IMPLEMENTATION OF K-TRUSS DECOMPOSITION ALGORITHM

Name: Xingyu Yan

Director: A. Erdem Sariyuce

University at Buffalo The State University of New York

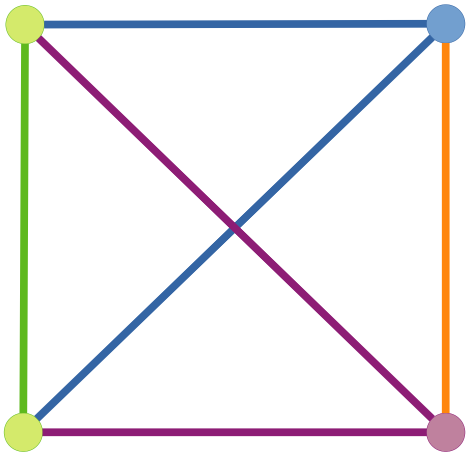
Contents

- K-truss Decomposition
- My Implementation
- Improvements by Researchers
- Next Step and Potential Improvement

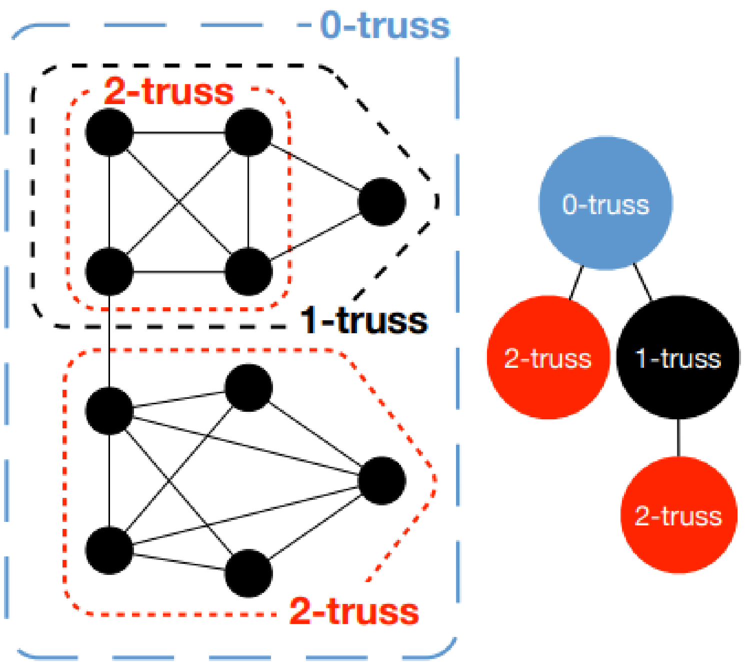
University at Buffalo The State University of New York

K-TRUSS DECOMPOSITION

Truss value for an edge



K-truss Decomposition



[Cohen, Jonathan]

University at Buffalo The State University of New York

MY CURRENT IMPLEMENTATION

Implementation:

Based on the work of Cohen

- Early & fundamental
- Peeling algorithm
- No optimization

Pseudo Code

```
removeUnsupportedEdges( G, j )
{
    //    this will hold edges to remove
    q ← ∅;

    //    count triangles C supporting each edge
    //    if count too small, queue edge for removal
    ∀e = (a,b)∈E do
        {
            put members of N(a) in a hash table T;
            c ← 0;
            ∀v∈N(b), if v∈T then c ← c+1;
            if c < j, then {q ← q∪e; remove e from G; }
                else C(e) ← c;
        };

    //    remove queued edges, perhaps queuing neighboring ones as well
    while q ≠ ∅ do
        {
            pull e = (a,b) from q;
            put members of N(a) in a hash table T;
            I ← ∅;
            ∀v∈N(b), if v∈T then I ← I∪{v};
            ∀e′ joining a or b to an element of I, do
                {
                    decrement C(e′);
                    if C(e′) < j, then {q ← q∪e′; remove e′ from G; }
                };
        };
```

[Cohen, Jonathan. "Trusses: Cohesive subgraphs for social network analysis." National security agency technical report 16 (2008): 3-1.]

