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

bestam@inf.ethz.ch, zur.shmaria@gmail.com, {schyanni, schwaleo}@student.ethz.ch, grzegkwas@gmail.com, glukas@inf.ethz.ch, jakub.beranek@vsb.cz, kacperj97@gmail.com, {tobiahol, sleising, peterta, esrefo, aballa}@student.ethz.ch, {mcopik@inf, plindenbe@math}.ethz.ch, marekko@agh.edu.pl, omutlu@inf.ethz.ch, htor@inf.ethz.ch

ABSTRACT

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 >2×), maximal clique listing (by >9×), k-clique listing (by up to 1.1×), and subgraph isomorphism (by 2.5×), also obtaining better theoretical performance bounds.

PVLDB Reference Format:
M. Besta et al.. GraphMineSuite: Enabling High-Performance and Programmable Graph Mining Algorithms with Set Algebra. PVLDB, 14(11): 1922 - 1936, 2021.
doi:10.14778/3476249.3476252

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://graphminesuite.spcl.inf.ethz.ch/.

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476252

1 INTRODUCTION AND MOTIVATION

Graph mining is used in social sciences, bioinformatics, chemistry, medicine, cybersecurity, and many others [31, 39, 60, 66]. Yet, graphs can reach one trillion edges (the Facebook graph (2015) [38]) or even 12 trillion edges (the Sogou webgraph (2018) [81]), requiring unprecedented amounts of compute power to solve even simple graph problems such as BFS [81]. Harder problems, such as mining k-cliques (time complexity is a high-degree polynomial) or maximal cliques (NP-hard in the worst case), face even larger challenges.

At the same time, massive parallelism has become prevalent in modern compute devices [12], bringing a promise of fast parallel graph mining algorithms. Yet, several issues hinder achieving this. First, a large number of graph mining algorithms and their variants make it hard to identify the most relevant baselines as either promising candidates for further improvement, or as appropriate comparison targets. Similarly, a plethora of available networks hinder selecting relevant input datasets for evaluation. Second, even when experimenting with a single specific algorithm, one often faces numerous design choices, for example which graph representation to use, whether to apply graph compression, how to represent

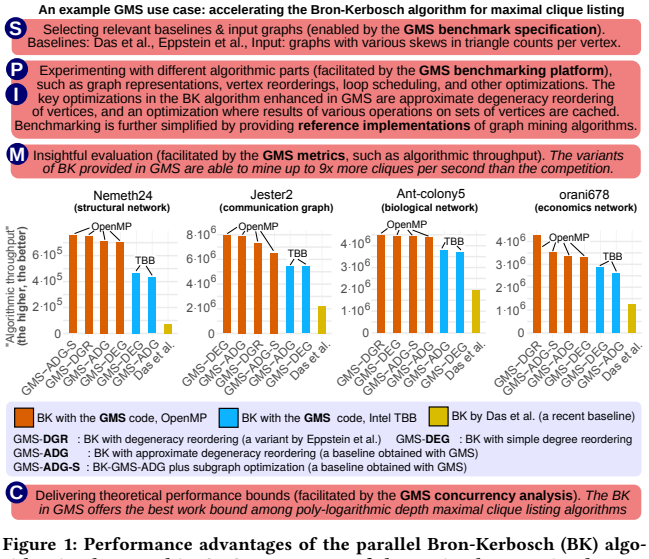


Figure 1: Performance advantages of the parallel Bron-Kerbosch (BK) algorithm implemented in GMS over a state-of-the-art implementation by Das et al. [42] and a recent algorithm by Eppstein et al. [51] (GMS-DGR) using a novel performance metric "algorithmic throughput" that shows a number of maximal cliques found per second. Details of experimental setup: Section 8.

auxiliary data structures, etc.. Such choices may significantly impact performance, often in a non-obvious way, and they may require a large coding effort when trying different options [41].

To address these issues, we introduce **GraphMineSuite (GMS)**, *a benchmarking suite* for *high-performance* graph mining *algorithms*. GMS provides an exhaustive benchmark specification (S). Moreover, GMS offers a novel performance metric (M) and a broad theoretical concurrency analysis (C) for deeper performance insights beyond simple empirical run-times. To maximize GMS' usability, we arm it with an accompanying software platform (P) with reference implementations of algorithms (I). We motivate the GMS platform in Figure 1, which illustrates example performance advantages (even more than 9×) of the GMS code over a state-of-the-art variant of the Bron-Kerbosch (BK) algorithm. This shows the key benefit of the platform: it facilitates developing, redesigning, and enhancing algorithms considered in the benchmark, and thus it enabled us to rapidly obtain large speedups over fast existing BK baselines.

To construct GMS, we first identify representative graph mining *problems*, *algorithms*, and *datasets*. We conduct an extensive literature review [5, 8, 31, 55, 66, 74, 75, 80, 83, 98–100, 112, 123], and obtain a ***benchmark specification*** (S) that can be used as a reference point when selecting relevant comparison targets.

Second, GMS comes with a ***benchmarking platform*** (P): a highly modular infrastructure for easy experimenting with different design choices in a given graph mining algorithm. A key idea for high modularity is exploiting *set algebra*. Here, we observe that data structures and subroutines in many mining algorithms are "set-centric": they can be expressed with sets and set operations, and the user can seamlessly use different implementations of the same specific "set-centric" part. This enables the user to seamlessly use new graph representations, data layouts, architectural features such as vectorization, and even use numerous graph compression schemes. We deliver ready-to-go parallel implementations of the above-mentioned elements, including more than 40 parallel ***reference implementations*** (I) of graph mining algorithms, as well as representations, data layouts, and compression schemes.

For more insightful performance analyses, we propose a novel ***performance metric*** (M) that assesses "algorithmic efficiency", i.e., "how efficiently a given algorithm mines selected graph motifs".

To ensure performance insights that are portable across different machines and independent of various implementation details, GMS also provides ***the first extensive concurrency analysis*** (C) of a wide selection of graph mining algorithms. We use *work-depth*, an established theoretical framework from parallel computing [18, 20], to show which algorithms come with more potential for high performance on today's massively parallel systems.

To show the potential of GMS, we ***enhance state-of-the-art algorithms*** that target some of the most researched graph mining problems. This includes maximal clique listing [42], k-clique listing [41], degeneracy reordering (core decomposition) [86], and subgraph isomorphism [27, 28]. By being able to rapidly experiment with different design choices, we get *speedups of* $>9\times$*, up to* $1.1\times$*,* $>2\times$*, and* $2.5\times$, respectively. We also *improve theoretical bounds*: for example, for maximal clique listing, we obtain $O(dm3^{(2+\varepsilon)d/3})$ work and $O(\log^2 n + d \log n)$ depth (d, m, n are the graph degeneracy, #edges, and #vertices, respectively). This is the best work bound among poly-logarithmic depth maximal clique listing algorithms, improving upon recent schemes [42, 51, 52].

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.

GMS vs. Graph-Related Benchmarks We motivate GMS as ***the first benchmark for graph mining***. There exist graph processing benchmarks, but they do not focus on graph mining; we illustrate this in Table 1 ("[B]"). They focus on graph *database workloads*, extreme-scale graph *traversals*, and different *"low-complexity"* (i.e., with run-times being low-degree polynomials in numbers of vertices or edges) parallel graph algorithms such as PageRank, triangle counting, and others, researched intensely in the parallel programming community. Despite some similarities (e.g., GBBS provides implementations of k-clique listing), none of these benchmarks targets general graph mining, and they do not offer novel performance metrics or detailed control over graph representations, data layouts, and others. We broadly analyze this in Table 2, where we compare GMS to other benchmarks in terms of the modularity of their software infrastructures, offered metrics, control over storage schemes, support for graph compression, provided theoretical analyses, and whether they improve state-of-the-art algorithms. Finally, GMS is the only benchmark that is used to directly enhance core state-of-the-art graph mining algorithms, achieving both better bounds and speedups in empirical evaluation.

GMS vs. Pattern Matching Frameworks Many graph mining *frameworks* have recently been proposed, for example Peregrine [64] and others [34, 35, 47, 62, 68, 87, 88, 115, 127, 128, 131]. GMS does *not* compete with such frameworks. First, as Table 1 shows, such frameworks do not target broad graph mining. Second, key offered functionalities also differ. These frameworks focus on *programming models* and *abstractions*, and on the underlying *runtime systems*. Contrarily, GMS focuses on benchmarking and tuning *specific parallel algorithms*, with provable performance properties, to accelerate the most competitive existing baselines.

2 NOTATION AND BASIC CONCEPTS

We model an undirected graph G as a tuple (V, E); V is a set of vertices and $E \subseteq V \times V$ is a set of edges; $|V| = n$ and $|E| = m$. The maximum degree of a graph is Δ. The neighbors and the degree of a given vertex v are denoted with $N(v)$ and $\Delta(v)$, respectively.

3 OVERVIEW OF GMS

We start with an overview; see Figure 2.

