Time Complexity: Recap

- Correct
  - amortized/average/expected vs worst case
- Simplified
  - If a >= b, O(a+b) -> O(a)
- Tight
  - time complexity to scan neighbors of a node u (adjacency list): O(deg(u))

Assignment 1 Review

Q4:

O(m)

Q5:

First-In-First-Out

2-Hop Recap: Reachability

$O(|L_{out}(u)| + |L_{in}(v)|)$: Precomputed order + merge join

Query Time Complexity: L is the average number of labels of a node
O(L)

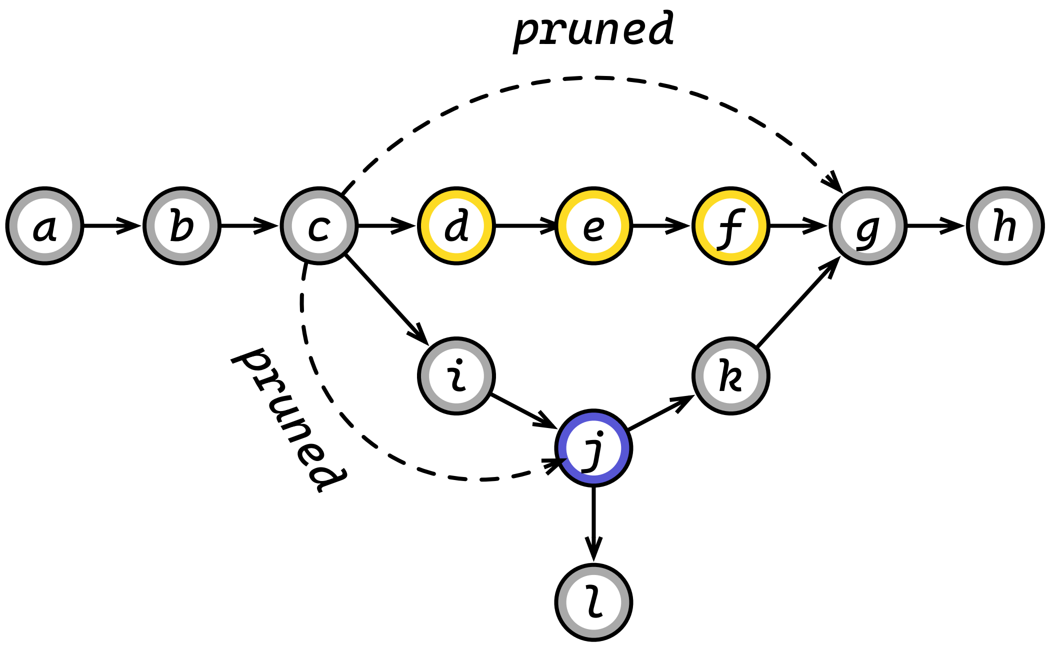
Index Space:
O(n*L)

2-Hop Recap: Reachability

For each node u in the graph from high-degree to low-degree: $O(n*m+n^2*L)$

- *add u into both $L_{in}(u)$ and $L_{out}(u)$;*
- ~~*mark u as processed;*~~
- *conduct BFS from u and for each reached node w:*
  - *if (u,w) has been covered or rank(w)<rank(u): stop exploring out-neighbors of w;*
  - *else: add u into $L_{in}(w)$;*
- *conduct reverse BFS from u and for each reached node w':*
  - *if (w',u) has been covered or rank(w')<rank(u): stop exploring in-neighbors of w';*
  - *else: add u into $L_{out}(w')$;*

2-Hop Recap: Reachability



1st round: process j

$L_{out}(a) = (j);$
$L_{out}(b) = (j);$
$L_{out}(c) = (j);$
$L_{out}(i) = (j);$
$L_{out}(j) = (j);$

$L_{in}(j) = (j);$
$L_{in}(k) = (j);$
$L_{in}(g) = (j);$
$L_{in}(h) = (j);$
$L_{in}(l) = (j);$

2nd round: process c

$L_{out}(a) = (j,c);$
$L_{out}(b) = (j,c);$
$L_{out}(c) = (j,c);$
$L_{out}(i) = (j);$
$L_{out}(j) = (j);$

$L_{in}(c) = (c);$
$L_{in}(d) = (c);$
$L_{in}(e) = (c);$
$L_{in}(f) = (c);$
$L_{in}(i) = (c);$
$L_{in}(j) = (j);$
$L_{in}(k) = (j);$
$L_{in}(g) = (j);$
$L_{in}(h) = (j);$
$L_{in}(l) = (j);$

Structural Diversity

Required knowledge from Topics 0—3

Feel free to discuss the question with your tutors …

Cohesive Subgraphs Detection

COMP9312_23T2

UNSW SYDNEY

Outline

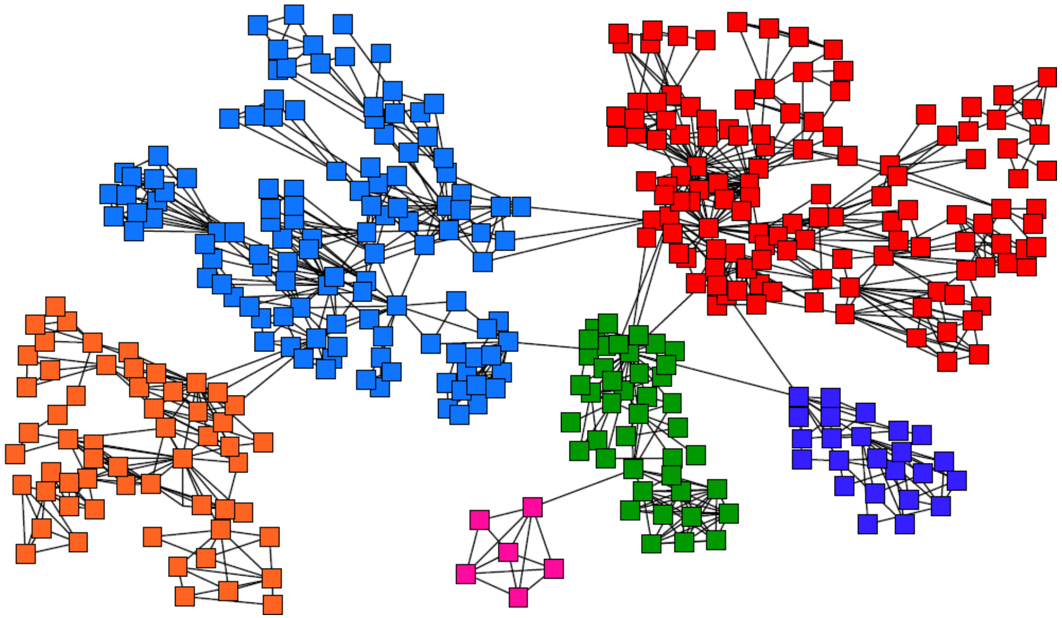
- Basic Concepts
- Applications
- Models and Algorithms

Community structure in graphs

<u>Community structure</u>: a **cohesive group** of nodes that are connected "more densely" to each other than to the nodes in other communities

- Within-group (intra-group) edges.
  High density

- Between-group (inter-group) edges.
  Low density

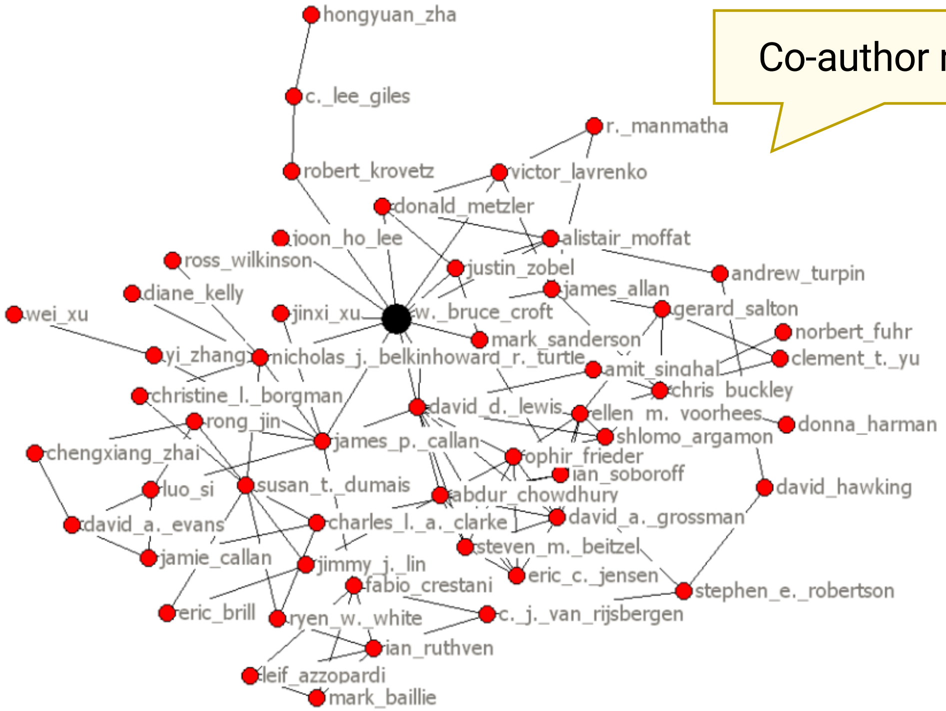


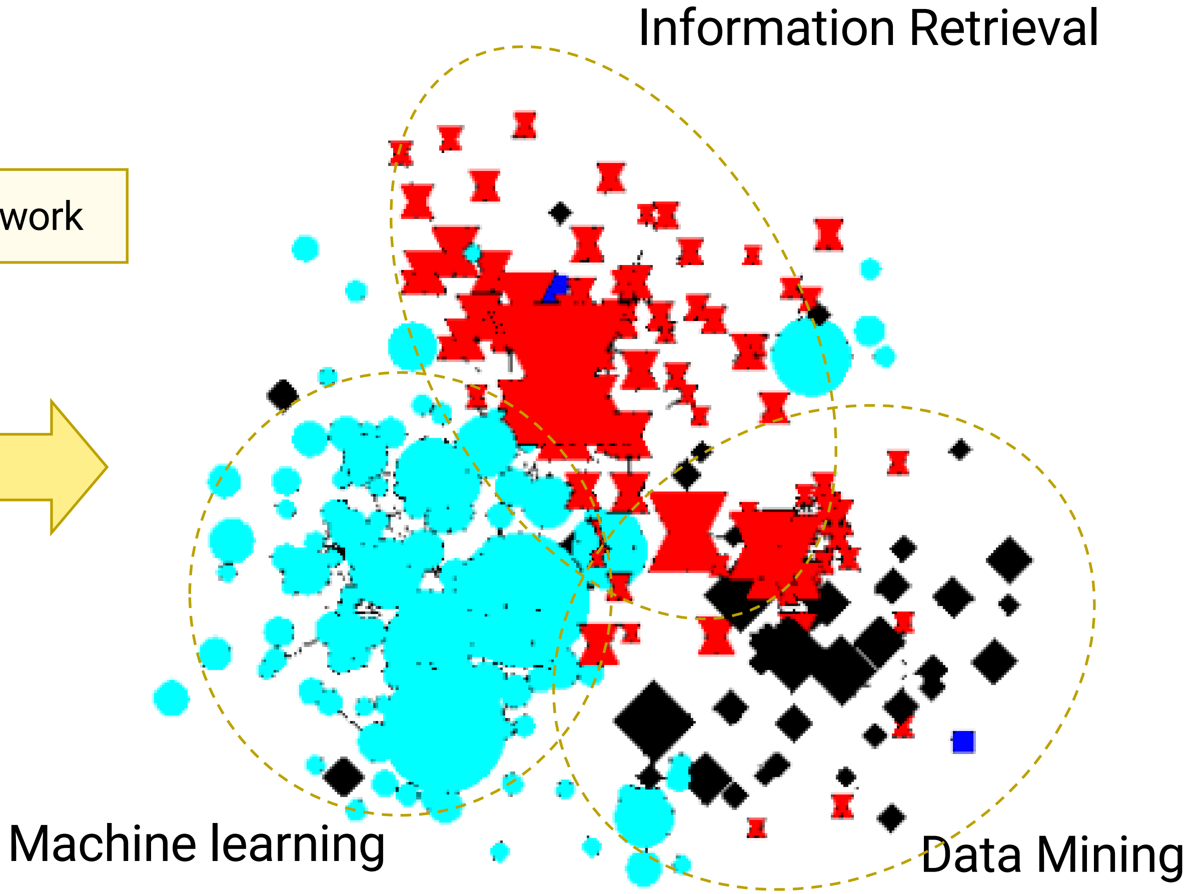
Community structures are quite common in real networks.

Finding Communities

Who tend to work together

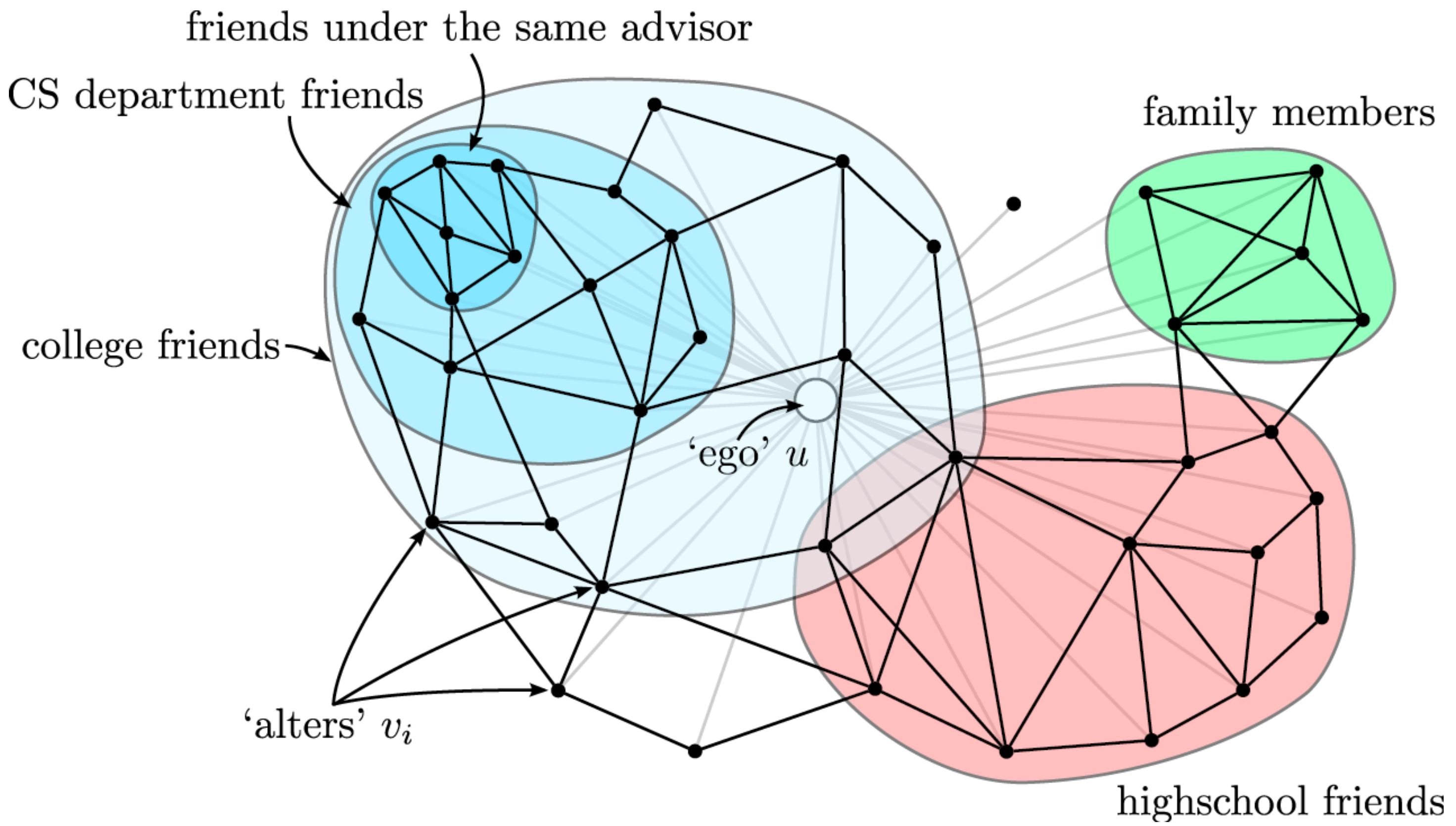
Co-author network





Q.Mei, D.Cai, D.Zhang, and C.Zhai, Topic Modeling with Hitting Time, WWW 2008

Finding Communities (Facebook or Twitter)



[McAuley, Leskovec: Discovering social circles in ego networks, 2012]

How to measure group cohesion?

- ❑ Mutuality of ties

- everybody in the group knows everybody else (**clique**)

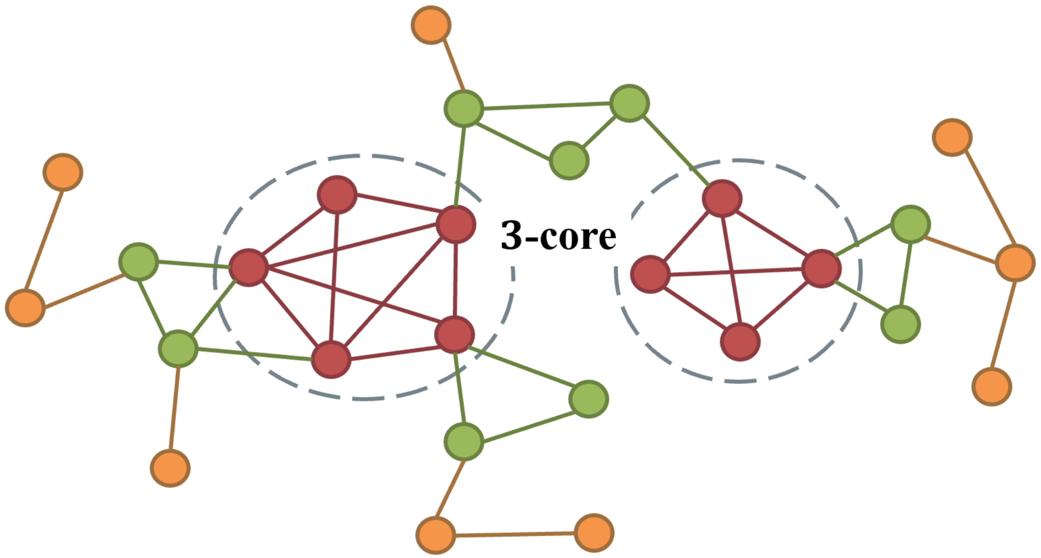
- ❑ frequency of ties among members

- everybody in the group has links to at least k others in the group (**k-core**)

- ❑ Others: density, quasi-clique, k-truss, k-edge connected component, etc.

Cohesive Subgraphs

In some models, a value k can be used to capture the cohesiveness of the subgraph. For example, k-core is a maximal subgraph in which each vertex has at least k neighbors in the subgraph.

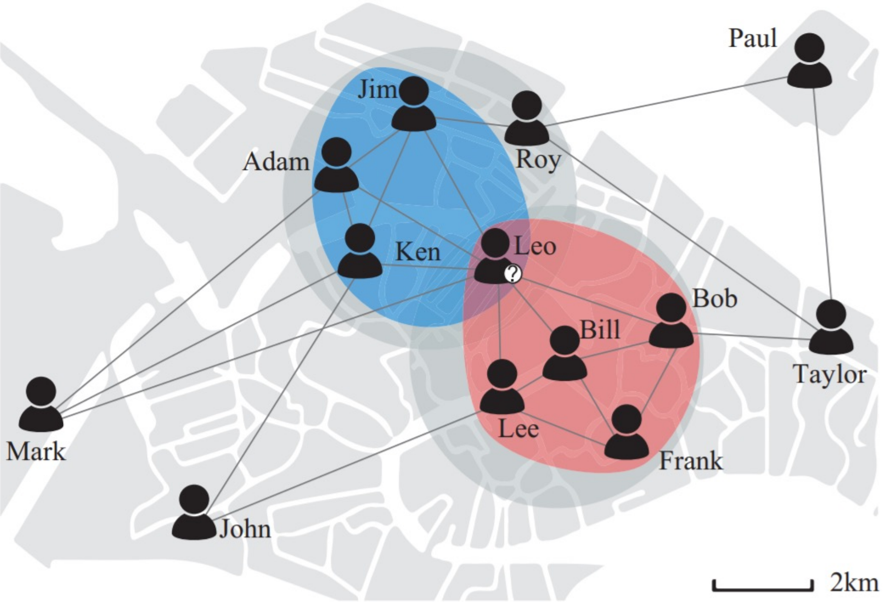


Application Summary

- Network Modeling and Analysis
- Network Visualization
- Reasoning the Collapse of a Network
- Discovering Influential Nodes
- Community Discovery
- Anomaly Detection
- Protein Function Prediction
- ……

Application – Community Discovery

<u>Social networks</u>: persistent community search [ICDE 2018], spatial community search [ICDE 2018], attributed community detection [VLDB 2017] [VLDB 2017] [VLDB 2018] , influential community search [VLDB 2015]

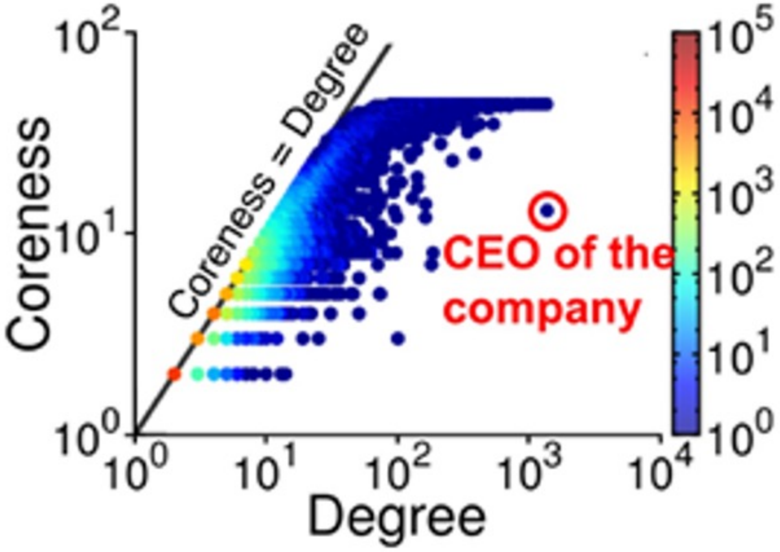


A geo-social network

Communities in Gowalla

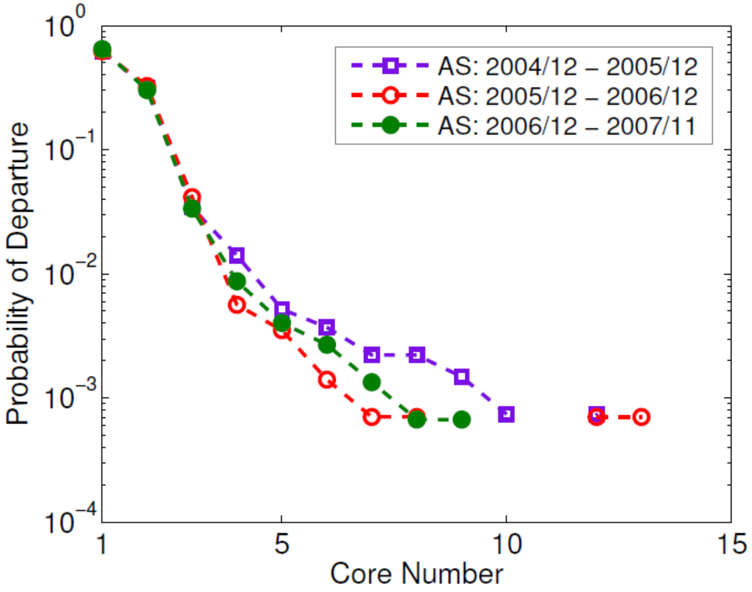
Application – Network Modeling and Analysis

Complex networks: pattern and anomaly analysis using k-core analysis [ICDM 2016] [KAIS 2018]

