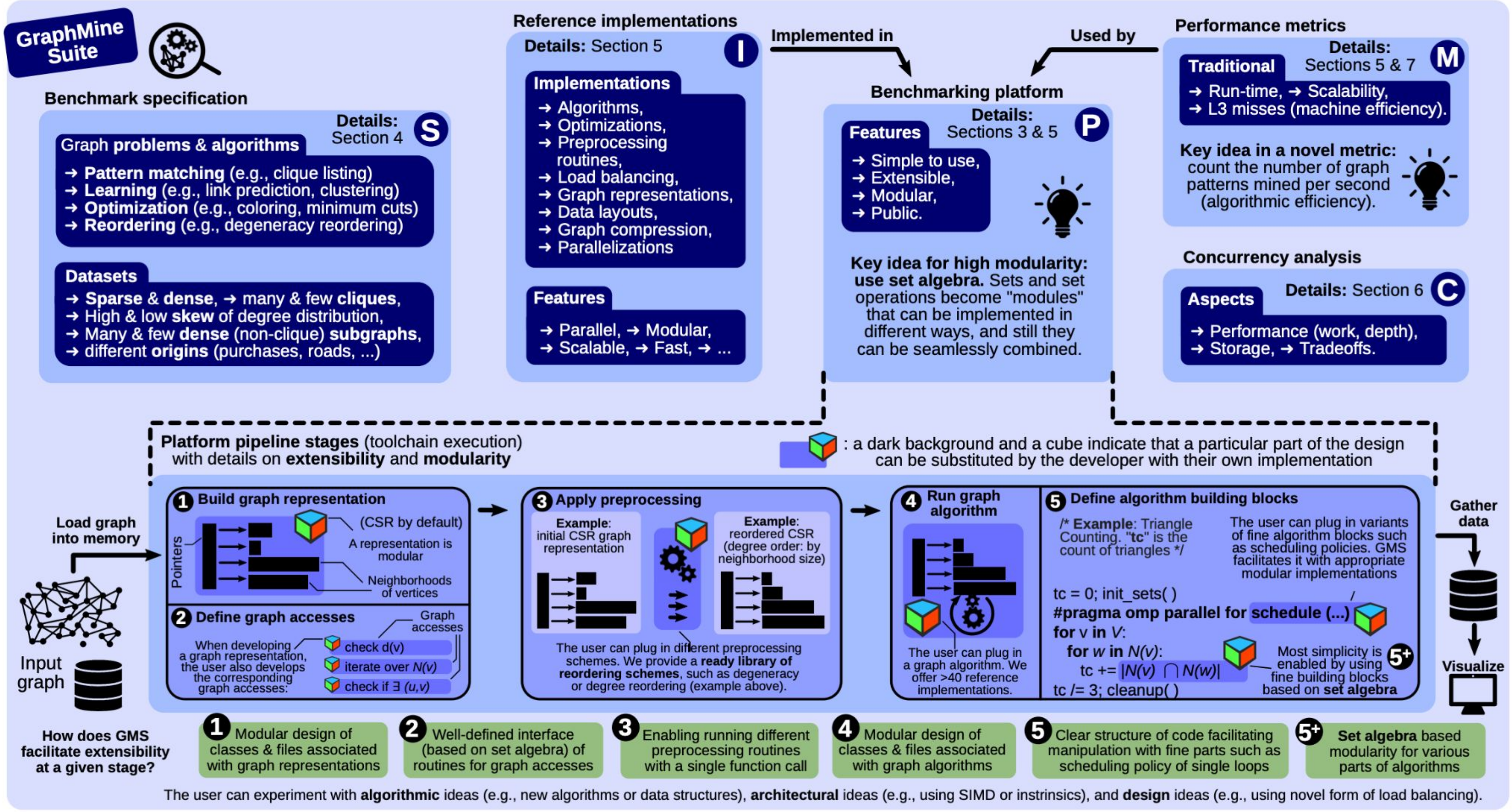
GraphMineSuite: Enabling High-Performance and Programmable Graph Mining Algorithms with Set Algebra

