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

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

Review by Kliment Serafimov
for 6.827
The GraphMiningSuite is a general and extensible framework for end-to-end development of high-performance graph algorithms.

Key features:

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

Also:

- Extensive literature review and comparison to other frameworks.
Research Problem Pipeline

High-Performance Graph Mining

Goal: construct a high-performance algorithm solving a selected graph mining problem

Different symbols indicate which elements of GMS are responsible for a given part of the construction process of a graph mining algorithm

Part 1: Design
Key questions:
S: What are relevant mining algorithms and datasets?
C: How to assess the scalability of a new algorithmic idea?

Part 2: Implementation & tuning
Key questions:
I: How to quickly benchmark new parallel graph mining algorithms, preprocessing schemes, data layouts, various optimizations?
P: How to effectively use different parallel architectures?

Part 3: Analysis
Key questions:
S: What are state-of-the-art comparison baselines?
P: How to analyze the performance, storage requirements, and other aspects of a new algorithm?
C: How to effectively evaluate algorithms?

Part 4: Evaluation
Key questions:
M: What are insightful performance metrics for graph mining?
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, 9, 10). “P.? (Preprocessing)” indicates if a given algorithm can be seamlessly used as a preprocessing routine; in the current GMS version, this feature is reserved for vertex reordering.
Graph Datasets

