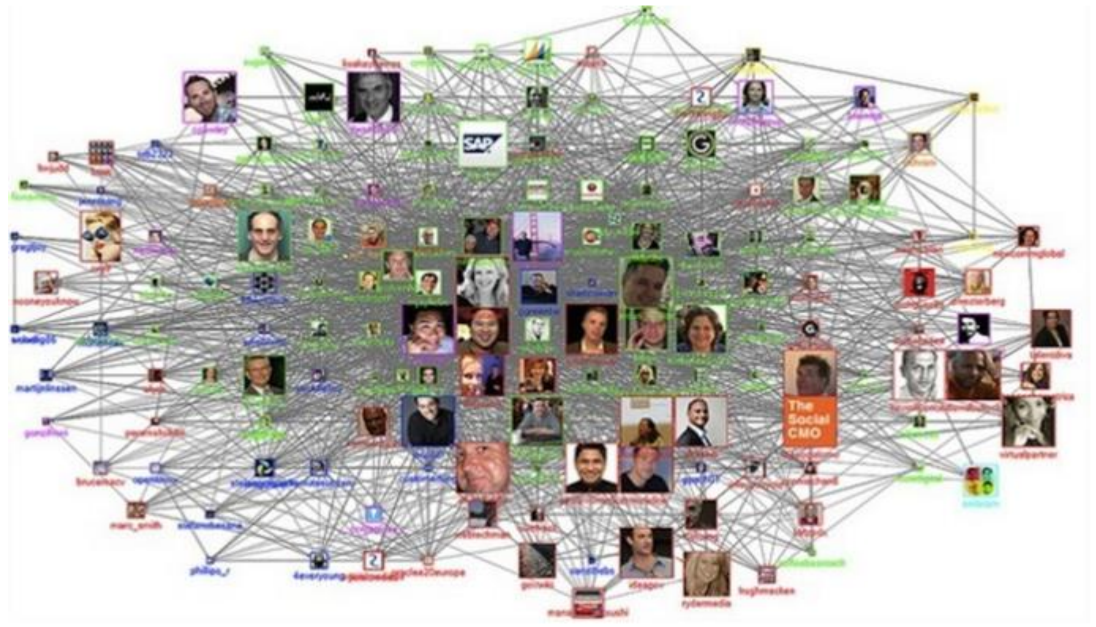
GraphMineSuite:
Enabling High-Performance and Programmable Graph Mining Algorithms with Set Algebra

- Presented by Louise He

Problems Need to be Solved

- Datamining in social network
- Datamining in bioinformation



- Unprecedented scale of modern graphs, reaching up to 12 trillion edges in web-scale datasets

- Algorithmic complexity of core mining problems such as k-clique enumeration (polynomial complexity) and maximal clique listing (NP-hard) are extremely high

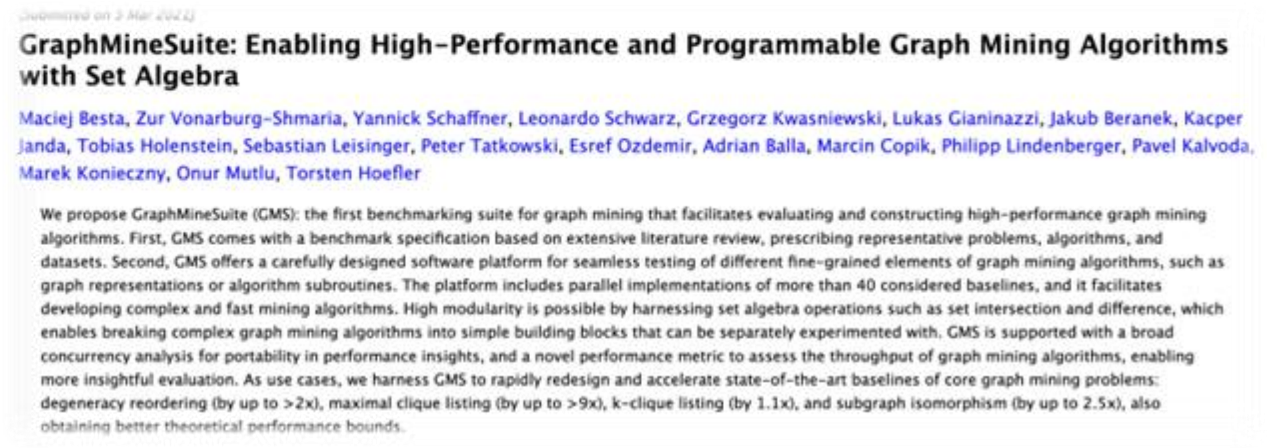
- lack of standardized benchmarking frameworks to systematically evaluate and compare different algorithmic approaches, graph representations, and optimization strategies

Why Important

- Advancing Standardization in Graph Mining Research
- Improving High-Performance Algorithm Design Efficiency
- Deeper Performance Insights Through Novel Metrics
- Enhanced Portability & Scalability Analysis
- Addressing Real-World Complex Graph Data Challenges
- Fostering Ecosystem Building & Community Collaboration

Background

- Published in 2021, PVLDB (Proceedings of the VLDB Endowment)
- Code available on GitHub: https://graphminesuite.spcl.inf.ethz.ch/

