



Efficient Shortest Path Counting on Large Road Networks

Yu-Xuan Qiu

AAII, University of Technology
Sydney
Australia
yuxuan.qiu@student.uts.edu.au

Dong Wen

The University of New South Wales
Australia
dong.wen@unsw.edu.au

Lu Qin

AAII, University of Technology
Sydney
Australia
lu.qin@uts.edu.au

Wentao Li

AAII, University of Technology
Sydney
Australia
wentao.li@uts.edu.au

Rong-Hua Li

Beijing Institute of Technology
China
lironghuabit@126.com

Ying Zhang*

AAII, University of Technology
Sydney
Australia
ying.zhang@uts.edu.au



Abstract

This paper aims to improve the efficiency of counting the shortest paths between two query vertices on a large road networks.

Introduction

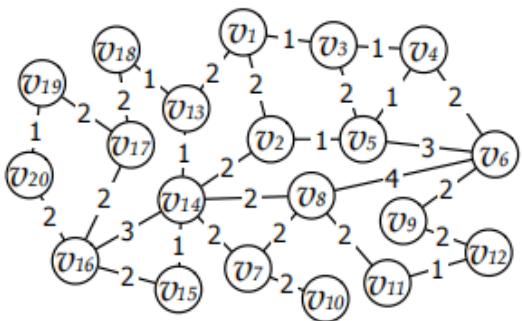


Figure 1: A road network $G(V, E)$.

- Each road is represented by an edge, and each intersection of roads is represented by a graph vertex. The real distance of each road is modeled as a weight value for each edge in the graph.
- The distance or length of a path is the sum of weights of all edges in the path.
- In real road network applications, more shortest paths indicate more traffic options and more flexibility for route planning from the start vertex to the destination.
- In addition to serving as a closeness metric, the shortest path count has been used as a building block of betweenness centrality computation

Introduction

- $C_B(u) = \sum_{s \neq u \neq t \in V} \frac{spc_u(s,t)}{spc(s,t)}$ *the betweenness centrality of u*
- $spc(s, t)$ is the number of shortest path between s and t, $spc_u(s, t)$ is the number of shortest paths between s and t
- $spc_u(s, t) = spc(s, c) \cdot spc(u, t)$ if $sd(s, t) = sd(s, u) + sd(u, t)$
- The State-Of-The-Art Solution:
- (1) a labeling-based index by assigning a total order for all vertices
- Challenges
- (1) First, based on a total vertex order, a low-ranking vertex may have a large number of labels in the index
- (2) Second, they order the labels for each vertex and perform a merge sort-like strategy to find common vertices in query processing.
- (3) Third, to compute the order-based labels, [42] searches every vertex in the induced subgraph of all vertices with lower ranks.



Introduction

- Our Approach——New labeling-based index structure
- (1) We adopt the concept of tree decomposition and propose a tree-based labeling structure, given that real-world road networks normally have a low average degree and small treewidth.
- (2) Our index is also a labeling-based structure and satisfies the concept of exact shortest path covering.
- (3) We propose several rules to reduce all the descendants of u in the tree while preserving the correctness of all shortest paths in the small reduced graph.



Introduction

- Contribution
- (1) A novel tree-based algorithm. We design a novel index structure called TL-Index.
- (2) *A new index construction paradigm.* we improve the time complexity of index construction from $\mathcal{O}(nh^2 + nh \log n)$ to $\mathcal{O}(nhw + n \log n)$, because w is typically several times smaller than h in practice.
- (3) *Extensive experiments and evaluations.*

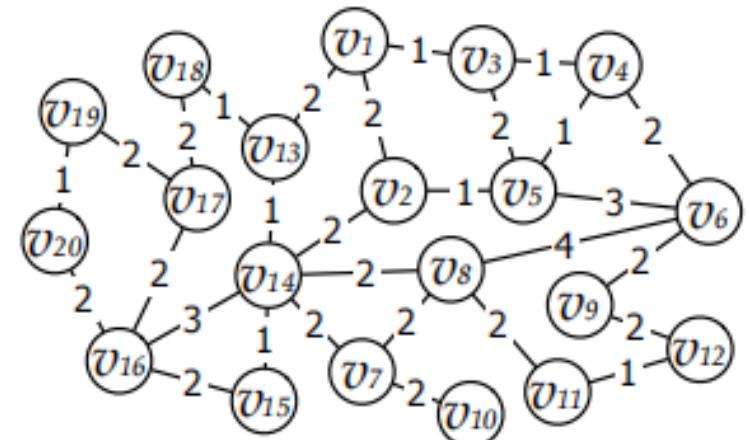


Preliminaries

- $G = (V, E, \phi)$ ϕ is a weight function for each edge
- $n = |V|$ $m = |E|$
- $N_G(v) = \{u | (u, v) \in E(G)\}$
- $p = (v_1, v_2, v_3 \dots v_k)$ a simple path between v1-vk
- $\phi(p) = \sum_{i=1}^{k-1} \phi(e(v_i, v_{i+1}))$
- $sd(s,t)$ the smallest length of all paths between s and t
- $P(s,t)$ the set of all the shortest path between s and t
- $spc(s, t) = |P(s, t)|$ the number of shortest paths

Preliminaries

- Problem Definition:
- Given a road network G and two query vertices $q = (s, t)$, the shortest path counting problem aims to efficiently compute the number of shortest paths between s and t .
- v6 and v16 : $p1 = (v6, v8, v14, v16)$ and $p2 = (v6, v4, v3, v1, v13, v14, v16)$
- $p1$ is the shortest path is 9
- The number of the shortest paths between $v6$ and $v16$ is 6 in G .

Figure 1: A road network $G(V, E)$.



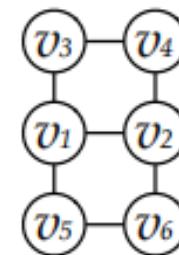
Preliminaries

- A Basic Online Method:
- Dijkstra's algorithm
- Given a vertex v , let $D[v]$ and $C[v]$ be intermediate shortest distance and shortest path count, respectively.
- $D[u] + \phi(u, v) < D[v]$, we replace $D[v]$ and $C[v]$ with $D[u] + \phi(u, v)$ and $C[u]$
- $D[u] + \phi(u, v) = D[v]$, we add $C[u]$ to $C[v]$.
- We do not update $C[v]$ and $D[v]$ if $D[u] + \phi(u, v) > D[v]$.

Preliminaries

- Hub Labeling—exact shortest path covering(ESPC)
- Each vertex u , they precompute a collection of labels $L(u)$, and each label is a triplet $(w, \text{sd}(u,w), \delta u, w)$

$$\text{spc}(u, v) = \sum_{w \in L(u), w \in L(v), \text{sd}(u,w) + \text{sd}(v,w) = \text{sd}(u,v)} \delta_{u,w} \cdot \delta_{v,w} \quad (1)$$



| Vertex | $L(\cdot)$ |
|--------|--|
| v_1 | $(v_1, 0, 1)$ |
| v_2 | $(v_2, 0, 1), (v_1, 1, 1)$ |
| v_3 | $(v_3, 0, 1), (v_2, 2, 1), (v_1, 1, 1)$ |
| v_4 | $(v_4, 0, 1), (v_3, 1, 1), (v_2, 1, 1), (v_1, 2, 2)$ |
| v_5 | $(v_5, 0, 1), (v_2, 2, 1), (v_1, 1, 1)$ |
| v_6 | $(v_6, 0, 1), (v_5, 1, 1), (v_2, 1, 1), (v_1, 2, 2)$ |

Figure 2: A simple graph and its hub-labeling index given the vertex order $v_1 \leq v_2 \leq v_3 \leq v_4 \leq v_5 \leq v_6$.



Preliminaries

- Limitation
- The label size for a vertex can be very large, and we need to scan all the labels of two query vertices in the worst case.
- The key to improving the efficiency of counting shortest paths is to reduce the number of label comparisons in query processing.

Tree-Based Shortest Path Counting

DEFINITION 1. (TREE DECOMPOSITION) Given a graph $G(V, E)$, a tree decomposition of G , denoted as T_G , is a tree in which every tree node $X \in T_G$ is a subset of V (i.e., $X \subseteq V$) such that the following conditions hold:

- (1) $\bigcup_{X \in T_G} X = V$;
- (2) for every $(u, v) \in E$, there exists $X \in T_G$ such that $u \in X$ and $v \in X$;
- (3) for every $u \in V$, $\{X | u \in X\}$ forms a connected subtree of T_G .

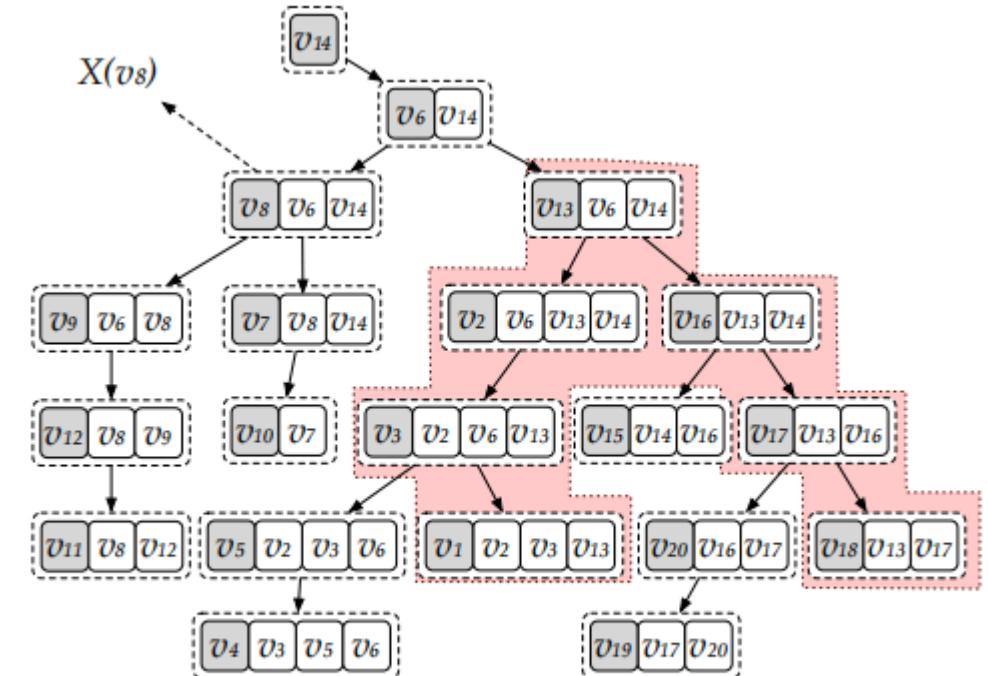


Figure 3: Tree decomposition T_G of G .

Tree-Based Shortest Path Counting

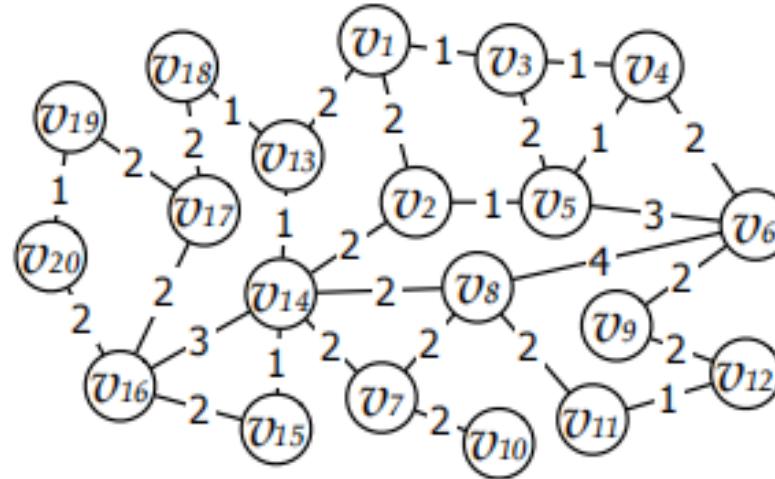


Figure 1: A road network $G(V, E)$.

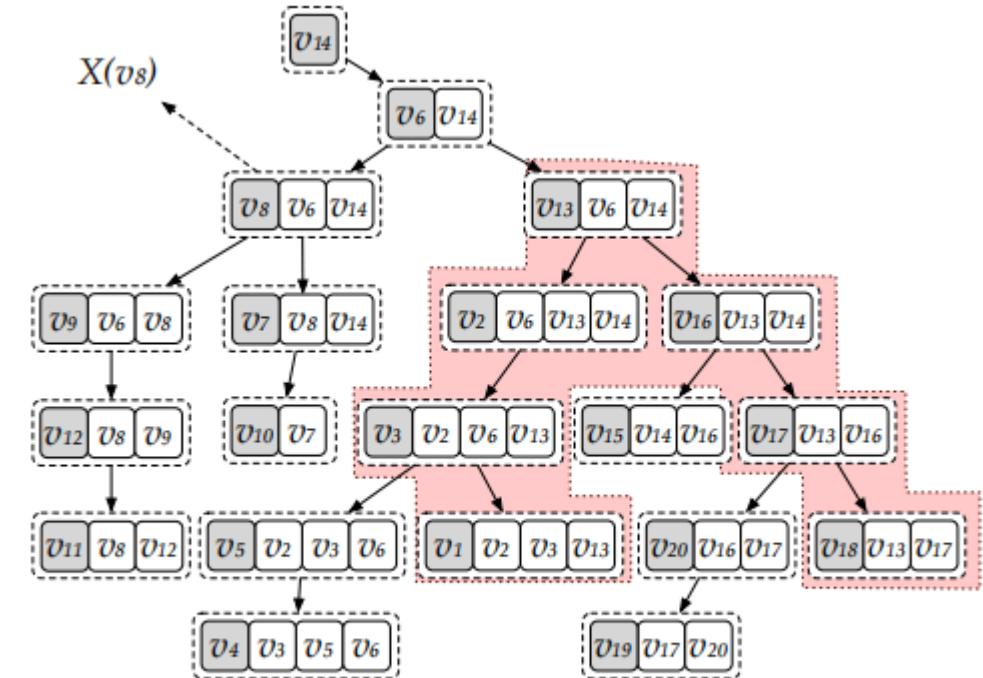
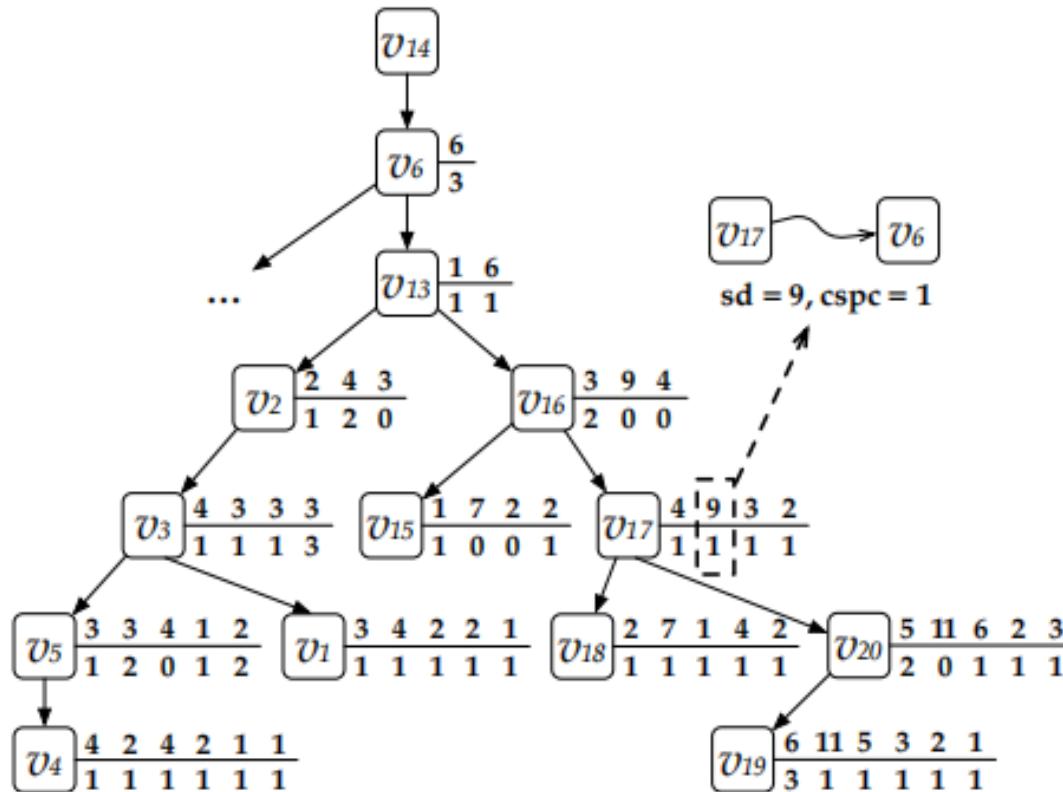


Figure 3: Tree decomposition T_G of G .

TL-Index

Figure 4: The TL-Index for G .

THEOREM 2. *The space complexity of TL-Index is $O(n \cdot h)$.*

DEFINITION 3. (CONVEX PATH) Given a tree decomposition T_G of graph $G(V, E)$, a path $p = (s, v_1, v_2, \dots, v_k, t)$ between two vertices s and t is a convex path if for every $1 \leq i \leq k$, the depth of $X(v_i)$ is larger than the smaller one of $X(s)$ and $X(t)$, i.e., $\forall 1 \leq i \leq k$, $\text{Depth}(v_i) > \min(\text{Depth}(s), \text{Depth}(t))$.