Maciej Besta[1*], Zur Vonarburg-Shmaria[1], Yannick Schaffner[1], Leonardo Schwarz[1], Grzegorz Kwasniewski[1], Lukas Gianinazzi[1], Jakub Beranek[2], Kacper Janda[3], Tobias Holenstein[1], Sebastian Leisinger[1], Peter Tatkowski[1], Esref Ozdemir[1], Adrian Balla[1], Marcin Copik[1], Philipp Lindenberger[1], Marek Konieczny[3], Onur Mutlu[1], Torsten Hoefler[1*]

[1]ETH Zurich, Zurich, Switzerland; [2]VSB, Ostrava, Czech Republic; [3]AGH-UST, Krakow, Poland; [*]Corresponding authors

What is graph mining?

- Graph mining is the process of finding and extracting useful information from graphs, i.e. sssp, triangle counting, k-cliques, maximal cliques, etc.
- Many real world applications: social sciences, bioinformatics, chemistry, medicine, cybersecurity, and many others
- Issue #1: graphs can be very large and require a lot of compute power
- Solution #1: Parallelism!
- Issue #2: Too many choices!
    - Hard to keep up and find relevant baseline graph mining algorithms to improve upon, a plethora of relevant datasets, numerous design choices
- Solution #2: **GraphMineSuite (GMS)** - a benchmarking suite for high-performance graph mining algorithms.



Picture taken from paper

Benchmark specification: Graph Problems

	Graph problem	Corresponding algorithms	E.?	P.?	Why included, what represents? (selected remarks)
Graph Pattern Matching	• Maximal Clique Listing [48]	Bron-Kerbosch [24] + optimizations (e.g., pivoting) [29, 51, 117]	🖒 ❺+	👎	Widely used, NP-complete, example of backtracking
	• k-Clique Listing [41]	Edge-Parallel and Vertex-Parallel general algorithms [41], different variants of Triangle Counting [104, 107]	🖒 ❺+	👎	P (high-degree polynomial), example of backtracking
	• Dense Subgraph Discovery [5]	Listing k-clique-stars [63] and k-cores [54] (exact & approximate)	🖒 ❺+	👎	Different relaxations of clique mining
	• Subgraph isomorphism [48]	VF2 [40], TurboISO [58], Glasgow [89], VF3 [26, 28], VF3-Light [27]	👍	👎	Induced vs. non-induced, and backtracking vs. indexing schemes
	• Frequent Subgraph Mining [5]	BFS and DFS exploration strategies, different isomorphism kernels	👍	👎	Useful when one is interested in many different motifs
Graph Learning	• Vertex similarity [75]	Jaccard, Overlap, Adamic Adar, Resource Allocation, Common Neighbors, Preferential Attachment, Total Neighbors [101]	🖒 ❺+	👎	A building block of many more comples schemes, different methods have different performance properties
	• Link Prediction [114]	Variants based on vertex similarity (see above) [7, 80, 83, 114], a scheme for assessing link prediction accuracy [121]	🖒 ❺+	👎	A very common problem in social network analysis
	• Clustering [103]	Jarvis-Patrick clustering [65] based on different vertex similarity measures (see above) [7, 80, 83, 114]	🖒 ❺+	👎	A very common problem in general data mining; the selected scheme is an example of overlapping and single-level clustering
	• Community detection	Label Propagation and Louvain Method [108]	👍	👎	Examples of convergence-based on non-overlapping clustering
Vertex Ordering	• Degree reordering	A straightforward integer parallel sort	👍	🖒	A simple scheme that was shown to bring speedups
	• Triangle count ranking	Computing triangle counts per vertex	🖒 ❺+	🖒	Ranking vertices based on their clustering coefficient
	• Degenerecy reordering	Exact and approximate [54] [70]	🖒 ❺+	🖒	Often used to accelerate Bron-Kerbosch and others

Table 3: Graph problems/algorithms considered in GMS. "E.? (Extensibility)" indicates how extensible given implementations are in the GMS benchmarking platform: "🖒" indicates full extensibility, including the possibility to provide new building blocks based on set algebra (❶ – ❺, ❺+). "👍": an algorithm that does not straightforwardly (or extensively) use set algebra. "P.? (Preprocessing) indicates if a given algorithm can be seamlessly used as a preprocessing routine; in the current GMS version, this feature is reserved for vertex reordering.

