Efficient Graph Analytics in Python for Large-scale Data Science

Xiantian Zhou and Carlos Ordonez

University of Houston, Houston, TX 77204, USA

Abstract. Graph analytics is important in data science research, where Python is nowadays the most popular language among data analysts. It facilitates many packages for graph analytics. However, those packages are either too specific or cannot work on graphs that cannot fit into the main memory. Moreover, it is hard to handle new graph algorithms or even customize existing ones according to the analyst's need. In this paper, we develop a general graph C++ function based on a semiring algorithm including two math operators. The function can help solve many graph problems. It also works for graphs that cannot fit in the main memory. Our function is developed in C++, but it can be easily called in Python. Experimental comparison with state-of-art Python packages show that our C++ function has comparative performance for both small and large graphs.

Keywords: Graph Analysis · Data Science · Python · Template Algorithm

1 Introduction

Graph analytics remains one of the most computationally intensive tasks in data science research mainly due to large graph sizes and the structure of the graphs. On the other hand, Python is the most popular system to perform data analysis because of its ample library of models, powerful data transformation operators, interpreted and interactive language. However, Python is slow to analyze large data sets, especially when data cannot fit in RAM. Many research progress on efficient analytic algorithms working on graph analytics engines, which frequently needs data exporting and importing [3, 4]. But exporting data sets from or to a graph engine is slow and redundant. Even though many libraries in Python enable graph analytics, these libraries require time to learn, or users need reprogramming existing functions, which limits their impact [1, 5, 6]. Moreover, graph functions provided by Python packages are too specific. If a user needs a new graph algorithm or customizes existing ones, none of the Python packages can be helpful. Graph analytics highly depends on the APIs that packages provide. However, graph analytics is a rapidly developing research field, in which there are many new graph metrics and algorithms appearing.

In this paper, we prove that it is possible to identify a general graph C++ function that can solve lots of graph algorithms. we develop the C++ function

which is the most computationally intensive part in many graph analytics with C++. The function is light-weighted and can be easily called in Python. Also, it can efficiently work for both small and large graphs which cannot fit in the main memory. Moreover, it is convenient to customize a graph algorithm or program a new one with our function.

2 Related Work

Graph analytics in Python is becoming more important in data science research. However, it is challenging because of large graph size and complex patterns embedded in the graph. There are many graph packages developed recently. Scikinetwork is a package based on SciPy, and it provides state-of-the-art algorithms for ranking, clustering, classifying, embedding and visualizing the nodes of a graph [1]. Python-iGraph is a library written in C++, which provides an interface to many graph algorithms [5]. The graph-tool is also an efficient package that is developed in C++ and it can works in multi-threads. GraphBLAS is another popular packages, but it needs much time to learn [2, 6]. Most of those libraries cannot work for graphs that can not fit in the main memory. Also, they provide too specific interfaces that we can not modify or customize graph algorithms.

3 Definitions

Let $G = (V, E)$ be a directed graph with $n = |V|$ vertices and $m = |E|$ edges, where V is a set of vertices and E is a set of edges. An edge in E links two vertices in V, has a direction and a weight. The adjacency matrix of G is a $n \times n$ matrix such that the entry i, j holds 1 when exists an edge from vertex i to vertex j. In sparse graph processing, it is preferable to store the adjacency matrix of G in a sparse form, which saves spaces and CPU resources. There are many different sparse formats, such as compressed sparse row, compressed sparse column, and coordinate list, and other data structures. In our work, the input and output are represented as a set of tuples (i, j, v) where i and j represent the source and the destination vertex, and v represents the value of edge (i, j). The set of tuples can be sorted by j or i. We denote E_i is a set of tuples sorted by i and E_j is sorted by j. Since entries where $v = 0$ are not stored, the space complexity is $m = |E|$. In sparse matrices, we assume $m = O(n)$.

4 A Graph Analytics function Based on a Semiring

We start by explaining classical graph problems, from which we derived the general graph analytics function which is an algorithm of a semiring including two math operators. Then, we introduce the architecture of the function. Finally, we use it to implement different graph algorithms and show the function can solve many graph problems.

4.1 A General Graph Analytics Function

Graph metrics are about either local information or global information of a graph. Some graph metrics such as in-degree and out-degree are about local information. Many other graph metrics such as PageRank, centrality, reachability, and minimum spanning tree are about global information. Those graph metrics require a part or full traversal of the graph. There are different ways to obtain local information of a graph, and adjacent matrix-matrix or matrix-vector operations [8] is one of the most commonly used solutions. For example, the matrix product $E \cdot S$, where S is an n-dimensional vector, helps finding the vertices with highest connectivity. $E \cdot E$ is used to find paths whose lengths are two.

