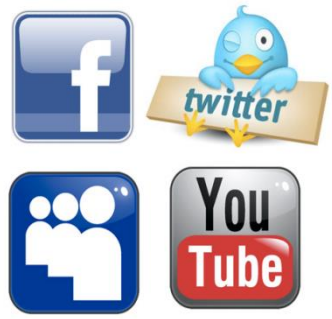
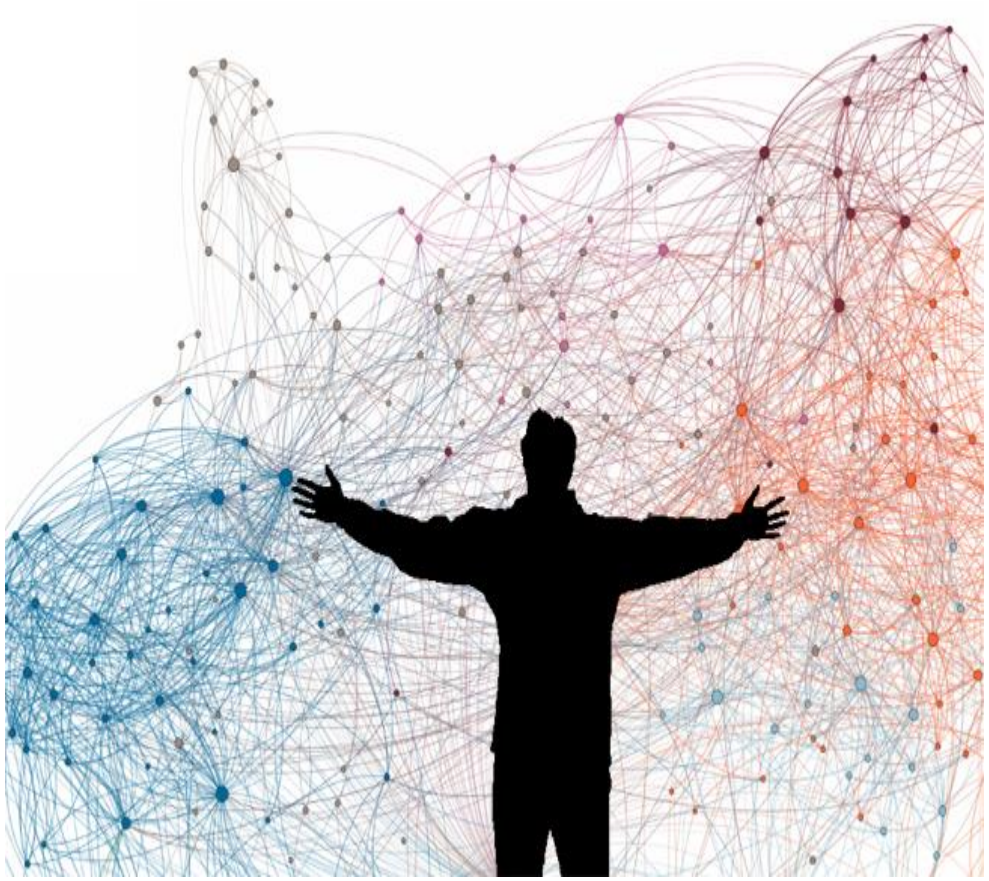
I/O Efficient Core Graph Decomposition at Web Scale

Jeffrey Xu Yu (于旭)
The Chinese University of Hong Kong
yu@se.cuhk.edu.hk, http://www.se.cuhk.edu.hk/~yu

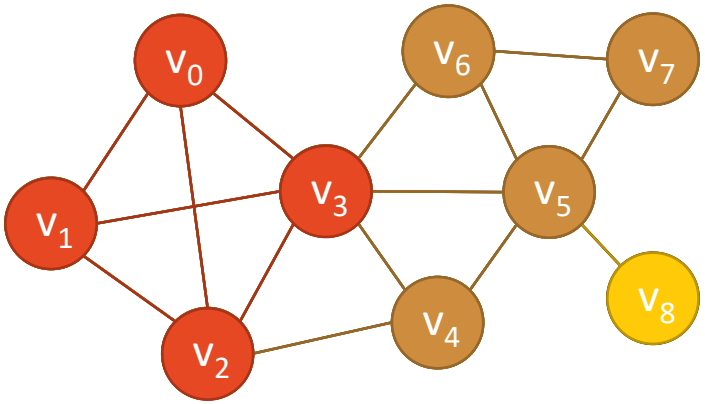
Big Graphs/Networks





K-Core and Core-Number

- The k-core of graph G is a maximal subgraph in which every node has a degree of at least k.
- The core number of a node v is the largest k, such that v is contained in a k-core.
- Compute the k-cores of G for all $1 \leq k \leq k_{max}$.
- Computing k-cores is the same as to compute core numbers.



Core-Decomposition of Graph G

1-Core = { v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8 }

2-Core = { v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7 }

3-Core = { v_0, v_1, v_2, v_3 }

Why K-Core?

- We can obtain a rather smaller graph to compute the maximum clique using k-core.
- It is known that a k-clique is ensured to be in a $(k-1)$-core.
- Core decomposition can be used to give a $\frac{1}{2}$-approximation algorithm for the densest subgraph problem, and a $\frac{1}{3}$-approximation algorithm for the densest at-least-k-subgraph problem.
- We can compute k-cores in linear for a graph.

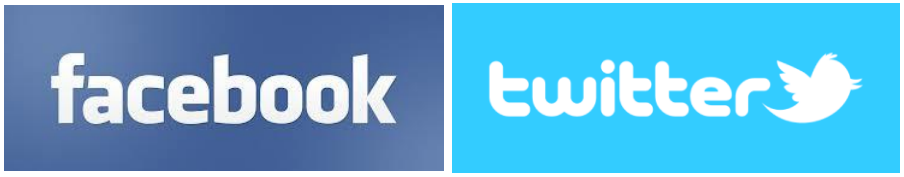
The Core Decomposition

- In-Memory Core Decomposition: *V. Batagelj and M. Zaversnik. An $o(m)$ algorithm for cores decomposition of networks. CoRR, cs.DS/0310049, 2003.*

  - It takes a bottom-up approach to compute smaller k-cores first.
  - It iteratively removes the nodes with minimum degree and incident edges.
  - It needs to random access nodes/edges where they are all in-memory.

Big Graph Data

- Facebook
  - 1.23 billion daily active users on average for Dec. 2016
  - 130 friends per user on average
- Twitter
  - 313M monthly active users for June, 2016.
  - 1 billion unique visits monthly to sites with embedded tweets.
- A sub-domain of the Web: Clueweb
  - 978.5 million nodes
  - 42.6 billion edges
- Hard to fit in the main memory of a single machine.



