ASAP: Fast, Approximate Graph Pattern Mining at Scale

Anand Iyer*, Zaoxing Liu*, Xin Jin*, Shivaram Venkataraman+, Vladimir Braverman*, Ion Stoica*

*UC Berkeley  *Johns Hopkins University  +University of Wisconsin & Microsoft

OSDI, October 10, 2018
Graphs popular in big data analytics

Social networks
Graphs popular in big data analytics

Social networks

Metabolic network of a single cell organism
Graphs popular in big data analytics

Social networks

Metabolic network of a single cell organism

Tuberculosis
Graphs popular in big data analytics

Also popular in traditional enterprises*

*“The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing” , Sahu et. al, VLDB 2018 (best paper)
Graphs popular in big data analytics

Also popular in traditional enterprises*

Products and customers

*“The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing”, Sahu et. al, VLDB 2018 (best paper)
Graphs popular in big data analytics

Also popular in traditional enterprises*

Products and customers

Transactions and involved entities

*“The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing”, Sahu et al., VLDB 2018 (best paper)
Graphs popular in big data analytics

Also popular in traditional enterprises*

*“The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing”, Sahu et al, VLDB 2018 (best paper)
Graphs popular in big data analytics

Also popular in traditional enterprises*

Which (classes of) products are frequently bought together?

Small deposits followed by large withdrawal

**“The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing”, Sahu et. al, VLDB 2018 (best paper)**
Graphs popular in big data analytics

Also popular in traditional enterprises*

*“The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing”, Sahu et. al, VLDB 2018 (best paper)
Graphs popular in big data analytics

Also popular in traditional enterprises*

Which (classes of) products are frequently bought together?

Small deposits followed by large withdrawal

*“The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing”, Sahu et al, VLDB 2018 (best paper)
Graph Pattern Mining

Discover structural patterns in the underlying graph
Graph Pattern Mining

Discover structural patterns in the underlying graph

Connected Motifs of size 4:
- Star
- Chain
- 3-loop-out
- Box
- Semi-Clique
- Clique

Connected Motifs of size 3:
- Clique
- Chain

Frequent Subgraphs

Motifs

Cliques
Graph Pattern Mining

Discover structural patterns in the underlying graph

**Standard approach**: Iterative expansion

Motifs

Frequent Subgraphs

Cliques
Graph Pattern Mining

Discover structural patterns in the underlying graph

**Standard approach:** Iterative expansion

Motifs

Frequent Subgraphs

Cliquases
Graph Pattern Mining

Discover structural patterns in the underlying graph

Standard approach: Iterative expansion
Graph Pattern Mining

Discover structural patterns in the underlying graph

Standard approach: Iterative expansion
Graph Pattern Mining

Discover structural patterns in the underlying graph

Standard approach: Iterative expansion

Huge intermediate data
Quickly intractable in large graphs
Graph Pattern Mining

Discover structural patterns in the underlying graph

**Standard approach**: Iterative expansion

Challenging to mine patterns in large graphs
Graph Pattern Mining

Arabesque (SOSP '15)
Graph Pattern Mining

Arabesque (SOSP ‘15)

Motifs with size = 3

Log scale

# Edges

Computation Time

~1 billion

11 hours
Graph Pattern Mining

Motifs with size = 3

Arabesque (SOSP ’15)

This work:

Log scale

# Edges

Computation Time

~1 billion

11 hours

1.5 billion

150 s
Graph Pattern Mining

Motifs with size = 3

Arabesque (SOSP '15)

11 hours

~1 billion

Computation Time

# Edges

1.5 billion

150 s

258x faster

This work:
Graph Pattern Mining

~1 billion
11 hours
Motifs with size = 3

Arabesque (SOSP '15)

This work:

# Edges
Computation Time

258x faster
5x less CPU & Memory
Graph Pattern Mining

Arabesque (SOSP ‘15)

Motifs with size = 3

11 hours

~1 billion

This work:

# Edges
Computation Time

# Edges

1.5 billion

150 s

258x faster

5x less CPU & Memory

<5% error
Many mining tasks do not need *exact* answers
Many mining tasks do not need *exact* answers. Leverage *approximation* for pattern mining.
Approximate Analytics

General approach: Apply algorithm on subset(s) (sample) of the input data
Approximate Analytics

General approach: Apply algorithm on subset(s) (sample) of the input data

graph
Approximate Analytics

General approach: Apply algorithm on subset(s) (sample) of the input data

Graph

Edge sampling (p=0.5)
General approach: Apply algorithm on subset(s) (sample) of the input data

triangle counting

e = 1
Approximate Analytics

General approach: Apply algorithm on subset(s) (sample) of the input data

graph

edge sampling (p=0.5)

triangle counting

result

\[ e = 1 \rightarrow e \cdot 2 = 2 \]
Approximate Analytics

General approach: Apply algorithm on subset(s) (sample) of the input data

\[ e = \frac{1}{\binom{n}{3}} \cdot \text{triangle counting} \]

