Arabesque

A system for distributed graph mining

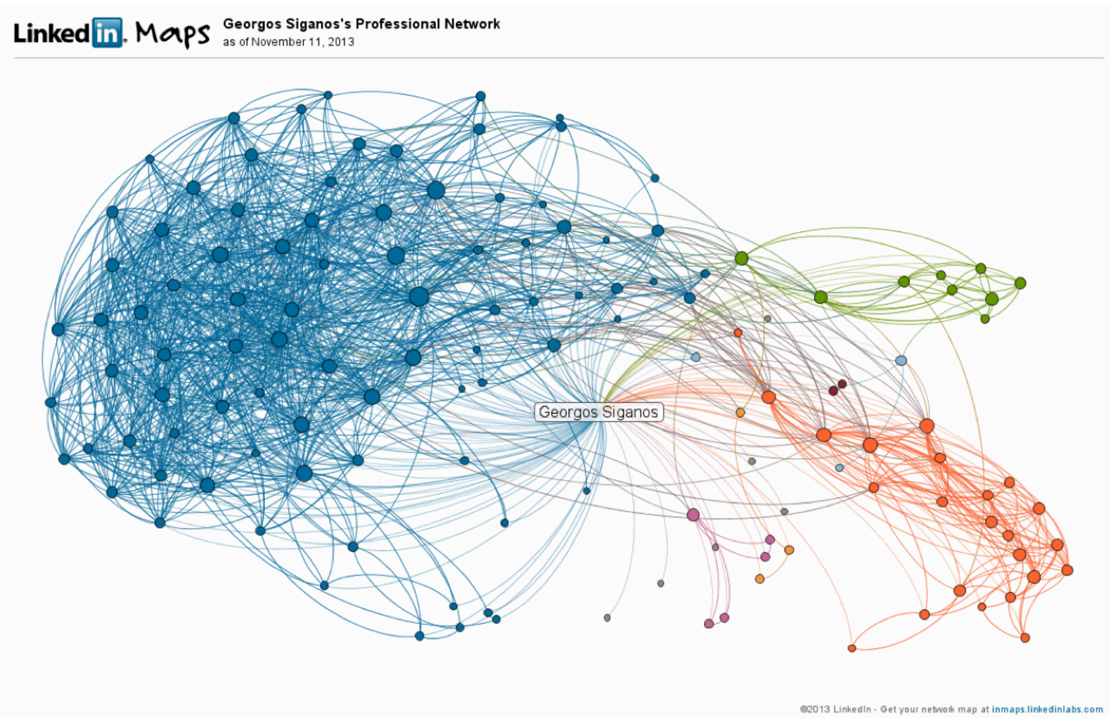
Mohammed Zaki, RPI

Carlos Teixeira, Alexandre Fonseca, Marco Serafini, Georgos Siganos,
Ashraf Aboulnaga, Qatar Computing Research Institute (QCRI)

Big Data

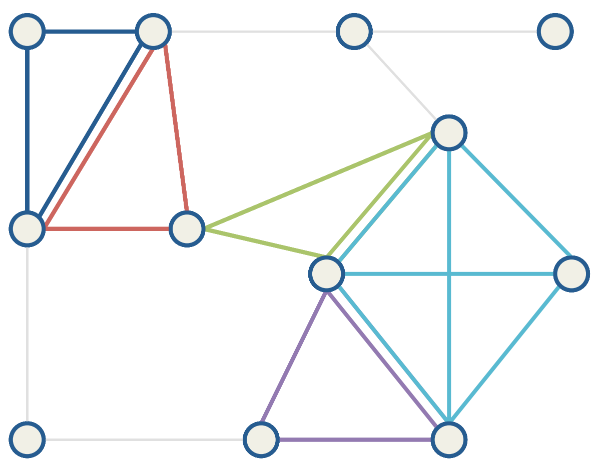
- Why has data analytics become so hot?
  - Physical and digital worlds increasingly intertwined
  - More and more digital breadcrumbs
  - More and more applications
  - Hadoop has made data analytics **accessible**
- **Key drivers in systems research**
  - Define **abstractions** that ease development
  - Systems that **efficiently** implement them

Graphs are Ubiquitous



Graph Mining Algorithms

- Finding subgraphs of interest in (labeled) input graphs
- Examples: Clique finding