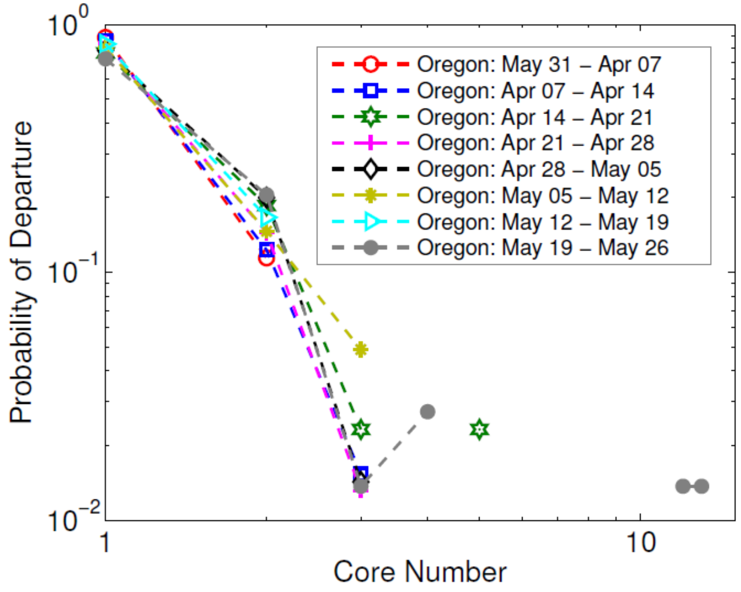


Application – Network Modeling and Analysis

Social networks: modeling engagement dynamics in social graphs [CIKM 2013] [Social Networks 1983]



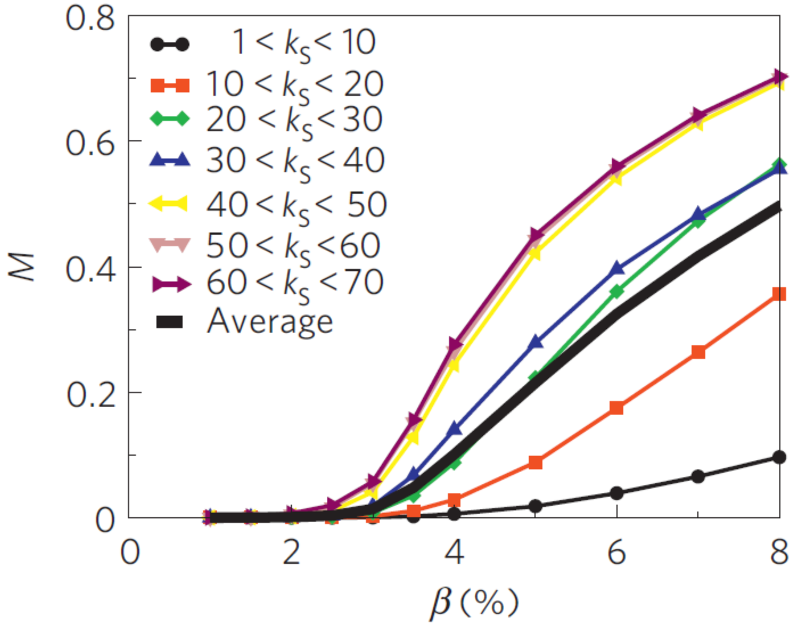
(a) CAIDA



(b) OREGON

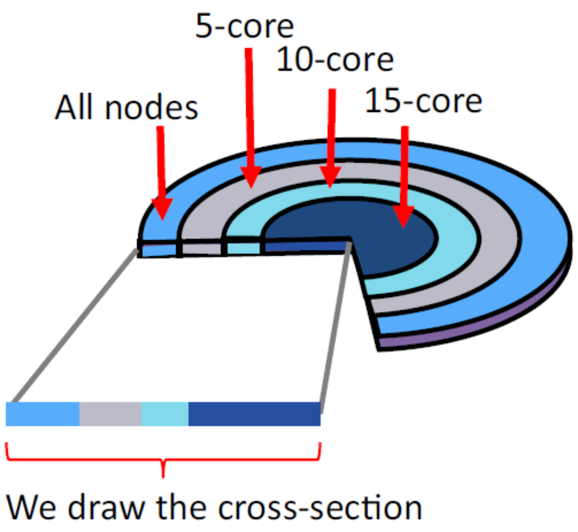
Application – Discovering Influential Nodes

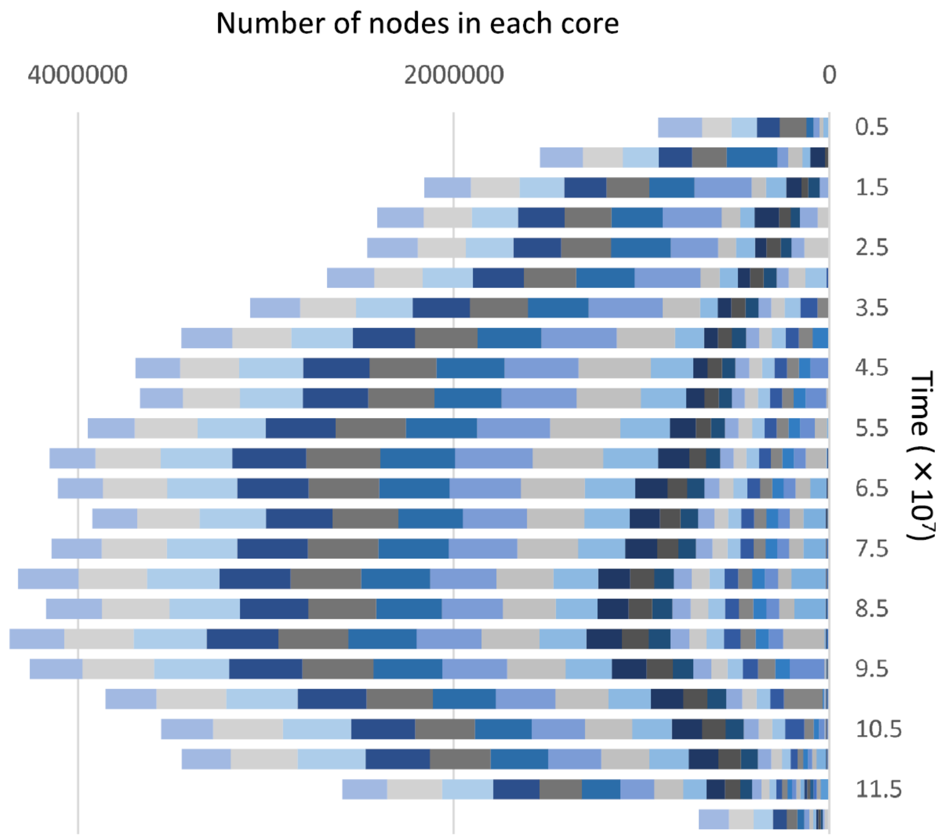
The most effective spreaders are located in the core of the network, fairly independent of their degree. Influence of the infection probability beta on the spreading efficiency M of nodes, grouped according to their *k*-shell values [Nature Physics 2010]



Application – Reasoning the Evolvement of a Social Network

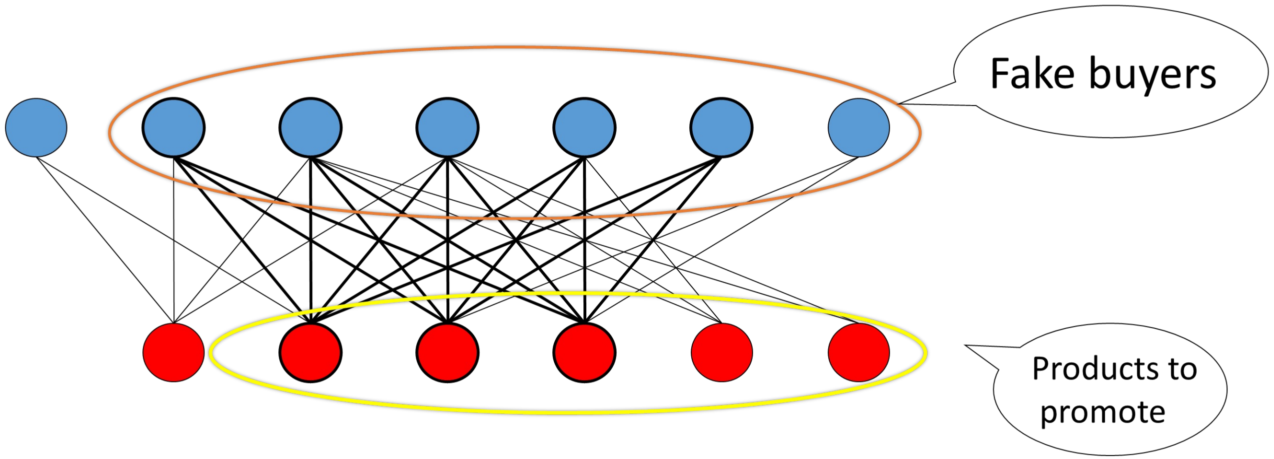
Friendster network: revealing the mechanism of collapse [SNAM 2017] [COSN 2013]





Application – Fraud Detection

User-item networks:



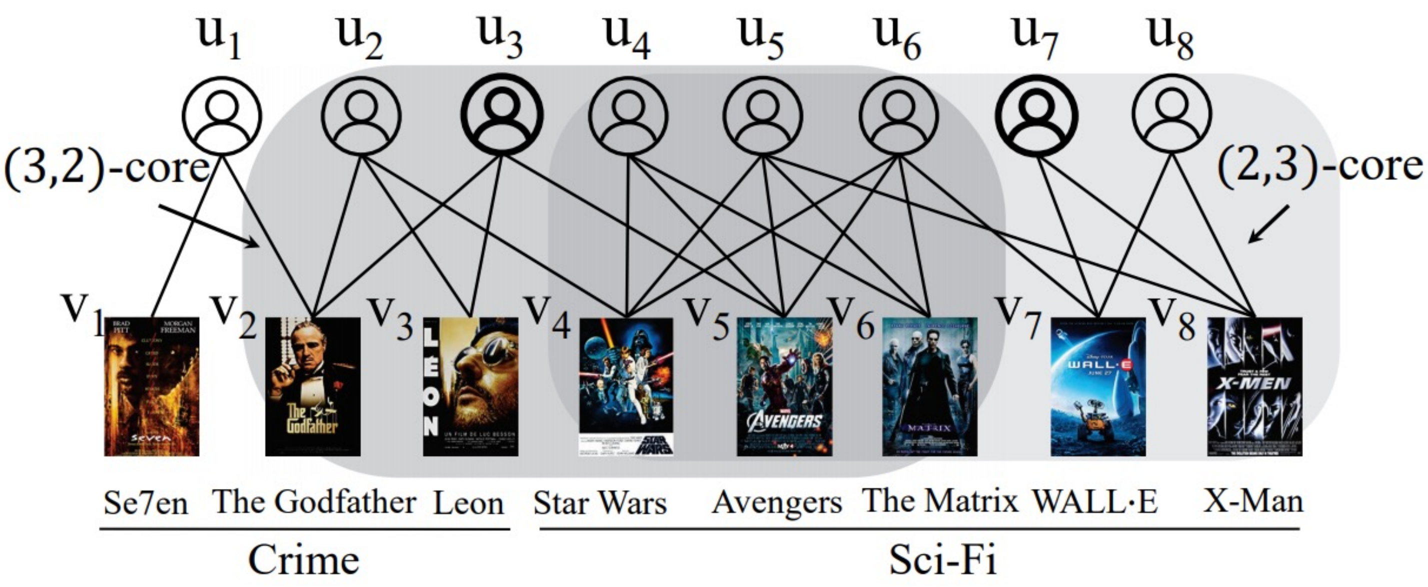
Challenges：
- Billions of buyers and productions, 10+ billions of transactions
- Dynamic data

Solution： Efficient biclique detection on bipartite graph
Outcome: Significantly increase the recall by 40% in double 11 festival in 2017

Lyu B, Qin L, Lin X, et al. **Maximum Biclique Search at Billion Scale**[J]. Proc. VLDB Endow., 2020, 13(9): 1359-1372. *[Best paper runner-up award in VLDB 2020]*

Application –Group Recommendation

Customer-movie network:

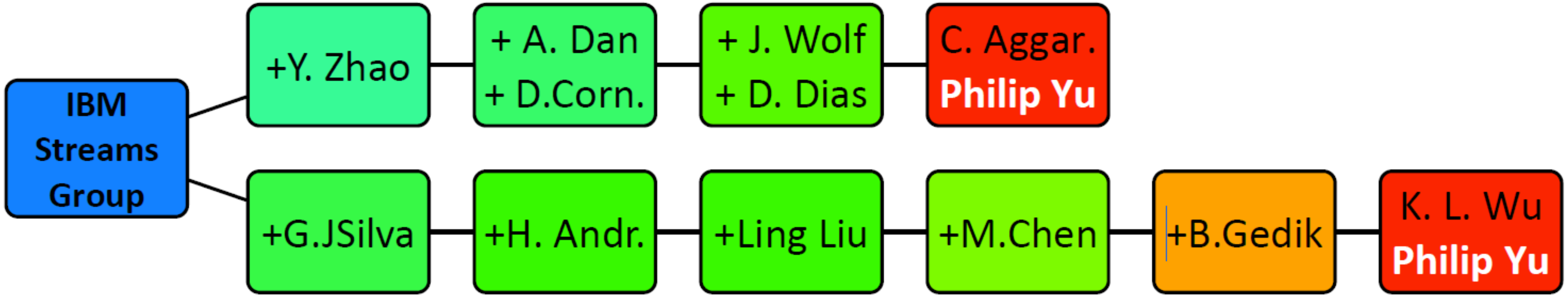


Liu B, Yuan L, Lin X, et al. Efficient (α, β)-core computation: An index-based approach[C], CIKM. 2019: 1130-1141.

Ding D, Li H, Huang Z, et al. Efficient fault-tolerant group recommendation using alpha-beta-core[C] CIKM

Application – Team Formation

<u>Author-paper networks</u>: analyzing the relationships between groups of collaborators in the same institution



Sarıyüce A E, Pinar A. Peeling bipartite networks for dense subgraph discovery[C]//Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining. 2018: 504-512.

More Applications

- **Graph clustering**

  Giatsidis, Christos, et al. "Corecluster: A degeneracy based graph clustering framework." AAAI. 2014.

- **Graph similarity**

  Nikolentzos, Giannis, et al. "A Degeneracy Framework for Graph Similarity." *IJCAI*. 2018.

- **Community evaluation**

  Giatsidis, Christos, Dimitrios M. Thilikos, and Michalis Vazirgiannis. "Evaluating cooperation in communities with the k-core structure." ASONAM, 2011.

- **Influence maximization**

  Elsharkawy, Sarah, et al. "Effectiveness of the k-core nodes as seeds for influence maximisation in dynamic cascades." *International Journal of Computers* 2 (2017).

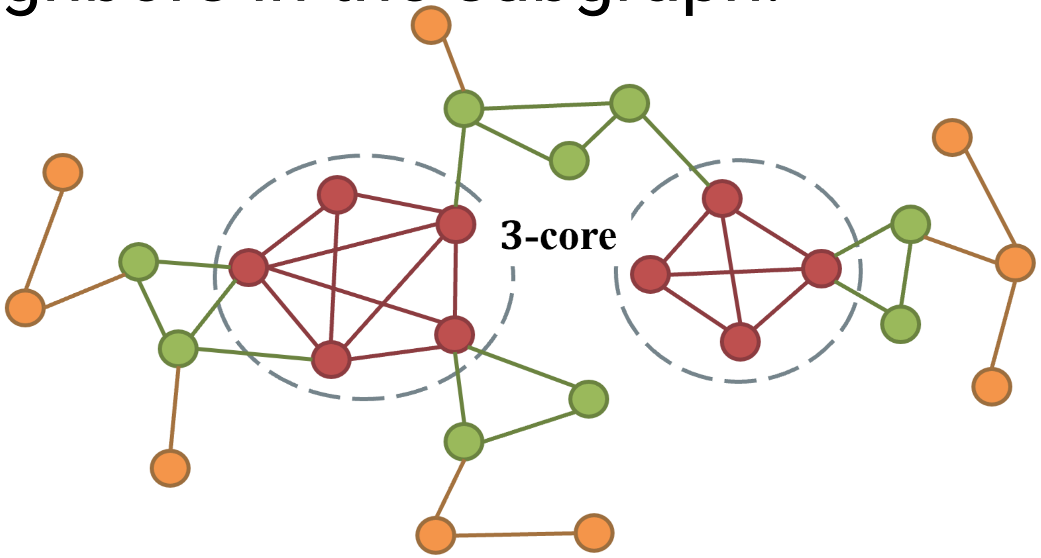
- **Graph generating**

  Baur, Michael, et al. "Generating graphs with predefined k-core structure." *Proceedings of the European Conference of Complex Systems*. 2007.

Degree-based Models: K-Core and its Variants

K-Core

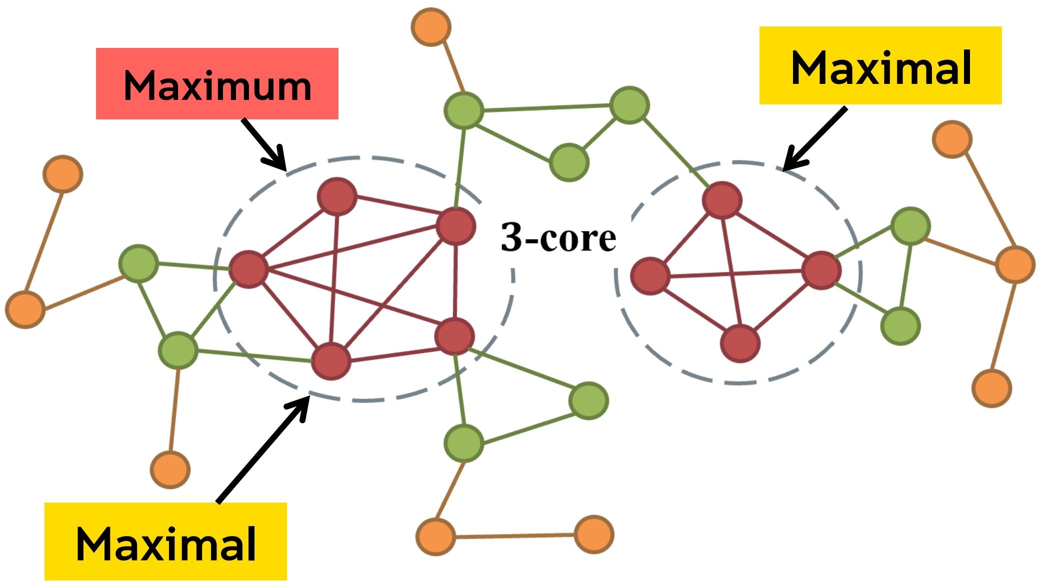
K-core is a maximal connected subgraph in which each vertex has at least k neighbors in the subgraph.



Erdős, Paul, and András Hajnal. "**On chromatic number of graphs and set-systems.**" Acta Mathematica Hungarica 17.1-2 (1966): 61-99.

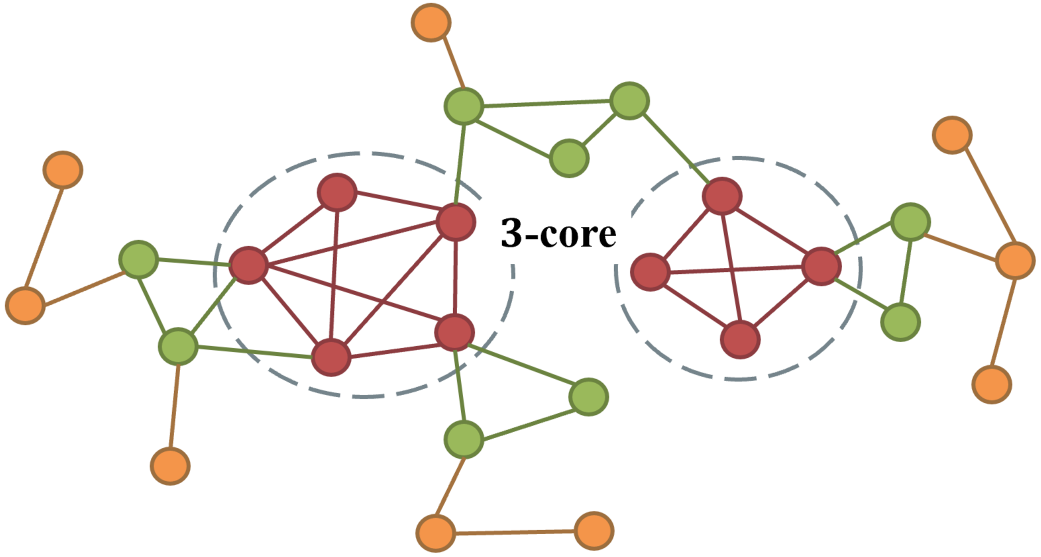
K-Core

Maximal subgraph is the once that cannot include more vertices.
Maximum subgraph is the one with the most vertices.



Graph Degeneracy (k_{max})

The largest k such that the k-core is not empty, which is also called the degeneracy.



Computing k-Core

Iteratively remove every vertex whose degree is less than k.

$O(m+n)$

Algorithm : ComputeCore(G, k)

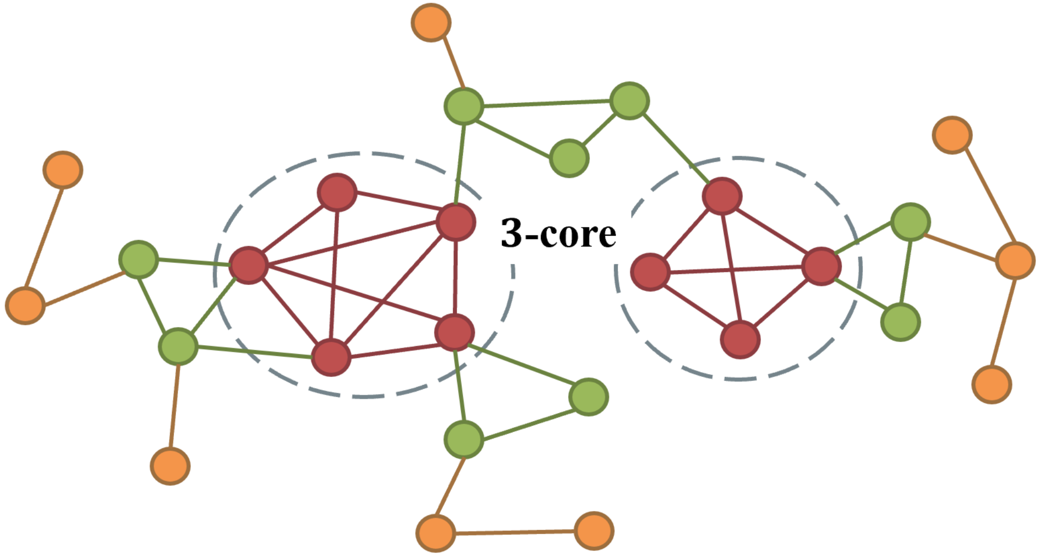
Input : G : a graph, k : degree constraint

Output : $C_k(G)$

while exists $v \in G$ with $deg(v, G) < k$ **do**

  $G \leftarrow G \setminus \{v \cup E(v, G)\}$;

return G



Computing k-Core

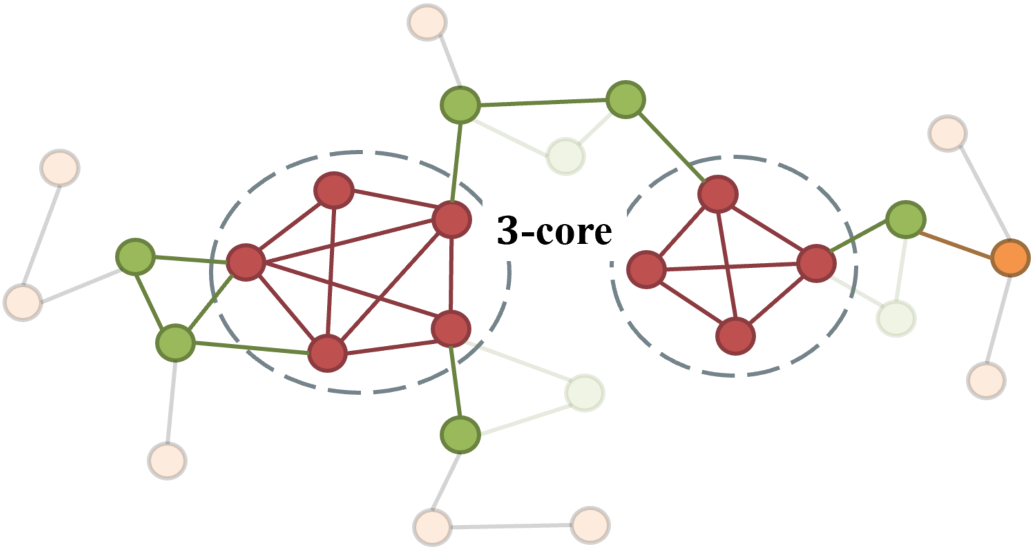
Iterately remove every vertex whose degree is less than k.

