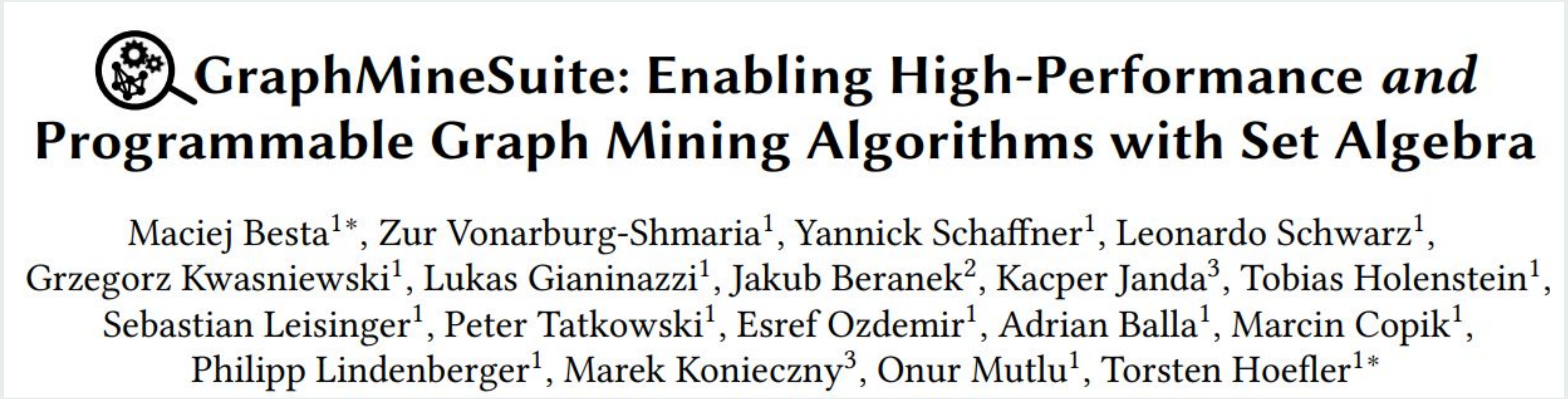


GraphMineSuite: Enabling High-Performance *and* Programmable Graph Mining Algorithms with Set Algebra

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*]

Review by Kliment Serafimov
for 6.827

Summary

The GraphMiningSuite is a **general and extensible framework for end-to-end development** of high-performance graph algorithms.

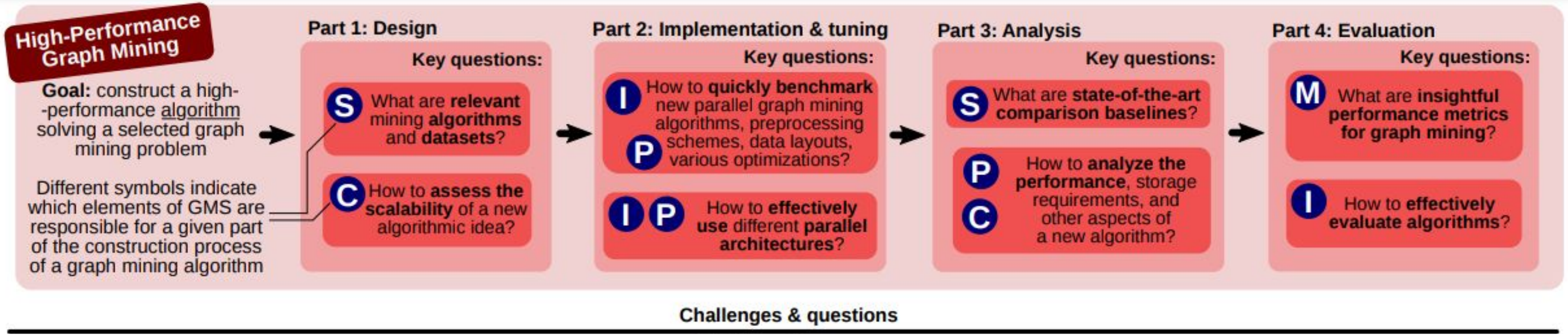
Key features:

- Provides a library of highly-optimized graph processing primitives based on sets and set algebra.
- Extensive set of graph problems and algorithms **> 40 baselines**
  - 3 families of graph problems
  - Variety input graph distributions, both real-world and syntetic.
- **Extensible framework allows users to implement their own modules**
  - For graph representation anb accessing
  - For pre-processing
- Novel performance metric: **algorithmic throughput** (‘graphlets per second’).
- Broad **theoretical concurrency analysis**
  - best work bound among poly-logarithmic depth maximal clique listing algorithms