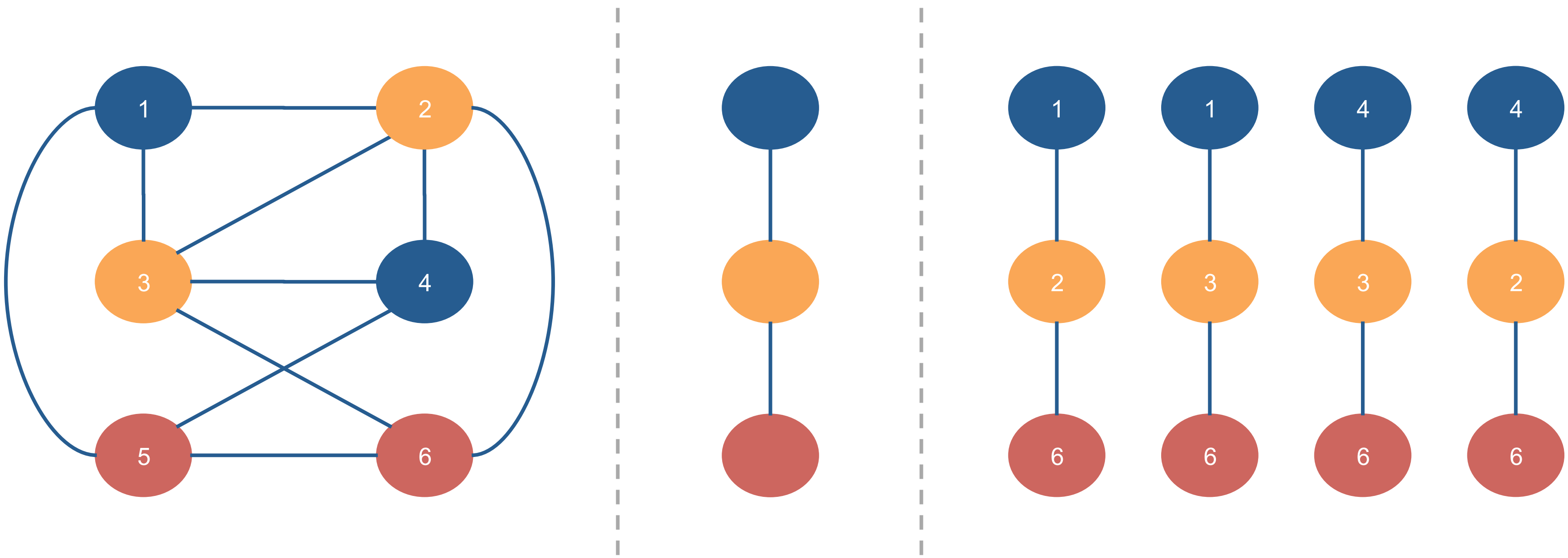


- Others: frequent subgraph mining, motifs

Applications

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

Some Terminology

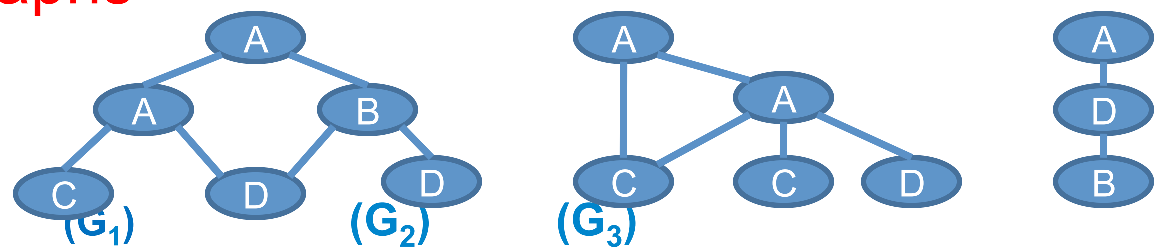


Input graph Pattern Embeddings

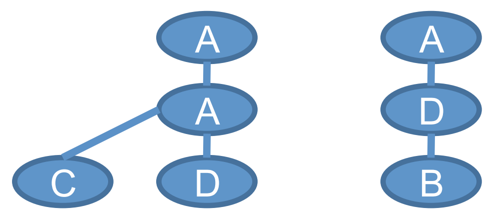
Example: Frequent Graph Mining

Frequent Subgraph Discovery

- Mining frequent subgraphs from a database of many graphs

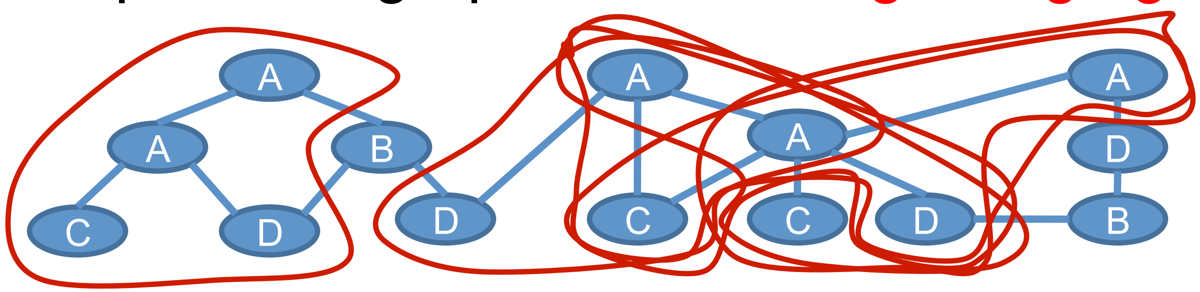


- Maximal Frequent Subgraphs with minimum support (minsup) = 2

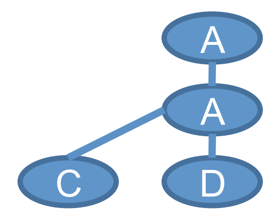


Frequent Subgraph Discovery

- Mining frequent subgraphs from a single large graph

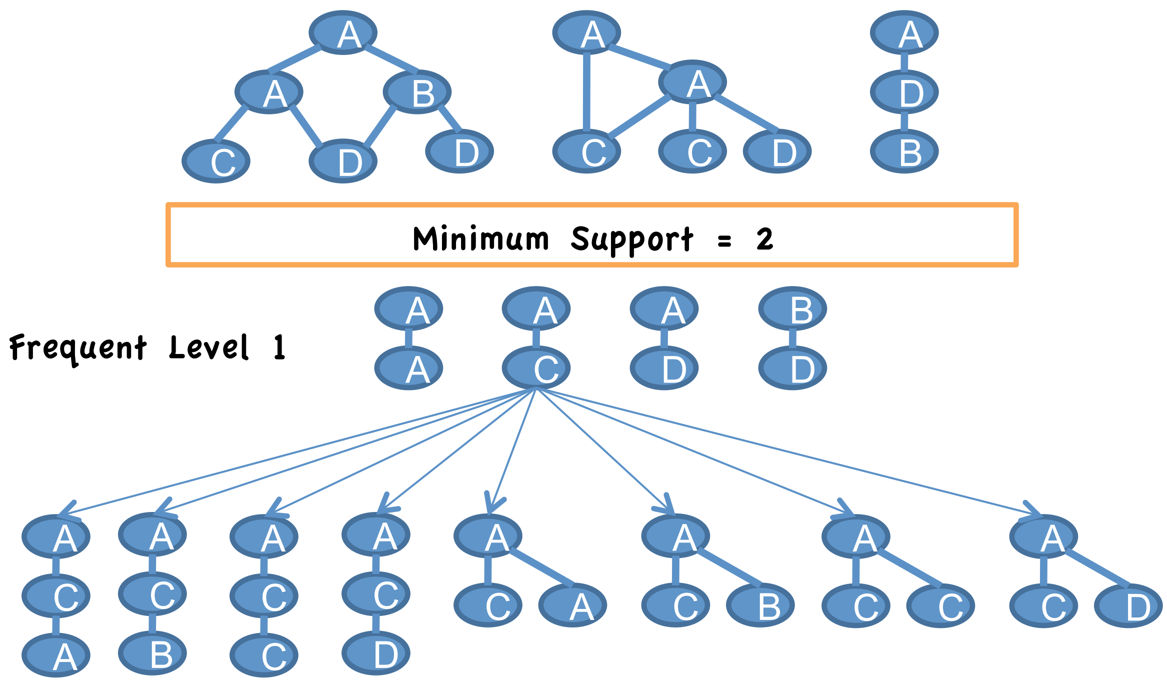


- Find subgraphs that have a minimum embedding count
  - Total (6)
  - Edge Disjoint (3)
  - Vertex Disjoint (2)
  - NP-Hard to find edge/vertex disjoint from total



Subgraph Mining: Complete Level-wise Search

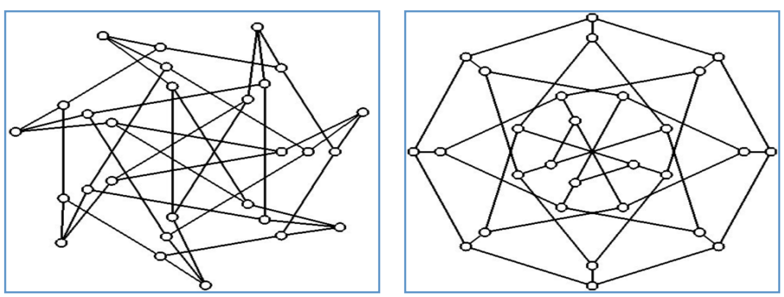
- **Candidate generation**: add one more edge; enumerate all extensions
- **Support counting**: check which are frequent; retain for next iteration

