Arabesque.io
A system for distributed graph mining
Carlos Teixeira, Alexandre Fonseca, Marco Serafini,
Georgos Siganos, Mohammed Zaki, Ashraf Aboulnaga
Graphs are ubiquitous
Graph Mining - Concepts

- **Label**
  - Distinguishable property of a vertex (e.g. color).

- **Pattern** - “Meta” sub-graph.
  - Captures subgraph structure and labelling

- **Embedding** - Instance of a pattern.
  - Actual vertices and edges
Graph Mining: Cliques

Property: Fully connected subgraphs
Graph Mining: Motifs

Motifs Size = 3

Motifs Size = 4
Graph Mining: FSM

• Frequent Subgraph mining in a single large graph.

• Find subgraphs that have a minimum embedding count
Applications

• **Web:**
  • Community detection, link spam detection

• **Semantic data:**
  • Attributed patterns in RDF

• **Biology:**
  • Characterize protein-protein or gene interaction
Challenges

Exponential number of embeddings

Size of embedding

# unique embedding (log-scale)

4K 22K 335K 7.8M 117M 1.7B

1 2 3 4 5 6
Challenges

• No standard way to solve these problems.
• No way to distribute the processing easily.
• Way too complicated for programmers (Many ...isms)
  • Detect and identify repeated subgraphs – Automorphisms
  • Aggregate to Pattern – Isomorphism

• Above all not all problems are tractable. No cluster grows exponentially.
## State of the Art: Custom Algorithms

<table>
<thead>
<tr>
<th>Easy to Code</th>
<th>Efficient Implementation</th>
<th>Transparent Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom Algorithms</td>
<td>✔️</td>
<td>❌</td>
</tr>
</tbody>
</table>
# State of the Art: Think Like a Vertex

<table>
<thead>
<tr>
<th></th>
<th>Easy to Code</th>
<th>Efficient Implementation</th>
<th>Transparent Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom Algorithms</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Think Like a Vertex</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>
Arabesque

• New execution model & system
  • Think Like an Embedding
  • Purpose-built for distributed graph mining
  • Hadoop-based

• Contributions:
  • Simple & Generic API
  • High performance
  • Distributed & Scalable by design
## Arabesque

<table>
<thead>
<tr>
<th></th>
<th>Easy to Code</th>
<th>Efficient Implementation</th>
<th>Transparent Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom Algorithms</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Think Like a Vertex</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Arabesque</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Graph Mining - Concepts

- **Label**
  - Distinguishable property of a vertex (e.g. color).

- **Pattern** - “Meta” sub-graph.
  - Captures subgraph structure and labelling

- **Embedding** - Instance of a pattern.
  - Actual vertices and edges
API Example: Clique finding

```java
boolean filter(Embedding e) {
    return isClique(e);
}

void process(Embedding e) {
    output(e);
}

boolean shouldExpand(Embedding embedding) {
    return embedding.getNumVertices() < maxsize;
}

boolean isClique(Embedding e) {
    return e.getNumEdgesAddedWithExpansion()==e.getNumberOfVertices()-1;
}
```

State of the Art (Mace, centralized)

4,621 LOC
API Example: Motif Counting

```java
boolean filter(Embedding e) {
    return true;
}

void process(Embedding embedding) {
    output(embedding);
    map(AGG_MOTIFS, embedding.getPattern(), reusableLongWritableUnit);
}

boolean shouldExpand(Embedding embedding) {
    return embedding.getNumVertices() < maxsize;
}
```

State of the Art
(GTrieScanner, centralized)

3,145 LOC
API Example: FSM

• Ours was the first distributed implementation
• 280 lines of Java Code
  • … of which 212 compute frequent metric

• Baseline (GRAMI): 5,443 lines of Java code.
Arabesque: An Efficient System

- As efficient as centralized state of the art

<table>
<thead>
<tr>
<th>Application - Graph</th>
<th>Centralized Baseline</th>
<th>Arabesque 1 thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motifs - MiCo (MS=3)</td>
<td>50s</td>
<td>37s</td>
</tr>
<tr>
<td>Cliques - MiCo (MS=4)</td>
<td>281s</td>
<td>385s</td>
</tr>
<tr>
<td>FSM - CiteSeer (S=300)</td>
<td>4.8s</td>
<td>5s</td>
</tr>
</tbody>
</table>
Arabesque: A Scalable System

