Graph Mining Algorithms

Problem: Find subgraphs of interest in an input graph
- Many variants of the problem: Frequent subgraph mining, counting motifs, finding cliques, etc.
- Each variant requires different algorithms

Applications:
- Web: Community detection, link spam detection
- Semantic data: Attributed patterns in RDF
- Biology: Protein-protein or gene interactions

Challenge: Must consider exponential number of subgraphs!

State of the Art

<table>
<thead>
<tr>
<th>Easy to code</th>
<th>High Performance</th>
<th>Transparent Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom Algorithms</td>
<td>☒</td>
<td>☑</td>
</tr>
<tr>
<td>Giraph (TLV)</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Arabesque</td>
<td>☑</td>
<td>☑</td>
</tr>
</tbody>
</table>

Arabesque

A new distributed system for graph mining
- Simple and generic API: Accessible to non-experts
- Simple applications, system does most of the work
- Application-independent system-level optimizations for efficient execution
- System takes care of distributing work and balancing load

Arabesque democratizes the development of distributed algorithms for graph mining

Arabesque API: Example

Finding cliques application using Arabesque (vertex-based exploration)

```java
boolean filter (Embedding e) {
    return isClique(e);
}
void process (Embedding e) {
    output(e);
}
boolean isClique(Embedding e) {
    return e.getNumEdgesAddded() == e.getNumVertices() - 1;
}
```

State of the art dedicated algorithm (Mace): 4,621 lines of C code, centralized

Think Like an Embedding

Coordination-Free Exploration

Avoid redundant work at different workers with embedding canonicality
- Use embedding canonicality: no coordination needed
- Incremental canonicality check in linear time

Example

Initial embedding: 1 – 3 – 6 → canonical
Expansions:
1 – 3 – 6 – 5 → canonical
1 – 3 – 6 – 4 → canonical
1 – 3 – 6 – 2 → not canonical (1 – 2 – 3 – 6)

Fast Pattern Aggregation

New optimization for group-by-pattern aggregation

Storing Embeddings Efficiently

ODAG: New data structure to compress embeddings and balance load

Evaluation

<table>
<thead>
<tr>
<th>Application - Graph</th>
<th>Arabesque - Num. Servers (32 threads)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motifs - MiCo</td>
<td>G-Tries: 8,664s 328s 140s 91s 70s</td>
</tr>
<tr>
<td>FSM - Citeseer</td>
<td>Grami+VFLib: 1,147s + 1,185s 41s 31s 25s</td>
</tr>
<tr>
<td>Cliques - MiCo</td>
<td>Mace: 14,901s 1,167s 74s 41s 31s</td>
</tr>
<tr>
<td>Motifs - Youtube</td>
<td>G-Tries: Fail</td>
</tr>
<tr>
<td>FSM - Patents</td>
<td>Grami+VFLib: 1,147s + 1,185s 548s 132s 102s 88s</td>
</tr>
</tbody>
</table>

Table: Execution time of Arabesque

Figure: Scalability of Arabesque: Speedup relative to the configuration with 5 servers (results from previous table).

Figure: Scalability of Alternative Paradigms: Think Like a Vertex (TLV) and Think Like a Pattern (TLP) using Frequent Subgraph Mining on CiteSeer.

More information at www.arabesque.io