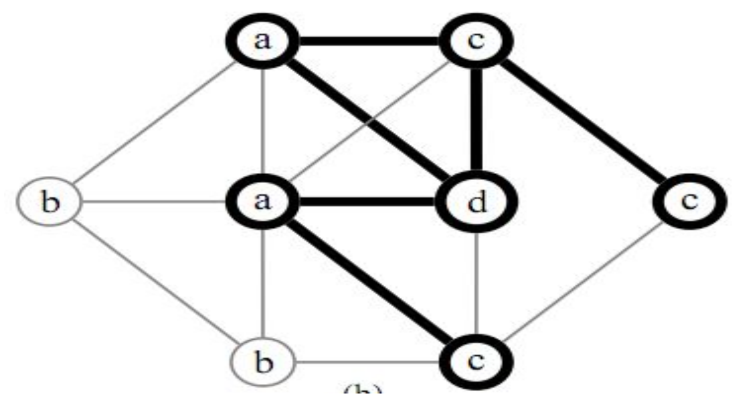


Candidate Level 2

Taming of the Morphisms

- Challenge of isomorphisms
- How to detect duplicates?
    - Graph Isomorphism

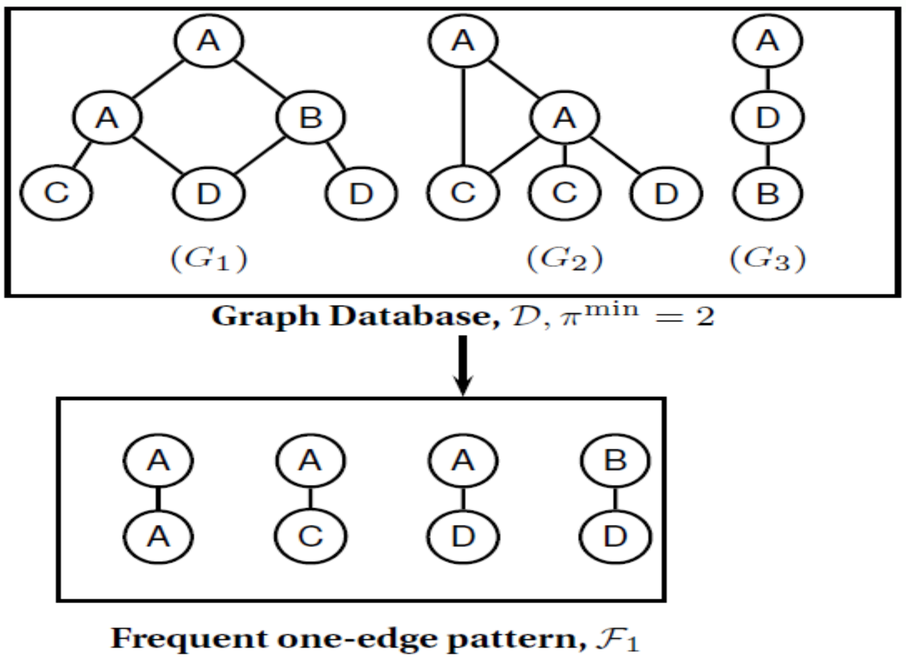


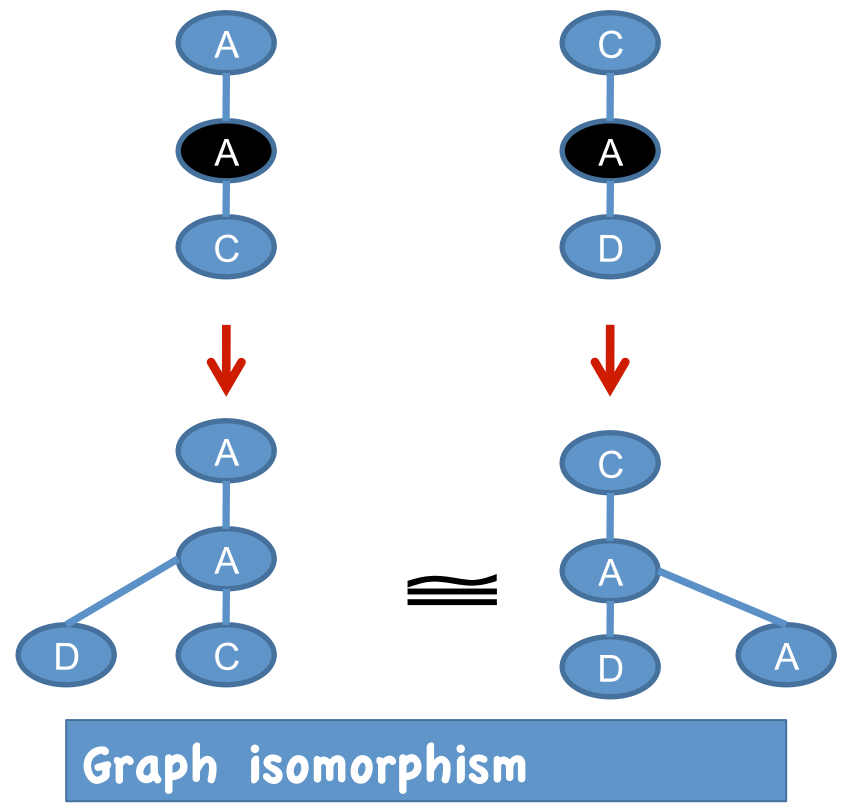


- How to count occurrences?
    - Subgraph Isomorphism

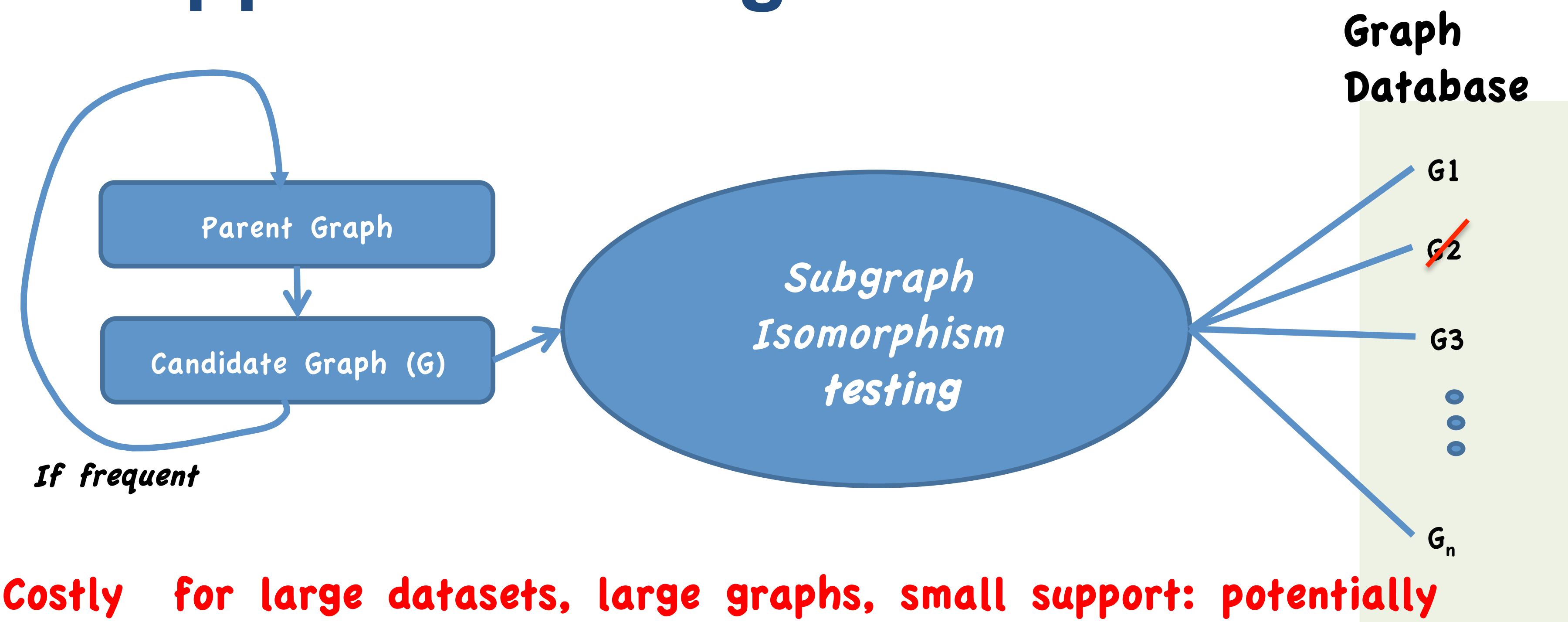
Candidate Generation

Can be very expensive: potentially millions of isomorphism checks





Support Counting

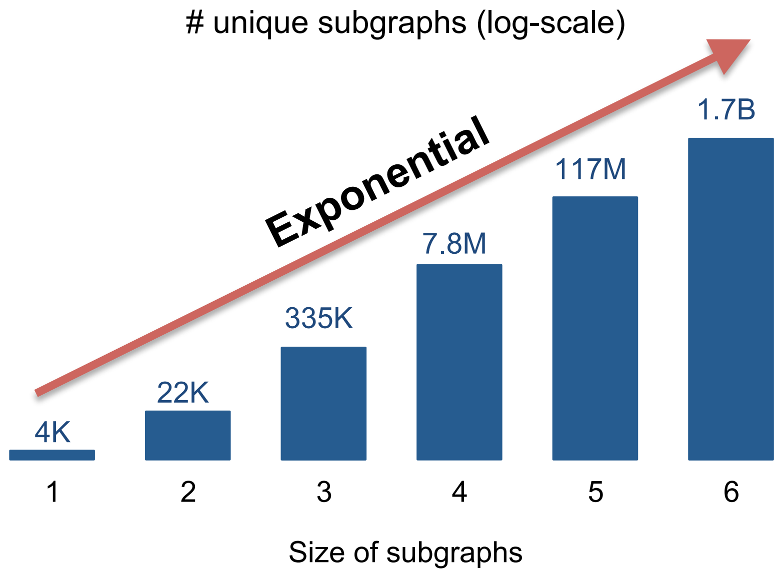


Costly for large datasets, large graphs, small support: potentially millions of subgraph isomorphism checks

Arabesque for Graph Mining

Challenge

- Exponential number of subgraphs/embeddings



State of the Art: Custom Algorithms

	Easy to Code	High Performance	Transparent Distribution
Custom Algorithms	✗	✓	✗

State of the Art: Think Like a Vertex

	Easy to Code	Efficient Implementation	Transparent Distribution
Custom Algorithms	✗	✓	✗
Think Like a Vertex	✗	✗	✓

APACHE GIRAPH

GraphLab

GraphX

Arabesque

- New system & execution model
  - Purpose-built for graph mining
  - New “Think Like an Embedding” model

- Contributions:
  - Simple & Generic API
  - High performance
  - Distributed & Scalable by design

Arabesque

	Easy to Code	High Performance	Transparent Distribution
Custom Algorithms	✗	✓	✗
Think Like a Vertex	✗	✗	✓
Arabesque	✓	✓	✓

Arabesque API - Clique finding

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

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

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

State of the Art
(Mace, centralized)

<u>4,621 LOC</u>

Arabesque API - Motif Counting

```
boolean filter(Embedding e) {
    return e.getNumVertices() <= MAX_SIZE;
}

void process(Embedding e) {
    mapOutput(e.getPattern(), 1);
}

Pair<Pattern, Integer> reduceOutput(Pattern p, List<Integer> counts) {
    return new Pair(p, sum(counts));
}
```

State of the Art
(GTrieScanner, centralized)

3,145 LOC

Arabesque API - Frequent Subgraph mining

- Ours was the first distributed implementation
- 280 lines of Java code…
    - … of which 212 compute frequency metric
- Baseline (GRAMI): 5,443 lines of Java

Arabesque: An Efficient System

- COST: As efficient as centralized state of the art

Application - Graph	Centralized Baseline	Arabesque 1 thread
Motifs - MiCo (MS=3)	50s	37s
Cliques - MiCo (MS=4)	281s	385s
FSM - CiteSeer (S=300)	4.8s	5s

Arabesque: A Scalable System

- Scalable to thousands of workers
- Hours/days → Minutes

Application - Graph	Centralized Baseline	Arabesque 640 cores
Motifs - MiCo	2 hours 24 minutes	25 seconds
Cliques - MiCo	4 hours 8 minutes	1 minute 10 seconds
FSM - Patents	> 1 day	1 minute 28 seconds

- Can process graphs with almost **1 billion** edges

Alternative Paradigms?

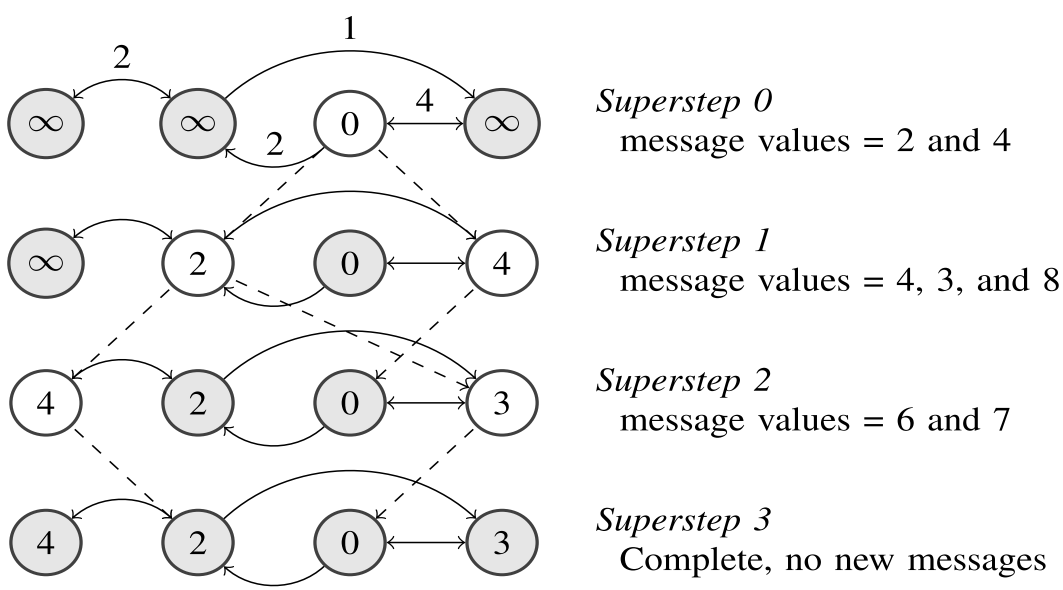
Think Like a Vertex

- Application = Stateful vertex object
- Vertices sends messages to their neighbors
- Easy to scale to large graphs: partition by vertex

- Bulk Synchronous Programming (BSP)
    1. Receive from all neighbors
    2. Compute new state
    3. Send to all neighbors

Example: Shortest Path

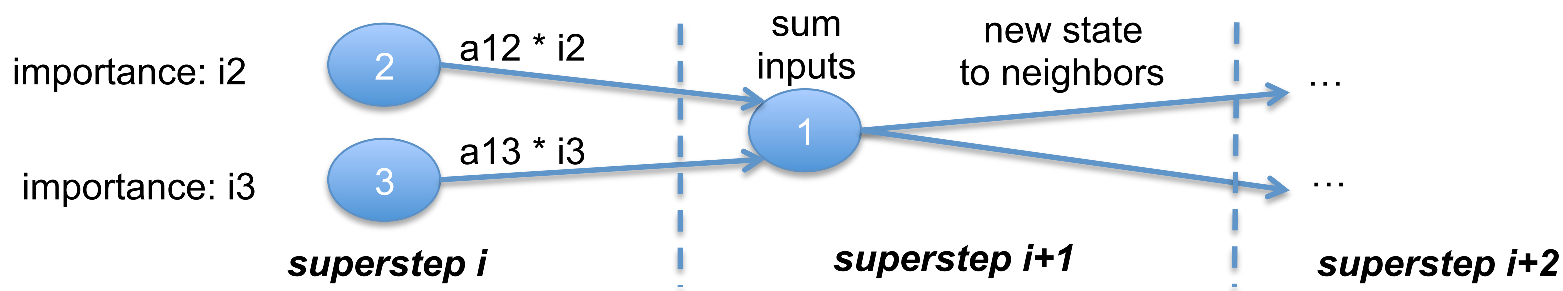
- **Input:** Graph (weighted edges), source vertex
- **Output:** Min source – vertex distance



Example taken from: [McCune *et al.*, arxiv:1507.04405 (2015)]

Matrix-Vector Multiplication

- E.g. Page-Rank style computation



links to v1

0	a12	a13
…	…	…

adjacency matrix (transposed)