- Scalable to thousands of workers
- Hours/days $\rightarrow$ Minutes

<table>
<thead>
<tr>
<th>Application - Graph</th>
<th>Centralized Baseline</th>
<th>Arabesque 640 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motifs</td>
<td>2 hours 24 minutes</td>
<td>25 seconds</td>
</tr>
<tr>
<td>Clique</td>
<td>4 hours 8 minutes</td>
<td>1 minute 10 seconds</td>
</tr>
<tr>
<td>FSM - Patents</td>
<td>&gt; 1 day</td>
<td>1 minute 28 seconds</td>
</tr>
</tbody>
</table>
How: Arabesque Optimizations

• Avoid Redundant Work
  • Efficient canonicality checking

• Subgraph Compression
  • Overapproximating Directed Acyclic Graphs (ODAGs)

• Efficient Aggregation
  • 2-level pattern aggregation
Outline

- Graph mining exploration & Arabesque fundamentals
- System Architecture & Optimizations
- Evaluation of System
- How to Run & Code
Graph mining exploration & Arabesque fundamentals
Graph Mining - Exploration

- Iterative expansion
  - Subgraph size $n \rightarrow$ Subgraph size $n + 1$
  - Connect to neighbours, one vertex at a time.

Input graph

Depth 1

Depth 2
Graph Mining - Exploration

Input graph

Depth 3
## Arabesque: Fundamentals

- Embeddings as 1st class citizens:
  - Think Like an Embedding model

<table>
<thead>
<tr>
<th>Arabesque responsibilities</th>
<th>User responsibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph Exploration</td>
<td>Filter</td>
</tr>
<tr>
<td>Load Balancing</td>
<td>Process</td>
</tr>
<tr>
<td>Aggregation (Isomorphism)</td>
<td></td>
</tr>
<tr>
<td>Automorphism Detection</td>
<td></td>
</tr>
</tbody>
</table>
Model - Think Like an Embedding

1. Start from a set of initial embeddings
2. **Candidates**: Expand by 1 vertex/edge
3. **Filter** uninteresting candidates
4. Produce **outputs**

**Exploration step 1**
- Input
- Output

**Exploration step 2**
- Input
- Output

**Exploration step 3**
- Input
- Output

**Filter**
- true
- false

**Discard**
- Save

User-defined functions
API Example: Clique finding

```java
boolean filter(Embedding e) {
    return isClique(e);
}
void process(Embedding e) {
    output(e);
}
boolean shouldExpand(Embedding embedding) {
    return embedding.getNumVertices() < maxsize;
}
boolean isClique(Embedding e) {
    return e.getNumEdgesAddedWithExpansion()==e.getNumberofVertices()-1;
}
```
Guarantee: Completeness

For each e, if filter(e) == true then Process(e) is executed

Requirement: Anti-monotonicity

We can prune and be sure that we won’t ignore desired embeddings
Aggregation during expansion

• Filter might need aggregated values
  • E.g.: Frequent subgraph mining
    • Frequency calculation $\rightarrow$ look at all candidates

• Aggregation in parallel with exploration step
  • Embeddings filtered as soon as aggregated values are ready.
Aggregation during expansion

- Filter function may depend on aggregated data
  - E.g.: Frequent subgraph mining
- Frequency requires looking at all candidates

1. **Initial embeddings and aggr. values**
2. **Candidates**: Expand by 1 vertex/edge
3. **Exploration step 1**
   - User-defined functions
4. **Process**
   - map(k, v)
5. **Exploration step 2**
   - Discard
   - Save

1-1. **Filter** using aggr. values
1-2. **Process** using aggr. values

**Aggr. key-value pairs from previous step**

**Aggr. key-value pairs for next step**
Arabesque API

• Main App-defined functions:

  boolean filter(E embedding);
  void process(E embedding);
  boolean shouldExpand(E newEmbedding); // Terminate early if max depth defined
  boolean aggregationFilter(E Embedding); // Ignore embedding
  boolean aggregationFilter(Pattern pattern); // Ignore pattern (ex. not frequent)
  void aggregationProcess(E embedding);
  void handleNoExpansions(E embedding);

• Performance improvements:

  void filter(E existingEmbedding, IntCollection extensionPoints); // prune extensions
  boolean filter(E existingEmbedding, int newWord); // Canonicality check

• Functions Provided by Arabesque:

  void output(String outputString);
  void map(String name, K key, V value);
  AggregationStorage<K, V> readAggregation(String name);
System Architecture & Optimizations
Arabesque Architecture

**Input**
Embeddings size n
- split 1
- split 4
- split 7
- split 2
- split 5
- split 8
- split 3
- split 6
- split 9

**Worker 1**

**Worker 2**

**Worker 3**

**Output**
Embeddings size n + 1
- split 1
- split 4
- split 7
- split 2
- split 5
- split 8
- split 3
- split 6
- split 9

