# Cohesive Subgraph Computation over Large Sparse Graphs

Springer Series in the Data Sciences 2018

Lijun Chang • Lu Qin

## Graph Terminologies

- V(g), E(g) denote the set of vertices and the set of edges of g
- The set of neighbors of a vertex u $N_g(u) = \{v \in V(g) \mid (u,v) \in E(g)\},$
- The degree of u in g $d_g(u) = |N_g(u)|.$
- The minimum vertex degree, the average vertex degree, and the maximum vert $d_{min}(g)$, $d_{avg}(g)$, and $d_{max}(g)$,
- The subgraph of g induced $g[V_s] = (V_s, \{(u,v) \in E(g) \mid u \in V_s, v \in V_s\})$
- The subgraph of g induced $g[E_s] = (\bigcup_{(u,v)\in E_s} \{u,v\}, E_s)$

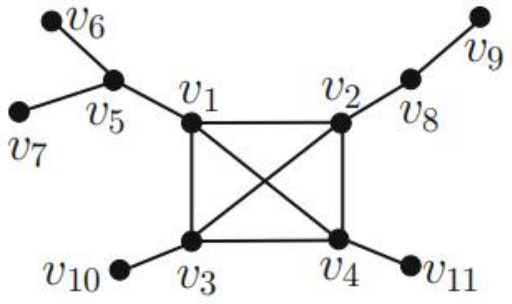



Fig. 1.1: An example unweighted undirected graph

# Real Graph Datasets

- Real Graph Data Repositories.
  - Stanford Network Analysis Project (SNAP)[1]: maintains a collection of more than 50 large network datasets from tens of thousands of vertices and edges to tens of millions of vertices and billions of edges
  - Laboratory for Web Algorithmics (LAW) : [2] mainly web graphs and social networks
  - Network Repository: [3] large network repository archiving thousands of graphs
  - The Koblenz Network Collection (KONECT): [4] contains several hundred network datasets with up-to tens of millions of vertices and billions of edges

[1] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.
[2] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In Proc. of *WWW*'04, pages 595–601, 2004.
[3] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. *In Proc. of AAAI'15*, 2015.
[4] http://konect.uni-koblenz.de/.

# 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 ……

| Graphs | $n$ | $m$ | $d_{avg}(G)$ | $d_{max}(G)$ | $\delta(G)$ |
|---|---|---|---|---|---|
| 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 $G$)