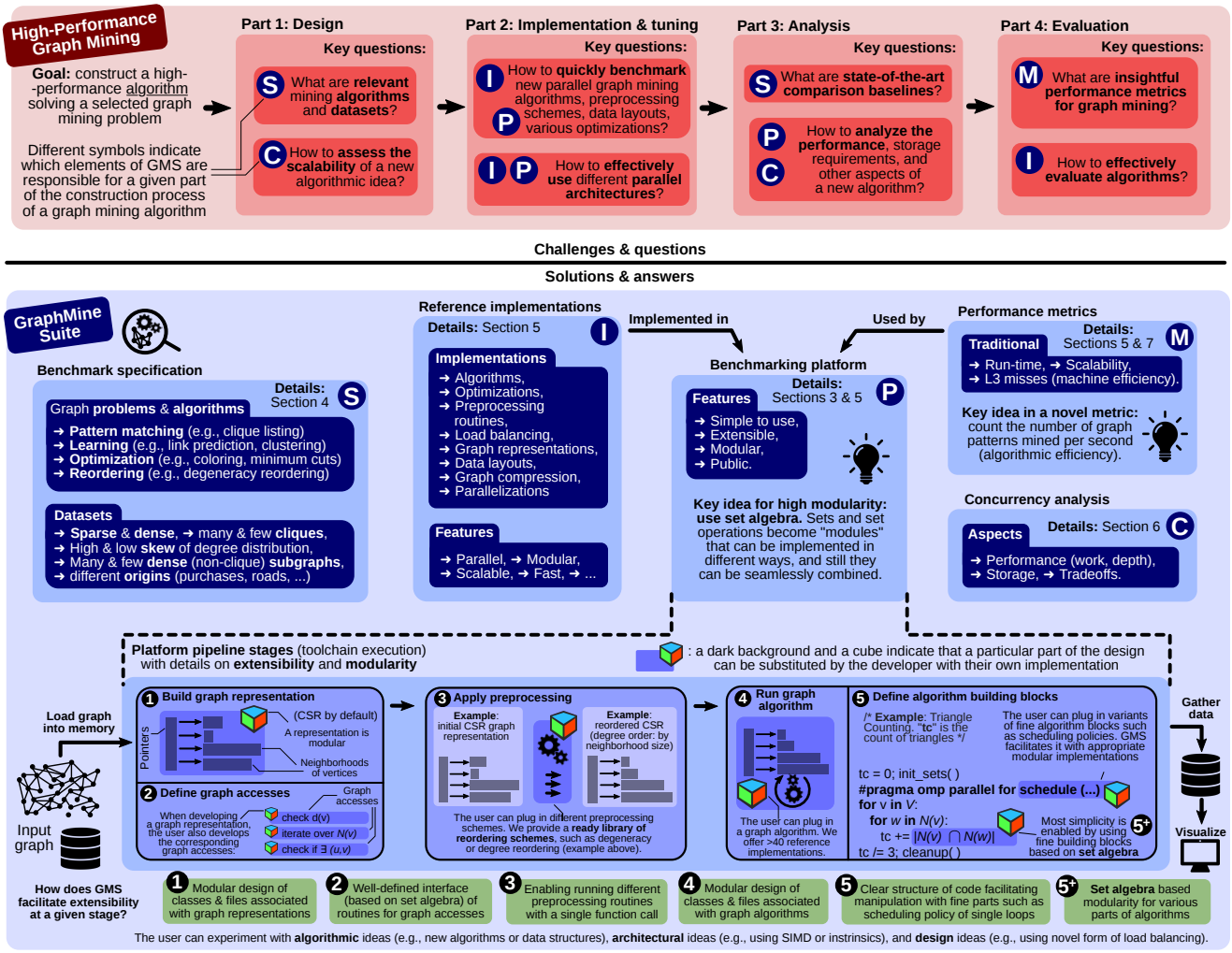


Figure 2: The overview of GMS and how it facilitates constructing, tuning, and benchmarking graph mining algorithms. The upper red part shows a process of constructing a graph mining algorithm, and the associated research questions. The middle blue part shows the corresponding different elements of the GMS suite (S – M). The bottom blue part illustrates the details of the GMS design benchmarking, with the stages of the GMS pipeline (execution toolchain) for running a given graph mining algorithm (1 – 5, 5+).

The GMS **benchmark specification** S (details in Section 4) motivates representative graph mining problems and state-of-the-art algorithms solving these problems, relevant datasets, performance metrics M, and a taxonomy that structures this information. The specification, in its entirety or in a selected subpart, enables choosing relevant comparison baselines and important datasets that stress different classes of algorithms.

The specification is implemented in the **benchmarking platform** P (details in Section 5). The platform facilitates developing and evaluating high-performance graph mining algorithms. The former is enabled by incorporating set algebra as the key driver for modularity *and* high performance. For the latter, the platform forms a processing pipeline with well-separated parts (see the bottom of Figure 2): loading the graph from I/O, constructing a graph representation (1 – 2), optional preprocessing (3) running selected graph algorithms (4 – 5, 5+), and gathering data.

The **reference implementation of algorithms** I (details in Section 6) offers publicly available, fast, and scalable baselines that effectively use massive parallelism in today's architectures. As data movement is dominating runtimes in irregular graph computations, we also provide a large number of *storage schemes*: graph representations, data layout schemes, and graph compression.

The **concurrency analysis** C (details in Section 7) offers a theoretical framework to analyze performance, storage, and the associated tradeoffs. We use work and depth [18, 20] that respectively describe the total work done by all executing processors, and the length of the associated longest execution path.

4 BENCHMARK SPECIFICATION

The GMS specification has four parts: graph mining ***problems***, ***algorithms***, ***datasets***, and ***metrics***[1].

4.1 Graph Problems and Algorithms

We identify **four** major classes of graph mining problems and the corresponding algorithms: **pattern matching, learning, reordering**, and (partially) **optimization**. For each given class of problems, we aimed to cover a *wide* range of problems and algorithms that

[1] We encourage participation in the GMS effort. If the reader would like to include some problem or algorithm in the specification and the platform, the authors would welcome the input.

differ in their design and performance characteristics, for example P and NP problems, heuristics and exact schemes, algorithms with time complexities described by low-degree and high-degree polynomials, etc.. The specification is summarized in Table 3.

4.1.1 ***Graph Pattern Matching.*** One large class is graph pattern matching [66], which focuses on finding specific subgraphs (also called *motifs* or *graphlets*) that are often (but not always) *dense*. Most algorithms solving such problems consist of the *searching part* (finding candidate subgraphs) and the *matching part* (deciding whether a given candidate subgraph satisfies the search criteria). The search criteria (the details of the searched subgraphs) influence the time complexity of both searching and matching. First, we pick **listing all cliques** in a graph, as this problem has a long and rich history in the graph mining domain, and numerous applications. We consider both **maximal** cliques (an NP-hard problem) and ***k*-cliques** (a problem with time complexity in $O(n^k)$), and the established associated algorithms, most importantly Bron-Kerbosch [24], Chiba-Nishizeki [37], and their various enhancements [29, 41, 51, 85, 117]. Next, we cover a more general problem of listing **dense subgraphs** [63, 74] such as *k*-cores, *k*-star-cliques, and others. GMS also includes the Frequent Subgraph Mining (FSM) problem [66], in which one finds **all subgraphs** (not just dense) that occur *more often than a specified threshold*. Finally, we include the established NP-complete **subgraph isomorphism** (SI) problem, because of its prominence in both the theory and practice of pattern matching, and because of a large number of variants that often have different performance characteristics [27, 40, 58, 89, 119]; SI is also used as a subroutine in the matching part of FSM.

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.

4.1.2 ***Graph Learning.*** We also consider various problems that can be loosely categorized as graph learning. These problems are mostly related to clustering, and they include **vertex similarity** [75, 101, 101] (verifying how similar two vertices are), **link prediction** [7, 80, 83, 114, 121] (predicting whether two non-adjacent vertices can become connected in the future, often based on vertex similarity scores), and **Clustering and Community Detection** [21, 65, 97] (finding various densely connected groups of vertices, also often incorporating vertex similarity as a subroutine).

4.1.3 ***Vertex Reordering.*** We also consider reordering of vertices. Intuitively, the order in which vertices are processed in some algorithm may impact the performance of this algorithm. For example, when counting triangles, ordering vertices by degrees (prior to counting) minimizes the number of times one triangle is (unnecessarily) counted more than once. In GMS, we first consider the abovementioned **degree ordering**. We also provide two algorithms for the **degeneracy ordering** [54] (**exact** and **approximate**), which was shown to improve the performance of maximal clique listing or graph coloring [15, 29, 51, 117].

4.1.4 ***Optimization.*** While GMS focuses less on optimization problems, we also include a representative problem of graph coloring, detailed in technical report.

4.1.5 ***Taxonomy and Discussion.*** Graph pattern matching, clustering, and optimization are related in that the problems from these classes focus on *finding certain subgraphs*. In the two former classes, such subgraphs are usually "local" groups of vertices, most often dense (e.g., cliques, clusters) [2–4, 14, 61, 96, 116], but sometimes can also be sparse (e.g., in FSM or SI). In optimization, a subgraph to be found can be "global", scattered over the whole graph (e.g., vertices with the same color). Moreover, *clustering and community detection (central problems in graph learning) are similar to dense subgraph discovery (a central problem in graph pattern matching)*. Yet, the latter use the notion of *absolute density*: a dense subgraph S is some relaxation of a clique (i.e., one does not consider what is "outside S"). Contrarily, the former use a concept of *relative density*: one compares different subgraphs to decide which one is dense [5].

4.2 Graph Datasets

We aim at a dataset selection that is computationally challenging for all considered problems and algorithms, cf. Table 3. We list both *large* and *small* graphs, to indicate datasets that can stress both low-complexity graph mining algorithms (e.g., centrality schemes or clustering) *and* high-complexity P, NP-complete, and NP-hard ones such as subgraph isomorphism.

So far, existing performance analyses on parallel graph algorithms focused on graphs with varying *sparsities* m/n (sparse and dense), *skews* in degree distribution (high and low skew), *diameters* (high and low), and *amounts of locality* that can be intuitively explained as the *number of inter-cluster* edges (many and few) [13]. In GMS, we recommend to use such graphs as well, as the above properties influence the runtimes of all described algorithms.

In Table 3, graphs with high degree distribution skews are indicated with large (relatively to n) maximum degrees Δ, which poses challenges for load balancing and others.

However, one of the insights that we gained with GMS is that ***the higher-order structure**, important for the performance of graph mining, can be little related to the above properties*. For example,

	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

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 (1 – 5, 5+). "👍": 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.

in § 8.6, we describe two graphs with almost identical sizes, sparsities, and diameters, but very different performance characteristics for 4-clique mining. As we detail in § 8.6, this is because the origin of these graphs determines whether a graph has many cliques *or* dense (but mostly non-clique) clusters. Thus, we also explicitly recommend to use graphs of *different origins*. We provide details of this particular case in § 8.6 (cf. Livemocha and Flickr).

In addition, we explicitly consider the count of triangles T, as (1) it indicates clustering properties (and thus implies the amount of locality), and it gives hints on different higher-order characteristics (e.g., the more triangles per vertex, the higher a chance for having k-cliques for $k > 3$). Here, we also recommend using graphs that have *large differences in counts of triangles per vertex* (i.e., large T-skew). Specifically, a large difference between the *average* number of triangles per vertex T/n and the *maximum* T/n indicates that a graph may pose additional load balancing problems for algorithms that list cliques of possibly unbounded sizes, for example Bron-Kerbosch. We also consider such graphs, see Table 3.

Finally, GMS enables using synthetic graphs with the random uniform (the Erdős-Rényi model [53]) and power-law (the Kronecker model [77]) degree distributions. This is enabled by integrating the GMS platform with existing graph generators [13]. Using such synthetic graphs enables analyzing performance effects while *systematically* changing a specific *single* graph property such as n, m, or m/n, which is not possible with real-world datasets.

We stress that we refrain from prescribing concrete datasets as benchmarking input (1) for flexibility, (2) because the datasets themselves evolve and (3) the compute and memory capacities of architectures grow continually, making it impractical to stick to a fixed-sized dataset. Instead, in GMS, we analyze and discuss publicly available datasets in Section 8, making suggestions on their applicability for stressing performance of different algorithms.

