Bitwise Algorithms to Compute the Transitive Closure of Graphs in Python

Xiantian Zhou[1], Abir Farouzi[2], Ladjel Bellatreche[2], and Carlos Ordonez[1]

[1] University of Houston, USA
[2] LIAS/ISAE-ENSMA, France

Abstract. The transitive closure (TC) of a graph is a core problem in graph analytics. There exists many High Performance Computing (HPC) and database solutions to solve the TC problem for big graphs. However, they generally require the graph to fit in main memory and they require converting into specific binary file formats. To solve such limitations, this paper presents a novel solution to solve TC within the Python library ecosystem, combining HPC techniques and database system algorithms. We introduce two complementary algorithms removing HPC memory limitations: (1) an algorithm that efficiently converts edges into bit vectors and (2) a database-oriented, bit-vector, highly parallel matrix algorithm, which processes the graph in blocks. An experimental evaluation shows our solution provides better performance than state of the art Python libraries.

1 Introduction

Transitive closure (TC) is one of the most computationally intensive tasks in data science research, primarily due to the large size and complex structure of graphs. It plays a crucial role in various graph problems. For instance, triangle enumeration represents the initial two steps in TC [4]. Consequently, several solutions leveraging HPC technologies have been proposed, including graph engine solutions, SQL-based solutions, and Python libraries [5]. SQL-based solutions offer elegance and memory limitations freedom, while Python is a popular language for data analysis, offering numerous libraries and packages for graph analysis such as GraphBLAS, Scikit-network, and NetworkX. These libraries provide state-of-the-art graph algorithms [2, 3]. However, Python may be slow when analyzing large graphs, particularly those that cannot fit in RAM. Moreover, significant research progress has been made on efficient analytic algorithms for TC, some of which are based on relational algebra operations. For example, some algorithms employ hash-based fragmentation or fragmentation based on the semantic content of data. [7] explored a double-hash data fragmentation scheme. Another class of parallel TC algorithms is based on matrix manipulation. [1] presented parallel algorithms for computing the transitive closure of a database relation, applicable on both shared-memory and message-passing architectures. Generally, these parallel transitive closure algorithms operate directly on the adjacency list. However, parallel TC algorithms that work on a matrix representation can be

more efficient. In fact, our experiments with different input graphs have shown that the TC graph density can exceed 70%, even if the input graph density is below 10%.

In this paper, we introduce disk-based distributed TC solutions that operate on the bit-matrix. We study how to develop TC algorithms within the Python ecosystem while adhering to principles of database systems. Our ultimate aim is to enable efficient processing of large graphs without memory limitations and with acceptable response times. Our implementations are suitable for reachability and path problems, and our experimental study shows the superiority of our solutions over existing popular analysis systems, suggesting potential advancements in bridging high-performance computing and Python.

2 Background

2.1 Graph

Let $G = (V, E)$ be a directed graph with $n = |V|$ vertices (V is the set of vertices) and $m = |E|$ edges (E is the set of edges). An edge in E links two vertices in V, and has a direction. The adjacency matrix of G is a $n \times n$ matrix where a 1 is stored in the entry (i, j) if there exists an edge from vertex i to vertex j. Storing the adjacency matrix in this sparse form helps conserve space and CPU resources. In our work, we do not use weight sice we are solving TC prolbem. Thus, the input graph is represented as an edge list $E(i, j)$ and can be sorted either by i (E_i) or j (E_j). Since only existing edges are stored, the space complexity is $O(m)$. In sparse matrices, we assume $m = O(n)$.

The TC graph G^* compute all vertices reachable from each vertex in G. It is defined as: $G^* = (V, E^*)$, where $E^* = \{(i, j)$ s.t. there is a path between i and $j\}$. TC graph is stored as an adjacency matrix, because TC graph is much denser than the sparse input graph. The entries of TC matrix can be stored in one bit. In our paper, we use a 64-bits integer in C to store 64 edges.

2.2 Classical Algorithms in Main Memory

Warshall's algorithm is recognized as the best algorithm for computing TC. It performs perfectly with HPC in main memory. However, when dealing with large graphs that cannot fit in main memory, it fetches random edges from disk and reads the entire matrix into memory at least N times. To address this issue, Warren proposed an improvement of Warshall's algorithm that reduces the number of I/O for large graphs by processing the matrix elements in a row order in two passes [6].So we think Warren is better for large graphs because it has lower I/O: it requires loading block less times from disk into RAM. Thus, it is chosen as the base algorithm for developing our solution.

