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 Tatkowski¹, Esref Ozdemir¹, Adrian Balla¹, Marcin Copik¹, Philipp Lindenberger¹, Marek Konieczny³, Onur Mutlu¹, Torsten Hoefler¹*

¹ETH Zurich, Zurich, Switzerland; ²VSB, Ostrava, Czech Republic; ³AGH-UST, Krakow, Poland; *Corresponding authors
What is graph mining?

- Graph mining is the process of finding and extracting useful information from graphs, i.e. sssp, triangle counting, k-cliques, maximal cliques, etc.
- Many real world applications: social sciences, bioinformatics, chemistry, medicine, cybersecurity, and many others
- Issue #1: graphs can be very large and require a lot of compute power
- Solution #1: Parallelism!
- Issue #2: Too many choices!
  - Hard to keep up and find relevant baseline graph mining algorithms to improve upon, a plethora of relevant datasets, numerous design choices
- Solution #2: **GraphMineSuite (GMS)** - a benchmarking suite for high-performance graph mining algorithms.
GraphMine Suite

**Benchmark specification**

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

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

**Reference implementations**

**Details**: Section 5

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

**Implemented in**

**Benchmarking platform**

**Used by**

**Performance metrics**

**Details**: Sections 5 & 7

- Run-time
- Scalability
- L3 misses (machine efficiency)

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

**Concurrency analysis**

**Aspects**
- Performance (work, depth)
- Storage
- Tradeoffs

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

1. **Build graph representation**
   - (CSR by default)
   - A representation is modular
   - Neighborhoods of vertices

2. **Define graph accesses**
   - When developing a graph representation, the user also develops the corresponding graph accesses.
   - Check (v)
   - Check if v < (u, v)

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

4. **Run graph algorithm**
   - */ Example: Triangle Counting. "tc" is the count of triangles */
   - for v in V:
     - tc = 0; init_sets()
     - #pragma omp parallel for schedule (...
     - for w in N(v):
       - tc += |N(v) ∩ N(w)|
     - tc /= 3; cleanup()
   - The user can plug in variants of fine algorithm blocks such as scheduling policies. GMS facilitates it with appropriate modular implementations.

5. **Define algorithm building blocks**
   - The user can experiment with algorithmic ideas (e.g., new algorithms or data structures), architectural ideas (e.g., using SIMD or intrinsics), and design ideas (e.g., using novel form of load balancing).

**Clear structure of code facilitating manipulation with fine parts such as scheduling policy of single loops**

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

**GraphMines facilitate extensibility at a given stage?**

- Modular design of classes & files associated with graph representations
- Well-defined interface (based on set algebra) of routines for graph accesses
- Enabling running different preprocessing routines with a single function call
- Modular design of classes & files associated with graph algorithms
- Clear structure of code facilitating manipulation with fine parts such as scheduling policy of single loops

The user can experiment with algorithmic ideas (e.g., new algorithms or data structures), architectural ideas (e.g., using SIMD or intrinsics), and design ideas (e.g., using novel form of load balancing).
# Benchmark specification: Graph Problems