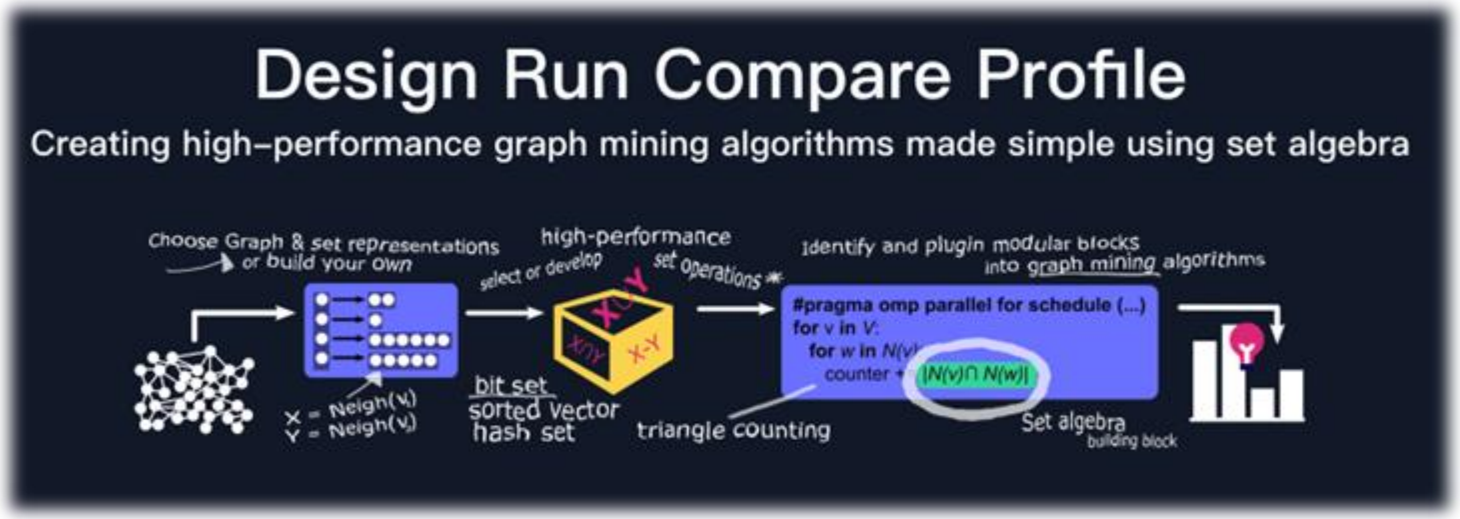


(Submitted on 3 Mar 2021)

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

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

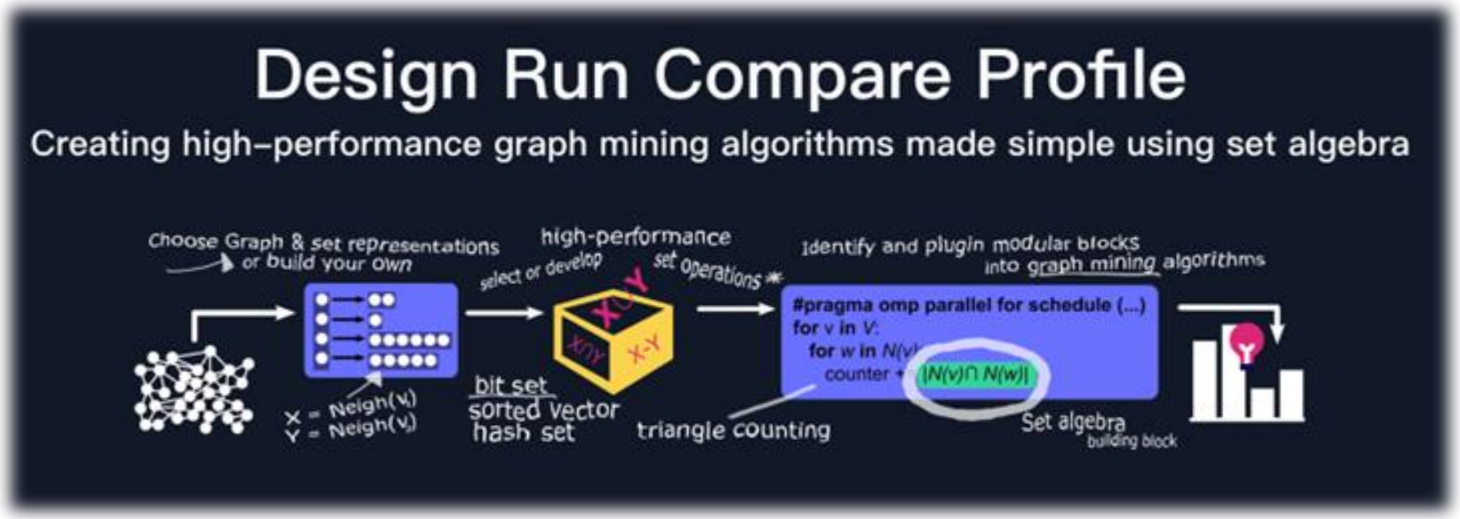
We propose GraphMineSuite (GMS): the first benchmarking suite for graph mining that facilitates evaluating and constructing high-performance graph mining algorithms. First, GMS comes with a benchmark specification based on extensive literature review, prescribing representative problems, algorithms, and datasets. Second, GMS offers a carefully designed software platform for seamless testing of different fine-grained elements of graph mining algorithms, such as graph representations or algorithm subroutines. The platform includes parallel implementations of more than 40 considered baselines, and it facilitates developing complex and fast mining algorithms. High modularity is possible by harnessing set algebra operations such as set intersection and difference, which enables breaking complex graph mining algorithms into simple building blocks that can be separately experimented with. GMS is supported with a broad concurrency analysis for portability in performance insights, and a novel performance metric to assess the throughput of graph mining algorithms, enabling more insightful evaluation. As use cases, we harness GMS to rapidly redesign and accelerate state-of-the-art baselines of core graph mining problems: degeneracy reordering (by up to >2x), maximal clique listing (by up to >9x), k-clique listing (by 1.1x), and subgraph isomorphism (by up to 2.5x), also obtaining better theoretical performance bounds.


Design Run Compare Profile
Creating high-performance graph mining algorithms made simple using set algebra
Choose Graph & set representations or build your own
high-performance set operations
select or develop
Identify and plugin modular blocks into graph mining algorithms
X∪Y
X∩Y
X-Y
#pragma omp parallel for schedule (...)
for v in V:
for w in N(v):
counter +
|N(v)∩N(w)|
X = Neigh(v)
Y = Neigh(v)
bit set
sorted vector
hash set
triangle counting
Set algebra
building block