4.3 Metrics

In GMS, we first use simple *running times* of algorithms (or their specific parts, for a *fine grained analysis*). Unless stated otherwise, we use all available CPU cores, to maximize utilization of the underlying system. We also consider *scalability analyses*, illustrating how the runtime changes with the increasing amount of parallelism (#threads). Comparison between the measured scaling behavior and the ideal speedup helps to identify potential scalability bottlenecks. Finally, we consider *memory consumption*.

We also assess the **machine-efficiency**, i.e., *how well a machine is utilized in terms of its memory bandwidth*. For this, we consider CPU core utilization, expressed with counts of stalled CPU cycles. One can measure this number easily with, for example, the established PAPI infrastructure [92] that enables gathering detailed performance data from hardware counters. As we will discuss in detail in Section 5, we seamlessly integrate GMS with PAPI, enabling gathering detailed data such as stalled CPU cycles but also more than that, for example cache misses and hits (L1, L2, L3, data vs. instruction, TLB), memory reads/writes, and many others.

Finally, we propose a new metric for measuring the "**algorithmic efficiency**" ("**algorithmic throughput**"). Specifically, we measure the *number of mined graph patterns in a time unit*. Intuitively, this metric indicates how efficient a given algorithm is in finding respective graph elements. An example such metric used in the past is *processed edges per second* (PEPS), used in the context of graph traversals and PageRank [81]. Here, we extend it to graph mining and to arbitrary graph patterns. In graph pattern matching, this metric is the number of the respective *graph subgraphs found per second* (e.g., maximal cliques per second). In graph learning, it is a count of vertex pairs with similarity derived per second (vertex similarity, link prediction), or the number of clusters/communities found per second (clustering, community detection). The algorithmic efficiency facilitates deriving performance insights associated with the structure of the processed graphs. By comparing relative throughput differences between different algorithms *for different input graphs*, one can conclude whether these differences consistently depend on *pattern (e.g., clique) density*.

5 GMS PLATFORM & SET ALGEBRA

We now detail the GMS platform and how it enables modularity, extensibility, and high performance. There are six main ways in which one can experiment with a graph mining algorithm using the GMS platform, indicated in Figure 2 with 1 – 5+ and a block 🧊.

First, the user can provide a *new graph representation* 1 and the associated *routines for accessing the graph structure* 2. By default, GMS uses Compressed Sparse Row (CSR). Adding a new graph representation is facilitated by a modular design of the representation

code, and a concise interface (checking the degree $d(v)$, loading neighbors $N(v)$, iterating over vertices V or edges E, and verifying if an edge (u, v) exists) between a representation and the rest of GMS. The GMS platform also supports compressed graph representations. While many compression schemes focus on minimizing the amount of used storage [22] and require expensive decompression, some graph compression techniques entail *mild* decompression overheads, and they can even lead to overall *speedups* due to lower pressure on the memory subsystem [17]. Here, we offer ready-to-go implementations of such schemes, including bit packing, vertex relabeling, Log(Graph) [17], and others.

Second, the user can seamlessly add *preprocessing routines* ❸ such as the reordering of vertices. Here, the main motivation is that by applying a relevant vertex reordering (relabeling), one can reduce the amount of work to be done in the actual following graph mining algorithm. For example, the degeneracy order can significantly reduce the work done when listing maximal cliques [54]. The user runs a selected preprocessing scheme with a single function call that takes as its argument a graph to be processed.

Third, one can plug in a *whole new graph algorithm* ❹. GMS also facilitates modifying *fine parts of an algorithm* ❺, such as a scheduling policy of a loop. For this, we ensure a *modular structure of the respective implementations, and annotate code.*

Finally, we use the fact that many graph algorithms, for example Bron-Kerbosch [24] and others [1, 27–29, 41, 42, 51, 51, 57, 117, 121], are formulated with *set algebra* and use a small group of well-defined operations such as set intersection $\cap$. In GMS, we enable the user to provide their own implementation of such operations and of the data layout of the associated sets. This facilitates controlling the layout of a single auxiliary data structure or an implementation of a particular subroutine (indicated with ❺'). Thus, one is able to break complex graph mining algorithms into simple building blocks, and work on these building blocks independently. We already implemented a wide selection of routines for $\cap$, $\cup$, $\setminus$, $|\cdot|$, and $\in$; we also offer different set layouts based on integer arrays, bit vectors, and compressed variants of these two.

Set algebra building blocks in GMS are sets, set operations, set elements, and set algebra based graph representations. The first three are grouped together in the `Set` interface. The last one is a separate class that appropriately combines the instances of a given `Set` implementation. We now detail each of these parts.

5.1 Set Interface

The `Set` interface, illustrated in Listing 1, encapsulates the representation of an arbitrary set and its elements, and the corresponding set algorithms. By default, set elements are vertex IDs (modeled as integers) but other elements (i.e., integer tuples to model edges) can also be used. Then, there are three types of methods in `Set`.

First, there are methods implementing set basic set algebra operations, i.e., "union" for $\cup$, "intersect" for $\cap$, and "diff" for $\setminus$. To enable performance tuning, they come in variants. "_inplace" indicates that the calling object is being modified, as opposed to the default method variant that returns a new set (avoiding excessive data copying). "_count" indicates that the result is the size of the resulting set, e.g., $|A \cap B|$ instead of $A \cap B$ (avoiding creating unnecessary structures). Then, `add` and `remove` enable devising optimized variants of $\cup$ and $\setminus$ in which only one set element is inserted or removed from a set; these methods always modify the calling set.

```
1 class Set {
2 public:
3 //In methods below, we denote "*this" pointer with A
4 //(1) Set algebra methods:
5   Set diff(const Set &B) const; //Return a new set C = A \ B
6   Set diff(SetElement b) const; //Return a new set C = A \ {b}
7   void diff_inplace(const Set &B); //Update A = A \ B
8   void diff_inplace(SetElement b); //Update A = A \ {b}
9   Set intersect(const Set &B) const; //Return a new set C = A ∩ B
10  size_t intersect_count(const Set &B) const; //Return |A ∩ B|
11  void intersect_inplace(const Set &B); //Update A = A ∩ B
12  Set union(const Set &B) const; //Return a new set C = A ∪ B
13  Set union(SetElement b) const; //Return a new set C = A ∪ {b}
14  Set union_count(const Set &B) const; //Return |A ∪ B|
15  void union_inplace(const Set &B); //Update A = A ∪ B
16  void union_inplace(SetElement b); //Update A = A ∪ {b}
17  bool contains(SetElement b) const; //Return b ∈ A ? true:false
18  void add(SetElement b); //Update A = A ∪ {b}
19  void remove(SetElement b); //Update A = A \ {b}
20  size_t cardinality() const; //Return set's cardinality
21 //(2) Constructors (selected):
22  Set(const SetElement *start, size_t count); //From an array
23  Set(); Set(Set &&); //Default and Move constructors
24  Set(SetElement); //Constructor of a single-element set
25  static Set Range(int bound); //Create set {0, 1, ..., bound − 1}
26 //(3) Other methods:
27  begin() const; //Return iterators to set's start
28  end() const; //Return iterators to set's end
29  Set clone() const; //Return a copy of the set
30  void toArray(int32_t *array) const; //Convert set to array
31  operator==; operator!=; //Set equality/inequality comparison
32
33 private:
34  using SetElement = GMS::NodeId; //(4) Define a set element
35 }
```

Algorithm 1: The set algebra interface provided by GMS.

GMS offers other methods for performance tuning. This includes constructors (e.g., a move constructor, a constructor of a single-element set, or constructors from an array, a vector, or an initializer list), and general methods such as `clone`, which is used because – by default – the copy constructor is disabled for sets to avoid accidental data copying. GMS also offers conversion of a set to an integer array to facilitate using established parallelization techniques.

5.2 Implementations of Sets & Set Algorithms

On one hand, a set A can be represented as a contiguous sparse array with integers modeling vertex IDs ("sparse" indicates that only non-zero elements are explicitly stored), of size $W \cdot |A|$, where W is the memory word size [bits]. This representation is commonly used to store vertex neighborhoods. However, one can *also* represent A with a dense bitvector of size n [bits], where the i-th set bit means that a vertex $i \in A$ ("dense" indicates that all zero bits are explicitly stored). While being usually larger than a sparse array, a dense bitvector is more space-efficient when A is *very* large, which happens when some vertex connects to the majority of all vertices. Now, depending on A's and B's representations, $A \cap B$ can itself be implemented with different set algorithms. For example, if A and B are sorted sparse arrays with similar sizes ($|A| \approx |B|$), one prefers the "merge" scheme where one simply iterates through A and B, identifying common elements (taking $O(|A| + |B|)$ time). If one set (e.g., B) is represented as a bitvector, one may prefer a scheme where one iterates over the elements of a sparse array A and checks if each element is in B, which takes $O(1)$ time, giving the total of $O(|A|)$ time for the whole intersection.

Moreover, a bitvector enables insertion or deletion of vertices into a set in $O(1)$ time, which is useful in algorithms that rely on dynamic sets, for example Bron-Kerbosch [29, 42, 51, 117]. There are more set representations with other performance characteristics, such as sparse [1, 57] or compressed [16] bitvectors, or hashtables, enabling further performance/storage tradeoffs.

Importantly, using different *set representations* or *set algorithms* does *not* impact the formulations of *graph algorithms*. GMS exploits this fact to facilitate development and experimentation.

By default, GMS offers three implementations of `Set` interface:

- **RoaringSet** A set is implemented with a bitmap compressed using recent "roaring bitmaps" [32, 76]. A roaring bitmap offers diverse compression forms within the same bitvector. They offer mild compression rates but do not incur expensive decompression. As we later show, these structures result in high performance of graph mining algorithms running on top of them.
- **SortedSet** GMS also offers sets stored as sorted vectors. This reflects the established CSR graph representation design, where each neighborhood is a sorted contiguous array of integers.
- **HashSet** Finally, GMS offers an implementation of Set with a hashtable. By default, we use the Robin Hood library [30].

5.3 Set-Centric Graph Representations

Sets are building blocks for a graph representation: one set implements one neighborhood. To enable using arbitrary set designs, GMS harnesses templates, typed by the used set definition, see Listing 2. GMS provides ready-to-go representations based on the RoaringSet, SortedSet, and HashSet set representations.

```
1 template <class TSet>
2 class SetGraph {
3 public:
4   using Set = TSet; int64_t num_nodes() const;
5   const Set& out_neigh(NodeId node) const;
6   int64_t out_degree(NodeId node) const;
7   /* Some functions omitted */ };
```

Algorithm 2: A generic graph representation.