*

i1
i2
i3

importance

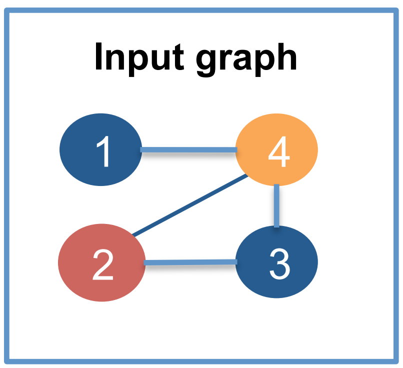
=

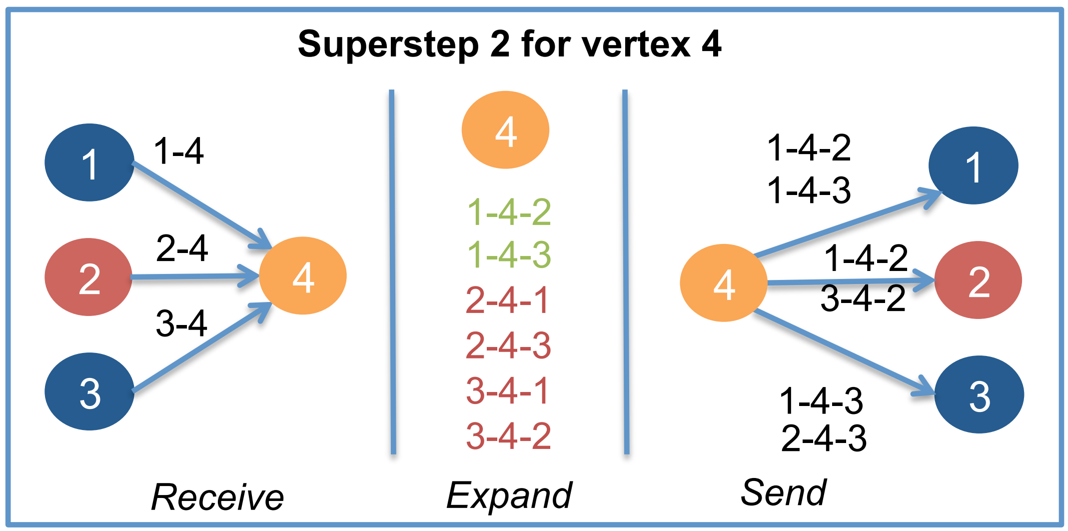
a12 * i2 + a13 * i3
…
…

new importance

Graph Exploration with TLV

1. Receive embeddings
2. Expand by adding neighboring vertices
3. Send *canonical* embeddings to their constituting vertices





Think Like a Pattern

- Many existing algorithms keep state by pattern
- Advantages
  - Rebuild embeddings from scratch
  - No need to materialize full intermediate state
- Idea of TLP:
  - Assign different patterns to different machines
  - Avoid storing materialized embedding

Arabesque Details

How: Arabesque Optimizations

- Avoid Redundant Work
  - Efficient canonicality checking
- Embedding Compression
  - Overapproximating Directed Acyclic Graphs (ODAGs)
- Efficient Aggregation
  - 2-level pattern aggregation

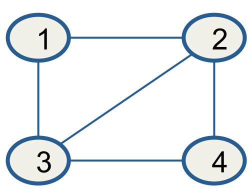
Arabesque: Fundamentals

- Subgraphs as 1st class citizens:
  - **Embedding** == Subgraph
  - **Think Like an Embedding** model

Arabesque responsibilities		User responsibilities
Graph Exploration	Aggregation (Isomorphism)	Filter
Load Balancing	No redundant work (Automorphism)	Process

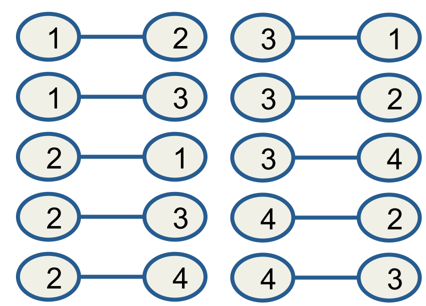
Graph Exploration

- Iterative expansion
  - Subgraph order n → Subgraph order n + 1
  - Connect to neighbours, one vertex at a time.