$O(m+n)$

Algorithm : ComputeCore(G, k)

Input : G : a graph, k : degree constraint
Output : $C_k(G)$

while exists $v \in G$ with $deg(v, G) < k$ **do**
  $G \leftarrow G \setminus \{v \cup E(v, G)\}$;
return G



Computing k-Core

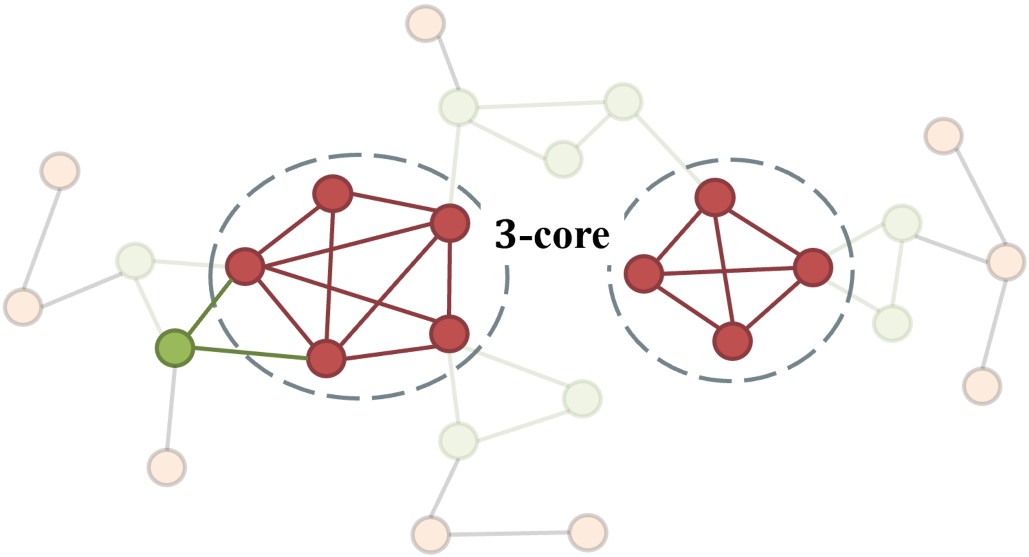
Iteratively remove every vertex whose degree is less than k.

$O(m+n)$

Algorithm : **ComputeCore**(G, k)

Input : G : a graph, k : degree constraint
Output : $C_k(G)$

```
while exists v ∈ G with deg(v, G) < k do
    G ← G \ {v ∪ E(v, G)};
return G
```



Computing k-Core

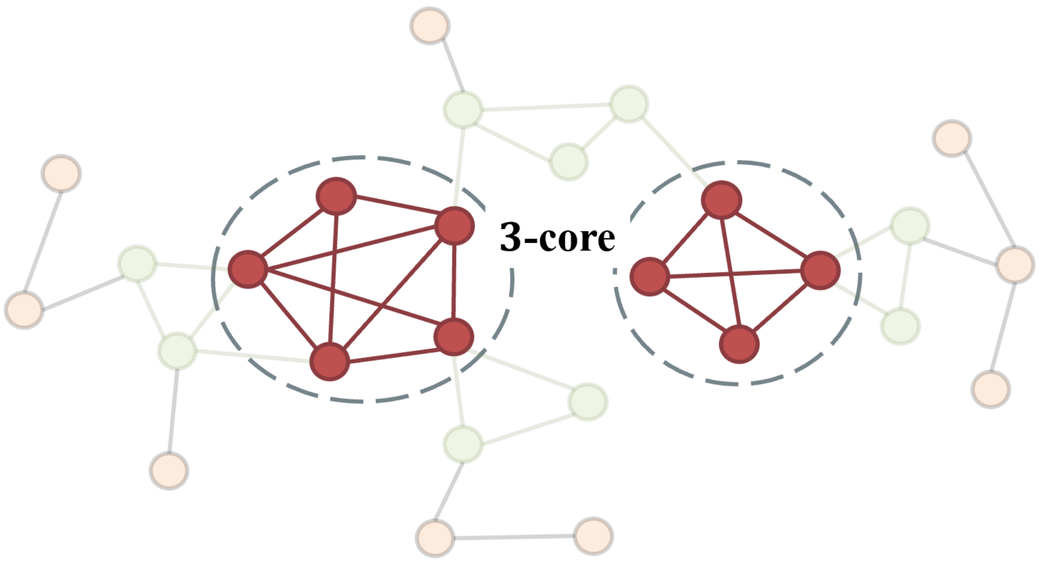
Iteratively remove every vertex whose degree is less than k.

$O(m+n)$

Algorithm : **ComputeCore**(G, k)

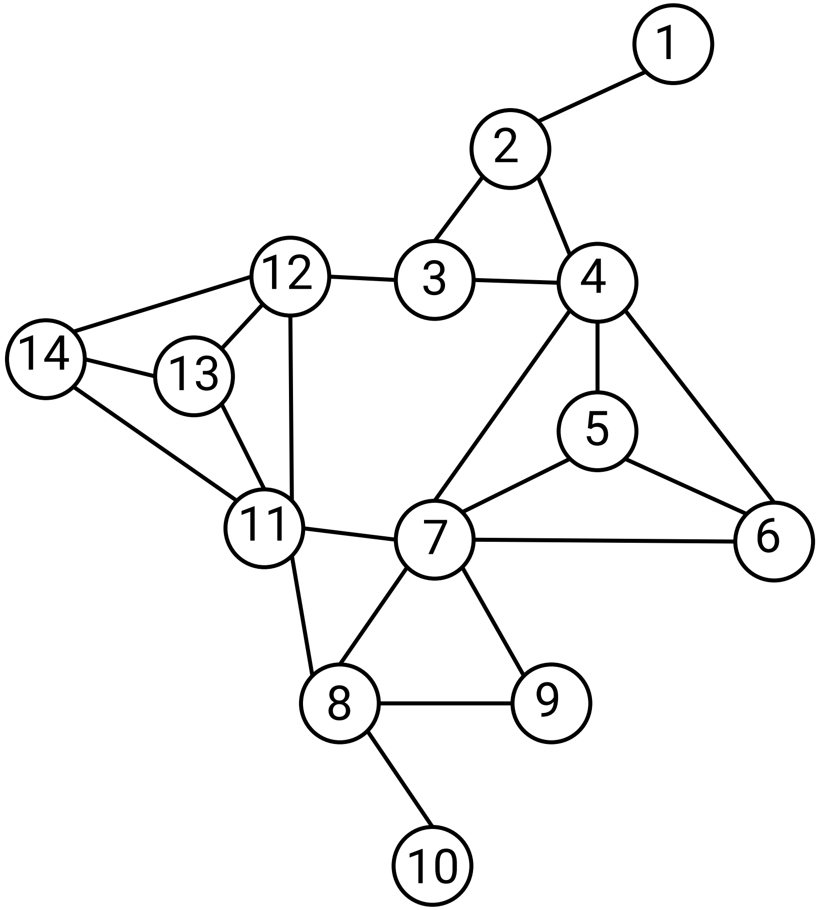
Input : G : a graph, k : degree constraint
Output : $C_k(G)$

```
while exists v ∈ G with deg(v, G) < k do
    G ← G \ {v ∪ E(v, G)};
return G
```



Example: Online k-core computation

Iteratively remove every vertex whose degree is less than k.



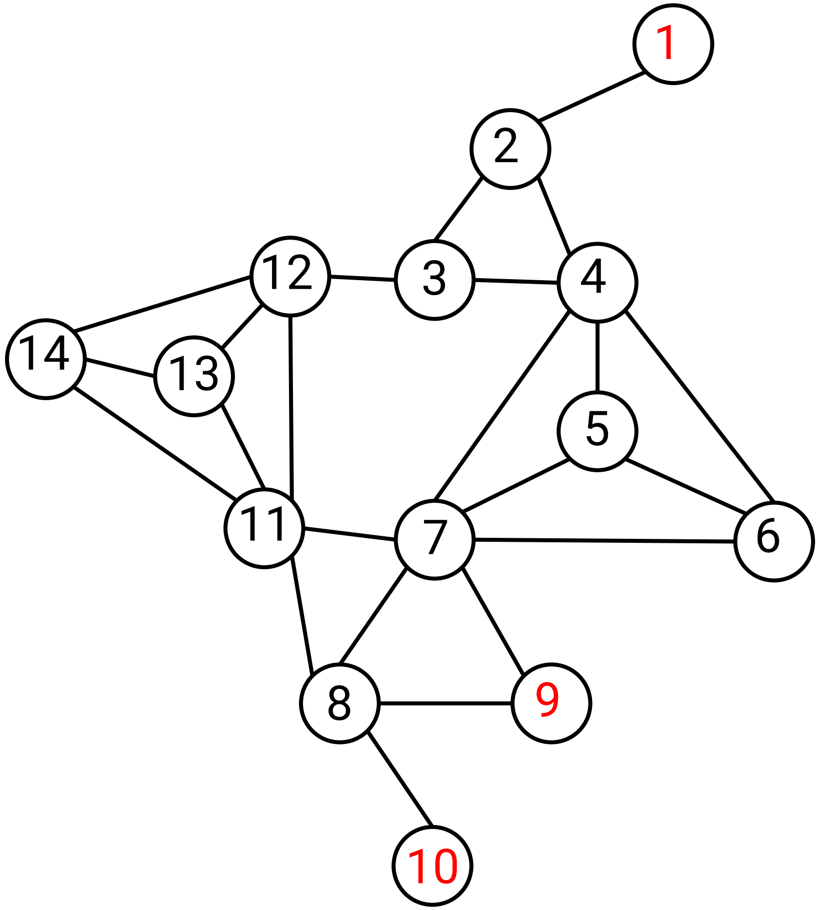
This table lists the status of the vertices:

vertex	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Delete	F	F	F	F	F	F	F	F	F	F	F	F	F	F

Find 3-core in this graph:

Example: Online k-core computation

Iteratively remove every vertex whose degree is less than k.



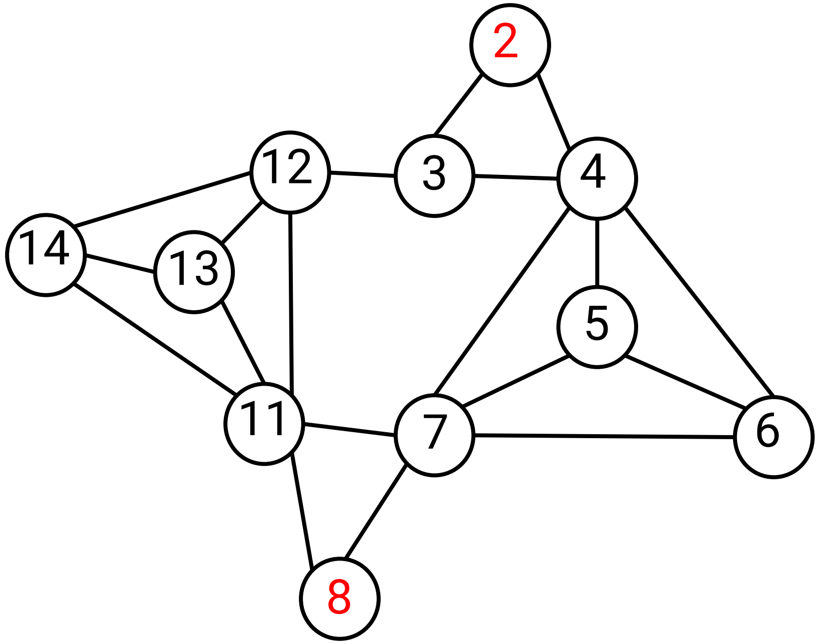
deg(v1)=1, deg(v9)=2, deg(v10)=1.
The degree of vertex 1, 9, 10 are smaller than 3, so we need to delete them.

vertex	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Delete	T	F	F	F	F	F	F	F	T	T	F	F	F	F

Find 3-core in this graph:

Example: Online k-core computation

Iterately remove every vertex whose degree is less than k.



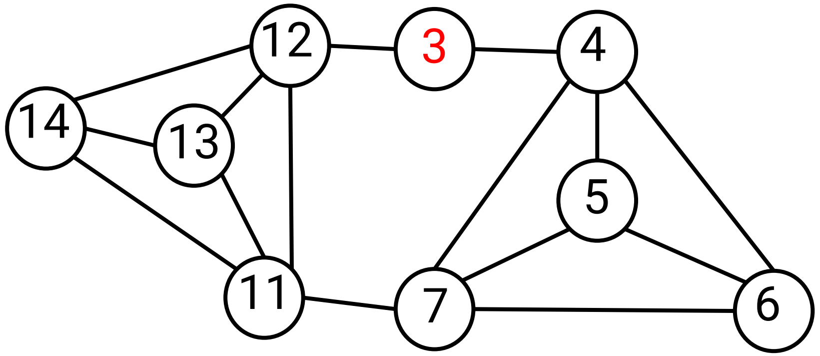
deg(v2)=2, deg(v8)=2.
The degree of vertex 2, 8 is smaller than 3, so we need to delete them.

vertex	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Delete	T	T	F	F	F	F	F	T	T	T	F	F	F	F

Find 3-core in this graph:

Example: Online k-core computation

Iteratively remove every vertex whose degree is less than k.



deg(v3)=2.
The degree of vertex 3 is smaller than 3, so we need to delete it.

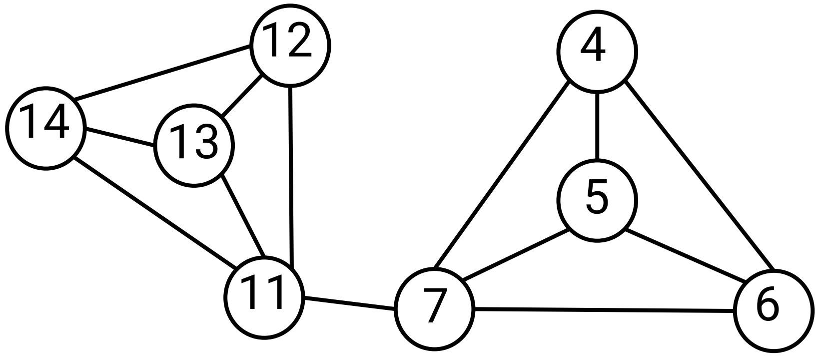
vertex	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Delete	T	T	T	F	F	F	F	T	T	T	F	F	F	F

Find 3-core in this graph:

Example: Online k-core computation

Iteratively remove every vertex whose degree is less than k.

The degree of all vertices is greater than 3, so the iteration is end. Then we get the result of 3-core.

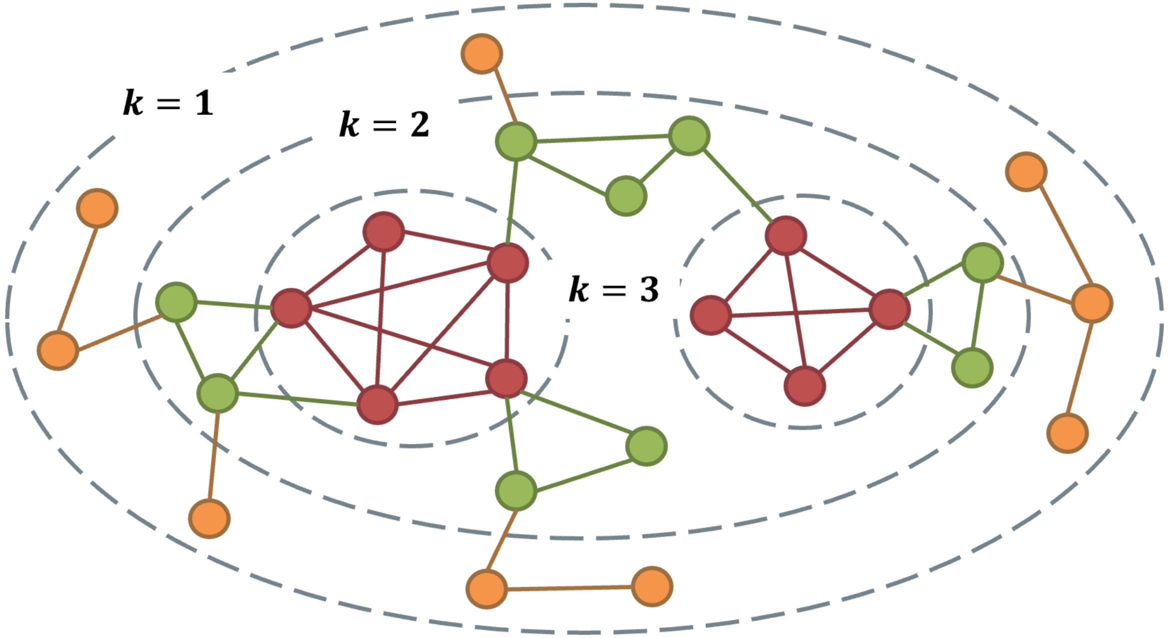


vertex	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Delete	T	T	T	F	F	F	F	T	T	T	F	F	F	F

Find 3-core in this graph:

Core Number

$k(v)$ = the largest k such that the k-core contains v

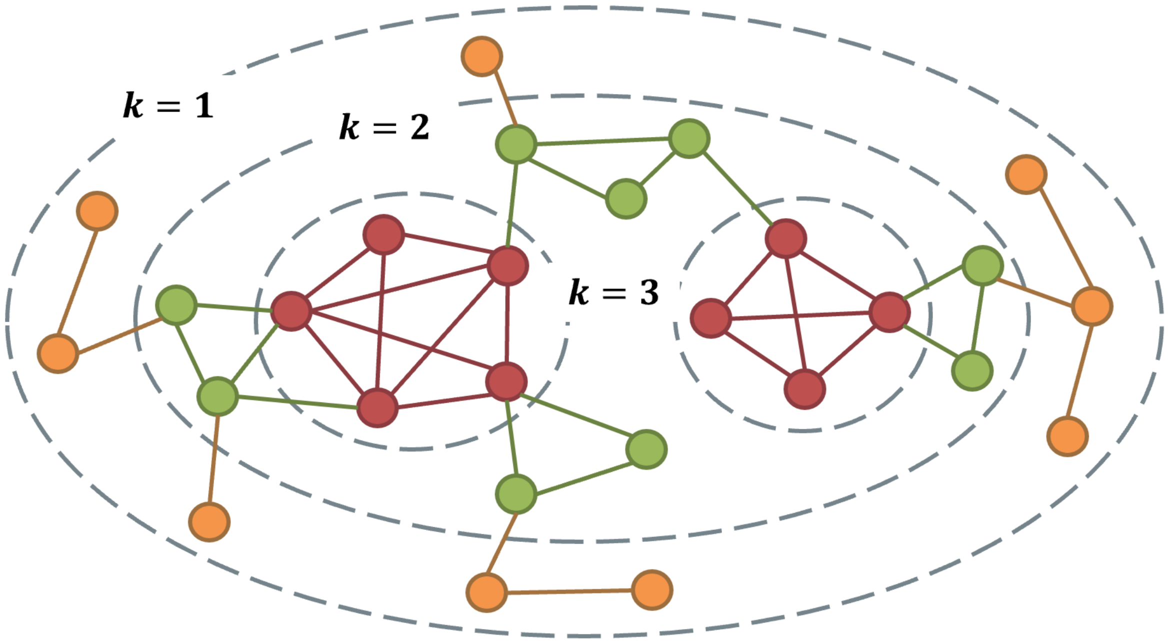


Core number of a vertex v:

Core Decomposition

Compute the core number of every vertex.

Note: If we store the core number of every vertex offline, given a graph G and a parameter k, all the vertices in the k-core (denoted as $C_k(V, E)$) of G can be returned in $O(|V(C_k)|)$ time.

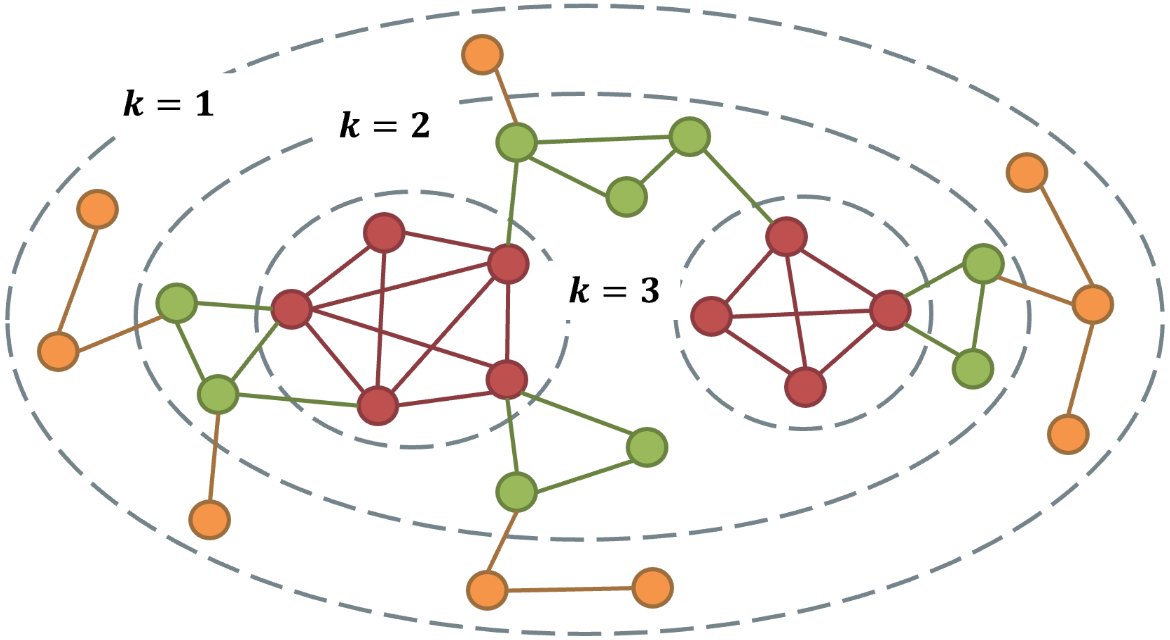


Tips: Store the vertices in decreasing order of their core numbers.

Core Decomposition (Cont.)

How to identify different groups?

There are two 3-cores in the example graph.



Core Decomposition: Methods

Global-view: peel low-degree vertices iteratively from the whole graph (which we introduce here).

Local-view: update the upper bound of core number for each vertex until converge.

