A GraphBLAS solution to the SIGMOD 2014 Programming Contest using multi-source BFS

Márton Elekes*, Attila Nagy*, Dávid Sándor*, János Benjamin Antal, Timothy A. Davis†, Gábor Szárnyas*‡
*Budapest University of Technology and Economics, Department of Measurement and Information Systems
†Computer Science and Engineering, Texas A&M University, College Station, TX, USA
‡MTA-BME Lendület Research Group on Cyber-Physical Systems
Email: elekes@mit.bme.hu, davis@tamu.edu, szarnyas@mit.bme.hu

***Abstract*—The GraphBLAS standard defines a set of fundamental building blocks for formulating graph algorithms in the language of linear algebra. Since its first release in 2017, the expressivity of the GraphBLAS API and the performance of its implementations (such as SuiteSparse:GraphBLAS) have been studied on a number of textbook graph algorithms such as BFS, single-source shortest path, and connected components. However, less attention was devoted to other aspects of graph processing such as handling typed and attributed graphs (also known as property graphs), and making use of complex graph query techniques (handling paths, aggregation, and filtering). To study these problems in more detail, we have used GraphBLAS to solve the case study of the 2014 SIGMOD Programming Contest, which defines complex graph processing tasks that require a diverse set of operations. Our solution makes heavy use of multi-source BFS algorithms expressed as sparse matrix-matrix multiplications along with other GraphBLAS techniques such as masking and submatrix extraction. While the queries can be formulated in GraphBLAS concisely, our performance evaluation shows mixed results. For some queries and data sets, the performance is competitive with the hand-optimized top solutions submitted to the contest, however, in some cases, it is currently outperformed by orders of magnitude.**

I. Introduction

Motivation. Since the release of GraphBLAS [20] in 2017, its applicability for graph analytical kernels has been studied in depth [17], [28], [9], [19], [30], including all six algorithms defined in the GAP Benchmark Suite [3]. While these efforts focused on algorithms defined on untyped graphs with no attributes (except edge weights for the shortest path algorithm), graphs with types and attributes received less attention. Such graphs, known as *attributed graphs* [24] or *property graphs* [1] have gained popularity in recent years [23] by providing an intuitive data model for modelling complex interconnected systems such as social networks, telecommunications, and financial transactions. The property graph data model allows users to query their data using a rich set of operators including graph pattern matching, cycle detection, using relational operators such as aggregation and filtering, running computations along paths, etc. Consequently, there is an increasing demand for *graph databases* capable of storing and processing such data sets [5]. Still, as of today, there is no single system available that ensures good performance on a diverse set of graph operations. The reason behind this is threefold: (1) the difficulty to overcome the "curse of connectedness", i.e. the inherently complex nature of graph data [25], (2) the lack of a unified algebra for diverse graph queries, (3) the lack of sophisticated optimization methods [24] which consider the graph structure, its properties and their correlations [22]. We believe that the GraphBLAS API and its implementations have already made significant progress in tackling the first two problems and can be extended to tackle the last one.

Challenge. Integrating GraphBLAS into a property graph query engine necessitates overcoming a number of challenges including loading the graph, handling types and properties, performing computations on induced subgraphs, evaluating cyclic queries, computing aggregation and filtering operations, etc. To demonstrate some of these problems, we use the 2014 SIGMOD Programming Contest as a case study [8].[1] The tasks of this contest are defined on a property graph and consist of complex graph queries involving a mix of *graph algorithms* (such as connected components), *complex graph patterns* (i.e. basic graph patterns extended with relational-like features such as filtering), and *navigation* [1].

Contribution. This paper presents a solution for the tasks of the 2014 SIGMOD Programming Contest. We formulated and implemented GraphBLAS algorithms for multiple BFS variants, including bidirectional and multi-source BFS.

Related systems. We are aware of two systems that use GraphBLAS to evaluate graph queries. RedisGraph [7] is a graph database that uses SuiteSparse:GraphBLAS to evaluate Cypher queries [13]. MAGiQ [14] is an RDF processing engine which maps SPARQL queries to linear algebra. Currently, neither of these systems supports workloads with queries as complex as the ones given in the programming contest.

Related contests. The 2018 Transformation Tool Contest's "Social Media" case required participants to solve two queries on a social network defined over a similar graph schema to the one in the SIGMOD 2014 contest. This contest focused on incremental evaluation and the complexity of the queries was limited. We gave a GraphBLAS solution for this in [12].

Structure. This paper is structured as follows. Sec. II describes the graph schema and the queries of the programming contest. Sec. III outlines the GraphBLAS standard and its relevant constructs. Sec. IV shows our GraphBLAS-based building blocks, Sec. V presents our query implementations, and Sec. VI discusses the experimental results. Finally, Sec. VII concludes.