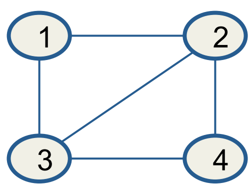
Input graph

Depth 1

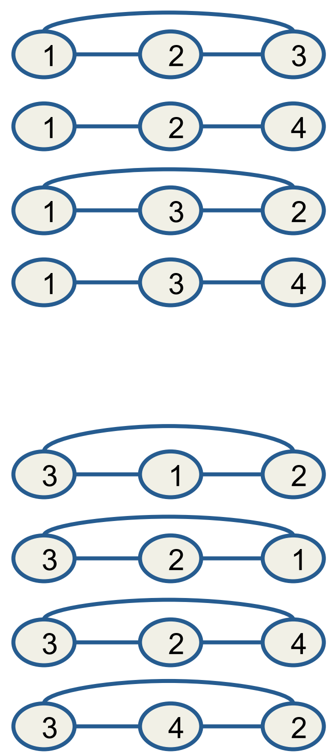


Depth 2

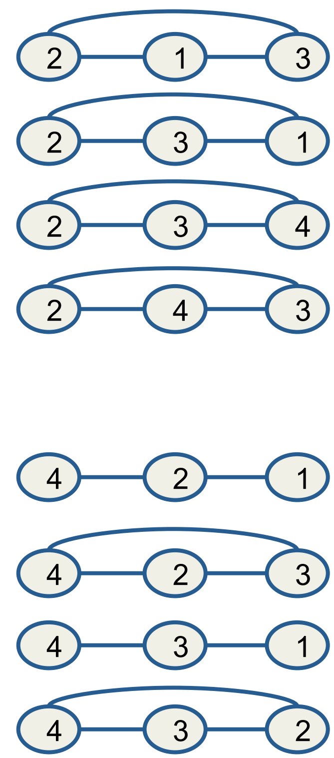
Graph Exploration



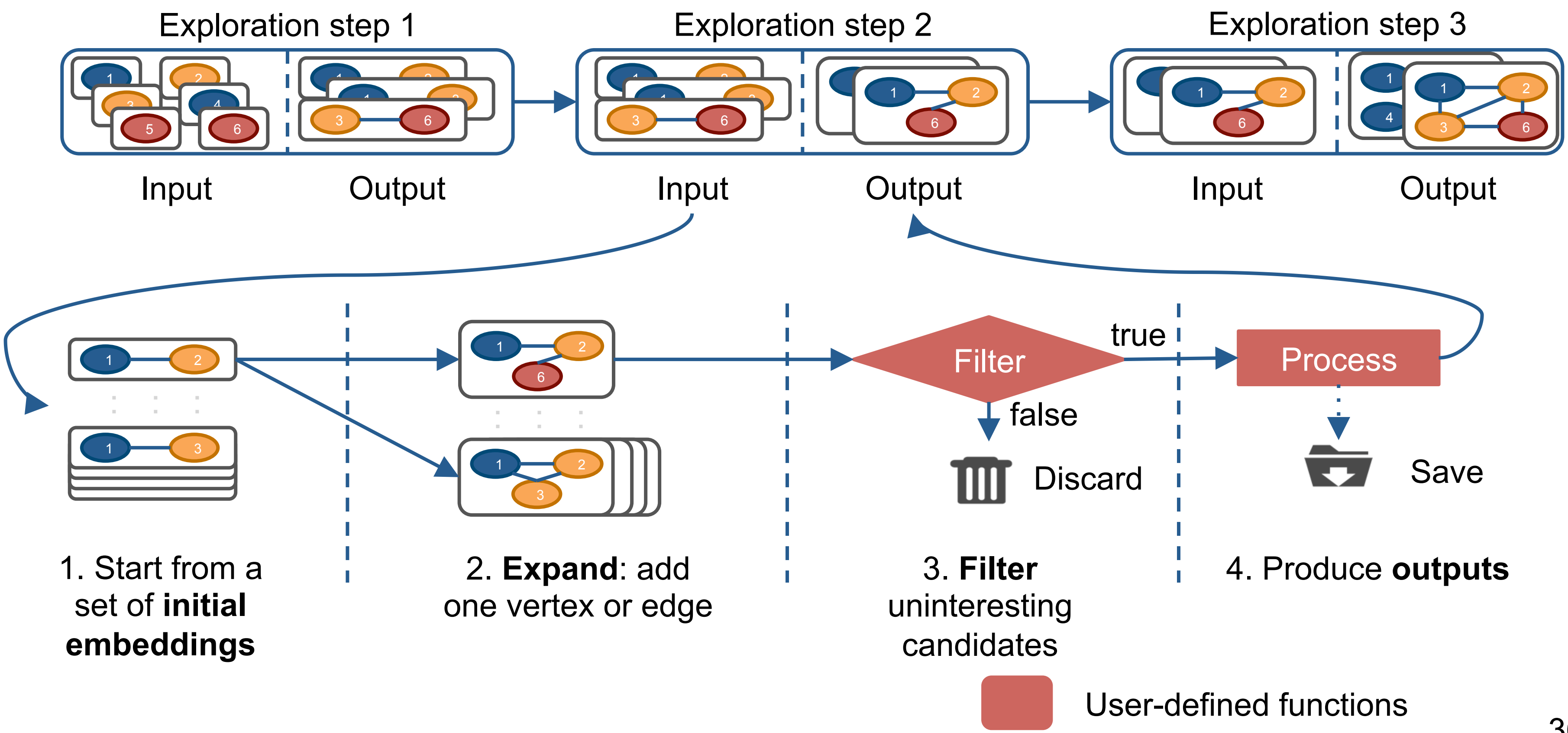
Input graph



Depth 3



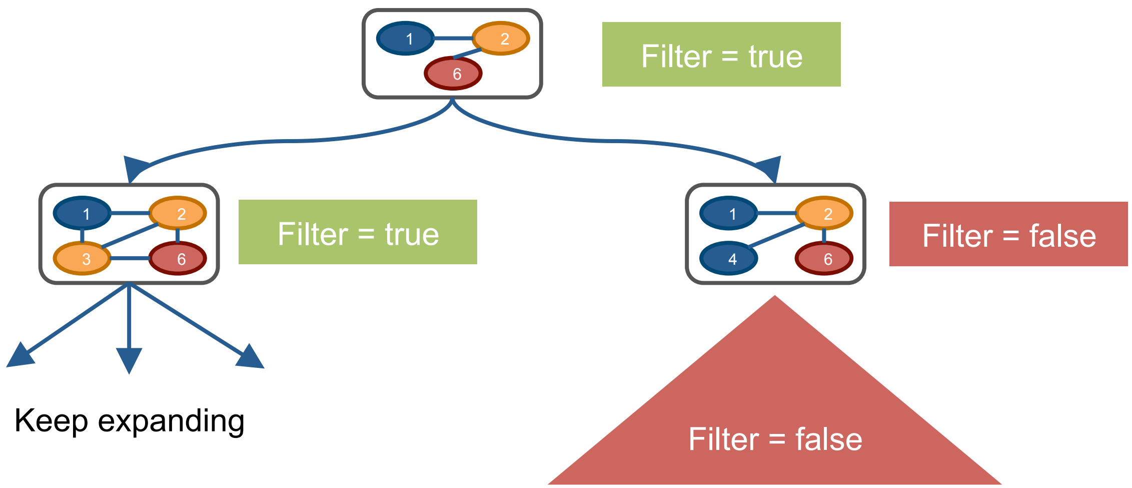
Model - Think Like an Embedding



Guarantee: Completeness

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

Requirement: <u>Anti-monotonicity</u>



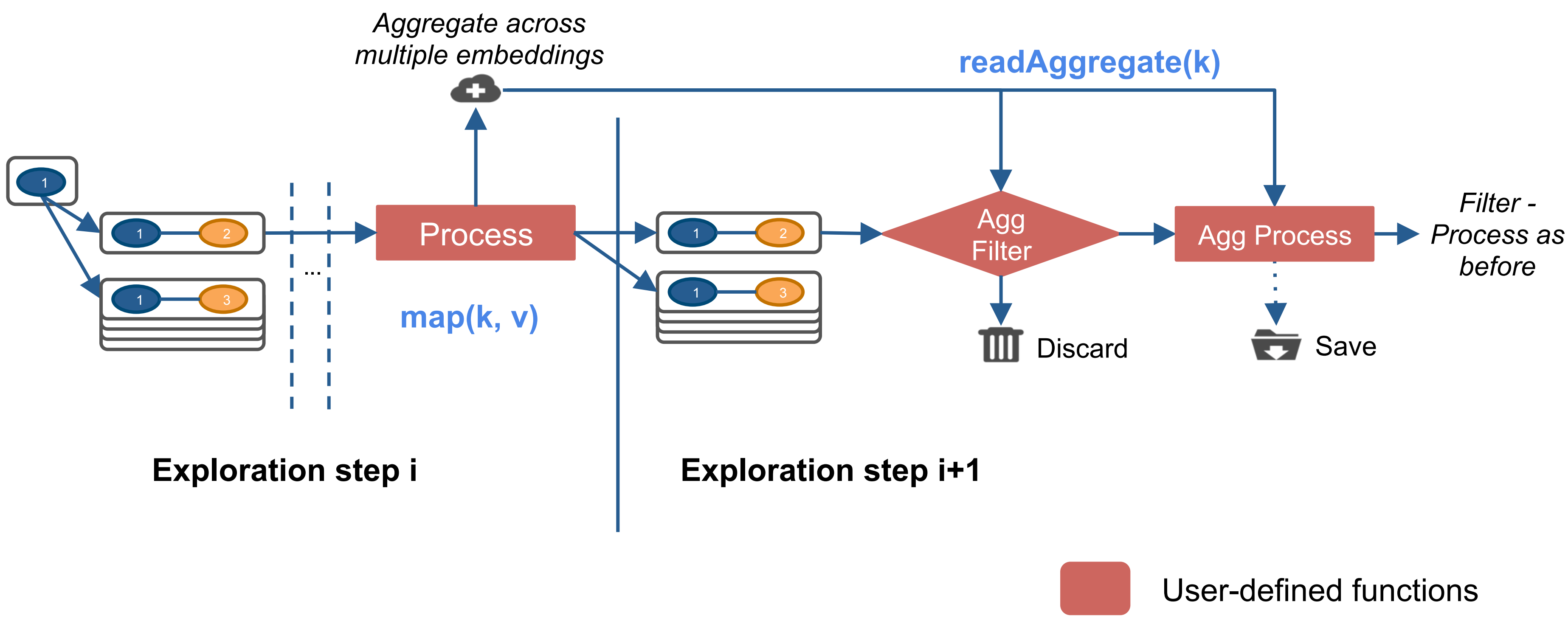
We can **prune** and be sure that we won't ignore desired embeddings

Aggregation

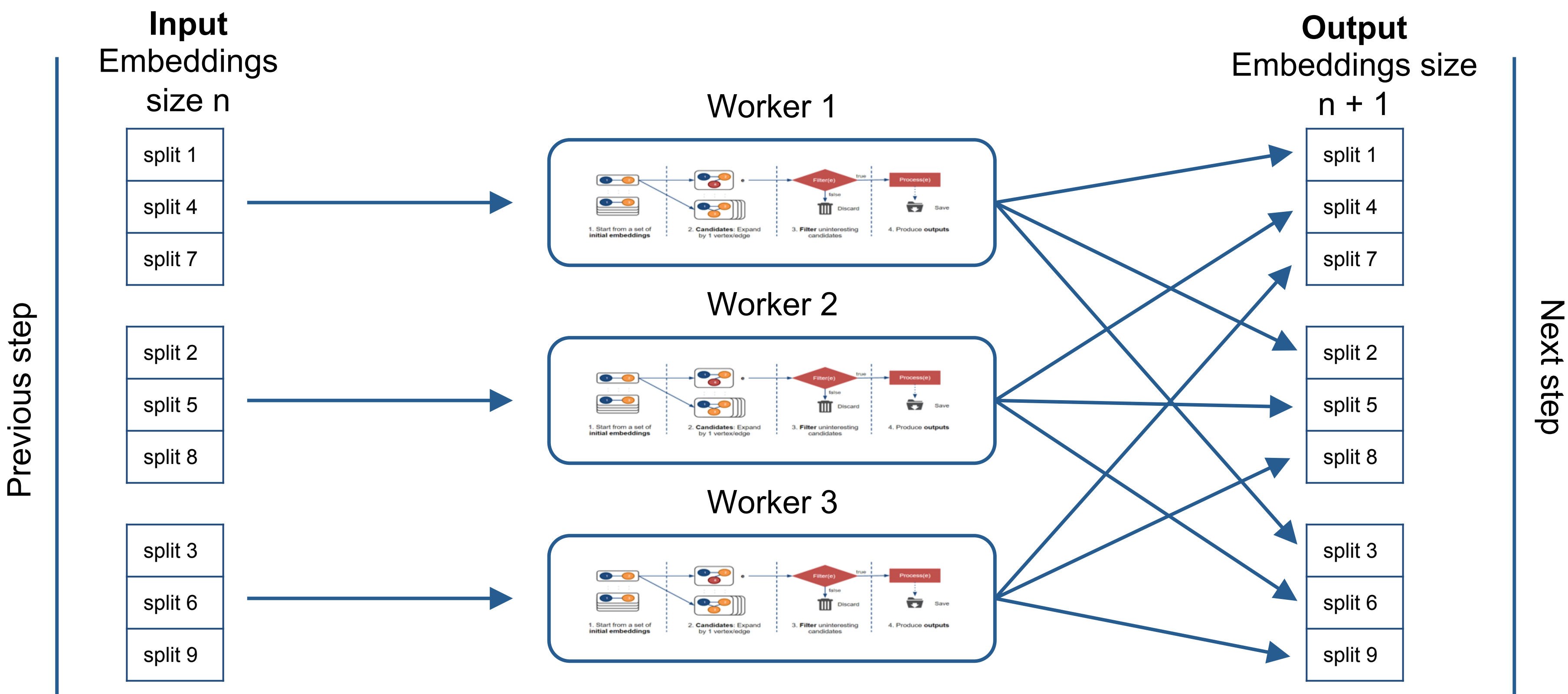
- Some applications must aggregate across embeddings
  - E.g., Frequent subgraph mining: Count embeddings with same pattern

- Aggregation in parallel with exploration step

Aggregation



System Architecture



Arabesque API

- **App-defined functions:**
  - `boolean filter(Embedding e)`
  - `void process(Embedding e)`

  - `boolean aggregationFilter(Embedding e)`
  - `void aggregationProcess(Embedding e)`

  - `Pair<K,V> reduce(K key, List<V> values)`
  - `Pair<K,V> reduceOutput(K key, List<V> values)`

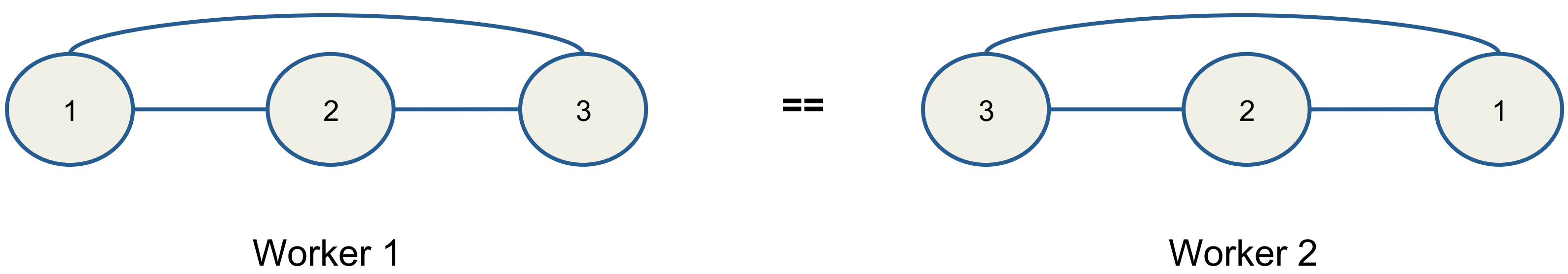
- **Functions provided by Arabesque:**
  - `void map(K key, V value)`
  - `V readAggregate(K key)`