Implementation:

```
while(!remove_queue.empty()){
    // remove currend edge
    int cur_edge = remove_queue.front();
    remove_queue.pop();
    int a = edges[cur_edge][0];
    int b = edges[cur_edge][1];
    remove(cur_edge, a, b, node_edges, node_neis, graph);
    truss_value.insert({cur_edge, k});

    // put others into the remove queue, if possible
    unordered_set<int> nei_a = node_neis.find(a)->second;
    unordered_set<int> nei_b = node_neis.find(b)->second;
    for(int v : nei_a){
        if(nei_b.find(v) != nei_b.end()){
            unordered_set<int> edge_v = node_edges.find(v)->second;
            for(int edge : edge_v){
                if(dedup.find(edge) != dedup.end()){
                    continue;
                }
                if(edges[edge][0] == a || edges[edge][0] == b || edges[edge][1] == a || edges[edge][1] == b){
                    int ct = --(count.find(edge)->second);
                    if(ct < k+1){
                        remove_queue.push(edge);
                        dedup.insert(edge);
                    }
                }
            }
        }
    }
}
```

Implementation:

```
while(!remove_queue.empty()){
    // remove currend edge
    int cur_edge = remove_queue.front();
    remove_queue.pop();
    int a = edges[cur_edge][0];
    int b = edges[cur_edge][1];
    remove(cur_edge, a, b, node_edges, node_neis, graph);
    truss_value.insert({cur_edge, k});

    // put others into the remove queue, if possible
    unordered_set<int> nei_a = node_neis.find(a)->second;
    unordered_set<int> nei_b = node_neis.find(b)->second;
    for(int v : nei_a){
        if(nei_b.find(v) != nei_b.end()){
            unordered_set<int> edge_v = node_edges.find(v)->second;
            for(int edge : edge_v){
                if(dedup.find(edge) != dedup.end()){
                    continue;
                }
                if(edges[edge][0] == a || edges[edge][0] == b || edges[edge][1] == a || edges[edge][1] == b){
                    int ct = --(count.find(edge)->second);
                    if(ct < k+1){
                        remove_queue.push(edge);
                        dedup.insert(edge);
                    }
                }
            }
        }
    }
}
```

Checked all neighbor vertices, even they not form a triangle

Implementation:

```
while(!remove_queue.empty()){
    // remove currend edge
    int cur_edge = remove_queue.front();
    remove_queue.pop();
    int a = edges[cur_edge][0];
    int b = edges[cur_edge][1];
    remove(cur_edge, a, b, node_edges, node_neis, graph);
    truss_value.insert({cur_edge, k});

    // put others into the remove queue, if possible
    unordered_set<int> nei_a = node_neis.find(a)->second;
    unordered_set<int> nei_b = node_neis.find(b)->second;
    for(int v : nei_a){
        if(nei_b.find(v) != nei_b.end()){
            unordered_set<int> edge_v = node_edges.find(v)->second;
            for(int edge : edge_v){
                if(dedup.find(edge) != dedup.end()){
                    continue;
                }
                if(edges[edge][0] == a || edges[edge][0] == b || edges[edge][1] == a || edges[edge][1] == b){
                    int ct = --(count.find(edge)->second);
                    if(ct < k+1){
                        remove_queue.push(edge);
                        dedup.insert(edge);
                    }
                }
            }
        }
    }
}
```

Reverse search to find triangle edge, which is inefficient

Implementation:

```cpp
void remove(int i, int a, int b, unordered_map<int, unordered_set<int>> &node_edges,
        unordered_map<int, unordered_set<int>> &node_neis, unordered_set<int> &graph){
    graph.erase(i);

    // erase edge
    unordered_set<int> *set1 = &node_edges.find(a)->second;
    (*set1).erase(i);
    unordered_set<int> *set2 = &node_edges.find(b)->second;
    (*set2).erase(i);

    // erase nei
    unordered_set<int> *set3 = &node_neis.find(a)->second;
    (*set3).erase(b);
    unordered_set<int> *set4 = &node_neis.find(b)->second;
    (*set4).erase(a);
}
```

Explicit remove vertices, edges, neighbors… not efficient

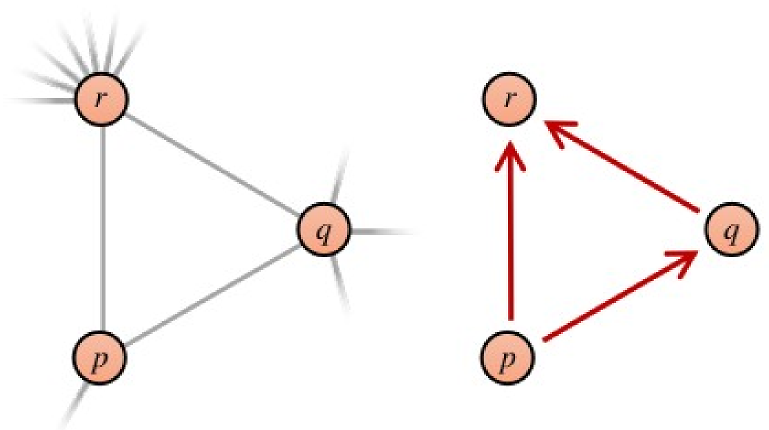
Implementation:

Reasons

- Not save the triangle information
- Each time, search & check
- set<> erase at all related place

IMPROVEMENTS FROM LITERATURE

How to use the triangle information?