[1]https://www.cs.albany.edu/~sigmod14contest/

II. Graph Schema and Queries

Graph schema. The social network instances used in the contest are represented as a property graph [1] over the schema shown in Fig. 1a. The edges in the graph are directed with the exception of the `knows` edges which are treated as undirected.
Data sets. Data sets are produced by the LDBC Datagen [22]. This generates realistic power-law degree distribution for the Person-`knows`-Person subgraph and introduces correlations and anti-correlations (e.g. people in neighbouring countries are more likely to become friends than people in distant countries).
Queries. The contest defines the following queries:

Q1. *Shortest Distance over Frequent Communication Paths* (Fig. 1b): Given two integer Person IDs `p1` and `p2`, and another integer x, find the minimum number of hops between `p1` and `p2` in the graph induced by Persons who *both* have made more than x Comments in reply to the other one's Comments, and know each other.

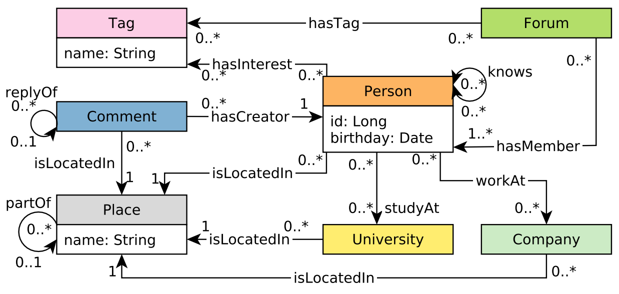
Q2. *Interests with Large Communities* (Fig. 1c): Given an integer k and a birthday d, find the top-k Tags. A Tag is characterized with its *range*, i.e. the size of the largest connected component in the graph induced by Persons who are interested in that Tag, were born on d or later, and know each other.

Q3. *Socialization Suggestion* (Fig. 1d): Given an integer k, an integer maximum hop count h, and a Place name p, find the top-k similar pairs of Persons based on the number of common interest Tags. For each of the k pairs mentioned above, the two Persons must be located in p or study or work at Organisations in p. Furthermore, these two Persons must be no more than h hops away from each other in the original `knows` graph.

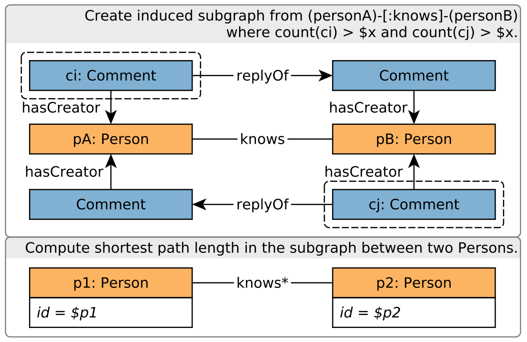
Q4. *Most Central People* (Fig. 1e): Given an integer k and a Tag name t, find the top-k Persons based on the *closeness centrality value* (CCV) in the graph induced by Persons who are members of Forums that have Tag t and know each other. For each Person, $CCV(p) = \frac{(C(p)-1)^2}{(n-1)\cdot s(p)}$, where $C(p)$ is the size of the connected component of vertex p, $s(p)$ is the sum of geodesic distances to all other reachable Persons from p, and n is the number of vertices in the induced graph. If the divisor is 0, the centrality is 0.

III. The GraphBLAS

Goal. The goal of GraphBLAS is to create a layer of abstraction between the graph algorithms and the graph analytics framework, separating the concerns of the algorithm developers from those of the framework developers and hardware designers. To achieve this, it builds on the theoretical framework of matrix operations on arbitrary semirings [16], which allows defining graph algorithms in the language of linear algebra. To ensure portability, the GraphBLAS standard defines a C API that can be implemented on a variety of hardware including GPUs.
Data structures. A graph with n vertices can be stored as a square adjacency matrix $\mathbf{A} \in \mathbb{N}^{n\times n}$, where rows and columns both represent vertices of the graph and element $A(i,j)$ contains the number of edges from vertex i to vertex j. If

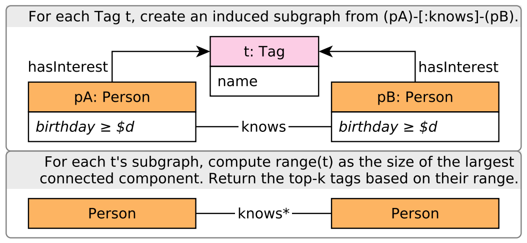


(a) Schema of the social network graphs. Only the relevant vertex, edge, and property types are shown.



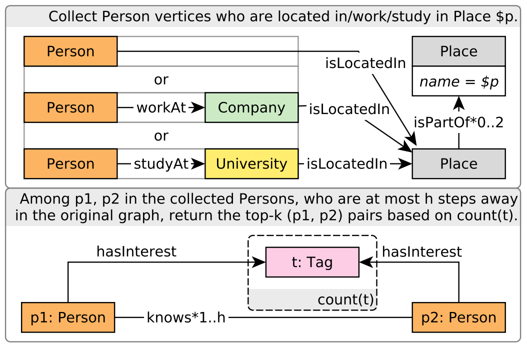
Create induced subgraph from (personA)-[:knows]-(personB) where count(ci) > $x and count(cj) > $x.
ci: Comment
replyOf
Comment
hasCreator
pA: Person
knows
pB: Person
Comment
cj: Comment
Compute shortest path length in the subgraph between two Persons.
p1: Person
id = $p1
knows*
p2: Person
id = $p2

(b) Q1($p1, $p2, $x) → shortest path length.