Answer: 10
Approximate Analytics

General approach: Apply algorithm on subset(s) (sample) of the input data

![Graph with edge sampling (p=0.5) and triangle counting result](image)

<table>
<thead>
<tr>
<th>Error (%)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>30</td>
<td>3</td>
</tr>
<tr>
<td>40</td>
<td>4</td>
</tr>
<tr>
<td>50</td>
<td>5</td>
</tr>
<tr>
<td>60</td>
<td>6</td>
</tr>
<tr>
<td>70</td>
<td>7</td>
</tr>
<tr>
<td>80</td>
<td>8</td>
</tr>
<tr>
<td>90</td>
<td>9</td>
</tr>
</tbody>
</table>

![Graph showing error and speedup](image)

- Error: 10
- Speedup: 2
Approximate Analytics

General approach: Apply algorithm on subset(s) (sample) of the input data

- Graph
- Edge sampling (p=0.5)
- Triangle counting
- Result

Applying *exact* algorithm on *sampled* graph(s) not the right approach for pattern mining

Answer: 10
ASAP *leverages* existing work in graph approximation theory and makes it *practical*
Graph Pattern Mining Theory

Sample instances of the pattern from the graph stream
Graph Pattern Mining Theory

*Sample instances of the pattern from the graph stream*

Graph

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Graph Pattern Mining Theory

Sample instances of the pattern from the graph stream

draw graph

e0

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Graph Pattern Mining Theory

Sample instances of the pattern from the graph stream

draw graph

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Graph Pattern Mining Theory

Sample instances of the pattern from the graph stream

graph

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Graph Pattern Mining Theory

Sample instances of the pattern from the graph stream

graph

edge stream: \((0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)\)
Graph Pattern Mining Theory

Sample instances of the pattern from the graph stream

diagram

datastream
Graph Pattern Mining Theory

Sample instances of the pattern from the graph stream

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Graph Pattern Mining Theory

Sample instances of the pattern from the graph stream

graph

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Graph Pattern Mining Theory

Sample instances of the pattern from the graph stream

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Sample instances of the pattern from the graph stream

graph

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Graph Pattern Mining Theory

Sample instances of the pattern from the graph stream

Graph

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Graph Pattern Mining Theory

Sample instances of the pattern from the graph stream

graph

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Graph Pattern Mining Theory

Sample instances of the pattern from the graph stream

Graph

graph

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)

$p = \frac{1}{10} \times \frac{1}{4}$
Graph Pattern Mining Theory

Sample instances of the pattern from the graph stream

graph

\[ p = \frac{1}{10} \times \frac{1}{4} \]
\[ e_0 = 40 \]

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Sample instances of the pattern from the graph stream

Graph Pattern Mining Theory

e_0 = 40

\[ p = \frac{1}{10} \times \frac{1}{4} \]

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Graph Pattern Mining Theory

Sample instances of the pattern from the graph stream

graph

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)

p = \frac{1}{10} \times \frac{1}{4}

e_0 = 40

e_1 = 0

e_2 = 0

e_3 = 0

\frac{1}{r} \sum_{i=0}^{r-1} e_i = 10
Graph Pattern Mining Theory

Sample instances of the pattern from the graph stream

graph

estimator (r=4)

neighborhood sampling

result

\[
\frac{1}{r} \sum_{i=0}^{r-1} e_i = 10
\]

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)

Pavan et al. Counting and sampling triangles from a graph stream, VLDB 2013
A Swift Approximate Pattern miner
graphA.patterns("a->b->c", "100s")
graphB.fourClique("5.0\%", "95.0\%")
Graphs stored on disk or main memory

1. `graphA.patterns(“a->b->c”, “100s”)`
2. `graphB.fourClique(“5.0%”, “95.0%”)`

Generalized Approximate Pattern Mining

Apache Spark

Copyright © 2017-present, the Swift Project. All rights reserved.
Graphs stored on disk or main memory

graphA.patterns("a->b->c", "100s")
graphB.fourClique("5.0\%", "95.0\%")

Estimator Count Selection

Generalized Approximate Pattern Mining

Apache Spark

Swift Approximate Pattern miner

Graphs stored on disk or main memory
Graphs stored on disk or main memory

1. A Swift Approximate Pattern miner
2. Generalized Approximate Pattern Mining
3. Estimator Count Selection
4. Graphs stored on disk or main memory

graphA.patterns(“a->b->c”, “100s”) graphB.fourClique(“5.0%”, “95.0%”)
Graphs stored on disk or main memory

