AutoMine: Harmonizing High-Level Abstraction and High Performance for Graph Mining

Daniel Mawhirter, Bo Wu

Presentation by: Abdullah Alomar

Outline

1. Introduction & Motivation
2. Overview of AutoMine
3. Key Features
4. Evaluation
5. Conclusions and Discussion

Introduction & Motivation

Introduction: State of current work

- Recently there have been many scalable frameworks designed for graph processing (e.g. Pregel). Most of these systems propose and follow certain programming paradigm (e.g. TLV) to implement graph computation algorithms (e.g. BFS, Page Rank).
- These frameworks are **unable** to handle more complicated workflows, namely **graph mining** workflows, as they involve much more complex algorithms and generate huge amounts of intermediate data. while the systems mentioned above can only maintain states of vertices/edges
- To overcome this, some frameworks has proposed maintaining the states of sub-graph embeddings such as **Arabesque** and **RStream**. The performance of these systems however is lacking performance is lacking.

How current systems perform?

In line with the previously presented paper, the authors compare the performance of RStream with a single threaded implementations. They find the single-threaded implementation outperforms RStream up to 5.7X speedup.

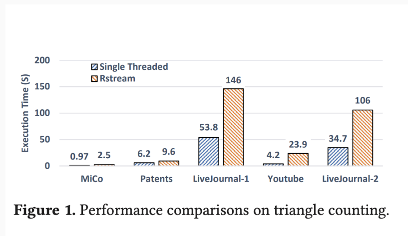

Figure 1. Performance comparisons on triangle counting.

A natural question that arises is: **why such degradation in performance happens?**

How current systems perform?

A natural question that arises is: **why such degradation in performance happens?**

Two Reasons:

1-Existing graph mining systems implement generic but low-efficiency mining algorithms.

In essence they sacrifice performance for high-level abstraction. e.g., triangle counting in Rstream takes $\mathcal{O}(|V|\Delta^2)$, which is higher than the simple counting algorithm ($\mathcal{O}(|E|\Delta)$).

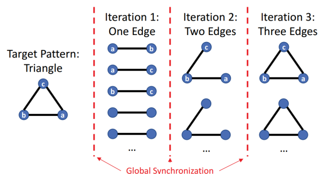

Figure 2. Embedding enumeration for triangle counting in RStream.

How current systems perform?

Two Reasons:

2-Existing graph mining systems have high memory consumption

Again, in triangle counting RStream needs to generate all the wedge embeddings before generating triangle embeddings.

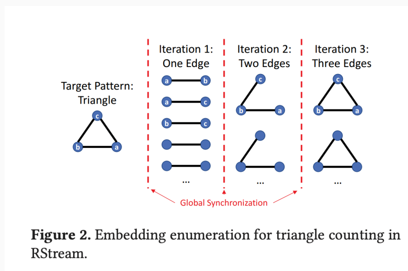

Figure 2. Embedding enumeration for triangle counting in RStream.

The Figure illustrates a possible solution to minimize memory consumption. To enumerate the triangle embedding (a, b, c), one needs only generate the wedge embedding (a, b, c) it depends on, instead of all the wedge embeddings.

Overview of AutoMine

AutoMine Objective

The goal of AutoMine is: **bridge the gap between high-level abstraction and high performance**.

Two **challenges** to achieve that:

1- How to automatically select an efficient algorithm from many possible algorithms to mine a particular pattern.

2- How to minimize memory consumption, and avoid the pitfall of other systems.

AutoMine Objective

For that, AutoMine come up with Three novel Ideas:

1- **Vertex composition set:** which save space and build a foundation of code generation.

2- **Schedule generator**, which transform the mining problem into a graph tournament problem.

3- **Merging algorithm** for counting multiple different patterns.

AutoMine architecture

Workflow: The workflow of the AutoMine system has the following phases:

(i) **Compilation Phase.** The compilation phase takes a high-level API and generates an optimized graph mining program.

(ii) **Execution Phase.** the mining program processes input graphs and returns the final results.