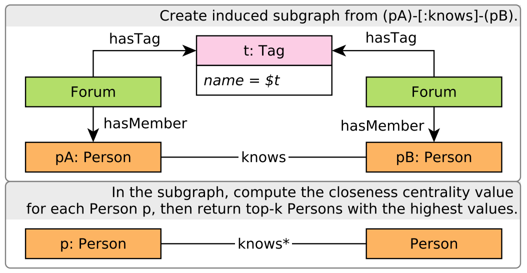
For each Tag t, create an induced subgraph from (pA)-[:knows]-(pB).
hasInterest
t: Tag
name
pA: Person
birthday ≥ $d
knows
pB: Person
birthday ≥ $d
For each t's subgraph, compute range(t) as the size of the largest connected component. Return the top-k tags based on their range.
Person
knows*
Person

(c) Q2($k, $d) → $k tag names.



Collect Person vertices who are located in/work/study in Place $p.
Person
or
workAt
Company
studyAt
University
isLocatedIn
Place
name = $p
isPartOf*0..2
Place
Among p1, p2 in the collected Persons, who are at most h steps away in the original graph, return the top-k (p1, p2) pairs based on count(t).
hasInterest
t: Tag
count(t)
p1: Person
knows*1..h
p2: Person

(d) Q3($k, $h, $place) → $k pairs of Person IDs p1 | p2 where p1 < p2.



Create induced subgraph from (pA)-[:knows]-(pB).
hasTag
t: Tag
name = $t
Forum
hasMember
pA: Person
knows
pB: Person
In the subgraph, compute the closeness centrality value for each Person p, then return top-k Persons with the highest values.
p: Person
knows*
Person

(e) Q4($k, $t) → $k Person IDs.

Fig. 1: Graph schema and queries.

GrB method	name	notation
mxm	matrix-matrix multiplication	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{A} \oplus.\otimes \mathbf{B}$
vxm	vector-matrix multiplication	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{u} \oplus.\otimes \mathbf{A}$
mxv	matrix-vector multiplication	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{A} \oplus.\otimes \mathbf{u}$
eWiseAdd	element-wise, set union of patterns	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{A} \oplus \mathbf{B}$ $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{u} \oplus \mathbf{v}$
eWiseMult	element-wise, set intersection of patterns	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{A} \otimes \mathbf{B}$ $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{u} \otimes \mathbf{v}$
extract	extract submatrix extract column vector extract row vector extract subvector	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{A}(I, J)$ $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{A}(:, j)$ $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{A}(i, :)$ $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{u}(I)$
extractElement	extract scalar element	$s = \mathbf{A}(i, j)$ $s = \mathbf{u}(i)$
apply	apply unary operator	$\mathbf{C}\langle\mathbf{M}\rangle = f(\mathbf{A})$ $\mathbf{w}\langle\mathbf{m}\rangle = f(\mathbf{u})$
select (GxB)	apply select operator	$\mathbf{C}\langle\mathbf{M}\rangle = f(\mathbf{A}, k)$ $\mathbf{w}\langle\mathbf{m}\rangle = f(\mathbf{u}, k)$
reduce	reduce to column vector reduce to scalar	$\mathbf{w}\langle\mathbf{m}\rangle = [\oplus_j \mathbf{A}(:, j)]$ $s = [\oplus_{ij} \mathbf{A}(i, j)]$
transpose	transpose	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{A}^\top$
build	matrix from tuples vector from tuples	$\mathbf{C} \leftarrow \{I, J, X\}$ $\mathbf{w} \leftarrow \{I, X\}$
extractTuples	extract index/value arrays	$\{I, J, X\} \leftarrow \mathbf{A}$ $\{I, X\} \leftarrow \mathbf{u}$

TABLE I: GraphBLAS operations used in this paper (based on [10]). *Notation:* Matrices and vectors are typeset in bold, starting with uppercase ($\mathbf{A}$) and lowercase ($\mathbf{u}$) letters, respectively. Scalars including indices are lowercase italic (s, i, j) while arrays are uppercase italic (X, I, J). $\oplus$ and $\otimes$ are addition and multiplication operators of an arbitrary semiring (defaulting to conventional arithmetic + and × operators). Masks $\langle\mathbf{M}\rangle$ and $\langle\mathbf{m}\rangle$ are used to selectively write to the result matrix/vector. The complement of a mask $\langle\mathbf{M}\rangle$ can be selected with $\langle\neg\mathbf{M}\rangle$.

the graph is undirected, the matrix is symmetric. For graphs with edge types, edges of each type can be represented as a bipartite graph. For example, instances of the `hasInterest` edge type between Person and Tag vertices can be stored in a Boolean matrix with a row for each Person and a column for each Tag, $\mathbf{HasInterest} \in \mathbb{B}^{|persons| \times |tags|}$. Edge types that have the same source and target type are captured as square matrices, e.g. $\mathbf{Knows} \in \mathbb{B}^{|persons| \times |persons|}$.