Estimator Count Selection

Error-Latency Profile (ELP)

Generalized Approximate Pattern Mining

Apache Spark
graphA.patterns("a->b->c", "100s")
graphB.fourClique("5.0\%", "95.0\%")

Estimates:{error: <5\%, time: 95s}
Estimates:{error: <5\%, time: 60s}

**Estimator Count Selection**

**Generalized Approximate Pattern Mining**

**Apache Spark**

**Error-Latency Profile (ELP)**

**Graphs stored on disk or main memory**
A Swift Approximate Pattern miner

Graphs stored on disk or main memory

1. `graphA.patterns("a->b->c", "100s")`
2. `graphB.fourClique("5.0%", "95.0%")`
3. `Estimator Count Selection`
4. Error-Latency Profile (ELP)
5. `count: 21453 +/- 14
count: 95%,
time: 92s`
6. `Embeddings (optional)`
7. `count: 21453 +/- 14
count: 95%,
time: 92s`
Graph updates

Graphs stored on disk or main memory

graphA.\texttt{patterns}("a->b->c", "100s")

graphB.\texttt{fourClique}("5.0\%", "95.0\%")

Estimates:{error: <5\%, time: 95s}
Estimates:{error: <5\%, time: 60s}

Estimator Count Selection

Generalized Approximate Pattern Mining

Apache Spark

Apache Spark

Embeddings (optional)

3

4

5

6

7

count: 21453 +/- 14
confidence: 95\%
time: 92s

Graphs stored on disk or main memory

Error-Latency Profile (ELP)
graphA.patterns("a->b->c", "100s")
graphB.fourClique("5.0%", "95.0%")

Estimates:{error: <5%, time: 95s}
Estimates:{error: <5%, time: 60s}

Generalized Approximate Pattern Mining

Estimator Count Selection

Error-Latency Profile (ELP)

Embeddings (optional)

Graphs stored on disk or main memory

Twitter Graph Profiling

Graph updates

count: 21453 +/- 14
confidence: 95%,
time: 92s

0 0.5M 1M 1.5M 2.1M
0 0.5M 1M 1.5M 2.1M
Generalized Approximate Pattern Mining

Contributions:
- Extends neighborhood sampling to general patterns
- Provides a unified API
- Applies approximate pattern mining in distributed settings

Estimator Count Selection

Graphs stored on disk or main memory

Errors:

Graph updates

Count: 21453 +/- 14
Confidence: 95%
Time: 92s

Apache Spark

Embeddings (optional)
Generalized Approximate Pattern Mining

Developers write a single estimator using ASAP’s API

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sampleVertex: ()→(v,p)</td>
<td>Uniformly sample one vertex from the graph.</td>
</tr>
<tr>
<td>SampleEdge: ()→(e,p)</td>
<td>Uniformly sample one edge from the graph.</td>
</tr>
<tr>
<td>ConditionalSampleVertex: (subgraph)→(v,p)</td>
<td>Uniformly sample a vertex that appears after a sampled subgraph.</td>
</tr>
<tr>
<td>ConditionalSampleEdge: (subgraph)→(e,p)</td>
<td>Uniformly sample an edge that is adjacent to the given subgraph and comes after the subgraph in the order.</td>
</tr>
<tr>
<td>ConditionalClose: (subgraph, subgraph)→boolean</td>
<td>Given a sampled subgraph, check if another subgraph that appears later in the order can be formed.</td>
</tr>
</tbody>
</table>
Generalized Approximate Pattern Mining

Developers write a single estimator using ASAP’s API

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SampleVertex</strong>: () → (v, p)</td>
<td>Uniformly sample one vertex from the graph.</td>
</tr>
<tr>
<td><strong>SampleEdge</strong>: () → (e, p)</td>
<td>Uniformly sample one edge from the graph.</td>
</tr>
<tr>
<td><strong>ConditionalSampleVertex</strong>: (subgraph) → (v, p)</td>
<td>Uniformly sample a vertex that appears after a sampled subgraph.</td>
</tr>
<tr>
<td><strong>ConditionalSampleEdge</strong>: (subgraph) → (e, p)</td>
<td>Uniformly sample an edge that is adjacent to the given subgraph and comes after the subgraph in the order.</td>
</tr>
<tr>
<td><strong>ConditionalClose</strong>: (subgraph, subgraph) → boolean</td>
<td>Given a sampled subgraph, check if another subgraph that appears later in the order can be formed.</td>
</tr>
</tbody>
</table>
Developers write a single estimator using ASAP’s API