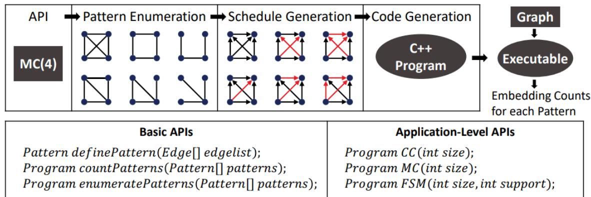

Figure 3. AutoMine architecture.

AutoMine architecture

(i) **Compilation Phase.**

The compilation phase takes a high-level API and generates an optimized graph mining program by invoking three components:

(1) **Pattern Enumerator:** Which transform the pattern semantics into an enumeration of all non-isomorphic subgraph patterns.

(2) **Schedule generator:** which select an optimal schedule (i.e., algorithm) to identify each of the subgraph patterns.

(3)**Code generator:** which considers data reuse in the generated schedules and produces the final mining program in C++.

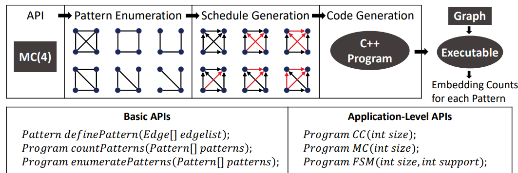

Figure 3. AutoMine architecture.

AutoMine architecture

APIs: AutoMine does not require the user to understand the mining algorithms or optimization details, but presents high-level APIs.

1 - **definePattern**: function defines a pattern with a list of 2-tuples, each representing an undirected edge. e.g., define a triangle pattern, by `Pattern p = definePattern([(a,b), (b, c), (c,a)])`.

2- **countPatterns** and **enumeratePatterns** to generate programs to respectively count and enumerate the embeddings of the given list of subgraph patterns.

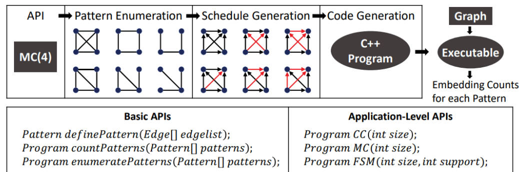

Figure 3. AutoMine architecture.

Summary of Results

In this paper, the authors empirically evaluate AutoMine and find that it indeed provides a system with high-level abstraction and superior performance. In particular AutoMine:

1- Outperforms Arabesque and RStream by orders of magnitude.

2- Even outperform the approximate system ASAP, a state-of-the-art approximate graph mining system, by up to 68.8X for size-3 motif counting

Key Features

Set-Based Representation

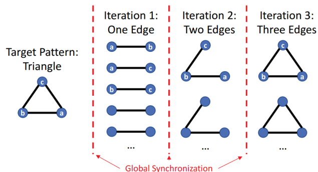

Figure 2. Embedding enumeration for triangle counting in RStream.

In the above algorithm, we keep all the intermediate embeddings, whereas in the the simple triangle counting algorithm allow for discarding the wedge embedding after all the more complex embeddings (i.e. triangles) are already discovered.

The authors suggest Set-Based representation to for any arbitrate pattern

Set-Based Representation

- Consider a connected pattern $\mathcal{P}_k$ on $k : k > 2$ vertices, and a sub-pattern $\mathcal{P}_{k-1}$. (e.g.$\mathcal{P}_3$ can be a triangle)
- An instance of $\mathcal{P}_k$ is an embedded sub-graph denoted as $\mathsf{E}_{\mathcal{P}_k}$ and composed of vertices $(v_0, ..., v_{k1})$ (e.g. $\mathsf{E}_{\mathcal{P}_3} = (a, b, c)$).
- Let the function $\mathcal{F}_k(\mathsf{E}_{\mathcal{P}_{k-1}})$ be a function that return a set V_k of all the vertices v_k that extend an embedding $\mathsf{E}_{\mathcal{P}_{k-1}}$ into an $\mathsf{E}_{\mathcal{P}_k}$.
- The function $\mathcal{F}_k$ must only apply set operations on the neighbor sets of $\mathsf{E}_{\mathcal{P}_{k-1}}$ vertices.