Navigation. The fundamental step in GraphBLAS is the multiplication of an adjacency matrix with another matrix or vector over a selected semiring. For example, the operation $\mathbf{HasMember}$ LOR.LAND $\mathbf{IsLocatedIn}$ computed over the *"logical or.logical and"* semiring returns a matrix representing the Places where a Forum's members are located in. Meanwhile, when computed over the conventional arithmetic *"plus.times"* semiring, $\mathbf{HasMember} \oplus.\otimes \mathbf{IsLocatedIn}$ also returns the number of such Persons. A traversal from a certain set of vertices can be expressed by using a boolean vector $\mathbf{f}$ (often referred to as the *frontier*, *wavefront*, or *queue*) and setting `true` values for the elements corresponding to source vertices. For example, for Forums $\mathbf{f} \in \mathbb{B}^{|forums|}$, $\mathbf{f}$ LOR.LAND $\mathbf{HasMember}$ returns the Persons who belong to any of the forums in $\mathbf{f}$. The BFS navigation step can also be captured using other semirings such as LOR.FIRST, where $\texttt{FIRST}(x, y) = x$; LOR.SECOND, where $\texttt{SECOND}(x, y) = y$; and ANY.PAIR, where $\texttt{ANY}(x, y)$ returns either x or y, and $\texttt{PAIR}(x, y) = 1$ [11].

Notation. Table I contains the list of GraphBLAS operations used in this paper. Additionally, we use $\mathbf{D} = \text{diag}(I, n)$ to construct a diagonal matrix with $\mathbf{D}(i, i) = 1$ for $i \in I$. For a more detailed overview of GraphBLAS, see [16] and [17].

IV. Building Blocks in GraphBLAS

A. Dense Vertex Relabelling

The vertices in the generated input graphs have *sparse IDs*, i.e. identifiers which can take any UINT64 value. To map a set of n sparse IDs to *dense IDs* which take up consecutive values in the $[0, n-1]$ range, we need to perform *dense vertex relabelling* [27], also known as *vertex permutation* [4] and *mapping from sparse to dense keys* [21]. A straightforward way to implement this mapping from an array of sparse IDs *sparseids* is to create a sparse vector as follows:

$$\mathbf{mapping} \leftarrow \{sparseids, [0, 1, \ldots, n-1]\}$$

Given a sparse ID s, the GraphBLAS extract element operation $d = \mathbf{mapping}(s)$ returns the corresponding dense ID d. Meanwhile, mapping from a dense ID d to a sparse ID can be performed trivially with an array lookup $sparseids[d]$.

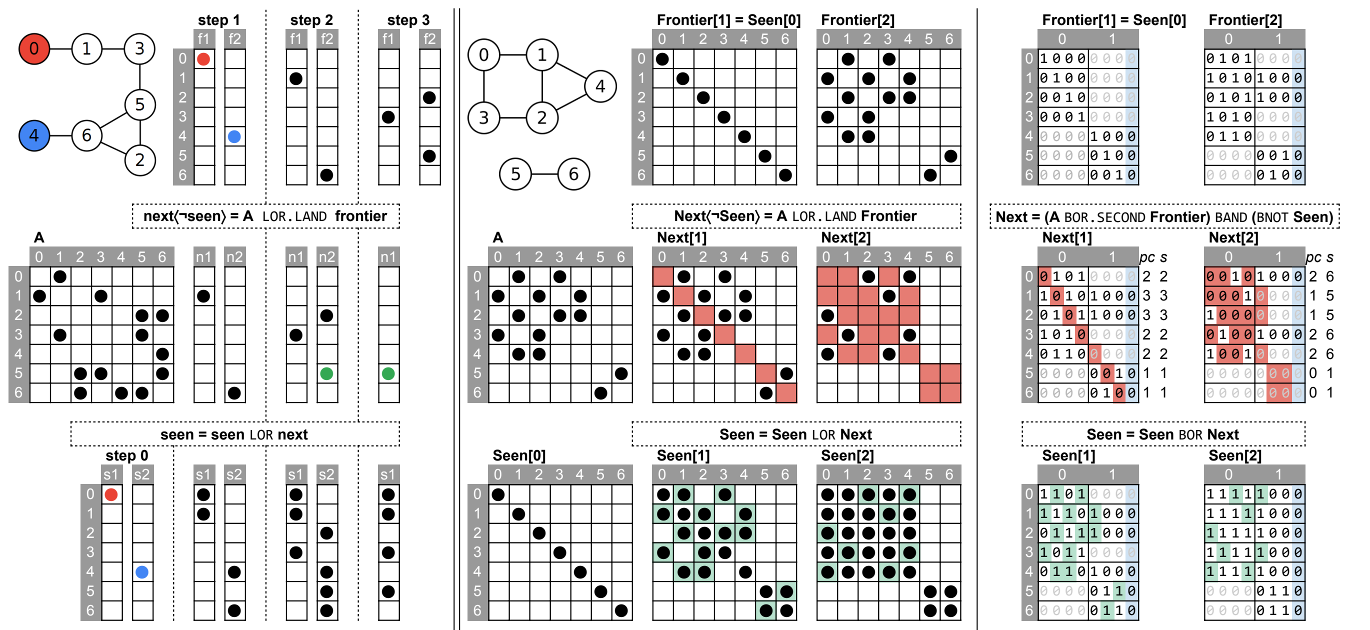
Implementation. Sparse vectors in SuiteSparse:GraphBLAS are stored with their indices in increasing order, therefore this step requires a sorting operation. Then, lookups are executed using a binary search among the vector's indices (of non-zero values). In the rest of the paper, we assume that identifiers have already been relabelled to dense.