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SampleVertex</strong>: ( ) → (v, p)</td>
<td>Uniformly sample one vertex from the graph.</td>
</tr>
<tr>
<td><strong>SampleEdge</strong>: ( ) → (e, p)</td>
<td>Uniformly sample one edge from the graph.</td>
</tr>
<tr>
<td><strong>ConditionalSampleVertex</strong>: (subgraph) → (v, p)</td>
<td>Uniformly sample a vertex that appears after a sampled subgraph.</td>
</tr>
<tr>
<td><strong>ConditionalSampleEdge</strong>: (subgraph) → (e, p)</td>
<td>Uniformly sample an edge that is adjacent to the given subgraph and comes after the subgraph in the order.</td>
</tr>
<tr>
<td><strong>ConditionalClose</strong>: (subgraph, subgraph) → boolean</td>
<td>Given a sampled subgraph, check if another subgraph that appears later in the order can be formed.</td>
</tr>
</tbody>
</table>
Using ASAP’s API

SampleTriangle

(e1, p1) = sampleEdge()
(e2, p2) = conditionalSampleEdge(Subgraph(e1))
if (!e2) return 0
subgraph1 = Subgraph(e1, e2)
subgraph2 = Triangle(e1, e2)-subgraph1
if conditionalClose(subgraph1, subgraph2)
    return 1/(p1.p2)
else return 0

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Using ASAP’s API

SampleTriangle

(e1, p1) = sampleEdge()
(e2, p2) = conditionalSampleEdge(Subgraph(e1))
if (!e2) return 0
subgraph1 = Subgraph(e1, e2)
subgraph2 = Triangle(e1, e2)-subgraph1
if conditionalClose(subgraph1, subgraph2)
    return 1/(p1.p2)
el se return 0

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Using ASAP’s API

SampleTriangle

\[
\begin{align*}
(e_1, p_1) &= \text{sampleEdge}() \\
(e_2, p_2) &= \text{conditionalSampleEdge}(\text{Subgraph}(e_1)) \\
\text{if} & \, (\neg e_2) \, \text{return} \, 0 \\
\text{subgraph1} &= \text{Subgraph}(e_1, e_2) \\
\text{subgraph2} &= \text{Triangle}(e_1, e_2) \cdot \text{subgraph1} \\
\text{if} & \, \text{conditionalClose}(\text{subgraph1}, \text{subgraph2}) \\
\quad & \text{return} \, 1/(p_1.p_2) \\
\text{else} & \, \text{return} \, 0
\end{align*}
\]

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Using ASAP’s API

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)

SampleTriangle

(e1, p1) = sampleEdge()
(e2, p2) = conditionalSampleEdge(Subgraph(e1))
if (!e2) return 0
subgraph1 = Subgraph(e1, e2)
subgraph2 = Triangle(e1, e2)-subgraph1
if conditionalClose(subgraph1, subgraph2)
    return 1/(p1.p2)
else return 0
Using ASAP’s API

SampleTriangle

```python
(e1, p1) = sampleEdge()
(e2, p2) = conditionalSampleEdge(Subgraph(e1))
if (!e2) return 0
subgraph1 = Subgraph(e1, e2)
subgraph2 = Triangle(e1, e2)-subgraph1
if conditionalClose(subgraph1, subgraph2)
    return 1/(p1*p2)
else return 0
```

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Using ASAP’s API

SampleTriangle

```python
(e1, p1) = sampleEdge()
(e2, p2) = conditionalSampleEdge(Subgraph(e1))
if (!e2) return 0
subgraph1 = Subgraph(e1, e2)
subgraph2 = Triangle(e1, e2) - subgraph1
if conditionalClose(subgraph1, subgraph2)
    return 1/(p1.p2)
else return 0
```

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Using ASAP’s API

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)

SampleTriangle

(e1, p1) = sampleEdge()
(e2, p2) = conditionalSampleEdge(Subgraph(e1))
if (!e2) return 0
subgraph1 = Subgraph(e1, e2)
subgraph2 = Triangle(e1, e2)-subgraph1
if conditionalClose(subgraph1, subgraph2)
    return 1/(p1.p2)
else return 0
Using ASAP’s API

Using ASAP’s API, the program developer can use several functions to sample subgraphs. The API provides five functions as shown below:

- **Uniformly sample one edge from the graph.**
- **Uniformly sample one vertex from the graph.**
- **Uniformly sample an edge that is adjacent to the given vertex.**
- **Uniformly sample a vertex that appears after a sampled vertex.**
- **Conditional sample an edge given that another edge has already been sampled.**

Using ASAP’s API, we can write a function to sample a triangle. Here’s an example of how to do it:

```plaintext
SampleTriangle

(e1, p1) = sampleEdge()
(e2, p2) = conditionalSampleEdge(Subgraph(e1))
if (!e2) return 0
subgraph1 = Subgraph(e1, e2)
subgraph2 = Triangle(e1, e2) - subgraph1
if conditionalClose(subgraph1, subgraph2)
    return 1/(p1.p2)
else return 0
```