3 TC solved in Python with Database Algorithms

Inspired by database systems, our solutions process the input graph by blocks instead of reading the entire graph into main memory.

```
   Input: E
   Output: E*
 1 for i ← 2 to n do
 2 |   for j ← 1 to i − 1 do
 3 |   |   if E[i, j] = 1 then
 4 |   |   |   set E[i, ∗] = E[i, ∗] ∨ E[j, ∗]
 5 |   |   end
 6 |   end
 7 end
 8 for i ← 1 to n − 1 do
 9 |   for j ← i + 1 to n do
10 |   |   if E[i, j] = 1 then
11 |   |   |   set E[i, ∗] = E[i, ∗] ∨ E[j, ∗]
12 |   |   end
13 |   end
14 end
```

Algorithm 1: Warren's Algorithm

3.1 Transforming the Edge Data Set into a Bit Matrix

Storing TC graph in a matrix has many advantages such as using one bit to store each edge, and doing a world-parallel "OR" instruction by storing the matrix in packed row-major order. Therefore, our solution will pre-process the input graph by transforming it into a bit-matrix.

For large graphs, we read and process the input graph by blocks in the main memory. We summarize the conversion of a block into a bit-matrix block below.

1. Initialize a bit-matrix block to zero.
2. For each neighbor j of a source vertex i, find the position of j bit in row i.
3. Set the bit entry (i, j) to 1.

We call the vector of bits performing OR together as a *bit-vector*. A bitmask will be used to extract or set a bit in the bit-vector. In our solution, each block is read from disk, converted to a bit-matrix, and written back to disk. Fig. 1 shows the workflow of converting the input data into a bit-matrix, whose size is $n/8$ bytes instead of $n * 4$ bytes (edges are stored with integer values).

3.2 Our Scalable Warren's Algorithm

We choose Warren's algorithm as the base algorithm to develop our solution for parallel systems, and we use Numpy library in Python to implement this algorithm. The bit-matrix is stored on disk in a row-wise manner, allowing for continuous access during bitwise OR operations, which promotes bit-level parallelism. In our solution, we employ parallel processing to read a bit-vector, thereby enhancing computational speed. The size of the vector is determined by the processor word size, which is dictated by the CPU. The parallel processes reduce the number of instructions that the system must execute.

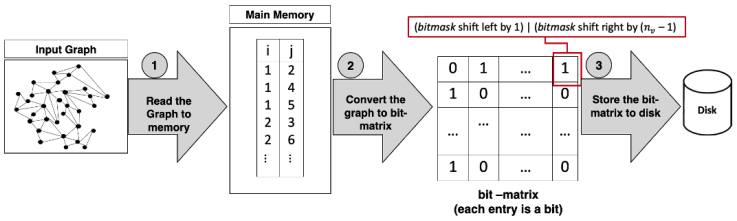

Fig. 1. Converting the input graph into a bit-matrix.

Input: $E_b, n_v, j_{start}, j_{end}$
Output: E_b^*

```
1  while not end of bit-matrix do
2      read a block, block_end ← min(block_start + block_size, n)
3      for i ← block_start to block_end do
4          bitmask ← 1
5          for j ← j_start to j_end do
6              bitmask ← (bitmask shift left by 1 ) | ( shift right by (n_v − 1))
7              if E_b(i, j/bsize) & bitmask = 1 then
8                  for k ← 1 to n/n_v do
9                      E_b[i, k] = E_b[i, k] ∨E_b[j, k]
10                 end
11             end
12         end
13     end
14     write the block to disk; block_start ← block_end + 1
15 end
```

Algorithm 2: *block_tc* algorithm.

Furthermore, we read and process a block once, since the entire input graph can be too large to fit in main memory. A block that contains several continuous rows is a plainly array of bits. When one block finishes processing, a new block will replaces the old one. At anytime during execution, there are two blocks in main memory, so the memory space needed is much smaller than $O(n^2)$. Algorithm 3 shows our improved TC algorithm. A variable *bitmask* which has the same bit size of *bitvector* will be used and initialized as 1 (the left most bit is 1). Note that TC will be dense for any connected graph, since each iteration makes the partial result denser. That is why a pure sparse solution becomes slow. If we use dense matrix format from the beginning, the competing time for each loop is related to the number of vertices which is a constant.

3.3 Time, Space and I/O Cost Analysis

Let us consider the limiting cases of a complete graph and a totally disconnected graph. That means the bit-matrix is an all-one matrix and an all-zero matrix.

Input: $E_b, bitvector, block_{size}$
Output: E_b^*
1 $n \leftarrow |V|$, $bitmask \leftarrow 1$, $n_v \leftarrow |bitvector|$, $block_{start} \leftarrow 2$
2 bloc_tc$(E_b, n_v, 1, i-1)$
3 $block_{start} \leftarrow 1$,
4 bloc_tc$(E_b, n_v, i+1, n)$

Algorithm 3: Our block-based TC system.