Set Algebra

- Many graph algorithms are/can be formulated with set algebra
- GMS allows users to implement their own sets, set operations, set elements, and set algebra based graph representations.
- Allows users to break complex graph mining algorithms into simple building blocks, and work on these building blocks independently.

```
class Set {
public:
//In methods below, we denote "*this" pointer with A
//(1) Set algebra methods:
  Set diff(const Set &B) const; //Return a new set C = A \ B
  Set diff(SetElement b) const; //Return a new set C = A \ {b}
  void diff_inplace(const Set &B); //Update A = A \ B
  void diff_inplace(SetElement b); //Update A = A \ {b}
  Set intersect(const Set &B) const; //Return a new set C = A ∩ B
  size_t intersect_count(const Set &B) const; //Return |A ∩ B|
  void intersect_inplace(const Set &B); //Update A = A ∩ B
  Set union(const Set &B) const; //Return a new set C = A ∪ B
  Set union(SetElement b) const; //Return a new set C = A ∪ {b}
  Set union_count(const Set &B) const; //Return |A ∪ B|
  void union_inplace(const Set &B); //Update A = A ∪ B
  void union_inplace(SetElement b); //Update A = A ∪ {b}
  bool contains(SetElement b) const; //Return b ∈ A ? true:false
  void add(SetElement b); //Update A = A ∪ {b}
  void remove(SetElement b); //Update A = A \ {b}
  size_t cardinality() const; //Return set's cardinality
//(2) Constructors (selected):
  Set(const SetElement *start, size_t count); //From an array
  Set(); Set(Set &&); //Default and Move constructors
  Set(SetElement); //Constructor of a single-element set
  static Set Range(int bound); //Create set {0, 1, ..., bound − 1}
//(3) Other methods:
  begin() const; //Return iterators to set's start
  end() const; //Return iterators to set's end
  Set clone() const; //Return a copy of the set
  void toArray(int32_t *array) const; //Convert set to array
  operator==; operator!=; //Set equality/inequality comparison

private:
  using SetElement = GMS::NodeId; //(4) Define a set element
}
```

Algorithm 1: The set algebra interface provided by GMS.

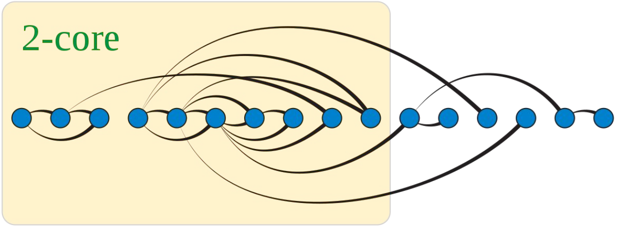
GMS Set Implementations

GMS offers three default set implementations:

- RoaringSet
  - Implemented with a “roaring bitmap” that allows for mild compression rates but inexpensive decompression
- SortedSet
- HashSet

Use Case # 1: Degeneracy Order & k-Cores

- The **degeneracy** of a graph G is the smallest d such that every subgraph in G has a vertex of degree at most d.
  - A measure of graph sparsity
- A **degeneracy ordering (DGR)** is an ordering of vertices of G such that each vertex has d or fewer neighbors that come later in this ordering
  - DGR can be obtained by repeatedly removing a vertex of minimum degree in a graph.
- A **k-core** of G is a maximal connected subgraph of G whose all vertices have degree at least k.
  - A k-core can be obtained by iterating over vertices in the DGR order, and removing vertices with out-degree less than k