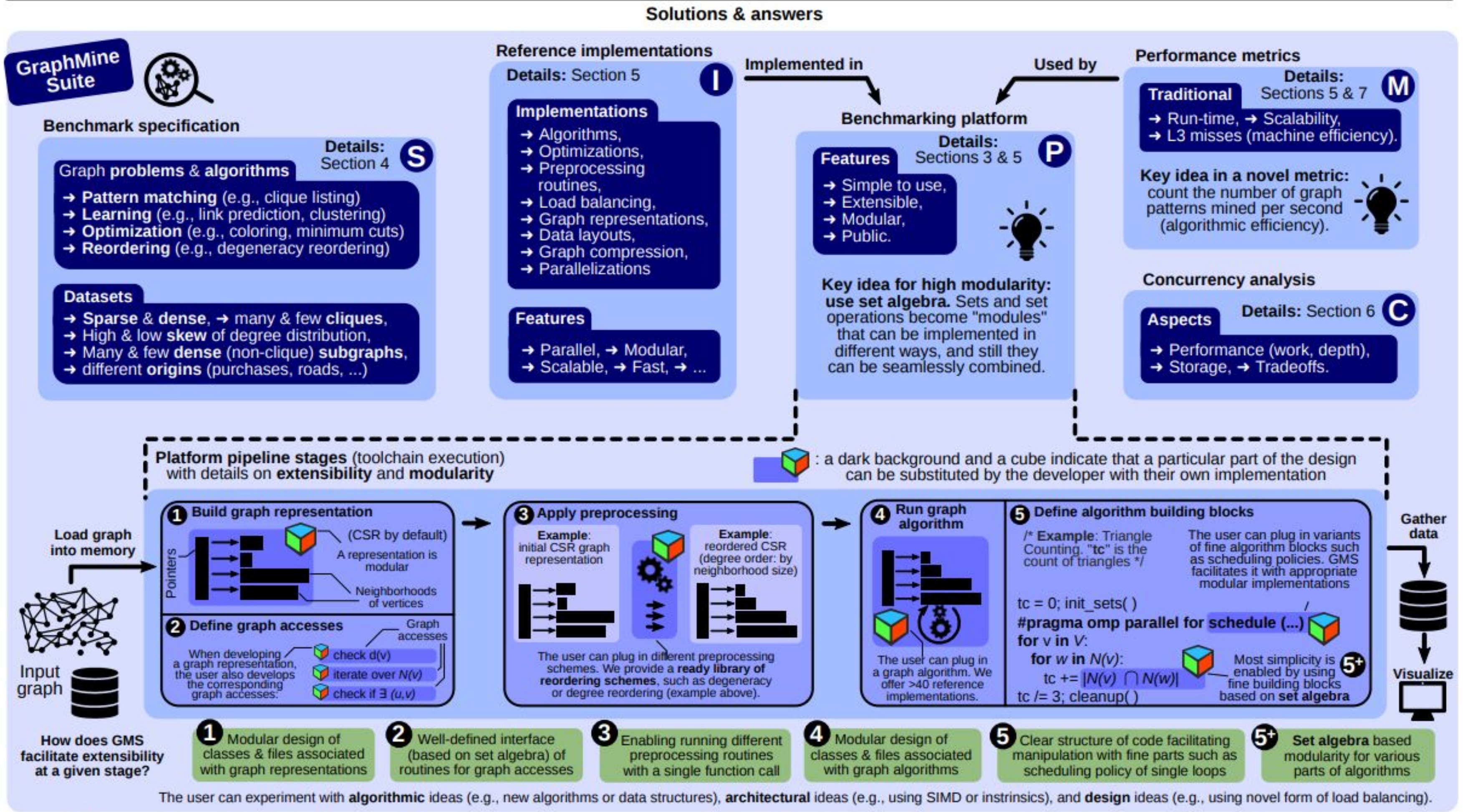
Also:

- Extensive literature review and comparison to other frameworks.

Research Problem Pipeline


High-Performance Graph Mining
Goal: construct a high-performance algorithm solving a selected graph mining problem
Different symbols indicate which elements of GMS are responsible for a given part of the construction process of a graph mining algorithm
Part 1: Design
Key questions:
S What are relevant mining algorithms and datasets?
C How to assess the scalability of a new algorithmic idea?
Part 2: Implementation & tuning
Key questions:
I P How to quickly benchmark new parallel graph mining algorithms, preprocessing schemes, data layouts, various optimizations?
I P How to effectively use different parallel architectures?
Part 3: Analysis
Key questions:
S What are state-of-the-art comparison baselines?
P C How to analyze the performance, storage requirements, and other aspects of a new algorithm?
Part 4: Evaluation
Key questions:
M What are insightful performance metrics for graph mining?
I How to effectively evaluate algorithms?
Challenges & questions

Framework pipeline



Solutions & answers

GraphMine Suite

Benchmark specification — **Details:** Section 4 — S

Graph **problems** & **algorithms**
- ➔ **Pattern matching** (e.g., clique listing)
- ➔ **Learning** (e.g., link prediction, clustering)
- ➔ **Optimization** (e.g., coloring, minimum cuts)
- ➔ **Reordering** (e.g., degeneracy reordering)

Datasets
- ➔ **Sparse** & **dense**, ➔ many & few **cliques**,
- ➔ High & low **skew** of degree distribution,
- ➔ Many & few **dense** (non-clique) **subgraphs**,
- ➔ different **origins** (purchases, roads, ...)