6 HIGH-PERFORMANCE & SIMPLICITY

We now detail how using the GMS benchmarking platform leads to simple (i.e., programmable) *and* high-performance implementations of many graph mining algorithms.

We now use the GMS benchmarking platform to enhance existing graph mining algorithms. We provide consistent speedups (detailed in Section 8). Some new schemes also come with theoretical advancements (detailed in Section 7). The following descriptions focus on (1) how we ensure the ***modularity*** of GMS algorithms (for programmability), and (2) what GMS design choices ensure ***speedups***. Selected modular parts are marked with the blue color and the type of modularity (❶ – ❺+). Marked set operations are implemented using the `Set` interface, see Listing 1.

Use Case 1: Degeneracy Order & k-Cores A *degeneracy* of a graph G is the smallest d such that every subgraph in G has a vertex of degree at most d. Thus, degeneracy can serve as a way to measure the graph sparsity that is "closed under taking a graph subgraph" (and thus more robust than, for example, the average degree). 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" [51]. DGR can be obtained by repeatedly removing a vertex of minimum degree in a graph. The derived DGR can be directly used to compute the k-core of G (a maximal connected subgraph of G whose all vertices have degree at least k). This is done by iterating over vertices in the DGR order, and removing vertices with out-degree less than k.

DGR, when used as a preprocessing routine, has been shown to accelerate different algorithms such as Bron-Kerbosch [51]. In the GMS benchmarking platform, we provide an implementation of DGR that is modular and can be seamlessly used with other graph algorithms as preprocessing (❸). Moreover, we alleviate the fact that the default DGR is not easily parallelizable and takes $O(n)$ iterations even in a parallel setting. For this, GMS delivers a modular implementation of a recent $(2+\varepsilon)$-*approximate* degeneracy order [15] (ADG), which has $O(\log n)$ iterations for *any* $\varepsilon > 0$. Deriving ADG is in Algorithm 3. It is similar to computing the DGR, which iteratively removes vertices of the smallest degree. The main difference is that one removes in parallel a *batch* of vertices with degrees smaller than $(1+\varepsilon)\widehat{\delta_U}$ (cf. set R and Line 7). The parameter $\varepsilon \geq 0$ controls the accuracy of the approximation; $\widehat{\delta_U}$ is the average degree in the induced subgraph $G(U, E[U])$, U is a "working set" that tracks changes to V. ADG relies on set cardinality and set difference, enabling the GMS set algebra modularity (❺+).

```
1 //Input: A graph G❶. Output: Approx. degeneracy order (ADG) η.
2 i = 1 // Iteration counter
3 U = V //U is the induced subgraph used in each iteration i
4 while U ≠ ∅ do:
5   δ̂_U = ( Σ_{v∈U} |N_U(v)|❷ ) / |U| //Get the average degree in U
6   //R contains vertices assigned priority in this iteration:
7   R = {v ∈ U : |N_U(v)|❷ ≤ (1+ε)δ̂_U }
8   for v ∈ R in parallel❷❺ do: η(v) = i //assign the ADG order
9   U = U \ R❺ //Remove assigned vertices
10  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.

Use Case 2: Maximal Clique Listing Maximal clique listing, in which one enumerates all *maximal* cliques (i.e., fully-connected subgraphs not contained in a larger such subgraph) in a graph, is one of core graph mining problems [29, 37, 42, 43, 49, 67, 71, 72, 79, 82, 84, 95, 105, 109, 111, 118, 124, 125, 130]. The recursive backtracking algorithm by Bron and Kerbosch (BK) [24] together with a series of enhancements [42, 51, 52, 117] (see Algorithm 4) is an established and, in practice, the most efficient way of solving this problem. Intuitively, in BK, one iteratively considers each vertex v in a given graph, and searches for all maximal cliques that contain v. The search process is conducted *recursively*, by starting with a single-vertex clique $\{v\}$, and augmenting it with v's neighbors, one at a time, until a maximal clique is found.

Importantly, the order in which all the vertices are selected for processing (at the *outermost* level of recursion) may heavily impact the amount of work in the following iterations [42, 51, 52]. Thus, in GMS, we use different vertex orderings, integrated using the GMS preprocessing modularity (❸). One of our core enhancements is to use the ADG order (see above). As we will show, this brings theoretical (Section 7) and empirical (Section 8) advancements.

A key part are vertex sets P, X, and R. They together navigate the way in which the recursive search is conducted. P ("Potential") contains candidate vertices that will be considered for belonging to the clique currently being expanded. X ("eXcluded") are the vertices that are definitely *not* to be included in the current clique (X is maintained to avoid outputting the same clique more than once). R is a currently considered clique (may be non-maximal). In GMS, we extensively experimented with different set representations for P, X, and R, which was facilitated by the set algebra based modularity (❺+). Our goal was to use representations that enable fast "bulk" set operations such as intersecting large sets (e.g., $X \cap N(v)$ in Line 23) but also efficient fine-grained modifications of such sets (e.g., $X = X \cup \{v\}$ in Line 28). For this, we use roaring bitmaps. As

we will show (Section 8), using such bitvectors as representations of P, X, and R brings overall speedups of even more than 9×.

Now, at the outermost recursion level, for each vertex v_i, we have $R = \{v_i\}$ (Line 13). This means that the considered clique starts with v_i. Then, we have $P = N(v_i) \cap \{v_{i+1}, ..., v_n\}$ and $X = N(v_i) \cap \{v_1, ..., v_{i-1}\}$. This removes unnecessary vertices from P and X. As we proceed in a fixed order of vertices in the main loop, when starting a recursive search for $\{v_i\}$, we will definitely *not* include vertices $\{v_1, ..., v_{i-1}\}$ in P, and thus we can limit P to $N(v_i) \cap \{v_{i+1}, ..., v_n\}$ (a similar argument applies to R). Note that these intersections may be implemented as simple splitting of the neighbors $N(v_i)$ into two sets, based on the vertex order. This is another example of the decoupling of general simple set algebraic formulations in GMS and the underlying implementations (5+).

In each recursive call of BK-Pivot, each vertex from P is added to R to create a new clique candidate R_{new} explored in the following recursive call. In this recursive call, P and X are respectively restricted to $P \cap N(v)$ and $X \cap N(v)$ (any other vertices besides $N(v)$ would not belong to the clique R_{new} anyway). After the recursive call returns, v is moved from P (as it was already considered) to X (to avoid redundant work in the future). The key condition for checking if R is a *maximal* clique is $P \cup X == \emptyset$. If this is true, then no more vertices can be added to R (including the ones from X that were already considered in the past) and thus R is maximal.

The BK variant in GMS also includes an additional important optimization called *pivoting* [117]. Here, for any vertex $u \in P \cup X$, only u and its *non* neighbors (i.e., $P \setminus N(u)$) need to be tested as candidates to be added to P. This is because any potential maximal clique must contain *either* u *or* one of its non-neighbors. Otherwise, a potential clique could be enlarged by adding u to it. Thus, when selecting u (Line 20), one may use any scheme that *minimizes* $|P \setminus N(u)|$ [117]. The advantage of pivoting is that it further prunes the search space and thus limits the number of recursive calls.

For further performance improvements, we also use roaring bitmaps to implement graph neighborhoods, exploiting the GMS modularity of representations and set algebra (1, 2, 5+). Finally, we also provide other optimizations based on set algebra that further reduce work; they are described in the extended technical report.

Use Case 3: k-Clique Listing GMS enabled us to enhance a state-of-the-art k-clique listing algorithm [41]. Our GMS formulation is shown in Algorithm 5. We reformulated the original scheme (without changing its time complexity) to expose the implicitly used set operations (e.g., Line 18), to make the overall algorithm more modular. In general, the algorithm uses recursive backtracking. One starts with iterating over edges (2-cliques), in Lines 11–12. In each backtracking search step, the algorithm augments the considered cliques by one vertex v and restricts the search to neighbors of v that come after v in the used vertex order.

Two schemes marked with 3 indicate two preprocessing routines that appropriately reorder vertices and – for the obtained order – assign directions to the edges of the input graph G. Both are well-known optimizations that reduce the search space size [41]. For such a modified G, we denote *out-neighbors* of any vertex u with $N^+(u)$. Then, operations marked with 5+ refer to accesses to the graph structure and different set operations that can be replaced with any implementation, as long as it preserves the semantics of set membership, set cardinality, and set intersection.

The modular design and using set algebra enables us to easily experiment with different implementations of C_i, $N^+(u) \cap C_i$, and

```
1 /* Input: A graph G (1). Output: all maximal cliques. */
2
3 //Preprocessing: reorder vertices with DGR or ADG.
4   (v_1, v_2, ..., v_n) = preprocess(V, /* selected vertex order */) (3)
5
6 //Main part: conduct the actual clique enumeration.
7 for v_i ∈ (v_1, v_2, ..., v_n) do: //Iterate over V in a specified order
8   //For each vertex v_i, find maximal cliques containing v_i.
9   //First, remove unnecessary vertices from P (candidates
10  //to be included in a clique) and X (vertices definitely
11  //not being in a clique) by intersecting N(v_i) with vertices
12  //that follow and precede v_i in the applied order.
13  P = N(v_i) ∩ {v_{i+1}, ..., v_n} (5+); X = N(v_i) ∩ {v_1, ..., v_{i-1}} (5+); R = {v_i}
14
15  //Run the Bron-Kerbosch routine recursively for P and X.
16  BK-Pivot(P, {v_i}, X)
17
18 BK-Pivot(P, R, X) //Definition of the recursive BK scheme
19   if P ∪ X == 0 (5+): Output R as a maximal clique
20   u = pivot(P ∪ X) (5+) //Choose a "pivot" vertex u ∈ P ∪ X
21   for v ∈ P \ N(u) (5+): // Use the pivot to prune search space
22     //New candidates for the recursive search
23     P_new = P ∩ N(v) (5+); X_new = X ∩ N(v) (5+); R_new = R ∪ {v} (5+)
24     //Search recursively for a maximal clique that contains v
25     BK-Pivot(P_new, R_new, X_new)
26     //After the recursive call, update P and X to reflect
27     //the fact that v was already considered
28     P = P \ {v} (5+); X = X ∪ {v} (5+)
```

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