<table>
<thead>
<tr>
<th>Graph problem</th>
<th>Corresponding algorithms</th>
<th>E.?</th>
<th>P.?</th>
<th>Why included, what represents? (selected remarks)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximal Clique Listing [48]</td>
<td>Bron-Kerbosch [24] + optimizations (e.g., pivoting) [29, 51, 117]</td>
<td>✓ 5</td>
<td>✅ 8</td>
<td>Widely used, NP-complete, example of backtracking</td>
</tr>
<tr>
<td>$k$-Clique Listing [41]</td>
<td>Edge-Parallel and Vertex-Parallel general algorithms [41], different variants of Triangle Counting [104, 107]</td>
<td>✓ 5</td>
<td>✅ 8</td>
<td>P (high-degree polynomial), example of backtracking</td>
</tr>
<tr>
<td>Dense Subgraph Discovery [5]</td>
<td>Listing $k$-clique-stars [63] and $k$-cores [54] (exact &amp; approximate)</td>
<td>✓ 5</td>
<td>✅ 8</td>
<td>Different relaxations of clique mining</td>
</tr>
<tr>
<td>Subgraph isomorphism [48]</td>
<td>VF2 [40], TurboSO [58], Glasgow [89], VF3 [26, 28], VF3-Light [27]</td>
<td>✓ 5</td>
<td>✅ 8</td>
<td>Induced vs. non-induced, and backtracking vs. indexing schemes</td>
</tr>
<tr>
<td>Frequent Subgraph Mining [5]</td>
<td>BFS and DFS exploration strategies, different isomorphism kernels</td>
<td>✓ 5</td>
<td>✅ 8</td>
<td>Useful when one is interested in many different motifs</td>
</tr>
<tr>
<td>Vertex similarity [75]</td>
<td>Jaccard, Overlap, Adamic Adar, Resource Allocation, Common Neighbors, Preferential Attachment, Total Neighbors [101]</td>
<td>✓ 5</td>
<td>✅ 8</td>
<td>A building block of many more complex schemes, different methods have different performance properties</td>
</tr>
<tr>
<td>Link Prediction [114]</td>
<td>Variants based on vertex similarity (see above) [7, 80, 83, 114], a scheme for assessing link prediction accuracy [121]</td>
<td>✓ 5</td>
<td>✅ 8</td>
<td>A very common problem in social network analysis</td>
</tr>
<tr>
<td>Clustering [103]</td>
<td>Jarvis-Patrick clustering [65] based on different vertex similarity measures (see above) [7, 80, 83, 114]</td>
<td>✓ 5</td>
<td>✅ 8</td>
<td>A very common problem in general data mining; the selected scheme is an example of overlapping and single-level clustering</td>
</tr>
<tr>
<td>Community detection</td>
<td>Label Propagation and Louvain Method [108]</td>
<td>✓ 5</td>
<td>✅ 8</td>
<td>Examples of convergence-based on non-overlapping clustering</td>
</tr>
<tr>
<td>Degree reordering</td>
<td>A straightforward integer parallel sort</td>
<td>✓ 5</td>
<td>✅ 8</td>
<td>A simple scheme that was shown to bring speedups</td>
</tr>
<tr>
<td>Triangle count ranking</td>
<td>Computing triangle counts per vertex</td>
<td>✓ 5</td>
<td>✅ 8</td>
<td>Ranking vertices based on their clustering coefficient</td>
</tr>
<tr>
<td>Degeneracy reordering</td>
<td>Exact and approximate [54] [70]</td>
<td>✓ 5</td>
<td>✅ 8</td>
<td>Often used to accelerate Bron-Kerbosch and others</td>
</tr>
</tbody>
</table>

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 (✓ – ✓, ✓). “✓”: 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.
Set Algebra

- Many graph algorithms are/can be formulated with set algebra
- GMS allows users to implement their own sets, set operations, set elements, and set algebra based graph representations.
- Allows users to break complex graph mining algorithms into simple building blocks, and work on these building blocks independently.

```cpp
class Set {
  public:
  // In methods below, we denote "*this" pointer with A
  // (1) Set algebra methods:
  Set diff(const Set &B) const; // Return a new set C = A \ B
  Set diff(SetElement b) const; // Return a new set C = A \ {b}
  void diff_inplace(const Set &B); // Update A = A \ B
  void diff_inplace(SetElement b); // Update A = A \ {b}
  Set intersect(const Set &B) const; // Return a new set C = A ∩ B
  size_t intersect_count(const Set &B) const; // Return |A ∩ B|
  void intersect_inplace(const Set &B); // Update A = A ∩ B
  Set union(const Set &B) const; // Return a new set C = A ∪ B
  Set union(SetElement b) const; // Return a new set C = A ∪ {b}
  Set union_count(const Set &B) const; // Return |A ∪ B|
  void union_inplace(const Set &B); // Update A = A ∪ B
  void union_inplace(SetElement b); // Update A = A ∪ {b}
  bool contains(SetElement b) const; // Return b ∈ A
  bool remove(SetElement b); // Return A = A \ {b}
  size_t cardinality() const; // Return set's cardinality
  // (2) Constructors (selected):
  Set(const SetElement *start, size_t count); // From an array
  Set(); Set(Set &B); // Default and Move constructors
  Set(SETElement); // Constructor of a single-element set
  static Set Range(int bound); // Create set {0, 1, ..., bound - 1}
  // (3) Other methods:
  begin() const; // Return iterators to set's start
  end() const; // Return iterators to set's end
  Set clone() const; // Return a copy of the set
  void to_array(int32_t *array) const; // Convert set to array
  operator=; operator!=; // Set equality/inequality comparison

private:
  using SetElement = GMS::NodeId; // (4) Define a set element
};
```