Lemma 1: *$\mathcal{F}_k$ can discover Vk by using only set intersection and subtraction.*

Set-Based Representation

Lemma 1: *$\mathcal{F}_k$ can discover Vk by using only set intersection and subtraction.*

Proof. 1- In order to construct $\mathcal{F}_k$, consider a vertex v_k which can form an embedding $\mathsf{E}_{\mathcal{P}_k}$ with the vertices from $\mathsf{E}_{\mathcal{P}_{k-1}}$

2-We partition $v_0, ..., v_{k1}$ into two sets V_T and V_F. V_T contains all the vertices that are neighbors of v_k in $\mathcal{P}_k$ and V_F includes the remaining vertices

3-We therefore construct $\mathcal{F}_k$ as follows:

$$\mathsf{V}_k = \mathcal{F}_k(\mathsf{E}_{\mathcal{P}_{k-1}}) = \bigcap_{v \in V_T} \mathcal{N}(v) - \bigcup_{v \in V_F} \mathcal{N}(v)$$

The proof introduces an algorithm to discover embeddings of a more complex pattern $\mathcal{P}_k$ based on any embedding of $\mathcal{P}_{k-1}$. The base pattern $\mathcal{P}_1$ is a vertex, with $\mathcal{P}_2$ being an edge. Thus the pattern is encoded by the sequence $\mathcal{F}_1, ..., \mathcal{F}_k$.

Schedule generation:modeling

The series of function described above encodes the relationships among the vertices of a pattern. these functions $\mathcal{F}_1, ..., \mathcal{F}_k$ must be applied in order when computing patterns to respect their dependencies. There can be, however, many possible series' of functions for the same pattern.

Schedule generation:modeling

Given a pattern, we:

1- build a colored complete graph by coloring all present edges black and all absent edges red; 2- we assign an order to add vertices; 3- We assign directions to the edges.

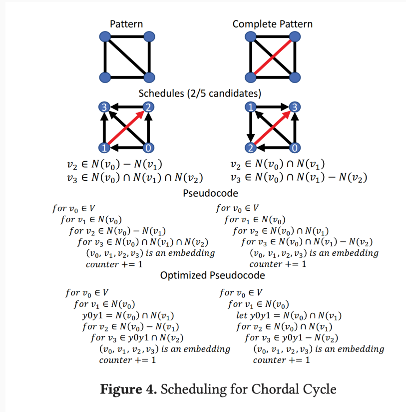

Figure 4. Scheduling for Chordal Cycle

Schedule generation:modeling

Lemma 2. A tournament must be acyclic for its corresponding schedule to exist.

Corollary 3. an acyclic tournament gives the vertices a total order, which is the necessary condition for a schedule to exist.

In a k-vertex complete graph, there are k! unique orderings of the vertices, and therefore k! possible acyclic tournaments. Due to the colored edges, some of these orderings are nonisomorphic, and worth exploring.

Schedule generation: Multiplicity

One can notice from the triangle counting algorithm that a schedule can have a multiplicity problem. In general, one needs to determine the multiplicity for a given pattern. Figure below shows the algorithm the authors provide for this task.

Algorithm 2: Computing Multiplicty for a Pattern

input : $\mathcal{P}_n$: the Pattern.
output: M : the multiplicity of $\mathcal{S}$ counting $\mathcal{P}_n$.

```
1 begin
2   M ← 0;
3   base_order ← range [0..n);
4   for order in permutations(base_order) do
5     Pattern P'_n ← empty;
6     for Edge (v_a, v_b) in P_n do
7       P'_n.add_edge(order[v_a], order[v_b]);
8       if equal(P_n, P'_n) then
9         M ← M + 1;
```

Code generation: Single pattern

schedule is represented by a series of functions $\mathcal{F}_1, \ldots, \mathcal{F}_n$, each depending on the vertices $[v_0, \ldots, v_{k-1}]$. Such a pattern naturally lends itself to a nested loop structure. At each loop level k, the loop body traverses the vertex set Vk–1 and apply $\mathcal{F}_k$ to $[v_0, \ldots, v_{k-1}]$ to create a vertex set V_k for the next loop.

