Iterative Recolor

Mahdihusain Momin, Shantwana Dixit

February 16, 2018

RecolorInsert

nodes is the set of all nodes.
v_c is the set of nodes whose core value may change.
k is the core number of nodes in v_v.
(a)Color of all nodes in v_c is true and rest of the nodes is false. The algorithm used to find the v_c is simple breadth first search.

```
void reColorInsert(nodes, k, v_c){

        Initialize map orig_mcd which stores the current mcd values of all nodes in v_c;
        Initialize set roots with nodes in v_c whose mcd values is equal to k;

        for each node w in roots do
                Initialize a queue Q;
                Q.enqueue(w);

                while Q is not empty do
                        u = Q.dequeue();

                        if (nodes[u].mcd==k and nodes[u].color==true) then

                                for each node x in neighbours of node u do
                                                if((nodes[x].color=true) and
                                                    (nodes[x].k == k) and
                                                    (nodes[x].mcd > k)){
                                                        nodes[x].mcd -= 1;
                                                        Q.enqueue(x);
                                                }

                                nodes[u].color = false;
                                nodes[u].mcd = orig_mcd[u];

}
```

The nodes in v_c whose color is true will get their core value updated.

RecolorDelete

nodes is the set of all nodes.
v_c is the set of nodes whose core value may change.
k is the core number of nodes in v_v.
node1, node2 are nodes of the edge being deleted.

```
void reColorDelete(nodes, node1, node2, k, v_c){

        if v_c is empty then return;

        Initialize set roots with node1 if mcd value of node1 is k-1;
        Add node2 to roots if mcd value of node2 is k-1;

        Initialize map orig_mcd which stores the current mcd values of all nodes in v_c;

        for each node w in roots do
                Initialize queue Q;
                Q.enqueue(w);

                while Q is not empty do
                        u = Q.dequeue();

                        if (nodes[u].mcd == k-1 && nodes[u].color=true) do
                                for each node x in neigbours of node u do

                                                if((nodes[x].color=true) &&
                                                   (nodes[x].k == k) &&
                                                   (nodes[x].mcd > k-1)){
                                                        nodes[x].mcd -= 1;
                                                        Q.enqueue(x);
                                                }

                                }
                                nodes[u].color = false;
                                nodes[u].mcd = orig_mcd[u];
                        }
                }
        }
}
```

The nodes in v_c whose color is false will get their core value updated.

Proof of correctness of Algorithm

Theorem: Under the case of insertion of an edge (u, v), the core number of a node needs to be updated if and only if its color is 1 after Algorithm terminates

First, we prove that if the core number of a node w needs to be updated, then its color is 1 after Algorithm terminates. We focus on the case of $C_u = C_v = c$, similar proof can be used to prove the other two cases. By our assumption, we have $w \in V_c$, where $V_c = V_{u \cup v}$ .After inserting an edge (u, v), the core number of the nodes in V_c increases by at most 1. Therefore, if C_w needs to be updated, then the updated core number of w must be c + 1. That is to say, node w must have c+1 neighbors whose core numbers are larger than or equal to c + 1. Now assume that the color of node w is 0. This means that $mcd = c$ when Algorithm terminates. This result implies that node w has at most c neighbors whose core numbers are larger than c, which is a contradiction.

Second, we prove that if a node has a color 1 after Algorithm terminates, then the core number of this node must be updated. Let V_1 be a set of nodes with color 1 after Algorithm terminates, and $V_{>c}$ be a set of nodes whose core numbers are greater than c. Denote by $G^* = (V_1 \cup V_{>c}, E^*)$ an subgraph induced by the nodes $V_1 \cup V_{>c}$. Consider a node w in such an induced subgraph G^*. Clearly, if w has a color 1 (i.e., $w \in V_1$), then it has mcd ($mcd > c$) neighbors in G^* . If w has a color 0 (i.e., $w \in V_{>c}$), then its core number C_w is larger than c. The induced subgraph G^* belongs to the (c+1)-core. Therefore, the core number of a node w with color 1 is at least c + 1. After inserting an edge (u, v), the core number of any nodes in graph G increases by at most 1. Consequently, the core number of the nodes with color 1 increases by 1.

