AutoMine
Harmonizing High-Level Abstraction and High Performance for Graph Mining

Daniel Mawhirter, Bo Wu
SOSP, October 30, 2019
• But the internet is big! (And so are other graph datasets)
Big Graphs

- 2 Billion Facebook users
- 3 Billion base pairs in human genome
- 20 Billion internet connected devices
- Trillions of connections between them

- Many graph processing systems are designed to optimize graph traversal problems
  - PowerGraph [OSDI’12], GraphChi [OSDI’12], GraphX [OSDI’14], X-Stream [SOSP’13]
  - Running BFS on Friendster in X-Stream takes 15s for just a linear-time traversal
Graph Mining

- Aims to discover *structural patterns* in a graph

- Examples:
  - Motif Counting finds all subgraphs of a given size
  - Frequent Subgraph Mining uses labels to further distinguish patterns

- Useful in anomaly/fraud detection, bioinformatics, large scale graph comparison
Triangle Counting

Pattern

Dataset
Triangle Counting

Pattern

Your Mutual Friend

You

Your Friend

Dataset
Triangle Counting is well-studied

COLORFUL TRIANGLE COUNTING AND A MAPREDUCE IMPLEMENTATION

RASMUS PAGH AND CHARALAMBOS Tsourakakis

Triangle Listing in Massive Networks

Shumo Chu
Nanyang Technological University, Singapore
shumo.chu@acm.org

ABSTRACT

Triangle listing is one of the fundamental algorithmic problems whose solution has numerous applications especially in the analysis of complex networks, such as the computation of clustering coefficient, transitivity, triangular connectivity, etc. Existing algorithms for triangle listing are mainly in-memory algorithms, whose performance cannot scale with the massive volume of today's fast growing networks. When the input graph cannot fit into main memory, they need to access disk several times, which can be very slow.

This paper studies I/O-efficient algorithms for the triangle listing problem and the triangle counting problem, whose solutions are basic operators in dealing with many other graph problems. In the former problem, given an undirected graph $G$, the objective is to find all the cliques involving 3 vertices in $G$. In the latter problem, the objective is to report just the number of such cliques, without having to enumerate them. Both problems have been well studied in internal memory, but still remain as difficult challenges when $G$ does not fit in memory, thus making it crucial to minimize the number of disk I/Os performed.

In particular, cycle of length 3 plays many important roles in the efficient computation of these tasks. In our work, we leverage this to design an efficient triangle listing algorithm that can work for any graph $G$.

The aforementioned triangle-centered measures have a large number of important applications. In addition, triangle listing also has

Counting Triangles in Real-World Networks using Projections

Xiaocheng Hu, Chinese University of Hong Kong
Yulei Tao, Chinese University of Hong Kong
Chin-Wan Chung, Korea Advanced Institute of Science and Technology

I/O-Efficient Algorithms on Triangle Listing and Counting
Triangle Counting is well-studied

**Scenarios**

**Notional Scenarios for the 2017 Hive Graph Challenge**

In this era of big data, the rates at which these data sets grow continue to accelerate. The ability to manage and analyze the largest data sets is always severely taxed. The most challenging of these data sets are those containing relational or network data. The Hive challenge is envisioned to be an annual challenge that will advance the state of the art in graph analytics on extremely large data sets. The primary focus of the challenges will be on the expansion and acceleration of graph analytic algorithms through improvements to algorithms and their implementations, and especially importantly, through special purpose hardware such as distributed and grid computers, and GPUs. Potential approaches to accelerate graph analytic algorithms include such methods as massively parallel computation, improvements to memory utilization, more efficient communications, and optimized data processing units.