External/Semi-External

- Let M be the size of main memory.
  - **External algorithm**: $M < |G|$
  - **Semi-external algorithm**: $k \cdot |V| \leq M < |G|$

- An External Partition-based Core Decomposition: *J.Cheng, Y.Ke, S.Chu and M.T.O ̈zsu. Efficient core decomposition in massive networks. ICDE'11, 2011.*

  - Take a top-down approach.
  - Divide the whole graph into partitions on disk.
  - Estimate the upper bound of each partition.
  - Compute the exact core number for partitions.
  - Cannot bound the memory size.

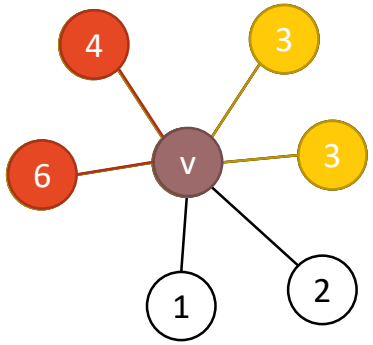
The New Semi-External Model

- The graph with largest node size in the following platforms by 2015:
  - SNAP with 79 real-world graphs: 65M vertices and 1.8G edges (FriendSter).
  - WebGraph with 75 real-world graphs: 978M vertices and 42.6G edges (ClueWeb).
- We can hold nodes in memory but not all edges.
- Keep all nodes in the memory while the edges are stored on disk.
  - Memory Size: $O(n)$

The Locality Theorem

- *A. Montresor, F. De Pellegrini, and D. Miorandi. Distributed k-core decomposition. TPDS, 24(2), 2013.*

- Let a node v has a core number k:
- There exist at least k neighbors with core number k.
- There do not exist $k+1$ neighbors with core number $k+1$.
- The $core(v)$ is: $\max k \; s.t. \; |\{u \in neighbor(v) \mid core(u) \geq k\}| \geq k$.



Core(V) = 3 — 4 neighbors with core number at least 3

Core(V) = 4 — Only 2 neighbors with core number at least 4

The Basic Algorithm

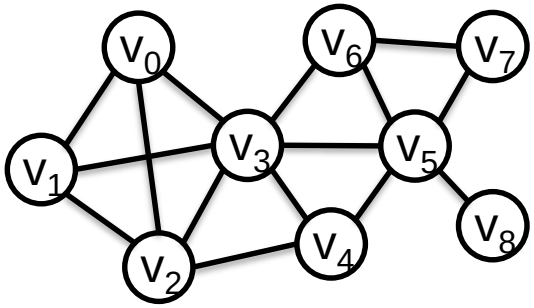
Algorithm 3 SemiCore(Graph G on Disk)

```
 1: core(v) ← deg(v) for all v ∈ V(G);
 2: update ← true;
 3: while update do
 4:     update ← false;
 5:     for v ← v_1 to v_n do
 6:         load nbr(v) from disk;
 7:         c_old ← core(v);
 8:         core(v) ← LocalCore(c_old, nbr(v));
 9:         if core(v) ≠ c_old then update ← true;
10: return core(v) for all v ∈ V(G);

11: Procedure LocalCore(c_old, nbr(v))
12: num(i) ← 0 for all 1 ≤ i ≤ c;
13: for all u ∈ nbr(v) do
14:     i ← min{c_old, core(u)};
15:     num(i) ← num(i) + 1;
16: s ← 0;
17: for k ← c_old to 1 do
18:     s ← s + num(k);
19:     if s ≥ k then break;
20: return k;
```

Memory Complexity: *O(n)*
I/O Complexity: *O(l(m+n)/B)*
Time Complexity: *O(l(m+n))*

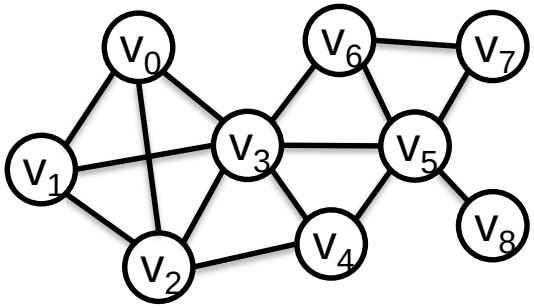
The Basic Algorithm



Initialize the core number by degree Iteration *1* Update: *False*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	4	6	3	5	3	2	1

The Basic Algorithm

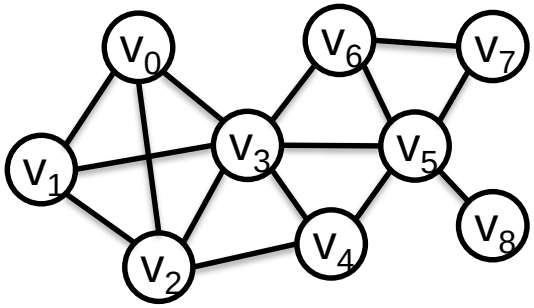


Iteration *1*

Update: *False*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	4	6	3	5	3	2	1

The Basic Algorithm

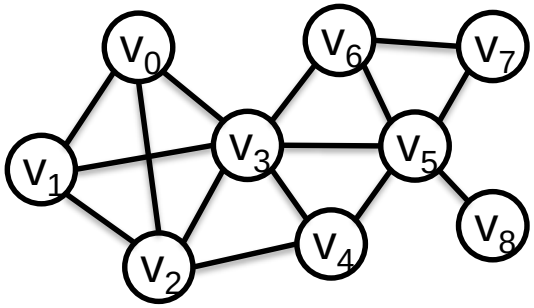


Iteration *1*

Update: *True*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	3	6	3	5	3	2	1

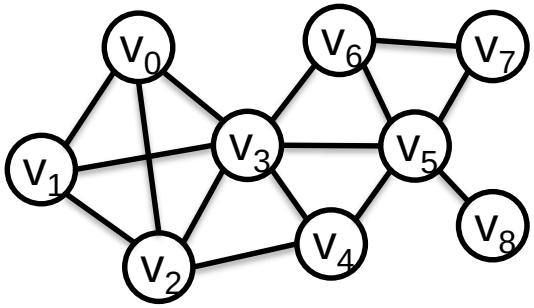
The Basic Algorithm



Iteration *1* Update: *True*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	3	3	3	5	3	2	1

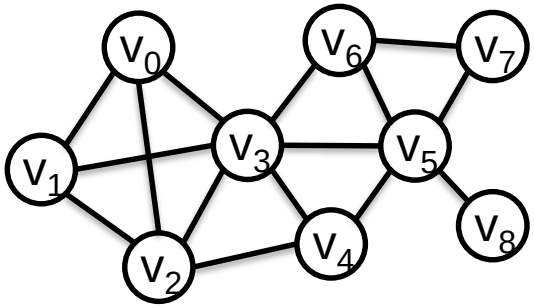
The Basic Algorithm



Iteration *1* Update: *True*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	3	3	3	5	3	2	1