Algorithm 1: The set algebra interface provided by GMS.
GMS Set Implementations

GMS offers three default set implementations:

- RoaringSet
  - Implemented with a “roaring bitmap” that allows for mild compression rates but inexpensive decompression
- SortedSet
- HashSet
Use Case # 1: Degeneracy Order & k-Cores

- The **degeneracy** of a graph $G$ is the smallest $d$ such that every subgraph in $G$ has a vertex of degree at most $d$.
  - A measure of graph sparsity

- 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.
  - DGR can be obtained by repeatedly removing a vertex of minimum degree in a graph.

- A **k-core** of $G$ is a maximal connected subgraph of $G$ whose all vertices have degree at least $k$.
  - A k-core can be obtained by iterating over vertices in the DGR order, and removing vertices with out-degree less than $k$. 
Use Case # 1: Degeneracy Order & k-Cores cont.

- **Issue**: Not easily parallelizable, $O(n)$ iterations!
- **Solution**: GMS offers a $(2+\epsilon)$-approximate degeneracy order (ADG), $O(\log n)$ iterations for any $\epsilon > 0$!

```
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
```
Use Case # 2: Maximal Clique Listing

- A maximal clique of a graph $G$ is a fully connected subgraph of $G$ that cannot be further extended by including one more adjacent vertex.

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

- A **k-Clique** of a graph $G$ is a fully connected subgraph of $G$ of $k$ vertices.

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

// Preprocessing: reorder vertices with DGR or ADG.
// Here, we also record the actual ordering and denote it as $\eta$
(v1, v2, ..., vn; $\eta$) = preprocess(V, /* selected vertex order */)

// Construct a directed version of $G$ using $\eta$. This is an additional optimization to reduce the search space:
G = dir(G) // An edge goes from $v$ to $u$ iff $\eta(v) < \eta(u)$
ck = 0 // We start with zero counted cliques.

for $u \in V$ in parallel do: // Count $u$'s neighboring $k$-cliques
    $C_2 = N^+(u)$; $ck += count(2, G, C_2)$

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

TAKEAWAY: GMS offers lots of modularity in implementing graph mining algorithms, specifically set algebra based modularity!
### Theoretical Analysis

- Obtained better bounds for maximal cliques
- Obtained similar bounds for k-clique, but scales better depending on graph