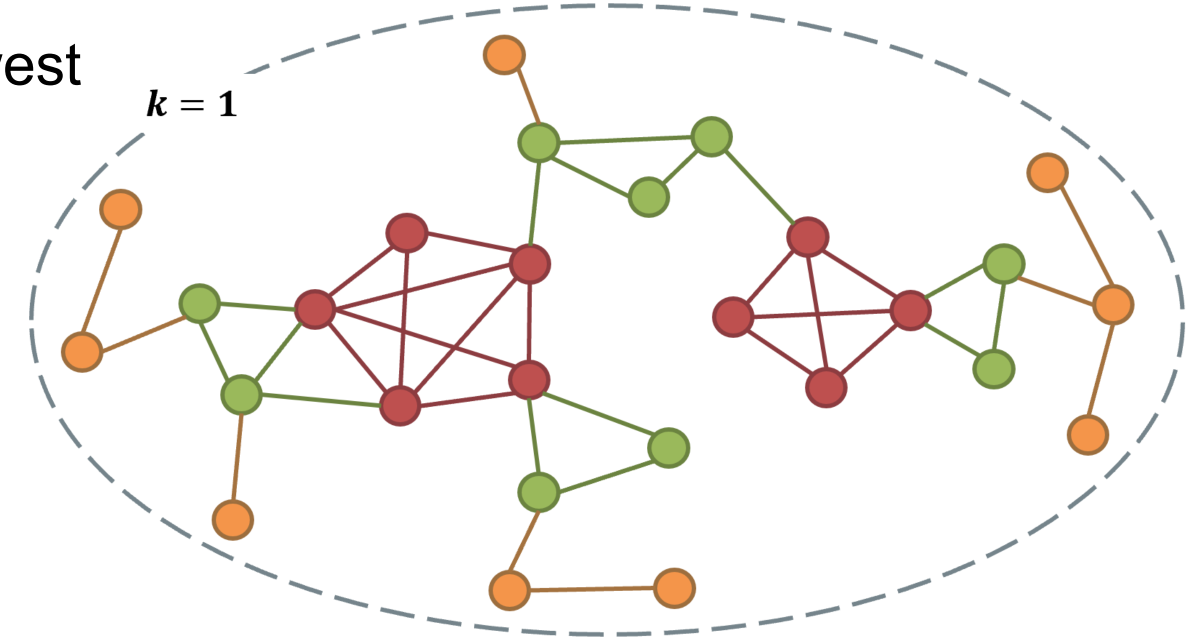
Core Decomposition: Global-view (Peeling)

For each unvisited vertex u with the lowest degree in G

assign core(u) as degree(u);

mark u as visited;

decrease the degree of its unvisited neighbors with higher degree than u by 1;



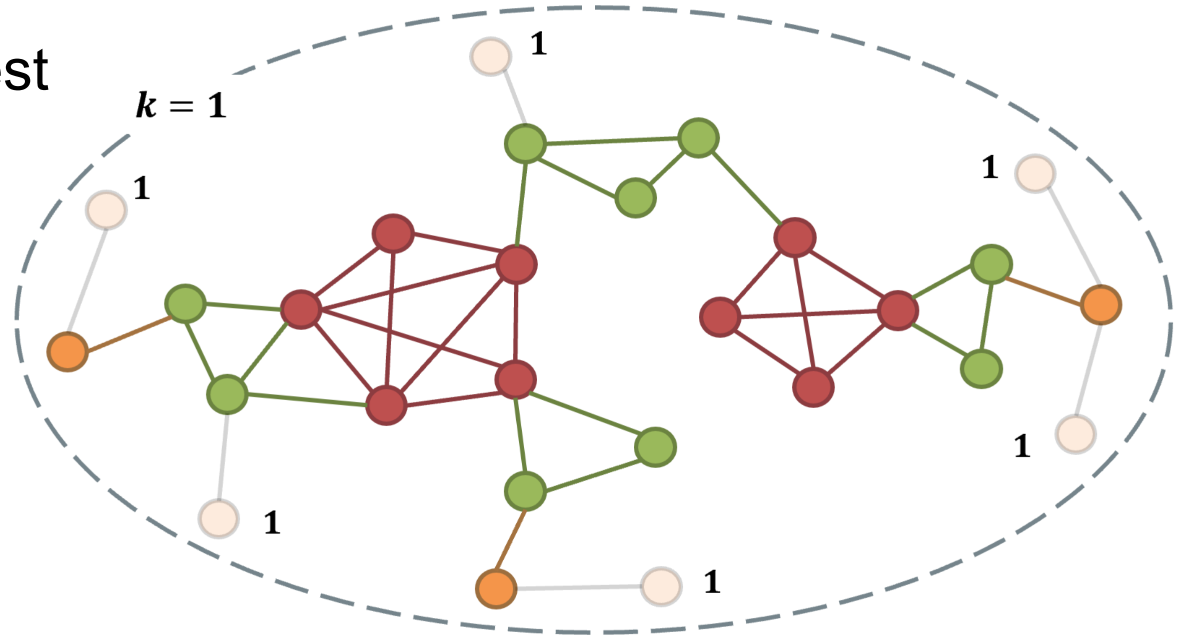
Core Decomposition: Global-view (Peeling)

For each unvisited vertex u with the lowest degree in G

assign core(u) as degree(u);

mark u as visited;

decrease the degree of its unvisited neighbors with higher degree than u by 1;



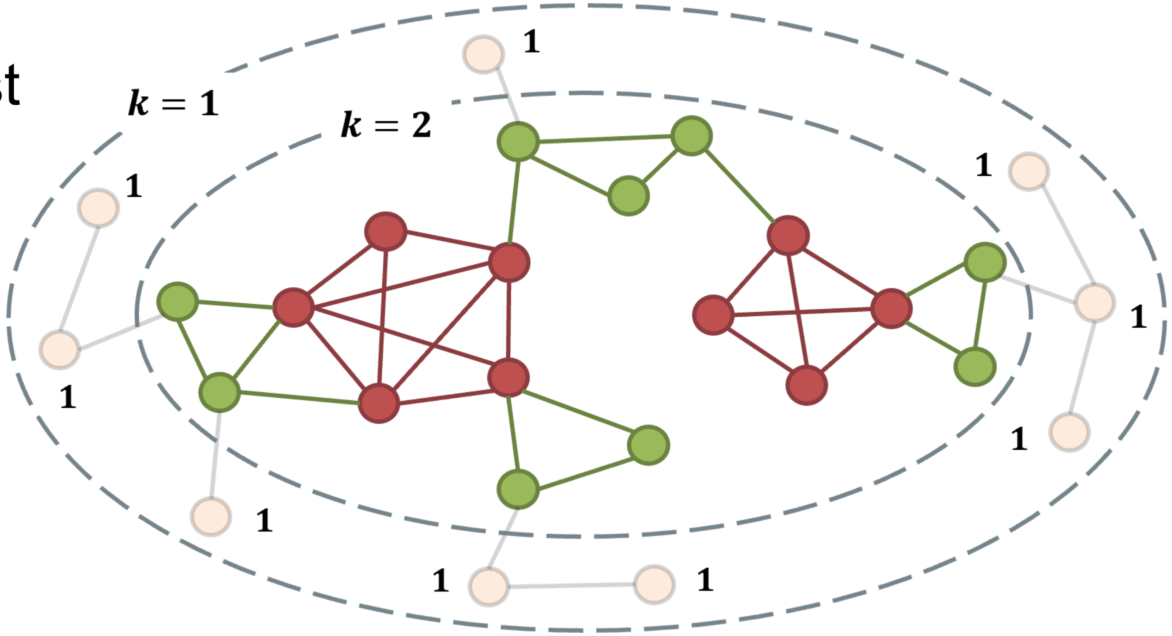
Core Decomposition: Global-view (Peeling)

For each unvisited vertex u with the lowest degree in G

assign core(u) as degree(u);

mark u as visited;

decrease the degree of its unvisited neighbors with higher degree than u by 1;



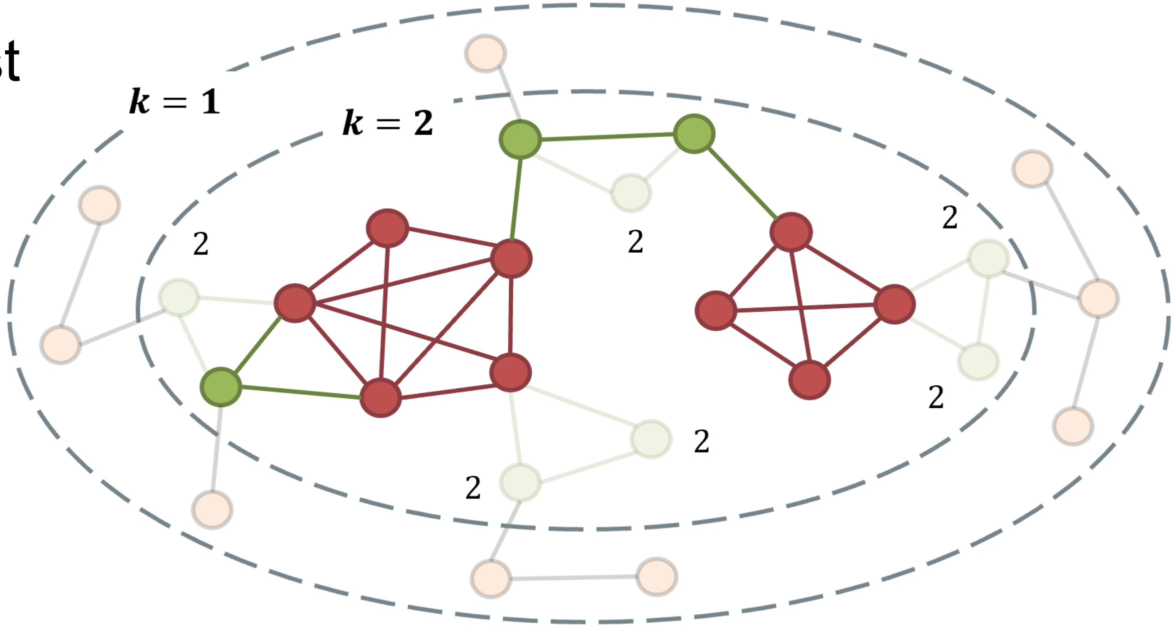
Core Decomposition: Global-view (Peeling)

For each unvisited vertex u with the lowest degree in G

assign core(u) as degree(u);

mark u as visited;

decrease the degree of its unvisited neighbors with higher degree than u by 1;



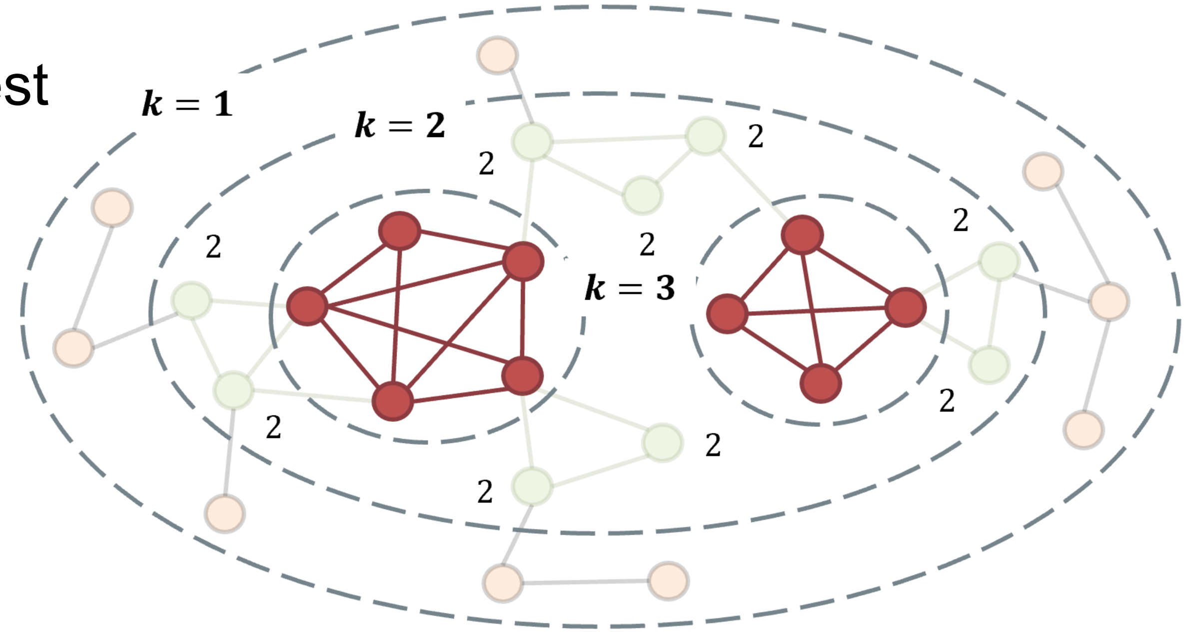
Core Decomposition: Global-view (Peeling)

For each unvisited vertex u with the lowest degree in G

assign core(u) as degree(u);

mark u as visited;

decrease the degree of its unvisited neighbors with higher degree than u by 1;



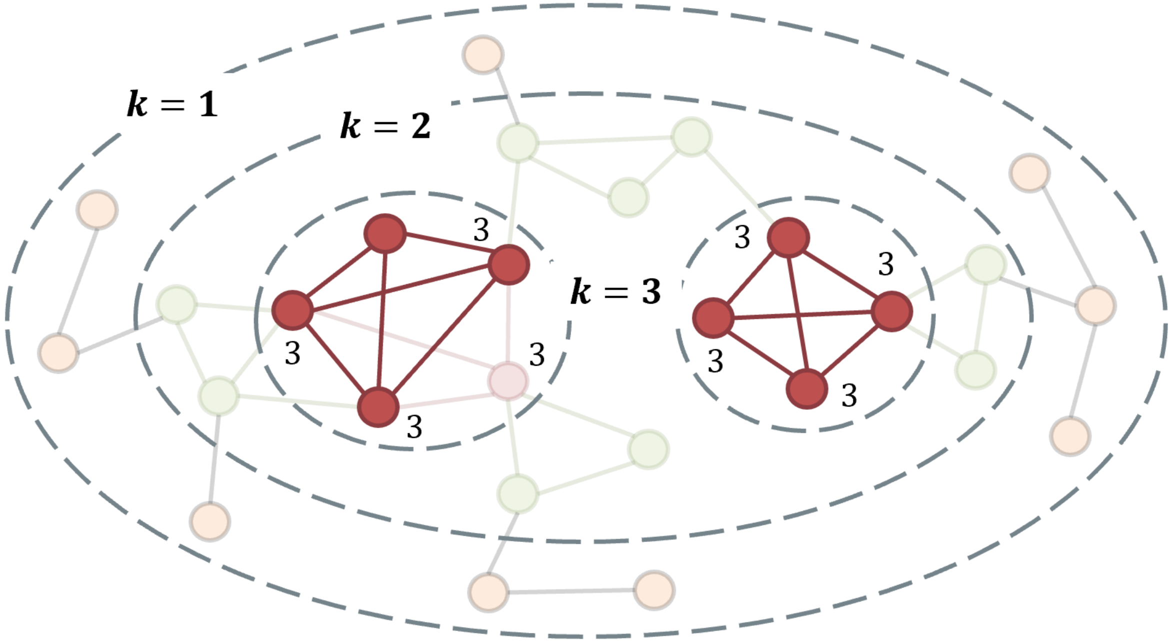
Core Decomposition: Global-view (Peeling)

For each unvisited vertex u with the lowest degree in G

assign core(u) as degree(u);

mark u as visited;

decrease the degree of its unvisited neighbors with higher degree than u by 1;



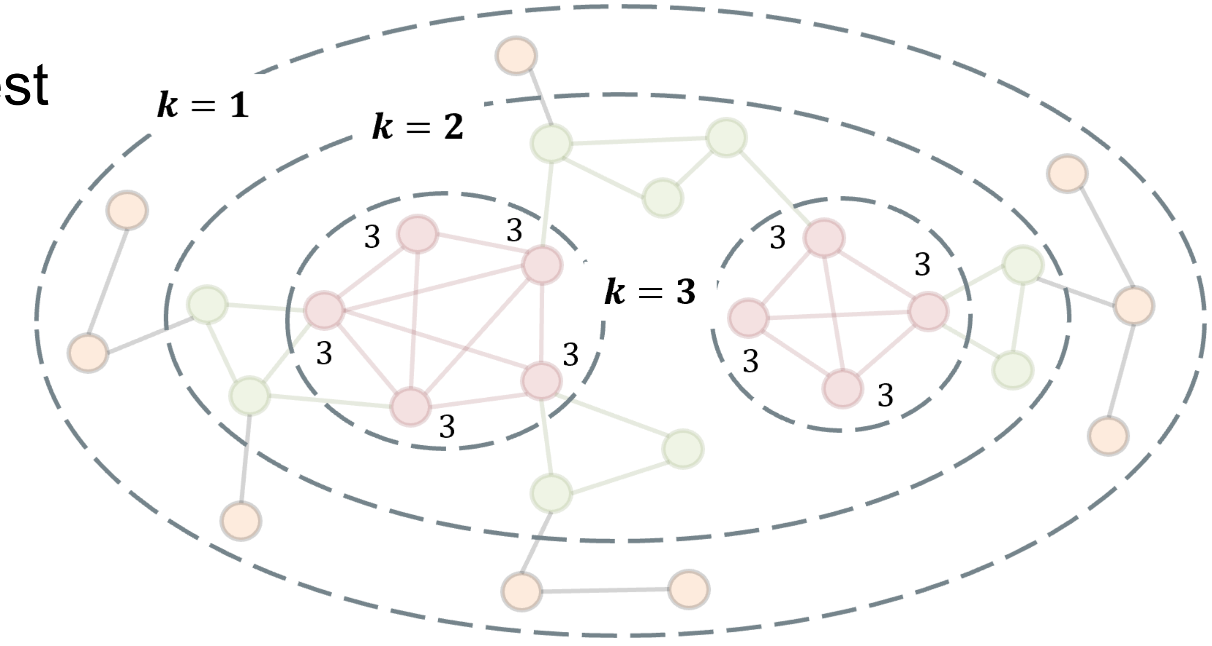
Core Decomposition: Global-view (Peeling)

For each unvisited vertex u with the lowest degree in G

assign core(u) as degree(u);

mark u as visited;

decrease the degree of its unvisited neighbors with higher degree than u by 1;



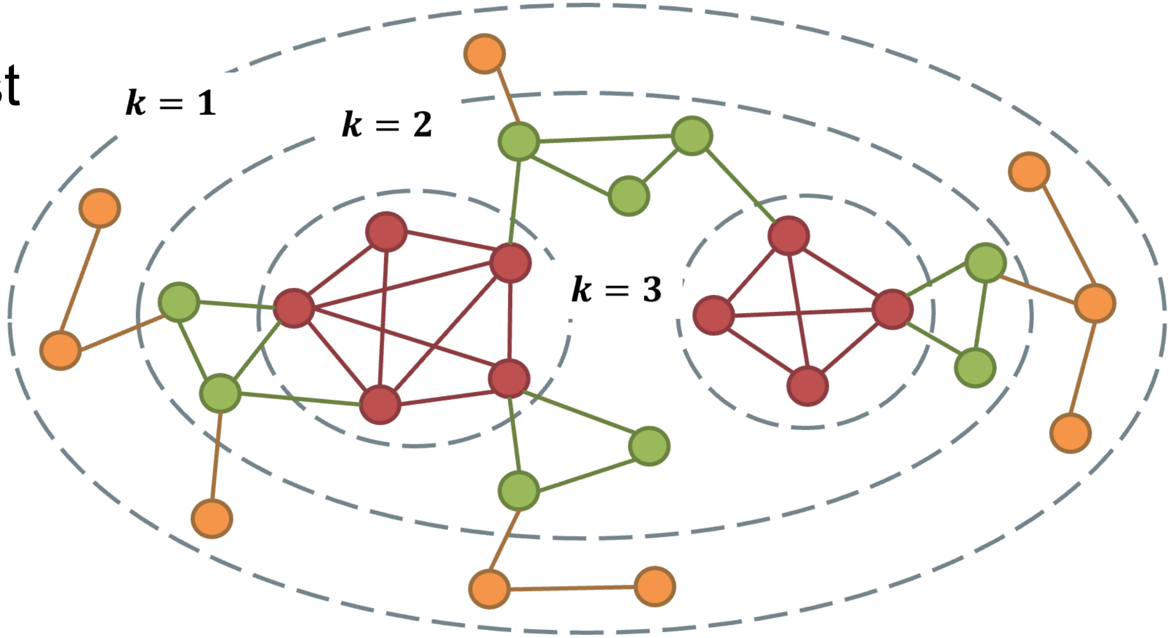
Core Decomposition: Global-view (Peeling)

For each unvisited vertex u with the lowest degree in G

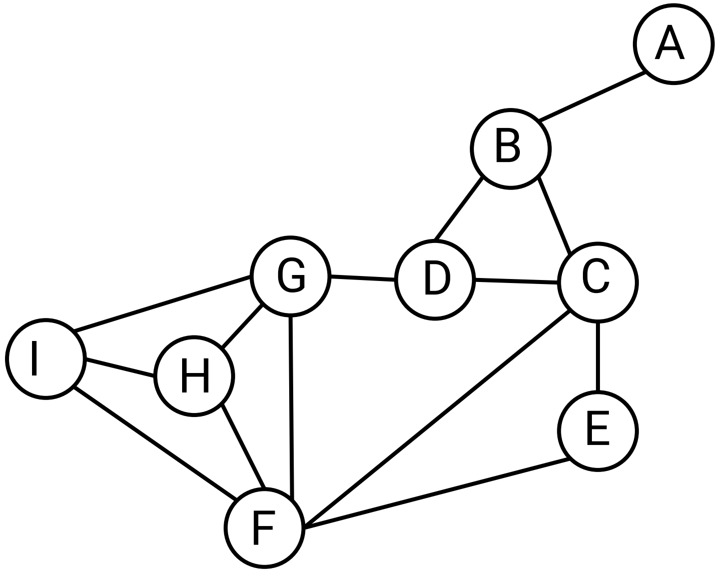
  assign core(u) as degree(u);

  mark u as visited;

  decrease the degree of its unvisited neighbors with higher degree than u by 1;



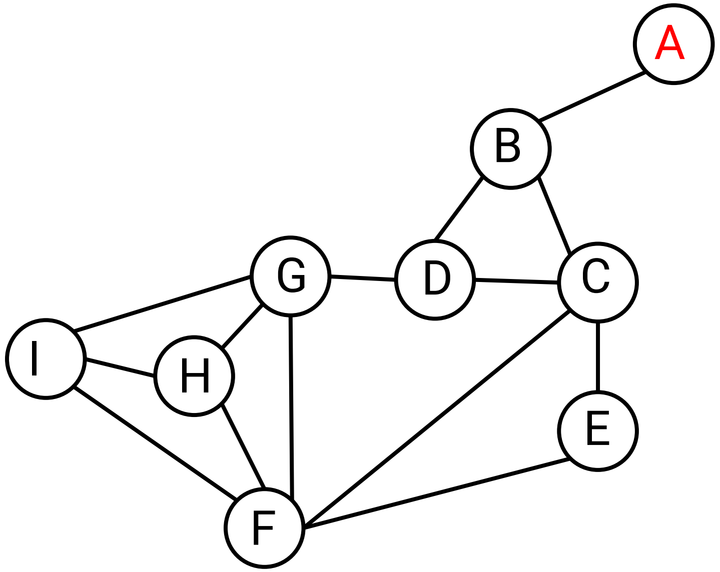
Example: O(n^2) Algorithm for Core Decomposition



degree	1	3	4	3	2	5	4	3	3
vertex	A	B	C	D	E	F	G	H	I

Iterate over all the vertices. For a deleting vertex v, if the degree of its unvisited neighbor u is greater than the degree of v, then decrease the degree of u by one.

Example: O(n^2) Algorithm for Core Decomposition



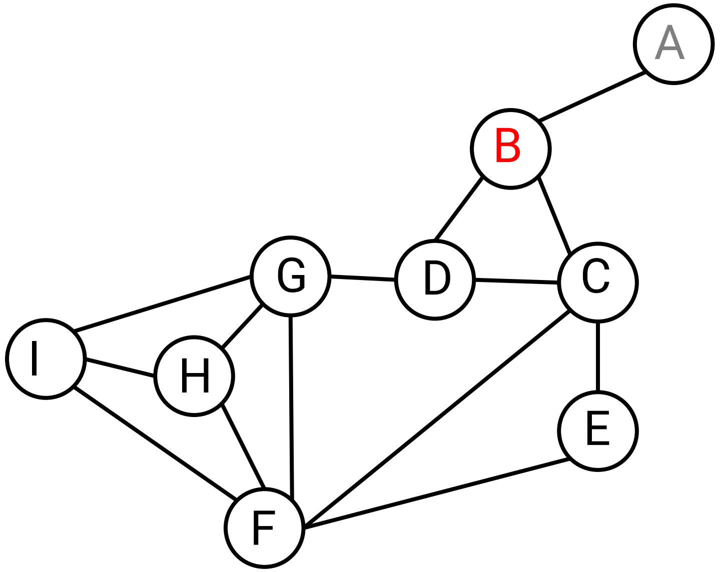
Start from the vertex A, A's neighbor is B.

degree	1	3	4	3	2	5	4	3	3
vertex	A	B	C	D	E	F	G	H	I

degree[B]>degree[A],
then degree[B]←degree[B]-1, degree[B]=2.

degree	1	2	4	3	2	5	4	3	3
vertex	A	B	C	D	E	F	G	H	I

Example: O(n^2) Algorithm for Core Decomposition



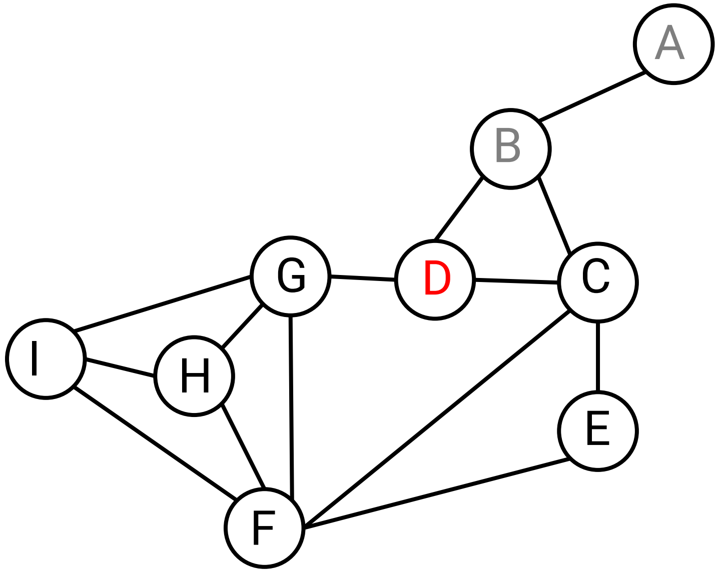
The next vertex is B, B's unvisited neighbors are C and D.

degree	1	2	4	3	2	5	4	3	3
vertex	A	B	C	D	E	F	G	H	I

degree[C]>degree[B],
then degree[C] ← degree[C]-1, degree[C]=3.
degree[D]>degree[B],
then degree[D] ← degree[D]-1, degree[D]=2.

degree	1	2	3	2	2	5	4	3	3
vertex	A	B	C	D	E	F	G	H	I

Example: O(n^2) Algorithm for Core Decomposition



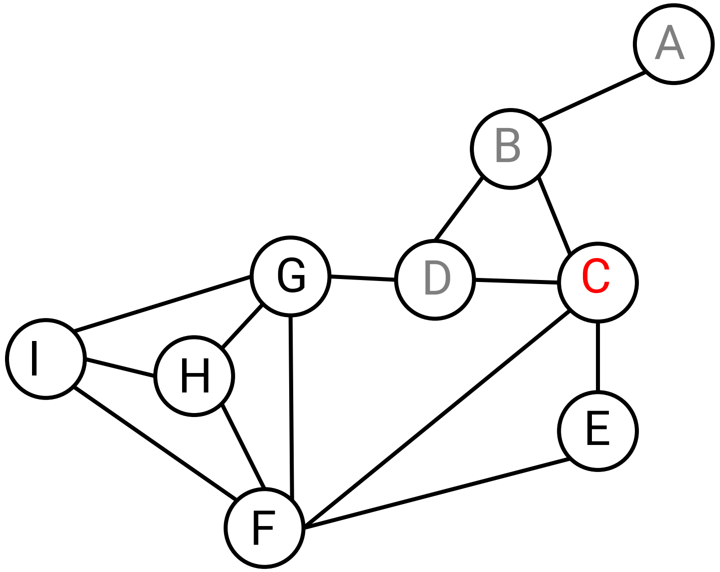
The next vertex with the smallest degree is D,
D's unvisited neighbors are C and G.

degree	1	2	3	2	2	5	4	3	3
vertex	A	B	C	D	E	F	G	H	I

degree[C]>degree[D],
then degree[C] ← degree[C]-1, degree[C]=2.
degree[G]>degree[D],
then degree[G] ← degree[G]-1, degree[G]=3.

degree	1	2	2	2	2	5	3	3	3
vertex	A	B	C	D	E	F	G	H	I

Example: O(n^2) Algorithm for Core Decomposition



The next vertex with the smallest degree is C,
C's unvisited neighbor is E.