DEFINITION 4. Given a road network G , TL-Index precomputes:

- (1) a tree structure of all vertices by tree decomposition;
- (2) the shortest distance from each vertex to all its ancestors;
- (3) the convex shortest path count from each vertex to all its ancestors.

Note that by a tree structure in Definition 4, we discard all vertices except v in each tree node $X(v)$ and use v as a tree node instead of the original vertex set $X(v)$.

Query Processing with TL-Index

Algorithm 1: TL-Query

Input: the TL-Index and two query vertices s, t

Output: the shortest distance $\text{sd}(s, t)$, and corresponding count $\text{spc}(s, t)$

```

1  $v \leftarrow$  the LCA of  $s$  and  $t$  in the tree;
2  $d \leftarrow \infty, c \leftarrow 0;$ 
3 foreach  $u \in A(v) \cup \{v\}$  do
4    $d' \leftarrow \text{sd}(s, u) + \text{sd}(u, t);$ 
5   if  $d' < d$  then
6      $d \leftarrow d';$ 
7      $c \leftarrow \text{cspc}_{s,u} \cdot \text{cspc}_{u,t};$ 
8   else if  $d' = d$  then
9      $c \leftarrow c + \text{cspc}_{s,u} \cdot \text{cspc}_{u,t};$ 
10 return  $d$  and  $c$ 
```

$$\text{spc}(s, t) = \sum_{v \in \text{CA}(s, t), \text{sd}(s, v) + \text{sd}(v, t) = \text{sd}(s, t)} \text{cspc}_{s,v} \cdot \text{cspc}_{v,t}, \quad (2)$$

where $\text{cspc}_{s,v}$ denotes the number of all convex shortest paths between vertices s and v . The shortest distance between s and t can be computed as follows.

$$\text{sd}(s, t) = \min_{v \in \text{CA}(s, t)} \text{sd}(s, v) + \text{sd}(v, t) \quad (3)$$

THEOREM 3. *The time complexity of Algorithm 1 is $O(h)$, where h is the treeheight of the tree decomposition T_G .*



Basic Index Construction

Algorithm 2: TL-Construct

```
Input: A road network  $G(V, E, \phi)$ 
Output: The TL-Index of  $G$ 
1  $T_G \leftarrow \text{TreeDecomposition}(G);$ 
2 foreach  $X(u) \in T_G$  in a top-down manner do
3   foreach  $v \in T(u)$  do  $D[v] = \infty;$ 
4    $D[u] = 0, C[u] = 1;$ 
5    $Q \leftarrow$  an empty queue prioritized by  $D[\cdot]$ ;
6    $Q.\text{enqueue}(u);$ 
7   while  $Q$  is not empty do
8      $v \leftarrow Q.\text{dequeue}();$ 
9      $d \leftarrow \min_{p \in A(u)} \text{sd}(u, p) + \text{sd}(p, v);$ 
10    if  $d < D[v]$  then
11       $\text{sd}(u, v) \leftarrow d, \text{cspc}(u, v) \leftarrow 0;$ 
12      continue
13    else  $\text{sd}(u, v) \leftarrow D[v], \text{cspc}(u, v) \leftarrow C[v];$ 
14    foreach  $v' \in N(v)$  do
15       $nd \leftarrow D[v] + \phi(v, v');$ 
16      if  $D[v'] > nd \wedge \text{Depth}(v') > \text{Depth}(u)$  then
17         $D[v'] \leftarrow nd, C[v'] \leftarrow C[v];$ 
18         $Q.\text{enqueue}(v')$ 
19      else if  $D[v'] = nd$  then
20         $C[v'] \leftarrow C[v'] + C[v];$ 
```

Graph Reduction

- DC-Graph
- $G = (V, E, \phi, \varsigma)$
- $\varsigma(p) = \prod_{e \in p} \varsigma(e)$

DEFINITION 5. (DCP-GRAFH) Given a road network $G(V, E, \phi)$, a DC-Graph $G'(V', E', \phi', \varsigma)$ is a DCP-Graph if $V' \subseteq V$ and for every pair $u, v \in V'$, $\text{sd}_G(u, v) = \text{sd}_{G'}(u, v)$ and $\text{spc}_G(u, v) = \text{spc}_{G'}(u, v)$.

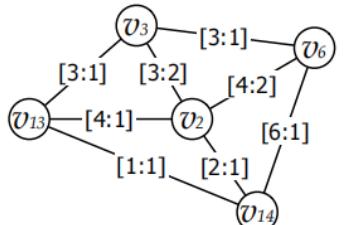
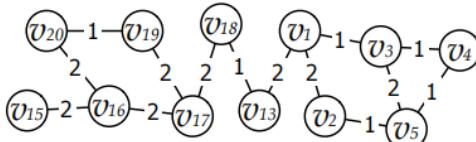
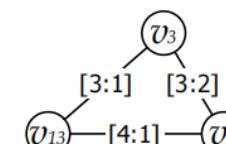
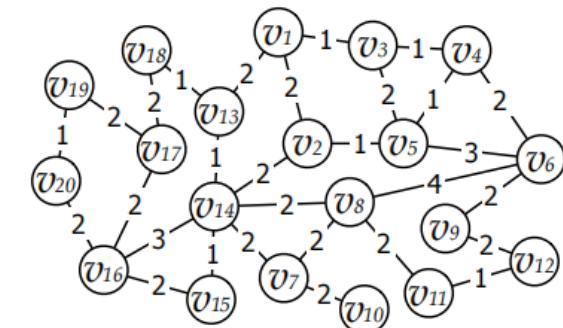
(a) A reduced graph G' of G (b) Local graph for v_3 and v_{13} in the original graph G .(c) Local graph for v_3 and v_{13} in the reduced graph G' .Figure 1: A road network $G(V, E)$.

Figure 5: Examples of optimizations in index construction based on the graph G in Figure 1 and its tree decomposition in Figure 3. For each edge in the subfigures (a) and (c), the label means [the distance weight ϕ : the count weight ς]. For the subfigure (b), the label means the distance of the edge.

Graph Reduction

Algorithm 4: DCP-TreeDecomposition

Input: $G(V, E, \phi)$
Output: Tree decomposition T_G

```

 $T_G \leftarrow \emptyset$ 
foreach  $e \in E$  do  $\varsigma(e) = 1$ ;
 $V' \leftarrow V, i \leftarrow 1$ ;
while  $V \neq \emptyset$  do
     $u \leftarrow$  the vertex with the smallest degree in  $V$ ;
     $X(u) \leftarrow \{u\} \cup N(u)$ ;
    create a tree node  $X(u)$  in  $T_G$ ;
     $G \leftarrow G \ominus u$ ;
     $\pi(u) = i$ ;
     $i \leftarrow i + 1$ ;
foreach  $u \in V'$  do
    if  $|X(u)| > 1$  then
         $v \leftarrow \arg \min_{v \in X(u) \setminus \{u\}} \pi(v)$ ;
        set  $X(v)$  be the parent of  $X(u)$  in  $T_G$ ;
return  $T_G$ 

```

Algorithm 3: Operator \ominus

Input: A road network G and a vertex $u \in V(G)$
Output: The graph $G \ominus u$

```

1 foreach  $v, w \in N(u)$  do
2     if  $(v, w) \notin E \vee \phi(v, w) > \phi(v, u) + \phi(u, w)$  then
3          $E \leftarrow E \cup \{(v, w)\}$ ;
4          $\phi(v, w) \leftarrow \phi(v, u) + \phi(u, w)$ ;
5          $\varsigma(v, w) \leftarrow \varsigma(v, u) \times \varsigma(u, w)$ ;
6     else if  $\phi(v, w) = \phi(v, u) + \phi(u, w)$  then
7          $\varsigma(v, w) \leftarrow \varsigma(v, w) + \varsigma(v, u) \times \varsigma(u, w)$ ;
8 remove  $u$  and all incident edges from  $G$ ;

```

LEMMA 5. The time complexity of Algorithm 4 is $O(n \cdot w^2 + n \log n)$.

Relaxing Convex Shortest path

DEFINITION 6. (LOCAL SHORTEST DISTANCE AND LOCAL SHORTEST PATH COUNT) Given a tree decomposition T_G of graph $G(V, E)$, the local shortest distance (resp. shortest path count) between two vertices u and v , denoted by $\text{sd}(u, v)^-$ (resp. $\text{cspc}(u, v)^-$), is the shortest distance (resp. shortest path count) of u and v in their local graph.

LEMMA 7. Algorithm 1 is correct if we replace the shortest distance and the convex shortest path count in TL-Index (Definition 4) by the local shortest distance and the local shortest path count, respectively.

Algorithm 5: TL-Construct*

Input: A road network $G(V, E, \phi)$

Output: The TL-Index of G

```
1  $T_G \leftarrow \text{DCP-TreeDecomposition}(G);$ 
2 foreach  $X(u) \in T_G$  in a top-down manner do
3   foreach  $v \in A(u)$  do
4     foreach  $u' \in X(u) \setminus \{u\}$  do
5       if  $\text{Depth}(u') < \text{Depth}(v)$  then continue;
6        $d \leftarrow \phi(u, u') + \text{sd}(u', v)^-;$ 
7        $c \leftarrow \varsigma(u, u') \cdot \text{cspc}(u', v)^-;$ 
8       if  $d < \text{sd}(u, v)^-$  then
9          $\text{sd}(u, v)^- \leftarrow d;$ 
10         $\text{cspc}(u, v)^- \leftarrow c;$ 
11       else if  $d = \text{sd}(u, v)^-$  then
12          $\text{cspc}(u, v)^- \leftarrow \text{cspc}(u, v)^- + c;$ 
```

Relaxing Convex Shortest path

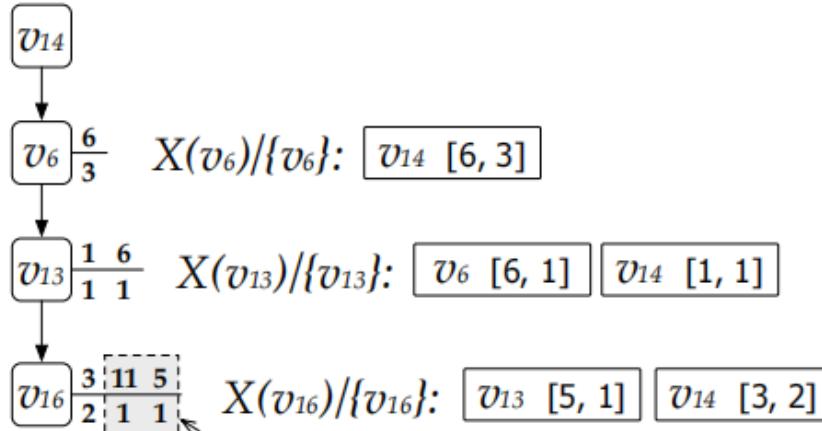


Figure 6: An example of TL-Construct*.

THEOREM 5. *The time complexity of Algorithm 5 is $O(n \log n + nhw)$, where n is the number of vertices, h is the treeheight, and w is the treewidth.*

Algorithm 5: TL-Construct*

```

Input: A road network  $G(V, E, \phi)$ 
Output: The TL-Index of  $G$ 
1  $T_G \leftarrow \text{DCP-TreeDecomposition}(G);$ 
2 foreach  $X(u) \in T_G$  in a top-down manner do
3   foreach  $v \in A(u)$  do
4     foreach  $u' \in X(u) \setminus \{u\}$  do
5       if  $\text{Depth}(u') < \text{Depth}(v)$  then continue;
6        $d \leftarrow \phi(u, u') + \text{sd}(u', v)^-;$ 
7        $c \leftarrow \varsigma(u, u') \cdot \text{cspc}(u', v)^-;$ 
8       if  $d < \text{sd}(u, v)^-$  then
9          $\text{sd}(u, v)^- \leftarrow d;$ 
10         $\text{cspc}(u, v)^- \leftarrow c;$ 
11       else if  $d = \text{sd}(u, v)^-$  then
12          $\text{cspc}(u, v)^- \leftarrow \text{cspc}(u, v)^- + c;$ 

```

Experiments

Table 1: Statistics of road networks.

| Name | Description | <i>n</i> | <i>m</i> | <i>h</i> | <i>w</i> |
|------|------------------|------------|------------|----------|----------|
| GRD | US Power Grid | 4,941 | 6,594 | 72 | 25 |
| SYD | Public Transport | 24,063 | 28,695 | 194 | 79 |
| NY | NYC | 264,346 | 733,846 | 505 | 134 |
| BAY | Bay Area | 321,270 | 800,172 | 403 | 108 |
| COL | Colorado | 435,666 | 1,057,066 | 465 | 146 |
| FLA | Florida | 1,070,376 | 2,712,798 | 520 | 136 |
| NW | Northwest US | 1,207,945 | 2,840,208 | 548 | 146 |
| NE | Northeast US | 1,524,453 | 3,897,636 | 828 | 219 |
| CAL | CA and NV | 1,890,815 | 4,657,742 | 713 | 215 |
| LKS | Great Lakes | 2,758,119 | 6,885,658 | 1325 | 370 |
| EUS | Eastern US | 3,598,623 | 8,778,114 | 1022 | 272 |
| WUS | Western US | 6,262,104 | 15,248,146 | 1041 | 326 |
| CUS | Central US | 14,081,816 | 34,292,496 | 2433 | 660 |
| USA | Full US | 23,947,347 | 58,333,344 | 2564 | 693 |

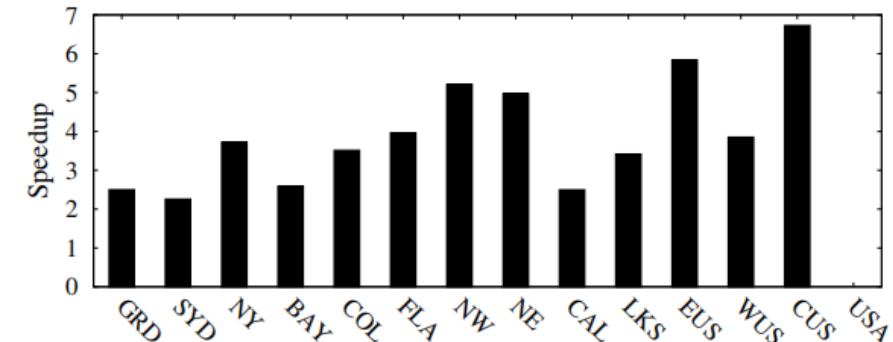
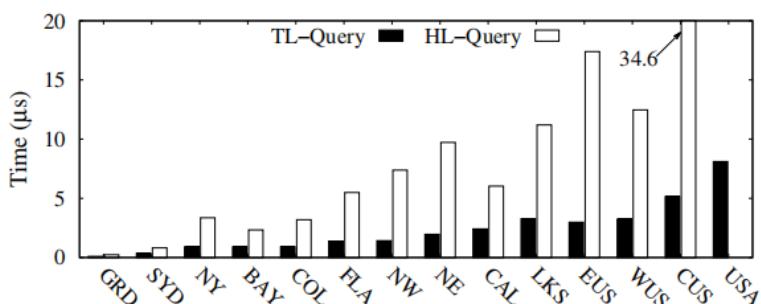


Figure 8: TL-Query speedup over HL-Query.

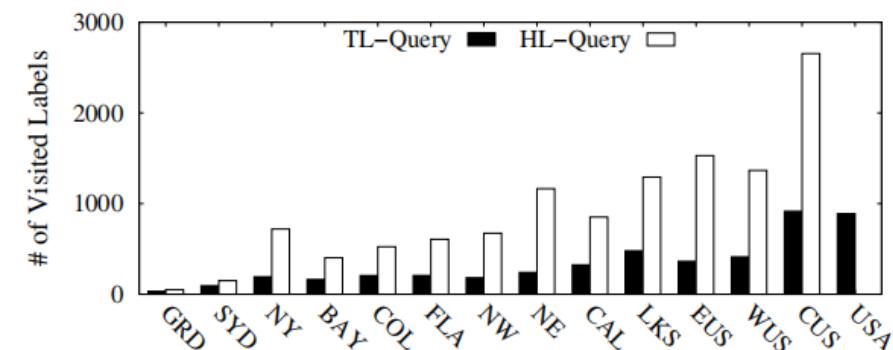


Figure 9: Number of visited labels in query processing.