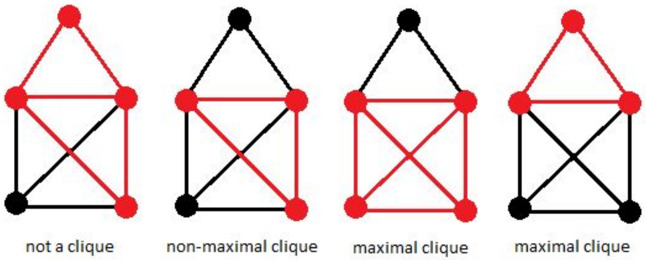
Use Case # 1: Degeneracy Order & k-Cores cont.

- Issue: Not easily parallelizable, O(n) iterations!
- Solution: GMS offers a (2+ϵ)-**approximate degeneracy order (ADG)**, O(logn) iterations for any ϵ > 0!

//**Input:** A graph G ❶. **Output:** Approx. degeneracy order (ADG) $\boldsymbol{\eta}$.
i = 1 // Iteration counter
U = V //U is the induced subgraph used in each iteration $\boldsymbol{i}$
while $U \neq \emptyset$ **do**:
    $\widehat{\delta_U} = \left(\sum_{v \in U} |N_U(v)| \right) / |U|$ ❷ //Get the average degree in U
    //R contains vertices assigned priority in this iteration:
    $R = \{v \in U : |N_U(v)| \leq (1+\varepsilon)\widehat{\delta_U}\}$ ❷
    for $v \in R$ **in parallel** ❷❺ **do**: $\eta(v)$ = i //assign the ADG order
    $U = U \setminus R$ ❺+ //Remove assigned vertices
    i = i+1

Use Case # 2: Maximal Clique Listing

- A **maximal clique** of a graph G is a fully connected subgraph of G that cannot be further extended by including one more adjacent vertex.




```
/* Input: A graph G 1 . Output: all maximal cliques. */

//Preprocessing: reorder vertices with DGR or ADG.
 (v1, v2, ..., vn) = preprocess(V, /* selected vertex order */) 3

//Main part: conduct the actual clique enumeration.
for vi ∈ (v1, v2, ..., vn) do: //Iterate over V in a specified order
  //For each vertex vi, find maximal cliques containing vi.
  //First, remove unnecessary vertices from P (candidates
  //to be included in a clique) and X (vertices definitely
  //not being in a clique) by intersecting N(vi) with vertices
  //that follow and precede vi in the applied order.
  P = N(vi) ∩ {vi+1, ..., vn} 5+ ; X = N(vi) ∩ {v1, ..., vi−1} 5+ ; R = {vi}

  //Run the Bron-Kerbosch routine recursively for P and X.
  BK-Pivot(P, {vi}, X)

BK-Pivot(P, R, X) //Definition of the recursive BK scheme
  if P ∪ X == 0 5+ : Output R as a maximal clique
  u = pivot(P ∪ X) 5+ //Choose a "pivot" vertex u ∈ P ∪ X
  for v ∈ P \ N(u) 5+ : // Use the pivot to prune search space
    //New candidates for the recursive search
    Pnew = P ∩ N(v) 5+ ; Xnew = X ∩ N(v) 5+ ; Rnew = R ∪ {v} 5+
    //Search recursively for a maximal clique that contains v
    BK-Pivot(Pnew, Rnew, Xnew)
    //After the recursive call, update P and X to reflect
    //the fact that v was already considered
    P = P \ {v} 5+ ; X = X ∪ {v} 5+
```