Idea Proposed - GMS

GraphMineSuite (GMS), the first benchmarking suite dedicated to graph mining

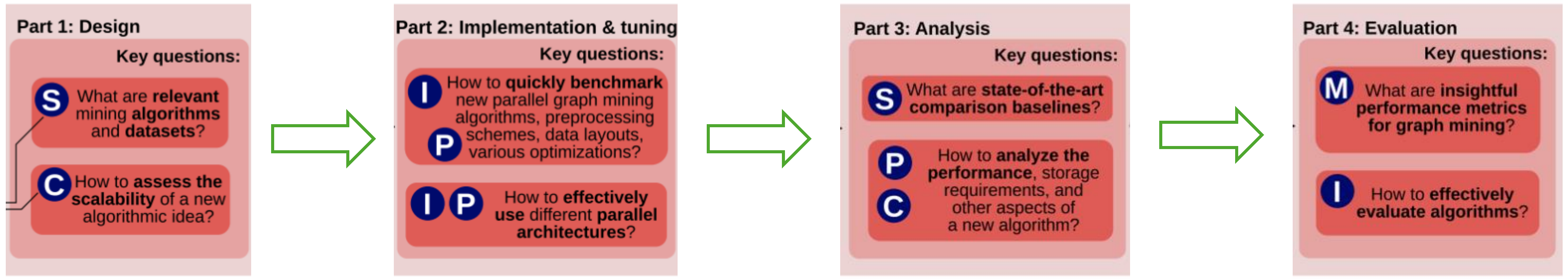
- Uses set algebra as the core computational abstraction.
- Adopts a highly modular architecture ("Lego-like" design).
- Introduces a new performance metric: "algorithmic throughput".
- Provides theoretical concurrency analysis using the work-depth model.
- Includes parallel implementations of over 40 state-of-the-art graph mining algorithms.
- Promotes standardization and fair comparison in graph mining research.
- Builds an integrated ecosystem for graph mining research.


Design Run Compare Profile
Creating high–performance graph mining algorithms made simple using set algebra
Choose Graph & set representations
or build your own
high-performance
select or develop
set operations
X∪Y
X∩Y
X-Y
Identify and plugin modular blocks
into graph mining algorithms
X = Neigh(v)
Y = Neigh(v)
bit set
sorted vector
hash set
triangle counting
|N(v)∩ N(w)|
Set algebra
building block

```
#pragma omp parallel for schedule (...)
for v in V:
  for w in N(v
    counter += |N(v)∩ N(w)|
```

GMS – How to Systematically Build High-Performance Graph Mining Algorithm

Goal: Build a high-performance graph mining algorithm to solve a selected graph mining problem



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**?

- Define the problem type (e.g., clique listing, subgraph isomorphism, etc.)
- select an appropriate dataset, and design an initial version of the algorithm

- Develop high-performance implementations of algorithms
- rapidly test different design choices (such as graph representations and load balancing methods) using platform tools

- Conduct theoretical and empirical analysis of algorithms to understand performance bottlenecks, memory overhead, and parallelism limits

- By comparing the efficiency and robustness of different algorithms through unified evaluation metrics, we ensure that conclusions are reproducible and interpretable

GMS – Core Components and Workflow

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 → **Benchmarking platform** ← **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**

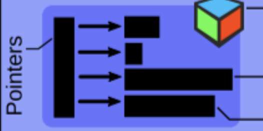
Aspects
- → Performance (work, depth),
- → Storage, → Tradeoffs.

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

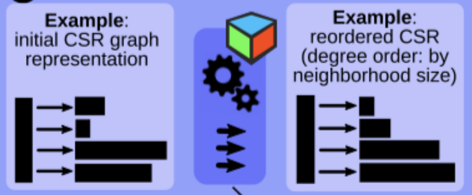


2 Define graph accesses

When developing a graph representation, the user also develops the corresponding graph accesses:
- check d(v)
- iterate over *N*(*v*)
- check if ∃ *(u,v)*

Graph accesses

→ **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 */

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

```
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( )
```

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

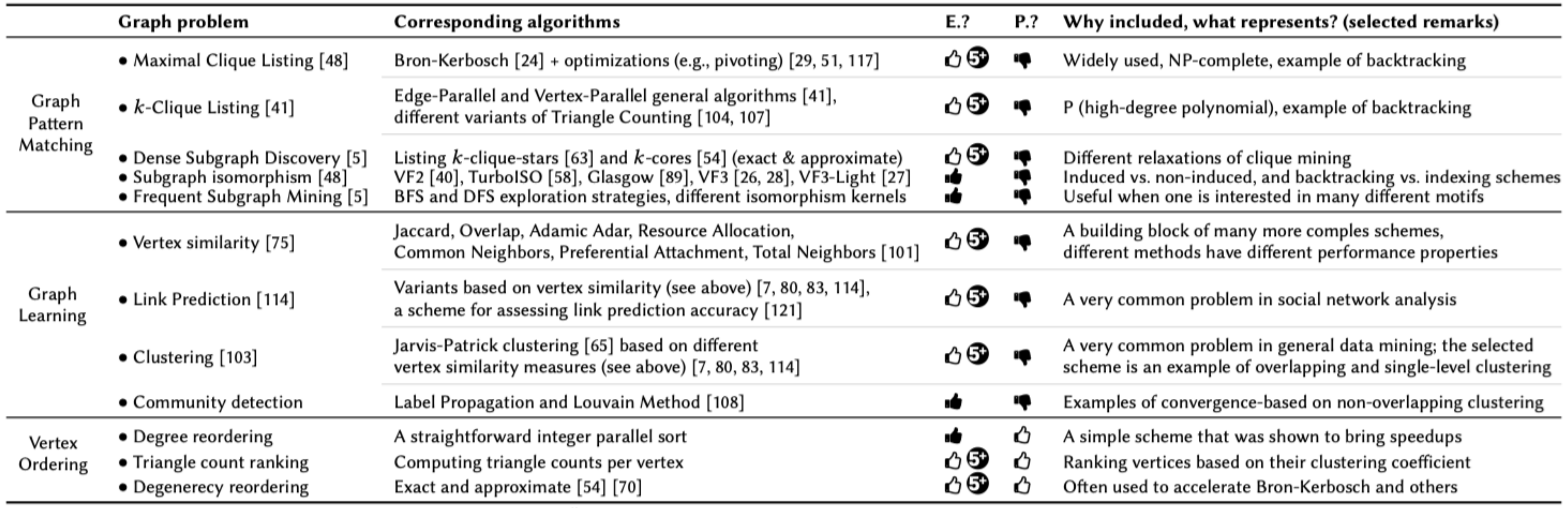
5+ **Set algebra** based modularity for various parts of algorithms

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).