```
1 /*Input: A graph G (1), k ∈ ℕ Output: Count of k-cliques ck ∈ ℕ. */
2
3 //Preprocessing: reorder vertices with DGR or ADG.
4 //Here, we also record the actual ordering and denote it as η
5   (v_1, v_2, ..., v_n; η) = preprocess(V, /* selected vertex order */) (3)
6
7 //Construct a directed version of G using η. This is an
8 //additional optimization to reduce the search space:
9 G = dir(G) (3) //An edge goes from v to u iff η(v) < η(u)
10 ck = 0 //We start with zero counted cliques.
11 for u ∈ V in parallel do: (2) //Count u's neighboring k-cliques
12   C_2 = N^+(u); ck += count(2, G, C_2)
13
14 function count(i, G, C_i):
15   if (i == k): return |C_k| (5+) //Count k-cliques
16   else:
17     ci = 0
18    for v ∈ C_i (5+) do: //search within neighborhood of v
19       C_{i+1} = N^+(v) ∩ C_i (5+) // C_i counts i-cliques.
20       ci += count(i+1, G, C_{i+1})
21    return ci
```

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

others. For example, we successfully and rapidly redesigned the reordering scheme, reducing the number of pointer chasing and the total amounts of communicated data. We investigated the generated assembly code of the respective part; it has 22 x86 mov instructions, compared to 31 before the design enhancement[2]. Moreover, we improved the memory consumption of the algorithm. The space allocated per subgraph C_i (e.g., 5+) is now upper bounded by $|C_i|^2$ (counted in vertices) instead of the default Δ^2. When parallelizing over edges, this significantly reduces the required memory (for large maximum degrees Δ, even up to >90%).

7 CONCURRENCY ANALYSIS

In this part of GMS, we show how to assess a priori the properties of parallel graph mining algorithms, *reducing time spent on algorithm design and development* and providing *performance insights that are*

[2]We used "compiler explorer" (https://godbolt.org/) for assembly 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	$O\left(n+k\left(\frac{d}{2}\right)^{k-1}\right)$	$O\left(n+k\left(\frac{d}{2}\right)^{k-2}+d^2\right)$	$O\left(k\left(d+\frac{\varepsilon}{2}\right)^{k-2}+\log^2 n+d^2\right)$	$O\left(\log^2 n\right)$	$O\left(dm3^{d/3}\right)$	$O\left(d\log n\right)$	$O\left(\log^2 n+d\log n\right)$	$O\left(\Delta^{k-1}\right)$	$O(\Delta)$
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.

portable across machines that differ in certain ways (e.g., in the sizes of their caches) and independent of various implementation details.

7.1 Methodology, Models, Tools

We use the established *work-depth analysis* for bounding run-times of parallel algorithms. Here, the total number of instructions performed by an algorithm (over all number of processors for a given input size) is the *work* of the algorithm. The longest chain of sequential dependencies (for a given input size) is the *depth* of an algorithm [18, 20]. This approach is used in most recent formal analyses of parallel algorithms in the shared-memory setting [45, 59]. Overall, we consider *four* aspects of a parallel algorithm: (1) the overhead compared to a sequential counterpart, quantified with work, (2) the scalability, which is illustrated by depth, (3) the space usage, and – when applicable – (4) the approximation ratio.

7.2 Discussion On Trade-Offs

For many problems, there is a **tradeoff between work, depth, space**, and sometimes **approximation ratio** [41, 69, 90]. Which algorithm is the best choice hence depends on the available number of processors and the available main memory. For today's shared memory machines, typically the number of processors/cores is relatively small (e.g., 18 on our machines) and main memory is not much bigger than the graphs we would like to process (e.g., 64GiB or 768GiB on our machines, see Section 8). Thus, *reducing work (and maintaining close to linear space in the input plus output)* is a high priority to obtain good performance in practice [45].

An algorithm with a work that is much larger than the best sequential algorithm will require many processors to be faster than the latter. An algorithm with large depth will stop scaling for a small number of processors. An estimate of the runtime of an algorithm with work W and depth D on p processors is $W/p + D$. This estimate is optimistic as it neglects the cost for scheduling threads and caching issues (e.g., false sharing). Yet, it has proven a useful model in developing efficient graph algorithms in practice [45].

The space used by a parallel algorithm limits the largest problem that can be solved on a fixed machine. This is crucial for graph mining problems with exponential time complexities where we want the *space to be close to the input size plus the output size.*

We illustrate a work / depth / space tradeoff with k-clique listing [41] (§ 6). All following designs are pareto-optimal in terms of the work / depth / space tradeoff and they are useful in different circumstances (for different machines). First, consider a **naive** algorithm variant. Starting from every vertex, one spawns parallel recursive searches to complete the current clique. The advantage of this approach is that is has low depth $O(k)$, but the work and space is $\Theta(n\Delta^{k-1})$, which can be prohibitive.

This approach can be enhanced by using the DGR order to guide the search as described in § 6 (the "**Node Parallel**" variant). Here, one invokes a parallel search starting from each vertex for cliques that contain this vertex as the first vertex in the order. This reduces the space to almost linear $\Theta(nd^2)$, where d is the degeneracy of the graph. The depth is increased to $\Theta(n + k(d/2)^{k-1})$. This design was reported to have poor scalability in practice [41].

One can also invoke a parallel search for every *edge* ("**Edge Parallel**") and try to find a clique that contains it (and follows the DGR order). The depth decreases by a factor of d to $\Theta(n+k(d/2)^{k-2}+d^2)$, but the space increases by a factor of $\frac{m}{n}$ to $O(md^2)$. This approach has a good work / depth / space tradeoff in practice [41].

7.3 Bounds for Graph Mining Algorithms

Table 4 presents work-depth and space bounds for considered graph mining algorithms. Here, we obtain *new better* bounds for maximal clique listing. The main idea is to combine existing corresponding algorithms [41, 42, 52] (columns 2 and 6) with the ADG ordering. The k-clique listing variant parametrized by degeneracy scales better than Danisch et al. [41] (column 2) if n is much bigger than kd^{k-2}. The new maximal clique listing improves upon the Eppstein et al. [52] and Das et al. [42]: our depth is better than both while work is better than that of [52] and adds only a small factor to work in [52]. We provide detailed proofs in the technical report.

8 EVALUATION

We describe how GMS facilitates performance analysis of various aspects of graph mining, and accelerates the state of the art.

8.1 Datasets, Methodology, Architectures

We first sketch the evaluation methodology. For measurements, we omit the first 1% of performance data as warmup. We derive enough data for the mean and 95% non-parametric confidence intervals. We use arithmetic means as summaries.

8.1.1 Datasets. We consider SNAP (S) [78], KONECT (K) [73], DIMACS (D) [44], Network Repository (N) [102], and WebGraph (W) [22] datasets. As explained in § 4.2, for flexibility, we do not fix specific datasets. Instead, we illustrate a wide selection of public datasets in Table 5, arguing *which parameters make them useful or challenging*. Details of these parameters are in § 4.2.

8.1.2 Comparison Baselines. For each considered graph mining problem, we compare different GMS variants to *the most optimized state-of-the-art algorithms available.* We compare to the original existing implementations. Details are stated in the following sections.

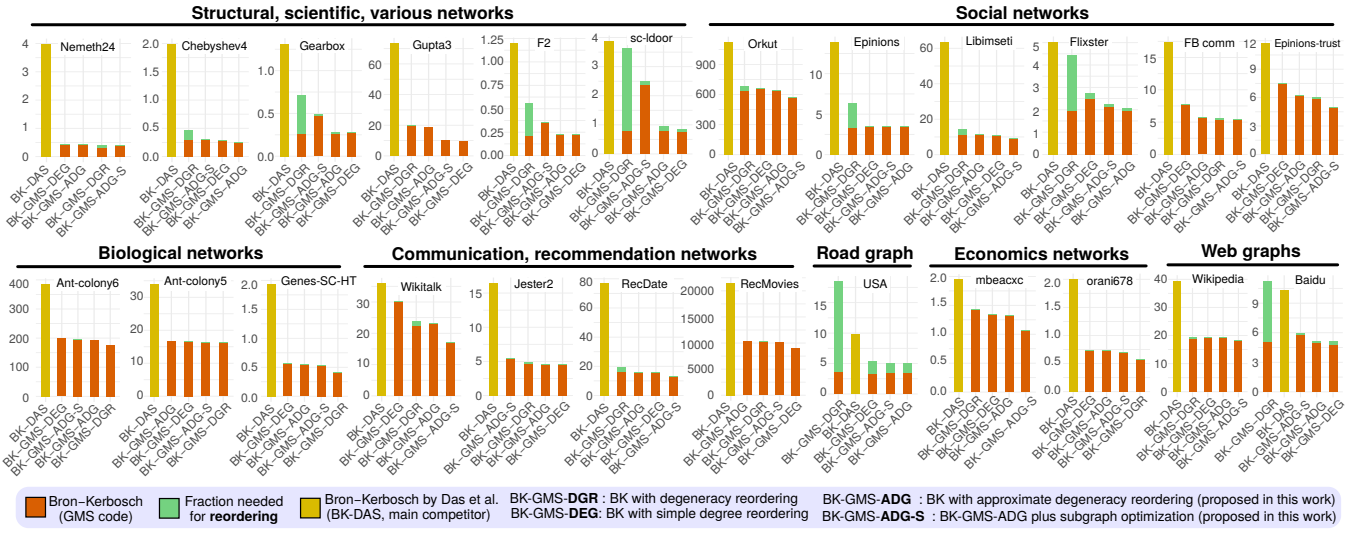


Figure 3: Speedups of the parallel GMS BK over a recent implementation by Das et al. [42] (BK-DAS) and a recent algorithm by Eppstein et al. [51] (BK-GMS-DGR). System: Daint.

8.1.3 Parallelism. Unless stated otherwise, we use *full parallelism*, i.e., we run algorithms on *the maximum number of cores available on a given system.*

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.

8.1.4 Architectures. We used different systems for a *broad evaluation and to analyze and ensure performance portability* of our implementations. First, we use an in-house Einstein and Euler servers. Einstein is a Dell PowerEdge R910 with an Intel Xeon X7550 CPUs @ 2.00GHz with 18MB L3 cache, 1TiB RAM, and 32 cores per CPU (grouped in four sockets). Euler has an HT-enabled Intel Xeon Gold 6150 CPUs @ 2.70GHz with 24.75MB L3 cache, 64 GiB RAM, and 36 cores per CPU (grouped in two sockets). We also use servers from the CSCS supercomputing center, most importantly a compute server with Intel Xeon Gold 6140 CPU @ 2.30GHz, 768 GiB RAM, 18 cores, and 24.75MB L3. Finally, we also used XC50 compute nodes in the Piz Daint Cray supercomputer (one such node comes with 12-core Intel Xeon E5-2690 HT-enabled CPU 64 GiB RAM).

8.2 Faster Maximal Clique Listing

We start with our key result: GMS enabled us to outperform a state-of-the-art fastest available algorithm for maximal clique listing by Das et al. [42] (BK-DAS) *by nearly an order of magnitude.* The results are in Figure 3. We compare BK-DAS with several variants of BK developed in GMS as described in § 6. BK-GMS-DGR uses the degeneracy order and is a variant of the Eppstein's scheme [51], enhanced in GMS. BK-GMS-DEG uses the simple degree ordering. BK-GMS-ADG and BK-GMS-ADG-S are two variants of a new BK algorithm proposed in this work, combining BK with the ADG ordering; the latter also uses the subgraph caching optimization (§ 6). We also compare to the original Eppstein scheme, it was always slower. GMS also enabled us to experiment with Intel Thread Building Blocks vs. OpenMP for threading in both the outermost loop and in inner loops (we exploit nested parallelism), we only show the OpenMP variants as they always outperform TBB.