Use Case # 3: k-Clique Listing

- A **k-Clique** of a graph G is a fully connected subgraph of G of k vertices.

```
/*Input: A graph G ❶ , k ∈ ℕ Output: Count of k-cliques ck ∈ ℕ. */

//Preprocessing: reorder vertices with DGR or ADG.
//Here, we also record the actual ordering and denote it as η
  (v1, v2, ..., vn; η) = preprocess(V, /* selected vertex order */) ❸

//Construct a directed version of G using η. This is an
//additional optimization to reduce the search space:
G = dir(G) ❸ //An edge goes from v to u iff η(v) < η(u)
ck = 0 //We start with zero counted cliques.
for u ∈ V in parallel do: ❷ //Count u's neighboring k-cliques
  C2 = N+(u); ck += count(2, G, C2)

function count(i, G, Ci):
  if (i == k): return |Ck| ❺+ //Count k-cliques
  else:
    ci = 0
   for v ∈ Ci ❺+ do: //search within neighborhood of v
      Ci+1 = N+(v) ∩ Ci ❺+ // Ci counts i-cliques.
      ci += count(i+1, G, Ci+1)
    return ci
```

TAKEAWAY: GMS offers lots of modularity in implementing graph mining algorithms, specifically set algebra based modularity!

Theoretical Analysis

	k-Clique Listing *Node Parallel* [41]	k-Clique Listing *Edge Parallel* [41]	★ **k-Clique Listing with ADG (§ 6)**	ADG (Section 6)	Max. Cliques Eppstein et al. [51]	Max. Cliques Das et al. [42]	★ **Max. Cliques with ADG (§ 7.3)**	Subgr. Isomorphism *Node Parallel* [26, 40]	Link Prediction†, JP Clustering
Work	$O\left(mk\left(\frac{d}{2}\right)^{k-2}\right)$	$O\left(mk\left(\frac{d}{2}\right)^{k-2}\right)$	$O\left(mk\left(d+\frac{\varepsilon}{2}\right)^{k-2}\right)$	$O(m)$	$O\left(dm3^{d/3}\right)$	$O\left(3^{n/3}\right)$	$O\left(dm3^{(2+\varepsilon)d/3}\right)$	$O\left(n\Delta^{k-1}\right)$	$O(m\Delta)$
Depth	$O\left(n+k\left(\frac{d}{2}\right)^{k-1}\right)$	$O\left(n+k\left(\frac{d}{2}\right)^{k-2}+d^2\right)$	$O\left(k\left(d+\frac{\varepsilon}{2}\right)^{k-2}+\log^2 n+d^2\right)$	$O\left(\log^2 n\right)$	$O\left(dm3^{d/3}\right)$	$O(d\log n)$	$O\left(\log^2 n+d\log n\right)$	$O\left(\Delta^{k-1}\right)$	$O(\Delta)$
Space	$O(nd^2+K)$	$O\left(md^2+K\right)$	$O\left(md^2+K\right)$	$O(m)$	$O(m+nd+K)$	$O(m+pd\Delta+K)$	$O(m+pd\Delta+K)$	$O(m+nk+K)$	$O(m\Delta)$

Table 4: Work, depth, and space for some graph mining algorithms in GMS. d is the graph degeneracy, K is the output size, Δ is the maximum degree, p is the number of processors, k is the number of vertices in the graph that we are mining for, n is the number of vertices in the graph that we are mining, and m is the number of edges in that graph. † Link prediction and the JP clustering complexities are valid for the Jaccard, Overlap, Adamic Adar, Resource Allocation, and Common Neighbors vertex similarity measures. ★Algorithms derived in this work.

- Obtained better bounds for maximal cliques
- Obtained similar bounds for k-clique, but scales better depending on graph