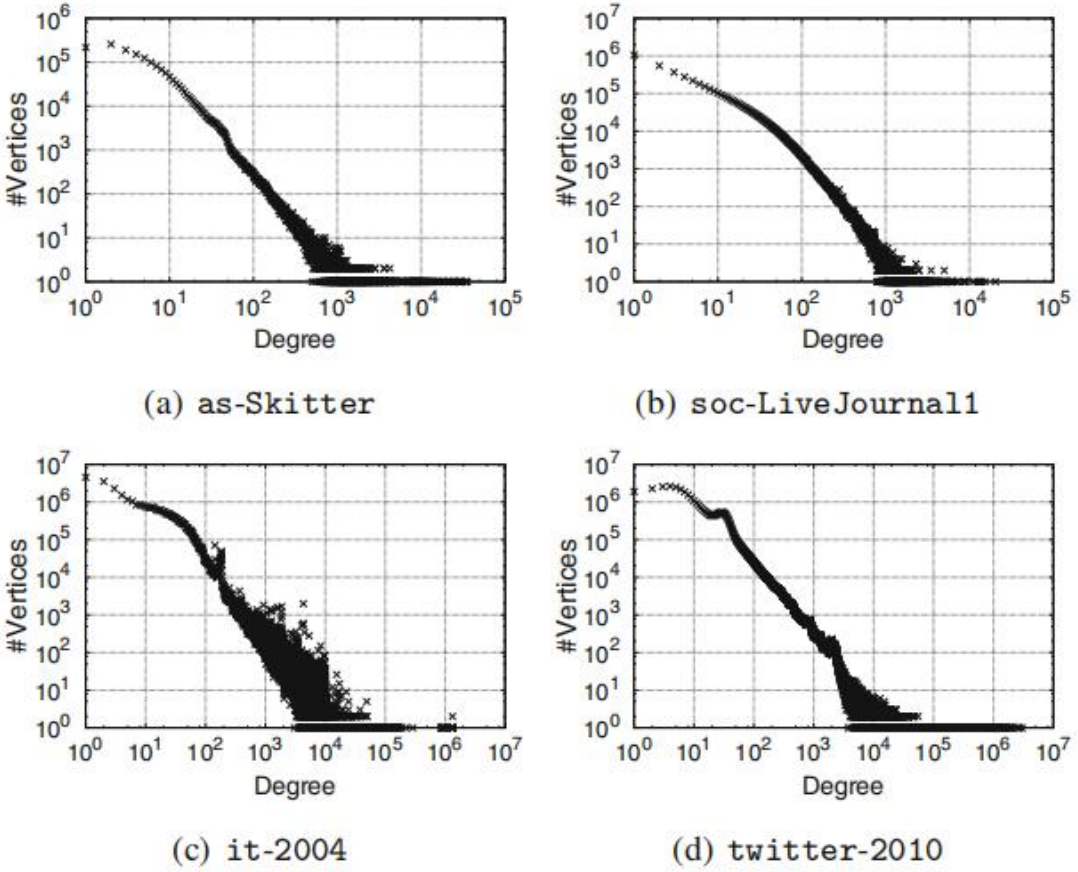



Fig. 1.2: Degree distributions

## 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],...,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];
```

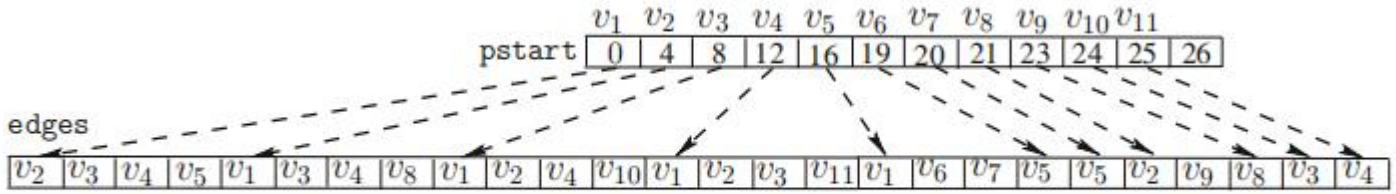



Fig. 1.3: Adjacency array graph representation

[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 $\theta$-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.$|E(g)| = \frac{|V(g)|(|V(g)-1|}{2}$).
- ***ɤ-quasi clique***: at least ***ɤ*** portion of its vertex are connected by edges (i.e., $|\mathrm{E(g)}| \geq \gamma \times \frac{|\mathrm{V(g)}|(|\mathrm{V(g)}|-1)}{2}$).
- ***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.

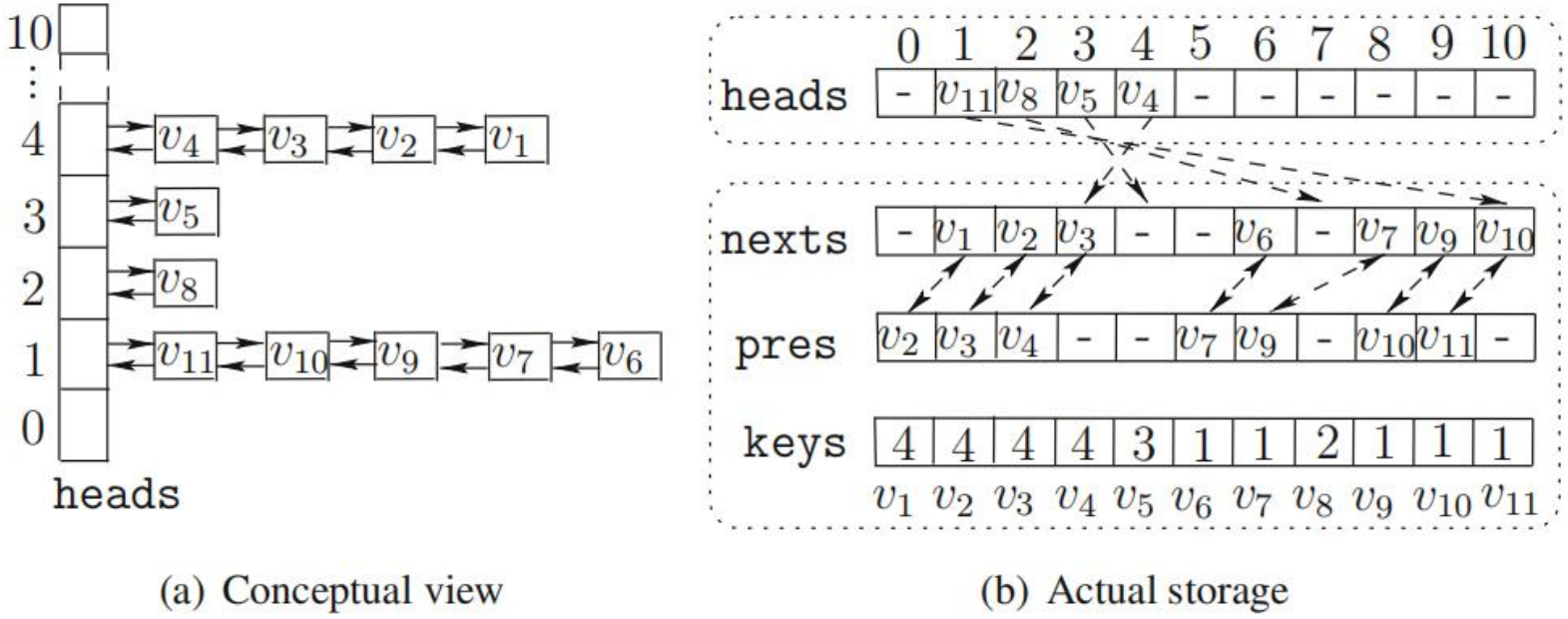
## Applications

**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 succeding 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 $\mathcal{O}(n + key\_cap)$ 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* $\mathcal{O}(\texttt{key\_cap} + x)$ *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* $\mathcal{O}(\texttt{\_key\_cap}+x)$ *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`, 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 $t^+ \leq t^- + \texttt{\_key\_cap}$. Therefore, the above time complexity is $\mathcal{O}(\texttt{\_key\_cap}+x)$.

# *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* $\mathcal{O}(\texttt{\_key\_cap} + \texttt{\_n})$ *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. □

## 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., rids[ids[$i$]]=$i$).

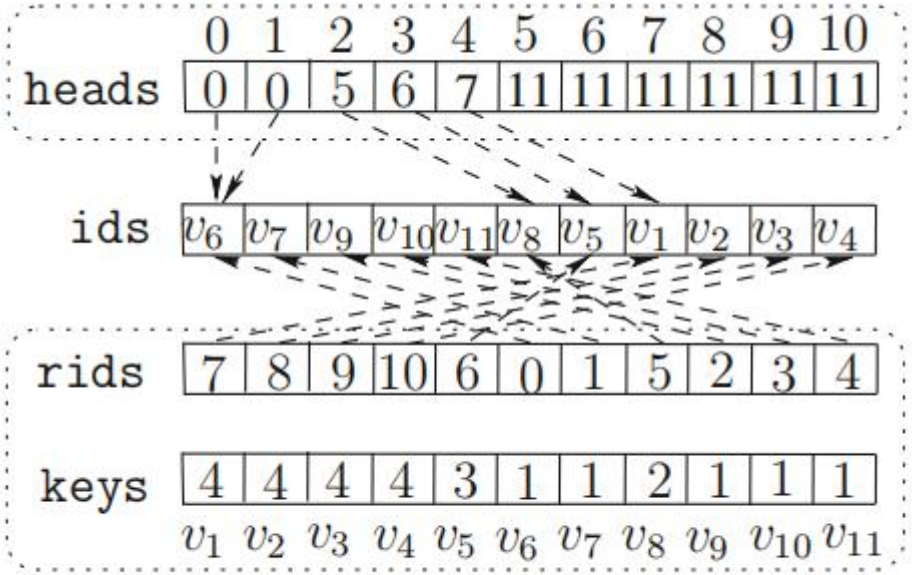



Fig. 2.2: An example of array-based linear heap

# 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 am 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
           /* element has the maximum key value                    */
9          key ← max_key;
10         return true;
11     else
           /* Insert element into the proper singly linked list    */
12         nexts[element] ← heads[keys[element]];
13         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. □

## 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 $g$.

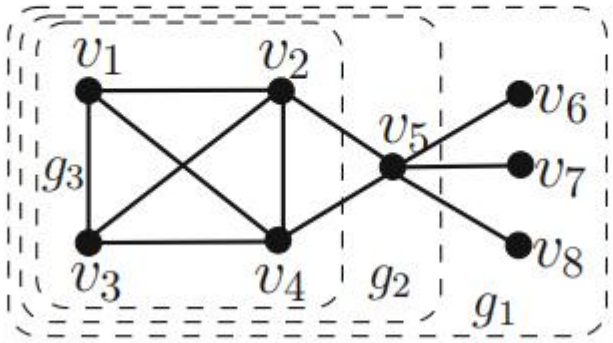



Fig. 3.1: An example graph and its $k$-cores

**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 $\texttt{core}(u)$, 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.** $\delta(G) \leq \lceil \sqrt{2m+n} \rceil$.

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

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

Thus, we have $\delta(G) \leq \lceil \sqrt{2m+n} \rceil$ 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., $\delta(G) = \max_{u \in V} \texttt{core}(u)$).*

*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, $\max_{u \in V} \texttt{core}(u) \leq \delta(G)$. Secondly, $G$ must contain a $\delta(G)$-core, since it has at least one subgraph whose minimum vertex degree is $\delta(G)$; thus, $\max_{u \in V} \texttt{core}(u) \geq \delta(G)$. The lemma follows. □

**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**: $\texttt{core}(v)$ for each vertex $v \in V$