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Using ASAP’s API

SampleTriangle

(e1, p1) = sampleEdge()
(e2, p2) = conditionalSampleEdge(Subgraph(e1))
if (!e2) return 0
subgraph1 = Subgraph(e1, e2)
subgraph2 = Triangle(e1, e2)-subgraph1
if conditionalClose(subgraph1, subgraph2)
    return 1/(p1\cdot p2)
else return 0

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Using ASAP’s API

SampleTriangle

\[(e_1, p_1) = \text{sampleEdge}()\]
\[(e_2, p_2) = \text{conditionalSampleEdge}(\text{Subgraph}(e_1))\]
if (!e2) return 0
subgraph1 = Subgraph(e1, e2)
subgraph2 = Triangle(e1, e2)-subgraph1
if conditionalClose(subgraph1, subgraph2)
    return \(1/(p_1.p_2)\)
else return 0

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Using ASAP's API

SampleTriangle

\[(e_1, p_1) = \text{sampleEdge}()\]
\[(e_2, p_2) = \text{conditionalSampleEdge}(\text{Subgraph}(e_1))\]
if (!e2) return 0
subgraph1 = \text{Subgraph}(e_1, e_2)
subgraph2 = \text{Triangle}(e_1, e_2) - \text{subgraph1}
if \text{conditionalClose}(\text{subgraph1}, \text{subgraph2})
  return 1/(p1.p2)
else return 0

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Using ASAP’s API

SampleTriangle

\[(e_1, p_1) = \text{sampleEdge}()\]
\[(e_2, p_2) = \text{conditionalSampleEdge}(\text{Subgraph}(e_1))\]
\[\text{if } (!e_2) \text{ return } 0\]
\[\text{subgraph1} = \text{Subgraph}(e_1, e_2)\]
\[\text{subgraph2} = \text{Triangle}(e_1, e_2)-\text{subgraph1}\]
\[\text{if } \text{conditionalClose}(\text{subgraph1}, \text{subgraph2})\]
\[\quad \text{return } 1/(p_1.p_2)\]
\[\text{else return } 0\]

edge stream: \((0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)\)
Using ASAP’s API

SampleTriangle

\[(e_1, p_1) = \text{sampleEdge}()\]
\[(e_2, p_2) = \text{conditionalSampleEdge}((\text{Subgraph}(e_1))\]  
\[\text{if } (!e_2) \text{ return 0}\]
\[\text{subgraph1} = \text{Subgraph}(e_1, e_2)\]
\[\text{subgraph2} = \text{Triangle}(e_1, e_2)-\text{subgraph1}\]
\[\text{if } \text{conditionalClose}((\text{subgraph1}, \text{subgraph2}))\]
\[\text{return } 1/(p_1.p_2)\]
\[\text{else return 0}\]

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Using ASAP’s API

![Diagram of a graph with nodes and edges]

SampleTriangle

(e1, p1) = sampleEdge()
(e2, p2) = conditionalSampleEdge(Subgraph(e1))
if (!e2) return 0
subgraph1 = Subgraph(e1, e2)
subgraph2 = Triangle(e1, e2)-subgraph1
if conditionalClose(subgraph1, subgraph2)
    return 1/(p1.p2)
else return 0

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Using ASAP’s API

SampleTriangle

(e1, p1) = sampleEdge()
(e2, p2) = conditionalSampleEdge(Subgraph(e1))
if (!e2) return 0
subgraph1 = Subgraph(e1, e2)
subgraph2 = Triangle(e1, e2)-subgraph1
if conditionalClose(subgraph1, subgraph2)
    return 1/(p1.p2)
else return 0

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Using ASAP’s API

SampleTriangle

(e1, p1) = sampleEdge()
(e2, p2) = conditionalSampleEdge(Subgraph(e1))
if (!e2) return 0
subgraph1 = Subgraph(e1, e2)
subgraph2 = Triangle(e1, e2)-subgraph1
if conditionalClose(subgraph1, subgraph2)
    return 1/(p1*p2)
else return 0

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Using ASAP’s API

SampleTriangle

\[(e_1, p_1) = \text{sampleEdge}()\]
\[(e_2, p_2) = \text{conditionalSampleEdge}(\text{Subgraph}(e_1))\]
if (!e2) return 0
subgraph1 = \text{Subgraph}(e_1, e_2)
subgraph2 = \text{Triangle}(e_1, e_2)\text{-subgraph1}
if \text{conditionalClose}(\text{subgraph1}, \text{subgraph2})
  return 1/(p_1.p_2)
else return 0