To obtain the global information, we can repeat the matrix-matrix or matrix-vector multiplication process. For example, the matrix product $E \cdot E \cdot E$ can be used to get all triangles in G, which has been identified as an important primitive operation. The iteration of $E \cdot E$, multiplying E k times(k up to $(n-1)$) gets all paths with length k, from which we can filter the shortest/longest ones by different aggregation operators and count them. Also, it gets all the intermediate vertices on those paths in order, which is essential for path analysis, such as betweenness centrality. Finally, $E \cdot E \cdot ...E$ ($n-1$ times) until a partial product vanishes is a demanding computation returning $G+$, the transitive closure (reachability) of G which gives a comprehensive picture about G connectivity [7]. Besides connectivity and paths, other graph metrics such as PageRank, closeness can also be computed as powers of a modified transition matrix. Note that although we express those graph algorithms as matrix multiplication, they are different semirings according to different graph algorithms. Semirings are algebraic structures defined as a tuple $s(R, \oplus, \otimes, 0, 1)$ consisting of a set R, an additive operator $\oplus$ with identity element 0, a product operator $\otimes$ with identity element 1. The regular matrix multiplication is defined under $(R, +, \times, 0, 1)$. A general definition of matrix multiplication expands it to any semiring. For example, (min, add) is used to solve shortest paths problem where min is the additive operator, and add is the product operator. The boolean semiring, with $\vee$ (logical OR) as the additive operator and $\wedge$ (logical AND) as the product operator, is frequently used in linear algebra to represent some graph algorithms. We use $\oplus, \otimes$ to denote the additive and product operator in a semiring.

Input: $E, V, \oplus, \otimes$
Output: R
1 $n \leftarrow |V|$;
2 **for** $i \leftarrow 1$ **to** n **do**
3 **for** $j \leftarrow 1$ **to** n **do**
4 **for** $k \leftarrow 1$ **to** n **do**
5 $R[i,j] = R[i,j] \oplus (E[i,k] \otimes E[k,j])$
6 **end**
7 **end**
8 **end**

Algorithm 1: General graph algorithm

We can see that all algorithms mentioned above can be expressed as a matrix-vector or matrix-matrix operation or an iteration of such operations under different semirings. The general algorithm is shown in Algorithm 1. We show it based on a dense matrix multiplication to represent our general graph algorithm. The number of loops can vary according to different graph algorithms. Based on this mathematical foundation, we develop a general graph C++ function $s(E, \oplus, \otimes, K)$ in Python which takes E as input and performs computation according to two operators $(\oplus, \otimes)$ of a semiring. For some graph algorithms, we only want to traverse parts instead of the entire graph, especially when the graph is large. So we use K to specify the maximum number of iterations. Intermediate vertices are needed for graph metrics such as betweenness centrality, K-step betweenness centrality. In contrast, they are not required for other graph metrics such as reachability and connectivity. So we add an optional boolean parameter h which indicates whether intermediate vertices on paths need to be recorded or not. The default value of h is false.

Input: $s(E, \oplus, \otimes, K)$
Output: R
$k \leftarrow 0$, $i \leftarrow 1$, $j \leftarrow 1$, $E_i \leftarrow E$ sorted by i, $E_j \leftarrow E$ sorted by j
for $k < K$ **do**
  while *not end of* E_i *and not end of* E_j **do**
    read a block R_b if $k \neq 0$;
    read a column block B_j from E_j with j maximum index is j_B;
    read a row block B_i from E_i with i maximum index is i_B; **while** $i_t < i_B$, $j_t < j_B$ **do**
      foreach *pair* $i_t = j_t$ **do**
        $R_b[i,j] = R_b[i,j] \oplus (B_i[i,j_t] \otimes B_j[i_t,j])$;
      end
    end
    periodically write the $R_b[i,j]$ into disk;
  end
  Merge all $R_b[i,j]$ into R;
  $k \leftarrow k + 1$
end

Algorithm 2: The database algorithm of our graph C++ function

Now we show how to do graph analytics with our function. By specifying the function as $s(E, min, add, n)$, we can calculate shortest paths, where $K = n$ because of traversing the full graph. Similarly, the reachability of all vertices can be calculated with $s(E, \vee, \wedge, n)$. Using $s(E, add, mul, n)$, we can obtain the number of paths between each pair of vertices. If we want to calculate K-step betweenness centrality which is not included in many Python packages, we can use $s(E, min, add, K, h = true)$ get all shortest paths in K steps with intermediate vertices along them. Even for many other graph metrics other than paths and connectivity, such as PageRank and closeness, the C++ function can also be used to calculate the most computation intensive part. The function triggers computation in a C++ environment. The results are written to disk. The architecture of the function is discussed in the following section.

4.2 System Architecture for Function Processing

Our goal is to provide a general C++ function running on a simple architecture to efficiently compute analytics on large graphs in a data science language. And we divide the input files into blocks and process block by block when the computation involves a large graph, so the function works for graphs that cannot fit in the main memory.

Input and Output: The input graph G is stored on a CSV text file with each edge in triple format (i, j, v). It can be in any text file format like .csv, .txt, and so on. Before processing, we first sort the input by i, then by j. The output file is also in (i, j, v) format and sorted by i.