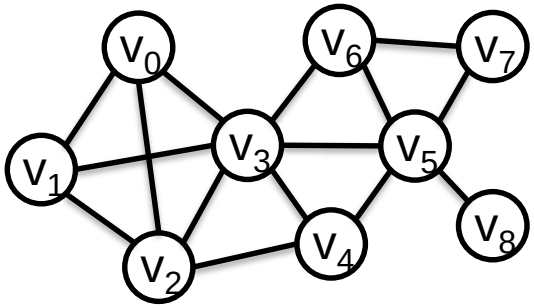
The Basic Algorithm



Iteration *1* Update: *True*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	3	3	3	3	3	2	1

The Basic Algorithm

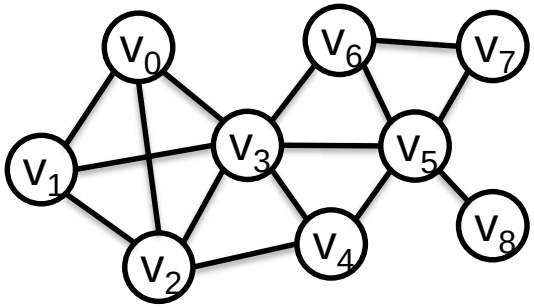


Iteration *1*

Update: *True*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	3	3	3	3	2	2	1

The Basic Algorithm

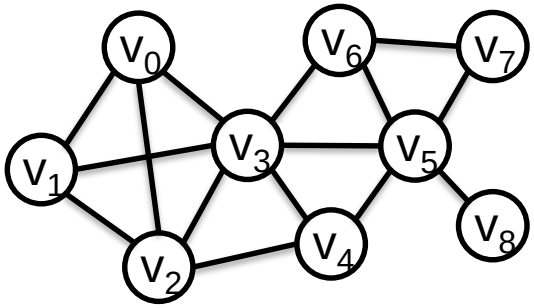


Iteration *1*

Update: *True*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	3	3	3	3	2	2	1

The Basic Algorithm

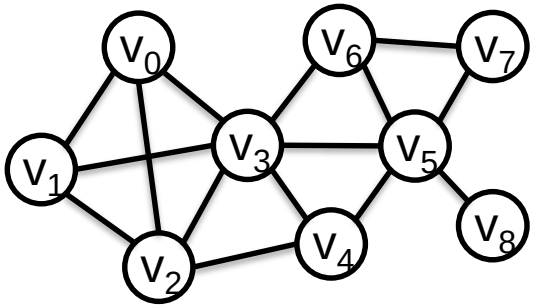


Iteration *1*

Update: *True*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	3	3	3	3	2	2	1

The Basic Algorithm

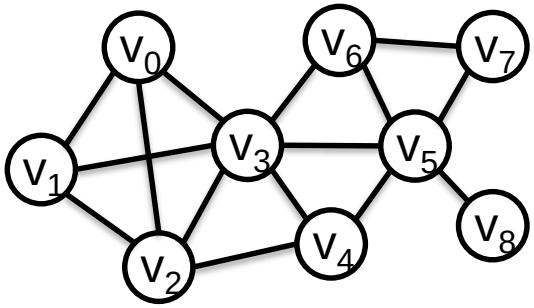


Iteration *1*

Update: *True*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	3	3	3	3	2	2	1

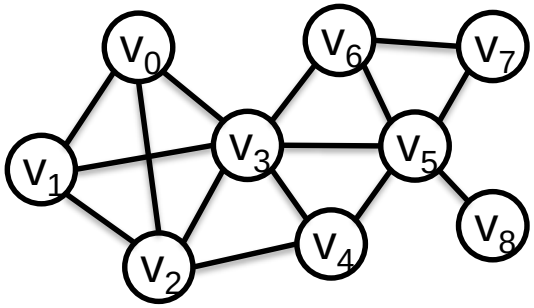
The Basic Algorithm



Iteration *2* Update: *False*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	3	3	3	3	2	2	1

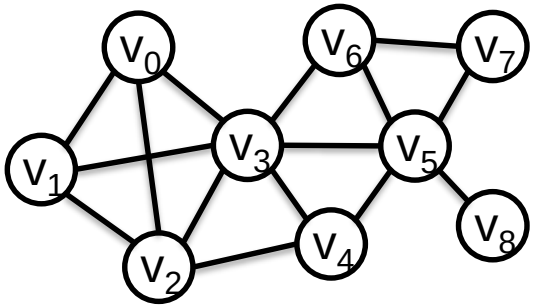
The Basic Algorithm



Iteration *2* Update: *True*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	3	3	3	2	2	2	1

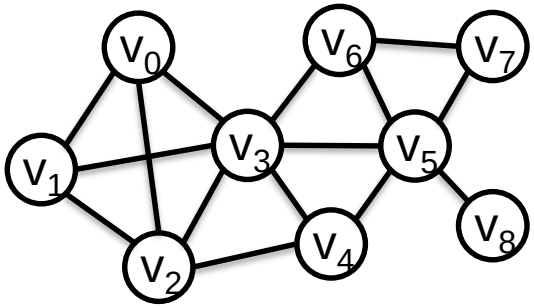
The Basic Algorithm



Iteration *3* Update: *True*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	3	3	2	2	2	2	1