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Using ASAP’s API

illustrate how to use these functions to sample patterns:

```python
SampleFourCliqueType1
SampleThreeNodeChain
SampleEdge
SampleVertex
ConditionalSampleEdge
ConditionalSampleVertex
ConditionalClose
```

The programming API provides five functions as shown:

- `SampleEdge`: Uniformly samples one edge from the graph. It takes no input, and outputs an edge.
- `SampleVertex`: Uniformly samples one vertex from the graph. It takes no input, and outputs a vertex.
- `ConditionalSampleEdge`: Given a sampled subgraph, check if another subgraph can be formed. It takes the two subgraphs as input and returns a boolean value indicating whether the second pattern can be formed and the sampling process only awaits the pattern where all nodes of a possible instance have been fixed and the sampling process only awaits the pattern where all nodes of a possible instance have been fixed.
- `ConditionalSampleVertex`: Uniformly samples one vertex that appears after a sampled vertex. It takes the two subgraphs as input and outputs a vertex.
- `ConditionalClose`: Uniformly samples one edge that is adjacent to the given vertex. It takes no input, and outputs an edge.

We describe each function in detail and a few use cases to illustrate the usage of the program’s information algorithms for graph pattern mining.

**Use Cases**

- **Example approximate pattern mining programs written using ASAP’s Approximate Pattern Mining API.**

```python
edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
```

**SampleTriangle**

```python
(e1, p1) = sampleEdge()
(e2, p2) = conditionalSampleEdge(Subgraph(e1))
if (!e2) return 0
subgraph1 = Subgraph(e1, e2)
subgraph2 = Triangle(e1, e2)-subgraph1
if conditionalClose(subgraph1, subgraph2)
    return 1/(p1.p2)
else return 0
```
Using ASAP’s API

SampleTriangle

\[
(e_1, p_1) = \text{sampleEdge}() \\
(e_2, p_2) = \text{conditionalSampleEdge}(\text{Subgraph}(e_1)) \\
\text{if } (!e_2) \text{ return } 0 \\
\text{subgraph1} = \text{Subgraph}(e_1, e_2) \\
\text{subgraph2} = \text{Triangle}(e_1, e_2) - \text{subgraph1} \\
\text{if } \text{conditionalClose}(\text{subgraph1}, \text{subgraph2}) \\
\text{return } 1/(p_1.p_2) \\
\text{else } \text{return } 0
\]

described as a function that is usually used as the final step to sample a pattern.

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Using ASAP’s API

SampleTriangle

(e1, p1) = sampleEdge()
(e2, p2) = conditionalSampleEdge(Subgraph(e1))
if (!e2) return 0
subgraph1 = Subgraph(e1, e2)
subgraph2 = Triangle(e1, e2)-subgraph1
if conditionalClose(subgraph1, subgraph2)
    return 1/(p1.p2)
else return 0

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Using ASAP’s API

SampleTriangle
(e1, p1) = sampleEdge()
(e2, p2) = conditionalSampleEdge(Subgraph(e1))
if (!e2) return 0
subgraph1 = Subgraph(e1, e2)
subgraph2 = Triangle(e1, e2)-subgraph1
if conditionalClose(subgraph1, subgraph2)
  return 1/(p1 * p2)
else return 0

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Using ASAP’s API

Sampling phase *fixes the vertices* for a particular instance of a pattern and closing phase *waits for remaining edges*

edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)

### SampleTriangle

\[(e_1, p_1) = \text{sampleEdge}()\]
\[(e_2, p_2) = \text{conditionalSampleEdge(Subgraph}(e_1))\]
if (!e2) return 0
subgraph1 = Subgraph(e1, e2)
subgraph2 = Triangle(e1, e2)-subgraph1
if conditionalClose(subgraph1, subgraph2)
  return 1/(p1.p2)
else return 0
Using ASAP’s API

SampleTriangle

(e1, p1) = sampleEdge()
(e2, p2) = conditionalSampleEdge(Subgraph(e1))
if (!e2) return 0
subgraph1 = Subgraph(e1, e2)
subgraph2 = Triangle(e1, e2)-subgraph1
if conditionalClose(subgraph1, subgraph2)
    return 1/(p1.p2)
else return 0

ASAP computes the right expectations, runs many instances of the estimator and aggregates results.
Using ASAP’s API

Using ASAP’s API to write a sampling function for a particular instance of a pattern where all nodes of a possible instance have additional edges to form the pattern.

\[ e_0 = 40 \]

**SampleTriangle**

\[
\begin{align*}
(e_1, p_1) &= \text{sampleEdge}() \\
(e_2, p_2) &= \text{conditionalSampleEdge}(<\text{Subgraph}(e_1)) \\
\text{if }(!e_2) \text{ return } 0 \\
\text{subgraph1} &= <\text{Subgraph}(e_1, e_2) \\
\text{subgraph2} &= <\text{Triangle}(e_1, e_2)-\text{subgraph1} \\
\text{if } \text{conditionalClose}(\text{subgraph1, subgraph2}) \\
\text{return } 1/(p_1.p_2) \\
\text{else return } 0
\end{align*}
\]

ASAP computes the right expectations, runs many instances of the estimator and aggregates results.