Experiments

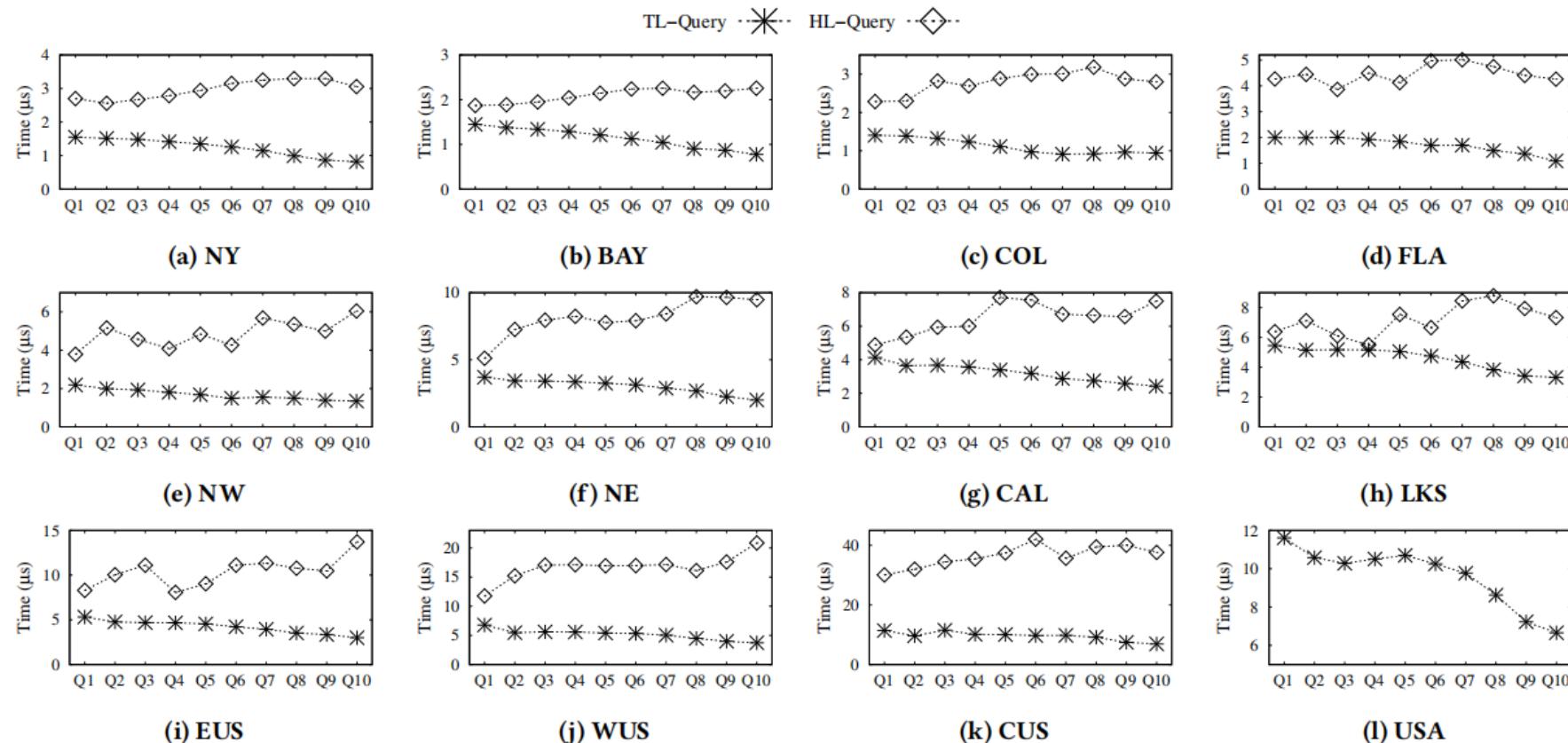


Figure 10: Query processing time varying query distance.

Experiments

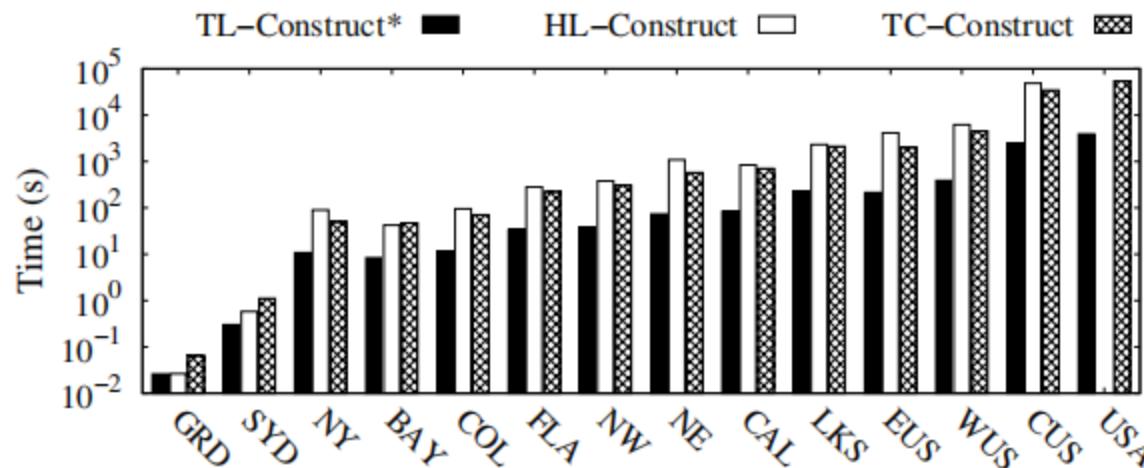


Figure 11: Index construction time.

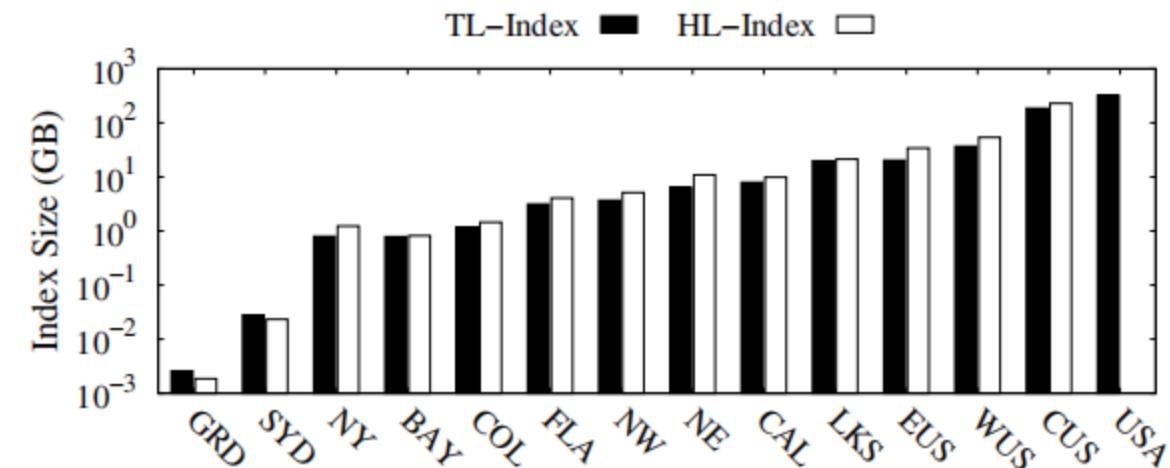
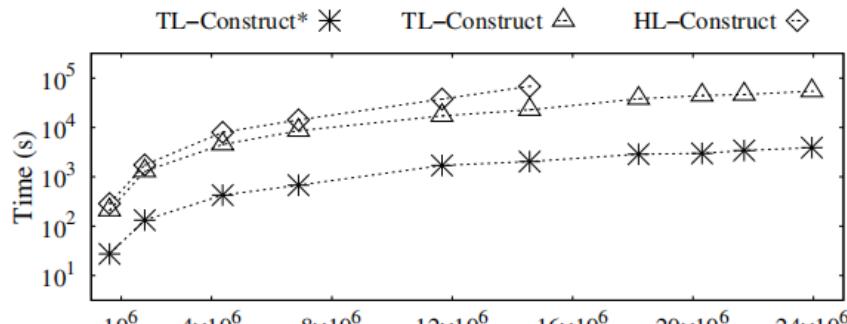


Figure 12: Index size (GB).

Experiments



(a) Indexing time.

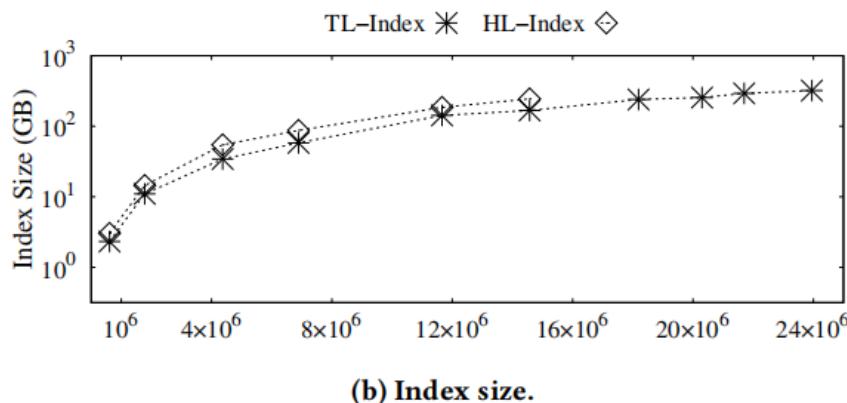


Figure 13: Scalability testing.

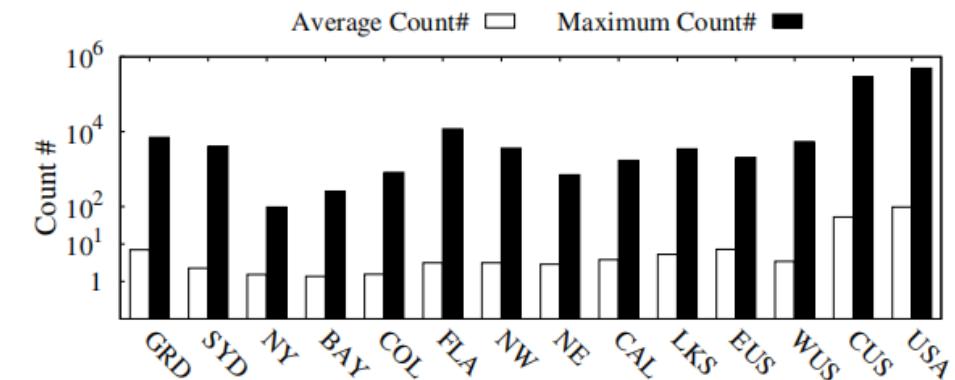


Figure 14: Shortest path count on different graphs.

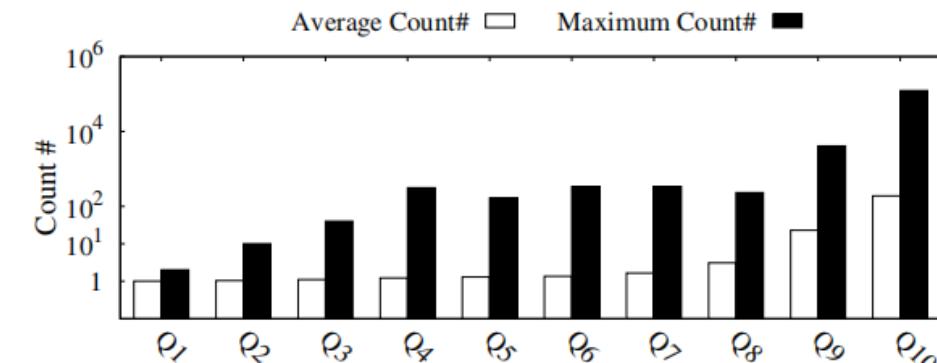


Figure 15: Shortest path count varying distance.

Real Graph Datasets

- Five Real-World Graphs
 - as-Skitter: from SNAP, is and internet topology graph
 - soc-LiveJournal1: from SNAP, is an online community
 - twitter-2010: from LAW, is a social network
 - uk-2005, it-2004: from LAW, two web graphs crawled within the .uk and .it domains

| Graphs | <i>n</i> | <i>m</i> | $d_{avg}(G)$ | $d_{max}(G)$ | $\delta(G)$ |
|------------------|------------|---------------|--------------|--------------|-------------|
| as-Skitter | 1,694,616 | 11,094,209 | 13.09 | 35,455 | 111 |
| soc-LiveJournal1 | 4,843,953 | 42,845,684 | 17.69 | 20,333 | 372 |
| uk-2005 | 39,252,879 | 781,439,892 | 39.82 | 1,776,858 | 588 |
| it-2004 | 41,290,577 | 1,027,474,895 | 49.77 | 1,326,744 | 3,224 |
| twitter-2010 | 41,652,230 | 1,202,513,046 | 57.74 | 2,997,487 | 2,488 |

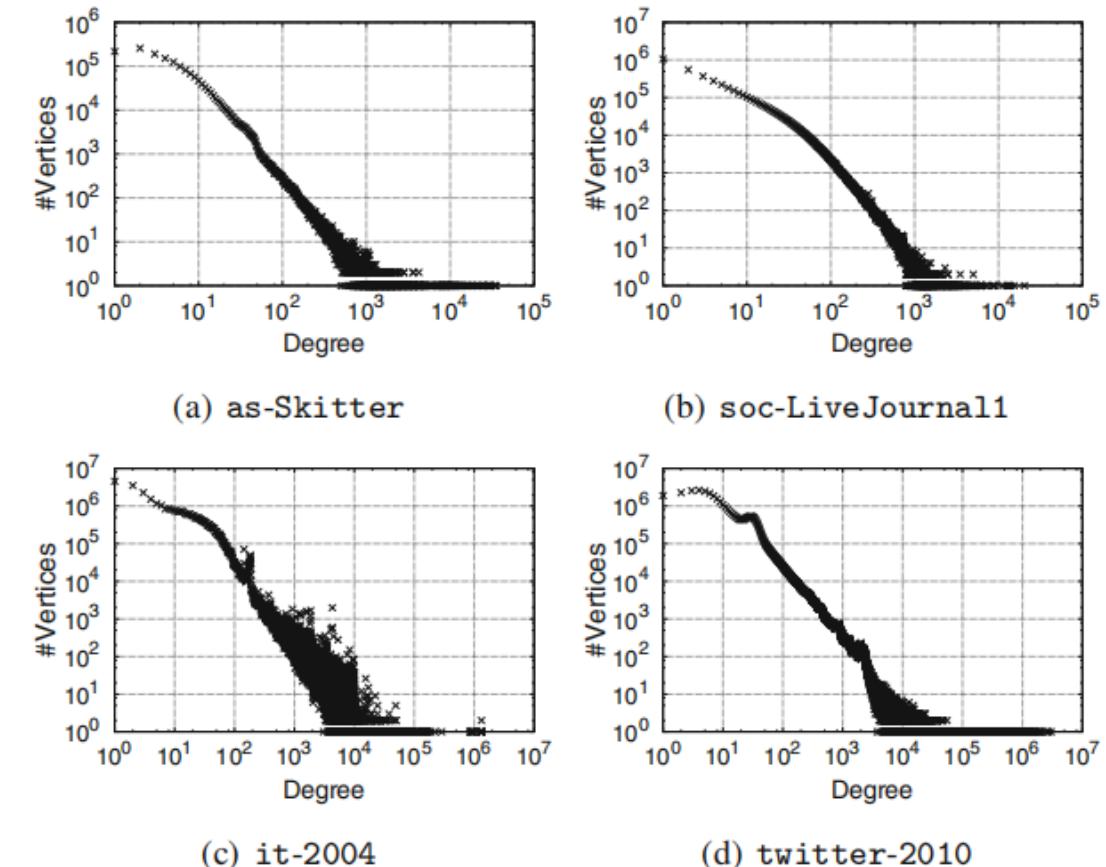
Table 1.1: Statistics of five real graphs ($\delta(G)$ is the degeneracy of G)

Fig. 1.2: Degree distributions

Representation of Large Sparse Graphs

- Use a variant of the adjacency list representation, called adjacency array representation, which is also known as the Compressed Sparse Row (CSR) [1].
- The start position (i.e., index) of the set of neighbors of vertex i in the array edges is stored in $pstart[i]$, while $pstart[n]$ stores the length of the array edges.
- In this way, the degree of vertex i can be obtained in constant time as $pstart[i + 1] - pstart[i]$, and the set of neighbors of vertex i is stored consecutively in the subarray edges [$pstart[i], \dots, pstart[i+1]-1$].

Listing 1.1: Graph memory allocation

```
typedef unsigned int uint;
uint *pstart = new uint[n+1];
uint *edges = new uint[2*m];
```

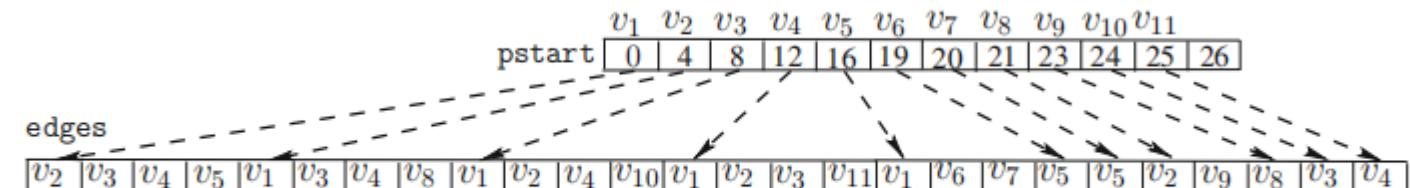


Fig. 1.3: Adjacency array graph representation

[1] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

Complexity Analysis

- Time Complexity analysis

In this paper, we will provide time complexity analysis for all the presented algorithms by using the big-O notation $\mathcal{O}(\cdot)$. Specifically, for two given functions $f(n)$ and $f'(n)$, $f'(n) \in \mathcal{O}(f(n))$ if there exist positive constants c and n_0 such that $f'(n) \leq c \times f(n)$ for all $n \geq n_0$; note that $\mathcal{O}(\cdot)$ denotes a set of functions. Occasionally, we will also use the θ -notation. Specifically, for two given functions $f(n)$ and $f'(n)$, $f'(n) \in \theta(f(n))$ if there exist positive constants c_1, c_2 , and n_0 , such that $c_1 \times f(n) \leq f'(n) \leq c_2 \times f(n)$ for all $n \geq n_0$.

- Space Complexity analysis
- As the number m of edges usually is much larger than the number n of vertices for large real-world graphs, we analyze the space complexity in the form of $c \times m + \sigma(n)$ by explicitly specifying the constant c , since $c \times m$ usually is the dominating factor.