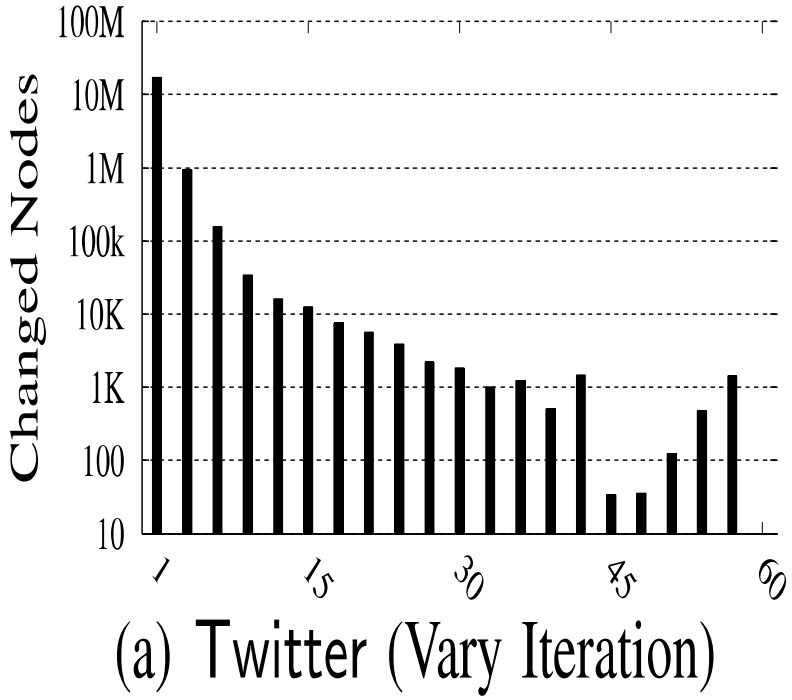
The Basic Algorithm



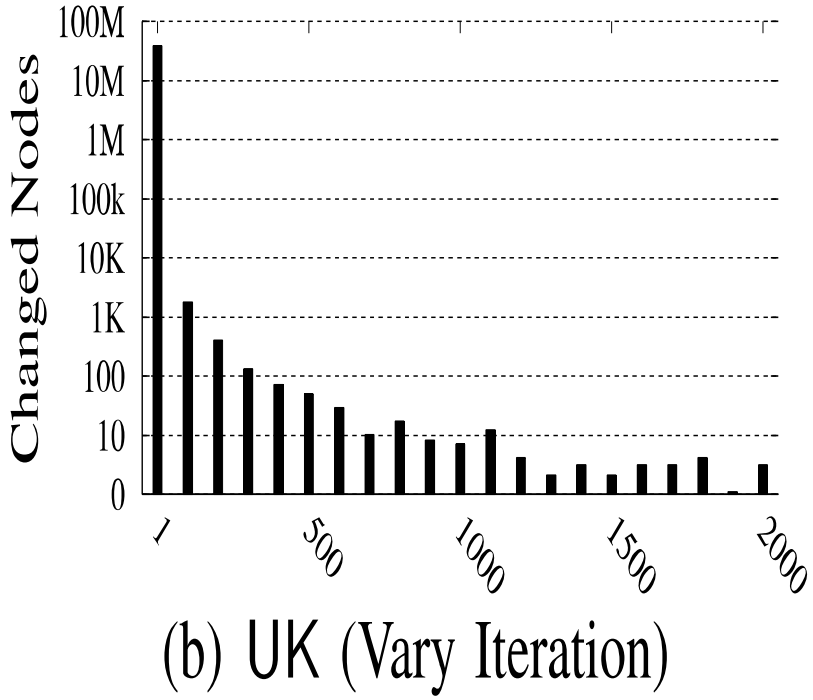
Iteration *4* Update: *False*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	3	3	2	2	2	2	1

Reduce Node Computation



(a) Twitter (Vary Iteration)



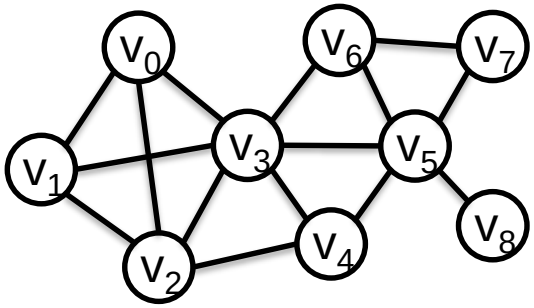
(b) UK (Vary Iteration)

- If we guarantee that $\overline{core}(v)$ is unchanged after recomputation, there is no need to load the v's neighbors from disk and ercompute $\overline{core}(v)$.

When to update?

- Recall $core(v)$: $\max k \; s.t. |\{u \in neighbor(v) \mid core(u) \geq k\}| \geq k$.
- The $core(v)$ is determined by the number of neighbors u with $core(u) \geq core(v)$.
- We define it by $\boldsymbol{cnt(v)} = |\{u \in neighbor(v) \mid \overline{core}(u) \geq \overline{core}(v)\}|$.
- We only need to update $\overline{core}(v)$ if $cnt(v) < \overline{core}(v)$.

The Improved Algorithm

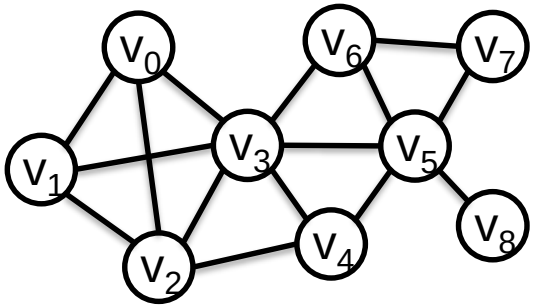


Update core number if $cnt(v) < \overline{core}(v)$

Initialize the core number by degree Iteration *1* Update: *False*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	4	6	3	5	3	2	1
cnt	3	0	0	0	0	0	0	0	0

The Improved Algorithm

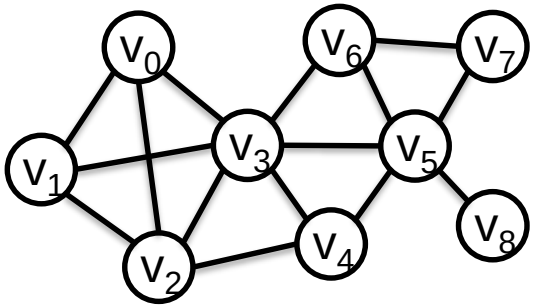


Update core number if $cnt(v) < \overline{core}(v)$

Iteration *1* Update: *False*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	4	6	3	5	3	2	1
cnt	3	3	0	0	0	0	0	0	0

The Improved Algorithm

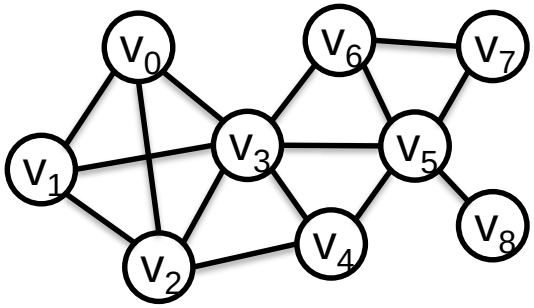


Update core number if $cnt(v) < \overline{core}(v)$

Iteration *1* Update: *True*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	3	6	3	5	3	2	1
cnt	3	3	4	0	0	0	0	0	0

The Improved Algorithm

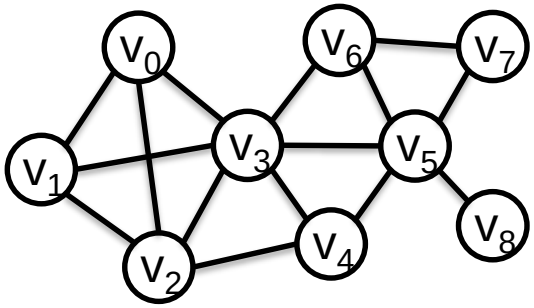


Update core number if $cnt(v) < \overline{core}(v)$

Iteration *1* Update: *True*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	3	3	3	5	3	2	1
cnt	3	3	4	6	0	0	0	0	0