Figure 3 shows consistent speedups of GMS variants over BK-DAS. We could quickly deliver these speedups by being able to plug in different set operations and optimizations in BK. Moreover, many plots show the large preprocessing overhead when using DGR. It sometimes helps to reduce the actual clique listing time (compared to ADG), but in most cases, "ADG plus clique listing" are faster than "DGR plus clique listing": ADG is very fast and it

reduces the BK runtime to the level comparable to that achieved by DGR. This confirms the theoretical predictions of the benefits of BK-GMS-ADG over BK-GMS-DGR or BK-DAS. Finally, the comparably high performance (for many graphs) of BK-GMS-ADG, BK-GMS-ADG-S, and BK-GMS-DEG is due to the optimizations based on set algebra, for example using fast *and* compressed roaring bitmaps to implement neighborhoods and auxiliary sets P, X, and R (cf. § 6), which enables fast set operations heavily used in BK. *Overall, BK-GMS is often faster than BK-DAS by >50%, in some cases even >9×.*

We stress that the speedups of the implementations included in the GMS benchmarking platform are consistent over many graphs of *different structural characteristics* (cf. Table 5) that entail deeply varying load balancing properties. For example, some graphs are *very* sparse, with virtually no cliques larger than triangles (e.g., the USA road network) while others are *relatively* sparse with *many* triangles (and higher cliques), with low or moderate skews in triangle counts per vertex (e.g., Gearbox or F2). Finally, some graphs have *large* or even *huge* skews in triangle counts per vertex (e.g., Gupta3 or RecDate), which gives significant differences in the depths of the backtracking trees and thus load imbalance.

We also derived the **algorithmic efficiency** results, i.e., the number of maximal cliques found per second; selected data is in Figure 1. The results follow the run-times; the GMS schemes consistently outperform BK-DAS (the plots are in the technical report). These results show more distinctively that BK-GMS finds maximal cliques consistently better than BK-DAS, even if input graphs have vastly different clustering properties. For example, BK-GMS-ADG outperforms BK-DAS for Gupta3 (huge T-skew), F2 (medium T-skew), and ldoor (low T-skew).

8.3 Faster k-Clique Listing

GMS also enabled us to accelerate a very recent k-clique listing algorithm [41]. We were able to rapidly experiment with different variants, such as node parallel and edge parallel schemes, described in § 6 and in Section 7. Our optimizations from § 6 (e.g., a memory-efficient layout of C_i) ensure consistent speedups of up to 10% for different parameters (e.g., clique size k), input graphs, and reordering routines. Additionally, we show that using the ADG order brings further speedups over DEG or DGR.

8.4 Faster Degeneracy Reordering and k-Cores

We also analyze in more detail the performance of different reordering routines (DEG, DGR, and ADG) and their impact on graph mining algorithms in GMS (cf. § 6). We also show their impact on the run-time of BK maximal clique listing by Eppstein et al. [51] (BK-E). The results are in Figure 4. ADG, due to its beneficial scalability properties (cf. Section 7), *outperforms the exact DGR.* At the same time, it *similarly* reduces the runtime of BK-E [51] (cf. leftmost and rightmost bars). The $2+\varepsilon$ approximation ratio has mild influence on performance. Specifically, the lower ε is, the more (mild) speedup is observed. This is because larger ε enables more parallelism, but then less accurate degeneracy ordering may incur more work when listing cliques. Moreover, ADG combined with BK-E cumulatively *outperforms the simple DEG reordering*: the latter is also fast, but its impact on the Bron-Kerbosch run-time is lower, ultimately failing to provide comparable speedups. We were able to rapidly experiment with different reorderings as – *thanks to GMS's modularity* – we could seamlessly integrate them with BK-E [51].

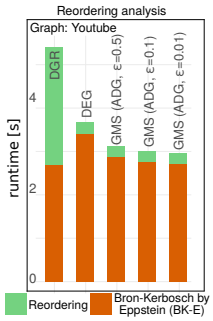


Figure 4: Speedups of ADG for different ε over DEG/DGR, details in § 8.4. System: Ault.

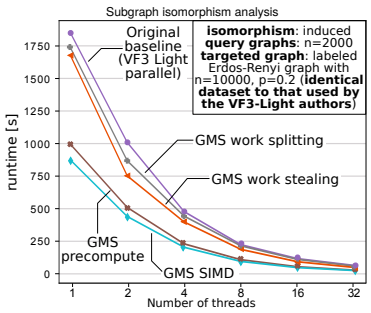


Figure 5: Speedups of different GMS variants of subgraph isomorphism over the state-of-the-art parallel VF3-Light baseline [28]. Details in § 8.5. System: Euler.

8.5 Faster Subgraph Isomorphism

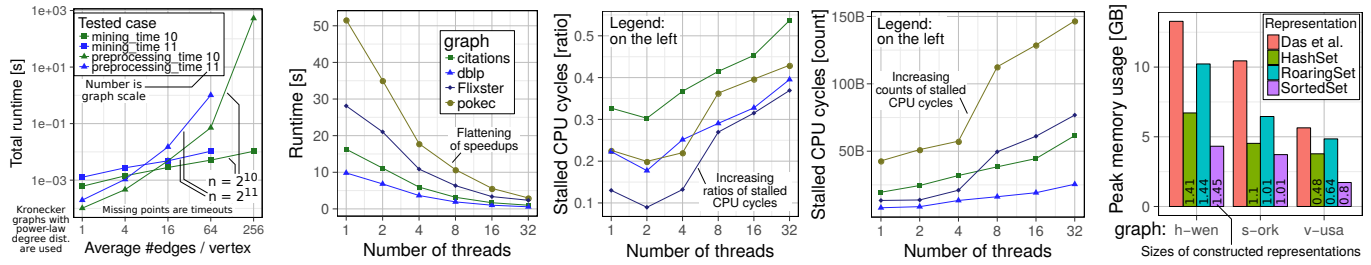
GMS enabled us to accelerate a very recent parallel VF3-Light subgraph isomorphism baseline by 2.5×. The results are in Figure 5 (we use the same dataset as in the original work [28]). We illustrate the impact from different optimizations outlined in § 6. We were also able to use SIMD vectorization in the binary search part of the algorithms, leading to additional 1.1× speedup.

8.6 Additional Analyses

Subtleties of Higher-Order Structure One of the insights that we gained with GMS is that graphs similar in terms of n, m, sparsity m/n, and degree distributions, may have very different characteristics in their higher-order structure. For example, a graph of photo relations in Flickr and a Livemocha social network (see Table 5 for details) are similar in the above properties, but the former has 9,578,965,096 4-cliques while the latter has only 4,359,646 4-cliques. This is because, while a in a social network 4-cliques of friendships may be only *relatively common*, they should *occur very often* in a network where photos are related if they share *some* metadata (e.g., location). Thus, one should *carefully select input datasets* to properly evaluate respective graph mining algorithms, as seemingly similar graphs may have very different higher-order characteristics, which may vastly impact performance and conclusions when developing a new algorithm.

Analysis of Synthetic Graphs We illustrate example results for synthetic graphs, see Figure 6a (with BK-GMS-DGR). Using power-law Kronecker graphs enable us to study the performance impact from varying the graph sparsity m/n while fixing all other parameters. For very sparse graphs, the cost of mining cliques is much lower than that of vertex reordering during preprocessing. However, as m/n increases, reordering begins to dominate. This is because Kronecker graphs in general do not have large cliques, which makes the mining process finish relatively fast, while reordering costs grow proportionally to m/n.

Machine Efficiency Analysis We show example analysis of CPU utilization, using the PAPI interface provided in GMS, see Figure 6b. The plots illustrate the flattening of speedups with the increasing #threads, accompanied by the steady growth of stalled CPU cycles (both total counts and ratios), illustrating that maximal clique listing is memory bound [36, 50, 64, 129, 130].



(a) Analysis of synthetic graphs. **(b) Analysis of machine efficiency.** **(c) Sizes of GMS graph representations.**

Figure 6: Additional analyses for the parallel GMS BK algorithm (BK-GMS-DGR). System: Daint.

Memory Consumption Analysis We illustrate example memory consumption results in Figure 6c; we compare the size of three GMS set-centric graph representations, showing both *peak* memory usage when constructing a representation (bars) and sizes of ready representations (all in GB). Interestingly, while the latter are similar (except for v-usa), peak memory usage is visibly highest for RoaringSet. We also compare to the representation used by Das et al. [42], it always comes with the highest peak storage costs.

Algorithmic Throughput Analysis The advantages of using algorithmic throughput can be seen by comparing Figure 1 and 3. While plain runtimes illustrate which algorithm is faster for which graph, the algorithmic throughput also *enables combining this outcome with the input graph structure*. For example, the GMS variants of BK have relatively lower advantages over BK by Das et al. [42] *whenever the input graph has a higher density of maximal cliques*. This motivates using the GMS variants of BK especially for very sparse graphs without large dense clusters. One can derive analogous insights for any other patterns such a k-cliques.

GMS and Graph Processing Benchmarks There is very little overlap with GMS and existing graph processing benchmarks, see Section 1 and Table 1. The closest one is GBBS [45], which supports the exact same variant of mining k-cliques. We compare GBBS to GMS (details are in the technical report); we also consider the edge-based very recent implementation by Danisch et al. [41]. GMS offers consistent advantages for different graphs and large clique sizes.

GMS and Pattern Matching Frameworks There is also little overlap between GMS and pattern matching frameworks, cf. Table 1. While they support mining patterns, they focus on patterns of fixed sizes (e.g., k-cliques). We compare GMS to two very recent frameworks that, similarly to GMS, target shared-memory parallelism, Peregrine [64] and RStream [120]. Peregrine can only list k-cliques. It does not offer a native scheme for maximal clique listing and we implement it by iterating over k-cliques of different sizes (we consult the authors of Peregrine to find the best scheme). RStream is only able to find k-cliques. Overall, GMS is much faster in all considered schemes (10-100× over Peregrine and more than 100× over RStream). This is because these systems focus on programming abstractions, which improves programmability but comes with performance overheads. GMS enables maximizing performance of tuned parallel algorithms targeting specific problems.

9 CONCLUSION