  - `void output(Object value)`
  - `void mapOutput(K key, V value)`

Technical Challenges

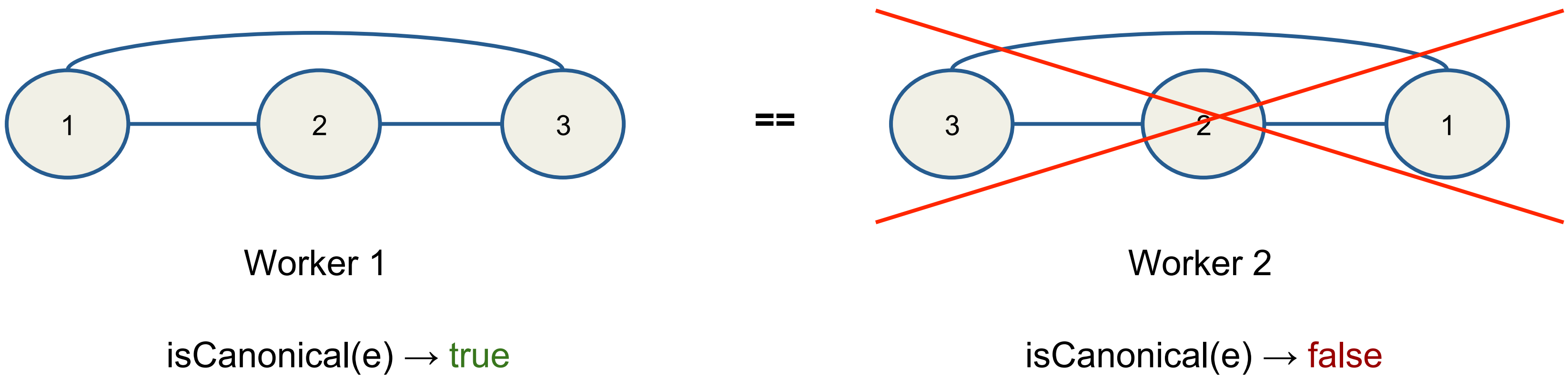
Avoiding redundant work

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



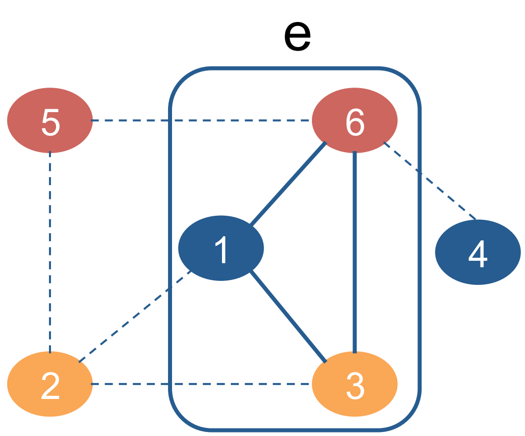
Avoiding redundant work

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



Embedding Canonicality

- isCanonical(e) *iff* at every step add neighbor with smallest ID



Initial embedding (e)
- 1 - 3 - 6

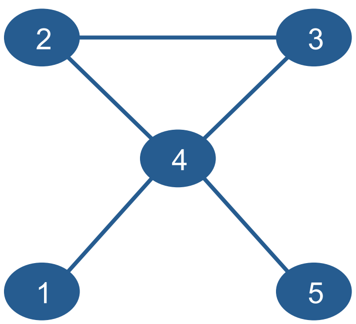
Expansions:
- 1 - 3 - 6 - 5 → canonical
- 1 - 3 - 6 - 4 → canonical

- 1 - 3 - 6 - 2 → not canonical (1 - 2 - 3 - 6)

Handling Exponential growth

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

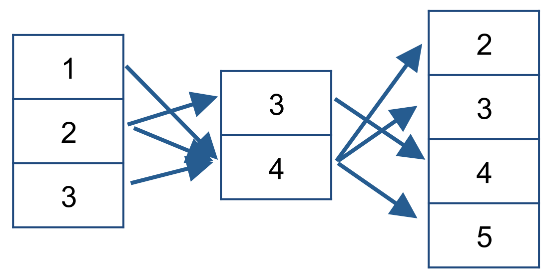
- **Solution: Overapproximating DAGs (ODAGs)**
  - Compress into less restrictive superset
  - Deal with spurious embeddings



Input Graph

Canonical Embeddings		
1	4	2
1	4	3
1	4	5
2	3	4
2	4	5
3	4	5

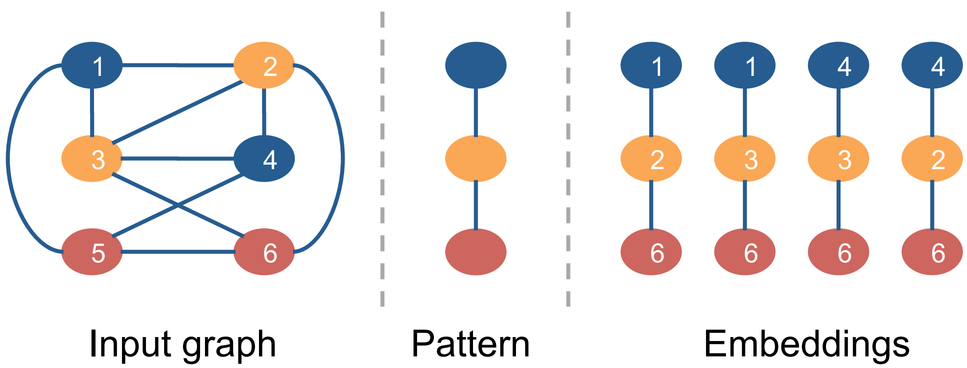
Embedding List



ODAG

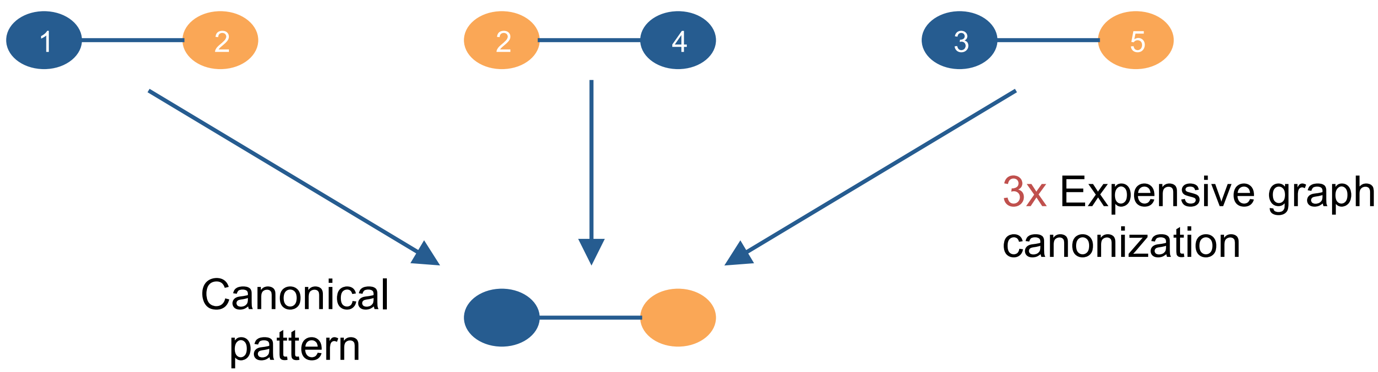
Aggregation by Pattern

- Label
  - Distinguishable property of a vertex (e.g. color).
- Pattern - “Meta” sub-graph or the template.
  - Captures subgraph structure and labelling
- Embedding - Instance of a pattern.
  - Actual vertices and edges



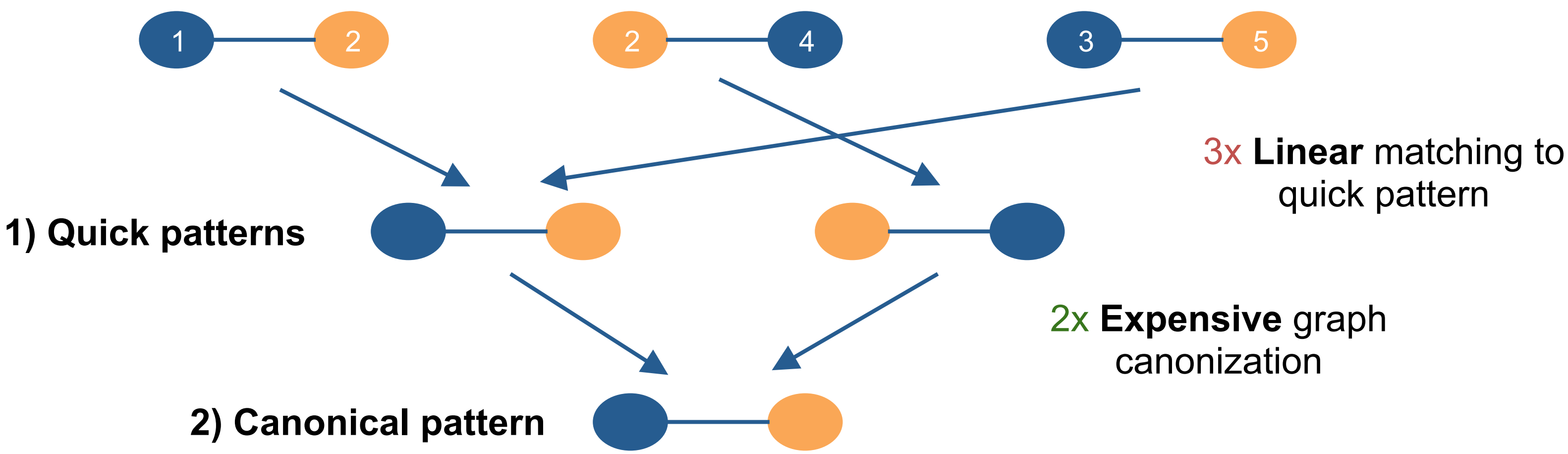
Efficient Pattern Aggregation

- **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



Evaluation

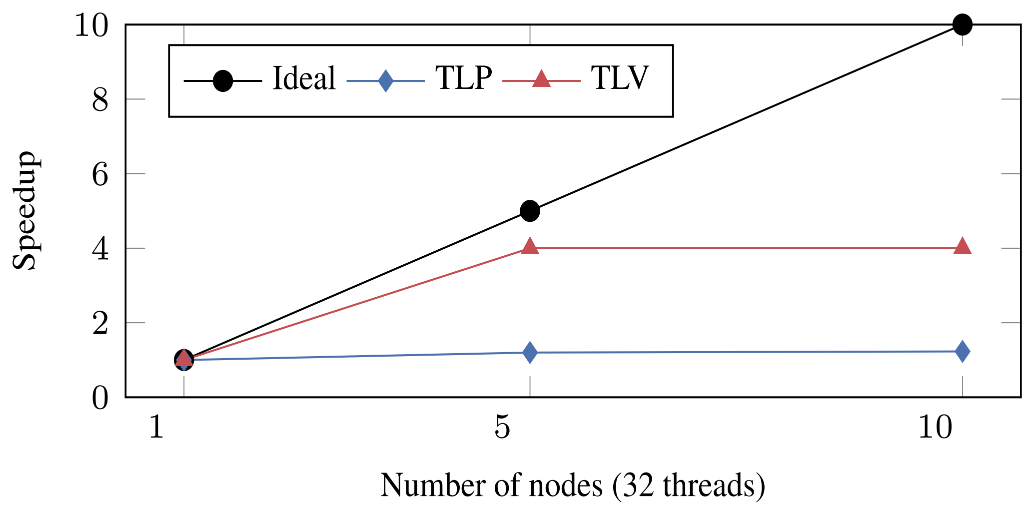
Evaluation - Setup

- 20 servers: 32 threads @ 2.67 GHz, 256GB RAM
- 10 Gbps network
- 3 algorithms: Frequent Subgraph Mining, Counting Motifs and Clique Finding

