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

Maciej Besta¹*, Zur Vonarburg-Shmaria¹, Yannick Schaffner¹, Leonardo Schwarz¹, Grzegorz Kwasniewski¹, Lukas Gianinazzi¹, Jakub Beranek², Kacper Janda³, Tobias Holenstein¹, Sebastian Leisinger¹, Peter Tatkovski¹, Esref Ozdemir¹, Adrian Balla¹, Marcin Copik¹, Philipp Lindenberger¹, Marek Konieczny³, Onur Mutlu¹, Torsten Hoefer¹∗

¹ETH Zurich, Zurich, Switzerland; ²VSB, Ostrava, Czech Republic; ³AGH-UST, Krakow, Poland; *Corresponding authors bestam@inf.ethz.ch, zur.shmaria@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.

1 INTRODUCTION AND MOTIVATION

Graph mining is used in social sciences, bioinformatics, chemistry, medicine, cybersecurity, and many others [31, 39, 66, 66]. Yet, graphs can reach one trillion edges (the Facebook graph [2019] [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 an example GMS use case: accelerating the Bron-Kerbosch algorithm for maximal clique listing

Selecting relevant baselines & input graphs (enabled by the GMS benchmark specification)

Experimenting with different algorithmic parts (facilitated by the GMS benchmarking platform)

Insightful evaluation (facilitated by the GMS metrics, such as algorithms throughput).


doi:10.14778/3476249.3476252

PVLDB Reference Format:
doi:10.14778/3476249.3476252

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://graphminesuite.spc1.inf.ethz.ch/.
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. Moreover, GMS offers a novel performance metric and a broad theoretical concurrency analysis for deeper performance insights beyond simple empirical run-times. To maximize GMS’s usability, we arm it with an accompanying software platform, with reference implementations of algorithms. 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 that can be used as a reference point when selecting relevant comparison targets.

Second, GMS comes with a benchmarking platform: 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 of graph mining algorithms, as well as representations, data layouts, and compression schemes.

For more insightful performance analyses, we propose a novel performance metric 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 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×, up to 1.1×, >2×, and 2.5×, respectively. We also improve theoretical bounds: for example, for maximal clique listing, we obtain O(dms(2e+3)/d)/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].

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 Perigrine [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 run-time 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 ⊆ V × 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 Δ(v), respectively.

3 OVERVIEW OF GMS We start with an overview; see Figure 2.
The GMS benchmark specification (details in Section 4) motivates representative graph mining problems and state-of-the-art algorithms solving these problems, relevant datasets, performance metrics, 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 (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, optional preprocessing, running different graph mining algorithms, and gathering data.

The reference implementation of algorithms (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 (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.  

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
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.

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, 104] (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 above-mentioned 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,
in § 8.6, we describe two graphs with almost identical sizes, sparsity, 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 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] and a block [2]. 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, LogiGraph [17], and others.

Second, the user can seamlessly add preprocessing routines \(^3\) 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 \(^4\). GMS also facilitates modifying fine parts of an algorithm \(^5\), 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, 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 \(^5\)). 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 \(\cap\) in which only one set element is inserted or removed from a set; these methods always modify the calling set.

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 c1one, 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, \(A \cup B\) 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 hashtab. 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.

```java
1 template <class TSet>  
2 class SetGraph (  
3   public:  
4   using Set = TSet;  
5   const int64_t num_nodes() 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 (\(\mathcal{M}_1\) - \(\mathcal{M}_5\)). 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 (\(\mathcal{M}_3\)). 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 \((1 +\varepsilon)\delta_G\) (cf. set \(R\) and Line 7). The parameter \(\varepsilon \geq 0\) controls the accuracy of the approximation; \(\delta_G\) 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 (\(\mathcal{M}_3\)).

**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] works 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 (\(\mathcal{M}_3\)). 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 \(P, X, R\). 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 (\(\mathcal{M}_5\)). 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 9x.

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 $P = N(v_i) \cap \{v_i, \ldots, v_n\}$ and $X = N(v_i) \cap \{v_1, \ldots, v_{i-1}\}$. 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 $\{v_1, \ldots, v_{i-1}\}$ in $P$, and thus we can limit $P$ to $N(v_i) \cap \{v_1, \ldots, v_{i-1}\}$ (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 (3).

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 roaming 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 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 thereby the obtained order assigns 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 $v$ with $N^+(v)$. 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 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 enhancement2. 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 $\Delta^2$. When parallelizing over edges, this significantly reduces the required memory (for large maximum degrees $\Delta$, even up to >90%).

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

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1. Input: A graph $G$, $k \in \mathbb{N}$ Output: Count of $k$-cliques $c_k \in \mathbb{N}$. */</td>
</tr>
<tr>
<td>3</td>
<td>2. // Preprocessing: reorder vertices with DSR or ADG.</td>
</tr>
<tr>
<td>4</td>
<td>3. // New candidates for the recursive search</td>
</tr>
<tr>
<td>5</td>
<td>4. // First, remove unnecessary vertices from $P$ (candidates to be included in a clique) and $X$ (vertices definitely not in a clique) by intersecting $N(v)$ with vertices that follow and precede $v$ in the applied order.</td>
</tr>
<tr>
<td>6</td>
<td>5. $P = N(v) \cap (v_1, v_2, \ldots, v_{i-1})$</td>
</tr>
<tr>
<td>7</td>
<td>6. $X = N(v) \cap (v_1, v_2, \ldots, v_{i-1})$</td>
</tr>
<tr>
<td>8</td>
<td>7. $R = {v_i}$</td>
</tr>
<tr>
<td>9</td>
<td>8. Run the Bron-Kerbosch routine recursively for $P$ and $X$</td>
</tr>
<tr>
<td>10</td>
<td>9. BK-Pivot($P$, $X$, $R$)</td>
</tr>
<tr>
<td>11</td>
<td>Algorithm 4: Enumeration of maximal cliques, a Bron-Kerbosch variant by Epstein et al. [52] with GMS enhancements.</td>
</tr>
</tbody>
</table>

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

1 We used “compiler explorer” (https://godbolt.org/) for assembly analysis.
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 \( \Theta(n d^2) \), 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 \( \Theta(n + k(d/2)^{k-1} + d^2) \), but the space increases by a factor of \( \frac{2}{K} \) to \( O(m d^2) \). 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 \( k d^{k-2} \). 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.
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.

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 Epstein’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 Epstein 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 comparatively 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 show 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 Idoro (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 $C_2$) 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 $\varepsilon$ is, the more (mild) speedup is observed. This is because larger $\varepsilon$ 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].

8.5 Faster Subgraph Isomorphism

GMS enabled us to accelerate a very recent parallel VF3-Light subgraph isomorphism baseline by $2.5\times$. 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 $1.1\times$ 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].
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.