The Improved Algorithm

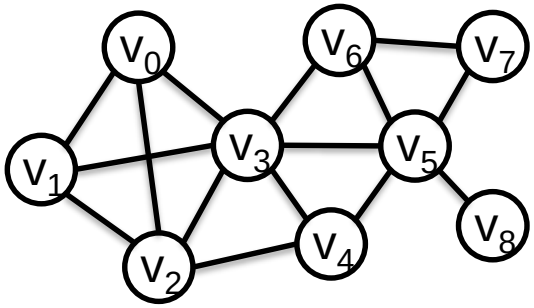


Update core number if $cnt(v) < \overline{core}(v)$

Iteration *1* Update: *True*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	3	3	3	5	3	2	1
cnt	3	3	4	6	3	0	0	0	0

The Improved Algorithm

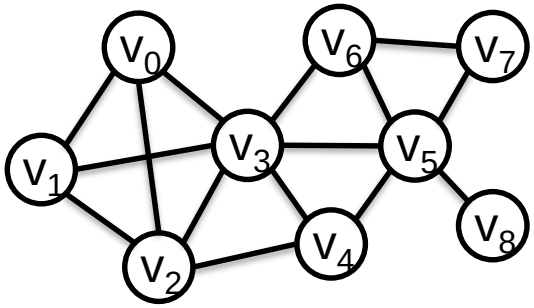


Update core number if $cnt(v) < \overline{core}(v)$

Iteration *1* Update: *True*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	3	3	3	3	3	2	1
cnt	3	3	4	6	3	3	0	0	0

Improved Algorithm

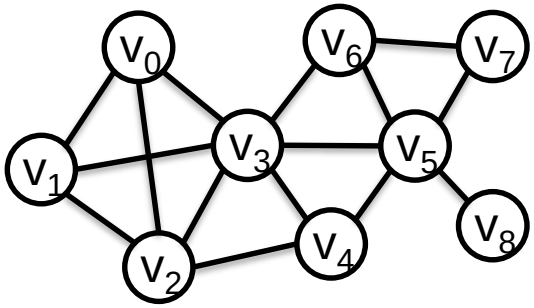


Update core number if $cnt(v) < \overline{core}(v)$

Iteration *1* Update: *True*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	3	3	3	3	2	2	1
cnt	3	3	4	5	3	2	3	0	0

The Improved Algorithm

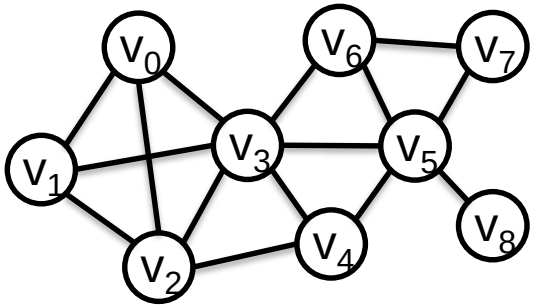


Update core number if $cnt(v) < \overline{core}(v)$

Iteration *1* Update: *True*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	3	3	3	3	2	2	1
cnt	3	3	4	5	3	2	3	2	0

The Improved Algorithm

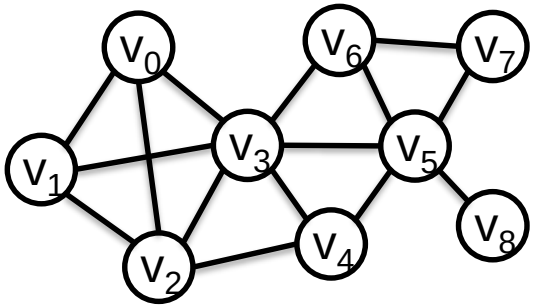


Update core number if $cnt(v) < \overline{core}(v)$

Iteration *1* Update: *True*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	3	3	3	3	2	2	1
cnt	3	3	4	5	3	2	3	2	1

The Improved Algorithm

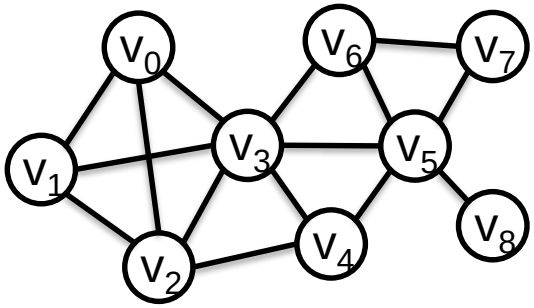


Update core number if $cnt(v) < \overline{core}(v)$

Iteration *1* Update: *True*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	3	3	3	3	2	2	1
cnt	3	3	4	5	3	2	3	2	1

The Improved Algorithm

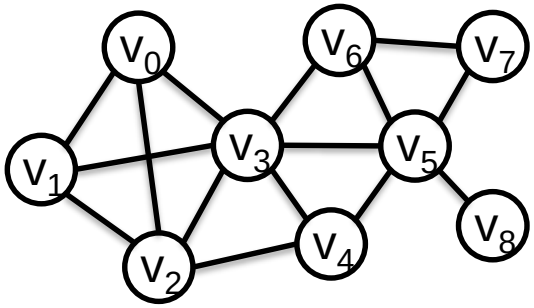


Update core number if $cnt(v) < \overline{core}(v)$

Iteration *2* Update: *True*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	3	3	3	2	2	2	1
cnt	3	3	4	5	2	4	3	2	1

The Improved Algorithm

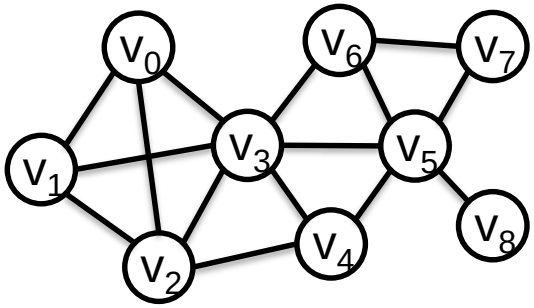


Update core number if $cnt(v) < \overline{core}(v)$

Iteration: *2* Update: *True*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	3	3	3	2	2	2	1
cnt	3	3	4	5	2	4	3	2	1

The Improved Algorithm

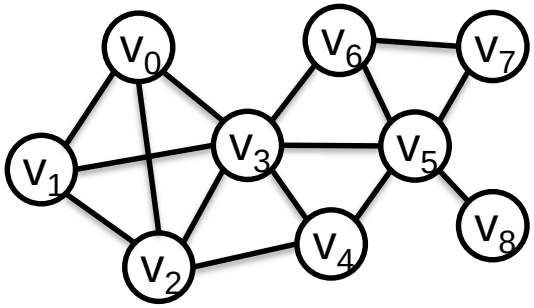


Update core number if $cnt(v) < \overline{core}(v)$

Iteration *3* Update: *True*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	3	3	2	2	2	2	1
cnt	3	3	4	5	3	4	3	2	1