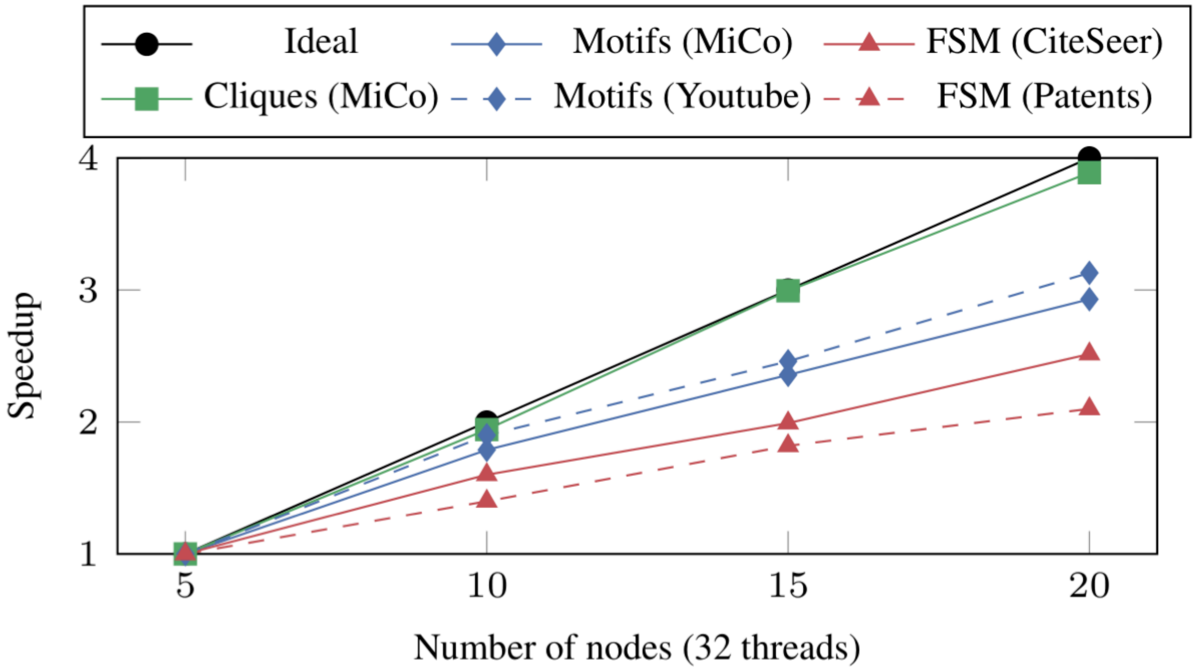
	# Vertices	# Edges	# Labels	Avg. Degree
CiteSeer	3,312	4,732	6	2.8
MiCO	100,000	1,080,298	29	21.6
Patents	2,745,761	13,965,409	37	10
Youtube	4,589,876	43,968,798	80	19
SN	5,022,893	198,613,776	0	79
Instagram	179,527,876	887,390,802	0	9.8

Evaluation - TLP & TLV

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



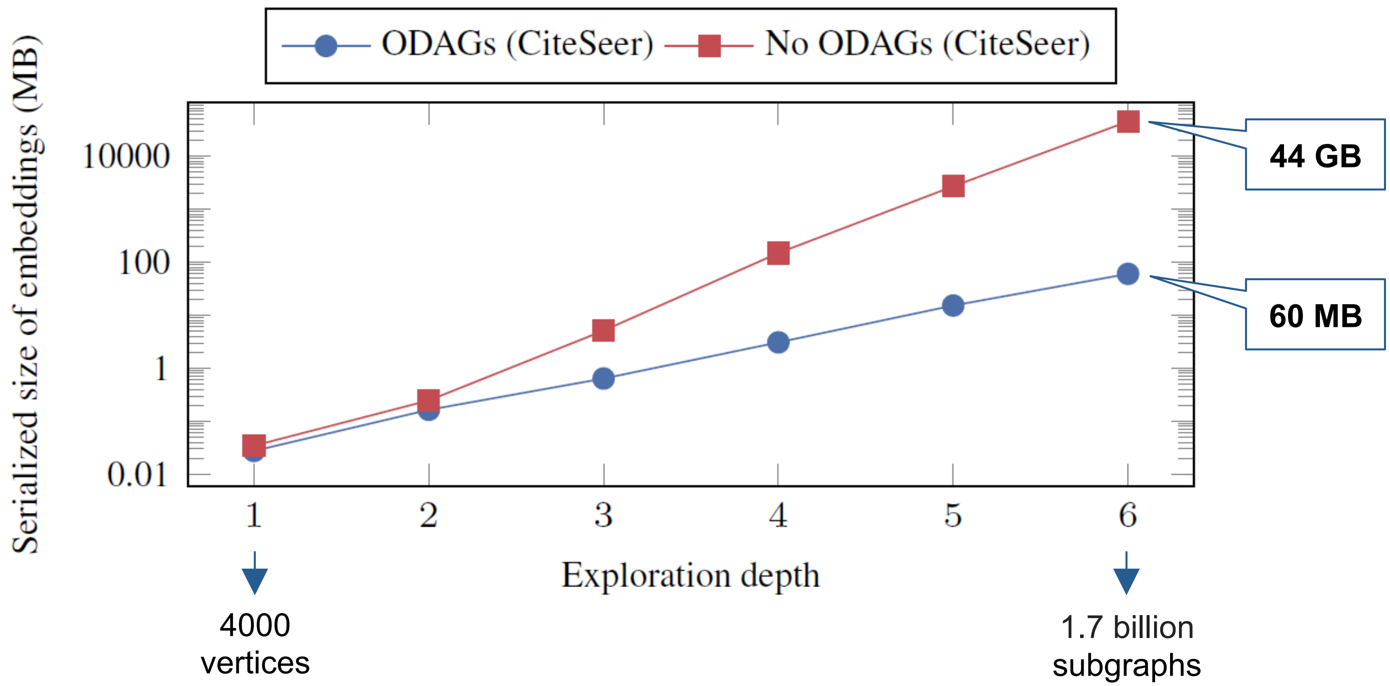
Evaluation - Araquesque Scalability



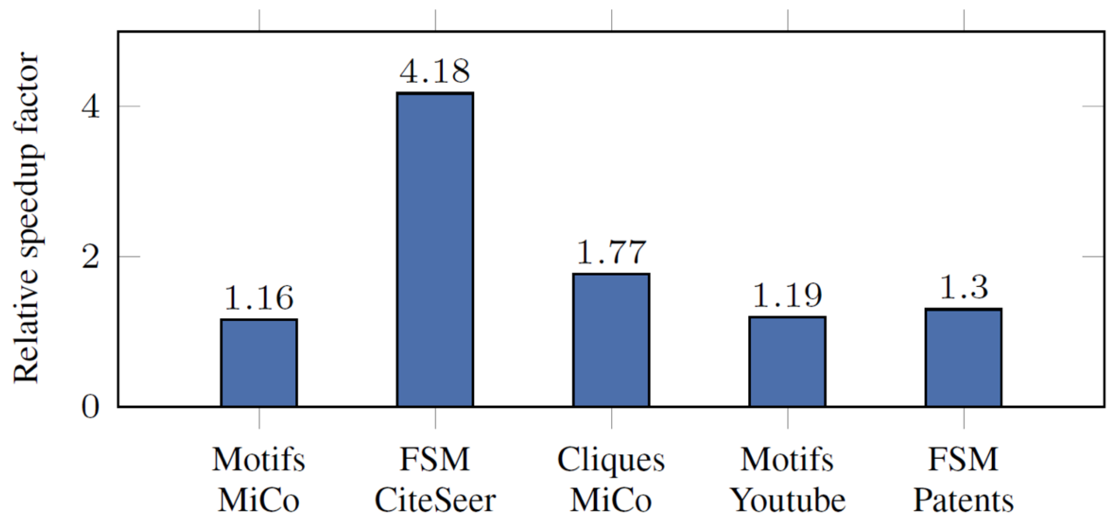
Evaluation – Arabesque Scalability

Application - Graph	Centralized Baseline	Arabesque - Num. Servers (32 threads)				
		1	5	10	15	20
Motifs - MiCo	8,664s	328s	74s	41s	31s	25s
FSM - Citeseer	1,813s	431s	105s	65s	52s	41s
Cliques - MiCo	14,901s	1,185s	272s	140s	91s	70s
Motifs - Youtube	Fail	8,995s	2,218s	1,167s	900s	709s
FSM - Patents	>19h	548s	186s	132s	102s	88s