```
1  for each v ∈ V do
2      Let d(v) be the degree of v in G;
3  max_core ← 0; seq ← ∅;
4  for i ← 1 to n do
5      u ← argmin_{v∈V\seq} d(v);
6      Add u to the tail of seq;
7      if d(u) > max_core then max_core ← d(u);
8      core(u) ← max_core;
9      for each neighbor v of u that is not in seq do
10         d(v) ← d(v) − 1;
```

**Definition 3.5 ([59]).** Given a graph $G$, a permutation $(v_1, v_2, \ldots, v_n)$ 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 $\{v_i, \ldots, v_n\}$.

## 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. □

**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* `seq`$[p_k, \ldots, n-1]$*, 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 `seq`$[p_k, \ldots, n-1]$. Firstly, the $k$-core of $G$ contains none of the vertices in `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. □

## 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 ∈ 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 ≠ ∅ do
6      u ← pop a vertex from Q;
       /* Remove u from the graph                    */
7      for each neighbor v of u do
8          d(v) ← d(v) − 1;
9          if d(v) = k − 1 then Push v to Q;
10 return the subgraph of G induced by vertices with d(·) ≥ 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, $\texttt{parent}[i] = i$ and $\texttt{rank}[i] = 0$.
- `UF_find`$(i)$ returns the representative of the unique set containing $i$.

- `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* $\mathcal{O}(x \times \mathtt{a}(x))$, *where* $\mathtt{a}(x)$ *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 $\mathtt{a}(x)$ is constant and ignore it in the time complexity analysis. The space complexity of the disjoint-set data structure is $2\mathtt{n} + \mathcal{O}(1)$ since each of the arrays `parent` and `rank` has size `n`.

## Core Hierarchy Tree

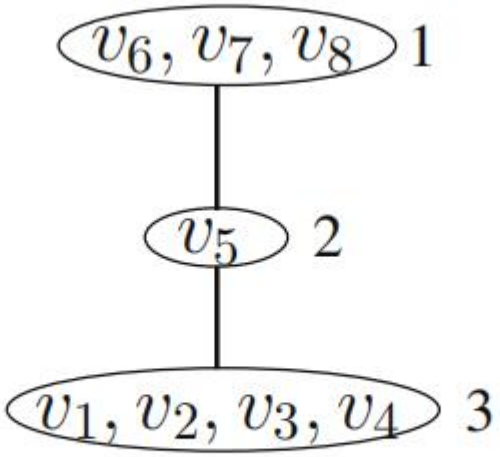



Fig. 3.3: The core hierarchy tree for the graph in Figure 3.1

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 `core`$(\cdot)$ of vertices
**Output**: A core hierarchy tree `CoreHT` of $G$

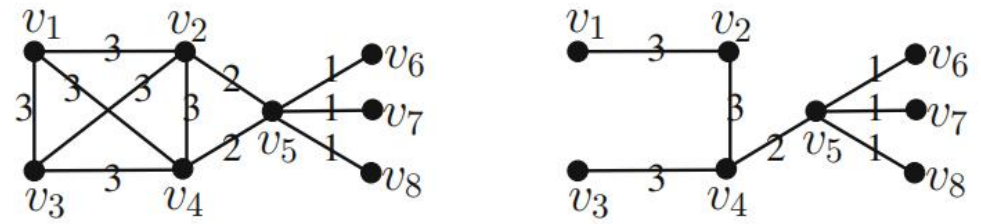
```
1  Initialize an empty CoreHT, and a disjoint-set data structure F for V;
2  for each vertex u ∈ V do
3      Add a node r_u, with weight core(u) and containing vertex u, to CoreHT;
4      Point u to r_u;
5  for each vertex u in seq in reverse order do
6      for each neighbor v of u in G that appear later than u in 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 F, respectively;
8          if r_v ≠ r_u then
               /* Update the CoreHT                                          */
9              if the weight of r_v equals the weight of r_u then
10                 Move the content (i.e., vertices and children) of r_v to r_u;
11             else Assign r_u as the parent of r_v in the CoreHT;
               /* Update the disjoint-set data structure F                   */
12             Union u and v in F, and point the updated representative of the set containing
               u to r_u;
13 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{core(u), 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. □




(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 `core`$(\cdot)$ of vertices

**Output**: A core spanning tree `CoreSPT` of $G$

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

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

# 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 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., $C(u) = \{\texttt{core}(v) \mid v \in N(u)\}$). Then, the* h-index *of $C(u)$ equals the core number of $u$ in $G$, that is:*

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

*Proof.* Firstly, we prove that $\texttt{core}(u) \geq \texttt{h-index}(C(u))$. Let $k$ be the value of $\texttt{h-index}(C(u))$. 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 $\texttt{core}(u) \geq k = \texttt{h-index}(C(u))$.