The Improved Algorithm

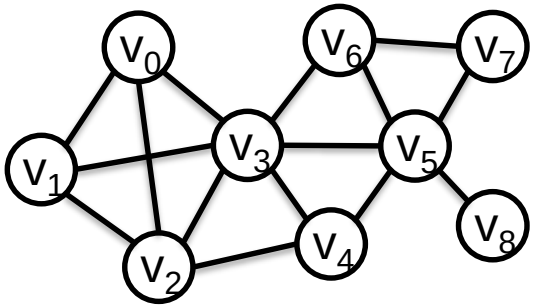


Update core number if $cnt(v) < \overline{core}(v)$

Iteration *3* isContinu *True*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	3	3	2	2	2	2	1
cnt	3	3	4	5	3	4	3	2	1

The Improved Algorithm



Update core number if $cnt(v) < \overline{core}(v)$

Iteration *4* Update: *False*

ID	0	1	2	3	4	5	6	7	8
$\overline{core}$	3	3	3	3	2	2	2	2	1
cnt	3	3	4	5	3	4	3	2	1

Algorithm Analysis

- Bounded Memory.
  - Follow the semi-external model and require only $O(n)$ memory.
- Read I/O Only.
  - Only require read I/Os by scanning the node and edge tables sequentially on disk.
- Simple In-memory Structure and Data Access.
  - Use two arrays.
  - Even more efficient than the in-memory algorithm.

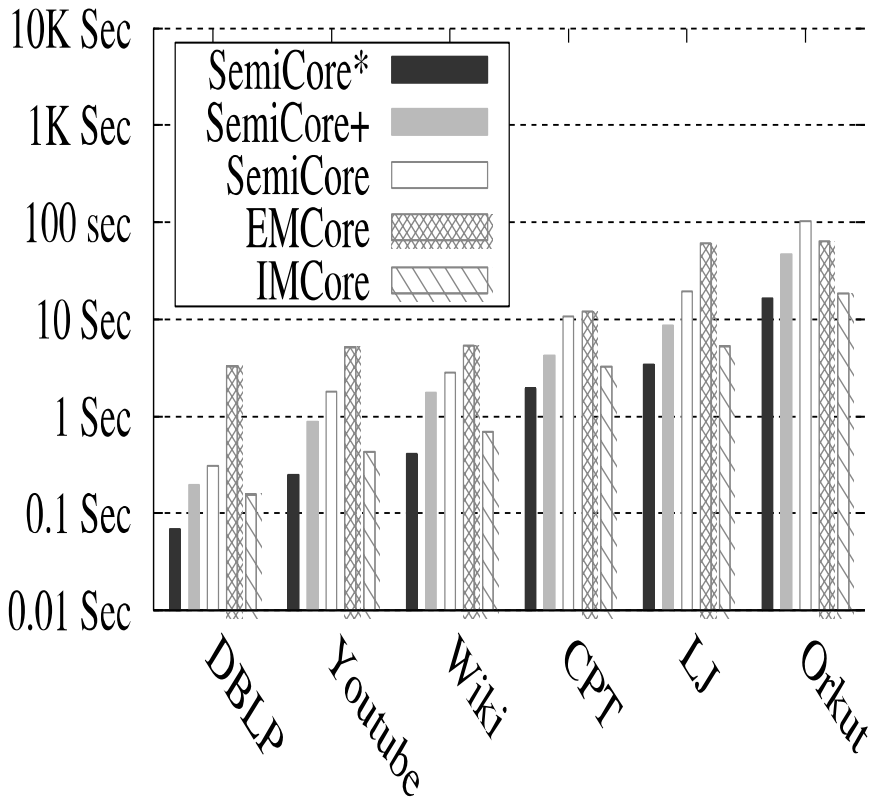
Performance Studies

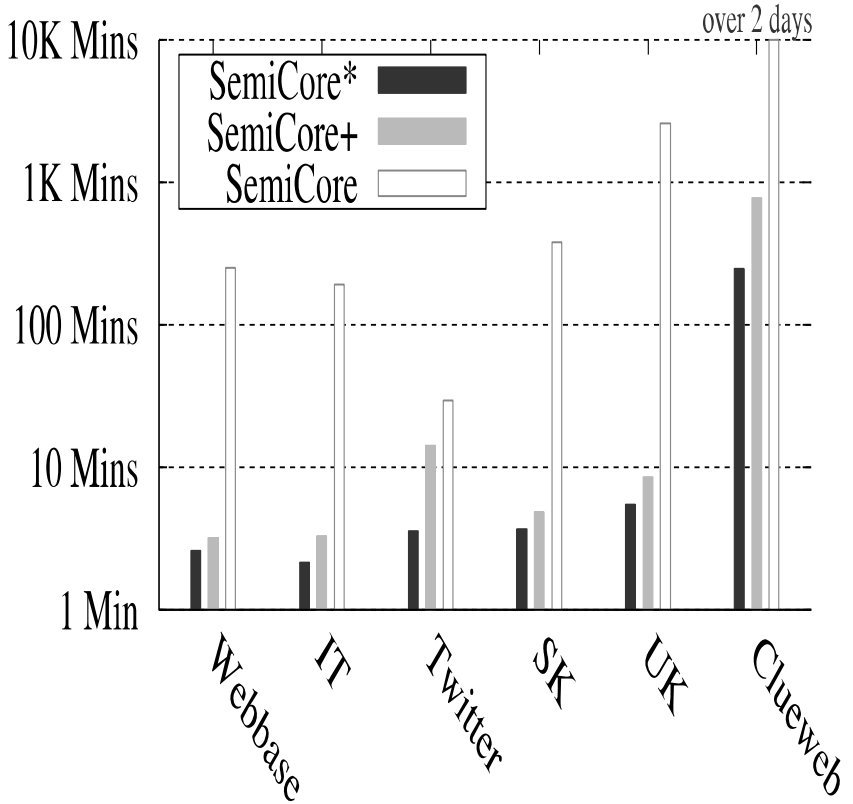
- Intel Xeon 3.4GHz CPU, 16GB RAM and 7200 RPM SATA Hard Drives (2TB)