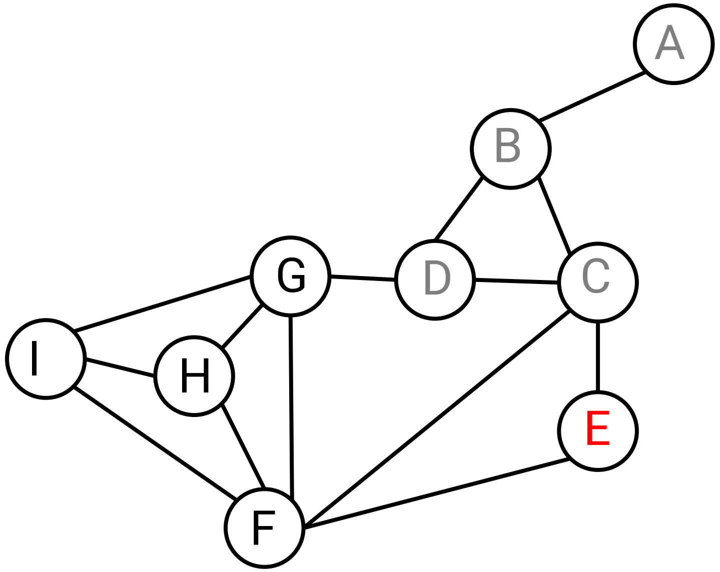
degree	1	2	2	2	2	5	3	3	3
vertex	A	B	C	D	E	F	G	H	I

degree[E]=degree[C], then nothing would be changed.

degree[F]>degree[C],
then degree[F] ← degree[F]-1, degree[F]=4.

degree	1	2	2	2	2	4	3	3	3
vertex	A	B	C	D	E	F	G	H	I

Example: O(n^2) Algorithm for Core Decomposition



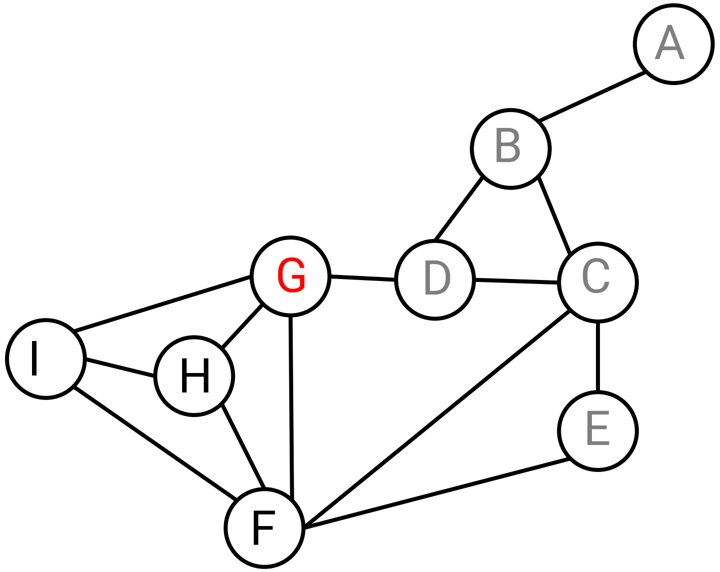
The next vertex with the smallest degree is E,
E's unvisited neighbor is F.

degree	1	2	2	2	2	4	3	3	3
vertex	A	B	C	D	E	F	G	H	I

degree[F]>degree[E],
then degree[F] ← degree[F]-1, degree[F]=3.

degree	1	2	2	2	2	3	3	3	3
vertex	A	B	C	D	E	F	G	H	I

Example: $O(n^2)$ Algorithm for Core Decomposition



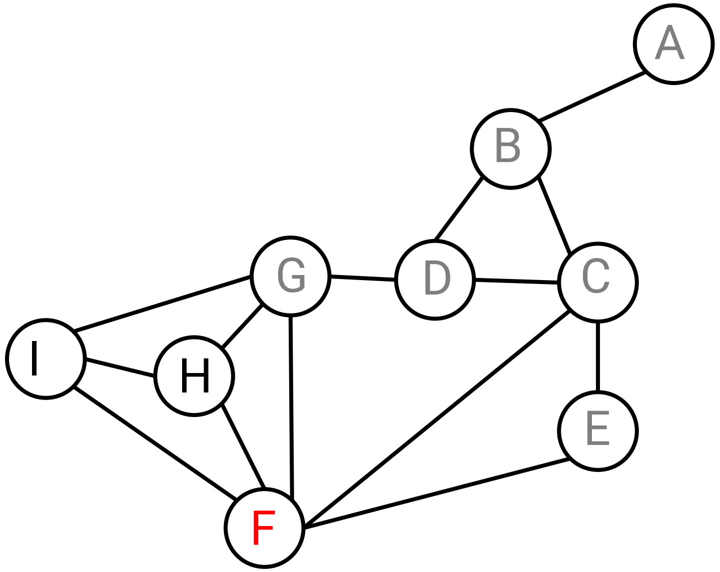
The next vertex with the smallest degree is G,
G's unvisited neighbors are F, H and I.

degree	1	2	2	2	2	3	3	3	3
vertex	A	B	C	D	E	F	G	H	I

degree[H]=degree[G], degree[I]=degree[G], degree[F]=degree[G], then nothing would be changed.

degree	1	2	2	2	2	3	3	3	3
vertex	A	B	C	D	E	F	G	H	I

Example: $O(n^2)$ Algorithm for Core Decomposition



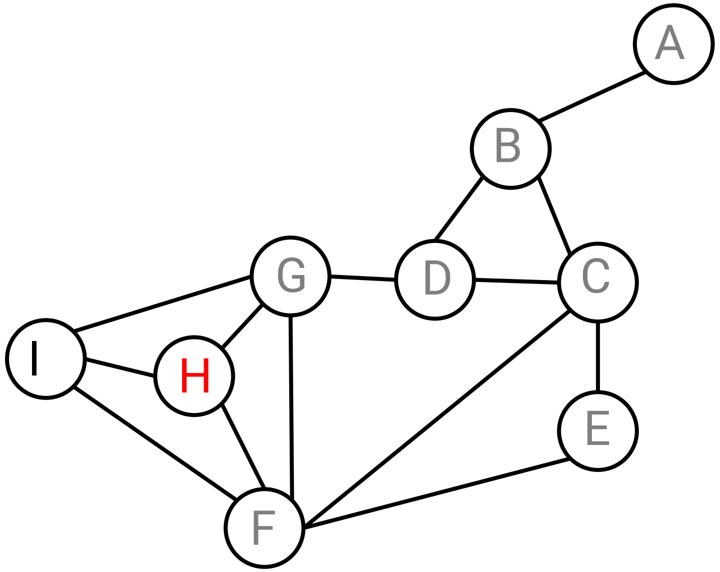
The next vertex with the smallest degree is F,
F's unvisited neighbors are H and I.

degree	1	2	2	2	2	3	3	3	3
vertex	A	B	C	D	E	F	G	H	I

degree[H]=degree[F], degree[I]=degree[F],
then nothing would be changed.

degree	1	2	2	2	2	3	3	3	3
vertex	A	B	C	D	E	F	G	H	I

Example: O(n^2) Algorithm for Core Decomposition



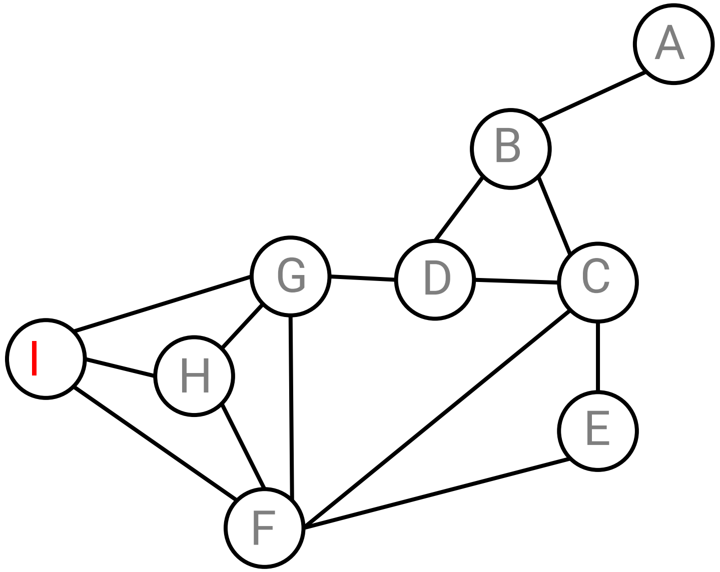
The next vertex with the smallest degree is H, H's unvisited neighbor is I.

degree	1	2	2	2	2	3	3	3	3
vertex	A	B	C	D	E	F	G	H	I

degree[I]=degree[H], then nothing would be changed.

degree	1	2	2	2	2	3	3	3	3
vertex	A	B	C	D	E	F	G	H	I

Example: O(n^2) Algorithm for Core Decomposition

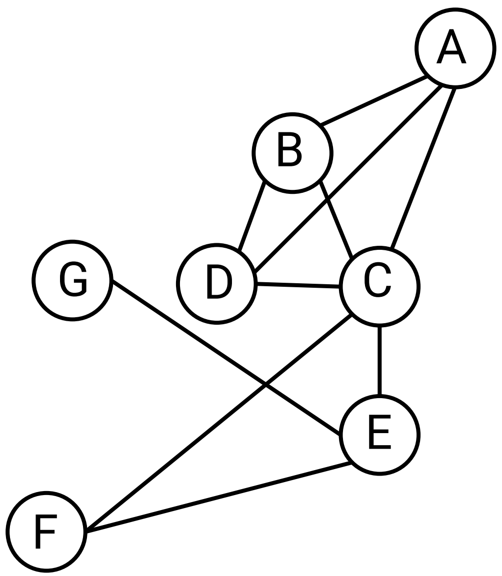


degree	1	2	2	2	2	3	3	3	3
vertex	A	B	C	D	E	F	G	H	I

The decomposition process is complete.

Quick exercise

Could you compute the core number of each vertex?



Example: O(n^2) Algorithm for Core Decomposition

The time complexity of the whole process is O(n^2+m) = O(n^2).

Need to get the vertex with minimum degree in each iteration:

Using heap (priority queue): O(m*log(n))
Using Fabonacci heap: O(m+n*log(n))

Any better solution?

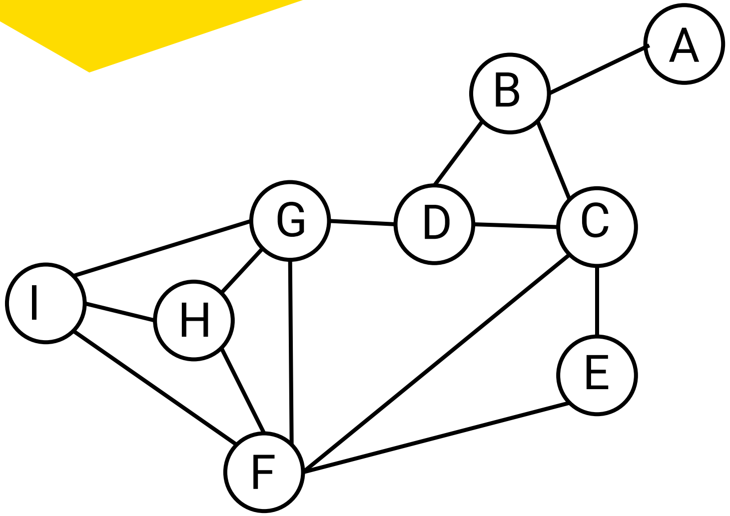
Core Decomposition using Doubly Linked List

For each degree d, store all vertices with degree d using a doubly linked list (DLL)

When the degree of a vertex u decreases, move u from old DLL to a new DLL.

Time complexity: O(m)
Drawback: cannot fully utilize CPU cache~

Core Decomposition using Flat Array



In order to achieve the time complexity of O(m), we first sort all vertices according to their degree.

degree	1	2	3	3	3	3	4	4	5
vertex	A	E	D	B	I	H	G	C	F

Algorithm : CoreDecomposition

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

Output : $\{cn(u) \mid u \in V\}$: core number of every vertex in G

1 $d(u) \leftarrow deg(u, G)$ for every $u \in V$;

2 order the vertices in V in increasing order of their degrees;

3 **for each** $u \in V$ in the order **do**

4     $cn(u) \leftarrow d(u)$;

5   **for each** $v \in N(u)$ with $d(v) > d(u)$ **do**

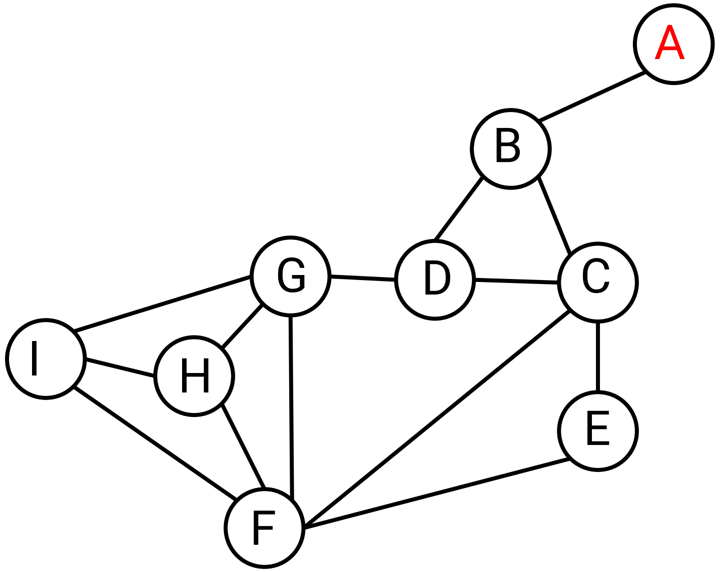
6         $d(v) \leftarrow d(v) - 1$;

7   reorder V accordingly;

8 **return** $cn(u)$ of every $u \in V$

Iterate over all the vertices. If the degree of neighbouring vertex u of vertex v is greater than the degree of v, decrease the degree of u by 1. Then, for the line 7, swap the positions of u and the first vertex with the same degree as u’s original degree. Because we use the bin sort, the time complexity of reorder the array is O(n). Thus, the total time complexity for core decomposition is O(m).

Core Decomposition using Flat Array



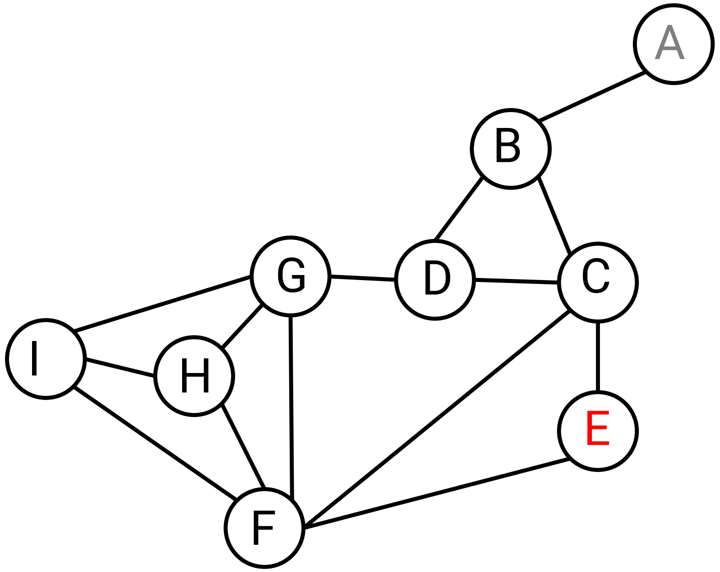
Loop according to the order of degrees in the table, start from the vertex A with the minimum degree.

degree	1	2	3	3	3	3	4	4	5
vertex	A	E	D	B	I	H	G	C	F

degree[B]>degree[A], then degree[B] ← degree[B]-1, degree[B]=2, swap the positions of B and the first vertex with the same degree as B's original degree (i.e., swap the position of B and D).

degree	1	2	2	3	3	3	4	4	5
vertex	A	E	B	D	I	H	G	C	F

Core Decomposition using Flat Array



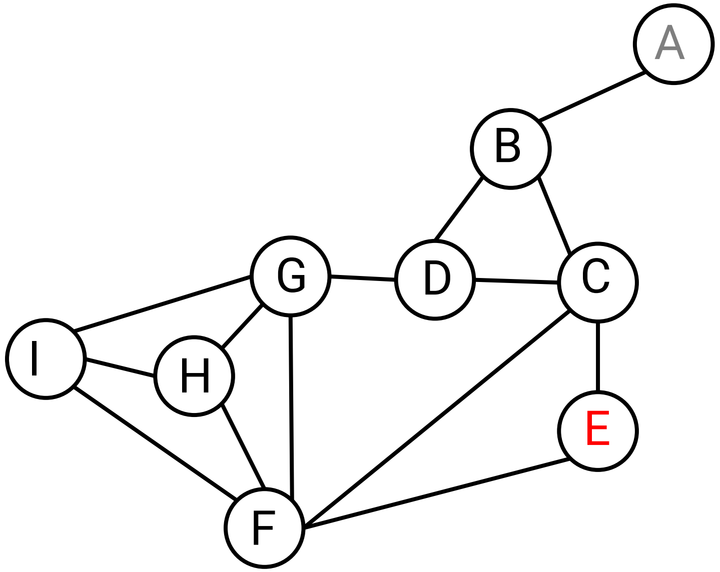
The next vertex in the table is E, E's neighbors are C and F.

degree	1	2	2	3	3	3	4	4	5
vertex	A	E	B	D	I	H	G	C	F

For vertex C,
degree[C]>degree[E],
then degree[C] ← degree[C]-1, degree[C]=3,
swap the position of C and G.

degree	1	2	2	3	3	3	3	4	5
vertex	A	E	B	D	I	H	C	G	F

Core Decomposition using Flat Array



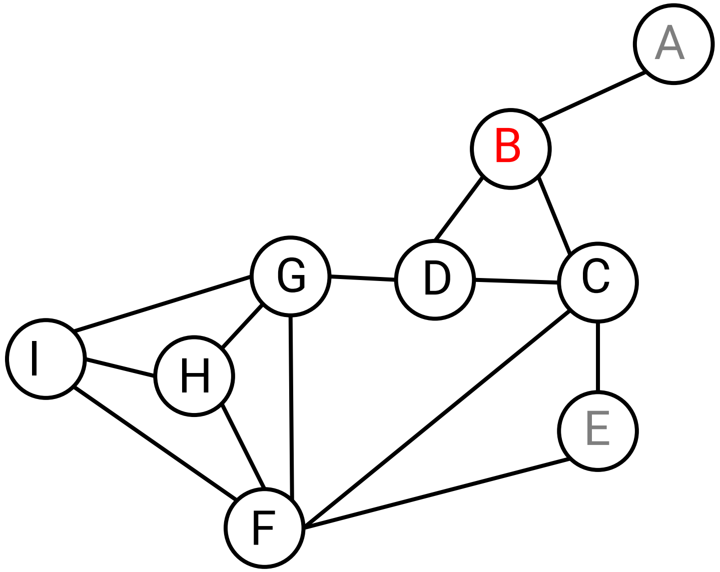
The next vertex in the table is E, E's neighbors are C and F.

degree	1	2	2	3	3	3	3	4	5
vertex	A	E	B	D	I	H	C	G	F

For vertex F,
degree[F]>degree[E],
then degree[F] ← degree[F]-1, degree[F]=4.

degree	1	2	2	3	3	3	3	4	4
vertex	A	E	B	D	I	H	C	G	F

Core Decomposition using Flat Array



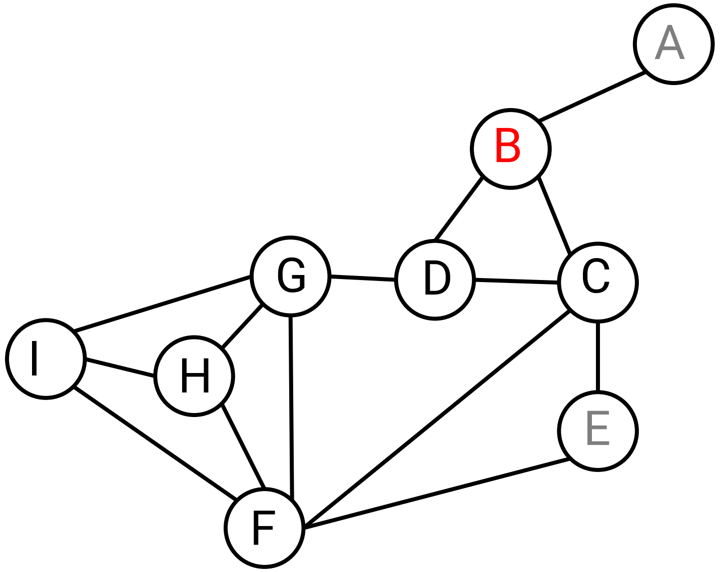
The next vertex in the table is B, B's unvisited neighbors are C and D.

degree	1	2	2	3	3	3	3	4	4
vertex	A	E	B	D	I	H	C	G	F

For vertex C,
degree[C]>degree[B],
then degree[C] ← degree[C]-1, degree[C]=2.
Swap the position of C and D.

degree	1	2	2	2	3	3	3	4	4
vertex	A	E	B	C	I	H	D	G	F

Core Decomposition using Flat Array



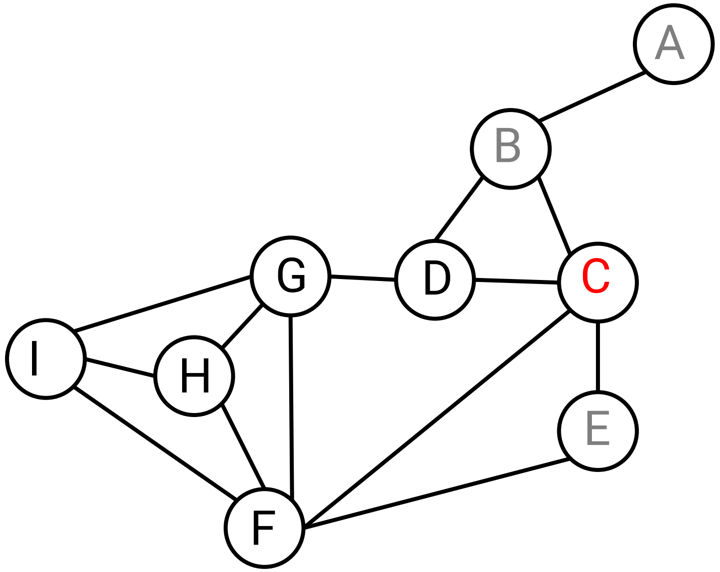
The next vertex in the table is B, B's neighbors are C and D.

degree	1	2	2	2	3	3	3	4	4
vertex	A	E	B	C	I	H	D	G	F

For neighbor D,
degree[D]>degree[B],
then degree[D] ← degree[D]-1, degree[D]=2.
Exchange the position of D and I.

degree	1	2	2	2	2	3	3	4	4
vertex	A	E	B	C	D	H	I	G	F

Core Decomposition using Flat Array



The next vertex in the table is C, C's unvisited neighbors are D and F.