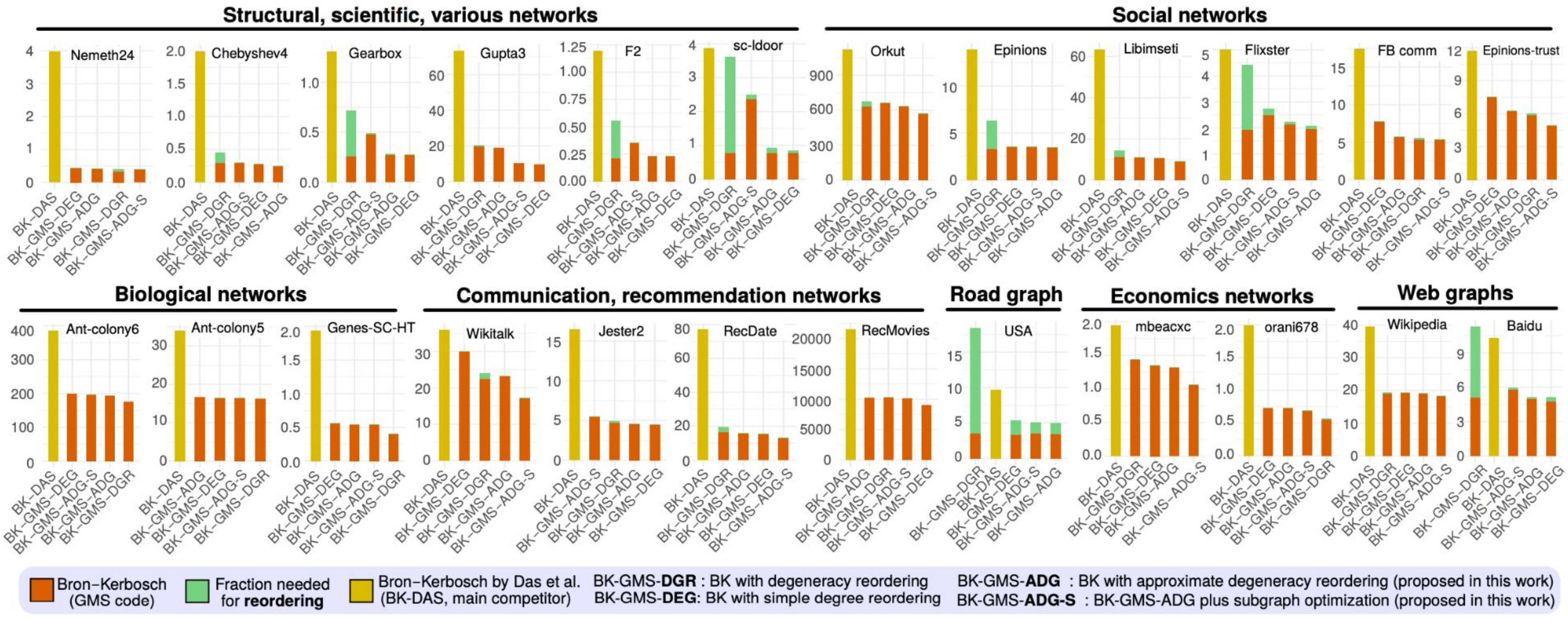
Evaluation: Datasets, Methodology, Architectures

- GMS uses a wide selection of public datasets for flexibility
- GMS compares GMS variants to the most optimized state-of-the-art algorithms.
- GMS runs algorithms on the maximum number of cores available on machine.