Datasets

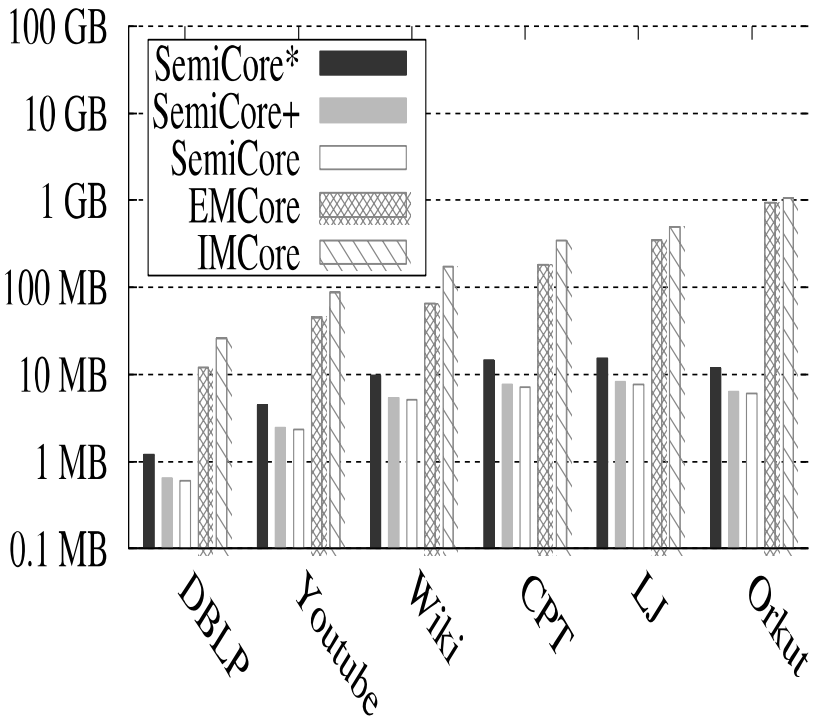
Datasets	|V|	|E|	density	k_{max}
DBLP	317,080	1,0490,866	3.31	113
Youtube	1,134,890	2,987,624	2.63	51
WIKI	2,394,385	5,021,410	2.10	131
Citation	3,774,768	16,518,948	4.38	64
LiveJournal	3,997,962	34,681,189	8.67	360
Orkut	3,072,441	117,185,083	38.14	253
Webbase	118,142,155	1,019,903,190	8.63	1,506
IT	41,291,594	1,150,725,436	27.86	3,224
Twitter	41,652,230	1,468,365,182	35.25	2,488
SK	50,636,154	1,949,412,601	38.49	4,510
UK	105,896,555	3,738,733,648	35.30	5,704
Clueweb	978,408,098	42,574,107,469	43.51	4,244