Secondly, we prove that $\texttt{core}(u) \leq \texttt{h-index}(C(u))$. Let $k$ be the value of $\texttt{core}(u)$. 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, $\texttt{h-index}(C(u)) \geq k = \texttt{core}(u)$. Thus, the lemma holds. □

## Core Decomposition in Other Environments

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

$$\texttt{core}(u) \leq \texttt{h-index}(\overline{C}(u)).$$

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

*Proof.* For a vertex $u$, let $k$ be the value of $\overline{\texttt{core}}(u)$ and consider the subgraph $g$ of $G$ induced by the set of vertices whose upper bounds of core numbers are at least $k$. As $\texttt{h-index}(\overline{C}(v)) = \overline{\texttt{core}}(v)$ 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, $\texttt{core}(u) = k = \overline{\texttt{core}}(u)$ and the lemma holds. □

## Core Decomposition in Other Environments

**Algorithm 5:** CoreD-Local: compute core numbers of vertices [54]

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

**for each** *vertex* $u \in V$ **do** $\overline{\texttt{core}}(u) \leftarrow$ the degree of $u$ in $G$;
update $\leftarrow$ `true`;
**while** update **do**
    update $\leftarrow$ `false`;
    **for each** *vertex* $u \in V$ **do**
        $old \leftarrow \overline{\texttt{core}}(u)$;
        $\overline{\texttt{core}}(u) \leftarrow \text{HIndex}(N(u), \overline{\texttt{core}})$;
        **if** $\overline{\texttt{core}}(u) \neq old$ **then** update $\leftarrow$ `true`;
**return** $\texttt{core}(v) \leftarrow \overline{\texttt{core}}(v)$ for each vertex $v \in V$;

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

# Core Decomposition in Other Environments

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

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

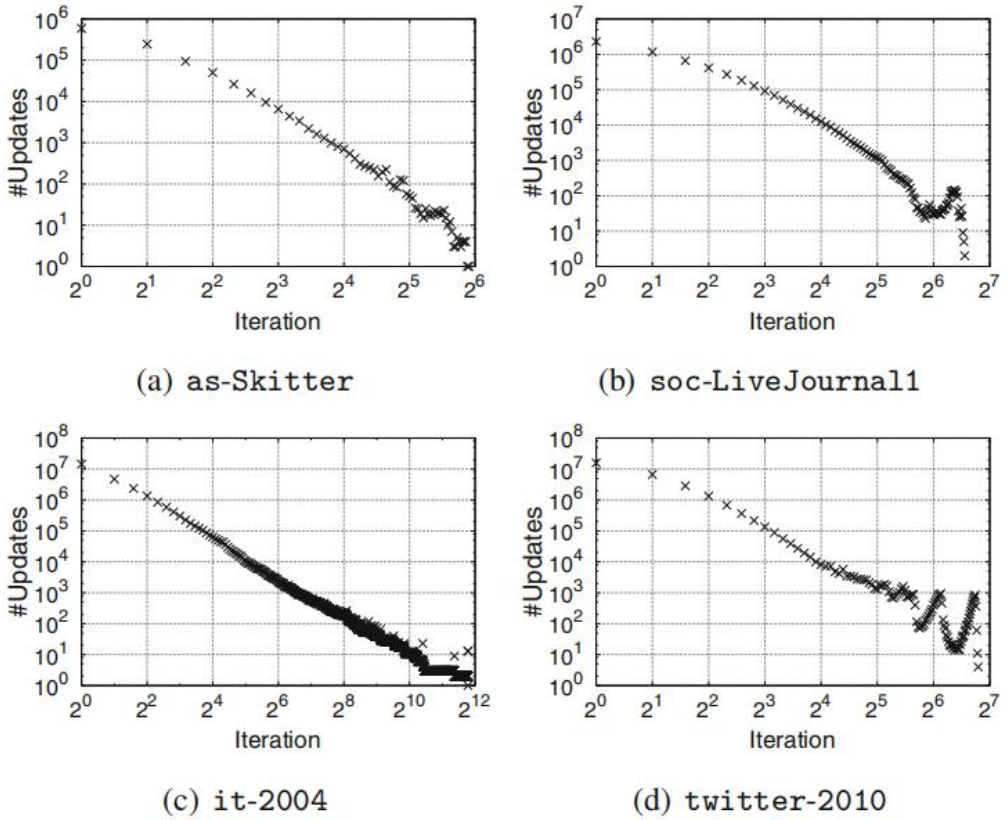
**Input**: A graph $G = (V, E)$
**Output**: $\texttt{core}(v)$ for each vertex $v \in V$

```
1  Q ← ∅;
2  for each vertex u ∈ V do
3      core̅(u) ← the degree of u in G;
4      cnt(u) ← the number of neighbors of u whose degrees are no smaller than that of u;
5      if cnt(u) < core̅(u) then Push u into Q;
6  while Q ≠ ∅ do
7      Q′ ← ∅;
8      while Q ≠ ∅ do
9          Pop a vertex u from Q;
10         old ← core̅(u);
11         core̅(u) ← HIndex(N(u), core̅);
12         cnt(u) ← 0;
13         for each neighbor v of u do
14             if core̅(v) ≥ core̅(u) then cnt(u) ← cnt(u) + 1;
15             if v ∉ Q and core̅(u) < core̅(v) ≤ old then
16                 if cnt(v) = core̅(v) then Push v into Q′;
17                 cnt(v) ← cnt(v) − 1;
18     Q ← Q′;
19 return core(v) ← core̅(v) for each vertex v ∈ V;
```