Reference implementations — **Details:** Section 5 — I

Implementations
- ➔ Algorithms,
- ➔ Optimizations,
- ➔ Preprocessing routines,
- ➔ Load balancing,
- ➔ Graph representations,
- ➔ Data layouts,
- ➔ Graph compression,
- ➔ Parallelizations

Features
➔ Parallel, ➔ Modular, ➔ Scalable, ➔ Fast, ➔ ...

Implemented in · **Used by**

Benchmarking platform — **Details:** Sections 3 & 5 — P

Features
- ➔ Simple to use,
- ➔ Extensible,
- ➔ Modular,
- ➔ Public.

Key idea for high modularity: use set algebra. Sets and set operations become "modules" that can be implemented in different ways, and still they can be seamlessly combined.

Performance metrics — **Details:** Sections 5 & 7 — M

Traditional
➔ Run-time, ➔ Scalability, ➔ L3 misses (machine efficiency).

Key idea in a novel metric: count the number of graph patterns mined per second (algorithmic efficiency).

Concurrency analysis — **Details:** Section 6 — C

Aspects
➔ Performance (work, depth), ➔ Storage, ➔ Tradeoffs.

Platform pipeline stages (toolchain execution) with details on **extensibility** and **modularity**

: a dark background and a cube indicate that a particular part of the design can be substituted by the developer with their own implementation

Input graph — **Load graph into memory**

1 Build graph representation
(CSR by default) A representation is modular
Pointers · Neighborhoods of vertices

2 Define graph accesses
When developing a graph representation, the user also develops the corresponding graph accesses:
Graph accesses: check d(v) · iterate over $N(v)$ · check if $\exists\ (u,v)$

3 Apply preprocessing
Example: initial CSR graph representation
Example: reordered CSR (degree order: by neighborhood size)
The user can plug in different preprocessing schemes. We provide a **ready library of reordering schemes**, such as degeneracy or degree reordering (example above).

4 Run graph algorithm
The user can plug in a graph algorithm. We offer >40 reference implementations.

5 Define algorithm building blocks

```
/* Example: Triangle Counting. "tc" is the count of triangles */
tc = 0; init_sets( )
#pragma omp parallel for schedule (...)
for v in V:
  for w in N(v):
    tc += |N(v) ∩ N(w)|
tc /= 3; cleanup( )
```

The user can plug in variants of fine algorithm blocks such as scheduling policies. GMS facilitates it with appropriate modular implementations

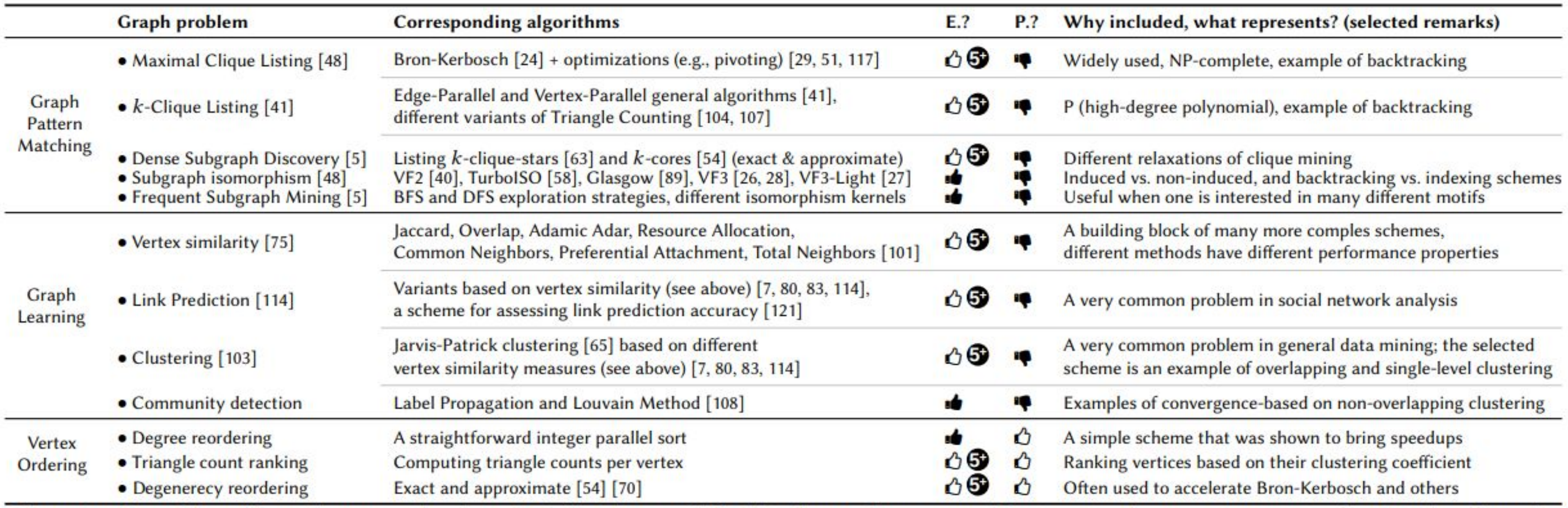
5+ Most simplicity is enabled by using fine building blocks based on **set algebra**