The 2017 Hive challenge is composed of two challenges: the first focuses on subgraph isomorphism and the second on community detection. The baseline algorithms for the first challenge are recently developed algorithms that find triangles and k-trusses (J. Wang 2012). The triangle counting algorithms can be considered a special case of subgraph isomorphism where the subgraph of interest is restricted to a triangle. Although these algorithms do not match subgraphs of a general description, they can be used as components in algorithms that do. K-truss search algorithms can potentially support subgraph isomorphism algorithms through the characterization of a larger graph and a subgraph of interest. Inconsistent k-truss features means that an isomorphism does not exist between two subgraphs while consistent...
What about other patterns?
What about other patterns?

• Things get complicated quickly!
Prior General Mining Systems

• Arabesque[SOSP’15] and RStream[OSDI’18] are two state-of-the-art graph mining systems
• Idea: Enumerate the embeddings (i.e., subgraph instances) and run isomorphism tests
• Arabesque is a distributed system that implements an embedding-centric interface
• RStream runs on a single-machine and supports disk-streaming
for \( v_0 \) in \( V \):
  for \( v_1 \) in Adj(A):
    \( \text{tris} = \text{Adj}(v_0) \cap \text{Adj}(v_1) \)
    \( \text{count} += \text{tris}.\text{size}() \)

---

### Single Thread Comparison

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Single Threaded</th>
<th>Rstream</th>
</tr>
</thead>
<tbody>
<tr>
<td>MiCo</td>
<td>0.97</td>
<td>2.5</td>
</tr>
<tr>
<td>Patents</td>
<td>6.2</td>
<td>9.6</td>
</tr>
<tr>
<td>LiveJournal-1</td>
<td>146</td>
<td>106</td>
</tr>
<tr>
<td>Youtube</td>
<td>4.2</td>
<td>23.9</td>
</tr>
<tr>
<td>LiveJournal-2</td>
<td>34.7</td>
<td></td>
</tr>
</tbody>
</table>
AutoMine

- First of its kind topological compiler for graph mining
- Automates the manual algorithm design process

\[
\begin{align*}
A & \in V \\
B_A & \in \text{Adj}(A) \\
C_{AB} & \in \text{Adj}(A) \cap \text{Adj}(B) \\
\text{instances}[\text{clique}_4] & += D_{ABC} \\
\text{instances}[\text{rectangle}] & += D_{BC}
\end{align*}
\]
Techniques

- Set Modeling
- Vertex M
- $\text{Adjacent}(M)$
- $R \in \text{Adj}(M)$
Techniques

- Set Operations
- Begin from vertex A
- Discover vertices B-D
- Insert missing edges to encode all relationships
- Intersection (\(\cap\)) and Difference (\(\setminus\)) are sufficient, proof in paper
Techniques

• Scheduling space (permutations)
• Different orders imply different order of operations
• All are correct, just with different performance implications
• Choice of order is described in the paper
AutoMine

• First of its kind topological compiler for graph mining
• Automates the manual algorithm design process

Dataset

\[
A \in V \\
B_A \in \text{Adj}(A) \\
C_{AB} \in \text{Adj}(A) \cap \text{Adj}(B) \\
\ldots \\
\text{instances}[\text{clique}_4] += D_{ABC} \\
\text{instances}[\text{rectangle}] += D_{BC}
\]

Results!
Map to low-level code

\[ A \in \mathbf{V} \]
\[ B_A \in Adj(A) \]
\[ C_{AB} \in Adj(A) \cap Adj(B_A) \]
\[ D_{ABC} \in Adj(A) \cap Adj(B_A) \cap Adj(C_{AB}) \]
\[ \text{instances}[\text{clique}_4] += D_{ABC} \]

\[ A \in \mathbf{V} \]
\[ B_A \in Adj(A) \]
\[ C_{AB} \in Adj(A) \cap Adj(B_A) \]
\[ D_{BC} \in Adj(B_A) \cap Adj(C_{AB}) - Adj(A) \]
\[ \text{instances}[\text{chordal}] += D_{BC} \]
\[ A \in V \]
\[ B_A \in \text{Adj}(A) \]
\[ C_{AB} \in \text{Adj}(A) \cap \text{Adj}(B_A) \]
\[ D_{ABC} \in C_{AB} \cap \text{Adj}(C_{AB}) \]
\[ \text{instances[clique_4]} \text{ } += \text{ } D_{ABC} \]