**Theorem 3.2.** *The time complexity of Algorithm 6 is* $\mathcal{O}(m \times \texttt{h-index}(G))$. *Here,* $\texttt{h-index}(G)$ *is the largest value computed by* $\mathsf{HIndex}(N(u), d)$ *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**: $\texttt{core}(v)$ for each vertex $v \in V$

```
1  v_min ← v_n; v_max ← v_1;
2  for vertex u ← v_1 to v_n do
3      core̅(u) ← the degree of u in G;
4      Load N(u) from disk;
5      cnt(u) ← the number of vertices in N(u) whose degrees are no smaller than that of u;
6      if cnt(u) < core̅(u) then
7          if u < v_min then v_min ← u;
8          v_max ← u;
9  while v_min ≤ v_max do
10     v'_min ← v_n; v'_max ← v_1;
11     for vertex u ← v_min to v_max s.t. core̅(u) > cnt(u) do
12         Load N(u) from disk;
13         old ← core̅(u);
14         core̅(u) ← HIndex(N(u), core̅);
15         cnt(u) ← 0;
16         for each vertex v ∈ N(u) do
17             if core̅(v) ≥ core̅(u) then cnt(u) ← cnt(u) + 1;
18             if core̅(u) < core̅(v) ≤ old then
19                 if cnt(v) = core̅(v) and not u < v ≤ v_max then
20                     if v < v'_min then v'_min ← v;
21                     if v > v'_max then v'_max ← v;
22                 cnt(v) ← cnt(v) − 1;
23     v_min ← v'_min; v_max ← v'_max;
24 return core(v) ← core̅(v) for each vertex v ∈ V;
```

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。

# 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 $d_{avg}(g) = \frac{\sum_{v \in V(g)} d_g(v)}{|V(g)|} = \frac{2 \times |E(g)|}{|V(g)|} = 2 \times \rho(g)$. □

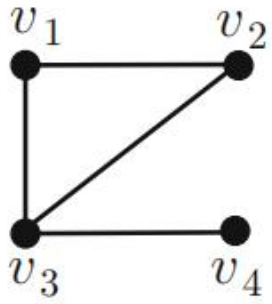



Fig. 4.1: An example graph

**Definition 4.2.** Given a graph $G$, a subgraph $g$ of $G$ is a ***densest subgraph*** of $G$ if $\rho(g) \geq \rho(g')$ for all subgraphs $g'$ of $G$.

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。

# Propertites

**Lemma 4.2.** *The minimum degree of $S^*$ is no smaller than $\rho^*$, 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 $S^* \backslash \{u\}$. 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. □

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

# 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]

**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  S̃ ← ∅;
3  S ← V; m_S ← |E|;
4  for each vertex u of G according to the degeneracy ordering do
5      ρ(S) ← m_S/|S|;
6      if S̃ = ∅ or ρ(S) > ρ(S̃) then
7          S̃ ← S;
8      S ← S\{u};
9      for each neighbor v of u such that v ∈ S do
10         m_S ← m_S − 1;
11 return S̃;
```

Time Complexity： O(m)

# 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+ε)-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  S̃ ← ∅;
2  S ← V; m_S ← |E|;
3  for each vertex u ∈ V do d_S(u) ← the degree of u in G;
4  while S ≠ ∅ do
5      ρ(S) ← m_S/|S|;
6      if S̃ = ∅ or ρ(S) > ρ(S̃) then
7          S̃ ← S;
8      Δ(S) ← {v ∈ S | d_S(v) ≤ 2 × (1+ε) × ρ(S)};
9      for each v ∈ Δ(S) do
10         S ← S \ {v};
11         for each neighbor u of v in S do
12             d_s(u) ← d_s(u) − 1;
13             m_S ← m_S − 1;
14 return S̃;
```

# A Steaming 2(1+ε)-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* $\mathcal{O}(\frac{m \times \log_{1+\varepsilon} n}{B})$*, where B is the disk block size.*

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

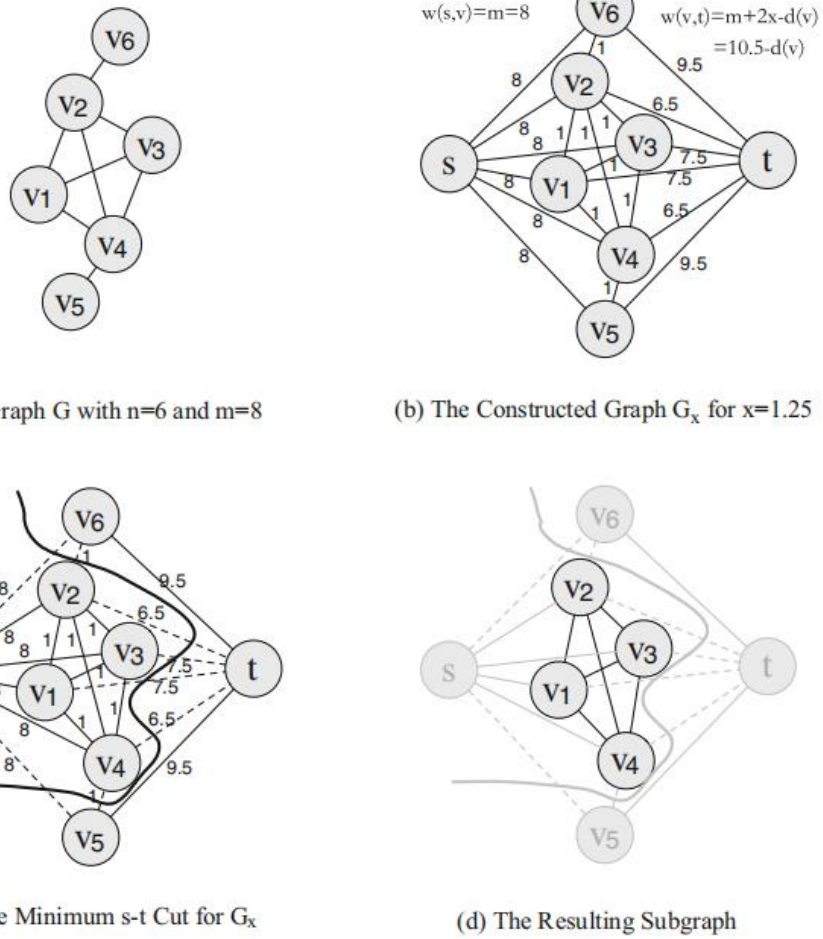
# Density Testing

- $V(G_x) = V \cup \{s,t\}$, 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=8