GMS – Benchmark Specification (Graph Problems & Algorithms)

GMS divides graph mining problems into four categories**: pattern matching, learning, reordering, and partial optimization problems**

- **Pattern Matching:** Finding specific subgraphs (motifs), e.g., cliques, dense subgraphs, frequent patterns, and subgraph isomorphism
- **Learning:** Tasks like vertex similarity, link prediction, clustering, and community detection
- **Reordering:** Vertex reordering schemes such as degeneracy or triangle-count-based ordering, used for preprocessing to accelerate other algorithms
- **Partial Optimization Problems:** Classic optimization problems like graph coloring and minimum cuts (partially covered).

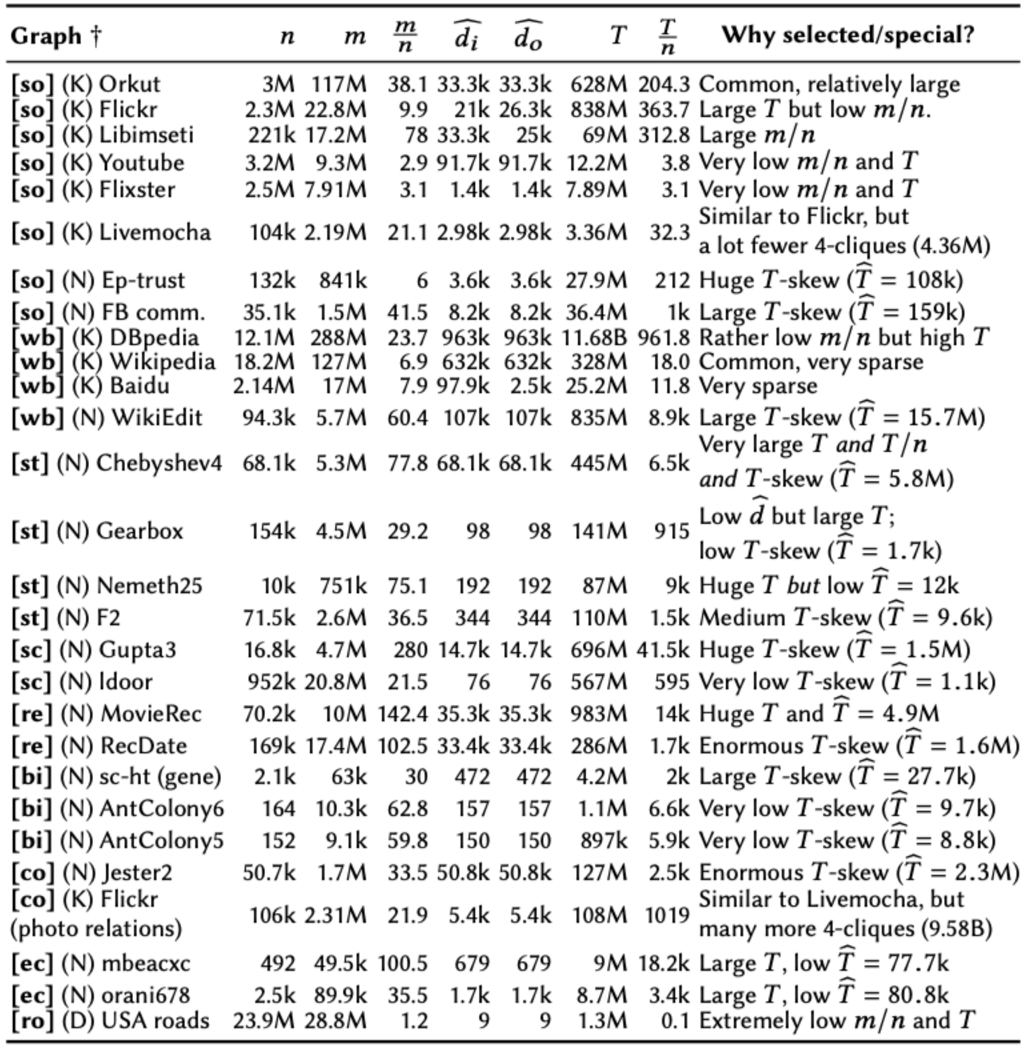


	Graph problem	Corresponding algorithms	E.?	P.?	Why included, what represents? (selected remarks)
Graph Pattern Matching	• Maximal Clique Listing [48]	Bron-Kerbosch [24] + optimizations (e.g., pivoting) [29, 51, 117]	👍 5+	👎	Widely used, NP-complete, example of backtracking
	• k-Clique Listing [41]	Edge-Parallel and Vertex-Parallel general algorithms [41], different variants of Triangle Counting [104, 107]	👍 5+	👎	P (high-degree polynomial), example of backtracking
	• Dense Subgraph Discovery [5]	Listing k-clique-stars [63] and k-cores [54] (exact & approximate)	👍 5+	👎	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]	👍 5+	👎	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]	👍 5+	👎	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]	👍 5+	👎	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	👍 5+	👍	Ranking vertices based on their clustering coefficient
	• Degenerecy reordering	Exact and approximate [54] [70]	👍 5+	👍	Often used to accelerate Bron-Kerbosch and others

GMS – Benchmark Specification (Dataset)

GMS emphasizes the **use of diverse real-world graph datasets** to comprehensively evaluate the performance of graph mining algorithms.