Previous step

Next step
Avoiding redundant work

• **Problem:** Automorphic embeddings
  • Automorphisms == subgraph equivalences
  • Redundant work

![Graphs showing worker 1 and worker 2 with equivalent subgraph structures]
Avoiding redundant work

- **Solution:** Decentralized Embedding Canonicality
  - No coordination
  - Efficient
Efficient Pattern Aggregation

• **Goal:** Aggregate automorphic patterns to single key
  • Find canonical pattern
  • No known polynomial solution
Efficient Pattern Aggregation

• **Solution:** 2-level pattern aggregation
  1. Embeddings → quick patterns
  2. Quick patterns → canonical pattern

![Diagram of pattern aggregation]

1) Quick patterns

2) Canonical pattern

3x Linear matching to quick pattern

2x Expensive graph canonization
Handling Exponential growth

- **Goal:** handle trillions+ different embeddings?

- **Solution:** Overapproximating DAGs (ODAGs)
  - Compress into less restrictive superset
  - Deal with spurious embeddings

![Input Graph](image)

<table>
<thead>
<tr>
<th>Canonical Embeddings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 4 2</td>
</tr>
<tr>
<td>1 4 3</td>
</tr>
<tr>
<td>1 4 5</td>
</tr>
<tr>
<td>2 3 4</td>
</tr>
<tr>
<td>2 4 5</td>
</tr>
<tr>
<td>3 4 5</td>
</tr>
</tbody>
</table>

![Embedding List](image)

![ODAG](image)
Performance
Evaluation - Setup

• 20 servers: 32 threads @ 2.67 GHz, 256GB RAM
• 10 Gbps network

• 3 algorithms: Frequent Subgraph Mining, Counting Motifs and Clique Finding

• Input graphs:

<table>
<thead>
<tr>
<th></th>
<th># Vertices</th>
<th># Edges</th>
<th># Labels</th>
<th>Avg. Degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>CiteSeer</td>
<td>3,312</td>
<td>4,732</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>MiCO</td>
<td>100,000</td>
<td>1,080,298</td>
<td>29</td>
<td>22</td>
</tr>
<tr>
<td>Patents</td>
<td>2,745,761</td>
<td>13,965,409</td>
<td>37</td>
<td>10</td>
</tr>
<tr>
<td>Youtube</td>
<td>4,589,876</td>
<td>43,968,798</td>
<td>80</td>
<td>19</td>
</tr>
<tr>
<td>SN</td>
<td>5,022,893</td>
<td>198,613,776</td>
<td>0</td>
<td>79</td>
</tr>
<tr>
<td>Instagram</td>
<td>179,527,876</td>
<td>887,390,802</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>
Evaluation - Scalability

![Graph showing scalability analysis with different methods including Ideal, Motifs (MiCo), FSM (CiteSeer), Cliques (MiCo), Motifs (Youtube), and FSM (Patents). The graph plots speedup against the number of nodes (32 threads).]
## Evaluation - Scalability

<table>
<thead>
<tr>
<th>Application - Graph</th>
<th>Centralized Baseline</th>
<th>Arabesque - Num. Servers (32 threads)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motifs - MiCo</td>
<td>8,664s</td>
<td>328s</td>
</tr>
<tr>
<td>FSM - Citeseer</td>
<td>1,813s</td>
<td>431s</td>
</tr>
<tr>
<td>Cliques - MiCo</td>
<td>14,901s</td>
<td>1,185s</td>
</tr>
<tr>
<td>Motifs - Youtube</td>
<td>Fail</td>
<td>8,995s</td>
</tr>
<tr>
<td>FSM - Patents</td>
<td>&gt;19h</td>
<td>548s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motifs - MiCo</td>
<td>8,664s</td>
<td>328s</td>
<td>74s</td>
<td>41s</td>
<td>31s</td>
</tr>
<tr>
<td>FSM - Citeseer</td>
<td>1,813s</td>
<td>431s</td>
<td>105s</td>
<td>65s</td>
<td>52s</td>
</tr>
<tr>
<td>Cliques - MiCo</td>
<td>14,901s</td>
<td>1,185s</td>
<td>272s</td>
<td>140s</td>
<td>91s</td>
</tr>
<tr>
<td>Motifs - Youtube</td>
<td>Fail</td>
<td>8,995s</td>
<td>2,218s</td>
<td>1,167s</td>
<td>900s</td>
</tr>
<tr>
<td>FSM - Patents</td>
<td>&gt;19h</td>
<td>548s</td>
<td>186s</td>
<td>132s</td>
<td>102s</td>
</tr>
</tbody>
</table>
Evaluation - ODAGs Compression