Cohesive Subgraphs Computation

Given an input graph G , *cohesive subgraph computation* is either to find all maximal subgraphs of G whose cohesiveness values are at least k for all possible k values, or to find the subgraph of G with the largest cohesiveness value.

- **Minimum degree** (aka, k -core, see Chapter 3): that is, the maximal subgraph whose minimum degree is at least k , which is called k -core. The problem is either to compute the k -core for a user-given k or to compute k -cores for all possible k values.
- **Average degree** (aka, dense subgraph, see Chapter 4); that is a subgraph with average degree at least k . The problem studied usually is to compute the subgraph with the largest average (i.e., dense subgraph).
- **Higher-order Variants of k -core and Densest Subgraph** (see Chapter 5); for example, the maximal subgraph in which each edge participates in at least k triangles within the subgraph (i.e., k -truss), the subgraph where the average number of triangles each vertex participates is the largest (i.e., triangle-dense subgraph).
- **Edge connectivity** (aka, k -edge connected components, see Chapter 6); that is, the maximal subgraphs each of which is k -edge connected. The problem studied is either to compute the k -edge connected components for a user-given k or to compute k -edge connected components for all possible k values.

Cohesive Subgraphs Computation

- **Clique**: each vertex is connected to all other vertices (i.e. $|E(g)| = \frac{|V(g)|(|V(g)|-1)}{2}$).
- **r -quasi clique**: at least r portion of its vertex are connected by edges (i.e., $|E(g)| \geq \gamma \times \frac{|V(g)|(|V(g)|-1)}{2}$).
- **k -plex** : every vertex of g is connected to all but no more than $(k-1)$ other vertices.
- Nevertheless, cohesive subgraph computation based on these definitions usually leads to NP-Hard problems, and thus are generally computationally too expensive to be applied to large graphs. Consequently, we do not consider these alternative cohesiveness measures in this book.



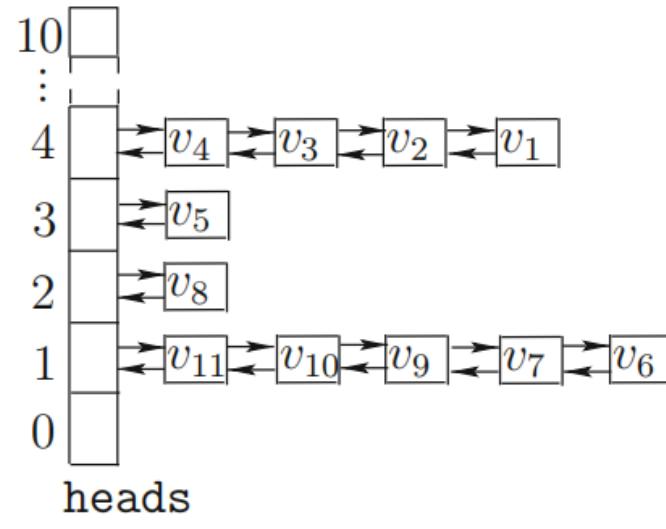
Applications

Cohesive subgraph computation can either be the main goal of a graph analysis task or act as a preprocessing step aiming to reduce/trim the graph by removing sparse parts such that more complex and time-consuming analysis can be conducted. For example, some of the applications of cohesive subgraph computation are illustrated as follows:

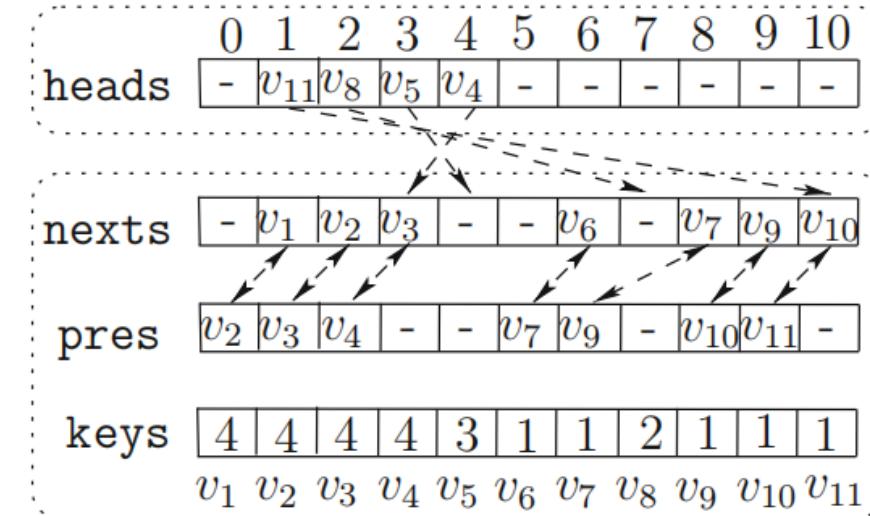
- **Community Search**
- **Locating Influential Nodes**
- **Keyword Extracting from Text**
- **Link Spam Detection**

Linked List-Based Linear Heap

- **heads:** the heads of doubly linked lists
- **pres and nexts:** the information of doubly linked lists
- **keys:** the key values of all elements



(a) Conceptual view



(b) Actual storage

Fig. 2.1: An example of linked list-based linear heap



Linked List-Based Linear Heap

- **key_cap:** maximum allowed key value
- **max_key:** upper bound of the current maximum key value
- **min_key:** lower_bound of the current minimum key value
- **Space complexity of ListLinearHeap:**
 $3n + key_cap + \mathcal{O}(1)$

Listing 2.1: Interface of a linked list-based linear heap

```
class ListLinearHeap {  
private:  
    uint n; // total number of possible distinct elements  
    uint key_cap; // maximum allowed key value  
    uint max_key; // upper bound of the current maximum key value  
    uint min_key; // lower bound of the current minimum key value  
    uint *keys; // key values of elements  
    uint *heads; // the first element in a doubly linked list  
    uint *pres; // previous element in a doubly linked list  
    uint *nexts; // next element in a doubly linked list  
  
public:  
    ListLinearHeap(uint _n, uint _key_cap) ;  
    ~ListLinearHeap() ;  
    void init(uint _n, uint _key_cap, uint *_elems, uint *_keys) ;  
    void insert(uint element, uint key) ;  
    uint remove(uint element) ;  
    uint get_n() { return n; }  
    uint get_key_cap() { return key_cap; }  
    uint get_key(uint element) { return keys[element]; }  
    uint increment(uint element, uint inc) ;  
    uint decrement(uint element, uint dec) ;  
    bool get_max(uint &element, uint &key) ;  
    bool pop_max(uint &element, uint &key) ;  
    bool get_min(uint &element, uint &key) ;  
    bool pop_min(uint &element, uint &key) ;  
}
```



Linked List-Based Linear Heap

Algorithm 1: init(_n, _key_cap, _elems, _keys)

```
/* Initialize max_key, min_key and heads */  
1 max_key ← 0; min_key ← key_cap;  
2 for key ← 0 to _key_cap do  
3   heads[key] ← null;  
  /* Insert (element, key) pairs into the data structure */  
4 for i ← 0 to _n – 1 do  
5   insert(_elems[i], _keys[i]);
```

Note that, to use ListLinearHeap after constructing it, the member function init needs to be invoked first to properly initialize the member variables before invoking any other member functions.

Linked List-Based Linear Heap

Algorithm 1: init(_n, _key_cap, _elems, _keys)

```
/* Initialize max_key, min_key and heads */  
1 max_key ← 0; min_key ← key_cap;  
2 for key ← 0 to _key_cap do  
3   heads[key] ← null;  
  /* Insert (element, key) pairs into the data structure */  
4 for i ← 0 to _n – 1 do  
5   insert(_elems[i], _keys[i]);
```

- (1) allocate proper memory space for the data structure,
- (2) assign proper initial values for max key, min key, and heads
- (3) insert the (element, key) pairs, supplied in the input parameter to init, into the data structure.



Linked List-Based Linear Heap

Algorithm 2: insert(element, key)

```
/* Update doubly linked list */  
1 keys[element] ← key; pres[element] ← null; nexts[element] ← heads[key];  
2 if heads[key] ≠ null then pres[heads[key]] ← element;  
3 heads[key] ← element;  
/* Update min_key and max_key */  
4 if key < min_key then min_key ← key;  
5 if key > max_key then max_key ← key;
```

- (1) Put element at the beginning of the doubly linked list pointed by heads[key]
- (2) Update the values of max_key and min_key



Linked List-Based Linear Heap

Algorithm 3: remove(element)

```
1 if pres[element] = null then
    /* element is at the beginning of a doubly linked list */
2     heads[keys[element]] ← nexts[element];
3     if nexts[element]! = null then pres[nexts[element]] ← null;
4 else
5     nexts[pres[element]] ← nexts[element];
6     if nexts[elements]! = null then pres[nexts[elements]] ← pres[element];
7 return keys[element];
```

- (1) Update by adding a direct link between the immediate preceding element `pres[element]` and the immediate succeeding element `nexts[element]` of element
- (2) Return the key value of the removed element.



Linked List-Based Linear Heap

Algorithm 4: decrement(element, dec)

```
1 key ← remove(element);  
2 key ← key – dec;  
3 insert(element, key);  
4 return key;
```

- (1) To update the key value of an element, the element is firstly removed from the doubly linked list corresponding to the key value `keys[element]` and is then inserted into the doubly linked list corresponding to the updated key value.
- (2) Return the key value of the updated element.



Linked List-Based Linear Heap

Algorithm 5: pop_min(element, key)

```
1 while min_key ≤ max_key and heads[min_key] = null do
2   min_key ← min_key + 1;
3 if min_key > max_key then
4   return false;
5 else
6   element ← heads[min_key]; key ← min_key;
7   remove(element);
8   return true;
```

- (1) To pop the element with the minimum key value from the data structure, the value of min key is firstly updated to be the smallest value such that $heads[min_key] \neq null$;
- (2) If such min_key exists, remove the first element from the data structure
- (3) Otherwise, the data structure currently contains no element.



Time Complexity of ListLinearHeap

- (1) **Initialization:** takes $\mathcal{O}(n + \text{key_cap})$ time
- (2) Each of the remaining member functions other than **get_min**, **pop_min**, **get_max**, and **pop_max** runs in constant time.
- (3) Pop_min, pop_max, get_min, get_max:

Theorem 2.1. *After the initialization by init, a sequence of `x decrement(id, 1)`, `increment(id, 1)`, `get_min`, `pop_min`, `get_max`, `pop_max`, and remove operations takes $\mathcal{O}(\text{key_cap} + x)$ time. Note that: (1) decrement and increment are only allowed to change the key value of an element by 1, and (2) insert is not allowed.*

Time Complexity of ListLinearHeap

Theorem 2.1. After the initialization by `init`, a sequence of `x decrement(id, 1)`, `increment(id, 1)`, `get_min`, `pop_min`, `get_max`, `pop_max`, and `remove` operations takes $\mathcal{O}(_key_cap + x)$ time. Note that: (1) decrement and increment are only allowed to change the key value of an element by 1, and (2) insert is not allowed.

The most time-consuming part of `get_min` (similar to Algorithm 5) is updating `min_key`, while other parts can be conducted in constant time. Let t^- denote the number of times that `min_key` is decreased; note that `min_key` is only decreased in `decrement(id, 1)` and by 1 each time. Thus, t^- equals the number of invocations of `decrement(id, 1)` and $t^- \leq x$. Similarly, let t^+ denote the number of times that `min_key` is increased; note that `min_key` is increased only in `get_min`, but not in `decrement`, `increment`, or `remove`. Then, the time complexity of a sequence of x `decrement(id, 1)`, `increment`, `get_min`, and `remove` operations is $\mathcal{O}(x + t^- + t^+)$. It can be verified that $t^+ \leq t^- + _key_cap$. Therefore, the above time complexity is $\mathcal{O}(_key_cap + x)$.



Time Complexity of ListLinearHeap

Lemma 2.1. *A set of n elements that are given as input to `init` can be sorted in non-decreasing key value order (or non-increasing key value order) in $\mathcal{O}(\text{key_cap} + n)$ time.*

Proof. To sort in non-decreasing key value order, we invoke `pop_min` n times after initializing by `init`. The time complexity follows from Theorem 2.1. \square

Array-based Linear Heap

- **ids:** all elements with the same key value are stored consecutively in an array ids
- **heads:** the start position of the elements for each distinct key value is stored in heads
- **keys:** the key values of all elements
- **rids:** the position of elements in ids are stored in an array rids (i.e., $\text{rids}[\text{ids}[i]] = i$).

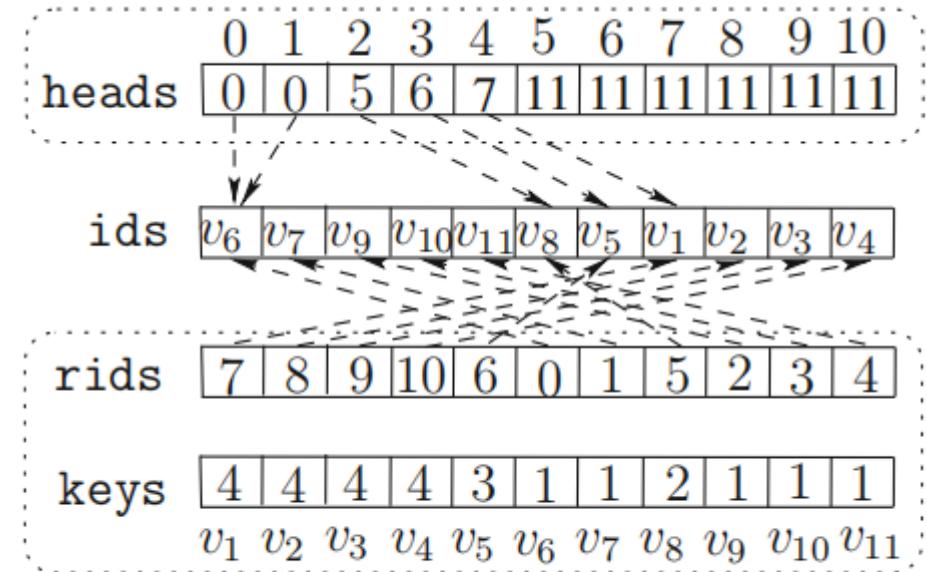


Fig. 2.2: An example of array-based linear heap



Array-based Linear Heap

Listing 2.2: Interface of an array-based linear heap

```
class ArrayLinearHeap {  
private:  
    uint n; // total number of possible distinct elements  
    uint key_cap; // maximum allowed key value  
    uint max_key; // upper bound of the current maximum key value  
    uint min_key; // lower bound of the current minimum key value  
    uint *keys; // key values of elements  
    uint *heads; // start position of elements with a specific key  
    uint *ids; // element ids  
    uint *rids; // reverse of ids, i.e., rids[ids[i]] = i  
  
public:  
    ArrayLinearHeap(uint _n, uint _key_cap) ;  
    ~ArrayLinearHeap() ;  
    void init(uint _n, uint _key_cap, uint *_ids, uint *_keys) ;  
    uint get_n() { return n; }  
    uint get_key_cap() { return key_cap; }  
    uint get_key(uint element) { return key_s[element]; }  
    void increment(uint element) ;  
    void decrement(uint element) ;  
    bool get_max(uint &element, uint &key) ;  
    bool pop_max(uint &element, uint &key) ;  
    bool get_min(uint &element, uint &key) ;  
    bool pop_min(uint &element, uint &key) ;  
}
```

- **Space complexity of Array-Based Linear Heap:** $3n + key_cap + \mathcal{O}(1)$
- Insert and remove are disabled
- Increment and decrement are only allowed to update the key value of an element by 1

Array-based Linear Heap

Algorithm 6: init(n, key_cap, ids, keys)

```
/* Initialize max_key, min_key, and keys */  
1 max_key ← 0; min_key ← key_cap;  
2 for i ← 0 to n − 1 do  
3   keys[_ids[i]] ← keys[i];  
4   if keys[i] > max_key then max_key ← keys[i];  
5   if keys[i] < min_key then min_key ← keys[i];  
/* Initialize ids, rids */  
6 Create an array cnt of size max_key + 1, with all entries 0; /*/  
7 for i ← 0 to n − 1 do cnt[keys[i]] ← cnt[keys[i]] + 1;  
8 for i ← 1 to max_key do cnt[i] ← cnt[i] + cnt[i − 1];  
9 for i ← 0 to n − 1 do  
10  cnt[keys[i]] ← cnt[keys[i]] − 1; /*/  
11  rids[_ids[i]] ← cnt[keys[i]];  
12 for i ← 0 to n − 1 do ids[rids[_ids[i]]] ← _ids[i]; /*/  
/* Initialize heads */  
13 heads[min_key] ← 0;  
14 for key ← min_key + 1 to max_key + 1 do  
15  heads[key] ← heads[key − 1];  
16  while heads[key] < n and keys[ids[heads[key]]] < key do  
17    heads[key] ← heads[key] + 1;
```

- It needs to sort all elements in ids in non-decreasing key value order
- Initialize max_key, min_key, and keys
- Initialize ids, rids
- Initialize heads



Array-based Linear Heap

Algorithm 7: decrement(element)

```
1 key ← keys[element];
2 if heads[key] ≠ rids[element] then
3     Swap the content of ids for positions heads[key] and rids[element];
4     rids[ids[rids[element]]] ← rids[element];
5     rids[ids[heads[key]]] ← heads[key];
6 if min_key = key then
7     min_key ← min_key - 1;
8     heads[min_key] ← heads[min_key + 1];
9 heads[key] ← heads[key] + 1; keys[element] ← keys[element] - 1;
10 return keys[element];
```

- Element is first moved to be at position heads[key] (by swapping with).
- The start position of elements in ids with key value key is increased by one (min_key may also be updated in decrement).