Gather data · **Visualize**

How does GMS facilitate extensibility at a given stage?
- 1 Modular design of classes & files associated with graph representations
- 2 Well-defined interface (based on set algebra) of routines for graph accesses
- 3 Enabling running different preprocessing routines with a single function call
- 4 Modular design of classes & files associated with graph algorithms
- 5 Clear structure of code facilitating manipulation with fine parts such as scheduling policy of single loops
- 5+ **Set algebra** based modularity for various parts of algorithms

The user can experiment with **algorithmic** ideas (e.g., new algorithms or data structures), **architectural** ideas (e.g., using SIMD or instrinsics), and **design** ideas (e.g., using novel form of load balancing).

Graph problems and algorithms implemented

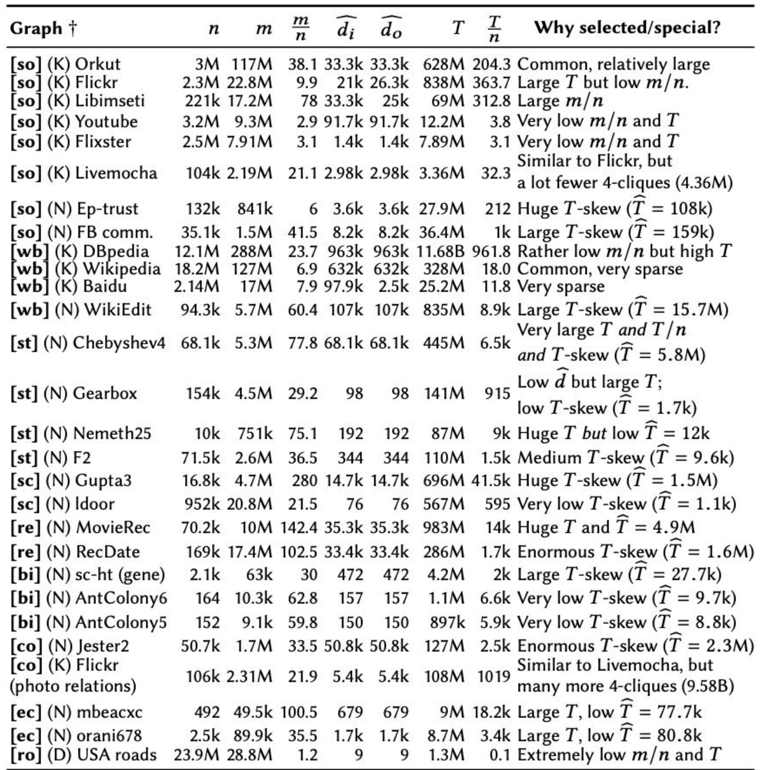


	Graph problem	Corresponding algorithms	E.?	P.?	Why included, what represents? (selected remarks)
Graph Pattern Matching	• Maximal Clique Listing [48]	Bron-Kerbosch [24] + optimizations (e.g., pivoting) [29, 51, 117]	🖒 ❺+	👎	Widely used, NP-complete, example of backtracking
	• k-Clique Listing [41]	Edge-Parallel 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.

Graph Datasets

- Real-world and synthetic graphs with varying statistics:
  - sparsities m/n (sparse and dense)
  - skews in degree distribution (high and low skew)
  - diameters (high and low)
  - amounts of locality i.e. inter-cluster edges (many and few)
  - Triangle-count
  - a large difference between the average number of triangles per vertex T /n and the maximum T/n (for clique algorithms)



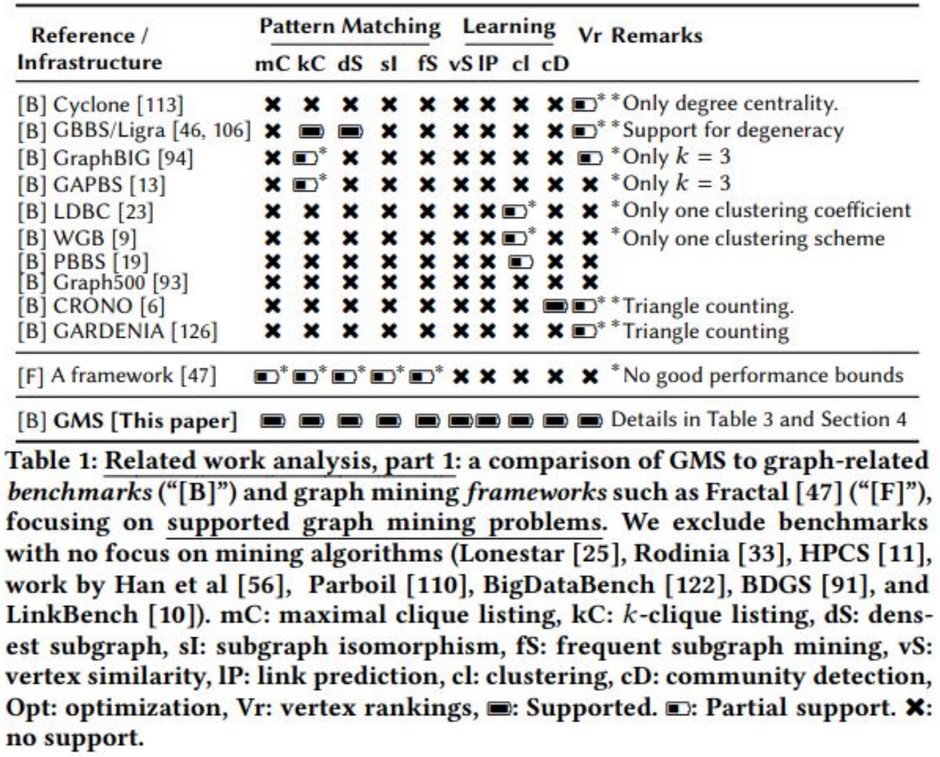
Graph †	n	m	$\frac{m}{n}$	$\widehat{d_i}$	$\widehat{d_o}$	T	$\frac{T}{n}$	**Why selected/special?**
[so] (K) Orkut	3M	117M	38.1	33.3k	33.3k	628M	204.3	Common, relatively large
[so] (K) Flickr	2.3M	22.8M	9.9	21k	26.3k	838M	363.7	Large T but low m/n.
[so] (K) Libimseti	221k	17.2M	78	33.3k	25k	69M	312.8	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) ldoor	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.