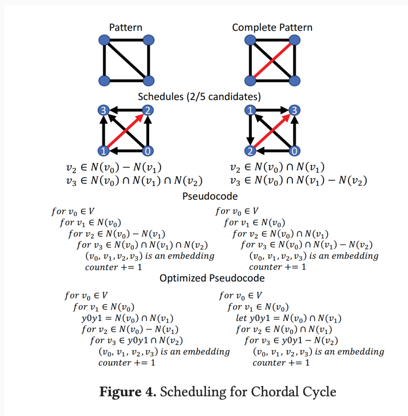

Figure 4. Scheduling for Chordal Cycle

Code generation: Single pattern

However, while memory footprint is minimized, we need to fetch the neighbors for each vertex each time in the inner loop.

To avoid that, We define a prefix $\mathcal{F}_k^p$ of $\mathcal{F}_k$ which contain all of its operations on only vertices $[v_0, ..., v_{p1}]$. The prefix is pre-computed and its results stored.

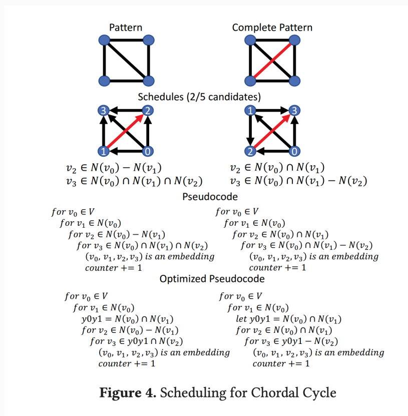

Figure 4. Scheduling for Chordal Cycle

Code generation: optimal schedule

As we have shown before, **many schedules** exist for a particular pattern.But **one** must be selected

This task is challenging because: 1- the embedded structure and the **complex set compositions** used in the schedules. 2- it is a function of the **input graph**.

The problem is simplified by considering a **random graph**, where each edge has a probability p.

Hence, the expected size of a neighbor set is $n \times p$. The expected size of $\mathcal{N}(v_i) \cap \mathcal{N}(v_j)$ and $\mathcal{N}(v_i) - \mathcal{N}(v_j)$ is hence np^2 and $np(1-p)$, respectively.

With the estimate for the two basic operations, we can further estimate the size of the resultant set of any function $\mathcal{F}_k$, and hence we obtain the complexity for the schedule.

Code generation: Multi-pattern schedule

If multiple patterns need to be counted, some of the schedule operations can be shared to avoid repeating computation

e.g., Schedules for every pattern start with the same $\mathcal{F}_1$ and $\mathcal{F}_2$, and may remain the same for levels beyond that.

Evaluation

Evaluation: summary

In this section, the authors evaluate AutoMine's performance against three graph mining systems: Arabesque, RStream, and ASAP.

The highlights of the results are as follows:

1) For 24 different mining workloads on real-world graphs, **AutoMine is up to 4 orders of magnitude faster than Arabesque, running on 10 machines, and RStream**.

2) ASAP uses approximation techniques to accelerate graph mining. Even when it uses 16 machines and 5% as the error target, **ASAP takes on average 12.8X longer time** to perform size-3 motif counting on 4 real-world graphs compared to AutoMine.

3) AutoMine, thanks to its **efficient memory use** and out-of-core processing capability, can successfully process a graph with more than 25 billion edges.

Evaluation: against Arabesque and RStream

App.	Sys.	CiteSeer	MiCo	Patents
TC	AM	0.01	0.04	0.14
	RS	0.01	2.5	9.6
	AR	38.1	43.1	114.9
	ST	0.003	0.97	6.2
3-MC	AM	0.016	0.12	0.5
	RS	0.13	1666.9	1149.1
	AR	40.6	51.7	116
4-MC	AM	0.024	22.0	20.0
	RS	2.2	T	F
	AR	F	F	F
5-CC	AM	0.024	11.4	0.17
	RS	0.075	F	134.1
	AR	42.8	132.0	174.5
3-FSM 300	AM	0.024	0.88	3.9
	RS	0.086	649.1	1453.2
	AR	F	F	F
3-FSM 500	AM	0.037	0.88	3.9
	RS	0.088	182.6	1002.8
	AR	F	F	F
3-FSM 1K	AM	0.033	0.87	4.1
	RS	0.09	2.5	81.5
	AR	35.6	5790.1	F
3-FSM 5K	AM	0.02	0.039	3.9
	RS	0.087	2.54	36.3
	AR	41.6	120.8	F

Table 2. Comparisons between AutoMine (AM), single-threaded triangle counting (ST), RStream (RS), and Arabesque (AR) on CiteSeer, MiCo, and Patents. 'T' indicates timeout after 48 hours of execution. 'F' indicates execution failure due to insufficient memory or disk space.

