Community Models — $k$-core and $k$-truss
Basic requirements

• key requirements of a community:
  • densely or cohesive;
  • in case edges are weighted, representing some cost, want least cost communities.
  • connected, of course!
  • possibly additional constraints which we will motivate and discuss later.
  • remarks hold for community detection as well as search. Think detection for now.
What kind of density are we after?

- high min. degree of a node in a community? what does that mean when you want to find all communities in a graph?
- high average degree?
- high edge density (#edges/#possible edges)?
- None of them allows us to see a graph as a hierarchy of communities.
- ==> $k$-core and $k$-truss, for various $k$. 
k-core — some definitions

- Def. graph $G = (V,E)$. $\text{deg}(v) := \text{degree of node } v \text{ in } G$.
- $H = (V',E')$, $V' \subseteq V$ — an induced subgraph of $G$.
- $\text{deg}_H(v) := |\{u \in V' \mid (u,v) \in E'\}| = \text{degree of } v \text{ in } H$.
- $H$ is a $k$-core iff $\forall v \in H : \text{deg}_H(v) \geq k$. and $H$ is maximal w.r.t. this property.

Example:

Recall, a $k$-core is required to be maximal.

core number of $v = \max\{k \mid v \text{ is in some } k\text{-core}\}$.

What are the various cores in this graph?
**k-core — some definitions**

- **Def.** graph $G = (V,E)$. $\text{deg}(v) :=$ degree of node $v$ in $G$.
- An induced subgraph $H = (V',E')$, $V' \subseteq V$ — an induced subgraph of $G$.
- $\text{deg}_H(v) := |\{u \in V' \mid (u,v) \in E'\}| =$ degree of $v$ in $H$.
- $H$ is a **$k$-core** iff $\forall v \in H : \text{deg}_H(v) \geq k$.

**Example:**

What are the various cores in this graph?

- $k=1.$
- $k=2.$
- $k=3.$
k-core — some definitions

- Def. graph $G = (V, E)$. $\text{deg}(v) := \text{degree of node } v \text{ in } G$.
- $H = (V', E')$, $V' \subseteq V$ — an induced subgraph of $G$.
- $\text{deg}_H(v) := |\{u \in V' \mid (u, v) \in E'\}| = \text{degree of } v \text{ in } H$.
- $H$ is a $k$-core iff $\forall v \in H: \text{deg}_H(v) \geq k$.

Example:

What are the various cores in this graph?

Each $(k+1)$-core is contained in some $k$-core.
**$k$-core — some definitions**

- Def. graph $G = (V, E)$. $\deg(v) := \text{degree of node } v \text{ in } G$.
- $H = (V', E')$, $V' \subseteq V$ — an induced subgraph of $G$.
  
  $\deg_H(v) := |\{u \in V' \mid (u, v) \in E'\}| = \text{degree of } v \text{ in } H$.

  $H$ is a **$k$-core** iff $\forall v \in H : \deg_H(v) \geq k$.

**Example:**

What are the various cores in this graph?

Each $(k+1)$-core is contained in some $k$-core.

A $k$-core need not be connected. E.g., add another $l$-core to this graph, disconnected from it.
Some properties of \( k \)-cores

- every vertex in a \( k \)-core has degree at least \( k \).
- hierarchical structure facilitates viewing a complex graph at flexible level of detail: zoom in (increase \( k \)) or zoom out (decrease \( k \)).
- can control density/cohesiveness interactively.
- any two \( k \)-cores are disjoint from each other.
- connected components partition \( k \)-cores.
- core can be based on in-degree, out-degree, or both (in case of directed graph). we will mainly consider (undirected) graphs below.
- some dense subgraphs are NP-hard to find (e.g., maximum cliques), while core decomposition can be found efficiently.
Finding core decomposition efficiently

A naive algorithm.

Input: graph $G$.
Output: core decomposition of $G$.

$G$ is a $0$-core.
k=1;
repeat {
    \begin{itemize}
    \item (recursively) remove all vertices of degree $< k$;
    \item report resulting subgraph as a $k$-core;
    \item $k++$
    \end{itemize}
} until $G$ is empty
Remarks on the naive algorithm

• Multiple passes over vertices.

• considerable redundant work.

• Can be implemented efficiently to take $O(m+n)$ time where $n = \#\text{vertices}$ and $m = \#\text{edges}$.

• key ideas:
  
  • compute vertex degrees in $O(m)$ time.
  
  • sort vertices by degree using bucket sort — $O(n)$ time.
    