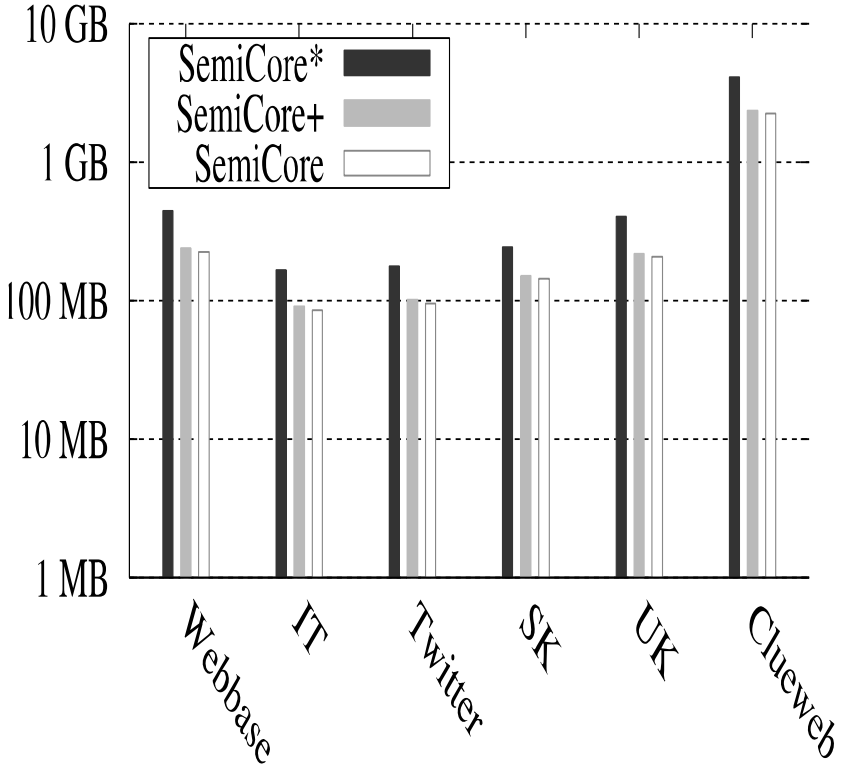
The Efficiency



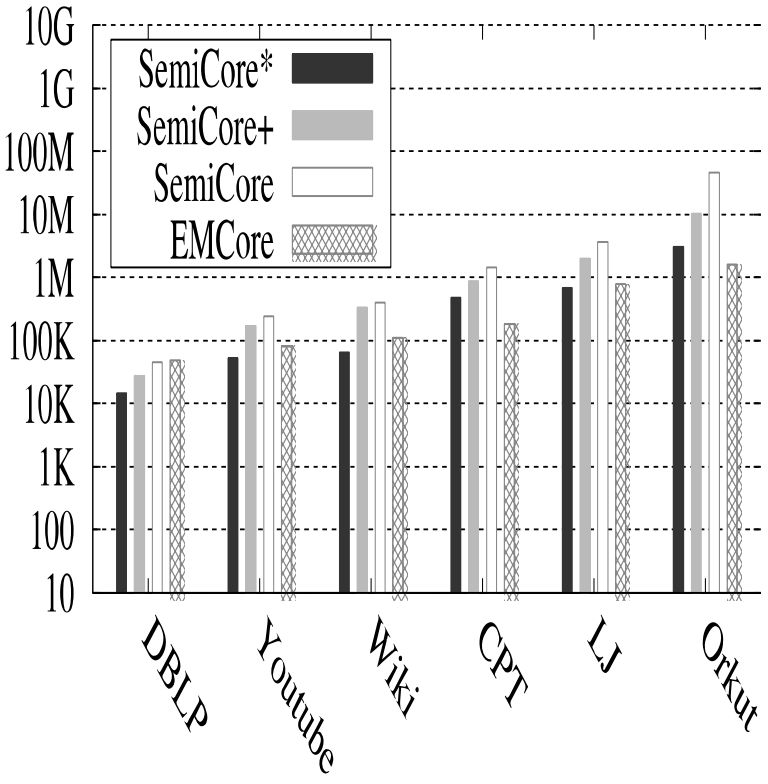


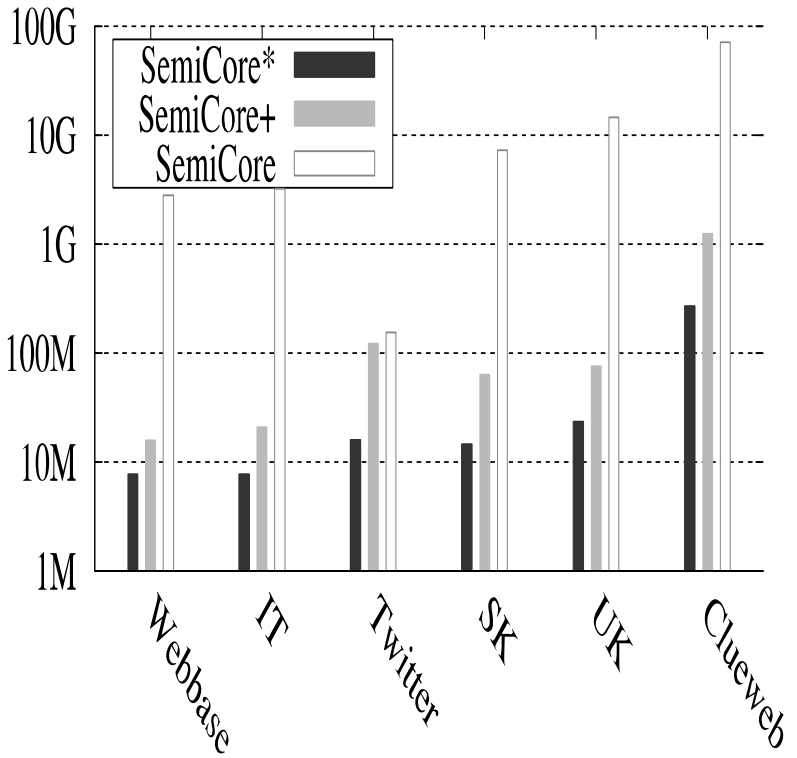
The Memory Size



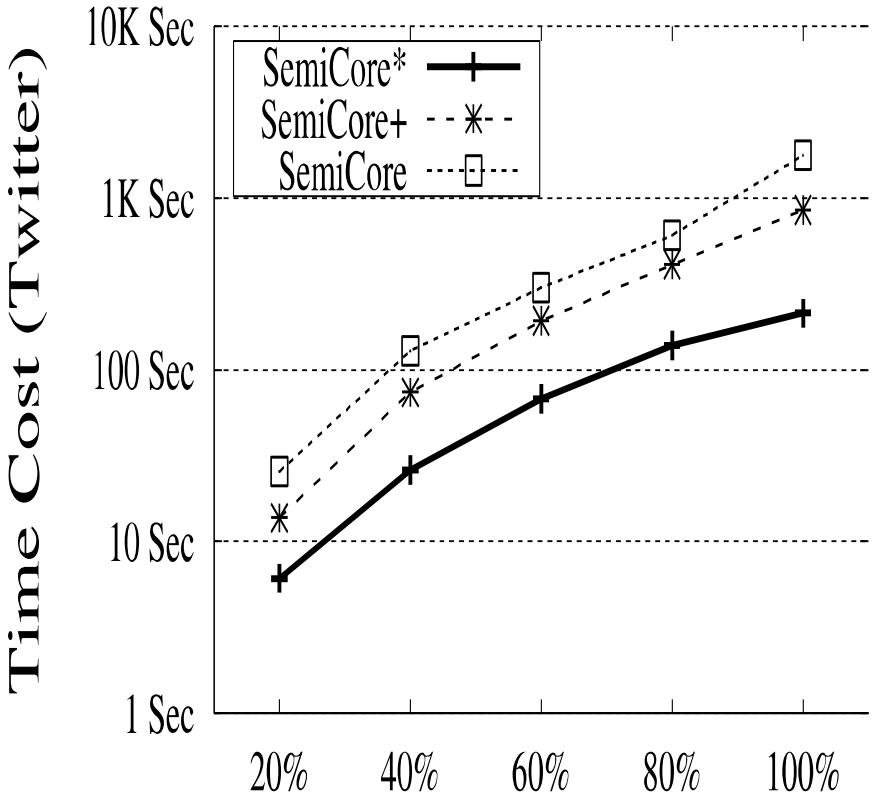


The I/O Cost

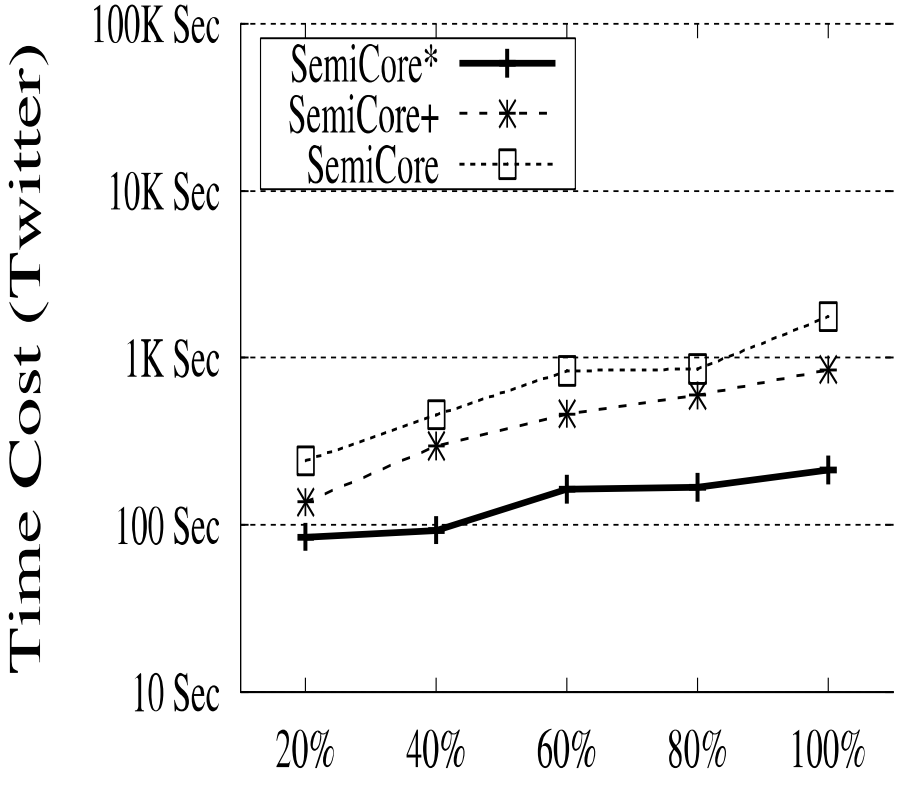

10G
1G
100M
10M
1M
100K
10K
1K
100
10
SemiCore*
SemiCore+
SemiCore
EMCore
DBLP
Youtube
Wiki
CPT
LJ
Orkut


100G
10G
1G
100M
10M
1M
SemiCore*
SemiCore+
SemiCore
Webbase
IT
Twitter
SK
UK
Clueweb

The Scalability



Vary |*V*| (Twitter)



Vary |*E*| (Twitter)

The Conclusion

- The first I/O efficient core decomposition algorithm with memory guarantee.
- Several optimization strategies to largely reduce the I/O and CPU cost.
- The first I/O efficient core decomposition algorithm to handle graph updates.

Acknowledgements

- Dong Wen
- Lu Qin
- Ying Zhang
- Xuemin Lin

ありがとうございました！