Array-based Linear Heap

- Time Complexity of Array-Linear-Heap

- (1) Are the same as that of List-Linear-Heap
- (2) The counting sort in init runs in $\mathcal{O}(_n+_key_cap)$ time

Theorem 2.2. *After the initialization by init, a sequence of x decrement, increment, get_min, pop_min, get_max, and pop_max operations takes $\mathcal{O}(_key_cap + x)$ time.*



LazyLinearHeap

- Firstly, LazyLinearHeap is stored by three rather than four arrays——singly linked list, pre is not needed
- Secondly, it does not greedily maintain an element into the proper adjacency list as done in ListLinearHeap, when its key value is decremented.

Listing 2.3: Interface of a lazy-update linear heap

```
class LazyLinearHeap {  
private:  
    uint n; // total number of possible distinct elements  
    uint key_cap; // maximum allowed key value  
    uint max_key; // upper bound of the current maximum key value  
    uint *keys; // key values of elements  
    uint *heads; // the first element in a singly linked list  
    uint *nexts; // next element in a singly linked list  
  
public:  
    LazyLinearHeap(uint _n, uint _key_cap) ;  
    ~LazyLinearHeap() ;  
    void init(uint _n, uint _key_cap, uint *_elems, uint *_keys) ;  
    uint get_n() { return n; }  
    uint get_key_cap() { return key_cap; }  
    uint get_key(uint element) { return keys[element]; }  
    uint decrement(uint element, uint dec) {  
        return keys[element] -= dec;  
    }  
    bool get_max(uint &element, uint &key) ;  
    bool pop_max(uint &element, uint &key) ;  
}
```

LazyLinearHeap

Algorithm 8: pop_max(element, key)

```
1 while true do
2     while max_key > 0 and heads[max_key] = null do
3         max_key ← max_key - 1;
4     if heads[max_key] = null then
5         return false;
6     element ← heads[max_key];
7     heads[max_key] ← nexts[element];           /* Remove element */;
8     if keys[element] = max_key then
9         /* element has the maximum key value          */
10        key ← max_key;
11        return true;
12    else
13        /* Insert element into the proper singly linked list */
14        nexts[element] ← heads[keys[element]];
15        heads[keys[element]] ← element;
```

- It iteratively retrieves and removes the first element in the singly linked list corresponding to the maximum key value max key.
- If the key value of element equals max key, then it indeed has the maximum key value and is returned.
- Otherwise, the key value of element must be smaller than max key, and thus element is inserted into the proper singly linked list.



LazyLinearHeap

Lemma 2.2. *For an arbitrary sequence of decrement and pop_max operations, the running time of LazyLinearHeap is no larger than that of ListLinearHeap.*

Proof. It is easy to verify that each element that was moved from one singly linked list to another singly linked list in pop_max (Algorithm 5) in LazyLinearHeap corresponds to a set of decrement(element, dec) operations in ListLinearHeap. Thus, the lemma holds. \square

Minimum Degree-Based Core Decomposition

Definition 3.1 ([81]). Given a graph G and an integer k , the k -core of G is the maximal subgraph g of G such that the *minimum degree* of g is at least k ; that is, every vertex in g is connected to at least k other vertices in g .

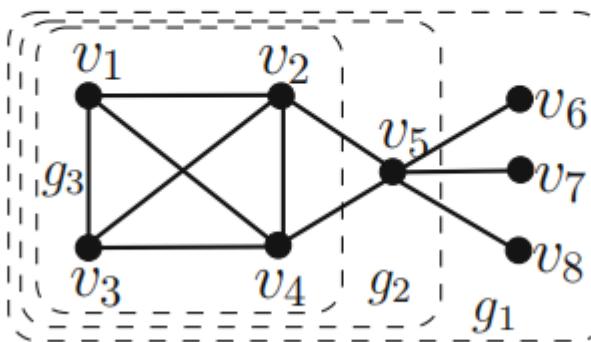


Fig. 3.1: An example graph and its k -cores

Core Decomposition. The problem of core decomposition is to compute the k -cores of an input graph G , for all possible values of k .

Definition 3.2. Given a graph G , the *core number* of a vertex u in G , denoted $\text{core}(u)$, is the largest k such that the k -core of G contains u .

Minimum Degree-Based Core Decomposition

Definition 3.3 ([53]). A graph G is k -degenerate if every subgraph g of G has a vertex with degree at most k in g . The **degeneracy** of G , denoted $\delta(G)$, is the smallest value of k for which G is k -degenerate.

That is, the degeneracy $\delta(G)$ of G equals the maximum value among the minimum vertex degrees of all subgraphs of G .

Definition 3.4 ([20]). The **arboricity** of a graph G , denoted $\alpha(G)$, is the minimum number of edge-disjoint spanning forests into which G can be decomposed, or equivalently, the minimum number of forests needed to cover all edges of G .

Minimum Degree-Based Core Decomposition

Upper Bounds of $\alpha(G)$ and $\delta(G)$. For an arbitrary graph G , it is easy to verify that $\alpha(G) \leq d_{max}(G)$ and $\delta(G) \leq d_{max}(G)$. Moreover, they can also be bounded by n and m as follows.

Lemma 3.3 ([20]). $\alpha(G) \leq \left\lceil \frac{\sqrt{2m+n}}{2} \right\rceil$.

Lemma 3.4. $\delta(G) \leq \left\lceil \sqrt{2m+n} \right\rceil$.

Proof. This lemma directly follows from Lemmas 3.2 and 3.3. That is:

$$\delta(G) < 2\alpha(G) \leq 2 \times \left\lceil \frac{\sqrt{2m+n}}{2} \right\rceil \leq 2 \times \left(\frac{\left\lceil \sqrt{2m+n} \right\rceil + 1}{2} \right) = \left\lceil \sqrt{2m+n} \right\rceil + 1.$$

Thus, we have $\delta(G) \leq \left\lceil \sqrt{2m+n} \right\rceil$ since $\delta(G)$ is an integer. □



Minimum Degree-Based Core Decomposition

Lemma 3.1. *The degeneracy of G equals the maximum value among core numbers of vertices in G (i.e., $\delta(G) = \max_{u \in V} \text{core}(u)$).*

Proof. Firstly, it is easy to see that G contains no $(\delta(G) + 1)$ -core, since the minimum vertex degree of a $(\delta(G) + 1)$ -core is $\delta(G) + 1$; thus, $\max_{u \in V} \text{core}(u) \leq \delta(G)$. Secondly, G must contain a $\delta(G)$ -core, since it has at least one subgraph whose minimum vertex degree is $\delta(G)$; thus, $\max_{u \in V} \text{core}(u) \geq \delta(G)$. The lemma follows. \square

Lemma 3.2. $\alpha(G) \leq \delta(G) < 2\alpha(G)$.

The peeling Algorithm

Algorithm 1: Peel: compute core numbers of all vertices [7, 59]**Input:** A graph $G = (V, E)$ **Output:** $\text{core}(v)$ for each vertex $v \in V$

```
1 for each  $v \in V$  do
2    $\lfloor$  Let  $d(v)$  be the degree of  $v$  in  $G$ ;
3    $\text{max\_core} \leftarrow 0$ ;  $\text{seq} \leftarrow \emptyset$ ;
4   for  $i \leftarrow 1$  to  $n$  do
5      $u \leftarrow \operatorname{argmin}_{v \in V \setminus \text{seq}} d(v)$ ;
6     Add  $u$  to the tail of  $\text{seq}$ ;
7     if  $d(u) > \text{max\_core}$  then  $\text{max\_core} \leftarrow d(u)$ ;
8      $\text{core}(u) \leftarrow \text{max\_core}$ ;
9     for each neighbor  $v$  of  $u$  that is not in  $\text{seq}$  do
10     $\lfloor d(v) \leftarrow d(v) - 1$ ;
```

Definition 3.5 ([59]). Given a graph G , a permutation (v_1, v_2, \dots, v_n) of all vertices of G is a **degeneracy ordering** of G if every vertex v_i has the minimum degree in the subgraph of G induced by $\{v_i, \dots, v_n\}$.

The peeling Algorithm

Lemma 3.5. *If the input graph G is oriented according to the degeneracy ordering (i.e., an undirected edge (u, v) is directed from u to v if u appears before v in seq), then the maximum out-degree in the resulting graph is bounded by $\delta(G)$.*

Proof. This lemma directly follows from the definition of degeneracy and the definition of degeneracy ordering. \square

Lemma 3.6. *For any given k such that $1 \leq k \leq \delta(G)$, the k -core of G is the subgraph of G induced by vertices in $\text{seq}[p_k, \dots, n - 1]$, where p_k is the position/index of the first vertex in the array seq whose core number is at least k .*

Proof. Let g be the subgraph of G induced by vertices in $\text{seq}[p_k, \dots, n - 1]$. Firstly, the k -core of G contains none of the vertices in $\text{seq}[0, p_k - 1]$, since their core numbers are all smaller than k . Secondly, the minimum degree of g is at least k , according to the definition of core number and the definition of degeneracy ordering. Thus, g is the k -core of G , and the lemma holds. \square

The peeling Algorithm

The space complexity of Algorithm 1 is $2m + \mathcal{O}(n)$, where the graph representation takes space $2m + \mathcal{O}(n)$ and the linear heap data structure takes space $\mathcal{O}(n)$.

| Graph G | Peel+ListLinearHeap | Peel+ArrayLinearHeap |
|------------------|---------------------|----------------------|
| as-Skitter | 0.522 | 0.688 |
| soc-LiveJournal1 | 5.107 | 5.615 |
| uk-2005 | 14.281 | 27.229 |
| it-2004 | 16.507 | 31.840 |
| twitter-2010 | 180 | 175 |

Compute k-Core

Algorithm 2: k-core: compute k-core

Input: A graph $G = (V, E)$ and an integer k

Output: k -core of G

```
1 Initialize an empty queue  $Q$ ;  
2 for each  $v \in V$  do  
3   Let  $d(v)$  be the degree of  $v$  in  $G$ ;  
4   if  $d(v) < k$  then Push  $v$  to  $Q$ ;  
5 while  $Q \neq \emptyset$  do  
6    $u \leftarrow$  pop a vertex from  $Q$ ;  
7   /* Remove  $u$  from the graph */  
8   for each neighbor  $v$  of  $u$  do  
9      $d(v) \leftarrow d(v) - 1$ ;  
10    if  $d(v) = k - 1$  then Push  $v$  to  $Q$ ;  
11 return the subgraph of  $G$  induced by vertices with  $d(\cdot) \geq k$ ;
```

The time complexity of Algorithm 2 is $\mathcal{O}(m)$ since each vertex of V will be popped out from Q at most once, and the space complexity of Algorithm 2 is $2m + \mathcal{O}(n)$.



Compute k-Core

Definition 3.6. Given a graph G , a *connected k-core* of G is a connected component of the k -core of G ; that is, a maximal *connected* subgraph g of G such that the minimum degree of g is at least k .

UnionFind

Listing 3.1: Interface of a disjoint-set data structure

```
class UnionFind {  
private:  
    uint n; // total number of elements  
    uint *parent; // parents of elements  
    uint *rank; // ranks of elements  
  
public:  
    UnionFind(uint _n) ;  
    ~UnionFind() ;  
    void init(uint _n) ; // allocate memory, and initialize the  
    // arrays: parent and rank  
    ui UF_find(ui i) ; // return the representative of the set  
    // containing i  
    void UF_union(ui i, ui j) ; // union the two sets that contain  
    // i and j, respectively  
}
```

- `init(_n)` initializes the data structure such that each element i forms a singleton set; specifically, $\text{parent}[i] = i$ and $\text{rank}[i] = 0$.
- `UF_find(i)` returns the representative of the unique set containing i .



- $\text{UF_union}(i, j)$ unites the two sets that contain i and j , respectively, into a single set that is the union of these two sets. Specifically, the parent of the representative of one set is set as the representative of the other set.

Theorem 3.1 ([24]). *By using both union by rank and path compression, the worst-case running time of a sequence of x UF_find and UF_union operations is $\mathcal{O}(x \times a(x))$, where $a(x)$ is the inverse of the Ackermann function and is at most 4 for all practical values of x .*

In the remainder of the book, we assume that $a(x)$ is constant and ignore it in the time complexity analysis. The space complexity of the disjoint-set data structure is $2n + \mathcal{O}(1)$ since each of the arrays parent and rank has size n .

Core Hierarchy Tree

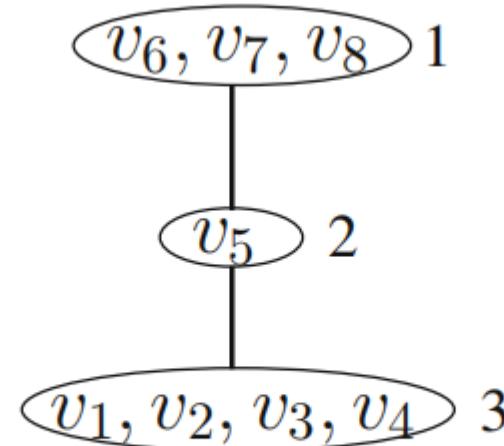


Fig. 3.3: The core hierarchy tree for the graph in Figure 3.1

Based on the disjoint-set data structure, the time complexity of CoreHierarchy is $\mathcal{O}(m)$, and the space complexity of CoreHierarchy is $2m + \mathcal{O}(n)$ where the size of CoreHT is $\mathcal{O}(n)$.

Core Hierarchy Tree

Algorithm 3: CoreHierarchy: compute the core hierarchy tree of a graph

Input: A graph $G = (V, E)$, a degeneracy ordering seq of vertices, and the core numbers $\text{core}(\cdot)$ of vertices

Output: A core hierarchy tree CoreHT of G

```
1 Initialize an empty CoreHT, and a disjoint-set data structure  $\mathcal{F}$  for  $V$ ;  
2 for each vertex  $u \in V$  do  
3   Add a node  $r_u$ , with weight  $\text{core}(u)$  and containing vertex  $u$ , to CoreHT;  
4   Point  $u$  to  $r_u$ ;  
5 for each vertex  $u$  in  $\text{seq}$  in reverse order do  
6   for each neighbor  $v$  of  $u$  in  $G$  that appear later than  $u$  in  $\text{seq}$  do  
7     Let  $r_v$  and  $r_u$  be the nodes of CoreHT pointed by the representatives of the sets  
    containing  $v$  and  $u$  in  $\mathcal{F}$ , respectively;  
8     if  $r_v \neq r_u$  then  
9       /* Update the CoreHT */  
10      if the weight of  $r_v$  equals the weight of  $r_u$  then  
11        /* Move the content (i.e., vertices and children) of  $r_v$  to  $r_u$  */  
12        Move the content (i.e., vertices and children) of  $r_v$  to  $r_u$ ;  
13      else Assign  $r_u$  as the parent of  $r_v$  in the CoreHT;  
14      /* Update the disjoint-set data structure  $\mathcal{F}$  */  
15      Union  $u$  and  $v$  in  $\mathcal{F}$ , and point the updated representative of the set containing  
     $u$  to  $r_u$ ;  
16  
17 return CoreHT;
```

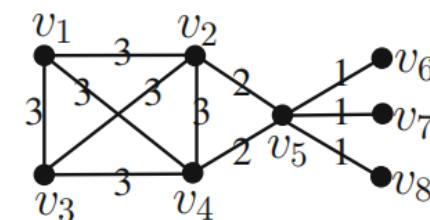
Core Spanning Tree

$W(u,v)$ of edge (u,v) is the largest k for which there is a k -core of G containing the edge;

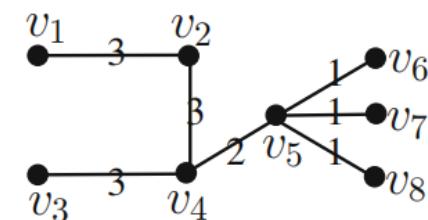
$$W(u,v) = \min\{\text{core}(u), \text{core}(v)\}$$

Lemma 3.7. *The k -core of G is the subgraph of G_w induced by the edges with weight at least k .*

Proof. The k -core of G is the subgraph of G induced by those vertices whose core numbers are at least k . It is also exactly the subgraph of G_w induced by those edges whose weights are at least k , according to the definition of edge weights. \square



(a) An edge-weighted graph



(b) Maximum spanning tree

Fig. 3.4: A core spanning tree for the graph in Figure 3.1

Core Spanning Tree

Algorithm 4: CoreSpanning: compute a core spanning tree of a graph

Input: A graph $G = (V, E)$, a degeneracy ordering seq of vertices, and core numbers $\text{core}(\cdot)$ of vertices

Output: A core spanning tree CoreSPT of G

```
1 CoreSPT  $\leftarrow (V, \emptyset)$ ; /* Initialize CoreSPT to be a graph consisting of  
   vertices  $V$  and no edges */;  
2 Initialize a disjoint-set data structure  $\mathcal{F}$  for  $V$ ;  
3 for each vertex  $u$  in  $\text{seq}$  in reverse order do  
4   for each neighbor  $v$  of  $u$  in  $G$  that appear after  $u$  in  $\text{seq}$  do  
5     if  $v$  and  $u$  are not in the same set in  $\mathcal{F}$  then  
6       Union  $u$  and  $v$  in  $\mathcal{F}$ ;  
7       Add edge  $(u, v)$  with weight  $\text{core}(u)$  into CoreSPT;  
8 return CoreSPT;
```

Time Complexity: $O(m)$ Space Complexity: $2m+O(n)$

Core Decomposition in Other Environments

An h-index-Based Local Algorithm

H-index:

Definition 3.7. Given a multi-set S of positive numbers, the **h-index** of S , denoted $\text{h-index}(S)$, is the largest integer k such that there are at least k numbers in S that are no smaller than k ; that is, the largest integer k such that $|\{s \in S \mid s \geq k\}| \geq k$.