- Include both sparse and dense graphs
- Have highly skewed and more uniform degree distributions
- Contain graphs with many or few cliques
- Contain abundant or scarce non-clique dense subgraphs
- Come from different domains, such as social networks, purchase records, road networks



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

GMS – Benchmark Specification (Metrics)

Traditional metrics:

- Run-time (algorithm execution time)
- Scalability (performance vs thread count)
- Memory usage
- Machine efficiency (e.g., CPU stalls, L3 cache misses via PAPI counters)

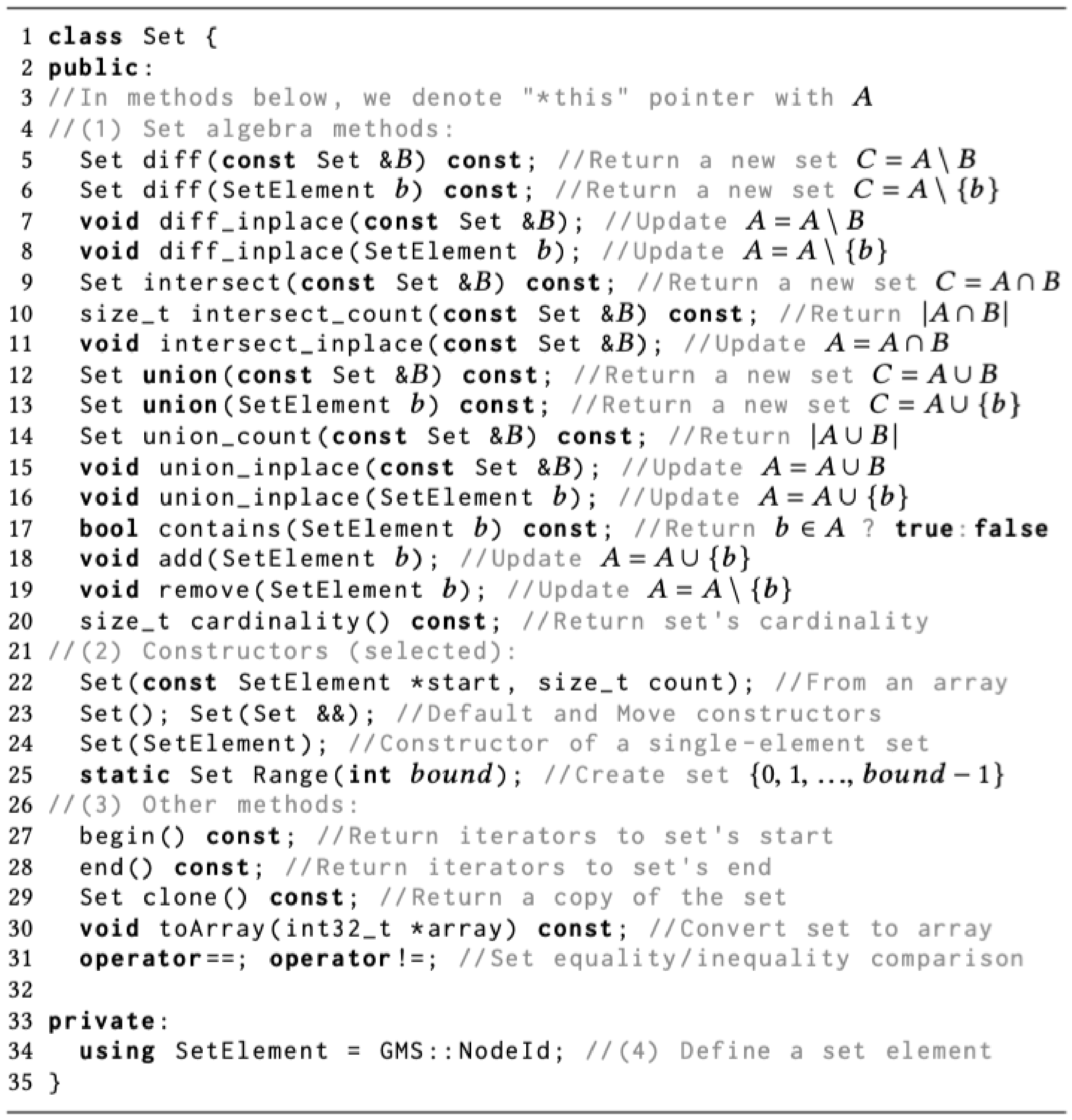
Novel metric introduced by GMS: Algorithmic Efficiency / Algorithmic Throughput

- Measures the number of graph patterns mined per second (e.g., cliques, subgraphs, clusters).
- Extends the idea of Processed Edges Per Second (PEPS) used in graph processing to graph mining.
- Provides insight into algorithmic behavior across graphs of different structures.

GMS – GMS Platform & Set Algebra

Unified Execution Pipeline:
GMS defines a standard five-stage workflow for all graph-mining experiments.