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Work $O\left( mk \left( \frac{d}{2} \right)^{k-2} \right)$</td>
<td>$O\left( mk \left( \frac{d}{2} \right)^{k-2} \right)$</td>
<td>$O\left( m \right)$</td>
<td>$O\left( d m^3 d^{3/3} \right)$</td>
<td>$O\left( \frac{n}{3} \right)$</td>
<td>$O\left( d m^3 (2 + e) d^{3/3} \right)$</td>
<td>$O\left( n \Delta^{k-1} \right)$</td>
<td>$O\left( m \Delta \right)$</td>
</tr>
<tr>
<td>Depth $O\left( n + k \left( \frac{d}{2} \right)^{k-1} \right)$</td>
<td>$O\left( n + k \left( \frac{d}{2} \right)^{k-1} + d^2 \right)$</td>
<td>$O\left( d \log n \right)$</td>
<td>$O\left( \log^2 n \right)$</td>
<td>$O\left( d \log n \right)$</td>
<td>$O\left( \log^2 n + d \log n \right)$</td>
<td>$O\left( \Delta^{k-1} \right)$</td>
<td>$O\left( \Delta \right)$</td>
</tr>
<tr>
<td>Space $O(m d^2 + K)$</td>
<td>$O\left( m d^2 + K \right)$</td>
<td>$O\left( m \right)$</td>
<td>$O\left( m + n d + K \right)$</td>
<td>$O\left( m + p d \Delta + K \right)$</td>
<td>$O\left( m + p d \Delta + K \right)$</td>
<td>$O\left( m + n k + K \right)$</td>
<td>$O\left( m \Delta \right)$</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.
Evaluation: Datasets, Methodology, Architectures

- GMS uses a wide selection of public datasets for flexibility
- GMS compares GMS variants to the most optimized state-of-the-art algorithms.
- GMS runs algorithms on the maximum number of cores available on machine.