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

<table>
<thead>
<tr>
<th>Graph</th>
<th>$n$</th>
<th>$m$</th>
<th>$m/n$</th>
<th>$d_i$</th>
<th>$d_o$</th>
<th>$T$</th>
<th>$T/n$</th>
<th>Why selected/special?</th>
</tr>
</thead>
<tbody>
<tr>
<td>(so)</td>
<td>Korkut</td>
<td>3M : 177M</td>
<td>38.1 : 33.3 : 33.3</td>
<td>628M : 204.3</td>
<td>Large $m/n$, relatively large</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(so)</td>
<td>Flickr</td>
<td>2.3M : 22.8M</td>
<td>9.9 : 21k : 26.3k</td>
<td>838M : 363.7</td>
<td>Large $T$ but low $m/n$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(so)</td>
<td>Libimseti</td>
<td>221k : 17.2M</td>
<td>78.33k : 25k</td>
<td>69M : 312.8</td>
<td>Large $m/n$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(so)</td>
<td>Youtube</td>
<td>3.2M : 9.3M</td>
<td>2.9 : 91.7k : 91.7k</td>
<td>12.2M : 3.8</td>
<td>Very low $m/n$ and $T$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(so)</td>
<td>Flickr</td>
<td>2.3M : 7.91M</td>
<td>3.1 : 14k</td>
<td>819M : 3.1</td>
<td>Very low $m/n$ and $T$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(so)</td>
<td>Livemocha</td>
<td>104k : 2.19M</td>
<td>21.1 : 298k : 298k</td>
<td>336M : 3.23</td>
<td>a lot fewer 4-cliques (43.6M)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(so)</td>
<td>Ep-trust</td>
<td>132k : 841k</td>
<td>6.3k : 6.3k</td>
<td>27.9M : 121</td>
<td>Huge $T$-skew ($\bar{T} = 108k$)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(so)</td>
<td>FB comm.</td>
<td>35.1k : 1.5M</td>
<td>41.5k : 8.2k</td>
<td>36.4M : 1k</td>
<td>$T$-skew ($\bar{T} = 159k$)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(wb)</td>
<td>DBpedia</td>
<td>12.1M : 268M</td>
<td>23.7 : 636k : 693k</td>
<td>11.6M : 961.8</td>
<td>Rather low $m/n$ but high $T$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(wb)</td>
<td>Wikipedia</td>
<td>18.2M : 127M</td>
<td>6.9 : 632k : 328M</td>
<td>18.0 : Very common, very sparse</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(wb)</td>
<td>Baidu</td>
<td>2.14M : 17M</td>
<td>7.9 : 79.9k</td>
<td>2.5M : 11.8</td>
<td>Very sparse</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(wb)</td>
<td>WikiEdit</td>
<td>94.3k : 5.7M</td>
<td>60.4 : 107k</td>
<td>107k : 835M</td>
<td>Large $T$-skew ($\bar{T} = 15.7M$)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(st)</td>
<td>Chebyshev4</td>
<td>68.1k : 5.3M</td>
<td>77.8 : 68.1k : 68.1k</td>
<td>445M : 6.5k</td>
<td>Very large $T$ and $T/n$ and $T$-skew ($\bar{T} = 5.8M$)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(st)</td>
<td>Gearbox</td>
<td>154k : 4.5M</td>
<td>29.2 : 98</td>
<td>94M : 915k</td>
<td>Low $\bar{d}$ but large $T$; low $T$-skew ($\bar{T} = 1.7k$)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(st)</td>
<td>Nemeth25</td>
<td>10k : 751k</td>
<td>75.1 : 192</td>
<td>192 : 87M</td>
<td>9k Huge $T$ but low $\bar{T} = 12k$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(st)</td>
<td>F2</td>
<td>71.5k : 2.6M</td>
<td>36.5 : 344</td>
<td>344 : 110M</td>
<td>1.5k $T$-skew ($\bar{T} = 9.6k$)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sc)</td>
<td>Gupt3A</td>
<td>16.8k : 4.7M</td>
<td>280 : 14.7k : 14.7k</td>
<td>696M : 41.5k</td>
<td>Huge $T$-skew ($\bar{T} = 1.5M$)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sc)</td>
<td>Idoor</td>
<td>95.2k : 20.8M</td>
<td>21.5 : 76</td>
<td>76 : 567M</td>
<td>395 $T$-skew ($\bar{T} = 1.1k$)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(re)</td>
<td>MovieRec</td>
<td>70.2k : 10M</td>
<td>34.3k : 35.3k</td>
<td>983M : 14k</td>
<td>Huge $T$ and $\bar{T} = 4.9M$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(re)</td>
<td>RecDate</td>
<td>169k : 17.4M</td>
<td>102.5k : 33.4k</td>
<td>33.4k : 286M</td>
<td>1.7k $T$-skew ($\bar{T} = 1.6M$)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(bi)</td>
<td>sc-hd (gene)</td>
<td>2.1k</td>
<td>63k</td>
<td>30</td>
<td>472</td>
<td>472</td>
<td>4.2</td>
<td>2k</td>
</tr>
<tr>
<td>(bi)</td>
<td>AntColony6</td>
<td>164</td>
<td>10.3k</td>
<td>62.8</td>
<td>157</td>
<td>157</td>
<td>1.1</td>
<td>6.9</td>
</tr>
<tr>
<td>(bi)</td>
<td>AntColony5</td>
<td>152</td>
<td>9.1k</td>
<td>59.8</td>
<td>150</td>
<td>150</td>
<td>897</td>
<td>5.9</td>
</tr>
<tr>
<td>(co)</td>
<td>Jestor2</td>
<td>50.7k</td>
<td>1.7M</td>
<td>33.5</td>
<td>50.8k</td>
<td>50.8k</td>
<td>127</td>
<td>2.5k</td>
</tr>
<tr>
<td>(co)</td>
<td>Flickr</td>
<td>(photo relations)</td>
<td>106k</td>
<td>2.31M</td>
<td>21.9</td>
<td>5.4k</td>
<td>5.4k</td>
<td>108M</td>
</tr>
<tr>
<td>(ee)</td>
<td>mbeatcc</td>
<td>492</td>
<td>49.5k</td>
<td>100.5</td>
<td>679</td>
<td>679</td>
<td>9M</td>
<td>18.2k</td>
</tr>
<tr>
<td>(ee)</td>
<td>oransit8</td>
<td>2.5k</td>
<td>89.9k</td>
<td>35.5</td>
<td>1.7k</td>
<td>1.7k</td>
<td>8.7M</td>
<td>3.4k</td>
</tr>
<tr>
<td>(re)</td>
<td>USA roads</td>
<td>23.9M</td>
<td>28.8M</td>
<td>1.2</td>
<td>9</td>
<td>9</td>
<td>1.3</td>
<td>0.1</td>
</tr>
</tbody>
</table>