Metrics

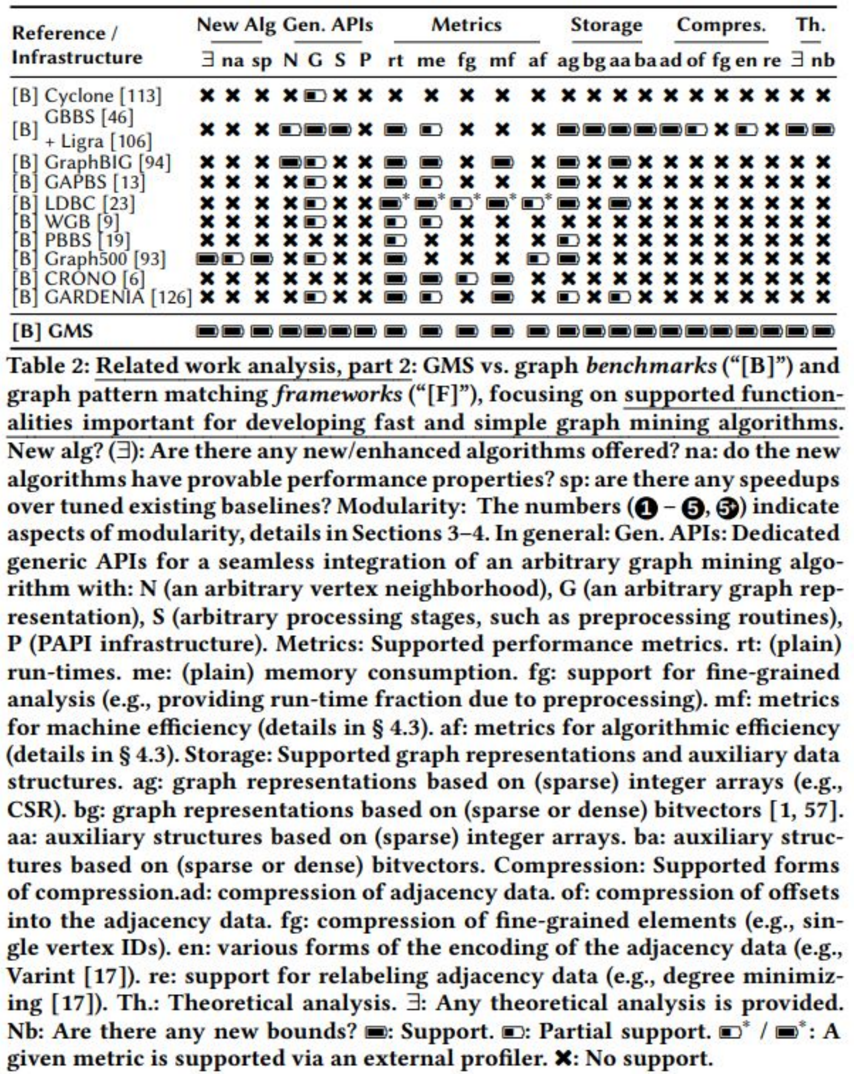
- Seamless integration with PAPI (for extracting hardware use stats)
  - CPU Core utilization (stalled CPU cycles).
  - Cache misses and cache hits (L1, L2, L3, data vs. instruction, TLB)
  - Memory reads/writes
- Algorithmic efficiency / algorithmic throughput
  - Generalization of ‘edges-per-second’.
  - Graphlets-per-second. Number of graph motives mined per second
  - Eg: cliques per second, clusters per second etc.

