APM: Efficient Approximate Graph Pattern Matching System

Progress Report

Yang Yiliu

1155157082

Abstract

Nowadays, the abstract relationship represented mainly by the Internet and social media are vastly developing and getting larger. Thus, such a system that can handle large graphs to solve problems in the huge abstract relationship is needed. This final report provides APM, an efficient approximate graph pattern matching system, which can handle general graph pattern matching problems in a huge graph within a faster time. Considering both time and space, we have adopted an estimation algorithm, neighborhood sampling, which has enough accuracy and speed in a practical problem. Furthermore, we have also developed an algorithm to automatically design sampling method and the recommended estimation number for each given pattern. This final report also compares its performance with other systems.

1. Introduction

Graphs are often used to represent abstract relationships, such as friendships and followers. Recently, scientists and engineers are using graph pattern matching to analyze a graph, especially the triangle (i.e., three-vertex clique) counting [1]. However, in most practical problems, the graph is so large that counting patterns precisely will cost unaffordable time. Therefore, a system that can handle large scale graphs with affordable time is needed. In view of approximation number with a certain accuracy is also useful in practical problems, we have developed APM to approximately count certain patterns in a large graph.

The general method to estimate the number of patterns is neighborhood sampling, which will be introduced later. There are also some precious counting algorithms, but most of them need costing hours to get the result. Besides, the implementation methods of the precious algorithm are very different from approximation counting. The principle of precise calculation is also difficult to learn from, so this report will not introduce these methods.

In this report, I will introduce our approximate graph pattern matching system, APM, which is an improved version of ASAP[2] in C++, additionally supporting general patterns. APM is a distributed system to process huge graphs and count the approximate number of patterns. This system will first divide the whole graph into small portions, send these portions to each worker. Each worker will parallelly execute estimation in its graph at certain times with its threads. Finally, each worker will return its result to a certain machine, in which the results will be synthesized and get the final result.

2. Implementation Details

2.1 Neighborhood Sampling

Neighborhood sampling is a common algorithm to approximately count patterns appear in a graph. The basic idea is to trade edges as a stream. When select one edge, all unselected edges that appear before this edge will be erased. In one estimation, randomly sample some edges with their probabilities. If these edges can form the pattern, the product of probabilities will be the probability of one pattern that appears in the whole graph. By doing estimation several times, the count will be close to the exact pattern count. Besides, the sampling has two types. One is sample edge from the whole graph, and another is sample edge from adjacent of selected edges. For simple patterns, it is enough if only choose type-I or type-II. But for most complex patterns, we need to choose type-I and type-II and the combination of type-I and type-II so that the estimation will be accurate. Thus, if the system needs to support general patterns, it should have an algorithm to

design the sampling method.

2.2 System Implementation

The whole system is operated in four parts.

- **Preprocessing**: Read the graph data from the file. Reorder all vertices and number each edge. Make two copies of the edge, which are E_1 and E_2. Sort E_1 with the starting point as the first keyword and the ending point as the second keyword. Sort E_2 with the starting point as the first keyword and the input order index as the second keyword. The reason why does these two sorts is that we can get all edges appear after a certain edge in $O(log(m))$ time and check whether two vertices are neighborhoods in $O(log(m))$ time. In this preprocessing part, the time complexity is $O(m * log(m))$, where m is the number of edges.

- **Partition**: The system divides the graph into a certain number of sub-graphs, where the number is the number of distributed computers. And pass the sub-graphs to each machine.

- **Estimation**: In estimation part, we have implemented some APIs.
  1. **SampleEdge**: () -> (e, p)

     In this SampleEdge function, it will randomly return an edge from the whole graph with the probability (i.e.,m).

  2. **ConditionalSampleEdge**: (SubGraph) -> (e, p)

     In this ConditionalSampleEdge function, it will randomly return an edge from the adjacent of SubGraph's edges with the probability (i.e., the number of adjacents), where the edge must appear after all edges in SubGraph.

3. **ConditionalClose**: (SubGraph, Pattern) -> (integer)

   In this ConditionalClose function, it will compare SubGraph and Pattern, and try to connect the missing edges, where the edges must appear after all edges in SubGraph. This function will return the number of methods to close the graph.

So far, we can design a triangle estimator.

```
(e1, p1) = SampleEdge();
(e2, p2) = ConditionalSampleEdge(SubGraph(e1));
(i3) = ConditionalClose(SubGraph(e1, e2), Triangle());
if (p1 * p2 == 0) return 0;
return i3 / (p1 * p2);
```

4. **Analysis**: (Pattern) -> (function)

   In this Analysis function, we analyze a certain pattern and return a certain method to sample edges. Firstly, we do maximum cardinality matching in this pattern, and get the number num, where num is the maximum number of times to use $SampleEdge$. Secondly, directly use $SampleEdge$ num times, then use $ConditionalSampleEdge$ to complete remaining edges, finally use $ConditionalClose$ to check if the SubGraph can form the pattern. Thirdly, decrease num and doing step 2 until num equals 1.

   Take 5-vertex clique as an example. The maximum cardinality matching of this pattern is 2. So, when num equals 2, we use $SampleEdge$ twice, then use $ConditionalSampleEdge$ to extend one of the sampled edges. Then combine these 2 parts and check if it can be closed to the pattern. Then comes to num equal 1, we use $SampleEdge$ once, then use $ConditionalSampleEdge$ three times to extend this edge. Finally, check if it can be closed.

So, for a worker in this distributed system, its task is to analyze the given pattern, and then run the given function several times. This number can be got from a mathematical formula, $\frac{3K}{\varepsilon^2\tau} * ln(\frac{2}{\delta})$, where K is the maximum number returned by one estimator, ϵ is error rate, τ is the pattern count, δ is confidence. Finally, each worker will return its result, $\frac{Sum\ of\ answers}{Estimation\ times}$, back to the master machine.

- **Synthesis**: When all machines have finished the estimation, they will return their answer to the master machine. The master machine will combine all answers and get the final answer.

3. Performance

We select ASAP[2] and GraphPi[3] as comparisons. Since we do not have such a distributed environment, so we choose one machine with 8 threads to test our system and GraphPi, where ASAP has 16 machines and each with 8 threads. Besides, our system has not been optimized, and I think it still has a lot of space for improvement. Both ASAP and APM are set as 5% error rate and 95% confidence.

3-Clique Runtime	APM(5%) 1x8	ASAP(5%) 16x8	GraphPi 1x8
LiveJournal[4]	53s	11.5s	35s
YouTube[4]	36s	4.5s	4.2s

4-Clique Runtime	APM(5%) 1x8	ASAP(5%) 16x8	GraphPi 1x8
LiveJournal[4]	41.6s	83s	197s
YouTube[4]	18s	36s	9.1s

4. Conclusion

This report presents APM, an efficient approximate graph pattern matching system.

Basically, it is another implementation version of ASAP[2], but we improve it to automatically support general patterns. As shown in the performance part, APM is also more efficient than ASAP. We are still working for it to support advanced mining and improve its time consumption.

References

[1] M. Bisson and M. Fatica, “High performance exact triangle counting on gpus,” vol. 28, no. 12, pp. 3501–3510, 2017.

[2] A. P. Iyer, Z. Liu, X. Jin, S. Venkataraman, V. Braverman, and I. Stoica, “{ASAP}: Fast, approximate graph pattern mining at scale,” 2018, pp. 745–761.

[3] T. Shi, M. Zhai, Y. Xu, and J. Zhai, “GraphPi: high performance graph pattern matching through effective redundancy elimination,” 2020, pp. 1–14.

[4] Leskovec, J., and Krevl, A. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.