\[ A \in V \]
\[ B_A \in \text{Adj}(A) \]
\[ C_{AB} \in \text{Adj}(A) \cap \text{Adj}(B_A) \]
\[ C_B \in \text{Adj}(B_A) \cap \text{Adj}(A) \]
\[ D_{BC} \in C_B \cap \text{Adj}(C_{AB}) \]
\[ \text{instances[chordal]} \text{ } += \text{ } D_{BC} \]
Map to low-level code

\[ A \in V \]
\[ B_A \in Adj(A) \]
\[ C_{AB} \in Adj(A) \cap Adj(B_A) \]
\[ C_B \in Adj(B_A) - Adj(A) \]
\[ D_{ABC} \in C_{AB} \cap Adj(C_{AB}) \]
\[ D_{BC} \in C_B \cap Adj(C_{AB}) \]

\[ \text{instances}[\text{clique}_4] += D_{ABC} \]
\[ \text{instances}[\text{chordal}] += D_{BC} \]
for v0 in V:
    for v1 in Adj(A):
        y0y1 = Adj(v0) ∩ Adj(v1)
        n0y1 = Adj(v1) - Adj(v0)
        for v2 : y0y1:
            y0y1y2 = y0y1 ∩ Adj(v2)
            n0y1y2 = n0y1 ∩ Adj(v2)
            counter_0 += y0y1y2.size()
            counter_1 += n0y1y2.size()
Map to low-level code

Graph g(file);
#pragma omp parallel for
for(vidType v0 = 0; v0 < n_vertices; v_0++) {
  for(vidType v1 : g.Adj(v0)) {
    VertexSet y0y1 = g.Adj(v0) & g.Adj(v1);
    VertexSet n0y1 = g.Adj(v1) - g.Adj(v0);
    for(vidType v2 : y0y1) {
      VertexSet y0y1y2 = y0y1 & Adj(v2);
      VertexSet n0y1y2 = n0y1 & Adj(v2);
      record_0(v0, v1, v2, y0y1y2);
      record_1(v0, v1, v2, n0y1y2);
    }
  }
}

Parallelization

Data Reuse

VertexSets no longer needed once they go out of scope
API (Automating the Whole Process)

Basic APIs:
Pattern definePattern(Edge[] edgelist);
Program countPatterns(Pattern[] patterns);
Program enumeratePatterns(Pattern[] patterns);

Application-Level APIs:
Program CC(int size);
Program MC(int size);
Program FSM(int size, int support);
Evaluation

• 2x 10-core Intel Xeon E5-2630 v4 CPUs (40 threads), 64gb memory

<table>
<thead>
<tr>
<th>Graph</th>
<th>Vertices</th>
<th>Edges</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>CiteSeer</td>
<td>3264</td>
<td>4536</td>
<td>Publication citation</td>
</tr>
<tr>
<td>MiCo</td>
<td>96638</td>
<td>1080156</td>
<td>Co-authorship</td>
</tr>
<tr>
<td>Patents</td>
<td>3.8M</td>
<td>16.5M</td>
<td>US Patents</td>
</tr>
<tr>
<td>LiveJournal-1</td>
<td>4.8M</td>
<td>42.9M</td>
<td>Social network</td>
</tr>
<tr>
<td>Orkut</td>
<td>3.1M</td>
<td>117.2M</td>
<td>Social network</td>
</tr>
<tr>
<td>UK-2005</td>
<td>39.5M</td>
<td>783M</td>
<td>Web graph</td>
</tr>
<tr>
<td>Youtube</td>
<td>1.1M</td>
<td>3M</td>
<td>Social network</td>
</tr>
<tr>
<td>LiveJournal-2</td>
<td>4M</td>
<td>34.7M</td>
<td>Social network</td>
</tr>
<tr>
<td>GSH-2015</td>
<td>988.5M</td>
<td>25.7B</td>
<td>Web graph</td>
</tr>
</tbody>
</table>
## Performance (Size 3)