Evaluation - ODAGs Compression

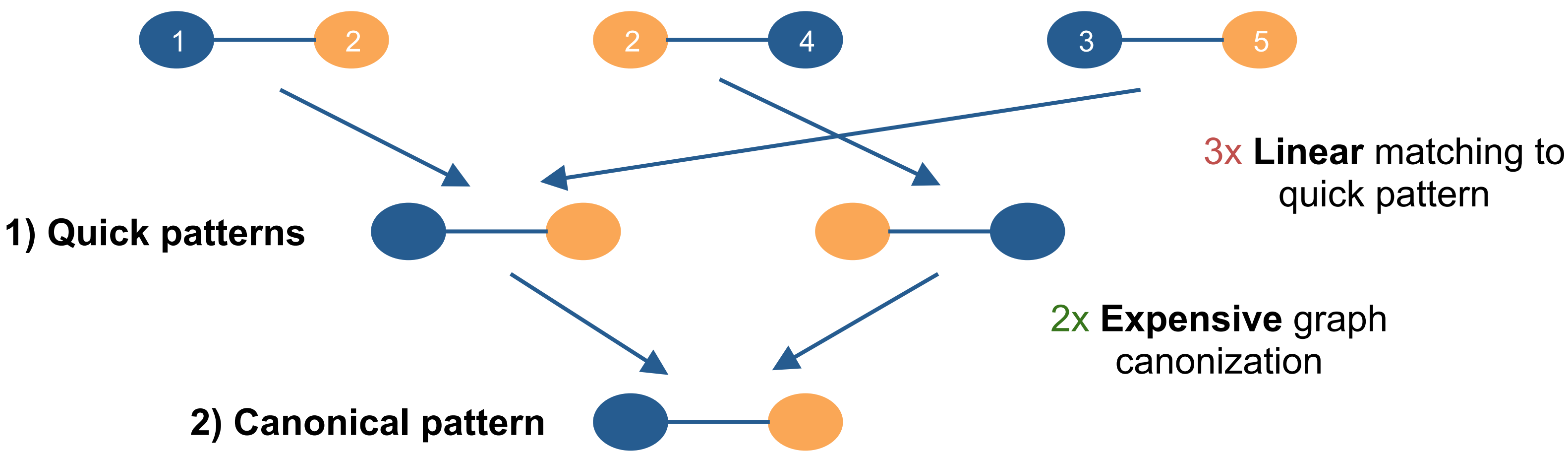


Evaluation - Speedup w ODAGs



Efficient Pattern Aggregation

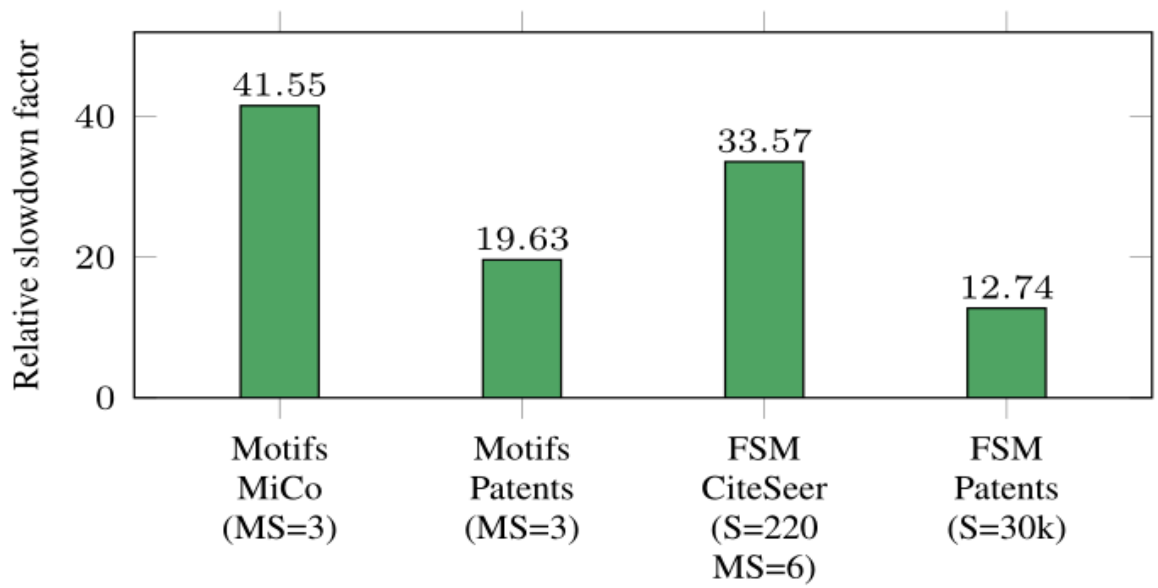
- **Solution:** 2-level pattern aggregation
  1. Embeddings → quick patterns
  2. Quick patterns → canonical pattern



Evaluation - Two-level aggregation

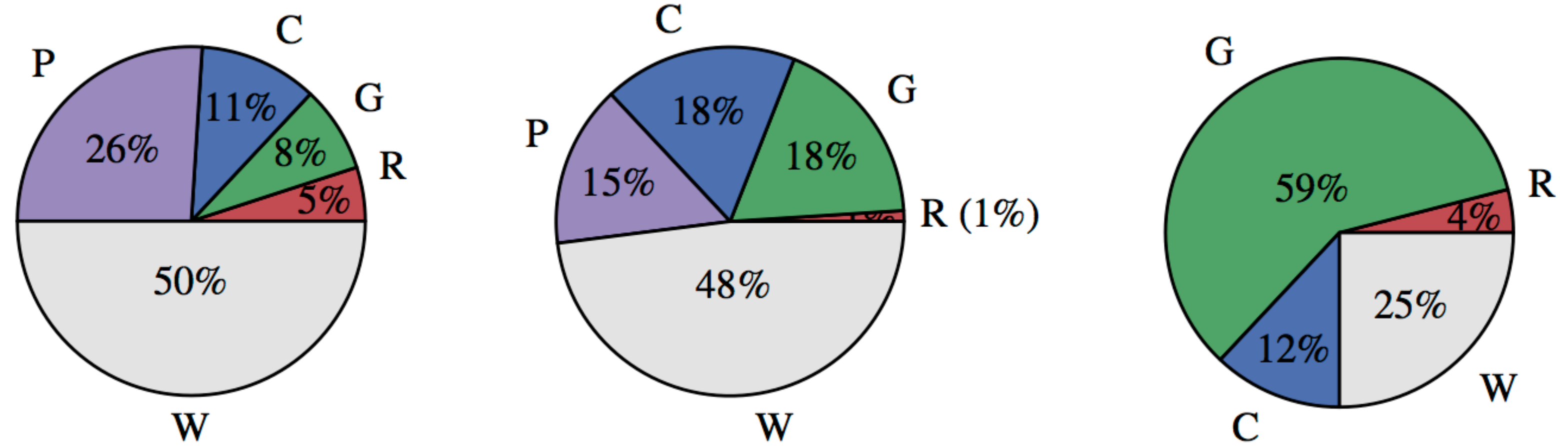
	Motifs MiCo (MS = 4)	Motifs Youtube (MS=4)	FSM CiteSeer (S=220, MS=7)	FSM Patents (S=24k)
Embeddings	10,957,439,024	218,909,854,429	1,680,983,703	1,910,611,704
Quick Patterns	21	21	1433	1800
Canonical Patterns	6	6	97	1348
Reduction Factor	521,782,810x	10,424,278,782x	1,173,052x	1,061,451x

Evaluation - Two-level aggregation



CPU Utilization Breakdown

- Advantages of a simple API
  - Arabesque does **all** the work (unlike TLV system)
  - Great opportunities for system-level optimizations



(a) FSM CiteSeer(S=220,MS=7) (b) Motifs MiCo (MS=4) (c) Cliques

P: Pattern Aggregation, C: canonicality checks, G: generate new candidates, R/W: Read/write embeddings

Large Graphs

Graph	# Vertices	# Edges	# Labels	Avg. Degree
SN	5,022,893	198,613,776	0	79
Instagram	179,527,876	887,390,802	0	9.8

Application	Time	Embeddings
Motifs-SN (MS=4)	6h 18m	8.4 trillion
Cliques-SN (MS=5)	29m	30 billion
Motifs-Instagram (MS=3)	10h 45m	5 trillion

What’s Next?

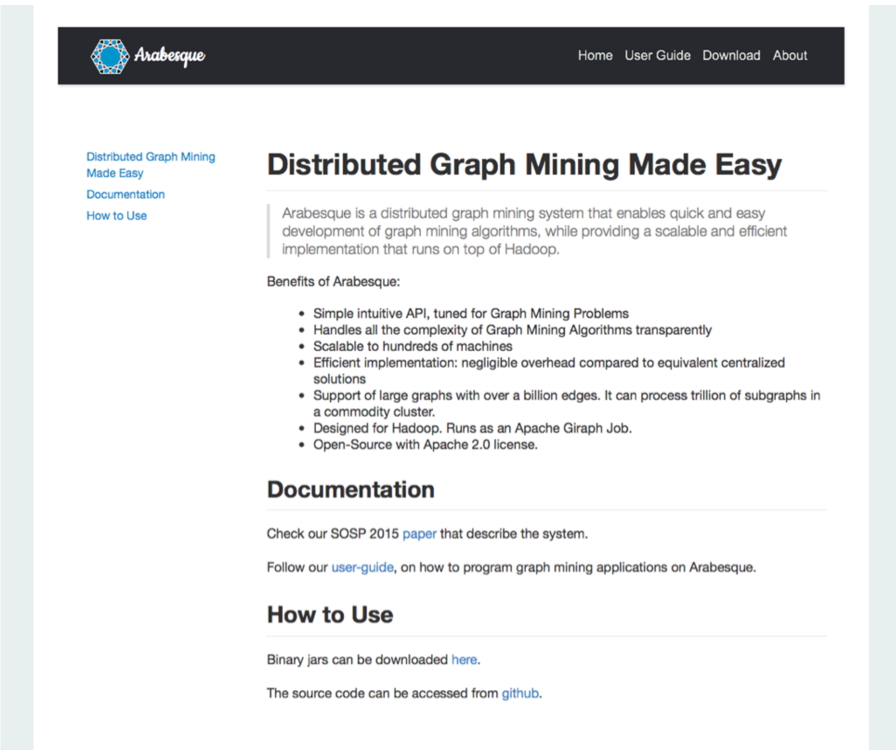
Future Work

- Better ways to organize intermediate state
  - Scale to larger intermediate states
  - Support for approximate exploration
  - Out-of-core?
- Support for real-time graphs
- Verticals and new applications

Conclusions

- Fundamental trend: democratizing data analytics
- Arabesque: graph mining system
  - Straightforward to code
  - Transparent and scalable distribution
  - High performance
- Only a first step: many opportunities for improvement

Download It, Play with It, Hack It



http://arabesque.io

- **Open-source (Apache 2.0)**
- Pre-compiled jar
- User guide

Thank you

arabesque.io