Literature Review and Comparisson



Reference / Infrastructure	Pattern Matching					Learning				Vr	Remarks
	mC	kC	dS	sI	fS	vS	lP	cl	cD		
[B] Cyclone [113]	✖	✖	✖	✖	✖	✖	✖	✖	✖	◧*	*Only degree centrality.
[B] GBBS/Ligra [46, 106]	✖	■	■	✖	✖	✖	✖	✖	✖	◧*	*Support for degeneracy
[B] GraphBIG [94]	✖	◧*	✖	✖	✖	✖	✖	✖	✖	◧	*Only $k = 3$
[B] GAPBS [13]	✖	◧*	✖	✖	✖	✖	✖	✖	✖	✖	*Only $k = 3$
[B] LDBC [23]	✖	✖	✖	✖	✖	✖	✖	◧*	✖	✖	*Only one clustering coefficient
[B] WGB [9]	✖	✖	✖	✖	✖	✖	✖	◧*	✖	✖	*Only one clustering scheme
[B] PBBS [19]	✖	✖	✖	✖	✖	✖	✖	◧	✖	✖	
[B] Graph500 [93]	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	
[B] CRONO [6]	✖	✖	✖	✖	✖	✖	✖	✖	■	◧*	*Triangle counting.
[B] GARDENIA [126]	✖	✖	✖	✖	✖	✖	✖	✖	✖	◧*	*Triangle counting
[F] A framework [47]	◧*	◧*	◧*	◧*	◧*	✖	✖	✖	✖	✖	*No good performance bounds
[B] **GMS [This paper]**	■	■	■	■	■	■	■	■	■	■	Details in Table 3 and Section 4

Table 1: Related work analysis, part 1: a comparison of GMS to graph-related *benchmarks* ("[B]") and graph mining *frameworks* such as Fractal [47] ("[F]"), focusing on supported graph mining problems. We exclude benchmarks with no focus on mining algorithms (Lonestar [25], Rodinia [33], HPCS [11], work by Han et al [56], Parboil [110], BigDataBench [122], BDGS [91], and LinkBench [10]). mC: maximal clique listing, kC: k-clique listing, dS: densest subgraph, sI: subgraph isomorphism, fS: frequent subgraph mining, vS: vertex similarity, lP: link prediction, cl: clustering, cD: community detection, Opt: optimization, Vr: vertex rankings, ■: Supported. ◧: Partial support. ✖: no support.



Reference / Infrastructure	New Alg			Gen. APIs				Metrics					Storage				Compres.					Th.	
	∃	na	sp	N	G	S	P	rt	me	fg	mf	af	ag	bg	aa	ba	ad	of	fg	en	re	∃	nb
[B] Cyclone [113]	✖	✖	✖	✖	◧	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[B] GBBS [46] + Ligra [106]	✖	✖	✖	◧	■	■	✖	■	◧	✖	✖	✖	■	■	■	■	■	◧	✖	◧	✖	■	■
[B] GraphBIG [94]	✖	✖	✖	■	◧	✖	✖	■	■	✖	■	✖	■	✖	■	✖	✖	✖	✖	✖	✖	✖	✖
[B] GAPBS [13]	✖	✖	✖	✖	◧	✖	✖	■	◧	✖	✖	✖	■	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[B] LDBC [23]	✖	✖	✖	✖	◧	✖	✖	■*	■*	◧*	■*	◧*	■	✖	■	✖	✖	✖	✖	✖	✖	✖	✖
[B] WGB [9]	✖	✖	✖	✖	◧	✖	✖	◧	◧	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[B] PBBS [19]	✖	✖	✖	✖	✖	✖	✖	◧	✖	✖	✖	✖	◧	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[B] Graph500 [93]	■	◧	■	✖	◧	✖	✖	■	✖	✖	✖	◧	■	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[B] CRONO [6]	✖	✖	✖	✖	✖	✖	✖	■	■	◧	■	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
[B] GARDENIA [126]	✖	✖	✖	✖	◧	✖	✖	■	◧	✖	■	✖	◧	✖	◧	✖	✖	✖	✖	✖	✖	✖	✖
[B] **GMS**	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■