The time complexity of our solution is between $O(n^2/(n_v))$ and $O(n^3/(n_v^2))$, where n_v is the size of bit-vector. We can perform 'or' operation on $vector\_size$ (n_v) bits at a time, which is the size of the *bitvector*. Then limiting cases can give n^2/n_v to n^3/n_v^2 number of 'Or' operations. Our solution is easy to be paralleled for Numpy library in Python, the time complexity can be $O(n^2/(n_v * P))$ to $O(n^3/(n_v^2 * P))$ for the limiting cases where p is the number of processors. For large graphs, it will be processed by blocks. Note that the entire graph will be read to main memory once, even our solution processes by blocks. Suppose a block contains n_b rows, the numbers of I/O is n/n_b.

For space complexity, since TC computes whether a vertex i can reach another vertex j $(i, j \in V)$, each entry of the TC bit matrix is represented by one bit. The size of the TC bit matrix is $n^2/8$ bytes since each bytes contains 8 bits. When processing by blocks, there are at most two blocks in main memory at the same time. Thus, the space complexity of our solution is $O(2 * n_b * n/8)$.

4 Performance Evaluation

4.1 Experimental Setup

Software and Hardware For the competing system, we choose Python NetworkX; a popular graph analytics in Python. It is used for the creation, manipulation, and study of the complex graphs. We execute each experiment five times on a virtual machine which has 8 cores, 20 GB RAM, 1 TB disk. The size of the bit-vector is 64 to match the number of bits of CPU registers (a 64 bit int).. The computing time of our solutions includes I/O, transforming the input table to a binary matrix, and computing TC.

Data Sets The used data sets are summarized in Table 1, obtained from the konect network data collection[3]. We chose graphs with different number of vertices and edges. Moreover, we use the density to indicate the graph structure, where graph density is $\frac{m}{n*n}$. We choose graphs with different density to evaluate our algorithm.

4.2 Comparing with Python NetworkX Graph library

The NetworkX library in Python has a function $transitive\_closure()$ that returns TC of a directed graph. Table 1 shows the average time measures. Our solution

[3] http://konect.cc/

demonstrates significant performance advantages over NetworkX across various types of graphs. Particularly for large graphs like Marvel and Gnutella, our solution successfully completes the execution while NetworkX fails. Notably, our solution exhibits efficiency not only for dense graphs but also for sparse graphs. It is important to note that the running time is influenced by the density of the graph, as observed during the analysis of WekiLinks and Marvel's execution.

Table 1. Comparing with NetworkX, time in seconds.

Data set				**Competitor**	**Our solution**
	n	m	*density*	NetworkX	*block_tc*
Hamster	1859	12K	<0.36%	191	4
WekiLinks	6K	439K	<0.67%	Stop	1200
Gnutella	10K	39K	<0.034%	Stop	112
Marvel	19K	96K	<0.025%	Fail	439

"Stop" when computation is more than 30 minutes.

5 Conclusions and Future Work

In this paper, we explore bitwise algorithms to study TC with Python libraries. We believe Python language is more feasible, even other tools are efficient. Inspired by the database processing and HPC, We presented a disk-based solution to compute the TC of graphs. Our experiments show that it consistently outperformed popular analytic platforms, NetworkX library in Python. For future work, we will explore the logarithmic algorithm, and exploit more HPC techniques, such as, multicore CPUs and GPUs to solve graph path problems.

References

1. Agrawal, R., Dar, S., Jagadish, H.V.: Direct transitive closure algorithms: Design and performance evaluation. ACM Trans. Database Syst. (1990)
2. 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)
3. Chamberlin, J., Zalewski, M., McMillan, S., Lumsdaine, A.: PyGB: GraphBLAS DSL in Python with dynamic compilation into efficient C++. In: IPDPS (2018)
4. Farouzi, A., Bellatreche, L., Ordonez, C., Pandurangan, G., Malki, M.: A scalable randomized algorithm for triangle enumeration on graphs based on SQL queries. In: The 22nd of DaWaK (2020)
5. Ordonez, C.: Optimization of linear recursive queries in SQL. IEEE Trans. Knowl. Data Eng. (2010)
6. Warren, H.S.: A modification of warshall's algorithm for the transitive closure of binary relations. Commun. ACM (1975)
7. Zhou, X., Zhang, Y., Orlowska, M.E.: Parallel transitive closure computation in relational databases. Inf. Sci. (1996)