    • vertex degrees in $[0,n-1]$ — allocate bucket per degree.
Efficient implementation of core decomposition using bucket sort

Example:

Core numbers

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Buckets

<table>
<thead>
<tr>
<th>Bucket</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>b,f,g</td>
</tr>
<tr>
<td>2</td>
<td>d,i,j</td>
</tr>
<tr>
<td>3</td>
<td>c,e,h</td>
</tr>
</tbody>
</table>

Diagram:

- Nodes a, b, c, d, e, f, g, h, i, j
- Edges between nodes

- Bucket assignments based on core numbers:
  - Bucket 0: a
  - Bucket 1: b, f, g
  - Bucket 2: d, i, j
  - Bucket 3: c, e, h
Efficient implementation of core decomposition using bucket sort

Example:

```
1 1 3 3 4 2 2 4 3 3
a b c d e f g h i j
```

```
0
1 ... a,b
2 ... f,g
3 ... d,i,j,c
4 ... e,h
5
6
7
8
9
```
Efficient implementation of core decomposition using bucket sort

Example:

$$\begin{array}{cccccccc}
1 & 1 & 4 & 3 & 3 & 2 & 2 & 3 \\
\text{a} & \text{b} & \text{c} & \text{d} & \text{e} & \text{f} & \text{g} & \text{h} & \text{i} & \text{j}
\end{array}$$

Stop!

All vertices in remaining subgraph have degree = 3.
Key steps of Algorithm

• compute vertex degrees and assign them to buckets; set \( \text{core}[v] = \text{deg}[v] \); — one pass \( O(m) \) time.

• maintain \( \text{core}[v] = \) current degree of \( v \).

• process vertices in degree order: for current vertex \( v \), if \( (u,v) \in E \) & \( \text{core}(u) > \text{core}(v) \), then decrement \( \text{core}(u) \) and move \( u \) up by a bin.
k-core and core decomposition summary

- k-core is a maximal subgraph where all nodes have degree $\geq k$ in the subgraph.

- hierarchical structure: k-core is contained in (k-1)-core.

- naive core decomposition is inefficient.

- using appropriate data structures and bucket sort, can obtain core decomposition in $O(m+n)$ time and space.

- (k-)cores can be a basis for community definition.
\( k \)-truss — some definitions

- if neighbors \( u \) and \( v \) have a common neighbor \( w \), we can think of \( w \) “endorsing” friendship of \( u \) and \( v \) \( \rightarrow \) triangle \( \{u,v,w\} \).

- more common neighbors, more triangles, more endorsement.

- (triangle) support of edge \( e = (u,v) \), \( \text{sup}(e) = \# \) triangles that \( e \) participates in; \( \text{sup}_H(e) = \) support in subgraph \( H \).

- a subgraph \( H = (V',E') \) is a \( k \)-truss if \( \forall e \in E' : \text{sup}_H(e) \geq k - 2 \). and \( H \) is maximal w.r.t. this property.

- A \( k \)-truss is \( (k-1) \)-connected.
\(k\)-truss — some definitions

- *truss number of* \(v\) = \(\max \{k \mid \text{v is in some } k\text{-truss}\}\).
- *truss number of* \(e=(x,y)\) is \(\max \{k \mid e \text{ is in some } k\text{-truss}\}\).
- *core number sometimes called coreness and truss number trussness.*
Trusses illustrated

• Example 1:

\[
t, p_1, p_2, p_3, p_4, r_1, r_2, r_3, q, s_1, s_2, s_3, s_4, x_1, x_2
\]

- 2-truss
- 3-truss
- 4-truss

\[k_{\text{max}} = 4\]
Trusses illustrated

Def. For an edge $e$, $\tau(e) = \max \{ k | e \text{ is in } k\text{-truss} \}$.

$\Phi_k(e) = \{ e | \tau(e) = k \}. \leftarrow k\text{-class}.

Example 2: $k$-class is just a technical notion. A $k$-class need not be dense.

$\tau = 2$.

$\tau = 3$.

$\tau = 4$. 
Finding truss decomposition efficiently

A naive algorithm.
Input: graph $G$.
Output: core decomposition of $G$.
$G$ is a 2-truss.
k=3;
repeat {
  • (recursively) remove all edges with support $< k-2$;
  • report resulting subgraph as a $k$-truss;
  • $k++$
}
until $G$ is empty
Limitations of the naive algorithm

- shares many of the limitations of the naive algorithm for core decomposition.

- solution efficiency can be improved by similar ideas.

- Here is a sketch of an efficient implementation.

- note that a truss decomposition can be represented using k-classes, $2 \leq k \leq k_{max}$. 
Efficient algorithm for truss decomposition

\[ k \leftarrow 2; \Phi_k \leftarrow \{ \}; \]

compute support of edges and sort using bucket sort;

(*) while there is an edge with support \( \leq k-2 \)

let \( e=(u,v) \) have least support and let

\( \deg(u) \leq \deg(v) \) w.l.o.g.

for each \( w \) in \( N(u) \)

if \( (v,w) \) is in \( E_G \)  

use hashing

sup(u,w)--; sup(v,w)--;

reorder \( (u,w) \) and \( (v,w) \) based on support;  

add \( e \) to \( \Phi_k \); remove \( e \) from \( G \);  

Done \( O(m) \) times.

if \( G \) is non-empty

k++; go to (*);

return \( \Phi_k, 2 \leq k \leq k_{max} \).
Complexity

$O(m + n)$ space, by similar argument to that for core decomposition — easy to see.