Theorem: Under the case of deletion of an edge (u, v), a node in V_c whose core number needs to update if and only if its color is 0 after Algorithm terminates.

First, we prove that if a node w in V_c whose core number needs to be updated, then its color is 0 after Algorithm terminates. By our assumption, after deleting an edge (u, v), C_w decreases by 1. This means that C_w decreases to c - 1. That is to say, w has c - 1 neighbors whose core numbers are no less than c - 1. Suppose that the color of w is 1 after the algorithm terminates. This implies that $mcd >= c$. Recall that mcd denotes the sum of the number of w's neighbors whose core numbers are larger than c and the number of w's neighbors whose color is 1. Note that a node with color 1 suggests that its core number equals to c. As a result, w has at least c neighbors whose core numbers are larger than or equal to c, which is a contradiction.

Second, we prove that if a node w in V_c is recolored by 0 after Algorithm terminates, then C_w must be updated. Let $V_{>c}$ be a set of nodes whose core numbers are larger than c. Then, after deleting an edge (u, v), we construct an induced subgraph by the nodes in $V_c \cup V_{>c}$, which is denoted as $G^* = (V_c \cup V_{>c}, E^*)$. Note that the core numbers of the nodes in V $\backslash V_c \cup V_{>c}$ are smaller than c. Therefore, they do not affect the core numbers of the nodes in $V_c \cup V_{>c}$. If a node $w \in V_c$ with a color 0 after Algorithm terminates, then $mcd < c$. This suggests that the node w in G^* has at most c - 1 neighbors. G^* at most belongs to the (c 1)-core. The core number of any nodes in G decreases by at most 1 after deleting an edge. Therefore, the core numbers of the nodes in V_c with color 0 decrease by 1. This completes the proof.

Dynamic MCD Updation

k is the core value of nodes set V_c.
pseudoMCD is the mcd calculated by the algorithm.
OriginalMCD is the mcd before running the algorithm.
RED denotes nodes in V_c whose core value donot changes.
GREEN denotes nodes in V_c whose core value changes.
BLACK denotes nodes that are not in V_c.

NODE	INSERTION	DELETION
RED	OriginalMCD	pseudoMCD
GREEN	pseudoMCD	OriginalMCD + no. of neighbour with core value k-1
BLACK	+1 if k-value is k+1	No change

Table 1: MCD value after insertion and deletion

Running Time Analysis

RED denotes nodes in V_c whose core value donot changes.
GREEN denotes nodes in V_c whose core value changes.
BLACK denotes nodes that are not in V_c.

NOTE: This runtime analysis is without considering efficient datastructures. But the BigOh runtime will still remain same after considering the efficient datastructures.

Insertion

Invariants:
(i) Each RED node's neighbors are only explored once over all the bfs.
(ii) We explore neighbors of a node only if it is RED node.
In worst case, every node can be a RED node. So, total no. of RED nodes possible is $|V_c|$.

So, runtime of Iterative RecolorInsert in worst case is $d_1 + d_2 + ... + d_{|V_c|} = \sum_{u \in V_c} d_u$.

In worst case, time taken to update mcd is $d_1 + d_2 + ... + d_{|V_c|} = \sum_{u \in V_c} d_u$ which will occur when all nodes are GREEN.

So, total time taken will be $O(\sum_{u \in V_c} d_u)$.

Deletion

Invariants:
(i) Each GREEN node's neighbors are only explored once over all the bfs.
(ii) We explore neighbors of a node only if it is GREEN node.
In worst case, every node can be a GREEN node. So, total no. of GREEN nodes possible is $|V_c|$.

So, runtime of Iterative RecolorDelete in worst case is $d_1 + d_2 + ... + d_{|V_c|} = \sum_{u \in V_c} d_u$.

In worst case, time taken to update mcd is $d_1 + d_2 + ... + d_{|V_c|} = \sum_{u \in V_c} d_u$ which will occur when all nodes are GREEN.

So, total time taken will be $O(\sum_{u \in V_c} d_u)$.

2014 Paper Running Time

Both insertion and deletion has $O(|V_c| * \sum_{u \in V_c} d_u)$ running time.
We have clearly a theoritical runtime gain of $|V_c|$ factor.

References

[1] Efficient Core Maintenance in Large Dynamic Graphs[2014]
`Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao`