ASAP: Fast, Approximate Graph Pattern Mining at Scale

Anand Iyer et al. @ OSDI 2018

Presenter: Yunang Chen
ASAP Design Overview

A Swift Approximate Pattern-miner
Navigates tradeoff between result accuracy and latency
Runs on general-purpose distributed dataflow platform
Supports for generalized graph pattern mining algorithms
Graph Pattern Mining

Standard approach: Iterative expansion

Lack of scalability
- Generate exponentially large intermediate candidate sets
- Need to store + exchange them in distributed environment
Graph Pattern Mining

Standard approach: Iterative expansion

Lack of scalability
- Generate exponentially large intermediate candidate sets
- Need to store and exchange them in distributed environment

Experiments performed on a cluster of 20 machines, each having 256GB of memory.
Many pattern mining tasks do not need exact answers.

- Frequent sub-graph mining (FSM) finds the frequency of subgraphs but with an end-goal of ordering them by occurrences.

Leverage approximation for pattern mining
Approximate Pattern Mining

**Previous approach**: Apply the exact same algorithm on subsets of the input data, then use the statistical properties of these subsets to estimate final results.

<table>
<thead>
<tr>
<th>graph</th>
<th>edge sampling (p=0.5)</th>
<th>triangle counting</th>
<th>result</th>
</tr>
</thead>
</table>

![Graph diagram](image)

- **Result**: $e = 1$ → $e \cdot 2 = 2$
Approximate Pattern Mining

Previous approach: Apply the exact same algorithm on subsets of the input data, then use the statistical properties of these subsets to estimate final results.

- No significant speedup
- Large error rate
Approximate Pattern Mining

Neighborhood sampling:
1. Model the edges in the graph as a stream
2. Sample one edge, $e_1$
3. Gradually add more adjacent edges, $e_2, ..., e_k$
4. Stop when the edges form the pattern or becomes impossible to do so
5. Use the probability of sampling to bound the total number of occurrences of the pattern:
   \[ P(e_1, ..., e_k) = P(e_1) \times P(e_2 | e_1) \times ... \times P(e_k | e_1, ..., e_{k-1}) \]
6. Repeat Step 1-5 multiple times
Approximate Pattern Mining

Neighborhood sampling: Triangle Counting

1. Model the edges in the graph as a stream

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

Neighborhood sampling: Triangle Counting

2. Sample one edge

diagram

element stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Approximate Pattern Mining

Neighborhood sampling: Triangle Counting
3. Gradually add more adjacent edges

Edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
Approximate Pattern Mining

Neighborhood sampling: Triangle Counting

4. Stop when the edges form the pattern or becomes impossible to do so

equation

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

Neighborhood sampling: Triangle Counting

4. Stop when the edges form the pattern or becomes impossible to do so

edge stream: (0, 1), (0, 2), (0, 3), (0, 4), (1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)
Neighborhood sampling: Triangle Counting

5. Use the probability of sampling to bound the total number of occurrences

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

Neighborhood sampling: Triangle Counting

6. Repeat Step 1-5 multiple times

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

1. `graphA.patterns("a->b->c", "100s")`
2. `graphB.fourClique("5.0%", "95.0%")`
3. Estimates: `{error: <5%, time: 95s}`
4. Estimates: `{error: <5%, time: 60s}`
5. **Error-Latency Profile (ELP) Building**
6. **Estimator Count Selection**
7. **Generalized Approximate Pattern Mining**
   - `count: 21453 +/- 14`
   - `confidence: 95%`
   - `time: 92s`

---

**Embeddings (optional)**

- Graphs stored on disk or main memory

---

**Graph updates**
Neighborhood sampling:
1. Model the edges in the graph as a stream
2. Sample one edge, $e_{\downarrow 1}$
3. Gradually add more adjacent edges, $e_{\downarrow 2}, ..., e_{\downarrow k}$
4. Stop when the edges form the pattern or becomes impossible to do so
5. Use the probability of sampling to bound the total number of occurrences of the pattern:
   \[
   P(e_{\downarrow 1}, ..., e_{\downarrow k}) = P(e_{\downarrow 1}) \times P(e_{\downarrow 2} | e_{\downarrow 1}) \times \ldots \times P(e_{\downarrow k} | e_{\downarrow 1}, ..., e_{\downarrow k-1})
   \]
6. Repeat Step 1-5 multiple times

API

- sampleVertex: () → (v, p)
- SampleEdge: () → (e, p)
- ConditionalSampleVertex: (subgraph) → (v, p)
- ConditionalSampleEdge: (subgraph) → (e, p)
- ConditionalClose: (subgraph, subgraph) → boolean
Programming API

```
(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
```

Sampling Phase: fix the vertices for a pattern

Closing Phase: waiting for remaining edges to complete the pattern
Distributed Execution

Rely on map and reduce operations

1. Partition the vertices across \( w \) workers
2. Apply estimator task on each subgraph to produce a partial count
3. Sum up partial counts
4. Adjust for underestimation by multiplying \( f(w) \)
e.g. for triangle count, \( f(w) = \frac{1}{w^2} \)

![Diagram showing map and reduce operations](image)
Rely on map and reduce operations

1. Partition the vertices across $w$ workers
2. Apply estimator task on each subgraph to produce a partial count
3. Sum up partial counts
4. Adjust for underestimation by multiplying $f(w)$
   e.g. for triangle count, $f(w) = w^\uparrow 2$

- Patterns across partitions are ignored
- Total occurrence is reduced by $1/f(w)$
ASAP Architecture

1. graphA.patterns("a->b->c", "100s")
2. Generalized Approximate Pattern Mining
3. graphB.fourClique("5.0\%", "95.0\%")
4. Estimator Count Selection
5. Error-Latency Profile (ELP) Building
6. Estimates: {error: <5\%, time: 95s}
7. Estimates: {error: <5\%, time: 60s}

Embeddings (optional)

Graphs stored on disk or main memory

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

Graph updates
Error-Latency Profile (ELP)

ASAP can perform tasks in two modes:
- Time budget $T$
- Error budget $\epsilon$

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

Running time scales **linearly** with number of estimators
Test exponentially spaced points + extrapolation to build a linear model
**Error-Latency Profile (ELP)**

Chernoff bound for triangle counting: \( N \leq K \times m \times \Delta / \epsilon^2 P \)

Estimate ground truth \( P \) on a small sample of the graph + scale to \( P \)
Evaluation

77x speedup with under 5% loss of accuracy for smaller graphs (0.01-30 million edges)
Evaluation

258x speedup with under 5% loss of accuracy for larger graphs
Conclusion

ASAP is the first system that does fast, scalable approximate graph pattern mining on large graphs.

ASAP outperforms Arabesque by more than a magnitude faster with a sacrifice of 5% accuracy.

ASAP scales to larger graphs whereas Arabesque fails to complete execution.
Reference

- [https://www.usenix.org/sites/default/files/conference/protected-files/osdi18_slides_iyer.pdf](https://www.usenix.org/sites/default/files/conference/protected-files/osdi18_slides_iyer.pdf)