<table>
<thead>
<tr>
<th></th>
<th>CiteSeer</th>
<th>MiCo</th>
<th>Patents</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Triangle Counting</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AutoMine</td>
<td>0.01</td>
<td>0.04</td>
<td>0.14</td>
</tr>
<tr>
<td>RStream</td>
<td>0.01</td>
<td>2.5</td>
<td>9.6</td>
</tr>
<tr>
<td>Arabesque</td>
<td>38.1</td>
<td>43.1</td>
<td>114.9</td>
</tr>
<tr>
<td><strong>Motif</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AutoMine</td>
<td>0.016</td>
<td>0.12</td>
<td>0.5</td>
</tr>
<tr>
<td>RStream</td>
<td>0.13</td>
<td>1666.9</td>
<td>1149.1</td>
</tr>
<tr>
<td>Arabesque</td>
<td>40.6</td>
<td>51.7</td>
<td>116</td>
</tr>
<tr>
<td><strong>Frequent Subgraph 5k</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AutoMine</td>
<td>0.02</td>
<td>0.039</td>
<td>3.9</td>
</tr>
<tr>
<td>RStream</td>
<td>0.087</td>
<td>2.54</td>
<td>36.3</td>
</tr>
<tr>
<td>Arabesque</td>
<td>41.6</td>
<td>120.8</td>
<td>F</td>
</tr>
</tbody>
</table>
Performance vs Rstream (Larger Graphs)

Execution Time (S)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>TC</th>
<th>3-MC</th>
<th>TC</th>
<th>3-MC</th>
<th>TC</th>
<th>3-MC</th>
<th>TC</th>
<th>3-MC</th>
</tr>
</thead>
<tbody>
<tr>
<td>LiveJournal-1</td>
<td>TC</td>
<td>3-MC</td>
<td>TC</td>
<td>3-MC</td>
<td>TC</td>
<td>3-MC</td>
<td>FAIL</td>
<td>FAIL</td>
</tr>
<tr>
<td>Orkut</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
</tr>
<tr>
<td>UK-2005</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
</tr>
<tr>
<td>LiveJournal-2</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
</tr>
<tr>
<td>Youtube</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
</tr>
</tbody>
</table>
Performance vs Rstream (FSM-4)
Intermediate Data

![Bar chart showing space need (bytes) for different datasets and models: CiteSeer, MiCo, and Patents. The models are AutoMine and RStream.]
Performance (Large Cliques)

Execution Time (S)

- 6-CC
- 7-CC
- 8-CC

For Youtube:
- 6-CC
- 7-CC
- 8-CC

For Orkut:
- 6-CC
- 7-CC
- 8-CC
Performance vs ASAP [OSDI’18]

- **Execution Time (S)**
- **AutoMine**
- **ASAP**

- **CiteSeer**
  - AutoMine: 0.016
  - ASAP: 1.1

- **MiCo**
  - AutoMine: 0.12
  - ASAP: 2.8

- **Youtube**
  - AutoMine: 0.78
  - ASAP: 4.5

- **LiveJournal-2**
  - AutoMine: 4
  - ASAP: 11.5
Conclusions

• Manual algorithms may be much faster than graph mining systems
• Manual algorithm design doesn’t scale to larger patterns
• AutoMine harmonizes the high-level abstraction and high performance for graph mining through automated algorithm and code generation
• Can we extend this idea to other domains?
AutoMine

Daniel Mawhirter
dmawhirt@mymail.mines.edu

Bo Wu
bwu@mines.edu