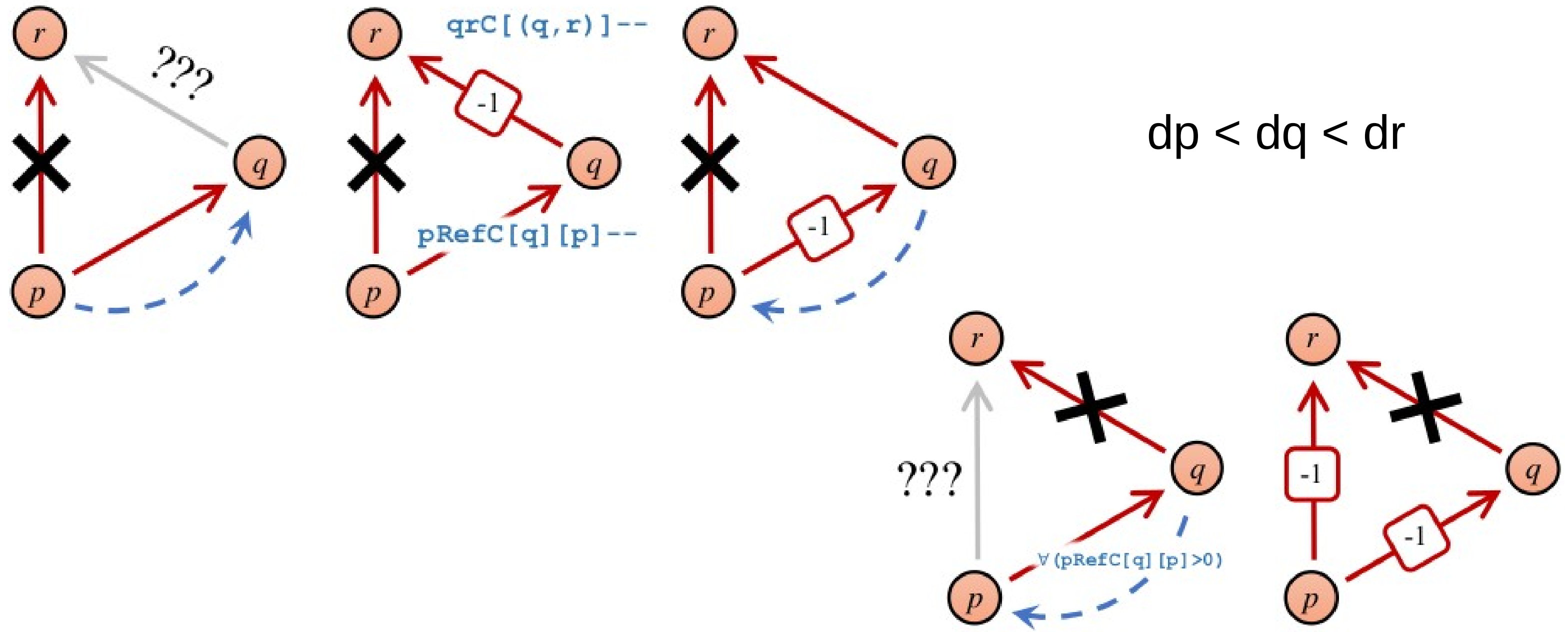


dp < dq < dr

TC[{u,v}]: triangle count
qrC[{q,r}]: count for closure of wedge
pRefC[q/r][p]: count for wedge edge

[Pearce, Roger, and Geoffrey Sanders. "K-truss decomposition for Scale-Free Graphs at Scale in Distributed Memory." 2018 IEEE High Performance extreme Computing Conference (HPEC). IEEE, 2018.]

How to use the triangle information?



dp < dq < dr

TC[{u,v}]: triangle count
qrC[{q,r}]: count for closure of wedge
pRefC[q/r][p]: count for wedge edge

[Pearce, Roger, and Geoffrey Sanders. "K-truss decomposition for Scale-Free Graphs at Scale in Distributed Memory." 2018 IEEE High Performance extreme Computing Conference (HPEC). IEEE, 2018.]

Prof. Saríyüce’s Work

Ego-Facebook data:

- Node: 4,039
- Edge: 88,234

Performance:

My code: 1min49s

Prof’s code: 0.6s

[Sariyüce, Ahmet Erdem, and Ali Pinar. "Fast hierarchy construction for dense subgraphs." Proceedings of the VLDB Endowment 10.3 (2016): 97-108.]

NEXT STEP

Next step:

- Understand the improvement by Prof. Sariyüce
- Try other methods to find set intersection (to find triangles)
    - Finding triangles ==
      finding the intersection of neighbor set of two edge vertices

THANKS!