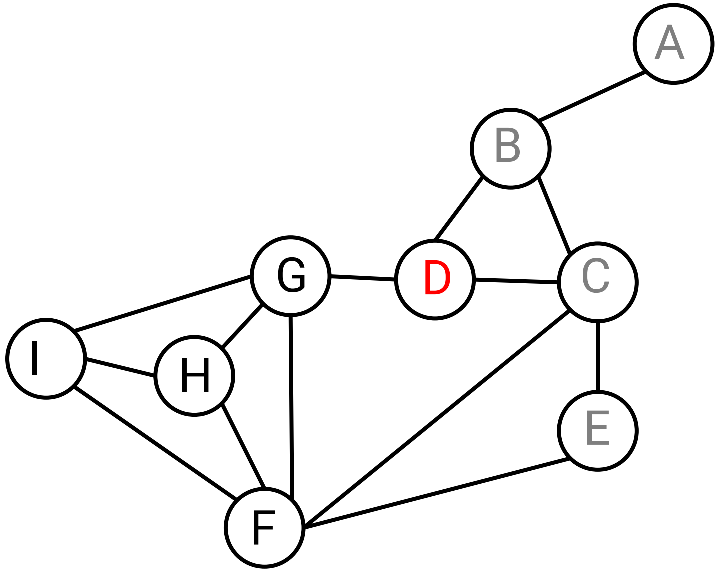
degree	1	2	2	2	2	3	3	4	4
vertex	A	E	B	C	D	H	I	G	F

For vertex D, the degree are equal to degree[C]
Then the order of table would not change.

degree[F] is updated to 3.
Exchange the position of F and G.

degree	1	2	2	2	2	3	3	3	4
vertex	A	E	B	C	D	H	I	F	G

Core Decomposition using Flat Array



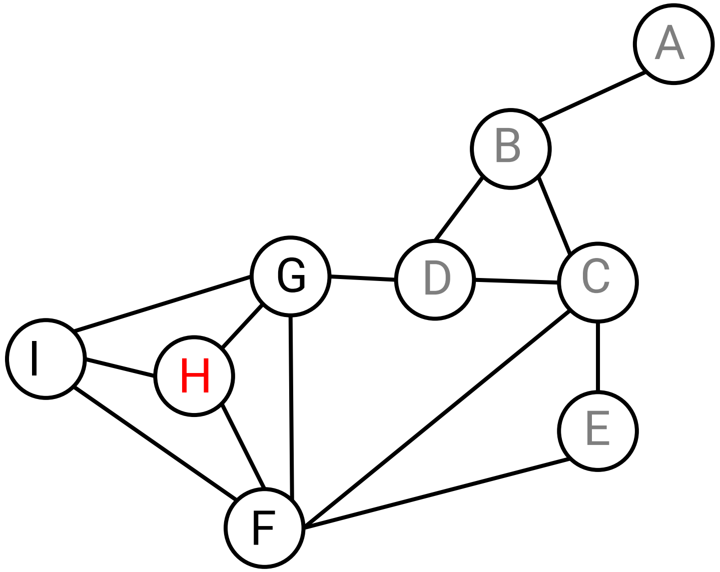
The next vertex in the table is D, D's unvisited neighbor is G.

degree	1	2	2	2	2	3	3	3	4
vertex	A	E	B	C	D	H	I	F	G

For vertex G,
degree[G]>degree[D],
then degree[G] ← degree[G]-1, degree[G]=3.

degree	1	2	2	2	2	3	3	3	3
vertex	A	E	B	C	D	H	I	F	G

Core Decomposition using Flat Array



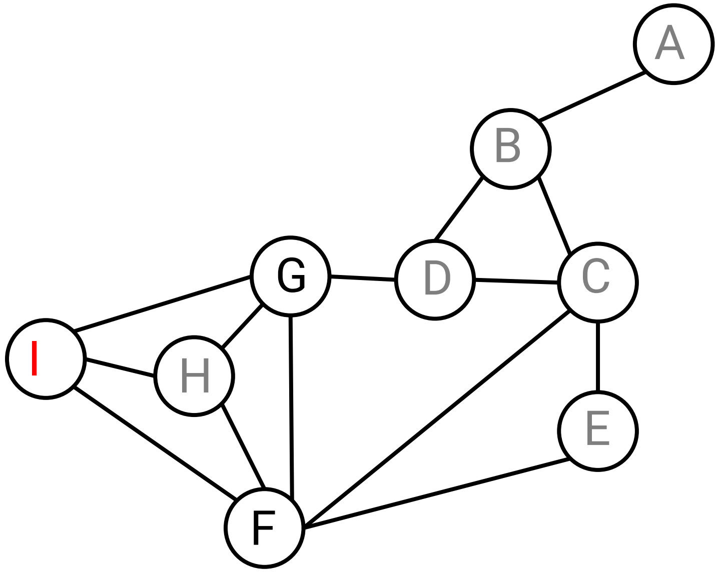
The next vertex in the table is H, H’s unvisited neighbors are F, G and I.

degree	1	2	2	2	2	3	3	3	3
vertex	A	E	B	C	D	H	I	F	G

For vertices F, G and I,
degree[F]=degree[G]= degree[H]=degree[I].
Then the order of table would not change.

degree	1	2	2	2	2	3	3	3	3
vertex	A	E	B	C	D	H	I	F	G

Core Decomposition using Flat Array

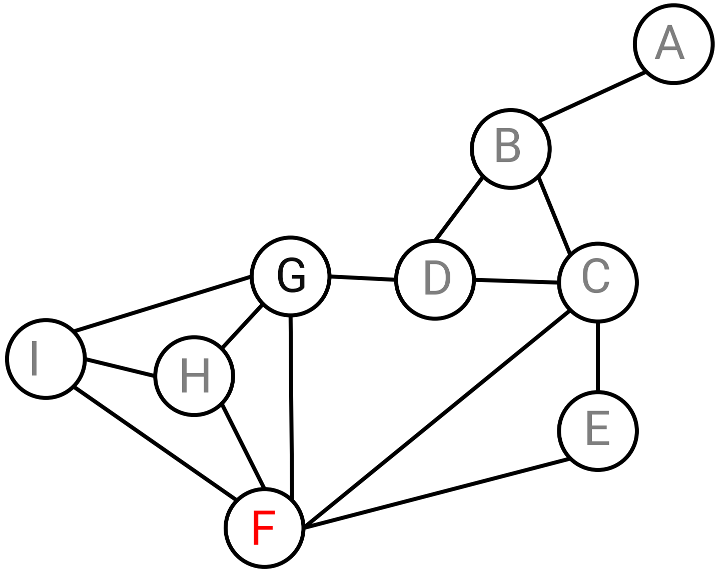


The next vertex in the table is I, I's unvisited neighbors are F and G.

degree	1	2	2	2	2	3	3	3	3
vertex	A	E	B	C	D	H	I	F	G

For vertex F, G,
degree[F]=degree[G]= degree[I].
Then the order of table would not change.

Core Decomposition using Flat Array

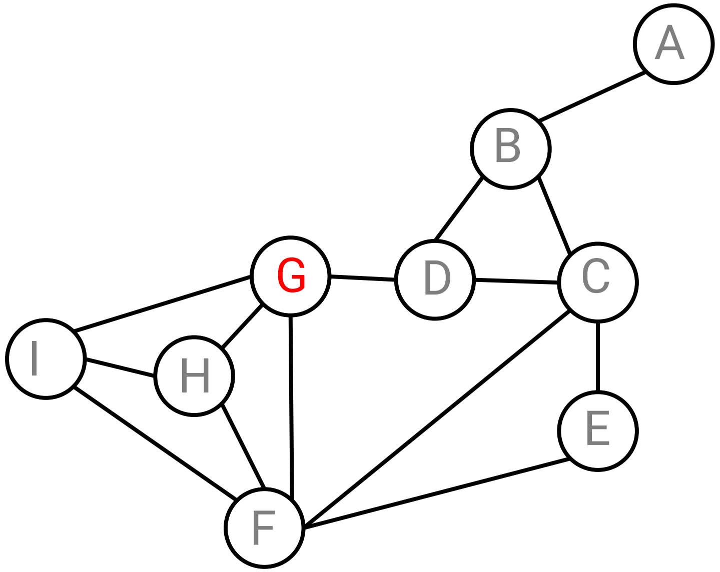


The next vertex in the table is F, F’s unvisited neighbor is G

degree	1	2	2	2	2	3	3	3	3
vertex	A	E	B	C	D	H	I	F	G

For vertex G, degree[G]=degree[F].
Then the order of table would not change.

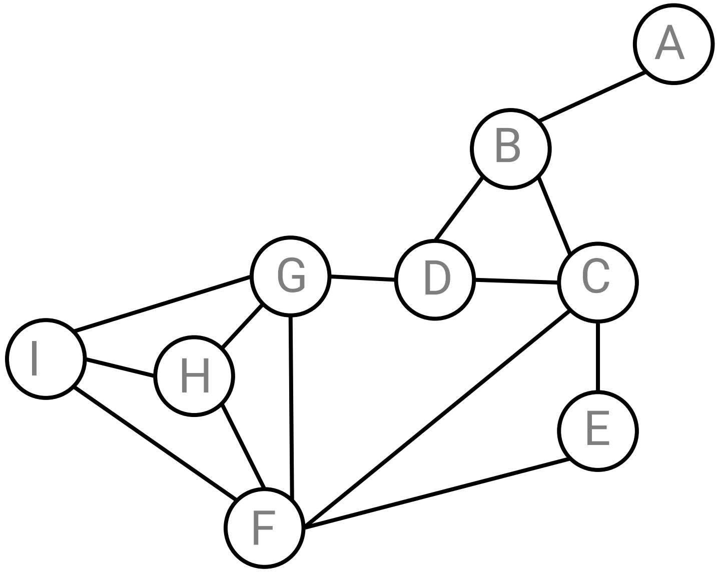
Core Decomposition using Flat Array



The next vertex in the table is G, there is no G's unvisited neighbor.

degree	1	2	2	2	2	3	3	3	3
vertex	A	E	B	C	D	H	I	F	G

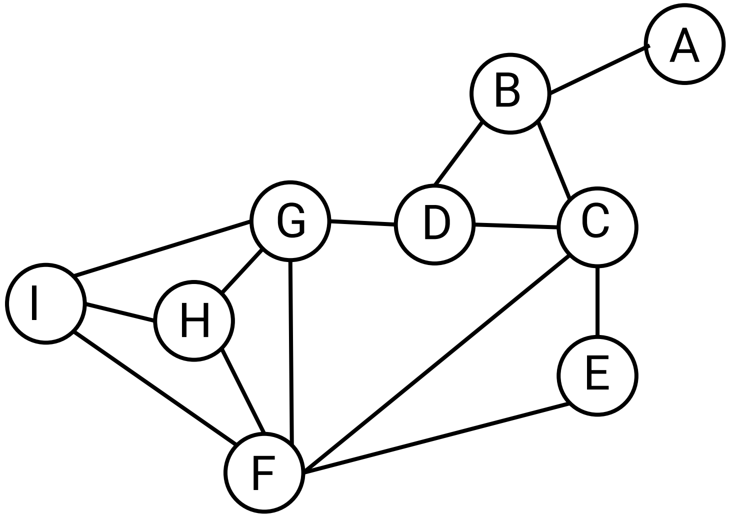
Core Decomposition using Flat Array



After we traverse all edges in this graph, we get the core number of each vertex in O(m) time.

degree	1	2	2	2	2	3	3	3	3
vertex	A	E	B	C	D	H	I	F	G

Core Decomposition using Flat Array

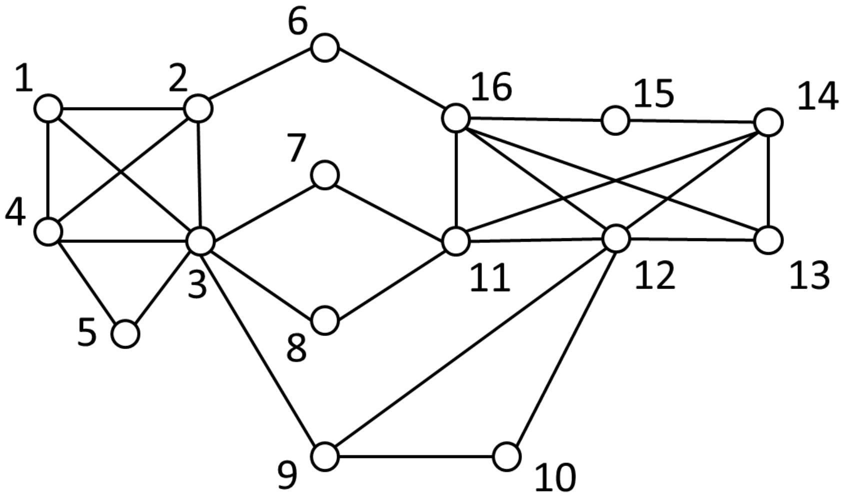


What we need to implement the O(m) algorithm

1. An array to sort vertices in non-decreasing order of degree
2. An array to locate the start position for each degree
3. An array to get the position of each vertex id
4. An array to maintain the degree of each vertex

degree	1	2	3	3	3	3	4	4	5
vertex	A	E	D	B	I	H	G	C	F

An example



index	**d**	**b**	**D**	**p**
1	3	0	5	7
2	4	1	6	10
3	7	7	7	16
4	4	10	8	11
5	2	13	10	1
6	2	15	15	2
7	2	16	1	3
8	2		9	4
9	3		13	8
10	2		2	5
11	5		4	13
12	6		14	15
13	3		11	9
14	4		16	12
15	2		12	6
16	5		3	14

function K-CORES(Graph G)
    initialize$(\mathbf{d}, \mathbf{b}, \mathbf{D}, \mathbf{p}, G)$
    for all $i \leftarrow 1$ **to** n **do**
        $v \leftarrow \mathbf{D}[i]$
        for all $u \in N_G(v)$ **do**
            if $\mathbf{d}[u] > \mathbf{d}[v]$ **then**
                $du \leftarrow \mathbf{d}[u]$, $pu \leftarrow \mathbf{p}[u]$
                $pw \leftarrow \mathbf{b}[du]$, $w \leftarrow \mathbf{D}[pw]$
                if $u \neq w$ **then**
                    $\mathbf{D}[pu] \leftarrow w$, $\mathbf{D}[pw] \leftarrow u$
                    $\mathbf{p}[u] \leftarrow pw$, $\mathbf{p}[w] \leftarrow pu$
                end if
                $\mathbf{b}[du]{+}{+}$, $\mathbf{d}[u]{-}{-}$
            end if
        end for
    end for
    return d
end function

https://www.vldb.org/pvldb/vol9/p13-khaouid.pdf

Variants of K-Core

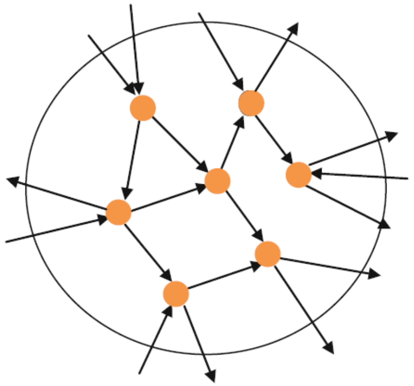
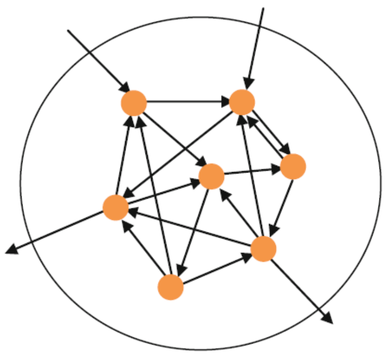
(optional)

k-Core on Directed Graphs

Optional

<u>Directed graphs</u>: citation networks, WWW, social networks (by following relations), P2P networks, etc.

(k, l)-Core on a directed graph: the maximal subgraph F where each vertex has at least k out-neighbors in F and at least l in-neighbors in F.

Part of a directed graph:   : (2,2)-core

Giatsidis, Christos, Dimitrios M. Thilikos, and Michalis Vazirgiannis. "D-cores: measuring collaboration of directed graphs based on degeneracy." Knowledge and information systems 35.2 (2013): 311-343.

K-Core on Weighted Graphs

Optional

<u>Weighted graphs</u>: air transportation networks, co-author networks, social networks (with tie strength), etc.

The weighted degree of a vertex i,

Neighbor set of i

Edge weight of (i, j)

$$d'(i) = \sum_{j \in N(i)} w_{ij}$$

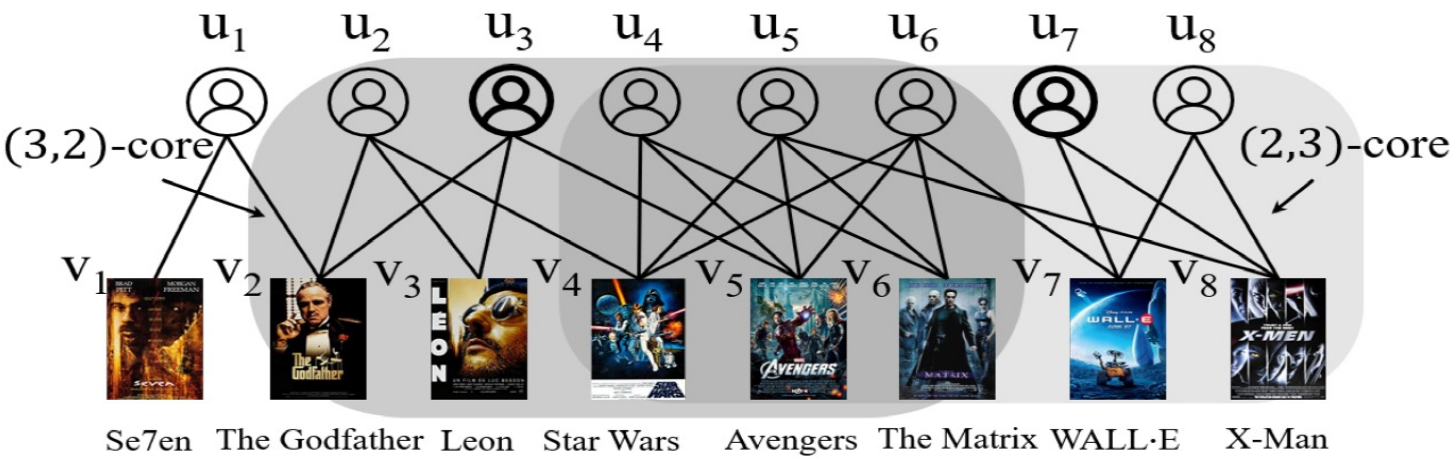
The k-core: $d'(i) \geq k$ for every vertex i inside.

*Eidsaa, Marius, and Eivind Almaas. "**S-core network decomposition: A generalization of k-core analysis to weighted networks.**" Physical Review E 88.6 (2013): 062819.*
*Garas, Antonios, Frank Schweitzer, and Shlomo Havlin. "**A k-shell decomposition method for weighted networks.**" New Journal of Physics 14.8 (2012): 083030.*

K-Core on Bipartite Graphs

Optional

Bipartite graphs: the vertices are divided into two disjoint sets U and L such that every edge connects a vertex in U to one in L.



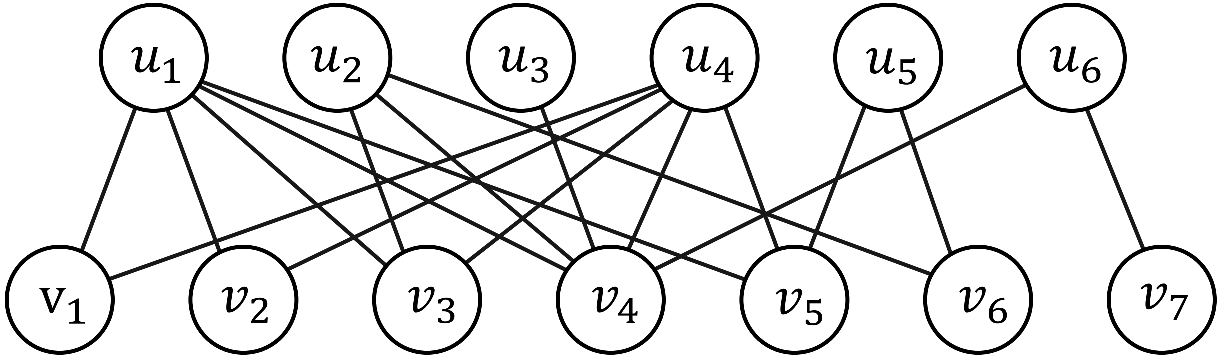
The (α, β)-core of G consists of two node sets $U' \subseteq U$ and $L' \subseteq L$ s.t. the subgraph S induced by $U' \cup L'$ is the maximal subgraph of G in which all the nodes in U' have degree at least α in S and all the nodes in L' have degree at least β in S.

Liu, Boge, et al. ***"Efficient (a,b)-core Computation an Index-based Approach".*** *WWW, 2019.*

Online computation of (α, β)-core

Optional

The algorithm filters out the upper/lower vertices whose degrees are less than alpha/beta (i.e., iteratively remove vertices without enough degrees).



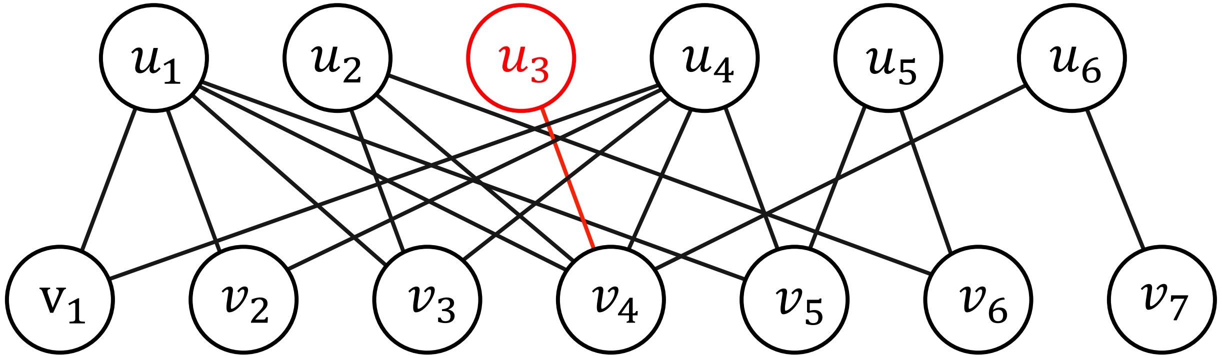
Example of computing (2,3)-core

Danhao Ding, Hui Li, Zhipeng Huang, and Nikos Mamoulis.2017. **Efficient fault-tolerant group recommendation using alpha-beta-core.** In Proceedings of the 2017 ACM on Conference on Information and Knowledge Management.2047–2050.

Online computation of (α, β)-core

Optional

Remove u3 and its incident edges, which are colored red.



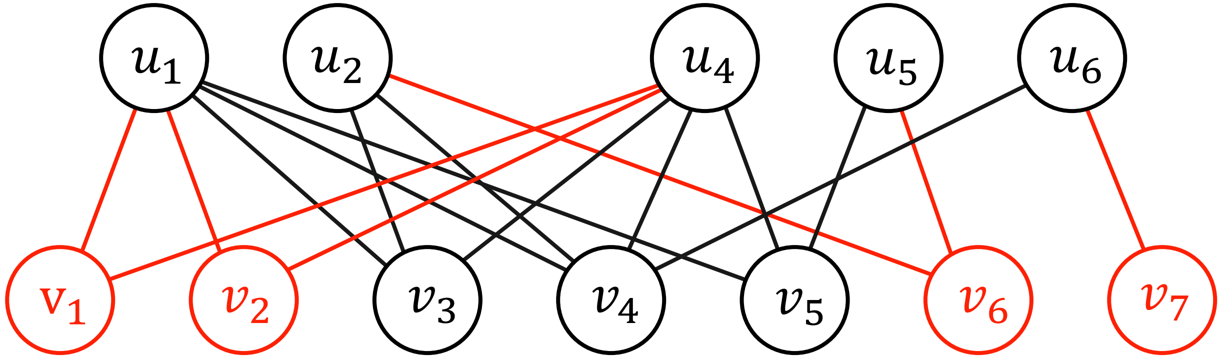
Example of computing (2,3)-core

Danhao Ding, Hui Li, Zhipeng Huang, and Nikos Mamoulis.2017. **Efficient fault-tolerant group recommendation using alpha-beta-core.** In Proceedings of the 2017 ACM on Conference on Information and Knowledge Management.2047–2050.

Online computation of (α, β)-core

Optional

Remove v1, v2, v6, v7 and their incident edges, which are colored red.