<table>
<thead>
<tr>
<th>Graph</th>
<th>$n$</th>
<th>$m$</th>
<th>$m/n$</th>
<th>$d_i$ (max in-degree)</th>
<th>$d_o$ (max out-degree)</th>
<th>$T$ (skew)</th>
<th>$T/n$</th>
<th>Why selected/special?</th>
</tr>
</thead>
<tbody>
<tr>
<td>[so] (K) Orkut</td>
<td>3M</td>
<td>117M</td>
<td>38.1</td>
<td>33.3k</td>
<td>33.3k</td>
<td>628M</td>
<td>204.3</td>
<td>Common, relatively large</td>
</tr>
<tr>
<td>[so] (K) Flickr</td>
<td>2.3M</td>
<td>22.8M</td>
<td>9.9</td>
<td>21k</td>
<td>26.3k</td>
<td>838M</td>
<td>363.7</td>
<td>Large $T$ but low $m/n$.</td>
</tr>
<tr>
<td>[so] (K) Libimseti</td>
<td>221k</td>
<td>17.2M</td>
<td>78</td>
<td>31.3k</td>
<td>25k</td>
<td>69M</td>
<td>312.8</td>
<td>Large $m/n$.</td>
</tr>
<tr>
<td>[so] (K) Youtube</td>
<td>3.2M</td>
<td>9.3M</td>
<td>3.2</td>
<td>917k</td>
<td>917k</td>
<td>12.2M</td>
<td>3.8</td>
<td>Very low $m/n$ and $T$.</td>
</tr>
<tr>
<td>[so] (K) Flickrxer</td>
<td>2.5M</td>
<td>7.91M</td>
<td>3.1</td>
<td>1.4k</td>
<td>1.4k</td>
<td>7.9M</td>
<td>3.1</td>
<td>Very low $m/n$ and $T$.</td>
</tr>
<tr>
<td>[so] (K) Livemocha</td>
<td>104k</td>
<td>2.19M</td>
<td>21.1</td>
<td>298k</td>
<td>298k</td>
<td>3.36M</td>
<td>32.3</td>
<td>a lot fewer 4-cliques (4.36M).</td>
</tr>
<tr>
<td>[so] (N) Ep-trust</td>
<td>32k</td>
<td>841k</td>
<td>6</td>
<td>3.6k</td>
<td>3.6k</td>
<td>27.9M</td>
<td>212</td>
<td>Huge $T$-skew ($T = 108k$).</td>
</tr>
<tr>
<td>[so] (N) FB comm.</td>
<td>35.1k</td>
<td>1.5M</td>
<td>41.5</td>
<td>8.2k</td>
<td>8.2k</td>
<td>36.4M</td>
<td>1k</td>
<td>Large $T$-skew ($T = 159k$).</td>
</tr>
<tr>
<td>[wb] (K) DBpedia</td>
<td>12.1M</td>
<td>288M</td>
<td>23.7</td>
<td>963k</td>
<td>963k</td>
<td>11.68B</td>
<td>961.8</td>
<td>Rather low $m/n$ but high $T$.</td>
</tr>
<tr>
<td>[wb] (K) Wikipedia</td>
<td>18.2M</td>
<td>127M</td>
<td>6.9</td>
<td>632k</td>
<td>632k</td>
<td>328M</td>
<td>18.0</td>
<td>Common, very sparse.</td>
</tr>
<tr>
<td>[wb] (K) Baidu</td>
<td>2.1M</td>
<td>17M</td>
<td>7.9</td>
<td>97.9k</td>
<td>2.5k</td>
<td>25.2M</td>
<td>11.8</td>
<td>Very sparse.</td>
</tr>
<tr>
<td>[wb] (N) WikiEdit</td>
<td>94.9k</td>
<td>5M</td>
<td>60.4</td>
<td>107k</td>
<td>107k</td>
<td>835M</td>
<td>8.9k</td>
<td>Large $T$-skew ($T = 15.7M$). Very large $T$ and $T/n$.</td>
</tr>
<tr>
<td>[st] (N) Chebyshev4</td>
<td>68.1k</td>
<td>5.3M</td>
<td>77.8</td>
<td>68.1k</td>
<td>68.1k</td>
<td>445M</td>
<td>6.5k</td>
<td>and $T$-skew ($\tilde{T} = 5.8M$).</td>
</tr>
<tr>
<td>[st] (N) Gearbox</td>
<td>154k</td>
<td>5.5M</td>
<td>29.2</td>
<td>98</td>
<td>98</td>
<td>141M</td>
<td>915</td>
<td>Low $\tilde{d}$ but large $T$; low $T$-skew ($\tilde{T} = 1.7k$).</td>
</tr>
<tr>
<td>[st] (N) Nemeth25</td>
<td>10k</td>
<td>751k</td>
<td>75.1</td>
<td>192</td>
<td>192</td>
<td>87M</td>
<td>9k</td>
<td>Huge $T$ but low $\tilde{T} = 12k$.</td>
</tr>
<tr>
<td>[ct] (N) F2</td>
<td>71.5k</td>
<td>2.6M</td>
<td>26.5</td>
<td>344</td>
<td>344</td>
<td>110M</td>
<td>1.5k</td>
<td>Medium $T$-skew ($\tilde{T} = 9.6k$).</td>
</tr>
<tr>
<td>[sc] (N) Gupta3</td>
<td>16.8k</td>
<td>4.7M</td>
<td>280</td>
<td>14.7k</td>
<td>14.7k</td>
<td>696M</td>
<td>41.5k</td>
<td>Huge $T$-skew ($\tilde{T} = 1.5M$).</td>
</tr>
<tr>
<td>[sc] (N) Idoor</td>
<td>952k</td>
<td>20.8M</td>
<td>21.5</td>
<td>76</td>
<td>76</td>
<td>567M</td>
<td>59k</td>
<td>Very low $T$-skew ($\tilde{T} = 1.1k$).</td>
</tr>
<tr>
<td>[re] (N) MovieRec</td>
<td>70.2k</td>
<td>10M</td>
<td>142.4k</td>
<td>35.3k</td>
<td>35.3k</td>
<td>983M</td>
<td>41k</td>
<td>Huge $T$ and $\tilde{T} = 4.9M$.</td>
</tr>
<tr>
<td>[re] (N) RecDate</td>
<td>16.7k</td>
<td>17.4M</td>
<td>102.5k</td>
<td>33.4k</td>
<td>33.4k</td>
<td>286M</td>
<td>1.7k</td>
<td>Enormous $T$-skew ($\tilde{T} = 1.6M$).</td>
</tr>
<tr>
<td>[bi] (N) sc-hlt (gene)</td>
<td>2.1k</td>
<td>63k</td>
<td>30</td>
<td>472</td>
<td>472</td>
<td>4.2M</td>
<td>2k</td>
<td>Large $T$-skew ($\tilde{T} = 27.7k$).</td>
</tr>
<tr>
<td>[bi] (N) AntColony6</td>
<td>164</td>
<td>10.3k</td>
<td>62.8</td>
<td>157</td>
<td>157</td>
<td>1.1M</td>
<td>6.6k</td>
<td>Very low $T$-skew ($\tilde{T} = 9.7k$).</td>
</tr>
<tr>
<td>[bi] (N) AntColony5</td>
<td>152</td>
<td>9.1k</td>
<td>59.8</td>
<td>150</td>
<td>150</td>
<td>89k</td>
<td>5.9k</td>
<td>Very low $T$-skew ($\tilde{T} = 8.8k$).</td>
</tr>
<tr>
<td>[co] (N) Jester2</td>
<td>50.7k</td>
<td>1.7M</td>
<td>33.3</td>
<td>50.8k</td>
<td>50.8k</td>
<td>127M</td>
<td>2.5k</td>
<td>Enormous $T$-skew ($\tilde{T} = 2.3M$).</td>
</tr>
<tr>
<td>[co] (K) Flickr (photo relations)</td>
<td>106k</td>
<td>2.3M</td>
<td>21.9</td>
<td>5.4k</td>
<td>5.4k</td>
<td>108M</td>
<td>1019</td>
<td>Similar to Livemocha, but</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[ec] (N) mbeaxcc</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>
<td>Large $T$, low $\tilde{T} = 77.7k$.</td>
</tr>
<tr>
<td>[ec] (N) orani678</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>
<td>Large $T$, low $\tilde{T} = 80.8k$.</td>
</tr>
<tr>
<td>[ro] (D) USA roads</td>
<td>23.9M</td>
<td>28.8M</td>
<td>1.2</td>
<td>9</td>
<td>9</td>
<td>1.3M</td>
<td>0.1</td>
<td>Extremely low $m/n$ and $T$.</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, [ec]: economics network, [ro]: road graph. Structural features: $m/n$: graph sparsity, $d_i$ (max in-degree), $d_o$ (max 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, $\tilde{T}$ 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.
Results: Maximal Clique