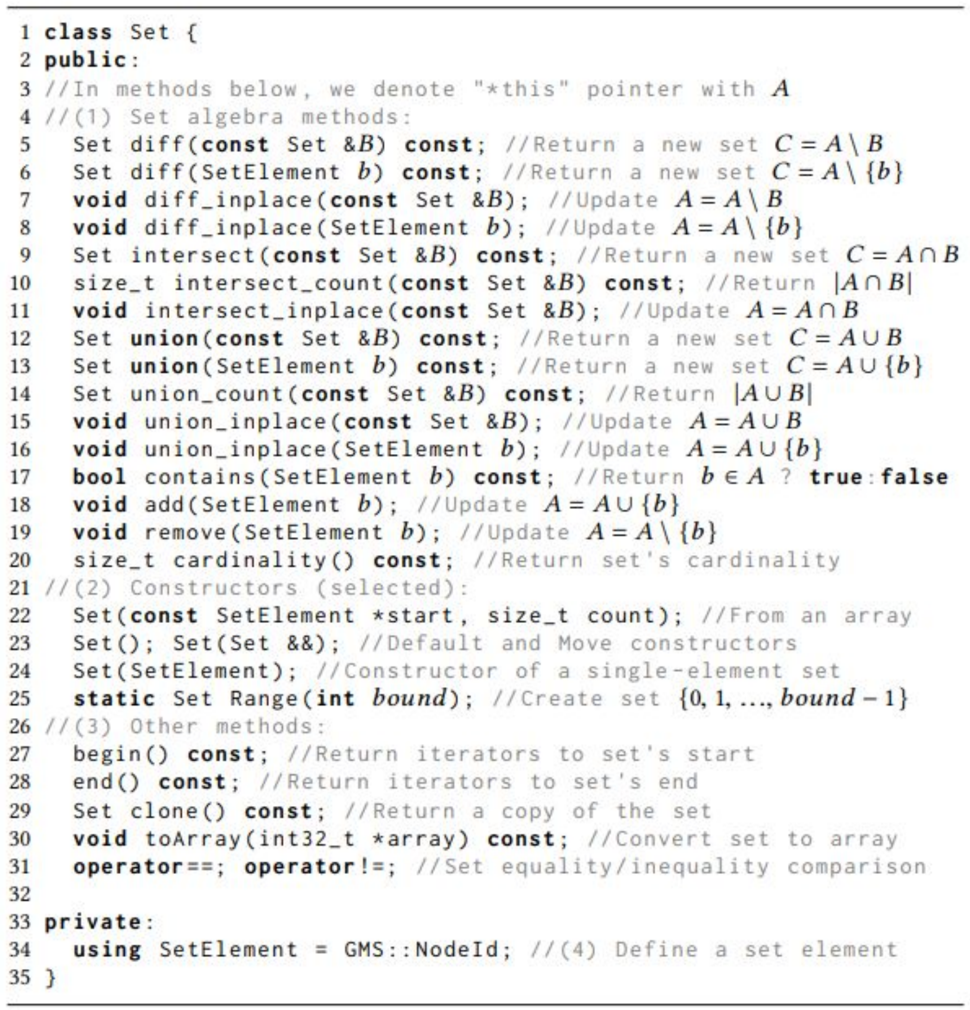
Table 2: Related work analysis, part 2: GMS vs. graph *benchmarks* ("[B]") and graph pattern matching *frameworks* ("[F]"), focusing on supported functionalities important for developing fast and simple graph mining algorithms. New alg? (∃): Are there any new/enhanced algorithms offered? na: do the new algorithms have provable performance properties? sp: are there any speedups over tuned existing baselines? Modularity: The numbers (❶ – ❺, ❺) indicate aspects of modularity, details in Sections 3–4. In general: Gen. APIs: Dedicated generic APIs for a seamless integration of an arbitrary graph mining algorithm with: N (an arbitrary vertex neighborhood), G (an arbitrary graph representation), S (arbitrary processing stages, such as preprocessing routines), P (PAPI infrastructure). Metrics: Supported performance metrics. rt: (plain) run-times. me: (plain) memory consumption. fg: support for fine-grained analysis (e.g., providing run-time fraction due to preprocessing). mf: metrics for machine efficiency (details in § 4.3). af: metrics for algorithmic efficiency (details in § 4.3). Storage: Supported graph representations and auxiliary data structures. ag: graph representations based on (sparse) integer arrays (e.g., CSR). bg: graph representations based on (sparse or dense) bitvectors [1, 57]. aa: auxiliary structures based on (sparse) integer arrays. ba: auxiliary structures based on (sparse or dense) bitvectors. Compression: Supported forms of compression.ad: compression of adjacency data. of: compression of offsets into the adjacency data. fg: compression of fine-grained elements (e.g., single vertex IDs). en: various forms of the encoding of the adjacency data (e.g., Varint [17]). re: support for relabeling adjacency data (e.g., degree minimizing [17]). Th.: Theoretical analysis. ∃: Any theoretical analysis is provided. Nb: Are there any new bounds? ■: Support. ◧: Partial support. ◧* / ■*: A given metric is supported via an external profiler. ✖: No support.

Primitives and Interfaces

- Sets and set-algebra primitives (right ->)
  - Union, intersection, difference
  - Contains, cardinality, iteration, equality
  - remove, add
  - Cloning, serialization
- Graph representation and access
  - Allows the user to implement their own
- Graph preprocessing
  - Allows the user to implement their own
- Algorithms
  - User can use existing algorithms
  - Implement their own algorithms
  - Or tweak existing algorithms

Implementation of set algebra

- Different representations for dense vs sparse sets:
  - 'Roaring' bitmaps set representation
  - SortedSet
  - HashSet