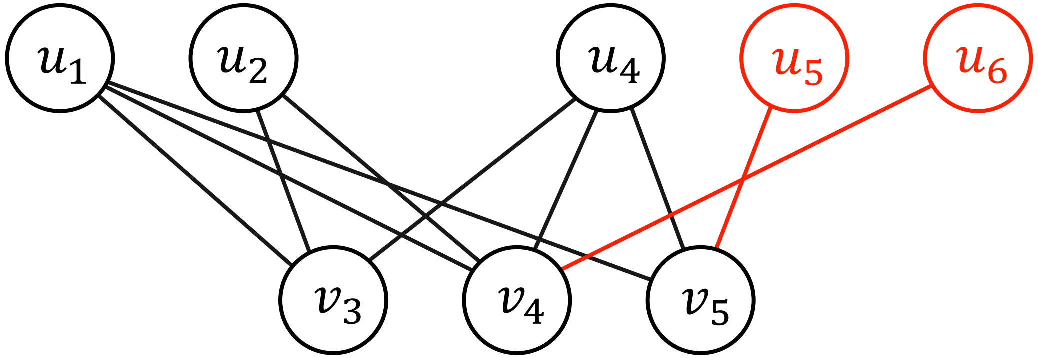
Example of computing (2,3)-core

Danhao Ding, Hui Li, Zhipeng Huang, and Nikos Mamoulis.2017. **Efficient fault-tolerant group recommendation using alpha-beta-core.** In Proceedings of the 2017 ACM on Conference on Information and Knowledge Management.2047–2050.

Online computation of (α, β)-core

Optional

Remove u5, u6 and their incident edges, which are colored red.



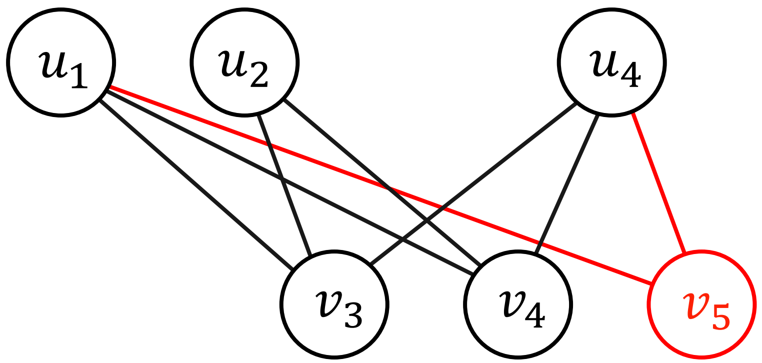
Example of computing (2,3)-core

Danhao Ding, Hui Li, Zhipeng Huang, and Nikos Mamoulis.2017. **Efficient fault-tolerant group recommendation using alpha-beta-core.** In Proceedings of the 2017 ACM on Conference on Information and Knowledge Management.2047–2050.

Online computation of (α, β)-core

Optional

Remove v5 and their incident edges, which are colored red.



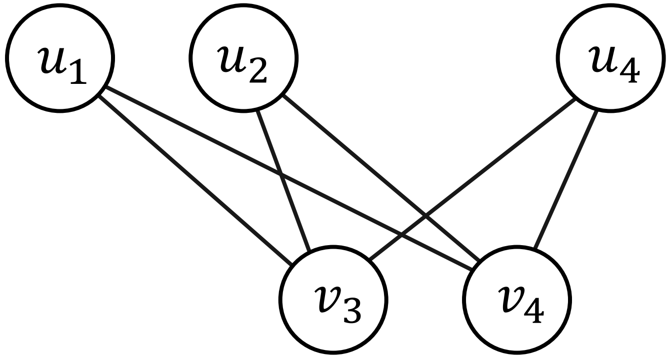
Example of computing (2,3)-core

Danhao Ding, Hui Li, Zhipeng Huang, and Nikos Mamoulis.2017. **Efficient fault-tolerant group recommendation using alpha-beta-core.** In Proceedings of the 2017 ACM on Conference on Information and Knowledge Management.2047–2050.

Online computation of (α, β)-core

Optional

Return the subgraph when all vertices satisfy the degree constraints.

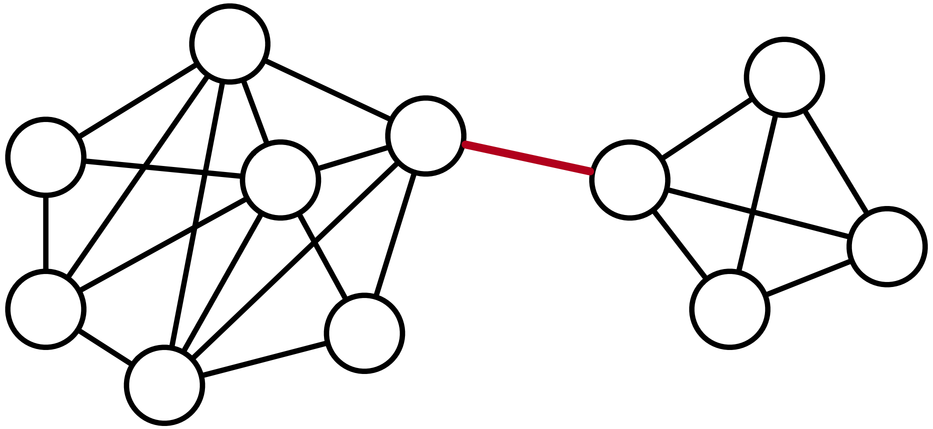


Example of computing (2,3)-core

Danhao Ding, Hui Li, Zhipeng Huang, and Nikos Mamoulis.2017. **Efficient fault-tolerant group recommendation using alpha-beta-core.** In Proceedings of the 2017 ACM on Conference on Information and Knowledge Management.2047–2050.

Other Models

Why not k-core?



K-Truss

A maximal subgraph where each edge is contained in at least *k-2* triangles in the subgraph, i.e., each edge has a support of at least *k*-2 in the subgraph.

J. Cohen. Trusses: Cohesive subgraphs for social network analysis. *National Security Agency Technical Report*, page 16, 2008.

- The strength of a tie can be estimated by the number of triangles containing it.

M. Wang, C. Wang, J. X. Yu, and J. Zhang, Community detection in social networks: An in-depth benchmarking study with a procedure-oriented framework. *PVLDB*, 8(10):998–1009, 2015.

- High quality on some community metrics.
- High accuracy on approximating some ground-truth communities.
- The most efficient one among all the evaluated algorithms.

Properties of k-Truss

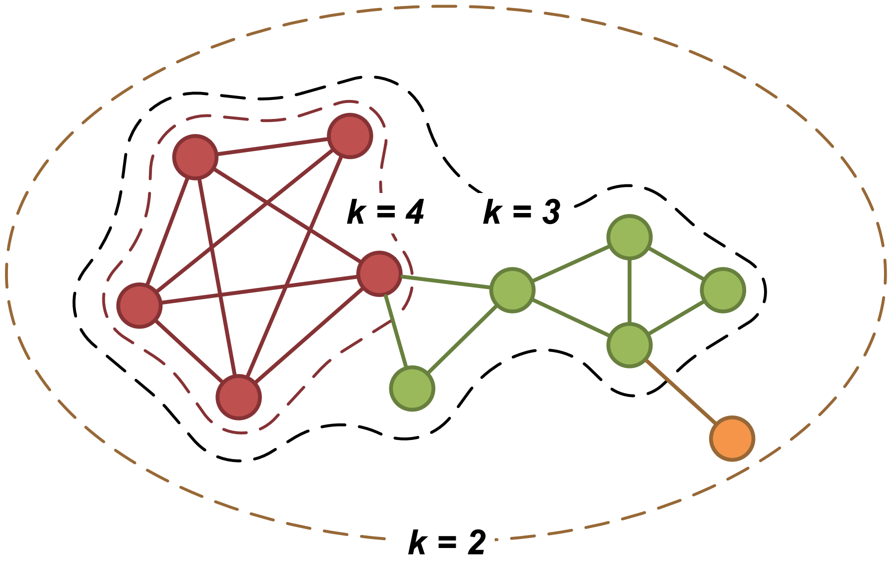
A maximal subgraph where each edge is contained in at least *k-2* triangles in the subgraph.

Each k-truss of G is a subgraph of a $(k-1)$-core of G.

Proof scratch:

- Each edge is contained in at least k-2 triangle.

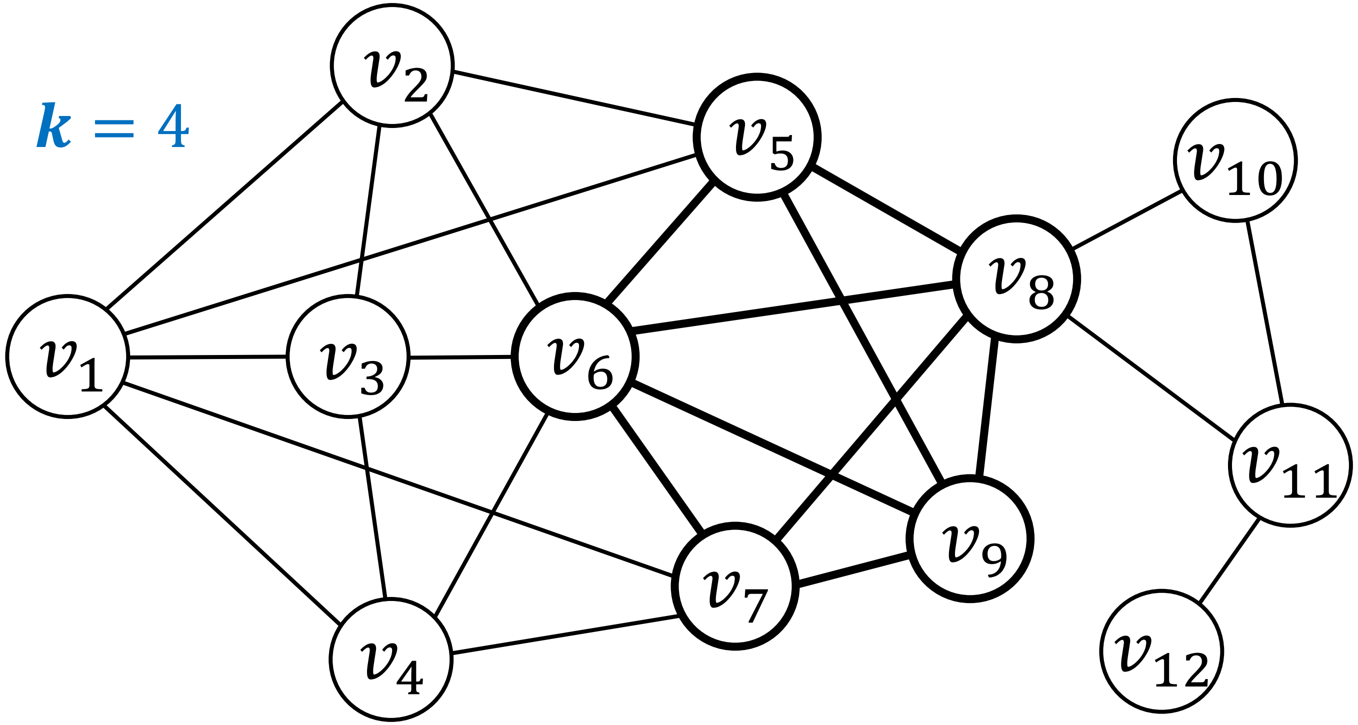
- To ensure this, each vertex should have at least k-1 neighbors, i.e., for a vertex u, if the edge (u, v) is contained in k-2 triangles, there should be at least k-2 common neighbors of u and v.



Cohen, Jonathan. "**Trusses: Cohesive subgraphs for social network analysis.**" National Security Agency Technical Report16 (2008): 3-1.

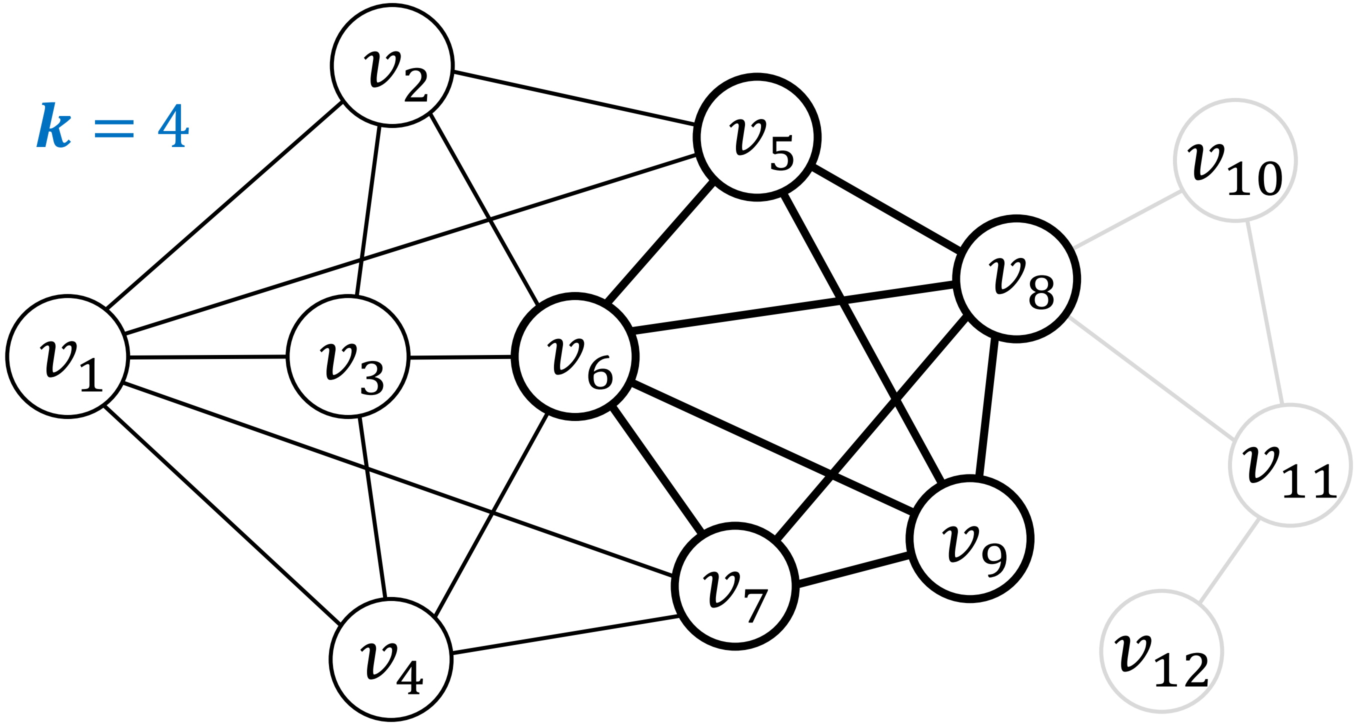
K-Truss Computation

1. Compute the (*k*-1)-core.
2. Compute the support of each edge.
3. Recursively delete each edge with support of less than *k*-2.
4. Delete the isolated vertices.



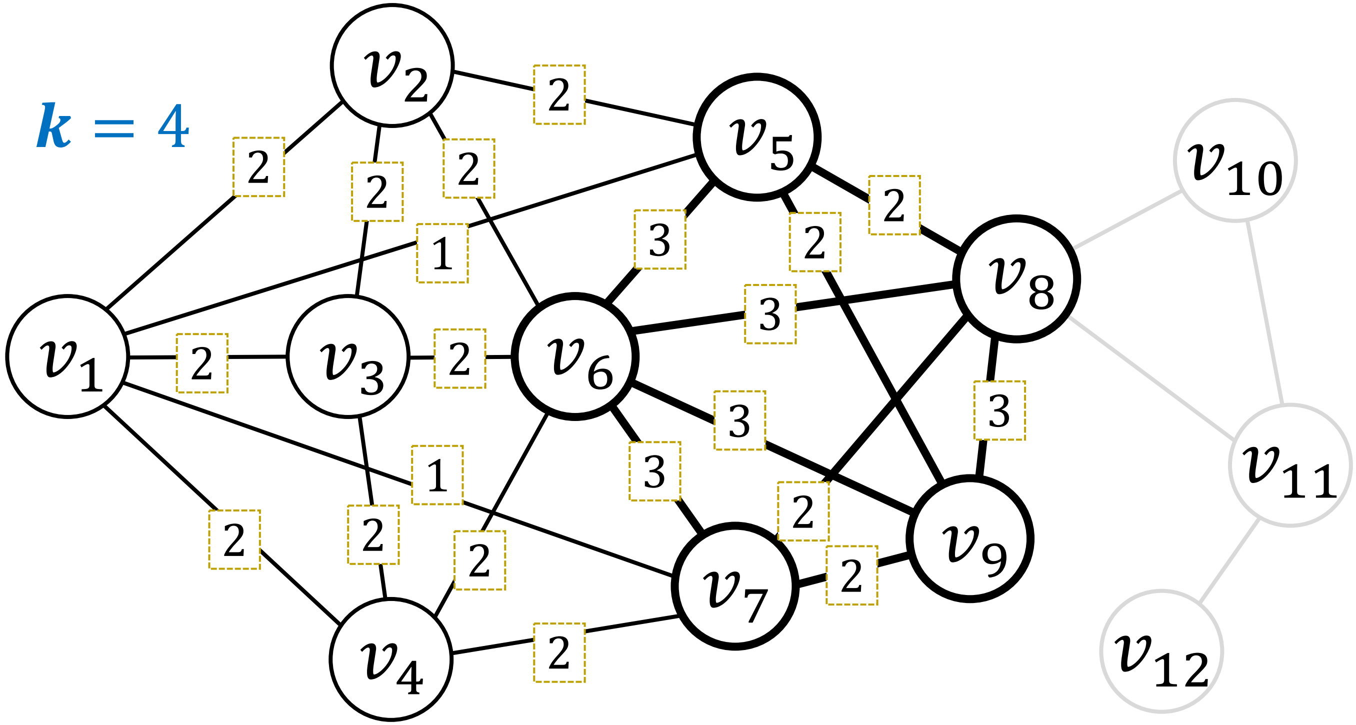
K-Truss Computation

1. **Compute the (*k*-1)-core.**
2. Compute the support of each edge.
3. Recursively delete each edge with support of less than *k*-2.
4. Delete the isolated vertices.



K-Truss Computation

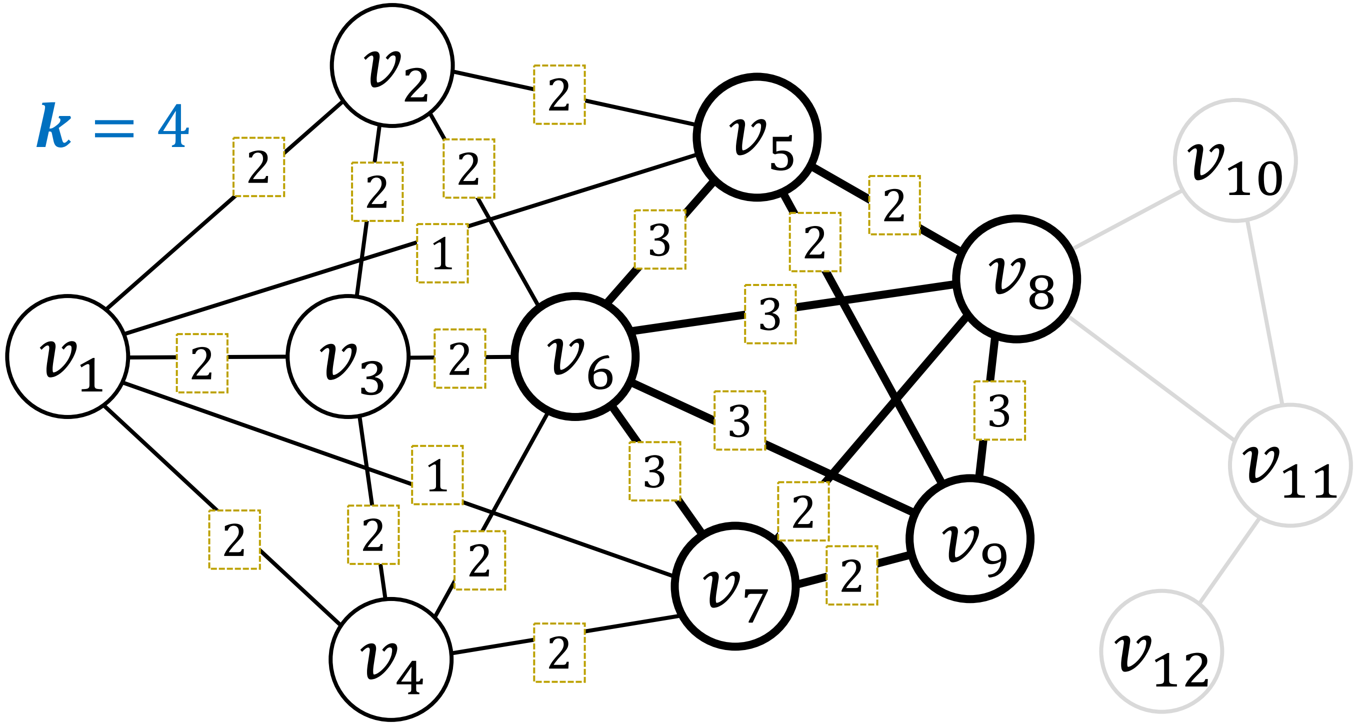
1. Compute the (k-1)-core.
2. **Compute the support of each edge.**
3. Recursively delete each edge with support of less than k-2.
4. Delete the isolated vertices.



K-Truss Computation

1. Compute the (*k*-1)-core.
2. Compute the support of each edge.
3. **Recursively delete each edge with support of less than *k*-2.**
4. Delete the isolated vertices.

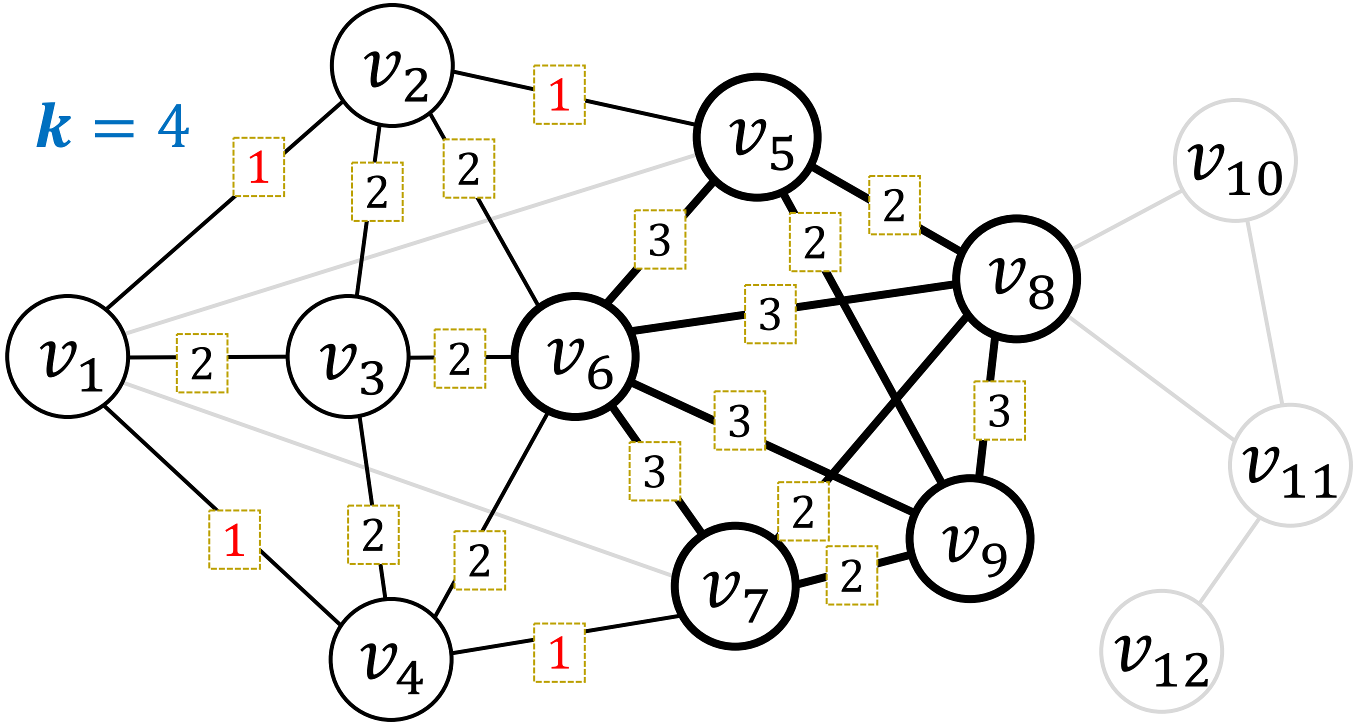
$k = 4$



K-Truss Computation

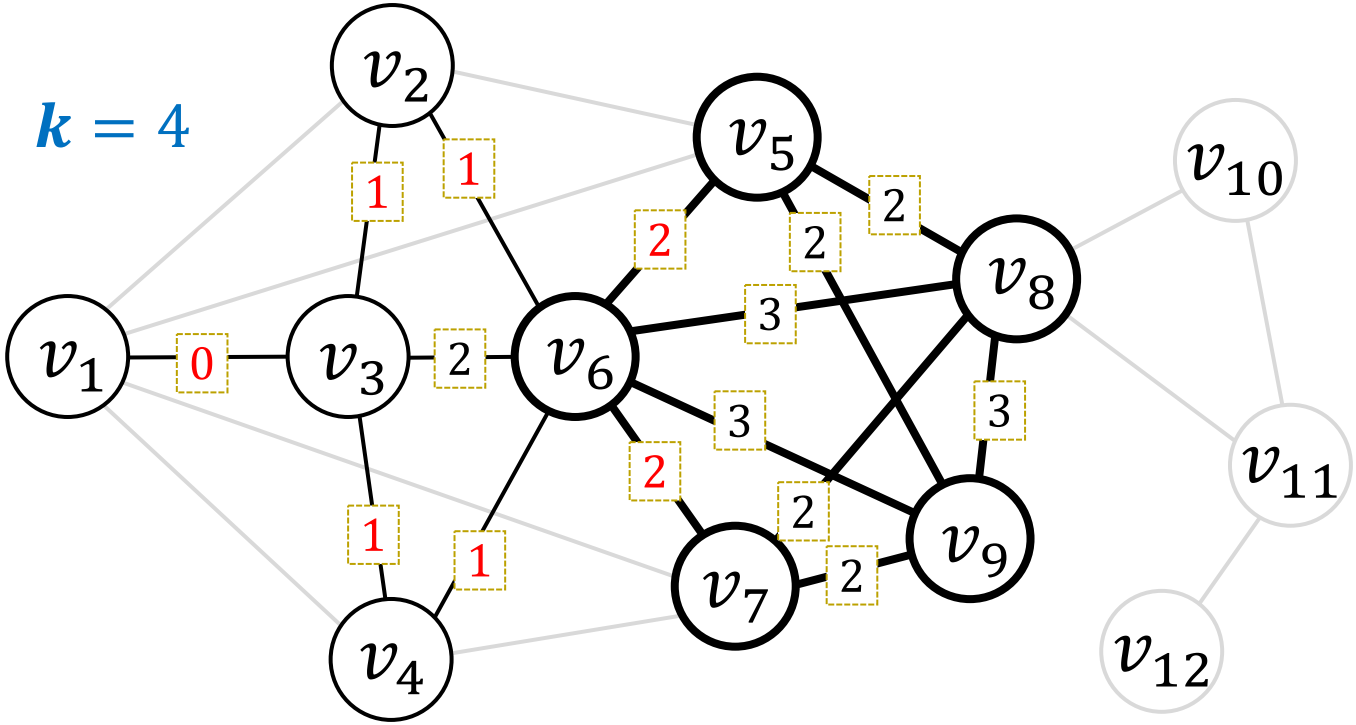
1. Compute the (*k*-1)-core.
2. Compute the support of each edge.
3. **Recursively delete each edge with support of less than *k*-2.**
4. Delete the isolated vertices.