- **Load Graph →** read input datasets from disk into memory.
- **Build Representation →** construct the chosen graph layout (e.g., CSR, bitset, blocked CSR).
- **Preprocess →** apply degree ordering, degeneracy ordering, or other transformations.
- **Run Algorithm →** execute pattern matching, learning, or optimization routines built on Set Algebra.
- **Collect Metrics →** gather runtime, throughput, and memory statistics for evaluation.



```
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.

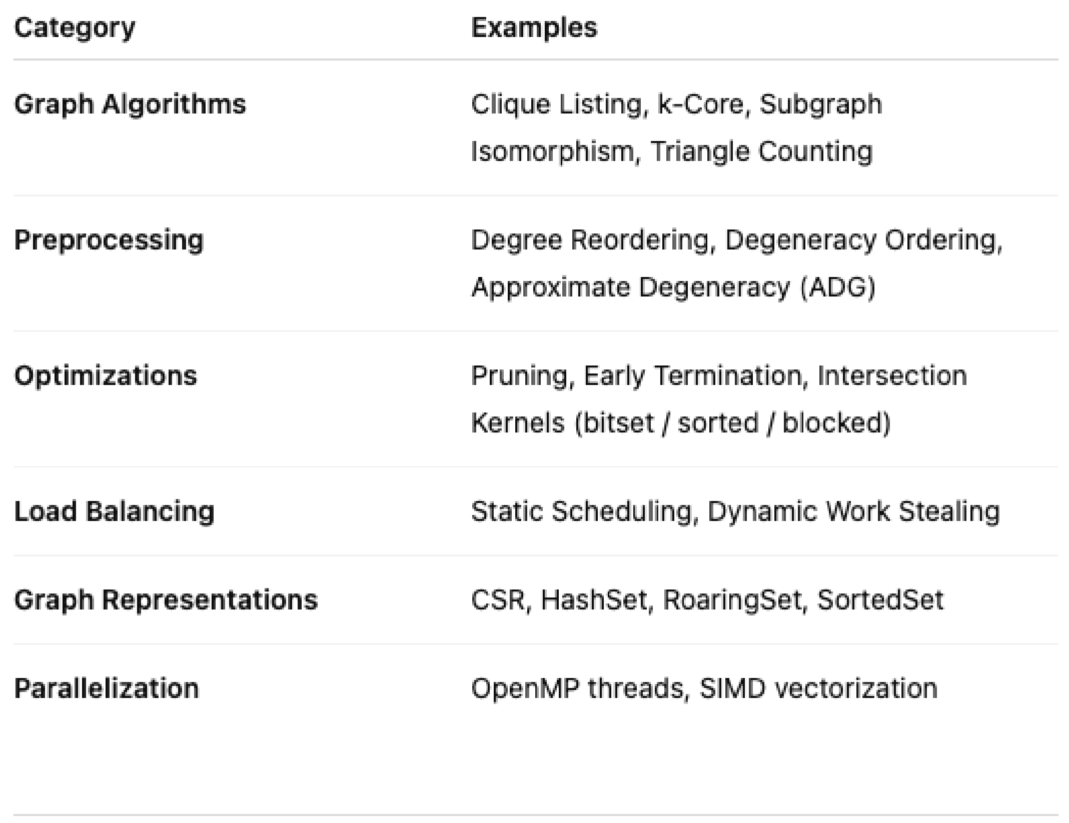
GMS – GMS Platform & Set Algebra

- **Each step is modular —** can replace or change any part (like data layout or preprocessing) without breaking the others.
- **Everything connects through the same "Set Interface."**
  Algorithms, data, and optimizations plug together like LEGO blocks.
- **Parallel by design —** runs on multiple CPU cores using OpenMP or TBB.
- **Hardware-independent —** same code works on different CPUs or servers.
- **Works with other parts of GMS:**
  - Reference algorithms run inside this pipeline.
  - Metrics are collected automatically.
  - Concurrency analysis checks how well the code scales.

- **Modularity:** Algorithms built from reusable set operators.
- **Parallelism:** Set operations map naturally to SIMD and multi-threaded processing.
- **Extensibility:** New storage layouts or hardware backends can be added easily.
- **Reproducibility:** Common API ensures fair and consistent benchmarking

GMS – Reference Implementation

Reference Implementation: Provide standardized, optimized, and reusable algorithmic building blocks that serve as the foundation for all graph-mining tasks within GMS



Category	Examples
Graph Algorithms	Clique Listing, k-Core, Subgraph Isomorphism, Triangle Counting
Preprocessing	Degree Reordering, Degeneracy Ordering, Approximate Degeneracy (ADG)
Optimizations	Pruning, Early Termination, Intersection Kernels (bitset / sorted / blocked)
Load Balancing	Static Scheduling, Dynamic Work Stealing
Graph Representations	CSR, HashSet, RoaringSet, SortedSet
Parallelization	OpenMP threads, SIMD vectorization

- Ensures **fair comparisons** across algorithms and datasets
- Facilitates **reproducibility** and **rapid prototyping**
- Bridges research and engineering by providing optimized, ready-to-use kernels

GMS – From Concept to System Implementation

Use Case 1: Maximal Clique Listing

•**Goal**: Find all maximal cliques (fully connected subgraphs) in a given graph.

•Expressed entirely through the Set Interface (∩, ∪, −).
•Each recursive expansion step becomes a set intersection between vertex neighborhoods (e.g., N(u) ∩ P).