Processing: When our general Python function an internal C++ is automatically triggered. Such C++ function, with many parameters, reads the input file from disk, efficiently performs computations in main memory and writes results to a text file format. If the graph is too big to fit in the main memory, the C++ function will read and process input files block by block. The algorithm is shown in Algorithm 2. Remember that E_i is sorted by i and E_j is sorted by j.

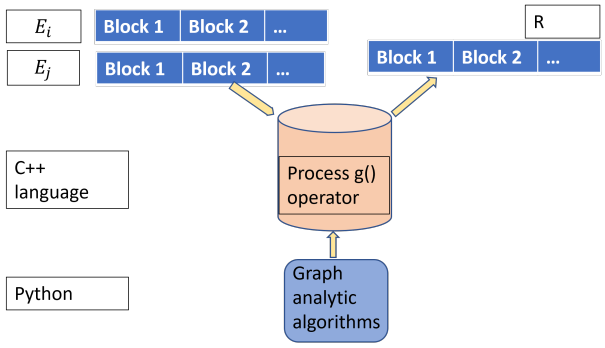

Fig. 1: An overview of our system

5 Experimental Evaluation

NumPy and SciPy are fast Python packages for processing matrices. SciPy has a high performance and low memory use achieved through a mix of fast sparse matrix-vector. Those two packages have the basic matrix-matrix multiplication operator. Thus, we choose them to compare with our C++ function. The computer used for the experiments has an Intel Pentium(R) CPU running at 1.6 GHz, 8 GB of RAM, 1TB disk, 224kb L1 cache, 2MB L2 cache and running Linux Ubuntu 14.04. We use both synthetic and real graph data sets for experimental evaluation. The real graphs are from the Stanford SNAP repository. The comparison results are shown in Table 1. If the running time is more than

one hour, we put a "Stop" sign in the table. From Table 1, we can see that our function is faster than NumPy, but it is slower than SciPy for small graphs. However, SciPy crashes for large graphs because it assumes data can fit in the main memory. Moreover, our function can perform matrix operations under different semirings while most other packages can only perform matrix multiplication.

Table 1: Summary of results(time in seconds).

Dataset	Type	n	m	NumPy	SciPy	Our C++ function
synthgraph1	Synthetic	1K	100K	12.0	0.2	1.0
synthgraph2	Synthetic	5K	2.5M	Stop	14.0	162.0
wiki-vote	Real	8K	103.6K	Crashed	0.1	1.6
webgoogle	Real	875K	5.1M	Crashed	Crashed	Stop

6 Conclusions

In this paper, we developed a powerful, yet easy to use function based on a semiring algorithm running on an efficient system architecture to process big graphs in a data science language. The function can be easily called in Python, and it can be used for programming, modifying and customizing most of existing and new graph algorithms. Moreover, it works for large graphs that cannot fit in the main memory. We compared the performance of our function with state of-the-art graph analytic packages, exhibiting similar performance for small graphs and better scalability for large graphs that cannot fit in the main memory.

For future work, we will study parallel processing, develop more efficient algorithms when results fit in main memory (e.g. PageRank, Connected Components). Finally, we plan to characterize which graph algorithms pattern fits our semiring algorithm.

References

1. Bonald, T., de Lara, N., Lutz, Q., Charpentier, B.: Scikit-network: Graph analysis in Python. J. Mach. Learn. Res. **21**, 185:1–185:6 (2020)
2. Chamberlin, J., Zalewski, M., McMillan, S., Lumsdaine, A.: PyGB: GraphBLAS DSL in Python with dynamic compilation into efficient C++. In: IPDPS Workshops 2018. pp. 310–319 (2018)
3. Ghrab, A., Romero, O., Jouili, S., Skhiri, S.: Graph BI & analytics: Current state and future challenges. In: DaWaK 2018. Lecture Notes in Computer Science, vol. 11031, pp. 3–18. Springer (2018)
4. Ho, L., Li, T., Wu, J., Liu, P.: Kylin: An efficient and scalable graph data processing system. In: 2013 IEEE Big Data. pp. 193–198 (2013)
5. Ju, W., Li, J., Yu, W., Zhang, R.: iGraph: an incremental data processing system for dynamic graph. Frontiers Comput. Sci. **10**(3), 462–476 (2016)
6. Kumar, M., Moreira, J.E., Pattnaik, P.: GraphBLAS: handling performance concerns in large graph analytics. In: 2015, ACM CF. pp. 260–267. ACM (2018)
7. Ordonez, C., Cabrera, W., Gurram, A.: Comparing columnar, row and array DBMSs to process recursive queries on graphs. Information Systems (2016)
8. Zhou, X., Ordonez, C.: Computing complex graph properties with SQL queries. In: 2019 IEEE Big Data. pp. 4808–4816. IEEE (2019)