Algorithm 1 Bidirectional BFS algorithm.

```
1: procedure BidirectionalBFS($\mathbf{A}, v_1, v_2$)
2:     Input: $\mathbf{A} \in \mathbb{B}^{n \times n}; v_1, v_2 \in \mathbb{N}$
3:     Data: $\mathbf{frontier1}, \mathbf{frontier2}, \mathbf{next1}, \mathbf{next2}, \mathbf{seen1}, \mathbf{seen2} \in \mathbb{B}^n$
4:     Output: $length \in \mathbb{N}$                ▷ length of unweighted shortest path
5:     if $v_1 = v_2$ then return $0$
6:     $\mathbf{frontier1}, \mathbf{seen1} \leftarrow \{[v_1], [\texttt{true}]\}$
7:     $\mathbf{frontier2}, \mathbf{seen2} \leftarrow \{[v_2], [\texttt{true}]\}$
8:     for $level = 1$ to $\lceil n/2 \rceil$ do
9:         $\mathbf{next1}\langle\neg\mathbf{seen1}\rangle = \mathbf{A}$ LOR.LAND $\mathbf{frontier1}$
10:        if $\mathbf{next1}$ is empty then return no path found
11:        if $\mathbf{next1}$ LAND $\mathbf{next2}$ is not empty then return $2 \times level - 1$
12:        $\mathbf{next2}\langle\neg\mathbf{seen2}\rangle = \mathbf{A}$ LOR.LAND $\mathbf{frontier2}$
13:        if $\mathbf{next2}$ is empty then return no path found
14:        if $\mathbf{next1}$ LAND $\mathbf{next2}$ is not empty then return $2 \times level$
15:        $\mathbf{seen1} = \mathbf{seen1}$ LOR $\mathbf{next1}$
16:        $\mathbf{seen2} = \mathbf{seen2}$ LOR $\mathbf{next2}$
17:        $\mathbf{frontier1} = \mathbf{next1}$
18:        $\mathbf{frontier2} = \mathbf{next2}$
```

B. Bidirectional Search

Both queries 1 and 3 require *bidirectional search*: the former searches for the shortest path between two Persons (where each pair of Persons along the path edge satisfies a constraint on the number of interactions), while the latter looks for pairs of Persons who are at most h hops away. Bidirectional search



(a) Bidirectional BFS from vertices 0 and 4. (b) Boolean MSBFS. (c) Bitwise MSBFS.

Fig. 2: Example BFS executions: bidirectional BFS and MSBFS algorithms. MSBFS algorithms start from all vertices. *Notation:* 🟥 blocked by mask ¬**Seen** (boolean) or by the `0` bit in `BNOT` **Seen** (bitwise), 🟩 new non-zero value/bit added, 🟦 `0`-padding.

Algorithm 2 Boolean all-source MSBFS algorithm.

```
 1: procedure BooleanMSBFS(A)
 2:     Input: $\mathbf{A} \in \mathbb{B}^{n\times n}$
 3:     Data:
 4:         Frontier, Seen $\in \mathbb{B}^{n\times n}$          ▷ initialized to all vertices
 5:         Next $\in \mathbb{B}^{n\times n}$                    ▷ initially empty
 6:     for $k = 0$ to $n-1$ do                                  ▷ initialize diagonal matrix
 7:         $I[k] = k,\ X[k] = $ true
 8:     Frontier, Seen $\leftarrowtail \{I, I, X\}$
 9:     for level = 1 to $n-1$ do
10:         Next⟨¬Seen⟩ = A LOR.LAND Frontier
11:         if Next is empty then break
12:         Frontier = Next
13:         Seen = Seen LOR Next
```