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, [ee]: economics network, [ro]: road graph. Structural features: $m/n$: graph sparsity, $d_i$: maximum in-degree, $d_o$: 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, $\bar{T}$ is the maximum number of triangles per vertex in a given graph. Dataset: (W), (S), (K), (D), (C), and (N) refer to the publicly available datasets, explained in §8.1. For more details, see §4.2.
Metrics

- Seamless integration with PAPI (for extracting hardware use stats)
  - CPU Core utilization (stalled CPU cycles).
  - Cache misses and cache hits (L1, L2, L3, data vs. instruction, TLB)
  - Memory reads/writes

- Algorithmic efficiency / algorithmic throughput
  - Generalization of ‘edges-per-second’.
  - Graphlets-per-second. Number of graph motives mined per second
  - Eg: cliques per second, clusters per second etc.
Literature Review and Comparison

<table>
<thead>
<tr>
<th>Reference / Infrastructure</th>
<th>Pattern Matching</th>
<th>Learning</th>
<th>Vr</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>[B] Cyclone [113]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[B] GBBS/Ligra [46, 106]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[B] GraphBIG [94]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[B] GAPBS [13]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[B] LDDBC [23]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[B] WGB [9]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[B] PPBS [19]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[B] GraphSDB [93]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[B] CRONO [6]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[B] GARDENIA [126]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[F] A framework [47]  

| [B] GMS [This paper]       |                  |          |    |         |

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 [109]). mC: maximal clique listing, kC: k-clique listing, dS: densest subgraph, sl: subgraph isomorphism, fS: frequent subgraph mining, vS: vertex similarity, IP: link prediction, cl: clustering, cD: community detection, Opt: optimization, Vr: vertex rankings, : Supported. : Partial support. : no support.

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? (3): 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 (8 - 13) 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 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? : Supported. : Partial support. / / /: A given metric is supported via an external profiler. : No support.
Primitives and Interfaces

- Sets and set-algebra primitives (right ->)
  - Union, intersection, difference
  - Contains, cardinality, iteration, equality
  - remove, add
  - Cloning, serialization

- Graph representation and access
  - Allows the user to implement their own

- Graph preprocessing
  - Allows the user to implement their own

- Algorithms
  - User can use existing algorithms
  - Implement their own algorithms
  - Or tweak existing algorithms

Implementation of set algebra

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

```
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    void union_inplace(const Set &B); // Update A = A ∪ B
15    void union_inplace(SetElement b); // Update A = A ∪ {b}
16    bool contains(SetElement b) const; // Return b ∈ A ? true: false
17    void add(SetElement b); // Update A = A ∪ {b}
18    void remove(SetElement b); // Update A = A \ {b}
19    size_t cardinality() const; // Return set's cardinality
20    // (2) Constructors (selected):
21    Set(const SetElement *start, size_t count); // From an array
22    Set(); Set(Set &S); // Default and Move constructors
23    Set(SetElement); // Constructor of a single-element set
24    static Set Range(int bound); // Create set {0, 1, ..., bound - 1}
25    // (3) Other methods:
26    begin() const; // Return iterators to set's start
27    end() const; // Return iterators to set's end
28    Set clone() const; // Return a copy of the set
29    void toArray(int32_t *array) const; // Convert set to array
30    operator==; operator!=; // Set equality/inequality comparison
31    // (4) Define a set element
32    private:
33    using SetElement = GMS::NodeId;
34  }
```

Algorithm 1: The set algebra interface provided by GMS.
Work-span analysis

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Work-span analysis</strong></td>
<td></td>
</tr>
<tr>
<td>$k$-Clique Listing</td>
<td></td>
</tr>
<tr>
<td>Node Parallel [41]</td>
<td>$O\left(\frac{mk}{d}^{k-2}\right)$</td>
</tr>
<tr>
<td>Edge Parallel [41]</td>
<td>$O\left(\frac{mk}{d}^{k-2}\right)$</td>
</tr>
<tr>
<td>Depth</td>
<td>$O\left(\frac{mk}{d}^{k-2}\right)$</td>
</tr>
<tr>
<td>Space</td>
<td>$O\left(mk^{2}+K\right)$</td>
</tr>
</tbody>
</table>