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

(c) The Minimum s-t Cut for $G_x$

(d) The Resulting Subgraph

Fig. 4.2: An example to illustrate DensityTest

# 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 ← G;
2 Assign a weight 1 to every edge of G_x;
3 Add a source vertex s ∉ V and a sink vertex t ∉ V to G_x;
4 for each vertex u ∈ 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 \ {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 $G$

$\tilde{S} \leftarrow \emptyset$;
$l \leftarrow 0$; $h \leftarrow m$;
**while** $h - l \geq \frac{1}{n(n-1)}$ **do**
    $x \leftarrow \frac{l+h}{2}$;
    $S \leftarrow \mathsf{DensityTest}(G, x)$;
    **if** $S \neq \emptyset$ **then**
        $\tilde{S} \leftarrow S$;
        $l \leftarrow x$;
    **else**
        $h \leftarrow x$;
**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) ≥ ··· ≥ d(v_n);
2 for each vertex u ← v_1, ..., v_n do
3     for each neighbor v ∈ N(u) do Mark v;
4     for each neighbor v ∈ N(u) do
5         for each neighbor w ∈ N(v) do
6             if w is marked and v < w then
7                 Output triangle △_{u,v,w};
8     for each neighbor v ∈ 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 ≺;
2 for each vertex u ∈ V do
3     for each out-neighbor v ∈ N+(u) do Mark v;
4     for each out-neighbor v ∈ N+(u) do
5         for each out-neighbor w ∈ N+(v) do
6             if w is marked then
7                 Output triangle △u,v,w;
8     for each out-neighbor v ∈ 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 ← ∅;
2  KClique-Enum(G, k, C);

   Procedure KClique-Enum(G_k, k, C)
3  if k = 2 then
4      for each edge (u, v) ∈ E(G_k) do
5          Output clique {u, v} ∪ C;
6  else
7      Sort vertices of G_k such that d_{G_k}(v_1) ≥ ··· ≥ d_{G_k}(v_{|V(G_k)|});
8      for each vertex u ← v_1, ..., v_{|V(G_k)|} do
9          G_{k-1} ← the subgraph of G_k induced by N_{G_k}(u);
10         KClique-Enum(G_{k-1}, k − 1, C ∪ {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:** KClique-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 $G$

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

**Procedure** KClique-EnumO$(G_k^+, k, C)$
**if** $k = 2$ **then**
    **for each** *edge* $(u, v) \in E(G_k^+)$ **do**
        Output clique $\{u, v\} \cup C$;
**else**
    **for each** *vertex* $u \in V(G_k^+)$ **do**
        $G_{k-1}^+ \leftarrow$ the subgraph of $G_k^+$ induced by $N_{G_k^+}^+(u)$;
        KClique-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.

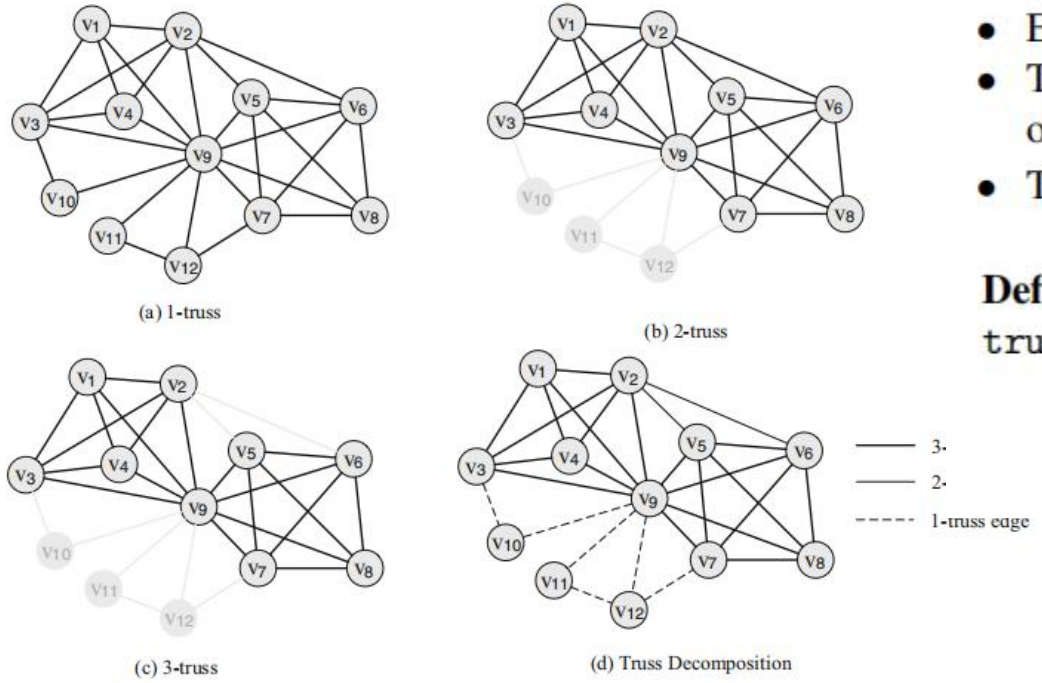



Fig. 5.2: Truss decomposition

- Each $k$-truss of a graph $G$ is a subgraph of the $(k+1)$-core of $G$.
- The edge connectivity of a $k$-truss is at least $k+1$ (see Chapter 6 for the definition of edge connectivity).
- The diameter of a $k$-truss $g$ is at most $\lfloor \frac{2\times|V(g)|-2}{k+2} \rfloor$.