Lemma 3.8 ([54]). Given a graph G , let $C(u)$ be the multi-set of core numbers of u 's neighbors (i.e., $C(u) = \{\text{core}(v) \mid v \in N(u)\}$). Then, the h-index of $C(u)$ equals the core number of u in G , that is:

$$\text{core}(u) = \text{h-index}(C(u)).$$

Proof. Firstly, we prove that $\text{core}(u) \geq \text{h-index}(C(u))$. Let k be the value of $\text{h-index}(C(u))$. We consider the k -core g of G . According to the definition of h-index, at least k neighbors of u are in g . Thus, u is also in g , and $\text{core}(u) \geq k = \text{h-index}(C(u))$.

Secondly, we prove that $\text{core}(u) \leq \text{h-index}(C(u))$. Let k be the value of $\text{core}(u)$. Consider the k -core g of G . Then, at least k neighbors of u are in g , each of which has a core number at least k . Thus, $\text{h-index}(C(u)) \geq k = \text{core}(u)$. Thus, the lemma holds. \square

Core Decomposition in Other Environments

Lemma 3.9 ([54]). Given a graph G , let $\bar{C}(u)$ be the multi-set of upper bounds of core numbers of u 's neighbors (i.e., $\bar{C}(u) = \{\overline{\text{core}}(v) \mid v \in N(u)\}$). Then, the h-index of $\bar{C}(u)$ is an upper bound of u 's core number, that is:

$$\text{core}(u) \leq \text{h-index}(\bar{C}(u)).$$

Lemma 3.10. Given the upper bounds $\overline{\text{core}}$ of the core numbers of vertices in G , if $\text{h-index}(\bar{C}(v)) = \overline{\text{core}}(v)$ for every vertex v in G , then the upper bounds $\overline{\text{core}}$ are the core numbers.

Proof. For a vertex u , let k be the value of $\overline{\text{core}}(u)$ and consider the subgraph g of G induced by the set of vertices whose upper bounds of core numbers are at least k . As $\text{h-index}(\bar{C}(v)) = \overline{\text{core}}(v)$ holds for every vertex v in G , each vertex in g has at least k neighbors in g . Thus, the core number of u is at least k . Moreover, k is an upper bound of the core number of u due to Lemma 3.9. Consequently, $\text{core}(u) = k = \overline{\text{core}}(u)$ and the lemma holds. \square

Core Decomposition in Other Environments

Algorithm 5: CoreD-Local: compute core numbers of vertices [54]**Input:** A graph $G = (V, E)$ **Output:** $\text{core}(v)$ of each vertex $v \in V$

```
1 for each vertex  $u \in V$  do  $\overline{\text{core}}(u) \leftarrow$  the degree of  $u$  in  $G$ ;  
2 update  $\leftarrow$  true;  
3 while update do  
4   update  $\leftarrow$  false;  
5   for each vertex  $u \in V$  do  
6      $old \leftarrow \overline{\text{core}}(u)$ ;  
7      $\overline{\text{core}}(u) \leftarrow \text{HIndex}(N(u), \overline{\text{core}})$ ;  
8     if  $\overline{\text{core}}(u) \neq old$  then update  $\leftarrow$  true;  
9 return  $\text{core}(v) \leftarrow \overline{\text{core}}(v)$  for each vertex  $v \in V$ ;
```

Procedure $\text{HIndex}(S, \overline{\text{core}})$

```
10 Initialize an array cnt of size  $|S| + 1$ , consisting of all zeros;  
11 for each vertex  $u \in S$  do  
12   if  $\overline{\text{core}}(u) > |S|$  then  $\text{cnt}[|S|] \leftarrow \text{cnt}[|S|] + 1$ ;  
13   else  $\text{cnt}[\overline{\text{core}}(u)] \leftarrow \text{cnt}[\overline{\text{core}}(u)] + 1$ ;  
14 for  $i \leftarrow |S|$  down to 1 do  
15   if  $\text{cnt}[i] \geq i$  then return  $i$ ;  
16    $\text{cnt}[i - 1] \leftarrow \text{cnt}[i - 1] + \text{cnt}[i]$ ;
```

Core Decomposition in Other Environments

Lemma 3.11. $\overline{\text{core}}(u) = \text{h-index}(\overline{C}(u))$ if and only if $\overline{\text{core}}(u) \leq \text{cnt}(u)$.

Algorithm 6: CoreD-Local-opt: compute core numbers of vertices

Input: A graph $G = (V, E)$
Output: $\text{core}(v)$ for each vertex $v \in V$

```

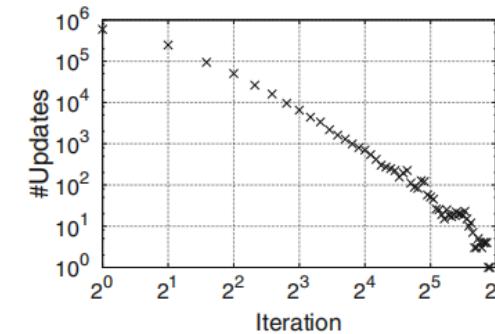
1  $Q \leftarrow \emptyset;$ 
2 for each vertex  $u \in V$  do
3    $\overline{\text{core}}(u) \leftarrow$  the degree of  $u$  in  $G$ ;
4    $\text{cnt}(u) \leftarrow$  the number of neighbors of  $u$  whose degrees are no smaller than that of  $u$ ;
5   if  $\text{cnt}(u) < \overline{\text{core}}(u)$  then Push  $u$  into  $Q$ ;
6 while  $Q \neq \emptyset$  do
7    $Q' \leftarrow \emptyset;$ 
8   while  $Q \neq \emptyset$  do
9     Pop a vertex  $u$  from  $Q$ ;
10     $old \leftarrow \overline{\text{core}}(u);$ 
11     $\overline{\text{core}}(u) \leftarrow \text{HIndex}(N(u), \overline{\text{core}});$ 
12     $\text{cnt}(u) \leftarrow 0;$ 
13    for each neighbor  $v$  of  $u$  do
14      if  $\overline{\text{core}}(v) \geq \overline{\text{core}}(u)$  then  $\text{cnt}(u) \leftarrow \text{cnt}(u) + 1;$ 
15      if  $v \notin Q$  and  $\overline{\text{core}}(u) < \overline{\text{core}}(v) \leq old$  then
16        if  $\text{cnt}(v) = \overline{\text{core}}(v)$  then Push  $v$  into  $Q'$ ;
17         $\text{cnt}(v) \leftarrow \text{cnt}(v) - 1;$ 
18    $Q \leftarrow Q';$ 
19 return  $\text{core}(v) \leftarrow \overline{\text{core}}(v)$  for each vertex  $v \in V;$ 

```

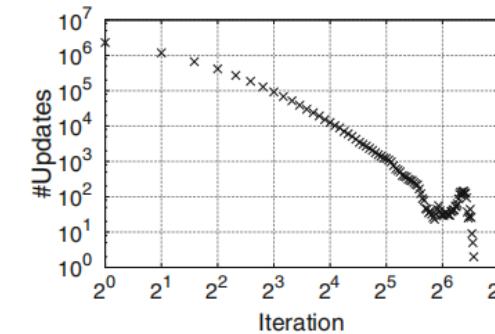
Theorem 3.2. The time complexity of Algorithm 6 is $\mathcal{O}(m \times \text{h-index}(G))$. Here, $\text{h-index}(G)$ is the largest value computed by $\text{HIndex}(N(u), d)$ among all vertices in G , where $d(\cdot)$ denotes the degrees of vertices.

Core Decomposition in Other Environments

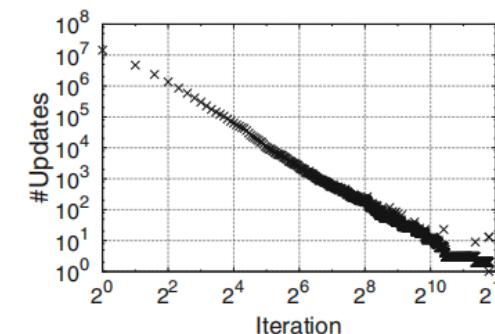
| Graph G | Peel | CoreD-Local-opt |
|------------------|--------|-----------------|
| as-Skitter | 0.550 | 0.645 |
| soc-LiveJournal1 | 4.232 | 7.765 |
| uk-2005 | 26.338 | 17.535 |
| it-2004 | 28.647 | 24.810 |
| twitter-2010 | 134 | 369 |



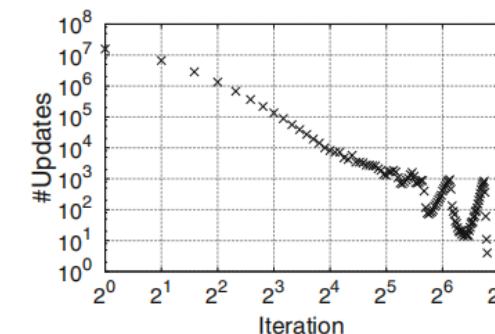
(a) as-Skitter



(b) soc-LiveJournal1



(c) it-2004



(d) twitter-2010

Fig. 3.5: Number of updates in different iterations

Core Decomposition in Other Environments

Algorithm 7: CoreD-IO: I/O efficiently computed core numbers of vertices

```
Input: A graph  $G = (V, E)$ 
Output:  $\text{core}(v)$  for each vertex  $v \in V$ 

1  $v_{\min} \leftarrow v_n; v_{\max} \leftarrow v_1;$ 
2 for vertex  $u \leftarrow v_1$  to  $v_n$  do
3    $\overline{\text{core}}(u) \leftarrow$  the degree of  $u$  in  $G$ ;
4   Load  $N(u)$  from disk;
5    $\text{cnt}(u) \leftarrow$  the number of vertices in  $N(u)$  whose degrees are no smaller than that of  $u$ ;
6   if  $\text{cnt}(u) < \overline{\text{core}}(u)$  then
7     if  $u < v_{\min}$  then  $v_{\min} \leftarrow u$ ;
8      $v_{\max} \leftarrow u$ ;
9 while  $v_{\min} \leq v_{\max}$  do
10   $v'_{\min} \leftarrow v_n; v'_{\max} \leftarrow v_1$ ;
11  for vertex  $u \leftarrow v_{\min}$  to  $v_{\max}$  s.t.  $\overline{\text{core}}(u) > \text{cnt}(u)$  do
12    Load  $N(u)$  from disk;
13     $old \leftarrow \overline{\text{core}}(u)$ ;
14     $\overline{\text{core}}(u) \leftarrow \text{HIndex}(N(u), \overline{\text{core}})$ ;
15     $\text{cnt}(u) \leftarrow 0$ ;
16    for each vertex  $v \in N(u)$  do
17      if  $\overline{\text{core}}(v) \geq \overline{\text{core}}(u)$  then  $\text{cnt}(u) \leftarrow \text{cnt}(u) + 1$ ;
18      if  $\overline{\text{core}}(u) < \overline{\text{core}}(v) \leq old$  then
19        if  $\text{cnt}(v) = \overline{\text{core}}(v)$  and  $not u < v \leq v_{\max}$  then
20          if  $v < v'_{\min}$  then  $v'_{\min} \leftarrow v$ ;
21          if  $v > v'_{\max}$  then  $v'_{\max} \leftarrow v$ ;
22         $\text{cnt}(v) \leftarrow \text{cnt}(v) - 1$ ;
23   $v_{\min} \leftarrow v'_{\min}; v_{\max} \leftarrow v'_{\max}$ ;
24 return  $\text{core}(v) \leftarrow \overline{\text{core}}(v)$  for each vertex  $v \in V$ ;
```

不用存全部的图，从磁盘中读即可

用 v_{\min} 和 v_{\max} 代替队列，这里是因为节点默认从0-n-1来进行编号，不是这个编号也可以做对应的映射。

Preliminaries

Definition 4.1. The *edge density* of a graph g , denoted $\rho(g)$, is

$$\rho(g) = \frac{|E(g)|}{|V(g)|}.$$

Lemma 4.1. The edge density is half of the average degree, i.e., $\rho(g) = \frac{d_{avg}(g)}{2}$.

Proof. The average degree of g is $d_{avg}(g) = \frac{\sum_{v \in V(g)} d_g(v)}{|V(g)|} = \frac{2 \times |E(g)|}{|V(g)|} = 2 \times \rho(g)$. \square

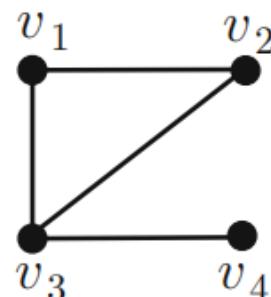


Fig. 4.1: An example graph

不用存全部的图，从磁盘中读即可

用vmin和vmax代替队列，这里是因为节点默认从0-n-1来进行编号，不是这个编号也可以做对应的映射。

Properties

Lemma 4.3. $\frac{\delta(G)}{2} \leq \rho^* \leq \delta(G)$ holds for every graph G ; recall that $\delta(G)$ is the degeneracy of G .

Proof. Firstly, recall that any graph G has a non-empty $\delta(G)$ -core, in which each vertex has at least $\delta(G)$ neighbors within the subgraph. Then, the average degree of the $\delta(G)$ -core is at least $\delta(G)$, and thus the density of the $\delta(G)$ -core is at least $\frac{\delta(G)}{2}$. Consequently, the density of the densest subgraph is at least $\frac{\delta(G)}{2}$; that is, $\rho^* \geq \frac{\delta(G)}{2}$.

Secondly, following Lemma 4.2, ρ^* must be no larger than $d_{\min}(S^*)$, which in turn is no larger than $\delta(G)$ by following the definition of degeneracy. Consequently, $\rho^* \leq \delta(G)$. \square

Lemma 4.2. The minimum degree of S^* is no smaller than ρ^* , that is:

$$d_{\min}(S^*) = \min_{u \in S^*} d_{S^*}(u) \geq \rho^* = \rho(S^*)$$

where $d_{S^*}(u)$ is the degree of u in the subgraph $G[S^*]$.

Proof. This lemma can be proved by contradiction. Suppose that there is a vertex $u \in S^*$ with degree $d_{S^*}(u) < \rho^* = \frac{|E(S^*)|}{|S^*|}$. Let S be $S^* \setminus \{u\}$. Then, the density of S is

$$\rho(S) = \frac{|E(S)|}{|S|} = \frac{|E(S^*)| - d_{S^*}(u)}{|S^*| - 1} > \frac{|E(S^*)| - \frac{|E(S^*)|}{|S^*|}}{|S^*| - 1} = \frac{|E(S^*)|}{|S^*|} = \rho(S^*),$$

which contradicts that S^* is the densest subgraph of G . Thus, the lemma holds. \square

An 2-Approximation Algorithm

Definition 4.3. For the problem of densest subgraph computation, an algorithm is a **θ -approximation algorithm** if for every graph G , it outputs a subgraph S such that

$$\rho(S) \geq \frac{\rho(S^*)}{\theta},$$

where S^* is the densest subgraph of G .

Theorem 4.1. *The $\delta(G)$ -core of a graph G is a 2-approximation of the densest subgraph of G .*

An 2-Approximation Algorithm

Algorithm 1: Densest-Greedy: a 2-approximation algorithm for densest subgraph [18]

Time Complexity: $O(m)$ **Input:** A graph $G = (V, E)$ **Output:** A dense subgraph \tilde{S}

- 1 Compute the degeneracy ordering of G by invoking Peel (Algorithm 1 in Chapter 3);
 - 2 $\tilde{S} \leftarrow \emptyset$;
 - 3 $S \leftarrow V$; $m_S \leftarrow |E|$;
 - 4 **for each** vertex u of G according to the degeneracy ordering **do**
 - 5 $\rho(S) \leftarrow m_S/|S|$;
 - 6 **if** $\tilde{S} = \emptyset$ **or** $\rho(S) > \rho(\tilde{S})$ **then**
 - 7 $\tilde{S} \leftarrow S$;
 - 8 $S \leftarrow S \setminus \{u\}$;
 - 9 **for each** neighbor v of u such that $v \in S$ **do**
 - 10 $m_S \leftarrow m_S - 1$;
 - 11 **return** \tilde{S} ;
-

An 2-Approximation Algorithm

Theorem 4.2 ([18]). Densest-Greedy (Algorithm 1) is a 2-approximation algorithm.