Graph †	n	m	$\frac{m}{n}$	$\widehat{d_i}$	$\widehat{d_o}$	T	$\frac{T}{n}$	**Why selected/special?**
[so] (K) Orkut	3M	117M	38.1	33.3k	33.3k	628M	204.3	Common, relatively large
[so] (K) Flickr	2.3M	22.8M	9.9	21k	26.3k	838M	363.7	Large T but low m/n.
[so] (K) Libimseti	221k	17.2M	78	33.3k	25k	69M	312.8	Large m/n
[so] (K) Youtube	3.2M	9.3M	2.9	91.7k	91.7k	12.2M	3.8	Very low m/n and T
[so] (K) Flixster	2.5M	7.91M	3.1	1.4k	1.4k	7.89M	3.1	Very low m/n and T
[so] (K) Livemocha	104k	2.19M	21.1	2.98k	2.98k	3.36M	32.3	Similar to Flickr, but a lot fewer 4-cliques (4.36M)
[so] (N) Ep-trust	132k	841k	6	3.6k	3.6k	27.9M	212	Huge T-skew ($\widehat{T}$ = 108k)
[so] (N) FB comm.	35.1k	1.5M	41.5	8.2k	8.2k	36.4M	1k	Large T-skew ($\widehat{T}$ = 159k)
[wb] (K) DBpedia	12.1M	288M	23.7	963k	963k	11.68B	961.8	Rather low m/n but high T
[wb] (K) Wikipedia	18.2M	127M	6.9	632k	632k	328M	18.0	Common, very sparse
[wb] (K) Baidu	2.14M	17M	7.9	97.9k	2.5k	25.2M	11.8	Very sparse
[wb] (N) WikiEdit	94.3k	5.7M	60.4	107k	107k	835M	8.9k	Large T-skew ($\widehat{T}$ = 15.7M)
[st] (N) Chebyshev4	68.1k	5.3M	77.8	68.1k	68.1k	445M	6.5k	Very large T *and* T/n and T-skew ($\widehat{T}$ = 5.8M)
[st] (N) Gearbox	154k	4.5M	29.2	98	98	141M	915	Low $\widehat{d}$ but large T; low T-skew ($\widehat{T}$ = 1.7k)
[st] (N) Nemeth25	10k	751k	75.1	192	192	87M	9k	Huge T *but* low $\widehat{T}$ = 12k
[st] (N) F2	71.5k	2.6M	36.5	344	344	110M	1.5k	Medium T-skew ($\widehat{T}$ = 9.6k)
[sc] (N) Gupta3	16.8k	4.7M	280	14.7k	14.7k	696M	41.5k	Huge T-skew ($\widehat{T}$ = 1.5M)
[sc] (N) Idoor	952k	20.8M	21.5	76	76	567M	595	Very low T-skew ($\widehat{T}$ = 1.1k)
[re] (N) MovieRec	70.2k	10M	142.4	35.3k	35.3k	983M	14k	Huge T and $\widehat{T}$ = 4.9M
[re] (N) RecDate	169k	17.4M	102.5	33.4k	33.4k	286M	1.7k	Enormous T-skew ($\widehat{T}$ = 1.6M)
[bi] (N) sc-ht (gene)	2.1k	63k	30	472	472	4.2M	2k	Large T-skew ($\widehat{T}$ = 27.7k)
[bi] (N) AntColony6	164	10.3k	62.8	157	157	1.1M	6.6k	Very low T-skew ($\widehat{T}$ = 9.7k)
[bi] (N) AntColony5	152	9.1k	59.8	150	150	897k	5.9k	Very low T-skew ($\widehat{T}$ = 8.8k)
[co] (N) Jester2	50.7k	1.7M	33.5	50.8k	50.8k	127M	2.5k	Enormous T-skew ($\widehat{T}$ = 2.3M)
[co] (K) Flickr (photo relations)	106k	2.31M	21.9	5.4k	5.4k	108M	1019	Similar to Livemocha, but many more 4-cliques (9.58B)
[ec] (N) mbeacxc	492	49.5k	100.5	679	679	9M	18.2k	Large T, low $\widehat{T}$ = 77.7k
[ec] (N) orani678	2.5k	89.9k	35.5	1.7k	1.7k	8.7M	3.4k	Large T, low $\widehat{T}$ = 80.8k
[ro] (D) USA roads	23.9M	28.8M	1.2	9	9	1.3M	0.1	Extremely low m/n and T