See paper for more examples & proof
Applying to Distributed Settings
Applying to Distributed Settings

graph

subgraph 0 → partial count $c_0$ (using $r$ estimators)
subgraph 1 → partial count $c_1$ (using $r$ estimators)
subgraph 2 → partial count $c_2$ (using $r$ estimators)
Applying to Distributed Settings

map: \( w(=3) \) workers

- subgraph 0 → partial count \( c_0 \) (using \( r \) estimators)
- subgraph 1 → partial count \( c_1 \) (using \( r \) estimators)
- subgraph 2 → partial count \( c_2 \) (using \( r \) estimators)
Applying to Distributed Settings

map: \( w(=3) \) workers

- subgraph 0
  - partial count \( c_0 \)
  - (using \( r \) estimators)

- subgraph 1
  - partial count \( c_1 \)
  - (using \( r \) estimators)

- subgraph 2
  - partial count \( c_2 \)
  - (using \( r \) estimators)
Applying to Distributed Settings

map: $w(=3)$ workers

- subgraph 0 → partial count $c_0$ (using $r$ estimators)
- subgraph 1 → partial count $c_1$ (using $r$ estimators)
- subgraph 2 → partial count $c_2$ (using $r$ estimators)

$$\sum_{i=0}^{w-1} c_i$$
Applying to Distributed Settings

map: \( w(=3) \) workers

reduce

\[
\sum_{i=0}^{w-1} c_i
\]
Applying to Distributed Settings

Random Vertex-cut Partitioning

Graph: $\text{graph}$

- Subgraph 0: partial count $c_0$ (using $r$ estimators)
- Subgraph 1: partial count $c_1$ (using $r$ estimators)
- Subgraph 2: partial count $c_2$ (using $r$ estimators)

Reduce:

$$\sum_{i=0}^{w-1} c_i$$
Applying to Distributed Settings

Random Vertex-cut Partitioning

- subgraph 0 → partial count $c_0$ (using $r$ estimators)
- subgraph 1 → partial count $c_1$ (using $r$ estimators)
- subgraph 2 → partial count $c_2$ (using $r$ estimators)

reduce

$f(w) \sum_{i=0}^{w-1} c_i$
Applying to Distributed Settings

Random Vertex-cut Partitioning

Graph

- subgraph 0
  - partial count $c_0$ (using $r$ estimators)

- subgraph 1
  - partial count $c_1$ (using $r$ estimators)

Lower bounds on $f(w)$ can be proved using Hajnal-Szemerédi theorem
graphA.patterns("a->b->c", "100s")

graphB.fourClique("5.0\%", "95.0\%")

Estimates:{error: <5\%, time: 95s}
Estimates:{error: <5\%, time: 60s}

Generalized Approximate Pattern Mining

Apache Spark

Estimator Count Selection

Graphs stored on disk or main memory

Embeddings (optional)

Error-Latency Profile (ELP)

count: 21453 +/- 14
count: 95\%,
time: 92s

Twitter Graph Profiling

Error Rate (%)

No. of Estimators

Runtime (min)

No. of Estimators

Twitter Graph Profiling

Graph updates

Graphs stored on disk

Graphs stored in main memory
Contribution:

- Novel way to build ELP very fast without the need to know the ground truth or running mining on the full graph.
Building Error-Latency Profile

Given a time / error bound, how many estimators should ASAP use?
Building Error-Latency Profile

Given a time / error bound, how many estimators should ASAP use?

Time vs Estimators

![Graph showing the relationship between time and number of estimators.](Image)
Building Error-Latency Profile

Given a time / error bound, how many estimators should ASAP use?