Proof. Let v be the smallest vertex of S^* according to the degeneracy ordering, and let S_v be the suffix of the degeneracy ordering starting at v . Then, it holds that

$$S^* \subseteq S_v$$

and

$$d_{avg}(S_v) \geq d_{min}(S_v) = d_{S_v}(v) \geq d_{S^*}(v) \geq \rho(S^*)$$

where the equality follows from the definition of degeneracy ordering (i.e., $d_{S_v}(v) = d_{min}(S_v)$), the second inequality follows from the fact that $S^* \subseteq S_v$, and the last inequality follows from Lemma 4.2. Following the nature of the algorithm (specifically Lines 6–7), it holds that

$$\rho(\tilde{S}) \geq \rho(S_v) = \frac{d_{avg}(S_v)}{2} \geq \frac{\rho(S^*)}{2}$$

A Steaming $2(1+\varepsilon)$ -Approximation Algorithm

Algorithm 2: Densest-Streaming: a streaming and approximation algorithm for densest subgraph [5]

Input: A graph $G = (V, E)$, a parameter $\varepsilon > 0$

Output: A dense subgraph \tilde{S}

```
1  $\tilde{S} \leftarrow \emptyset$ ;
2  $S \leftarrow V$ ;  $m_S \leftarrow |E|$ ;
3 for each vertex  $u \in V$  do  $d_S(u) \leftarrow$  the degree of  $u$  in  $G$ ;
4 while  $S \neq \emptyset$  do
5    $\rho(S) \leftarrow m_S/|S|$ ;
6   if  $\tilde{S} = \emptyset$  or  $\rho(S) > \rho(\tilde{S})$  then
7      $\tilde{S} \leftarrow S$ ;
8    $\Delta(S) \leftarrow \{v \in S \mid d_S(v) \leq 2 \times (1 + \varepsilon) \times \rho(S)\}$ ;
9   for each  $v \in \Delta(S)$  do
10     $S \leftarrow S \setminus \{v\}$ ;
11    for each neighbor  $u$  of  $v$  in  $S$  do
12       $d_S(u) \leftarrow d_S(u) - 1$ ;
13       $m_S \leftarrow m_S - 1$ ;
14 return  $\tilde{S}$ ;
```

A Steaming $2(1+\varepsilon)$ -Approximation Algorithm

Lemma 4.4 ([5]). Densest-Streaming (i.e., Algorithm 2) terminates in $\mathcal{O}(\log_{1+\varepsilon} n)$ iterations.

Proof. In each iteration, a set $\Delta(S) = \{v \in S \mid d_S(v) \leq 2 \times (1 + \varepsilon) \times \rho(S)\}$ of vertices are removed from S . To bound the number of iterations, the main problem is to bound either $|\Delta(S)|$ or $|S \setminus \Delta(S)|$. It is easy to see that $\Delta(S)$ is not empty, since there is at least one vertex in S whose degree is no greater than the average degree of all vertices in S . For each vertex v that is not removed (i.e., $v \in S \setminus \Delta(S)$), it holds that $d_S(v) > 2 \times (1 + \varepsilon) \times \rho(S)$. Thus:

$$\begin{aligned} 2 \times |E(S)| &= \sum_{v \in S} d_S(v) \\ &= \sum_{v \in \Delta(S)} d_S(v) + \sum_{v \in S \setminus \Delta(S)} d_S(v) \\ &> \sum_{v \in S \setminus \Delta(S)} d_S(v) \\ &> \sum_{v \in S \setminus \Delta(S)} 2 \times (1 + \varepsilon) \times \rho(S) \\ &= |S \setminus \Delta(S)| \times 2 \times (1 + \varepsilon) \times \frac{|E(S)|}{|S|}. \end{aligned}$$

This results in the following inequality:

$$|S \setminus \Delta(S)| < \frac{1}{1 + \varepsilon} |S|.$$

Theorem 4.3 ([5]). The I/O complexity of Densest-Streaming (i.e., Algorithm 2) is $\mathcal{O}\left(\frac{m \times \log_{1+\varepsilon} n}{B}\right)$, where B is the disk block size.

Theorem 4.4 ([5]). Densest-Streaming (Algorithm 2) is a $2(1 + \varepsilon)$ -approximation algorithm.

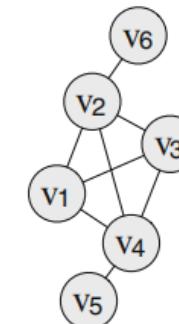
Density Testing

- $V(G_x) = V \cup \{s, t\}$, where $s \notin V$ and $t \notin V$.
 - $E(G_x) = E \cup \{(s, v) \mid v \in V\} \cup \{(u, t) \mid u \in V\}$.
 - $w(u, v) = 1$, for $(u, v) \in E$.
 - $w(s, v) = m$, for $v \in V$.
 - $w(u, t) = m + 2x - d(u)$, for $u \in V$.

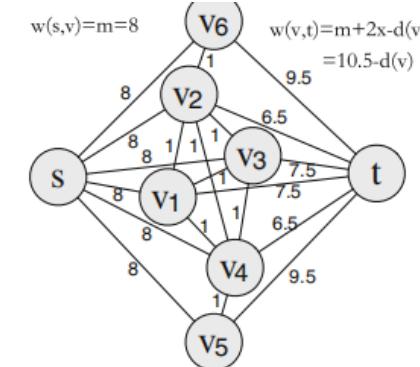
$$\omega(A, B) = \sum_{(u,v) \in E(G_x) \text{ s.t. } u \in A, v \in B} w(u, v).$$

Theorem 4.5 ([35]). Assume that $(\{s\} \cup S, \{t\} \cup T)$ is a minimum $s-t$ cut in the augmented graph G_x .

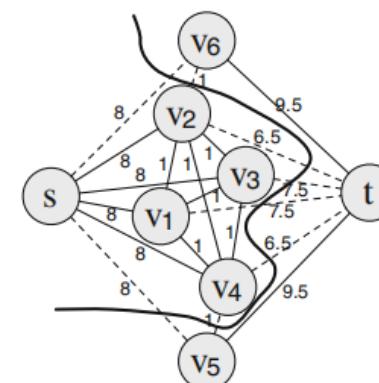
- If $S \neq \emptyset$, then $\rho^* \geq \rho(S) \geq x$.
 - If $S = \emptyset$, then $\rho^* \leq x$.



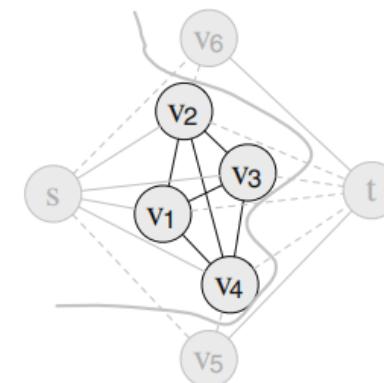
(a) A Graph G with $n=6$ and $m=$



(b) The Constructed Graph G_x for $x=1.25$



(c) The Minimum s-t Cut for G.



(d) The Resulting Subgraph

Fig. 4.2: An example to illustrate Density Test

Density Testing

Algorithm 3: DensityTest: test whether $\rho^* \geq x$ [35]**Input:** A graph $G = (V, E)$, a real number $x \geq 0$ **Output:** A vertex subset S , such that if $S \neq \emptyset$ then $\rho^* \geq \rho(S) \geq x$ and otherwise $\rho^* \leq x$

- 1 $G_x \leftarrow G$;
 - 2 Assign a weight 1 to every edge of G_x ;
 - 3 Add a source vertex $s \notin V$ and a sink vertex $t \notin V$ to G_x ;
 - 4 **for each** vertex $u \in V$ **do**
 - 5 Add edge (s, u) with weight m to G_x ;
 - 6 Add edge (u, t) with weight $m + 2x - d(u)$ to G_x ;
 - 7 Compute a minimum $s-t$ cut in G_x and denote it by (A, B) ;
 - 8 **return** $A \setminus \{s\}$;
-

Lemma 4.5 ([35]). Given two subgraphs S_1 and S_2 , if $\rho(S_1) > \rho(S_2)$, then it holds that $\rho(S_1) - \rho(S_2) \geq \frac{1}{n(n-1)}$.

Densest-Exact

Algorithm 4: Densest-Exact: an exact algorithm for densest subgraph [35]**Input:** A graph $G = (V, E)$ **Output:** A densest subgraph of G

```
1  $\tilde{S} \leftarrow \emptyset;$ 
2  $l \leftarrow 0; h \leftarrow m;$ 
3 while  $h - l \geq \frac{1}{n(n-1)}$  do
4    $x \leftarrow \frac{l+h}{2};$ 
5    $S \leftarrow \text{DensityTest}(G, x);$ 
6   if  $S \neq \emptyset$  then
7      $\tilde{S} \leftarrow S;$ 
8      $l \leftarrow x;$ 
9   else
10     $h \leftarrow x;$ 
11 return  $\tilde{S};$ 
```



High-Order Core Decomposition

Springer Series in the Data Sciences 2018

Lijun Chang • Lu Qin

K3 Algorithm

Algorithm 1: K3: enumerate all triangles of a graph [20]

Input: An undirected graph $G = (V, E)$

Output: All triangles in G

```

1 Sort vertices of  $G$  such that  $d(v_1) \geq \dots \geq d(v_n)$ ;
2 for each vertex  $u \leftarrow v_1, \dots, v_n$  do
3   for each neighbor  $v \in N(u)$  do Mark  $v$ ;
4   for each neighbor  $v \in N(u)$  do
5     for each neighbor  $w \in N(v)$  do
6       if  $w$  is marked and  $v < w$  then
7         Output triangle  $\triangle_{u,v,w}$ ;
8   for each neighbor  $v \in N(u)$  do Unmark  $v$ ;
9   Remove  $u$  from  $G$ ;

```

Lemma 5.1 ([20]). Given a graph $G = (V, E)$, it holds that

$$\sum_{(u,v) \in E} \min\{d(u), d(v)\} \leq 2 \times \alpha(G) \times m,$$

Corollary 5.1. The number of triangles in G is at most $\frac{2}{3} \times \alpha(G) \times m$.

TriE Algorithm

Algorithm 2: TriE: enumerate all triangles in a graph [64]**Input:** An undirected graph $G = (V, E)$, and a total order \prec of V **Output:** All triangles in G

```
1 Construct the oriented graph  $G^+ = (V, E^+)$  of  $G$  with respect to  $\prec$ ;  
2 for each vertex  $u \in V$  do  
3   for each out-neighbor  $v \in N^+(u)$  do Mark  $v$ ;  
4   for each out-neighbor  $v \in N^+(u)$  do  
5     for each out-neighbor  $w \in N^+(v)$  do  
6       if  $w$  is marked then  
7         Output triangle  $\triangle_{u,v,w}$ ;  
8   for each out-neighbor  $v \in N^+(u)$  do Unmark  $v$ ;
```

Total Orders. Three popularly used total orders are as follows:

1. *Degree decreasing ordering.* For any two vertices $u, v \in V$, $u \prec v$ (i.e., u ranks higher than v) if:
 - $d(u) > d(v)$, or
 - $d(u) = d(v)$ and u has a larger vertex ID than v .
2. *Degree increasing ordering.* For any two vertices $u, v \in V$, $u \prec v$ (i.e., u ranks higher than v) if:
 - $d(u) < d(v)$, or
 - $d(u) = d(v)$ and u has a smaller vertex ID than v .
3. *Smallest-first ordering* (aka, degeneracy ordering). For any two vertices $u, v \in V$, $u \prec v$ (i.e., u ranks higher than v) if u is before v in the degeneracy ordering of V (see Chapter 3 for the definition of degeneracy ordering).

有向图，且内部节点存在关系

K-clique Enumeration Algorithm

Algorithm 3: KClIQUE-ChibaN: enumerate all k -cliques in a graph [20]**Input:** An undirected graph $G = (V, E)$, and an integer k **Output:** All k -cliques in G

```
1  $C \leftarrow \emptyset;$ 
2 KClIQUE-Enum( $G, k, C$ );
Procedure KClIQUE-Enum( $G_k, k, C$ )
3 if  $k = 2$  then
4   for each edge  $(u, v) \in E(G_k)$  do
5      $\quad \text{Output clique } \{u, v\} \cup C;$ 
6 else
7   Sort vertices of  $G_k$  such that  $d_{G_k}(v_1) \geq \dots \geq d_{G_k}(v_{|V(G_k)|})$ ;
8   for each vertex  $u \leftarrow v_1, \dots, v_{|V(G_k)|}$  do
9      $\quad G_{k-1} \leftarrow \text{the subgraph of } G_k \text{ induced by } N_{G_k}(u);$ 
10    KClIQUE-Enum( $G_{k-1}, k - 1, C \cup \{u\}$ );
11    Remove  $u$  from  $G_k$ ;
```

Theorem 5.1 ([20]). KClIQUE-ChibaN enumerates all k -cliques in a graph G in $\mathcal{O}(k \times (\alpha(G))^{k-2} \times m)$ time and $\mathcal{O}(n + m)$ space.

K-clique Enumeration Algorithm

Algorithm 4: KClque-Oriented: enumerate all k -cliques in a graph [25]**Input:** An undirected graph $G = (V, E)$, and an integer k **Output:** All k -cliques in G

- 1 Compute the degeneracy ordering of V ;
- 2 Construct the oriented graph $G^+ = (V, E^+)$ of G with respect to the degeneracy ordering;
- 3 $C \leftarrow \emptyset$;
- 4 KClque-EnumO(G^+, k, C);

Procedure KClque-EnumO(G_k^+, k, C)

- 5 **if** $k = 2$ **then**
 - 6 **for each** edge $(u, v) \in E(G_k^+)$ **do**
 - 7 **Output clique** $\{u, v\} \cup C$;
 - 8 **else**
 - 9 **for each** vertex $u \in V(G_k^+)$ **do**
 - 10 $G_{k-1}^+ \leftarrow$ the subgraph of G_k^+ induced by $N_{G_k^+}^+(u)$;
 - 11 KClque-EnumO($G_{k-1}^+, k - 1, C \cup \{u\}$);
-

k-truss

Definition 5.1. Given a graph g , the *support of an edge* $(u, v) \in E(g)$, denoted $\text{supp}_g(u, v)$, is the number of triangles in g that contain the edge (u, v) ; that is:

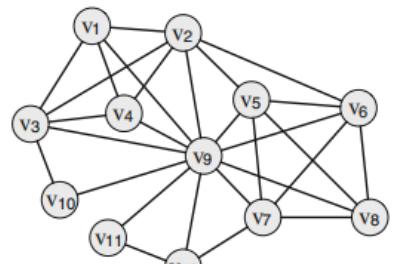
$$\text{supp}_g(u, v) = |\{w \in V(g) \mid \triangle_{u,v,w} \in g\}|,$$

Definition 5.2. Given a graph G and an integer $k \geq 2$, the *k -truss* of G is the maximal subgraph g of G such that for any edge $(u, v) \in E(g)$, the support of (u, v) in g is at least k (i.e., $\text{supp}_g(u, v) \geq k$).

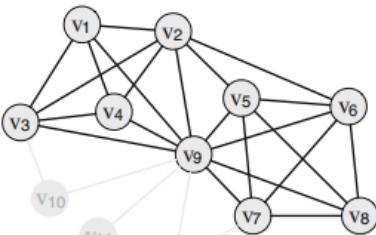
k-truss

Properties of k-Truss. k -truss has the following properties as proved in [22]:

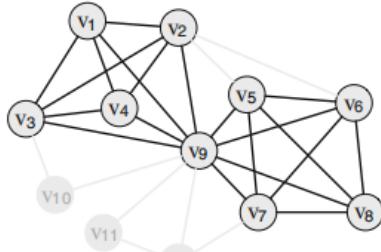
- Each k -truss of a graph G is a subgraph of the $(k-1)$ -truss of G ; for example, in Figure 5.2, the 3-truss is a subgraph of the 2-truss which in turn is a subgraph of the 1-truss.



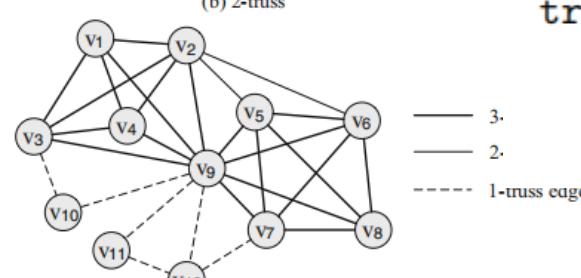
(a) 1-truss



(b) 2-truss



(c) 3-truss



— 3.
— 2.
- - - 1-truss edge

Definition 5.3. Given a graph G , the **truss number** of an edge (u, v) in G , denoted $\text{truss}(u, v)$, is the largest k such that the k -truss of G contains (u, v) , that is:

$$\text{truss}(u, v) = \max\{k \mid \text{the } k\text{-truss of } G \text{ contains } (u, v)\}.$$

Fig. 5.2: Truss decomposition

k-truss

Algorithm 5: PeelTruss: compute truss numbers of all edges [92]**Input:** A graph $G = (V, E)$ **Output:** $\text{truss}(u, v)$ for each edge $(u, v) \in E$

```
1 for each edge  $(u, v) \in E$  do
2   Compute the number  $t(u, v)$  of triangles in  $G$  containing  $(u, v)$ ;
3    $\text{supp}(u, v) \leftarrow t(u, v);$ 
4  $\max\_truss \leftarrow 0; \text{seq} \leftarrow \emptyset;$ 
5 for  $i \leftarrow 1$  to  $m$  do
6    $(u, v) \leftarrow \arg\min_{(u', v') \in E \setminus \text{seq}} \text{supp}(u', v');$ 
7   Add  $(u, v)$  to the tail of  $\text{seq}$ ;
8   if  $\text{supp}(u, v) > \max\_truss$  then  $\max\_truss \leftarrow \text{supp}(u, v);$ 
9    $\text{truss}(u, v) \leftarrow \max\_truss;$ 
10  for each triangle  $\Delta_{u,v,w}$  that contains edge  $(u, v)$  do
11     $\text{supp}(u, v) \leftarrow \text{supp}(u, v) - 1;$ 
12     $\text{supp}(u, w) \leftarrow \text{supp}(u, w) - 1;$ 
13  Remove edge  $(u, v)$  from the graph;
```

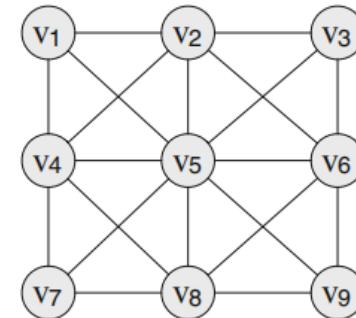
Nuclear-decomposition

Definition 5.4. Given a graph g , two positive integers $1 \leq r < s$, and an r -clique C^r in g , the s -clique support of C^r in g , denoted $\text{supp}_g^s(C^r)$, is the number of s -cliques in g containing C^r .