Algorithm 3 Bitwise all-source MSBFS algorithm. *Notation: popcount* is a unary operator that counts the number of bits in an `UINT64` value. Lines 18–21 compute the CCV (cf. Alg. 7).

```
 1: procedure BitwiseMSBFS(A)
 2:     Input: $\mathbf{A} \in \mathbb{B}^{n\times n}$
 3:     Data:
 4:         $I, J, X \in$ UINT64$^n$
 5:         Frontier, Next, Seen $\in$ UINT64$^{n\times\lceil n/64\rceil}$
 6:         sp $\in \mathbb{N}^n$, cpmo $\in [0,1]^n$                  ▷ $s(p), C(p)-1$
 7:     Output: ccv $\in [0,1]^n$                                    ▷ closeness centrality values
 8:     for $k = 0$ to $n-1$ do                                      ▷ initialize bit diagonal matrix
 9:         $I[k] = k,\ J[k] = k/64,\ X[k] = 1 \ll (64 - (k \bmod 64))$
10:     Frontier, Seen $\leftarrowtail \{I, J, X\}$
11:     for level = 1 to $n-1$ do
12:         Next = A BOR.SECOND Frontier
13:         Next⟨Next⟩ BAND= BNOT(Seen)
14:         Next = GxB_NONZERO(Next)                                 ▷ prune explicit zeros
15:         if Next is empty then break
16:         Frontier = Next
17:         Seen = Seen BOR Next
18:         nextCount = $[\oplus_j popcount(\mathbf{Next})(:, j)]$
19:         sp $\oplus$= nextCount $\times$ level
20:     cpmo = $[\oplus_j popcount(\mathbf{Seen})(:, j)] - 1$            ▷ $C(p)-1$
21:     return (cpmo $\otimes$ cpmo) $\oslash$ $((n-1) \otimes$ sp)      ▷ $\frac{(C(p)-1)^2}{(n-1)\cdot s(p)}$
```

in GraphBLAS can be implemented as two alternating BFS traversals as shown in Alg. 1 and illustrated in Fig. 2a. To perform a search performed between vertices v_1 and v_2, we initialize two **frontier** vectors, each containing one non-zero value at position v_1 and v_2, respectively. In each iteration l, we advance the first frontier and check whether its intersection with the current (not yet advanced) state of the second frontier contains any elements. If so, we found a path of length $2\times l-1$. If not, we advance the second frontier and intersect it with the first frontier. If the intersection has any elements, we found a path of length $2\times l$. If the new frontier is empty in either case, no path can be found between vertices v_1 and v_2.

C. Multi-Source Breadth-First Search

Boolean MSBFS. Highly-optimized multi-source BFS algorithms have been used by multiple teams in the programming contest [26], [15] to efficiently evaluate queries 3 and 4. In the GraphBLAS community, it is established that *matrix-matrix multiplication* is a natural and efficient way to express MSBFS [29]. Using this idea, a GraphBLAS-based MSBFS algorithm is shown in Alg. 2 and illustrated in Fig. 2b.

Bitwise MSBFS. A key optimization among top-ranking teams was using *bit arrays and bitwise manipulations* to improve the performance of MSBFS. With the recent introduction of *bitwise operators* (e.g. `GrB_BAND`) in GraphBLAS v1.3 [6], it is possible to use this optimization for MSBFS, shown in Alg. 3.

V. Queries

Query 1. Our implementation for Q1 (Alg. 4) first determines the induced subgraph. If the threshold x for the number of interactions is -1, we use the $\mathbf{Knows}$ matrix, otherwise, we produce a matrix by traversing the `hasCreator` and `replyOf` edges, then filter for values larger than x. The length of the shortest path between the selected Persons is determined by running a bidirectional BFS (Alg. 1) on the induced subgraph.

Query 2. In Alg. 5, we first create a mask that corresponds to Persons born on d or later. We then iterate through each Tag (implemented as an outer OpenMP parallel for loop in our code), and determine Persons who are interested in said Tag and are kept by the mask. We extract the `knows` subgraph and compute connected components using the FastSV algorithm [30], then determine the size of the largest component. Finally, we return the top-k tags based on their component size.

Query 3. In Alg. 6, we first compute the "local" Persons for the given *place*. The query requires us to determine pairs of Persons whose distance in the `knows` graph is at most h hops. However, instead of evaluating the h-neighbourhood for each Person (which can get excessively large due to the exponential growth of the frontier), we look for "meeting vertices" ($\mathbf{meetings}$) that are reachable from both Persons in at most $\lfloor h/2 \rfloor$ steps. Therefore, we initiate an MSBFS from all Persons (Lines 5–9) and run it for $\lfloor h/2 \rfloor$ iterations. In the resulting $\mathbf{Seen}$ matrix, row i represents the vertices reachable from Person i. A column j with more than one non-zero value captures a "meeting vertex". For example, non-zero values in elements $\mathbf{Seen}(x, j)$ and $\mathbf{Seen}(y, j)$ imply that j is a meeting vertex between Persons x and y. We enumerate the pairs of Persons for each meeting vertex (concurrently) and store them in matrix $\mathbf{CommonInt}$ which represents the common interests of Persons. Finally, we return top-k maximum values.