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

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

# k-truss

**Algorithm 5:** PeelTruss: compute truss numbers of all edges [92]

**Input**: A graph $G = (V, E)$
**Output**: $\texttt{truss}(u, v)$ for each edge $(u, v) \in E$

```
for each edge (u,v) ∈ E do
    Compute the number t(u,v) of triangles in G containing (u,v);
    supp(u,v) ← t(u,v);
max_truss ← 0; seq ← ∅;
for i ← 1 to m do
    (u,v) ← arg min_{(u',v')∈E\seq} supp(u',v');
    Add (u,v) to the tail of seq;
    if supp(u,v) > max_truss then max_truss ← supp(u,v);
    truss(u,v) ← max_truss;
    for each triangle △_{u,v,w} that contains edge (u,v) do
        supp(u,v) ← supp(u,v) − 1;
        supp(u,w) ← supp(u,w) − 1;
    Remove edge (u,v) from the graph;
```

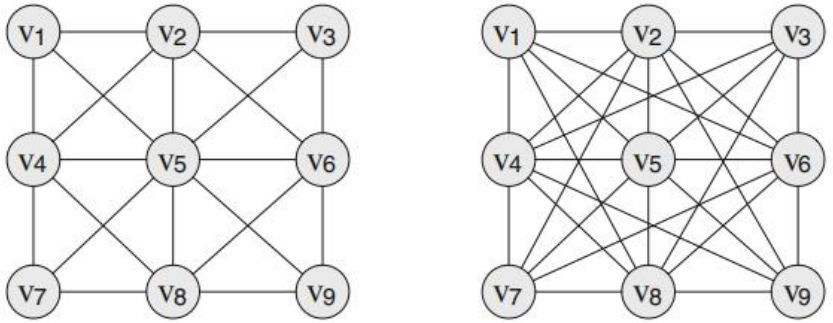
# Nuclear-decomposition

**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 $g$, $C_1^r = C_1, C_2, \ldots, C_k = C_2^r$, 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

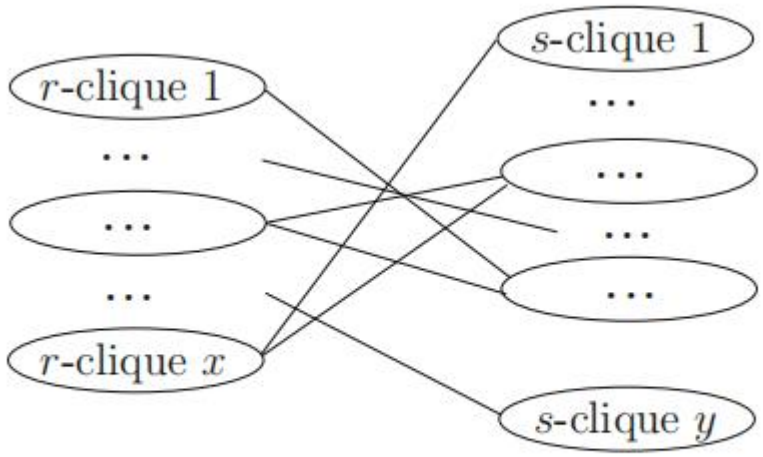



Fig. 5.4: Bipartite graph for nucleus decomposition

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

$\mathscr{C}_r \leftarrow$ the set of all $r$-cliques in $G$;
$\mathscr{C}_s \leftarrow$ the set of all $s$-cliques in $G$;
**for each** $C^r \in \mathscr{C}_r$ **do**
  $\texttt{supp}^s(C^r) \leftarrow |\{C^s \in \mathscr{C}_s \mid C^r \subset C^s\}|$;
$\texttt{max\_nucleus} \leftarrow 0$;
**while** $\mathscr{C}_r \neq \emptyset$ **do**
  $C^r \leftarrow$ the $r$-clique in $\mathscr{C}_r$ with the minimum support;
  **if** $\texttt{supp}^s(C^r) > \texttt{max\_nucleus}$ **then** $\texttt{max\_nucleus} \leftarrow \texttt{supp}^s(C^r)$;
  $\texttt{nucleus}_s(C^r) \leftarrow \texttt{max\_nucleus}$;
  $\mathscr{C}_r \leftarrow \mathscr{C}_r \setminus \{C^r\}$;
  **for each** *s-clique* $C^s \in \mathscr{C}_s$ *containing* $C^r$ **do**
    **for each** *r-clique* $C^r_1 \in \mathscr{C}_r$ *contained in* $C^s$ **do**
      $\texttt{supp}^s(C^r_1) \leftarrow \texttt{supp}^s(C^r_1) - 1$;