•Uses interchangeable intersection kernels (merge-based, bitset, blocked CSR) for optimal performance.
•Applies Degeneracy Ordering or Approximate Degeneracy (ADG) to reduce recursion depth and prune search space.
•Parallelized using OpenMP tasks with load balancing guided by the Work–Depth mode

Result:
•Achieves **> 9× speedup** compared to previous optimized baselines (Eppstein et al., Das et al.).
•Scales efficiently across dense and sparse graphs.
•Demonstrates that a complex recursive pattern-mining task can be *cleanly* expressed through GMS's Set Algebra.

GMS – From Concept to System Implementation

Use Case 2: Approximate Degeneracy Ordering (ADG) / k-Core Decomposition

•**Goal**: Compute vertex orderings (or k-core levels) that capture structural hierarchy and compress graph complexity

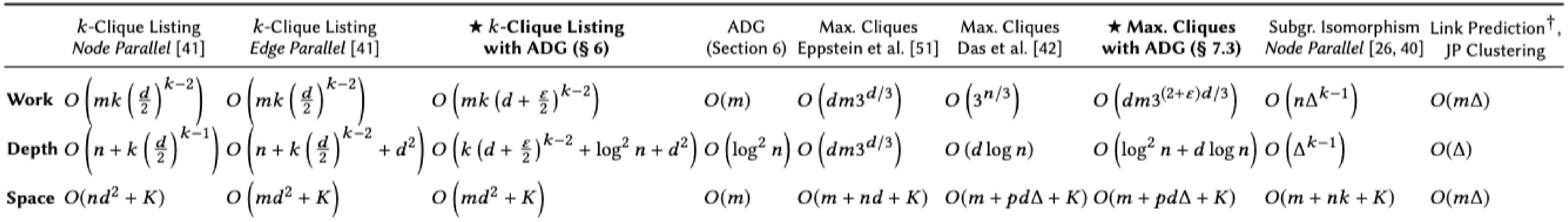
•Described entirely through **Set Interface** operations — vertex peeling becomes repeated *set difference* operations.
•Uses **adaptive degree buckets** and **sampling heuristics** to lower total work (W) while maintaining ordering quality.
•Employs the same modular pipeline as in other GMS algorithms: preprocessing → set ops → output.
•Fully parallelized via OpenMP / TBB, reusing the same scheduling and load-balancing mechanisms.

Result:
•**1.5–2× faster** than exact Degeneracy Reordering (DGR) with negligible accuracy loss.
•Lower ε → higher precision but smaller speedup.
•Demonstrates that GMS supports *both exact and approximate* algorithms efficiently.

GMS – Concurrency Analysis

Work–Depth / Work–Span Model

- Work (W): total operations if run sequentially.
- Depth/Span (D): length of the longest dependency chain (critical path).
- Parallelism: Π = W / D
- Brent bound: on P processors, T (p) ≤ max(W/P, D)



	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	$O(nd^2+K)$	$O\left(md^2+K\right)$	$O\left(md^2+K\right)$	$O(m)$	$O(m+nd+K)$	$O(m+pd\Delta+K)$	$O(m+pd\Delta+K)$	$O(m+nk+K)$	$O(m\Delta)$

- How fast can this algorithm get?
- Is it suitable for multi-core and massively parallel systems?
- What are its time and space requirements?

GMS – Concurrency Analysis

No single optimization is universally best

•Algorithms that use less work may have greater depth → good sequential cost but poor scalability.

•Algorithms with small depth (high parallelism) often require more work or space.

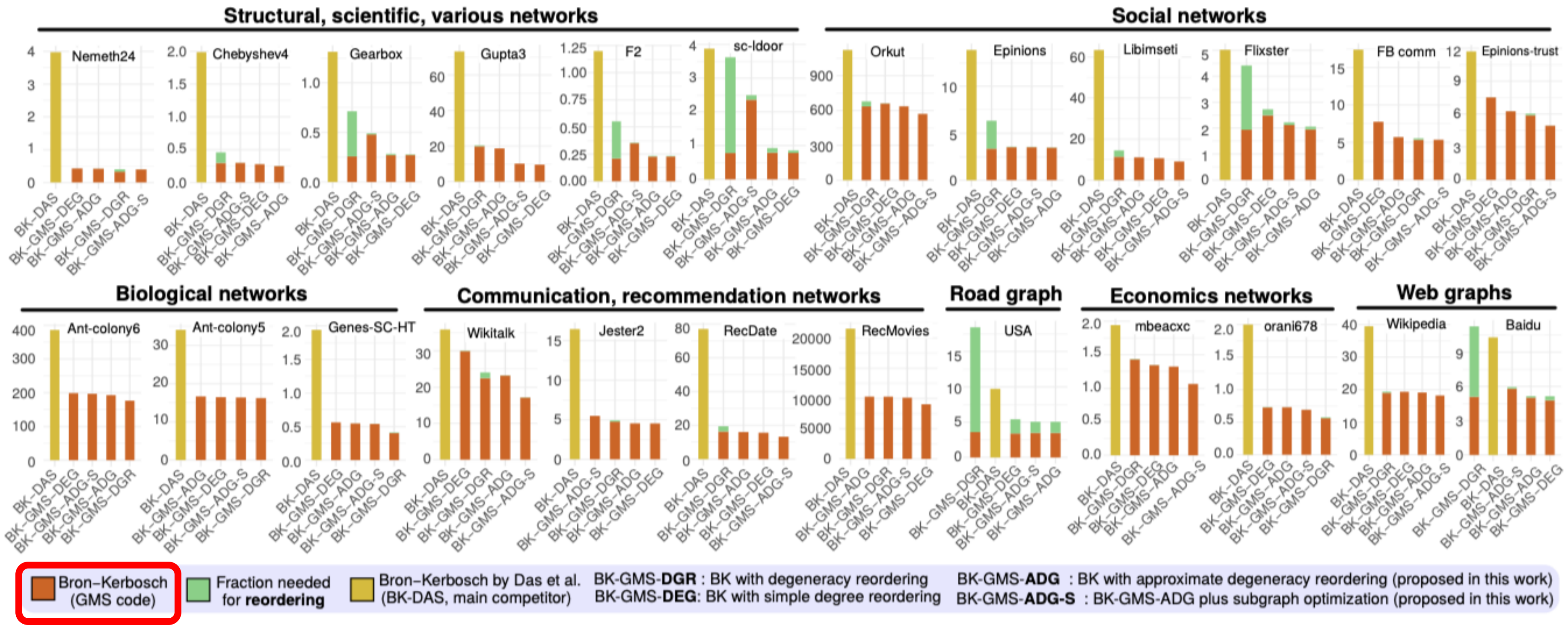
•The “best” algorithm depends on available cores and memory → The more cores, the more suitable for low D algorithms; when memory is limited, prioritize low S algorithms

•Runtime can be approximated by TP≈W/P+DT_P ≈ W/P + DTP≈W/P+D → reducing W and D are both valuable, but which one matters more depends on P.