$k = 4$



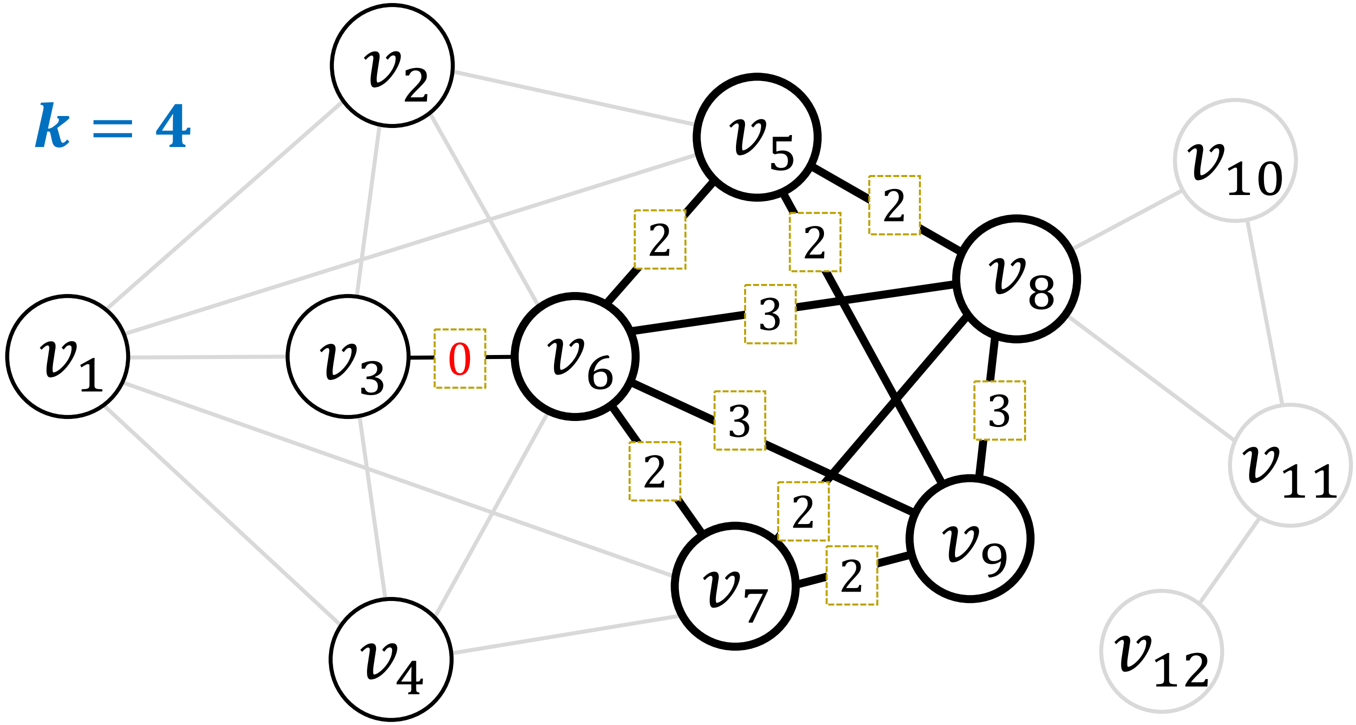
K-Truss Computation

1. Compute the (*k*-1)-core.
2. Compute the support of each edge.
3. **Recursively delete each edge with support of less than *k*-2.**
4. Delete the isolated vertices.



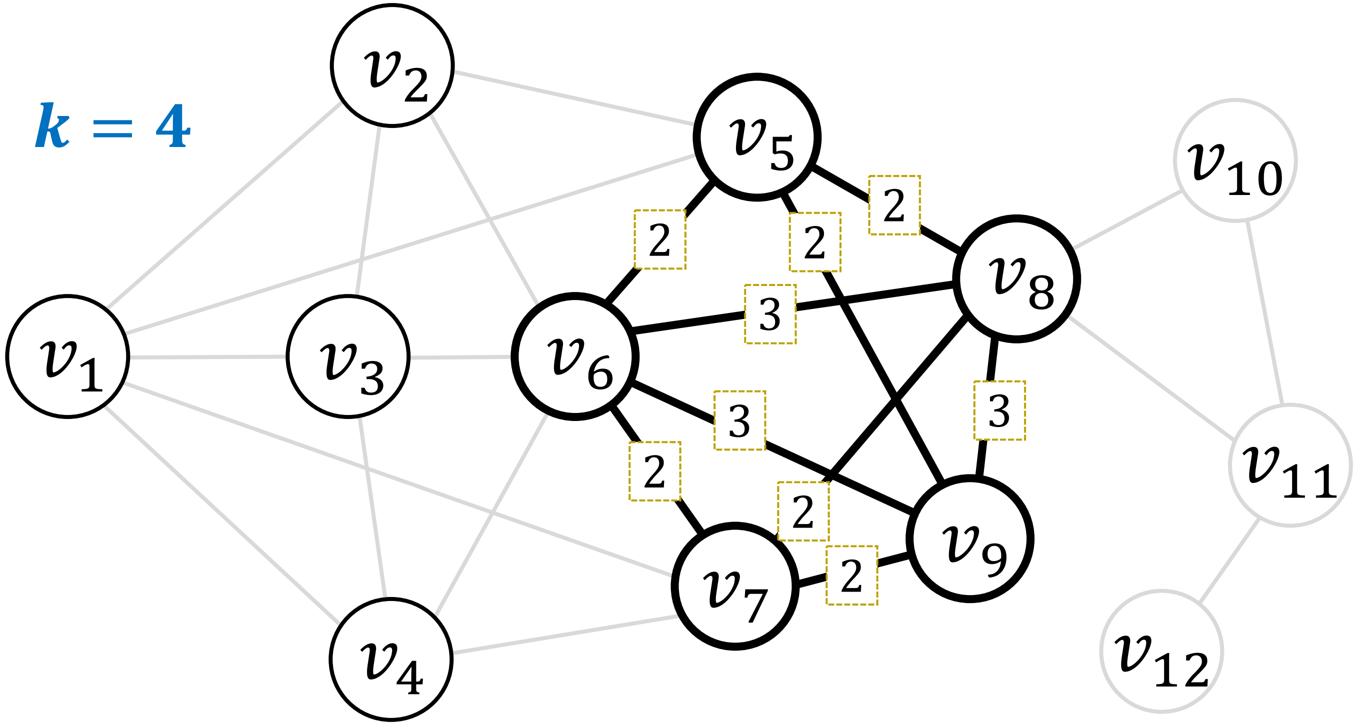
K-Truss Computation

1. Compute the (k-1)-core.
2. Compute the support of each edge.
3. **Recursively delete each edge with support of less than *k*-2.**
4. Delete the isolated vertices.



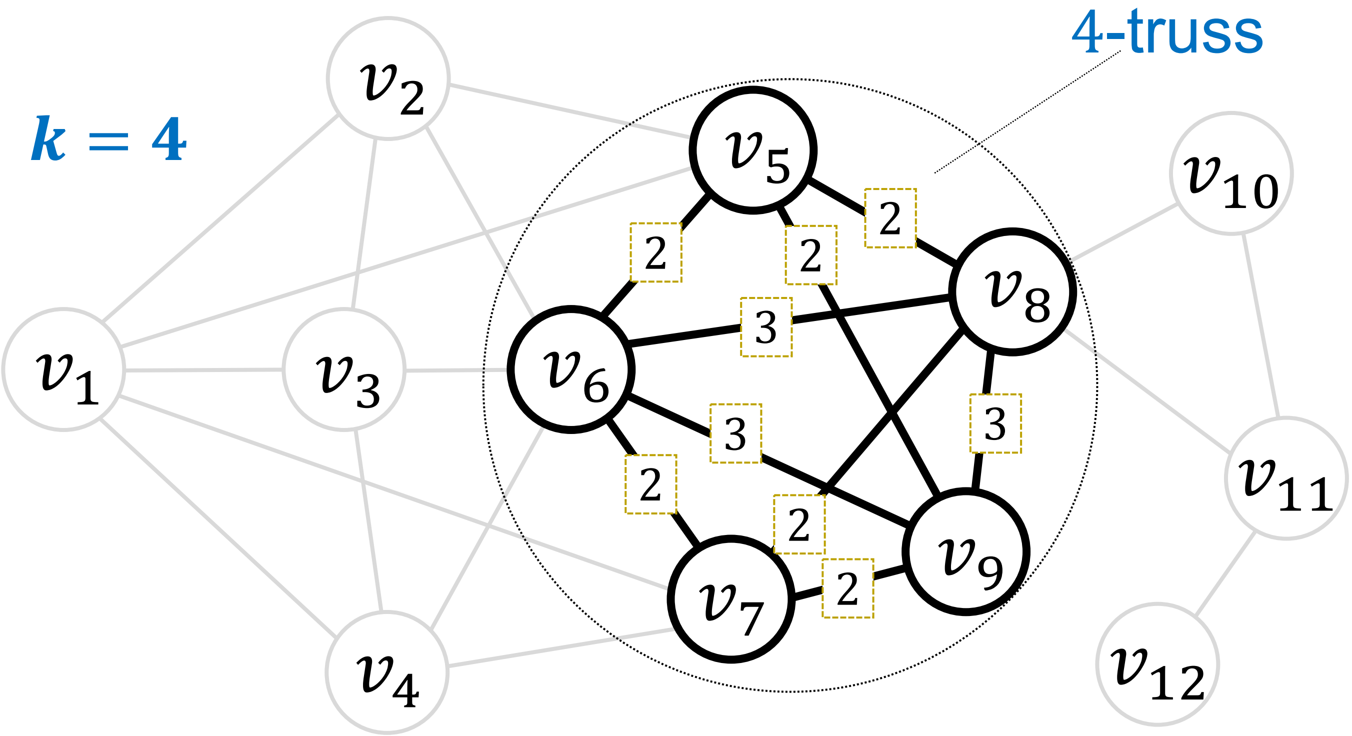
K-Truss Computation

1. Compute the (k-1)-core.
2. Compute the support of each edge.
3. **Recursively delete each edge with support of less than *k*-2.**
4. Delete the isolated vertices.

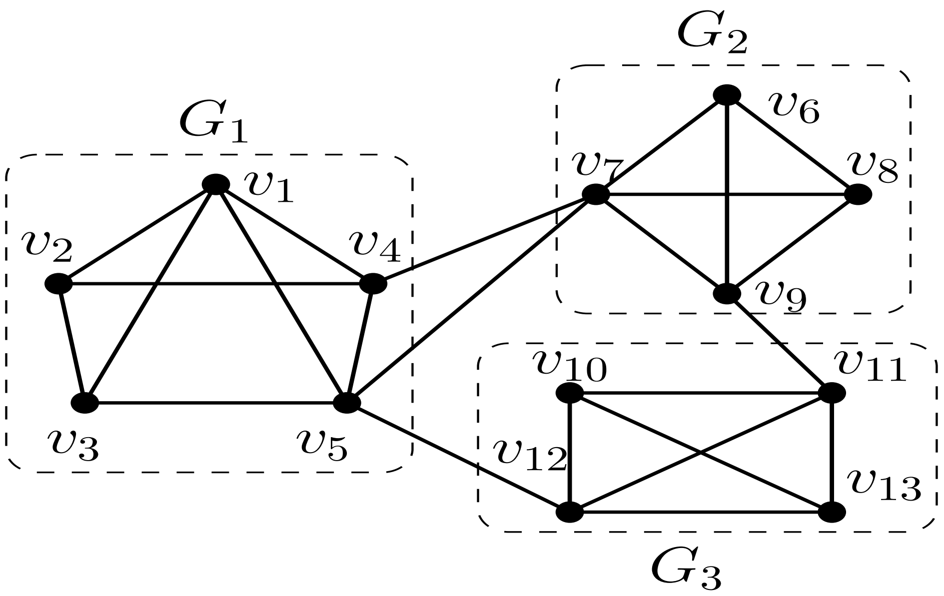


K-Truss Computation

1. Compute the (k+1)-core.
2. Compute the support of each edge.
3. Recursively delete each edge with support of less than k-2.
4. **Delete the isolated vertices.**



K-Edge Connected Component

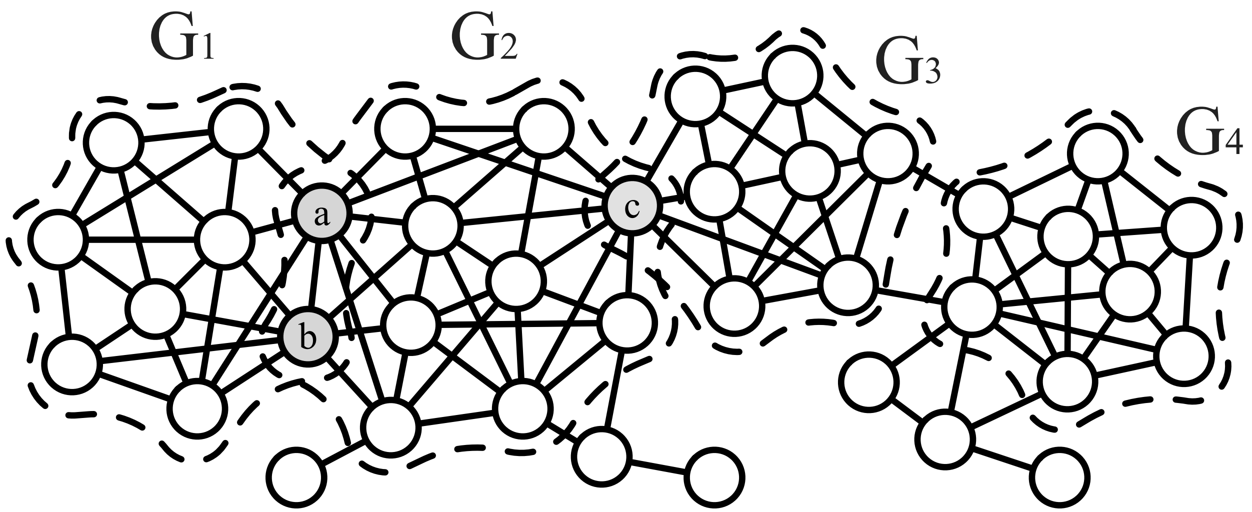


3-ECCs

A graph is ***k*-edge connected** if it is still connected after removing any set of $k-1$ edges from it.

A ***k*-Edge Connected Component (k-ECC)** is a maximal k-edge connected subgraph

K-Vertex Connected Component



4-Core: {G1∪G2∪G3∪G4} 4-ECC: {G1∪G2∪G3, G4} 4-VCC: {G1, G2, G3, G4}

A graph is ***k*-vertex connected** if it is still connected after removing any set of $k-1$ vertices from it.

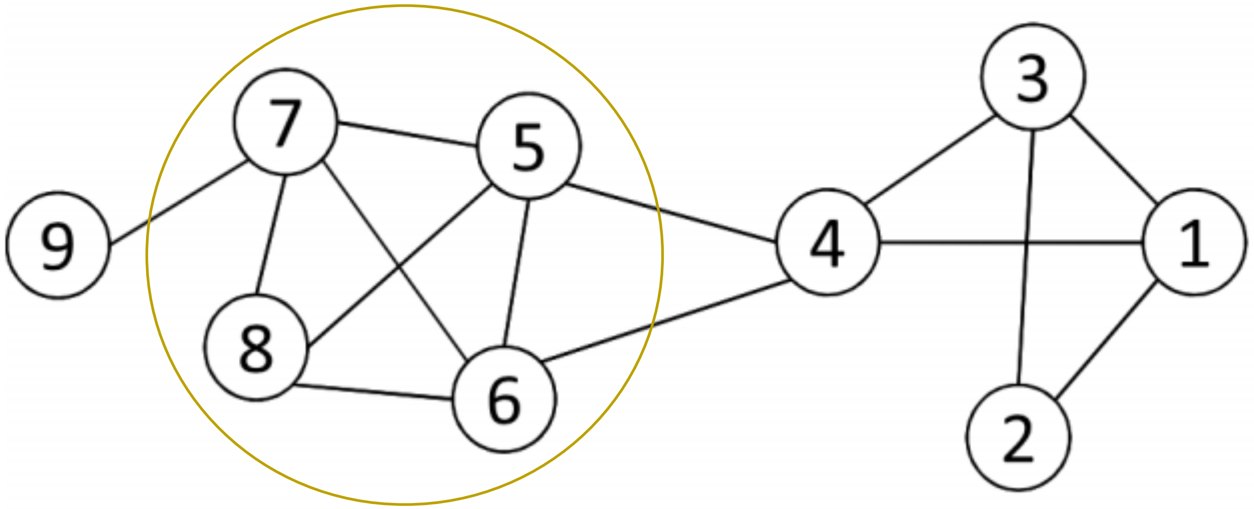
A ***k*-Vertex Connected Component (k-VCC)** is a maximal k-vertex connected subgraph.

Cliques

Every pair of vertices pair is connected

A clique is called maximal clique if there exist no other bigger cliques that contain it

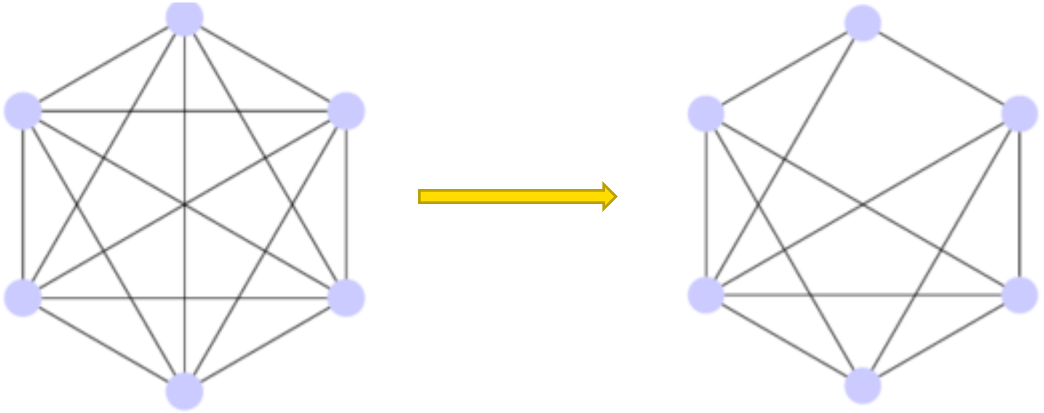
Also called complete graph



R. D. Luce and A. D. Perry, “A method of matrix analysis of group structure,” *Psychometrika*, vol. 14, no. 2, pp. 95–116, 1949

Variations of cliques: quasi-clique

Quasi-clique: relax on density or degree.



$\gamma = 0.8$

total # of edges:

$\frac{|V|(|V|-1)}{2}$ to

$\gamma \frac{|V|(|V|-1)}{2}$

H. Matsuda, T. Ishihara, A. Hashimoto. Classifying molecular sequences using a linkage graph with their pairwise similarities. Theor. Comput. Sci., 1999

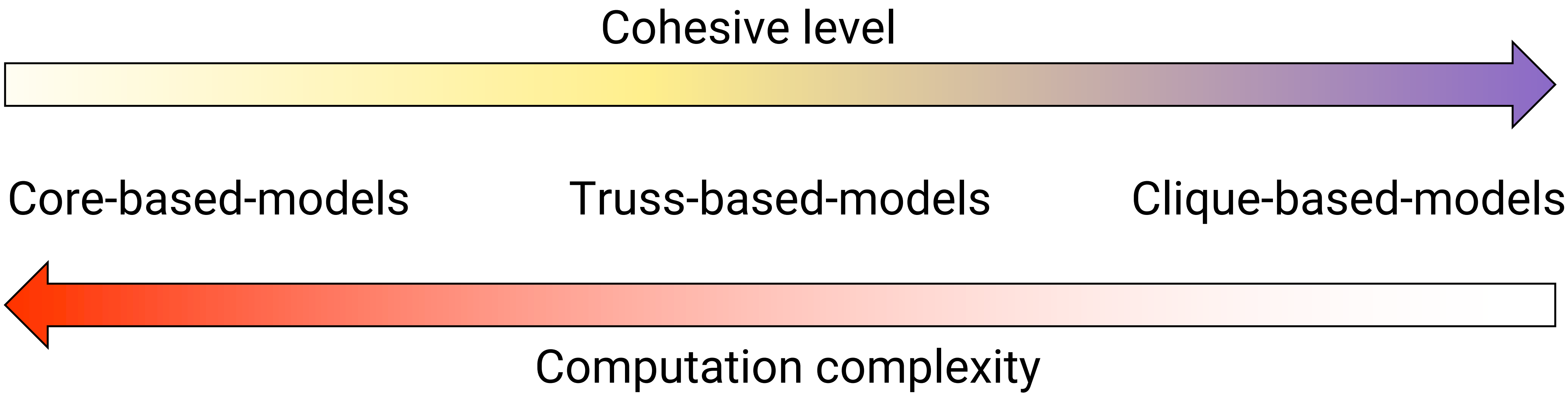
Cohesive Subgraph Models

Global cohesiveness: cliques and variants

Node and edge constraints: k-core, k-truss

Connectivity: k-ECC, k-VCC

Comparison of models



Danhao Ding, Hui Li, Zhipeng Huang, and Nikos Mamoulis.2017. Efficient fault-tolerant group recommendation using alpha-beta-core. In Proceedings of the 2017 ACM on Conference on Information and Knowledge Management.2047–2050.

Z.Zou, “Bitruss decomposition of bipartite graphs,” in DASFAA. Springer, 2016, pp. 218–233.

Yun Zhang, Charles A Phillips, Gary L Rogers, Erich J Baker, Elissa J Chesler, and Michael A Langston. 2014. On finding bicliques in bipartite graphs:a novel algorithm and its application to the integration of diverse biological data types. BMC bioinformatics 15,1(2014),110.

Learning Outcome

- Know the definition of the introduced cohesive subgraph models

- Understand the algorithms to compute k-core and the process to compute k-truss