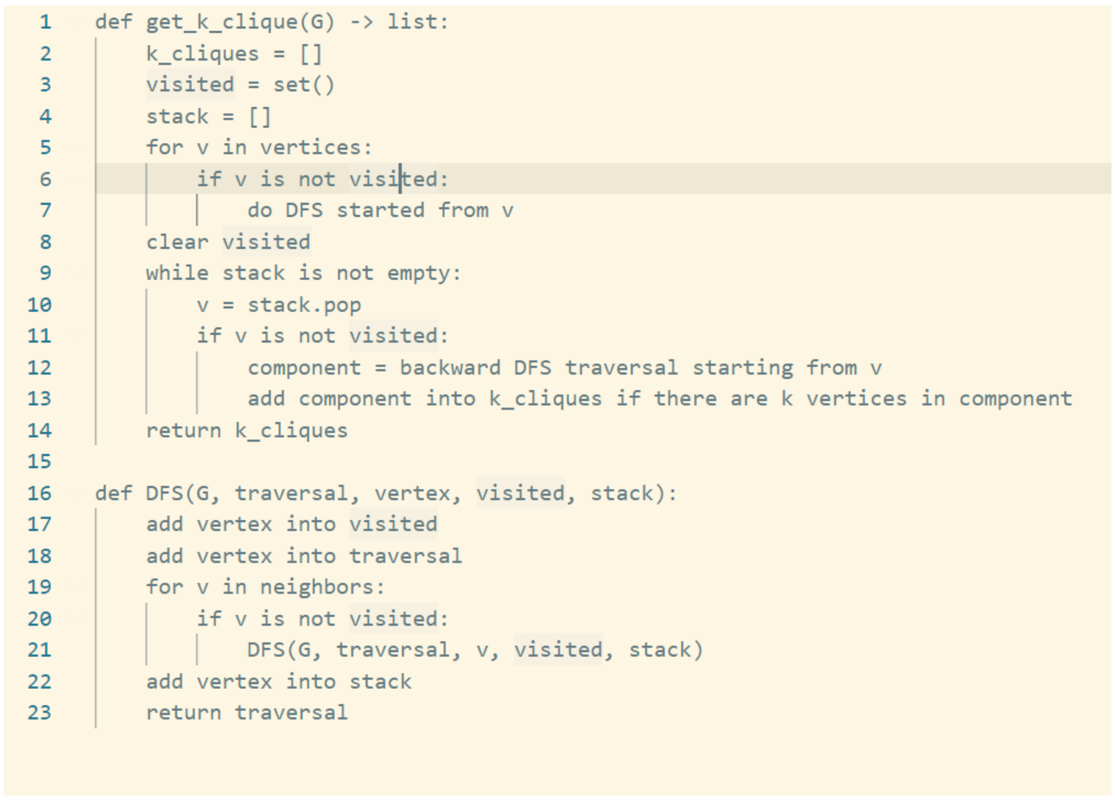
Q9

```
H^1 =   [[0.5000, 0.9000, 1.2000, 0.4000],
         [0.1333, 0.2667, 0.8000, 0.0333],
         [0.8667, 0.2667, 0.6667, 0.1667],
         [0.4000, 0.6500, 0.6000, 0.2500],
         [0.4333, 0.0333, 0.7333, 0.0000],
         [0.0000, 0.1000, 0.8000, 0.0000],
         [0.0000, 0.4400, 0.5600, 0.0000],
         [0.2500, 0.6500, 0.3000, 0.1000],
         [0.0000, 0.1000, 0.7000, 0.0000]]
```

Q10

10.1
We can do DFS. At some point, if we get the source node, that means there is a cycle.

10.2
Pseudo-code:



```
def get_k_clique(G) -> list:
    k_cliques = []
    visited = set()
    stack = []
    for v in vertices:
        if v is not visited:
            do DFS started from v
    clear visited
    while stack is not empty:
        v = stack.pop
        if v is not visited:
            component = backward DFS traversal starting from v
            add component into k_cliques if there are k vertices in component
    return k_cliques

def DFS(G, traversal, vertex, visited, stack):
    add vertex into visited
    add vertex into traversal
    for v in neighbors:
        if v is not visited:
            DFS(G, traversal, v, visited, stack)
    add vertex into stack
    return traversal
```

Python code:

```
def get_k_clique(G, k) ->list:
    SCCs = []
    visited = set()
    stack = []
    vertices = list(G.vertex_dict.keys())
    for v in vertices:
        if v not in visited:
            DFS(G, [], v, visited, stack)
    print(stack)
    reverse = reverse_G(G)
    visited = set()
    while stack:
        v = stack.pop()
        if v not in visited:
            component = DFS(reverse, [], v, visited, [])
            if len(component >= k):
                SCCs.append(component)
    return SCCs

def DFS(G, traversal, vertex, visited, stack):
    visited.add(vertex)
    traversal.append(vertex)
    neighbors = G.adj_list_out[G.vertex_dict[vertex]]
    for out_edge in neighbors:
        v = out_edge[0]
        if v not in visited:
            DFS(G, traversal, v, visited, stack)
    stack.append(vertex)
    return traversal

def reverse_G(G):
    r_G = copy.deepcopy(G)
    temp = copy.deepcopy(r_G.adj_list_in)
    r_G.adj_list_in = r_G.adj_list_out
    r_G.adj_list_out = temp
    return r_G
```

Time complexity:
Doing DFS to all unvisited node cost O(m) time. Doing backward DFS for all nodes in the stack cost O(mn) time. In all, the time complexity is O(mn).