Evaluation: against Arabesque and RStream

Figure below shows the capacity needed by RStream and AutoMine to fit the entire workload (graph plus intermediates) into the main memory.

Triangle counting incurs on average **520MB** space overhead for intermediates. AutoMine reduces the average space overhead to only **8.4KB**.

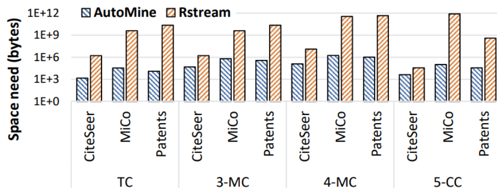

Figure 5. Needed space for RStream and AutoMine to fit all the data into memory.

Evaluation: against ASAP

Despite producing the **exact** counts using a single machine, AutoMine outperforms ASAP, running on 16 machines, by up to **68.8X** (**on average 12.8X**).

The reason is that ASAP basic approach to enumerate and store embeddings, hence inheriting the major weaknesses of inefficient algorithms and high memory consumption.

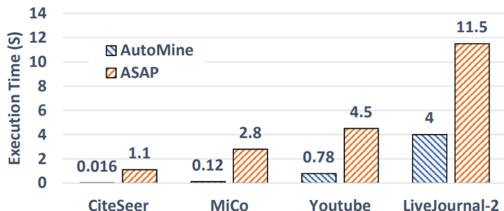

Figure 8. Results versus ASAP

Evaluation: Schedule Selection

AutoMine's greedy approach successfully select a good schedule that is 2.4X faster than the slowest schedule and only is 9.9% slower than the optimal schedule (In 4-motif counting).

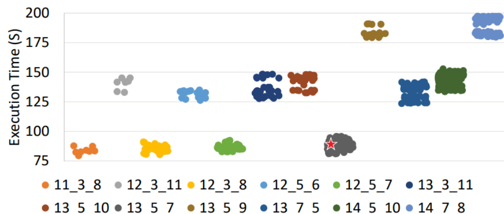

Figure 9. Candidate Schedules

Evaluation: Scalability

From 1 thread to 10 threads, AutoMine enjoys almost **linear** scalability, which becomes worse beyond 10 threads and further degrade beyond 20 threads. Perhaps due to NUMA effect.

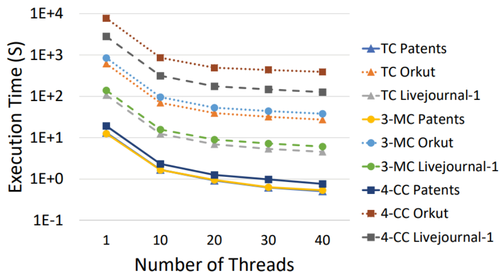

Figure 10. Threading scalability

Evaluation: Large patterns

These are the first 8-node pattern results published for graphs of this scale,.

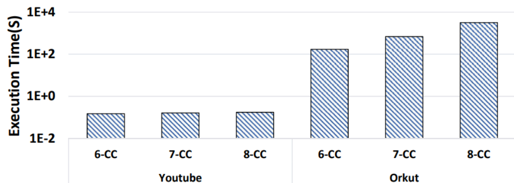

Figure 11. AutoMine Performance for Large Patterns

Conclusions and Discussion

Conclusions and Discussion

- Auto Mine gives good performance while maintaining abstraction by overcoming two main challenges:
    1. Efficient utilization of memory
    2. Using efficient graph mining algorithm automatically.
- the proposed system produces up to several order-of-magnitude higher performance than existing systems for a variety of graph mining tasks on real-world graphs, despite running on a single machine.

Discussion:

1. A common theme in the papers we have seen in the last couple of weeks is the trade off between high level abstraction and performance. This paper shows that it is possible to provide both. What other systems can you think of that has achieved similar feat.
2. This is paper is quite recent (October 2019). How do you think it will be received in the graph mining community?

Thank You