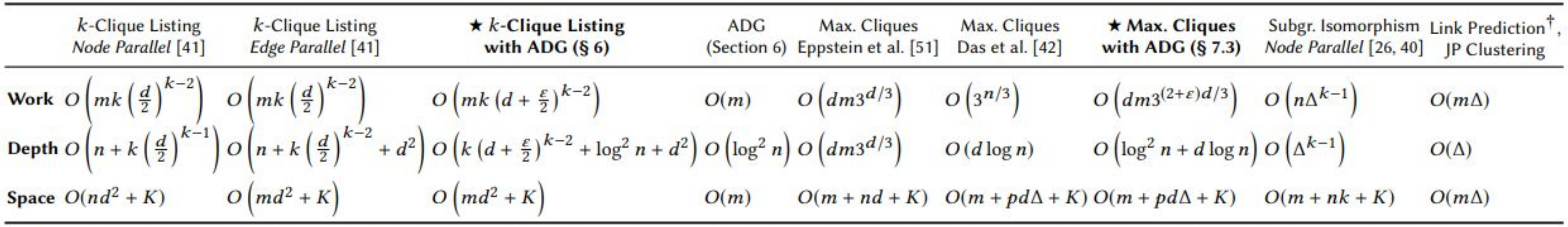


```
class Set {
public:
//In methods below, we denote "*this" pointer with A
//(1) Set algebra methods:
  Set diff(const Set &B) const; //Return a new set C = A \ B
  Set diff(SetElement b) const; //Return a new set C = A \ {b}
  void diff_inplace(const Set &B); //Update A = A \ B
  void diff_inplace(SetElement 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.

Work-span 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.

Use-cases

- Approximate degeneracy order
- Max-clique
- K-clique

Approximate Degeneracy Order

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

Algorithm 3: Deriving the approximate degeneracy order (ADG) in GMS. More than one number indicates that a given snippet is associated with more than one modularity type.

Enumerating Cliques

```
/*Input: A graph $G$ ❶ , $k \in \mathbb{N}$ Output: Count of $k$-cliques $ck \in \mathbb{N}$. */

//Preprocessing: reorder vertices with DGR or ADG.
//Here, we also record the actual ordering and denote it as $\eta$
  $(v_1, v_2, \ldots, v_n; \eta)$ = preprocess($V$, /* selected vertex order */) ❸

//Construct a directed version of $G$ using $\eta$. This is an
//additional optimization to reduce the search space:
$G$ = dir($G$) ❸ //An edge goes from $v$ to $u$ iff $\eta(v) < \eta(u)$
$ck$ = 0 //We start with zero counted cliques.
for $u \in V$ in parallel do: ❷ //Count u's neighboring $k$-cliques
  $C_2$ = $N^+(u)$; $ck$ += count(2, $G$, $C_2$)

function count($i$, $G$, $C_i$):
  if ($i$ == $k$): return $|C_k|$ ❺+ //Count $k$-cliques
  else:
    ci = 0
    for $v \in C_i$ ❺+ do: //search within neighborhood of v
      $C_{i+1}$ = $N^+(v) \cap C_i$ ❺+ // $C_i$ counts $i$-cliques.
      $ci$ += count(i+1, $G$, $C_{i+1}$)
    return ci
```

Algorithm 5: k-Clique Counting; see Listing 3 for the explanation of symbols.

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

//Preprocessing: reorder vertices with DGR or ADG.
  $(v_1, v_2, \ldots, v_n)$ = preprocess($V$, /* selected vertex order */) ❸

//Main part: conduct the actual clique enumeration.
for $v_i \in (v_1, v_2, \ldots, v_n)$ do: //Iterate over $V$ in a specified order
  //For each vertex $v_i$, find maximal cliques containing $v_i$.
  //First, remove unnecessary vertices from $P$ (candidates
  //to be included in a clique) and $X$ (vertices definitely
  //not being in a clique) by intersecting $N(v_i)$ with vertices
  //that follow and precede $v_i$ in the applied order.
  $P$ = $N(v_i) \cap \{v_{i+1}, \ldots, v_n\}$ ❺+; $X$ = $N(v_i) \cap \{v_1, \ldots, v_{i-1}\}$ ❺+; $R$ = $\{v_i\}$

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

BK-Pivot($P, R, X$) //Definition of the recursive BK scheme
  if $P \cup X == 0$ ❺+: Output $R$ as a maximal clique
  $u$ = pivot($P \cup X$) ❺+ //Choose a "pivot" vertex $u \in P \cup X$
  for $v \in P \setminus N(u)$ ❺+: // Use the pivot to prune search space
    //New candidates for the recursive search
    $P_{new}$ = $P \cap N(v)$ ❺+; $X_{new}$ = $X \cap N(v)$ ❺+; $R_{new}$ = $R \cup \{v\}$ ❺+
    //Search recursively for a maximal clique that contains $v$
    BK-Pivot($P_{new}, R_{new}, X_{new}$)
    //After the recursive call, update $P$ and $X$ to reflect
    //the fact that $v$ was already considered
    $P$ = $P \setminus \{v\}$ ❺+; $X$ = $X \cup \{v\}$ ❺+
```

Algorithm 4: Enumeration of maximal cliques, a Bron-Kerbosch variant by Eppstein et al. [52] with GMS enhancements.

Experiments

- Machines:
    - two single machines (1TB and 64GB RAM),
    - Nodes from supercomputers

Thank you!

Any questions?