**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  S̃ ← ∅;
2  S ← V; c_k(S) ← 0;
3  for each vertex u ∈ V do
4      Compute the number f_S(u) of k-cliques in G containing u;
5      c_k(S) ← c_k(S) + f_S(u);
6  while S ≠ ∅ do
7      ρ_k(S) ← c_k(S)/|S|;
8      if S̃ = ∅ or ρ_k(S) > ρ_k(S̃) then
9          S̃ ← S;
10     Obtain the vertex u in S that has the smallest f_S(u) value;
11     S ← S \ {u};
12     for each (k−1)-clique C in S ∩ N(u) do
13         for each vertex v ∈ C do
14             f_S(v) ← f_S(v) − 1;
15         c_k(S) ← c_k(S) − k;
16 return S̃;
```

# Higher-Order Densest Subgraph Computation

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

- $V(G_x) = \{s, t\} \cup V \cup \mathscr{C}_{k-1}(G)$ where $\mathscr{C}_{k-1}(G)$ 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 $C \in \mathscr{C}_{k-1}(G)$ 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 $C \in \mathscr{C}_{k-1}(G)$, there is a directed edge of weight $\infty$ 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 $G$

$\tilde{S} \leftarrow \emptyset$;
$l \leftarrow 0$; $h \leftarrow n^k$;
$\mathscr{C}_{k-1}(G) \leftarrow$ the set of $(k-1)$-cliques in $G$;
**while** $h - l \geq \frac{1}{n(n-1)}$ **do**
  $x \leftarrow \frac{l+h}{2}$;
  $G_x \leftarrow \mathsf{AugmentedGraph}(G, \mathscr{C}_{k-1}(G), x)$;
  $(A, B) \leftarrow$ minimum $s$–$t$ cut in $G_x$;
  **if** *the value of the cut $(A, B)$ is smaller than $k \times c_k(G)$* **then**
    $S^* \leftarrow A \cap V$;
    $l \leftarrow x$;
  **else**
    $h \leftarrow x$;
**return** $S^*$;

# Edge Connectivity-Based Graph Decomposition

Springer Series in the Data Sciences 2018

Lijun Chang • Lu Qin

# Higher-Order Densest Subgraph Computation

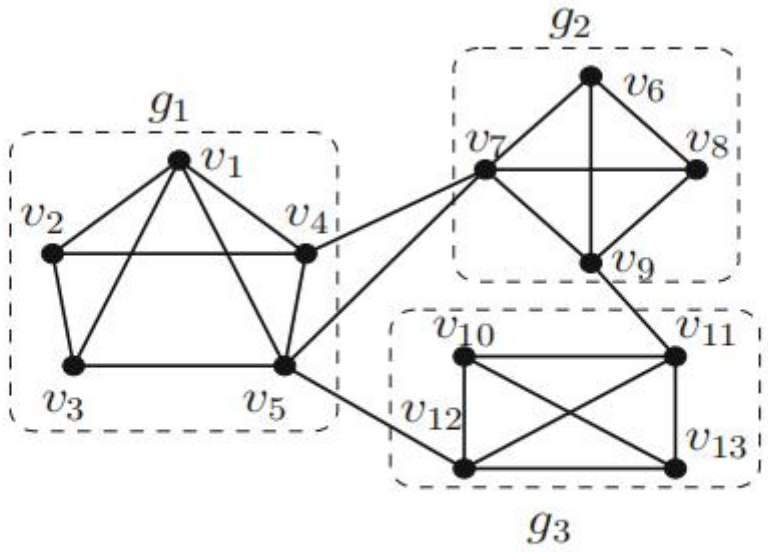



Fig. 6.1: A graph and its 3-edge connected components [17]

**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 𝒬 consisting of a single graph G;
2 while 𝒬 is not empty do
3     Pop a graph g from 𝒬;
4     𝒢_k ← Partition(g, k);
5     if 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 𝒢_k into 𝒬;
```

Partition函数是重点

# Higher-Order Densest Subgraph Computation

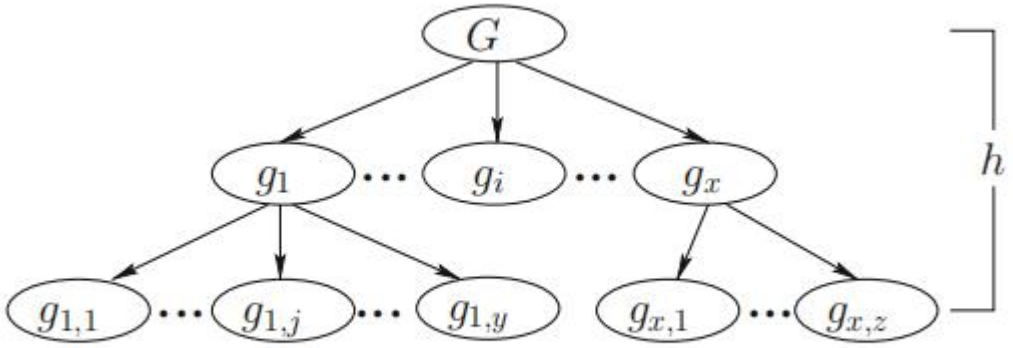



Fig. 6.2: A graph partition tree

**Algorithm 2:** Partition-TwoWay$(g,k)$

```
if g is k-edge connected then
    return {g};
else
    Compute a cut C of value smaller than k;
    Let g1, g2 be the two connected subgraphs obtained by removing C from g;
    return {g1, g2};
```

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

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

```
L ← {an arbitrary vertex of V(g)};
while |L| ≠ |V(g)| do
    Let u be the most tightly connected vertex to L, i.e., u = argmax_{v∈V(g)\L} ω(L, v);
    Add u to the tail of L;
Let s and t be the two vertices most recently added to L, and s is added to L prior to t;
return the cut (L\{t}, {t}) as a minimum s–t cut;
```

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

```
g' ← g;
while g' contains at least two vertices do
    Let (S,T) be the minimum s–t cut obtained by MAS(g');
    if ω(S,T) < k then return {g[S], g[T]};
    else Let g' be the resulting graph of contracting s and t into a super-vertex;
return {g};
```

# Higher-Order Densest Subgraph Computation

**gorithm 5:** Partition-DS$(g,k)$ [17]

```
f the minimum degree of g is smaller than k then
    Let u be a vertex of degree smaller than k;
    return {g\u, u};
D ← {arg max_{v∈V(g)} d(v)}; /* D contains the maximum-degree vertex
F ← {u ∈ V(g)\D | ∃v ∈ D, s.t., (u,v) ∈ E(g)};
while D∪F ≠ V(g) do
    Let u be the vertex with the maximum degree in g[V(g)\D];
    Compute a minimum cut (S,T) of g such that D ⊆ S and u ∈ T;
    if the value of (S,T) is smaller than k then
        return {g[S], g[T]};
    D ← D∪{u};
    F ← {u ∈ V(g)\D | ∃v ∈ D, s.t., (u,v) ∈ E(g)};
eturn {g};
```

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

```
1  g ← the k-core of g;
2  g' ← 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 ← {an arbitrary vertex u of V(g')};
7  for each v ∈ V(g') do ω(v) ← ω(L, v);
8  while |L| ≠ |V(g')| do
9      u ← arg max_{v∈V(g')\L} ω(v);
10     Add u to the tail of L, and initialize a queue Q with u;
11     while Q ≠ ∅ do
12         v ← pop a vertex from Q;
13         for each (v,w) ∈ E(g') with w ∉ L do
14             ω(w) ← ω(w) + 1;
15             if ω(w) = k then Push w into Q;
16         if u ≠ 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 ！