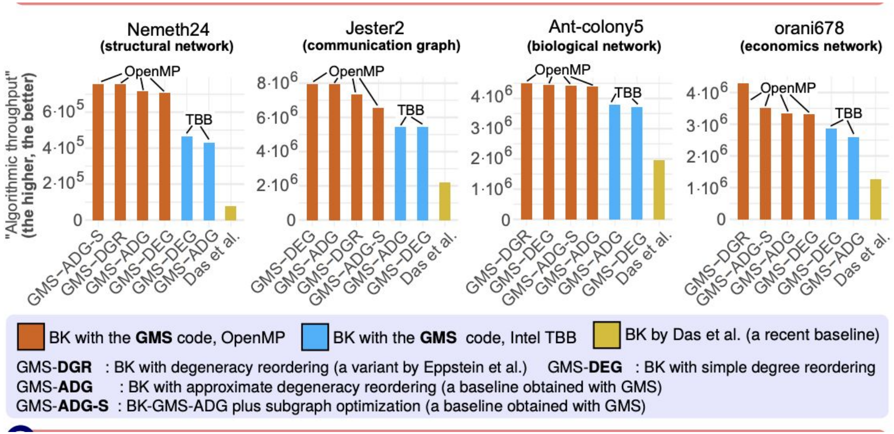
Table 5: Some considered real-world graphs. Graph class/origin: [so]: social network, [wb]: web graph, [st]: structural network, [sc]: scientific computing, [re]: recommendation network, [bi]: biological network, [co]: communication network, [ec]: economics network, [ro]: road graph. Structural features: m/n: graph sparsity, $\widehat{d_i}$: maximum in-degree, $\widehat{d_o}$: maximum out-degree, T: number of triangles, T/n: average triangle count per vertex, T-skew: a skew of triangle counts per vertex (i.e., the difference between the smallest and the largest number of triangles per vertex). Here, $\widehat{T}$ is the maximum number of triangles per vertex in a given graph. Dataset: (W), (S), (K), (D), (C), and (N) refer to the publicly available datasets, explained in § 8.1. For more details, see § 4.2.

Results: Maximal Clique



Results: Maximal Clique cont.

- Achieved a significant improvement in maximal clique using ADG or DGR.
- GMS variants often faster than main competitor by >50%, sometimes even >9x.
  - Consistent over graphs of different structural characteristics
- Another view (**algorithmic efficiency**): number of maximal cliques found per second



More Results

- Up to 10% speedup on k-clique algorithm with different parameters.
- ADG outperforms DGR as a preprocessor for Bron-Kerbosch.

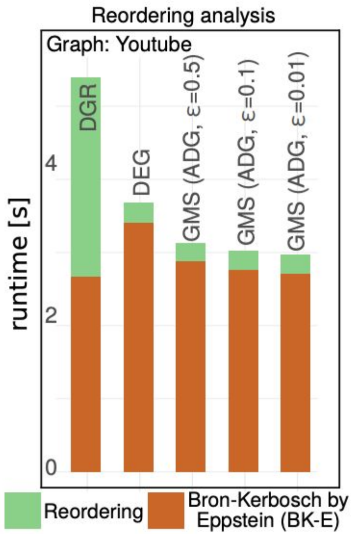
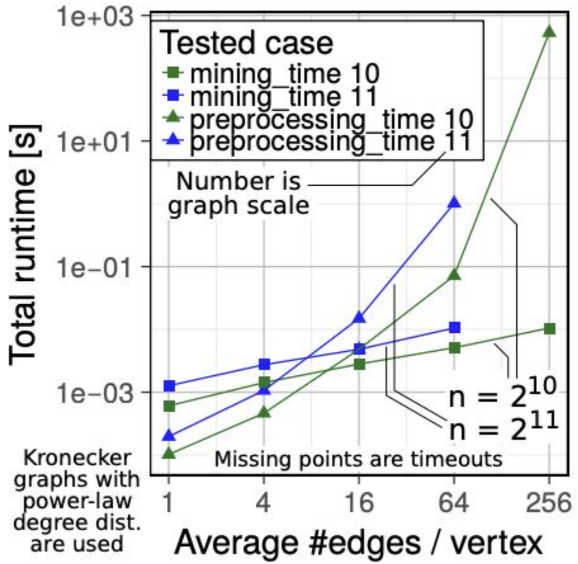


Figure 4: Speedups of ADG for different ε over DEG/DGR, details in § 8.4. System: Ault.

Additional Analyses

- Subtleties of higher-order structures
  - Graphs similar in number of vertices, number of edges, sparsity, degree distribution etc., can have very different higher-order structures, such as number of 4-cliques. Choose datasets wisely!
- Using synthetic graphs can affect whether vertex reordering or mining dominates



(a) Analysis of synthetic graphs.

Thank you! Any questions?