Proof of $O(m^{1.5})$ time:

- support computation — $O(m^{1.5})$ time (see below for how).
- sorting edges on support — $O(m)$ time using bucket sort.
- red rectangle executed $(\deg(u) \cdot \#_{\geq}(u))$ times, where
  \[ \#_{\geq}(u) = \# \text{neighbors } v \text{ of } u \text{ with } \deg(v) \geq \deg(u). \]

Claim: For any node $u$, $\#_{\geq}(u) \leq \sqrt{2m}$. (tighter than in paper.)

Time complexity of algorithm follows from this:

\[ \sum_{u \in V_G} \deg(u) \cdot \#_{\geq}(u) \leq \sqrt{2} \cdot \sum_{u \in V_G} \deg(u) \cdot \sqrt{m} \in O(m^{1.5}). \]
Complexity

Proof of Claim: \[ \#_{\geq}(u) \leq \text{deg}(u). \]
\[ \#_{\geq}(u) \leq 2m/\text{deg}(u). \]

From these, we have \[ \#_{\geq}(u) \leq \min\{\text{deg}(u), 2m/\text{deg}(u)\}. \]

The RHS is maximized when \[ \text{deg}^2(u) = 2m. \]

\[ \implies \text{deg}(u) = \sqrt{2m}. \]
Computing edge support in $O(m^{1.5})$ time

**Algorithm** FastSupport;

1. Arrange the nodes of G in non-increasing degree order: $\text{deg}(u) > \text{deg}(v) \Rightarrow \text{visit}(u) < \text{visit}(v)$.
2. Initialize an empty array A of n adjacency arrays;
3. For each node v in increasing order of visit:
   3.1. For each u in $\text{N}(v)$ with $\text{visit}(u) > \text{visit}(v)$:
      3.1.1. For each w in $A[u] \cap A[v]$:
          $\text{sup}(v,u)++$; $\text{sup}(u,w)++$; $\text{sup}(v,w)++$;
      3.1.2. Add v to $A[u]$;

Illustrative example

Visit ordered adjacency array

b 1:
a 2:
e 3:
c 4:
d 5:

edge support tally
Illustrative example

Visit ordered adjacency array

- **b** 1:
  - a 2: b
  - e 3: b
  - c 4: b
  - d 5: b

edge support tally
Illustrative example

Visit ordered adjacency array

b  1:
a  2: b
e  3: b
c  4: b
d  5: b

edge support tally
(a,b): 1+1
(b,e): 1
(a,e): 1
(a,d): 1
(b,d): 1
Illustrative example

Visit ordered adjacency array

- **b**: 1
  - 1: 
- **a**: 2: b
- **e**: 3: b
- **c**: 4: b,e
- **d**: 5: b

Edge support tally

- (a,b): 1+1
- (b,c): 1
- (b,e): 1+1
- (c,e): 1
- (a,e): 1
- (a,d): 1
- (b,d): 1
Illustrative example

Visit ordered adjacency array

- b 1: 
  - a 2: b
  - e 3: b
  - c 4: b, e
  - d 5: b

edge support tally
- (a,b): 1+1  (b,c): 1
- (b,e): 1+1  (c,e): 1
- (a,e): 1
- (a,d): 1
- (b,d): 1
Illustrative example

Visit ordered adjacency array

- \textbf{b} 1:
  - a 2: b
  - e 3: b
  - c 4: b,e
  - d 5: b

Check: all edge supports correctly calculated.

edge support tally:
- (a,b): 1+1  (b,c): 1
- (b,e): 1+1  (c,e): 1
- (a,e): 1
- (a,d): 1
- (b,d): 1
Complexity

line 1: $O(n \log n)$ time.
line 2: trivial.
line 3: make a pass over all edges — $O(m)$.
  Using adjacency adjacency lists (which are sorted in visit order, by construction), can do this in time proportional to the size of the lists.
Claim: $|A[u]|$ is $O(\sqrt{m})$.
Intuition: Let $A(u) := \{v \in N(u) \mid \deg(v) \geq \deg(u)\}$. Notice $A[u]$ is a subset of $A(u)$. So, suffices to show $|A(u)|$ is $O(\sqrt{m})$. If $|A(u)|$ was bigger, then these would account for $\omega(m)$ edges, which is impossible.
Overall complexity of $O(m^{1.5})$ follows from this. □