- 4000 vertices
- 1.7 billion embeddings
- ODAGs (CiteSeer): 44 GB
- No ODAGs (CiteSeer): 60 MB

Exploration depth:
- 1.7 billion embeddings
- 4000 vertices
Evaluation - Speedup w ODAGs

![Graph showing relative speedup factors for various datasets with ODAGs.](image)

- Motifs MiCo: 1.16
- FSM CiteSeer: 4.18
- Cliques MiCo: 1.77
- Motifs Youtube: 1.19
- FSM Patents: 1.3
# Evaluation - 2-level aggregation

<table>
<thead>
<tr>
<th></th>
<th>Motifs MiCo (MS = 4)</th>
<th>Motifs Youtube (MS=4)</th>
<th>FSM CiteSeer (S=220, MS=7)</th>
<th>FSM Patents (S=24k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Embeddings</td>
<td>10,957,439,024</td>
<td>218,909,854,429</td>
<td>1,680,983,703</td>
<td>1,910,611,704</td>
</tr>
<tr>
<td>Quick Patterns</td>
<td>21</td>
<td>21</td>
<td>1433</td>
<td>1800</td>
</tr>
<tr>
<td>Canonical Patterns</td>
<td>6</td>
<td>6</td>
<td>97</td>
<td>1348</td>
</tr>
<tr>
<td>Reduction Factor</td>
<td>521,782,810x</td>
<td>10,424,278,782x</td>
<td>1,173,052x</td>
<td>1,061,451x</td>
</tr>
</tbody>
</table>
Evaluation - 2-level aggregation

![Chart showing relative slowdown factors for Motifs MiCo with MS=3, Motifs Patents with MS=3, FSM CiteSeer with S=220 MS=6, and FSM Patents with S=30k. The factors are 41.55, 19.63, 33.57, and 12.74 respectively.](chart.png)
How to Run & Code
Requirements

• Hadoop installation:
  • Runs a map-reduce job (Giraph based)
• To develop:
  • Java 7
Input Graph

- Graphs:
  - labels on vertices
  - labels on edges
  - Multiple edges with labels between two vertices
- Graph should have sequential vertex ids, and it should be ordered
How to Run?

./run_arabesque.sh cluster.yaml application.yaml
Cluster.yaml

```
num_workers: 10
num_compute_threads: 16
output_active: yes

# Giraph configuration
#giraph.nettyClientThreads: 32
#giraph.nettyServerThreads: 32
#giraph.nettyClientExecutionThreads: 32
#giraph.channelsPerServer: 4
#giraph.useBigDataIOForMessages: true
#giraph.useNettyPooledAllocator: true
#giraph.useNettyDirectMemory: true
#giraph.nettyRequestEncoderBufferSize: 1048576
```
Fsm.yaml

computation: io.arabesque.examples.fsm.FSMComputation
master_computation: io.arabesque.examples.fsm.FSMMasterComputation

input_graph_path: citeseer.graph
output_path: FSM_Output

#communication_strategy: embeddings

# Custom parameters
arabesque.fsm.support: 300
#arabesque.fsm.maxsize: 7
# Split all aggregations in 10 parts for parallel aggregation
# (use only with heavy aggregations)
# arabesque.aggregators.default_splits: 10
Cliques.yaml

```
computation: io.arabesque.examples.clique.CliqueComputation
input_graph_path: citeseer-single-label.graph
output_path: Cliques_Output

# communication_strategy: embeddings

optimizations:
  - io.arabesque.optimization.CliqueOptimization

# Custom parameters
arabesque.clique.maxsize: 4
```
http://arabesque.io

https://github.com/Qatar-Computing-Research-Institute/Arabesque
Conclusion

- Graph mining is complex
- Existing approaches not ideal
- Arabesque - facilitate distributed graph mining algorithms
  - General & Simple API
  - Efficient & Scalable
- Just the beginning!!!
Graph Exploration with TLV

1. Receive embeddings
2. Expand by adding neighboring vertices
3. Send *canonical* embeddings to their constituting vertices
Evaluation - TLP & TLV

- Use case: frequent subgraph mining
- No scalability. Bottlenecks:
  - TLV: Replication of embeddings, hotspots
  - TLP: very few patterns do all the work

![Graph showing speedup vs number of nodes (32 threads)]