Since MSBFS steps move all frontiers simultaneously, special care needs to be taken for odd h values to ensure that the meeting vertex is at most $\lfloor h/2 \rfloor$ steps from one local Person and $\lfloor h/2 \rfloor + 1$ steps from another one (yielding a total value of h). Therefore, once we performed the first $\lfloor h/2 \rfloor$ steps, we do an extra MSBFS step to advance the frontier and set the found elements to value 0.5 (Lines 10–12). Then, we filter for columns with a value larger than 1 (Line 13), thus omitting meeting vertices that only have two 0.5 values. Finally, we check the corresponding columns in $\mathbf{Seen}$ and only keep pairs of Persons whose summed values are larger than 1 (Line 16).

Query 4. Our solution for Q4 (Alg. 7) first navigates from the given Tag t to its Forums then to member Persons of such Forums. Then, it selects the corresponding rows/columns of the $\mathbf{Knows}$ matrix and computes the CCV values using a bitwise MSBFS algorithm (Alg. 3).

Algorithm 4 Implementation of Q1 using Alg. 1.

```
Input: $p_1, p_2 \in \mathbb{N}$                      ▷ target persons
    $x \in \{-1, 0, 1, 2, \ldots\}$                    ▷ threshold
Output: $l \in \mathbb{Z}^+$     ▷ length of the shortest path between $p_1$ and $p_2$
if $x = -1$ then
    $\mathbf{A} = \mathbf{Knows}$
else
    $\mathbf{PA2Comment} = \mathbf{HasCreator}^\top \oplus.\otimes \mathbf{ReplyOf}$
    $\mathbf{PA2PB}\langle\mathbf{Knows}\rangle = \mathbf{PA2Comment} \oplus.\otimes \mathbf{HasCreator}$
    $\mathbf{A} = \mathit{GreaterThan}(\mathbf{PA2PB}, x)$
    $\mathbf{A} = \mathbf{A}$ LAND $\mathbf{A}^\top$   ▷ ensure that $x$ interactions happened both ways
return BidirectionalBFS($\mathbf{A}, p_1, p_2$)
```

Algorithm 5 Implementation of Q2. For ConnectedComponents, we use the FastSV algorithm [30].

```
Input
    $k \in \mathbb{N}, d \in$ Date   ▷ number of top-$k$ tags, lower bound for birthdays
    $\mathbf{birthDay} \in \text{Date}^{|persons|}$
Output: $[t_1, \ldots, t_k] \in \mathbb{N}^k$                ▷ top-$k$ tags
$\mathbf{birthDayMask} = \mathit{GreaterThanOrEq}(\mathbf{birthDay}, d)$
for $t = 0$ to $|tags| - 1$ do
    $\mathbf{interestedPerson}\langle\mathbf{birthDayMask}\rangle = \mathbf{HasInterest}(:, t)$
    $\{P_t, \_\} = \mathbf{interestedPerson}$
    $\mathbf{Knows\_t} = \mathbf{Knows}(P_t, P_t)$
    $\mathbf{component\_ids} = \text{ConnectedComponents}(\mathbf{Knows\_t})$
    $scores[t] = (\max_{i \in \mathbf{component\_ids}} \text{count}_i(\mathbf{component\_ids}), t)$
return top-$k$ tags $t$ based on their $score$ from $scores$
```

Algorithm 6 Implementation of Q3 using Alg. 2. The call GetPersonsForPlace returns Persons for a given Place.

```
Input: $place \in \mathbb{N}, k \in \mathbb{N}$          ▷ place, number of top tags to return
Data: $\mathbf{Frontier}, \mathbf{Next} \in \mathbb{B}^{n \times n}, \mathbf{Seen} \in \mathbb{Q}^{n \times n}$
Output: $\{(p_{1,1}, p_{2,1}), \ldots, (p_{1,k}, p_{2,k})\} \subset \mathbb{N}^k \times \mathbb{N}^k$     ▷ top-$k$ pairs
$\mathbf{localPersons} = \text{GetPersonsForPlace}(place)$      ▷ details omitted
$\mathbf{Frontier}, \mathbf{Seen} = \text{diag}(\mathbf{localPersons}, n)$
for $level = 1$ to $\lfloor h/2 \rfloor$ do
    $\mathbf{Next}\langle\neg\mathbf{Seen}\rangle = \mathbf{Frontier}$ LOR.LAND $\mathbf{Knows}$
    $\mathbf{Seen} = \mathbf{Seen}$ LOR $\mathbf{Next}$
    $\mathbf{Frontier} = \mathbf{Next}$
if $h$ is an odd number then
    $\mathbf{Next}\langle\neg\mathbf{Seen}\rangle = \mathbf{Frontier}$ LOR.LAND $\mathbf{Knows}$
    $\mathbf{Seen}\langle\mathbf{Next}\rangle = 0.5$                  ▷ mark the last frontier
$\mathbf{meetings} = \mathit{GreaterThan}([\oplus_i \mathbf{Seen}(i, :)], 1)$        ▷ pruning
for all $j$ where $\mathbf{meetings}(j)$ is non-zero do
    $\{I, X\} \leftarrow \mathbf{Seen}(:, j)$
    for all pairs of person indices $p_1, p_2 \in I$ where $p_1 < p_2$ do
        if $X[p_1] + X[p_2] > 1$ then
            $\mathbf{CommonInt}(p_1, p_2) = 0$       ▷ common interest count
$\mathbf{CommonInt}\langle\mathbf{CommonInt}\rangle = \mathbf{HasInterest} \oplus.\otimes \mathbf{HasInterest}^\top$
return top-$k$ $(p_1, p_2)$ pairs where $p_1 < p_2$ in $\mathbf{CommonInt}$
```