We introduce GraphMineSuite (GMS), the first benchmarking suite for graph mining algorithms. GMS offers an extensive *benchmark specification* and taxonomy that distill more than 300 related works and can aid in selecting appropriate comparison baselines. Moreover, GMS delivers a highly modular *benchmarking platform*, with dozens of parallel implementations of key graph mining algorithms and graph representations. Unlike *frameworks for pattern matching* which focus on abstractions and programming models for expressing mining specific patterns, GMS simplifies designing high-performance *algorithms for solving specific graph mining problems* from a *wide* graph mining area. Extending GMS towards distributed-memory systems or dynamic workloads are interesting future lines of work. Third, GMS' *concurrency analysis* illustrates theoretical tradeoffs between time, work, storage, and accuracy, of several representative problems in graph mining; it can be used as a guide when rapidly analyzing the scalability of a planned graph mining scheme. Finally, we show GMS' potential by using it to *enhance state-of-the-art graph mining algorithms*, leading to theoretical and empirical advancements in maximal clique listing (speedups by >9× and better work-depth bounds over the *fastest known Bron-Kerbosch baseline*), degeneracy reordering and core decomposition (speedups by >2×), k-clique listing (speedups by up to 1.1× and better bounds), and subgraph isomorphism (speedups by 2.5×).

ACKNOWLEDGEMENTS

We thank Hussein Harake, Colin McMurtrie, Mark Klein, Angelo Mangili, and the whole CSCS team granting access to the Ault and Daint machines, and for their excellent technical support. We thank Timo Schneider for immense help with computing infrastructure at SPCL. We thank Maximilien Danisch, Oana Balalau, Mauro Sozio, Apurba Das, Seyed-Vahid Sanei-Mehri, and Srikanta Tirthapura for the implementations of their algorithms for solving k-clique and maximal clique listing. We thank Dimitrios Lekkas, Athina Sotiropoulou, Foteini Strati, Andreas Triantafyllos, Kenza Amara, Chia-I Hu, Ajaykumar Unagar, Roger Baumgartner, Severin Kistler, Emanuel Peter, and Alain Senn for helping with the implementation in the early stages of the project. Funded by the Google European Doctoral Fellowship and European Research Council (ERC) under the European Union's Horizon 2020 programme grant No. 678880.

REFERENCES

[1] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)*, 42(4):1–44, 2017.

[2] M. Adedoyin-Olowe, M. M. Gaber, and F. Stahl. A survey of data mining techniques for social media analysis. *arXiv preprint arXiv:1312.4617*, 2013.

[3] C. C. Aggarwal and H. Wang. Graph data management and mining: A survey of algorithms and applications. In *Managing and mining graph data*, pages 13–68. Springer, 2010.

[4] C. C. Aggarwal and H. Wang. A survey of clustering algorithms for graph data. In *Managing and mining graph data*, pages 275–301. Springer, 2010.

[5] C. C. Aggarwal, H. Wang, et al. *Managing and mining graph data*, volume 40. Springer, 2010.

[6] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan. Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *2015 IEEE International Symposium on Workload Characterization*, pages 44–55. IEEE, 2015.

[7] M. Al Hasan et al. Link prediction using supervised learning. In *SDM*, 2006.

[8] M. Al Hasan and M. J. Zaki. A survey of link prediction in social networks. In *Social network data analytics*, pages 243–275. Springer, 2011.

[9] K. Ammar and M. T. Özsu. Wgb: Towards a universal graph benchmark. In *Advancing Big Data Benchmarks*, pages 58–72. Springer, 2013.

[10] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: a database benchmark based on the facebook social graph. In *ACM SIGMOD*, pages 1185–1196, 2013.

[11] D. A. Bader and K. Madduri. Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. In *International Conference on High-Performance Computing*, pages 465–476. Springer, 2005.

[12] S. Bassini, M. Danelutto, and P. Dazzi. *Parallel Computing is Everywhere*, volume 32. IOS Press, 2018.

[13] S. Beamer, K. Asanović, and D. Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[14] P. Berkhin. A survey of clustering data mining techniques. In *Grouping multidimensional data*, pages 25–71. Springer, 2006.

[15] M. Besta, A. Carigiet, Z. Vonarburg-Shmaria, K. Janda, L. Gianinazzi, and T. Hoefler. High-performance parallel graph coloring with strong guarantees on work, depth, and quality. In *ACM/IEEE Supercomputing*, 2020.

[16] M. Besta and T. Hoefler. Survey and taxonomy of lossless graph compression and space-efficient graph representations. *arXiv preprint arXiv:1806.01799*, 2018.

[17] M. Besta, D. Stanojevic, T. Zivic, J. Singh, M. Hoerold, and T. Hoefler. Log (graph): a near-optimal high-performance graph representation. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, page 7. ACM, 2018.

[18] G. Bilardi and A. Pietracaprina. *Models of Computation, Theoretical*, pages 1150–1158. Springer US, Boston, MA, 2011.

[19] G. E. Blelloch. Problem based benchmark suite, 2011.

[20] G. E. Blelloch and B. M. Maggs. *Parallel Algorithms*, page 25. Chapman & Hall/CRC, 2 edition, 2010.

[21] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[22] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *World Wide Web Conf. (WWW)*, pages 595–601, 2004.

[23] P. Boncz. LDBC: benchmarks for graph and RDF data management. In *IDEAS*, 2013.

[24] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *CACM*, 1973.

[25] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on gpus. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 141–151. IEEE, 2012.

[26] V. Carletti et al. Introducing vf3: A new algorithm for subgraph isomorphism. In *Springer GbRPR*, 2017.

[27] V. Carletti et al. The VF3-light subgraph isomorphism algorithm: when doing less is more effective. In *Springer S+SSPR*, 2018.

[28] V. Carletti et al. A parallel algorithm for subgraph isomorphism. In *Springer GbRPR*, 2019.

[29] F. Cazals and C. Karande. A note on the problem of reporting maximal cliques. *Theoretical Computer Science*, 407(1-3):564–568, 2008.

[30] P. Celis. *Robin hood hashing*. University of Waterloo, 1986.

[31] D. Chakrabarti and C. Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM CSUR*, 2006.

[32] S. Chambi, D. Lemire, O. Kaser, and R. Godin. Better bitmap performance with roaring bitmaps. *Software: practice and experience*, 46(5):709–719, 2016.

[33] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee, 2009.

[34] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng. G-miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference*, page 32. ACM, 2018.

[35] X. Chen, R. Dathathri, G. Gill, and K. Pingali. Pangolin: An efficient and flexible graph mining system on cpu and gpu. *arXiv preprint arXiv:1911.06969*, 2019.

[36] J. Cheng, L. Zhu, Y. Ke, and S. Chu. Fast algorithms for maximal clique enumeration with limited memory. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1240–1248, 2012.

[37] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on computing*, 14(1):210–223, 1985.

[38] A. Ching et al. One trillion edges: Graph processing at facebook-scale. *VLDB*, 2015.

[39] D. J. Cook and L. B. Holder. *Mining graph data*. John Wiley & Sons, 2006.

[40] L. P. Cordella et al. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE TPAMI*, 2004.

[41] M. Danisch et al. Listing k-cliques in sparse real-world graphs. In *WWW*, 2018.

[42] A. Das et al. Shared-memory parallel maximal clique enumeration. In *IEEE HiPC*, 2018.

[43] A. Das, M. Svendsen, and S. Tirthapura. Change-sensitive algorithms for maintaining maximal cliques in a dynamic graph. *CoRR, vol. abs/1601.06311*, 2016.

[44] C. Demetrescu, A. V. Goldberg, and D. S. Johnson. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74. American Math. Soc., 2009.

[45] L. Dhulipala et al. Theoretically efficient parallel graph algorithms can be fast and scalable. In *ACM SPAA*, 2018.

[46] L. Dhulipala, J. Shi, T. Tseng, G. E. Blelloch, and J. Shun. The graph based benchmark suite (gbbs). In *GRADES and NDA*, pages 1–8, 2020.

[47] V. Dias et al. Fractal: A general-purpose graph pattern mining system. In *ACM SIGMOD*, 2019.

[48] R. Diestel. *Graph theory*. Springer, 2018.

[49] N. Du, B. Wu, L. Xu, B. Wang, and X. Pei. A parallel algorithm for enumerating all maximal cliques in complex network. In *Sixth IEEE International Conference on Data Mining-Workshops (ICDMW'06)*, pages 320–324. IEEE, 2006.

[50] J. D. Eblen, C. A. Phillips, G. L. Rogers, and M. A. Langston. The maximum clique enumeration problem: algorithms, applications, and implementations. In *BMC bioinformatics*, volume 13, page S5. Springer, 2012.

[51] D. Eppstein et al. Listing all maximal cliques in sparse graphs in near-optimal time. In *SAAC*, 2010.

[52] D. Eppstein and D. Strash. Listing all maximal cliques in large sparse real-world graphs. In *International Symposium on Experimental Algorithms*, pages 364–375. Springer, 2011.

[53] P. Erdős and A. Rényi. On the evolution of random graphs. *Selected Papers of Alfréd Rényi*, 1976.

[54] M. Farach-Colton and M. Tsai. Computing the degeneracy of large graphs. In *LATIN*, 2014.

[55] B. Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. In *AAAI Fall Symposium: Capturing and Using Patterns for Evidence Detection*, pages 45–53, 2006.

[56] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin. An experimental comparison of pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 7(12):1047–1058, 2014.

[57] S. Han, L. Zou, and J. X. Yu. Speeding up set intersections in graph algorithms using simd instructions. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1587–1602. ACM, 2018.

[58] W.-S. Han et al. Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *ACM SIGMOD/PODS*. ACM, 2013.

[59] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson. Ordering heuristics for parallel graph coloring. In *26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '14, Prague, Czech Republic - June 23 - 25, 2014*, pages 166–177, 2014.

[60] T. Horváth et al. Cyclic pattern kernels for predictive graph mining. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2004.

[61] M. Injadat, F. Salo, and A. B. Nassif. Data mining techniques in social media: A survey. *Neurocomputing*, 214:654–670, 2016.

[62] A. P. Iyer, Z. Liu, X. Jin, S. Venkataraman, V. Braverman, and I. Stoica. {ASAP}: Fast, approximate graph pattern mining at scale. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 745–761, 2018.