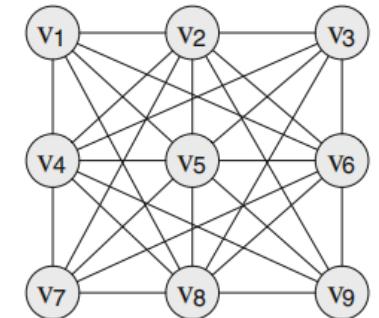
Definition 5.5. Given a graph g , two positive integers $1 \leq r < s$, and two r -cliques C_1^r and C_2^r , C_1^r and C_2^r are **s -clique-connected** if there exists a sequence of r -cliques in g , $C_1^r = C_1, C_2, \dots, C_k = C_2^r$, such that for each $1 \leq i < k$, there exists an s -clique in g containing both C_i and C_{i+1} .

Definition 5.6. Given a graph G and three positive integers, k , r , and s with $r < s$, a k - (r,s) -nucleus of G is a maximal union g of r -cliques in G such that:

- for any r -clique C^r in g , its s -clique support in g is at least k (i.e., $\text{supp}_g^s(C^r) \geq k$), and
- every pair of two r -cliques in g are s -clique-connected in g .



(a) 2-(2,3)-nucleus



(b) 2-(2,4)-nucleus

Fig. 5.3: Nucleus decomposition [78]

Nuclear-decomposition

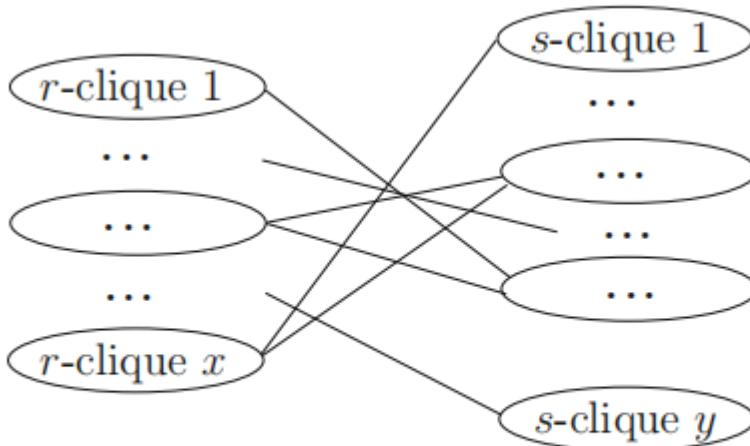


Fig. 5.4: Bipartite graph for nucleus decomposition

Algorithm 6: PeelNucleus: compute nucleus numbers of all r -cliques [78]

Input: A graph $G = (V, E)$, and integers r and s with $r < s$

Output: $\text{nucleus}_s(C^r)$ for each r -clique C^r in G

```

1  $\mathcal{C}_r \leftarrow$  the set of all  $r$ -cliques in  $G$ ;
2  $\mathcal{C}_s \leftarrow$  the set of all  $s$ -cliques in  $G$ ;
3 for each  $C^r \in \mathcal{C}_r$  do
4    $\text{supp}^s(C^r) \leftarrow |\{C^s \in \mathcal{C}_s \mid C^r \subset C^s\}|$ ;
5    $\text{max\_nucleus} \leftarrow 0$ ;
6 while  $\mathcal{C}_r \neq \emptyset$  do
7    $C^r \leftarrow$  the  $r$ -clique in  $\mathcal{C}_r$  with the minimum support;
8   if  $\text{supp}^s(C^r) > \text{max\_nucleus}$  then  $\text{max\_nucleus} \leftarrow \text{supp}^s(C^r)$ ;
9    $\text{nucleus}_s(C^r) \leftarrow \text{max\_nucleus}$ ;
10   $\mathcal{C}_r \leftarrow \mathcal{C}_r \setminus \{C^r\}$ ;
11  for each  $s$ -clique  $C^s \in \mathcal{C}_s$  containing  $C^r$  do
12    for each  $r$ -clique  $C_1^r \in \mathcal{C}_r$  contained in  $C^s$  do
13       $\text{supp}^s(C_1^r) \leftarrow \text{supp}^s(C_1^r) - 1$ ;

```

Theorem 5.3. The time complexity of PeelNucleus is $\mathcal{O}(s \times (\alpha(G))^{s-2} \times m)$ and the space complexity of PeelNucleus is $\mathcal{O}(n_r + n_s)$.

Higher-Order Densest Subgraph Computation

Definition 5.8 ([89]). The *k-clique density* of a graph g for $k \geq 2$, denoted $\rho_k(g)$, is

$$\rho_k(g) = \frac{c_k(g)}{|V(g)|},$$

where $c_k(g)$ denotes the number of k -cliques in g .

Note that the edge density studied in Chapter 4 is simply $\rho_2(g)$, since each edge is a 2-clique.

Definition 5.9 ([89]). Given a graph G , a subgraph g of G is a *k-clique densest subgraph* of G for a given k if $\rho_k(g) \geq \rho_k(g')$ for all subgraphs g' of G .

Higher-Order Densest Subgraph Computation

Lemma 5.6. Let S^* be the k -clique densest subgraph of G , then:

$$\min_{u \in S^*} f_{S^*}(u) \geq \rho_k(S^*)$$

Proof. As S^* is the k -clique densest subgraph of G , for any $u \in S^*$, it satisfies that

Algorithm 7: CDS-Greedy: a k -approximation algorithm for k -clique densest subgraph [89]

Input: A graph $G = (V, E)$

Output: A k -clique dense subgraph \tilde{S}

```

1  $\tilde{S} \leftarrow \emptyset$ ;
2  $S \leftarrow V$ ;  $c_k(S) \leftarrow 0$ ;
3 for each vertex  $u \in V$  do
4   Compute the number  $f_S(u)$  of  $k$ -cliques in  $G$  containing  $u$ ;
5    $c_k(S) \leftarrow c_k(S) + f_S(u)$ ;
6 while  $S \neq \emptyset$  do
7    $\rho_k(S) \leftarrow c_k(S)/|S|$ ;
8   if  $\tilde{S} = \emptyset$  or  $\rho_k(S) > \rho_k(\tilde{S})$  then
9      $\tilde{S} \leftarrow S$ ;
10  Obtain the vertex  $u$  in  $S$  that has the smallest  $f_S(u)$  value;
11   $S \leftarrow S \setminus \{u\}$ ;
12  for each  $(k-1)$ -clique  $C$  in  $S \cap N(u)$  do
13    for each vertex  $v \in C$  do
14       $f_S(v) \leftarrow f_S(v) - 1$ ;
15     $c_k(S) \leftarrow c_k(S) - k$ ;
16 return  $\tilde{S}$ ;

```

Higher-Order Densest Subgraph Computation

Density Testing. Let ρ_k^* be the k -clique density of the k -clique densest subgraph of G . To test whether ρ_k^* is larger than a positive real number x , an augmented and weighted directed graph $G_x = (V(G_x), E(G_x), w)$ is constructed from $G = (V, E)$ as follows [60]:

- $V(G_x) = \{s, t\} \cup V \cup \mathcal{C}_{k-1}(G)$ where $\mathcal{C}_{k-1}(G)$ denotes the set of all $(k-1)$ -cliques in G .
- For each vertex $u \in V$, there is a directed edge of weight $f_G(u)$ from s to u , where $f_G(u)$ is the number of k -cliques in G containing u .
- For each vertex $u \in V$, there is a directed edge of weight $k \times x$ from u to t .
- For each vertex $u \in V$ and each $(k-1)$ -clique $C \in \mathcal{C}_{k-1}(G)$ such that u and C form a k -clique, there is a directed edge of weight 1 from u to C .
- For each $(k-1)$ -clique $C \in \mathcal{C}_{k-1}(G)$, there is a directed edge of weight ∞ from C to each vertex $u \in C$.

Theorem 5.5 ([60]). For any positive x , $\rho_k^* > x$ if and only if the minimum $s-t$ cut in G_x is of value smaller than $k \times c_k(G)$.

Algorithm 8: CDS-Exact: an exact algorithm for k -clique densest subgraph [60]

Input: A graph $G = (V, E)$
Output: A k -clique densest subgraph S^* of G

```

1  $\tilde{S} \leftarrow \emptyset$ ;
2  $l \leftarrow 0; h \leftarrow n^k$ ;
3  $\mathcal{C}_{k-1}(G) \leftarrow$  the set of  $(k-1)$ -cliques in  $G$ ;
4 while  $h - l \geq \frac{1}{n(n-1)}$  do
5    $x \leftarrow \frac{l+h}{2}$ ;
6    $G_x \leftarrow$  AugmentedGraph( $G, \mathcal{C}_{k-1}(G), x$ );
7    $(A, B) \leftarrow$  minimum  $s-t$  cut in  $G_x$ ;
8   if the value of the cut  $(A, B)$  is smaller than  $k \times c_k(G)$  then
9      $S^* \leftarrow A \cap V$ ;
10     $l \leftarrow x$ ;
11   else
12      $h \leftarrow x$ ;
13 return  $S^*$ ;

```



Edge Connectivity-Based Graph Decomposition

Springer Series in the Data Sciences 2018

Lijun Chang • Lu Qin

Higher-Order Densest Subgraph Computation

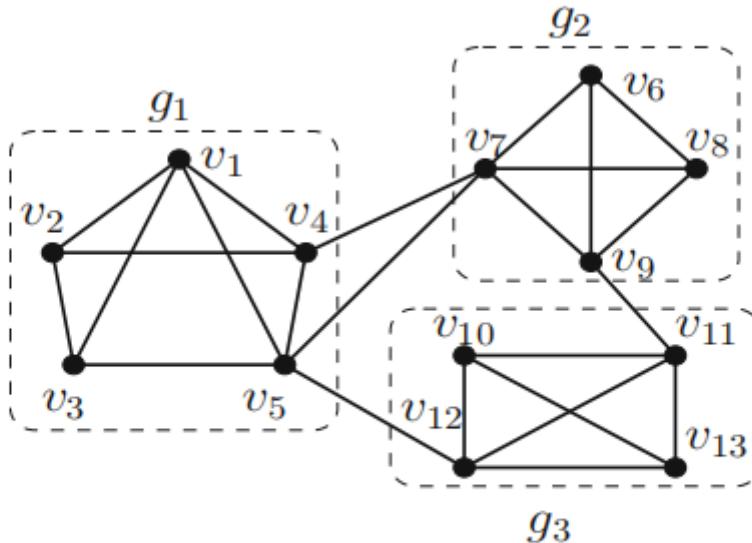


Fig. 6.1: A graph and its 3-edge connected components [17]

Algorithm 1: KECC: compute k -edge connected components [17]

Input: A graph $G = (V, E)$ and an integer k
Output: k -edge connected components of G

- 1 Initialize a queue \mathcal{Q} consisting of a single graph G ;
- 2 **while** \mathcal{Q} is not empty **do**
- 3 Pop a graph g from \mathcal{Q} ;
- 4 $\mathcal{G}_k \leftarrow \text{Partition}(g, k)$;
- 5 **if** \mathcal{G}_k consists of only one graph g' **then**
- 6 **Output** g' as a k -edge connected component of G ;
- 7 **else**
- 8 Push all graphs of \mathcal{G}_k into \mathcal{Q} ;

Partition函数是重点

Higher-Order Densest Subgraph Computation

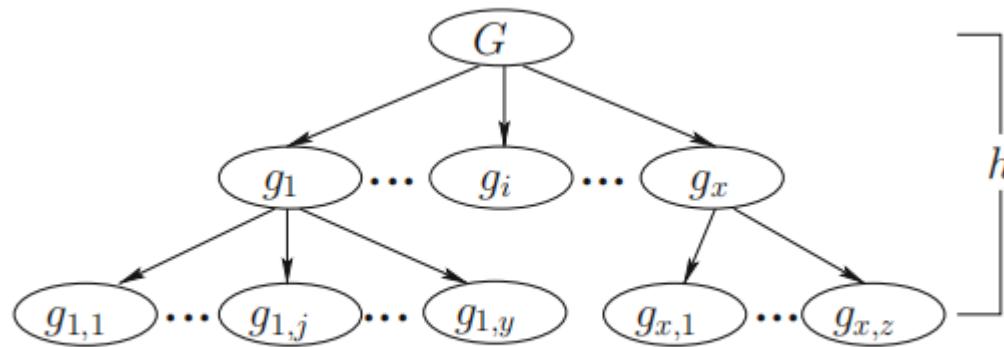


Fig. 6.2: A graph partition tree

Algorithm 4: Partition-MAS(g, k) [102]

```

1  $g' \leftarrow g$ ;
2 while  $g'$  contains at least two vertices do
3   Let  $(S, T)$  be the minimum  $s-t$  cut obtained by MAS( $g'$ );
4   if  $\omega(S, T) < k$  then return  $\{g[S], g[T]\}$ ;
5   else Let  $g'$  be the resulting graph of contracting  $s$  and  $t$  into a super-vertex;
6 return  $\{g\}$ ;

```

Algorithm 2: Partition-TwoWay(g, k)

```

1 if  $g$  is  $k$ -edge connected then
2   return  $\{g\}$ ;
3 else
4   Compute a cut  $C$  of value smaller than  $k$ ;
5   Let  $g_1, g_2$  be the two connected subgraphs obtained by removing  $C$  from  $g$ ;
6   return  $\{g_1, g_2\}$ ;

```

Algorithm 3: MAS: compute a minimum cut [85]

Input: A graph $g = (V(g), E(g))$
Output: A minimum $s-t$ cut (S, T)

```

1  $L \leftarrow \{\text{an arbitrary vertex of } V(g)\}$ ;
2 while  $|L| \neq |V(g)|$  do
3   Let  $u$  be the most tightly connected vertex to  $L$ , i.e.,  $u = \arg\max_{v \in V(g) \setminus L} \omega(L, v)$ ;
4   Add  $u$  to the tail of  $L$ ;
5 Let  $s$  and  $t$  be the two vertices most recently added to  $L$ , and  $s$  is added to  $L$  prior to  $t$ ;
6 return the cut  $(L \setminus \{t\}, \{t\})$  as a minimum  $s-t$  cut;

```

Higher-Order Densest Subgraph Computation

Algorithm 5: Partition-DS(g, k) [17]

```

 $\text{if the minimum degree of } g \text{ is smaller than } k \text{ then}$ 
    Let  $u$  be a vertex of degree smaller than  $k$ ;
    return  $\{g \setminus u, u\}$ ;
 $D \leftarrow \{\arg \max_{v \in V(g)} d(v)\}; /* D contains the maximum-degree vertex$ 
 $T \leftarrow \{u \in V(g) \setminus D \mid \exists v \in D, s.t., (u, v) \in E(g)\};$ 
while  $D \cup T \neq V(g)$  do
    Let  $u$  be the vertex with the maximum degree in  $g[V(g) \setminus D]$ ;
    Compute a minimum cut  $(S, T)$  of  $g$  such that  $D \subseteq S$  and  $u \in T$ ;
    if the value of  $(S, T)$  is smaller than  $k$  then
        return  $\{g[S], g[T]\}$ ;
     $D \leftarrow D \cup \{u\};$ 
     $F \leftarrow \{u \in V(g) \setminus D \mid \exists v \in D, s.t., (u, v) \in E(g)\};$ 
return  $\{g\}$ ;

```

Algorithm 6: Partition-MultiWay(g, k) [17]

```

1  $g \leftarrow$  the  $k$ -core of  $g$ ;
2  $g' \leftarrow g$ ;
3 while  $g'$  contains more than one vertices do
4   MAS-Opt( $g', k, g$ ); /*  $g'$  and  $g$  are modified in MAS-Opt */;
5 return the set of connected components of  $g$ ;

Procedure MAS-Opt( $g', k, g$ )
6  $L \leftarrow \{\text{an arbitrary vertex } u \text{ of } V(g')\}$ ;
7 for each  $v \in V(g')$  do  $\omega(v) \leftarrow \omega(L, v)$ ;
8 while  $|L| \neq |V(g')|$  do
9    $u \leftarrow \arg \max_{v \in V(g') \setminus L} \omega(v)$ ;
10  Add  $u$  to the tail of  $L$ , and initialize a queue  $Q$  with  $u$ ;
11  while  $Q \neq \emptyset$  do
12     $v \leftarrow \text{pop a vertex from } Q$ ;
13    for each  $(v, w) \in E(g')$  with  $w \notin L$  do
14       $\omega(w) \leftarrow \omega(w) + 1$ ;
15      if  $\omega(w) = k$  then Push  $w$  into  $Q$ ;
16    if  $u \neq v$  then Contract vertices  $u$  and  $v$  in  $g'$ ;
17 while  $|L| > 1$  and the value of the cut  $C$  implied by the last two vertices in  $L$  is less than  $k$ 
18   Remove the last vertex of  $L$  from both  $g'$  and  $L$ ;
19   Remove all edges of  $C$  from  $g$ ;

```



谢谢大家！