Time vs Estimators

Error vs Estimators

![Graph showing time vs number of estimators and error vs number of estimators.](image-url)
Building Estimators vs Time Profile

Time complexity linear in number of estimators
Building Estimators vs Time Profile

Time complexity linear in number of estimators
Building Estimators vs Time Profile

Time complexity linear in number of estimators

ASAP sets a profiling cost and picks maximum points within the budget
Building Estimators vs Time Profile

Time complexity linear in number of estimators

![Graph showing runtime vs number of estimators for Twitter Graph Profiling]
Building Estimators vs Error Profile

Error complexity non-linear in number of estimators

Twitter Graph

No. of Estimators

Error Rate (%)
Building Estimators vs Error Profile

Error complexity non-linear in number of estimators

Key idea: Use a very small sample of the graph to build the ELP

- Chernoff analysis provides a loose upper bound on the number of estimators.
- In small graphs, a large number of estimators can get us very close to ground truth.
Building Estimators vs Error Profile

Error complexity non-linear in number of estimators
Advanced Mining

Predicate Matching

• Find patterns where vertices are of type “electronics”
• ASAP allows simple edge and vertex predicates

Motif Mining

• Some patterns are building blocks for other patterns
• ASAP caches state of the estimators and reuses them

Accuracy Refinement

• Users may require more accurate answer later
• ASAP can checkpoint and reuse estimators
Implementation & Evaluation

- **Implemented on Apache Spark**
  - Not limited to it, only relies on simple dataflow operators
- **Evaluated in a 16 node cluster**
  - Twitter: 1.47B edges
  - Friendster: 1.8B edges
  - UK: 3.73B edges
- **Comparison using representative patterns:**
  - 3 (2 patterns), 4 (6 patterns) and 5 motifs (21 patterns)
Performance on Small Graphs

4-Motifs (6 patterns)

<table>
<thead>
<tr>
<th>Graph</th>
<th>Time (s) Arabesque</th>
<th>Time (s) ASAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>CiteSeer</td>
<td>12.1</td>
<td>7.3</td>
</tr>
<tr>
<td>Mico</td>
<td>162</td>
<td>14.9</td>
</tr>
<tr>
<td>Youtube</td>
<td>291.4</td>
<td>18.1</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>3161</td>
<td>41.6</td>
</tr>
</tbody>
</table>
Performance on Small Graphs

4-Motifs (6 patterns)

<table>
<thead>
<tr>
<th>Graph</th>
<th>Time (s)</th>
<th>Motifs (6 patterns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CiteSeer</td>
<td>12.1</td>
<td>7.3</td>
</tr>
<tr>
<td>Mico</td>
<td>162</td>
<td>14.9</td>
</tr>
<tr>
<td>Youtube</td>
<td>291.4</td>
<td>18.1</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>3161</td>
<td>41.6</td>
</tr>
</tbody>
</table>

- Arabesque
- ASAP
Performance on Small Graphs

4-Motifs (6 patterns)

<table>
<thead>
<tr>
<th></th>
<th>CiteSeer</th>
<th>Mico</th>
<th>Youtube</th>
<th>LiveJournal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arabesque</td>
<td>12.1</td>
<td>162</td>
<td>291.4</td>
<td>3161</td>
</tr>
<tr>
<td>ASAP</td>
<td>7.3</td>
<td>14.9</td>
<td>18.1</td>
<td>41.6</td>
</tr>
</tbody>
</table>

77 x <5% error
Large Graphs & Simple Patterns

3-Motifs (2 patterns)

Time (min)

# Edges (Billions)

<table>
<thead>
<tr>
<th># Edges (Billions)</th>
<th>Arabesque</th>
<th>ASAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9</td>
<td>645</td>
<td>2.5</td>
</tr>
<tr>
<td>1.5</td>
<td>5</td>
<td>5.9</td>
</tr>
<tr>
<td>1.8</td>
<td>3.7</td>
<td></td>
</tr>
</tbody>
</table>
Large Graphs & Simple Patterns

3-Motifs (2 patterns)

Proprietary graph, 20 machines (256GB each)

Time (min)

# Edges (Billions)

Arabesque
ASAP
Large Graphs & Simple Patterns

3-Motifs (2 patterns)

Proprietary graph, 20 machines (256GB each)

258 x <5% error

Time (min)

# Edges (Billions)

Twitter 1.5
Friendster 5
UK 3.7

Arabesque
ASAP
Large Graphs & Simple Patterns

3-Motifs (2 patterns)

Proprietary graph, 20 machines (256GB each)

Time (min)

# Edges (Billions)

258 x <5% error
Large Graphs & Complex Patterns

4-Motifs

Twitter

UK

Time (min)

22

47
Large Graphs & Complex Patterns

4-Motifs

Time (min)

Twitter | UK
---|---
22 | 47

5-House

Time (min)

Twitter | UK
---|---
12.3 | 22.1
5.6 | 14.2

5% | 10%
Summary

- Pattern mining important & challenging problem
  - Applications in many domains

- ASAP uses approximation for fast pattern mining
  - Leverages graph mining theory & makes it practical
  - Simple API for developers

- ASAP outperforms existing solutions
  - Can handle much larger graphs with fewer resources

http://www.cs.berkeley.edu/~api
api@cs.berkeley.edu