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

- Approximate degeneracy order
- Max-clique
- K-clique
Approximate Degeneracy Order

1 //Input: A graph $G$. Output: Approx. degeneracy order (ADG) $\eta$.
2 $i = 1$ // Iteration counter
3 $U = V$ // $U$ is the induced subgraph used in each iteration $i$
4 while $U \neq \emptyset$ do:
5    $\overline{\delta_U} = \left( \sum_{v \in U} |N_U(v)| \right) / |U|$ //Get the average degree in $U$
6    // $R$ contains vertices assigned priority in this iteration:
7    $R = \{ v \in U : |N_U(v)| \leq (1 + \epsilon)\overline{\delta_U} \}$
8    for $v \in R$ in parallel do: $\eta(v) = i$ //assign the ADG order
9    $U = U \setminus R$ //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.
Enumerating Cliques

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

//Preprocessing: reorder vertices with DGR or ADG.
4 //Here, we also record the actual ordering and denote it as $\eta$
5 $(v_1, v_2, \ldots, v_n; \eta) = \text{preprocess}(V, \text{/* selected vertex order */})$

7 //Construct a directed version of $G$ using $\eta$. This is an
8 //additional optimization to reduce the search space:
9 $G = \text{dir}(G)$ //An edge goes from $v$ to $u$ iff $\eta(v) < \eta(u)$
10 $c_k = 0$ //We start with zero counted cliques.
11 for $u \in V$ in parallel do: //Count $u$'s neighboring $k$-cliques
12 $C_2 = N^+(u); c_k += \text{count}(2, G, C_2)$
14 function count($i, G, C_i$):
15 if ($i == k$): return $|C_k|$ //Count $k$-cliques
16 else:
17 $c_i = 0$
18 for $v \in C_i$ do: //search within neighborhood of $v$
19 $C_{i+1} = N^+(v) \cap C_i$ // $C_i$ counts $i$-cliques.
20 $c_i += \text{count}(i+1, G, C_{i+1})$
21 return $c_i$

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

//Main part: conduct the actual clique enumeration.
7 for $v_i \in (v_1, v_2, \ldots, v_n)$ 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 belonging in a clique) by intersecting $N(v_i)$ with vertices
12 //that follow and precede $v_i$ in the applied order.
13 $P = N(v_i) \cap \{v_{i+1}, \ldots, v_n\} \subseteq \{v_i\}$; $X = N(v_i) \cap \{v_1, \ldots, v_{i-1}\} \subseteq \{v_i\}$; $R = \{v_i\}$
14 //Run the Bron-Kerbosch routine recursively for $P$ and $X$.
15 BK-Pivot($P$, $\{v_i\}$, $X$)
18 BK-Pivot($P, R, X$) //Definition of the recursive BK scheme
19 if $P \cup X \neq 0$:
20 $u = \text{pivot}(P \cup X)$ //Choose a "pivot" vertex $u \in P \cup X$
21 for $v \in P \setminus N(u)$ do: //Use the pivot to prune search space
22 //New candidates for the recursive search
23 $P_{\text{new}} = P \cap N(v) \subseteq \{v\}$; $X_{\text{new}} = X \cap N(v) \subseteq \{v\}$; $R_{\text{new}} = R \cup \{v\} \subseteq \{v\}$
24 //Search recursively for a maximal clique that contains $v$
25 BK-Pivot($P_{\text{new}}, R_{\text{new}}, X_{\text{new}}$)
26 //After the recursive call, update $P$ and $X$ to reflect
27 //the fact that $v$ was already considered
28 $P = P \setminus \{v\} \subseteq \{v\}$; $X = X \cup \{v\} \subseteq \{v\}$

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

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

Any questions?