CS599 Graph Analytics: Assignment 1
Saurav vara prasad Chennuri.

Question 1

Consider some frequent graph operations (add edge, remove the edge, add a node with neighbors, remove the node with neighbors, check if an edge exists, compute degrees, etc.), and discuss the pros and cons of the graph representations we saw in class.

Answer

Graph Operations: Add edge, remove the edge, add a node with neighbors, remove the node with neighbors, check if an edge exists, compute degrees.

Graph Representations: Adjacency Matrix, Adjacency List, Compressed Sparse row representation, edge LIst

Adjacency Matrix

Space Complexity : n*n

1) Add Edge: O(1)
    - Should traverse to that index of row and column of nodes being connected
2) Remove Edge: O(1)
    - Should traverse to that index of row and column of nodes being disconnected
3) Add Node with neighbors: O(n)
    - Should put the values to '1' at all the rows and columns of the node in focus with neighbors
4) Remove Node with neighbors: O(n)?
    - Should equate the values to zero in the row and column of the node
5) Compute Degree: O(n)
    - Should sum the values in the row/column of the node in focus

Pros: Easy to find an edge between any node and any node.

Cons: Requires a lot of space, inefficient when the graph is sparse

Adjacency List

Space Complexity: O(no. of vertices + no. of edges)

1) Add Edge: O(1) or O(n) #Depends on where you are adding the node on the linked list(first or last)
    - Should traverse to the end of the linked list for that particular node if adding at the end. In this case, time complexity would be O(n)
    - We could also insert the node at the beginning after the first node. In this case, the time complexity would be O(1)
2) Remove Edge: O(n)
    - Should traverse all the neighbors of the nodes being removed and remove the node from those linked lists, and then delete the linked lists from these nodes.
3) Add Node with neighbors: O(n^2)

        - Should add the node at the end of the linked list for all its neighbors and then we also need to start a linked list from this node.

4) Remove Node with neighbors: O(n^2)?
    - Should go to all the neighbors of the node, then find the node by traversing through the end of the linked list of those nodes and then delete that node.
5) Compute Degree: O(n)
    - Should go through to the end of the linked list starting from that node.

Pros: Efficient in terms of storage of data.

- New nodes can be added easily
- New nodes can be connected with existing nodes easily based on how you are attaching the node in the neighbor's linked list
-

Cons: But when looking at edges, deleting a node, etc, the time complexity is high. We will have to traverse till the end of the linked list from that node

- To tell if an edge exists between any two nodes, it takes lot of time

<u>Compressed Sparse Row representation</u>
Space Complexity: O(sum of degrees of all nodes + number of nodes)

1) Add Edge: O(n)
    - Since the values are hashed, we can directly go to the position of the nodes indexes, and add new number value after that position
    - then update the array indexes for the rest of the nodes in F arrays index (we generally dont need to do this as programming languages usually track the indexes automatically)
    - After that, we also would need to update the index numbers in the offset array

2) Remove Edge: O(n)
    - Should go through the end of the edge array, deleting new node number for the nodes which are being disconnected, then update the array indexes for all the node values
    - After that, we also would need to update the index numbers in the offset array

3) Add Node with neighbors: O(n)
    - Should go through the end of the edge array, then insert the neighbor node values of the node being added
    - Then traverse through the end of the edge array again, adding the new node to each of the node's neighbors
    - Then traverse through the end of the edge array, updating the indexes. We should also go through the end of the index numbers of the offset array by traversing through the end of it.

4) Remove Node with neighbors: O(n)
    - Should delete all the nodes from the start index value in the edge array for the node being deleted. We can access these values via the offset array as it stores indexes where we stored the neighbors of the node in the edge array
    - Then traverse through the end of the edge array updating the indexes of the edge array (this may not be required based on the implementation in python as we don't have to index any array)

        - Then traverse through the end of the offset array updating the index values of each node

5) Compute Degree: O(1)
    - We can look at the offset array, and get to know the node's degree by subtracting the value at the index[node+1] and index[node]
    - degree(n) = offset[n+1] - offset[n]

Pros: Really fast to compute degrees of nodes. This is specifically designed for the purpose of finding degrees

Cons: Adding a new node, and removing a new node would be a hassle as we will have to traverse through all the subparts of the edge array where we have the neighbors of the node being added and then add the current node as a neighbor for all the other nodes.

<u>**Edge List**</u>
Space Complexity: O(number of edges + number of nodes)

E - Number of edges
n - Number of nodes

1) Add Edge: O(1)
    - Add an edge at the end of the edge list
2) Remove Edge: O(E)
    - We can hash those edges, so we can find it really fast. Then remove that edge
3) Add Node with neighbors: O(degree(node))
    - Should add all the edges of that node at the end of the edge list
4) Remove Node with neighbors: O(E)
    - Should traverse through all the edges in the list to find the edges with the node to be deleted, and delete them
5) Compute Degree: O(E)
    - Should traverse through all the edges in the edge list to find the number of edges that this node. If the edges are repeated (A- B, B- A), then half the count would be the degree.

Pros: Easy to add edges

Cons: Takes really long to delete an edge or delete a node. Also, takes lot of memory to store all the edges.
- Hard to tell if an edge exists between A and B
- Hard to find the degree of a node. We will have to traverse through all the edges in the edgelist

Proof of triangle packing to be NP complete Second Explanation

Given problem: graph with vertices |V| = 3n, we need to prove that the problem is NP-complete

To prove a problem is NP complete

1) We need to show it is NP
2) We need to show that another NP algorithm converges towards our problem. There should be a certificate in our current problem that is polynomial in length that can be verified in polynomial time

   The certificate can be described this way as a set of 'k' triples { (v1a, v1b, v1c), (v2a, v2b, v2c), ……., (vka, vkb, vkc) }. We can check in polynomial time if each of these vertex sets form a triangle.

   Step2

   Now we have to prove that there is another NP problem whose input converges towards our problem of 'k' node disjoint triangles. An example would be 3DM (3-dimensional matchin) problem which was proved to be an NP problem back in 1972 bu Karp et.al in his paper "reducibility among combinatorial problems"

   3DM Problem Description

   Say, we have three sets of elements X,Y, W with equal cardinalities. |X| = |Y| = |Z| = q

   And X = {x1, x2, x3, … , xq}
   Y = {y1, y2, y3, … , yq}
   Z = {z1, z2, z3, … , zq}

   Let T be a set of all Triples that we can generate using the three sets X,Y,Z

   Let M be a subset of all the Triples that we can generat using X, Y, Z. M is called a 3-dimensional matching if there exists (x1,y1,z1) in M and (x2, y2, x2) in M where x1≠x2, y1≠y2, z1≠z2. Basically all are different coordinates.

   The 3d problem can be defined in one way as given this set of triples T, and an integer 'k', we have to decide weather there exists a 3-d matching M that is a subset of T, where the number of such triples in M ≥ 'k' (|M| ≥ k).

Now we redefine our problem as a graph ‘G’ which is a union of vertices (X x Y x Z), and the cardinalities of all these three sets are equal. |X| = |Y| = |Z|. And for all the edges we find{xy, yz, zx }, we can define a triple (x,y,z)

The solution then becomes as if we are trying to find k 3-d matches, with every triple (x,y,z) in M would be a solution. This way, we can see that there exists another another problem which is NP solvable that can be reduced to our problem making our problem NP-complete.