Structural, scientific, various networks

Biological networks

Communication, recommendation networks

Road graph

Economics networks

Web graphs

Bron–Kerbosch (GMS code)  Fraction needed for reordering  Bron–Kerbosch by Das et al. (Bk-DAS, main competitor)  BK-GMS-DGR: BK with degeneracy reordering  BK-GMS-DEG: BK with simple degree reordering  BK-GMS-ADG: BK with approximate degeneracy reordering (proposed in this work)  BK-GMS-ADG-S: BK-GMS-ADG plus subgraph optimization (proposed in this work)
Results: Maximal Clique cont.

- Achieved a significant improvement in maximal clique using ADG or DGR.
- GMS variants often faster than main competitor by >50%, sometimes even >9x.
  - Consistent over graphs of different structural characteristics
- Another view (**algorithmic efficiency**): number of maximal cliques found per second
More Results

- Up to 10% speedup on k-clique algorithm with different parameters.
- ADG outperforms DGR as a preprocessor for Bron-Kerbosch.

Figure 4: Speedups of ADG for different $\epsilon$ over DEG/DGR, details in § 8.4. System: Ault.
Additional Analyses

- **Subtleties of higher-order structures**
  - Graphs similar in number of vertices, number of edges, sparsity, degree distribution etc., can have very different higher-order structures, such as number of 4-cliques. Choose datasets wisely!
- **Using synthetic graphs** can affect whether vertex reordering or mining dominates
Thank you! Any questions?