GMS – Result

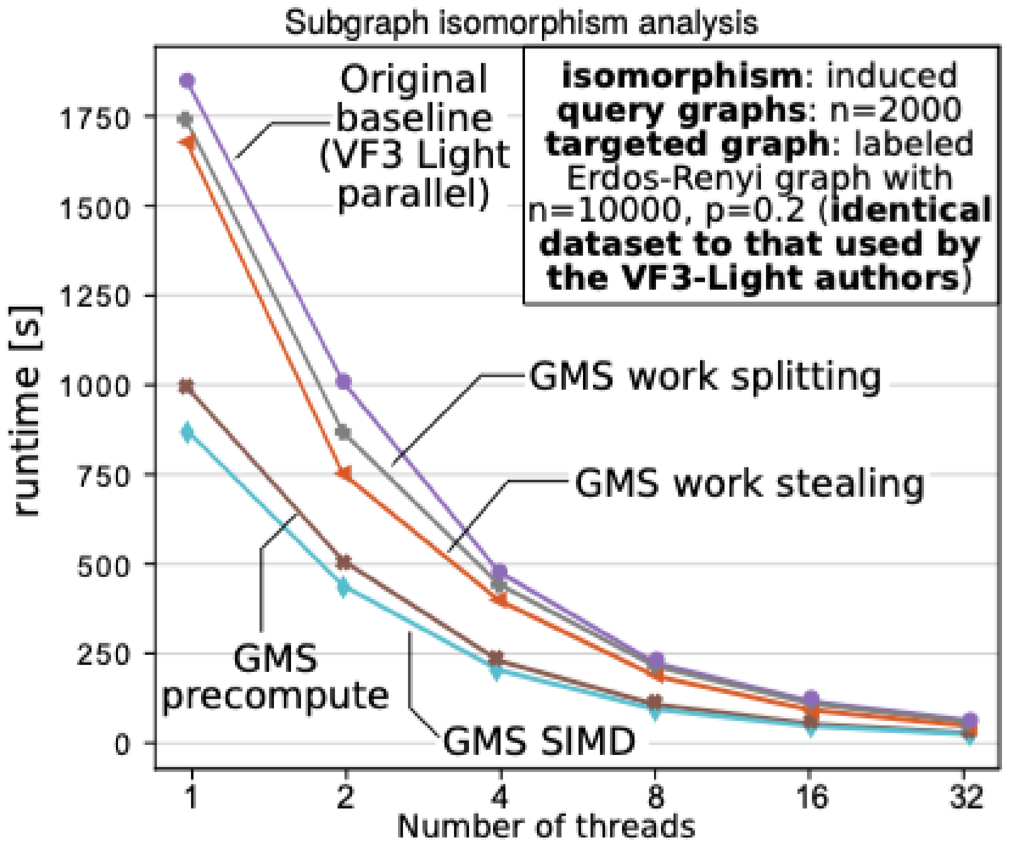
Setup: Runs on multi-core shared-memory machines; evaluates with runtime, memory, hardware counters, and Algorithmic Throughput (patterns per second)

- Maximal Clique Listing: Up to **>9×** more cliques per second than strong baselines, aided by reordering and cached set ops
- Degeneracy Reordering / k-Core: **>2×** speedups for reordering/core decomposition variants
- k-Clique Listing: Up to **~1.1×** improvements with better bounds/layouts.
- Subgraph Isomorphism: Around **2.5×** speedup over a recent baseline.



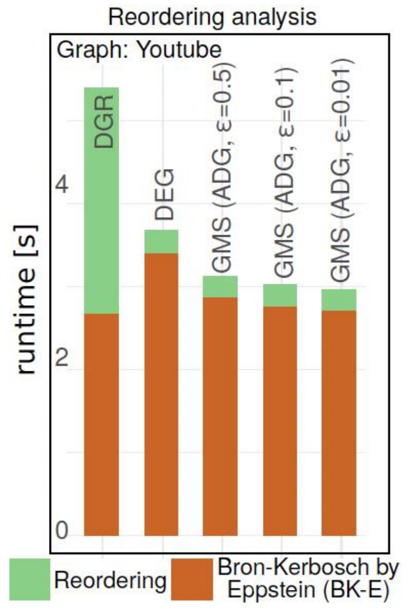
GMS – Result

Scalability & Theory match: Observed speedups track the **Work–Depth** predictions: good scaling with threads; designs chosen to keep work low and depth manageable.

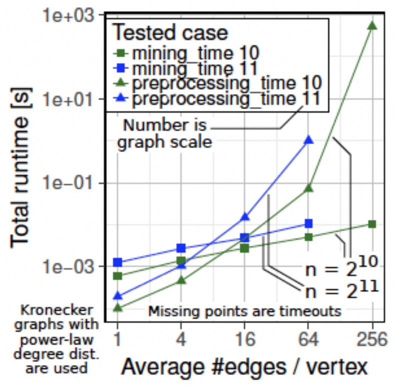


GMS – Result

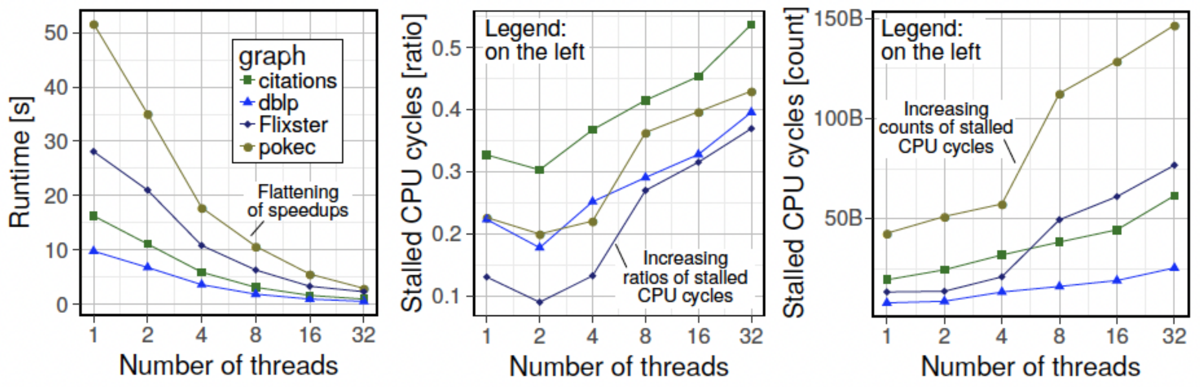
ADG ε : Demonstrate GMS Modular Experiment Capabilities



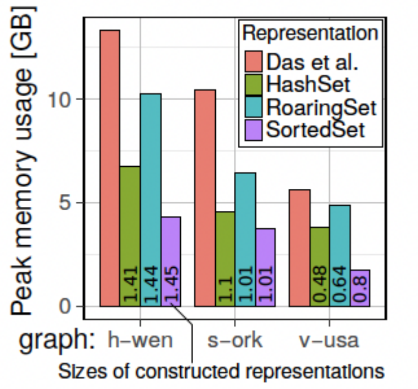
Machine Efficiency



(a) Analysis of synthetic graphs.



(b) Analysis of machine efficiency.



(c) Sizes of GMS graph representations.

Strengths & Weakness

- High Performance and Scalability
- Unified Benchmark Suite
- Modular Design via Set Algebra
- Clear Parallelism Analysis
- Transparent Performance Metrics
- Practical Relevance

- Limited Hardware Scope
- Memory Bound Scalability
- Preprocessing Overhead
- Trade-off Complexity
- Energy and I/O Metrics Missing

Potential Future Work Directions

- Further optimize performance on GPUs and distributed-memory systems, moving beyond the current focus on shared-memory multicore architectures.
- Expand the system to heterogeneous setups like CPU+GPU, enabling efficient processing of extremely large graphs.
- Introduce more comprehensive performance metrics
- Reduce manual configuration burden on users
- Build an automated configuration recommendation system
- Lower usage barriers and improve usability

Discussions

- What are the key challenges GMS would face when extending from shared-memory multicore systems to GPUs or distributed clusters?

- The paper mentions significant speedups after re-implementing existing algorithms in GMS. Are these gains due to platform optimizations or algorithmic improvements? How can we disentangle the contributions of each?

- Modular design makes GMS easy to extend, but could it come at the cost of performance? How is the trade-off between flexibility and efficiency managed?

- GMS emphasizes concurrency analysis using the work-depth model. How effective is this in predicting algorithm performance on real hardware?