[63] S. Jabbour, N. Mhadhbi, B. Raddaoui, and L. Sais. Pushing the envelope in overlapping communities detection. In *International Symposium on Intelligent Data Analysis*, pages 151–163. Springer, 2018.

[64] K. Jamshidi, R. Mahadasa, and K. Vora. Peregrine: a pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[65] R. A. Jarvis and E. A. Patrick. Clustering using a similarity measure based on shared near neighbors. *IEEE Transactions on computers*, 100(11):1025–1034, 1973.

[66] C. Jiang, F. Coenen, and M. Zito. A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review*, 28(1):75–105, 2013.

[67] D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27(3):119–123, 1988.

[68] A. Joshi, Y. Zhang, P. Bogdanov, and J.-H. Hwang. An efficient system for subgraph discovery. In *2018 IEEE International Conference on Big Data (Big*

Data), pages 703–712. IEEE, 2018.
[69] D. R. Karger and C. Stein. A new approach to the minimum cut problem. *Journal of the ACM (JACM)*, 43(4):601–640, 1996.
[70] W. Khaouid et al. K-core decomposition of large networks on a single pc. *Proceedings of the VLDB Endowment*, 9(1):13–23, 2015.
[71] I. Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theoretical Computer Science*, 250(1-2):1–30, 2001.
[72] F. Kose, W. Weckwerth, T. Linke, and O. Fiehn. Visualizing plant metabolomic correlation networks using clique–metabolite matrices. *Bioinformatics*, 17(12):1198–1208, 2001.
[73] J. Kunegis. Konect: the koblenz network collection. In *Proc. of Intl. Conf. on World Wide Web (WWW)*, pages 1343–1350. ACM, 2013.
[74] V. E. Lee, N. Ruan, R. Jin, and C. Aggarwal. A survey of algorithms for dense subgraph discovery. In *Managing and Mining Graph Data*, pages 303–336. Springer, 2010.
[75] E. A. Leicht et al. Vertex similarity in networks. *Physical Review E*, 73(2):026120, 2006.
[76] D. Lemire, O. Kaser, N. Kurz, L. Deri, C. O'Hara, F. Saint-Jacques, and G. Ssi-Yan-Kai. Roaring bitmaps: Implementation of an optimized software library. *Software: Practice and Experience*, 48(4):867–895, 2018.
[77] J. Leskovec et al. Kronecker graphs: An approach to modeling networks. *J. of Machine Learning Research*, 11(Feb):985–1042, 2010.
[78] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.
[79] B. Lessley, T. Perciano, M. Mathai, H. Childs, and E. W. Bethel. Maximal clique enumeration with data-parallel primitives. In *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 16–25. IEEE, 2017.
[80] D. Liben-Nowell and J. Kleinberg. The link-prediction problem for social networks. *Journal of the American society for information science and technology*, 58(7):1019–1031, 2007.
[81] H. Lin et al. Shentu: processing multi-trillion edge graphs on millions of cores in seconds. In *ACM/IEEE Supercomputing*. IEEE Press, 2018.
[82] L. Lu, Y. Gu, and R. Grossman. dmaximalcliques: A distributed algorithm for enumerating all maximal cliques and maximal clique distribution. In *2010 IEEE International Conference on Data Mining Workshops*, pages 1320–1327. IEEE, 2010.
[83] L. Lü and T. Zhou. Link prediction in complex networks: A survey. *Physica A: statistical mechanics and its applications*, 390(6):1150–1170, 2011.
[84] K. Makino and T. Uno. New algorithms for enumerating all maximal cliques. In *Scandinavian workshop on algorithm theory*, pages 260–272. Springer, 2004.
[85] G. Manoussakis. An output sensitive algorithm for maximal clique enumeration in sparse graphs. In *12th International Symposium on Parameterized and Exact Computation (IPEC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.
[86] D. W. Matula and L. L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *JACM*, 1983.
[87] D. Mawhirter, S. Reinehr, C. Holmes, T. Liu, and B. Wu. Graphzero: Breaking symmetry for efficient graph mining. *arXiv preprint arXiv:1911.12877*, 2019.
[88] D. Mawhirter and B. Wu. Automine: harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 509–523. ACM, 2019.
[89] C. McCreesh and P. Prosser. A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs. In *CP*. Springer, 2015.
[90] G. L. Miller et al. Improved parallel algorithms for spanners and hopsets. In *ACM SPAA*. ACM, 2015.
[91] Z. Ming, C. Luo, W. Gao, R. Han, Q. Yang, L. Wang, and J. Zhan. Bdgs: A scalable big data generator suite in big data benchmarking. In *Advancing Big Data Benchmarks*, pages 138–154. Springer, 2013.
[92] P. J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710, 1999.
[93] R. C. Murphy et al. Introducing the graph 500. *Cray User's Group (CUG)*, 2010.
[94] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin. Graphbig: understanding graph computing in the context of industrial solutions. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.
[95] T. J. Ottosen and J. Vomlel. Honour thy neighbour—clique maintenance in dynamic graphs. *on Probabilistic Graphical Models*, page 201, 2010.
[96] S. Parthasarathy, S. Tatikonda, and D. Ucar. A survey of graph mining techniques for biological datasets. In *Managing and mining graph data*, pages 547–580. Springer, 2010.
[97] U. N. Raghavan et al. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E*, 76(3):036106, 2007.
[98] T. Ramraj and R. Prabhakar. Frequent subgraph mining algorithms-a survey. *Procedia Computer Science*, 47:197–204, 2015.
[99] S. U. Rehman, A. U. Khan, and S. Fong. Graph mining: A survey of graph mining techniques. In *Seventh International Conference on Digital Information Management (ICDIM 2012)*, pages 88–92. IEEE, 2012.
[100] P. Ribeiro, P. Paredes, M. E. Silva, D. Aparicio, and F. Silva. A survey on subgraph counting: Concepts, algorithms and applications to network motifs and graphlets. *arXiv preprint arXiv:1910.13011*, 2019.
[101] I. Robinson, J. Webber, and E. Eifrem. *Graph databases*. " O'Reilly Media, Inc.", 2013.
[102] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.
[103] S. E. Schaeffer. Graph clustering. *Computer science review*, 1(1):27–64, 2007.
[104] T. Schank. Algorithmic aspects of triangle-based network analysis. *Phd in computer science, University Karlsruhe*, 3, 2007.
[105] M. C. Schmidt, N. F. Samatova, K. Thomas, and B.-H. Park. A scalable, parallel algorithm for maximal clique enumeration. *Journal of Parallel and Distributed Computing*, 69(4):417–428, 2009.
[106] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Notices*, volume 48, pages 135–146, 2013.
[107] J. Shun and K. Tangwongsan. Multicore triangle computations without tuning. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 149–160. IEEE, 2015.
[108] C. L. Staudt and H. Meyerhenke. Engineering parallel algorithms for community detection in massive networks. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):171–184, 2015.
[109] V. Stix. Finding all maximal cliques in dynamic graphs. *Computational Optimization and applications*, 27(2):173–186, 2004.
[110] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.
[111] M. Svendsen, A. P. Mukherjee, and S. Tirthapura. Mining maximal cliques from a large graph using mapreduce: Tackling highly uneven subproblem sizes. *Journal of Parallel and distributed computing*, 79:104–114, 2015.
[112] L. Tang and H. Liu. Graph mining applications to social network analysis. In *Managing and Mining Graph Data*, pages 487–513. Springer, 2010.
[113] Y. Tang. Benchmarking graph databases with cyclone benchmark. 2016.
[114] B. Taskar et al. Link prediction in relational data. In *NIPS*, pages 659–666, 2004.
[115] C. H. Teixeira et al. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 425–440. ACM, 2015.
[116] L. T. Thomas, S. R. Valluri, and K. Karlapalem. Margin: Maximal frequent subgraph mining. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 4(3):1–42, 2010.
[117] E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.*, 363(1):28–42, 2006.
[118] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM Journal on Computing*, 6(3):505–517, 1977.
[119] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.
[120] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu. Rstream: marrying relational algebra with streaming for efficient graph mining on a single machine. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 763–782, 2018.
[121] L. Wang, K. Hu, and Y. Tang. Robustness of link-prediction algorithm based on similarity and application to biological networks. *Current Bioinformatics*, 9(3):246–252, 2014.
[122] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, et al. Bigdatabench: A big data benchmark suite from internet services. In *2014 IEEE 20th international symposium on high performance computer architecture (HPCA)*, pages 488–499. IEEE, 2014.
[123] T. Washio and H. Motoda. State of the art of graph-based data mining. *Acm Sigkdd Explorations Newsletter*, 5(1):59–68, 2003.
[124] B. Wu, S. Yang, H. Zhao, and B. Wang. A distributed algorithm to enumerate all maximal cliques in mapreduce. In *2009 Fourth International Conference on Frontier of Computer Science and Technology*, pages 45–51. IEEE, 2009.
[125] Y. Xu, J. Cheng, and A. W.-C. Fu. Distributed maximal clique computation and management. *IEEE Transactions on Services Computing*, 9(1):110–122, 2015.
[126] Z. Xu, X. Chen, J. Shen, Y. Zhang, C. Chen, and C. Yang. Gardenia: A graph processing benchmark suite for next-generation accelerators. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 15(1):1–13, 2019.
[127] D. Yan, H. Chen, J. Cheng, M. T. Özsu, Q. Zhang, and J. Lui. G-thinker: big graph mining made easier and faster. *arXiv preprint arXiv:1709.03110*, 2017.
[128] D. Yan, W. Qu, G. Guo, and X. Wang. Prefixfpm: A parallel framework for general-purpose frequent pattern mining. In *Proceedings of the 36th IEEE International Conference on Data Engineering (ICDE) 2020*, 2020.
[129] P. Yao et al. A locality-aware energy-efficient accelerator for graph mining applications. In *IEEE/ACM MICRO*, pages 895–907. IEEE, 2020.
[130] Y. Zhang et al. Genome-scale computational approaches to memory-intensive applications in systems biology. In *ACM/IEEE Supercomputing*, pages 12–12. IEEE, 2005.
[131] C. Zhao, Z. Zhang, P. Xu, T. Zheng, and X. Cheng. Kaleido: An efficient out-of-core graph mining system on a single machine. *arXiv preprint arXiv:1905.09572*, 2019.