Algorithm 7 Implementation of Q4 using Alg. 3.

```
Input: $k \in \mathbb{N}, t \in \mathbb{N}$        ▷ number of top persons to return, tag id
Output: $l \in [0, 1]$                       ▷ closeness centrality value
$\{I, \_\} = (\mathbf{HasTag}(:, t)$ LOR.LAND $\mathbf{HasMember})$
$\mathbf{A} = \text{diag}(I, n)$ LOR.LAND $\mathbf{Knows}$ LOR.LAND $\text{diag}(I, n)$
return BitwiseMSBFS($\mathbf{A}$)
```

VI. Evaluation

A. Benchmark Setup

Goal. We designed an experiment to compare the performance and scalability of our implementation against the top solutions.

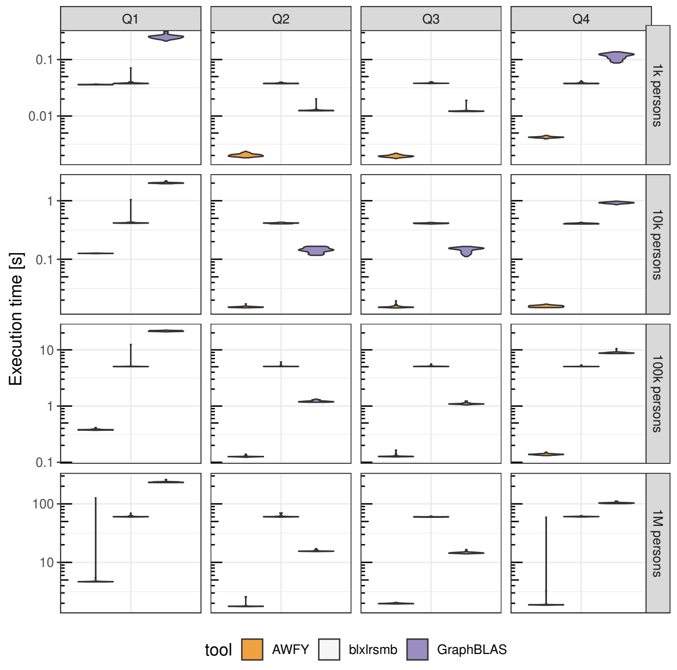
data set (#persons)	1k	10k	100k	1M
total vertices	611 434	6 389 475	63 391 437	631 648 214
total edges	1 910 684	19 769 080	195 547 183	1 948 640 608

TABLE II: Number of entities in the input graphs.

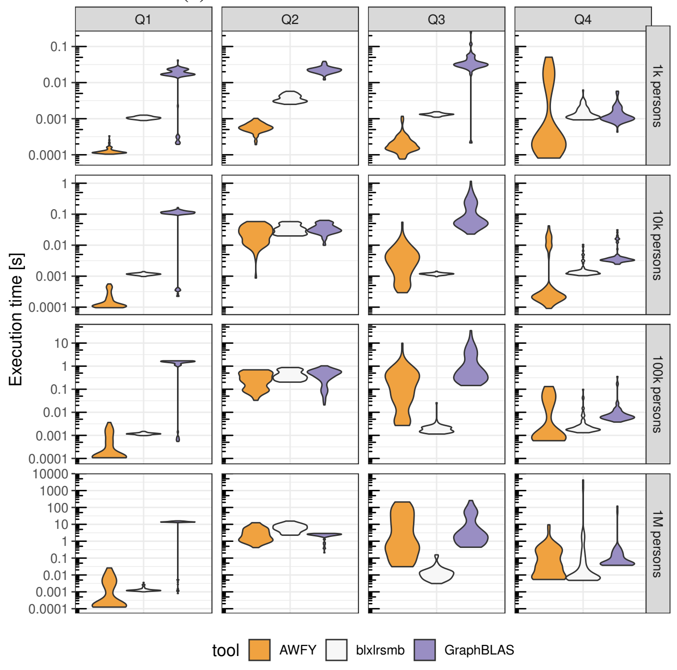
Solutions. We implemented our solution in C++ using version v3.3.3 of SuiteSparse:GraphBLAS and the latest LAGraph library [19].[2] We compared our implementation against two solutions of the programming contest created by teams "AWFY" (ranked 1st) and "blxlrsmb" (ranked 4th), updated for GCC 9.[3]
Data sets. We were unable to obtain the data sets used in the contest, therefore we generated similar ones using the LDBC SNB Datagen's [22] 2014 version.[4] The data set statistics are shown in Table II. We also implemented a parameter generator that produces query input parameters using uniform sampling.
Environment. We performed the experiments on a cloud virtual machine with 32 (logical) Intel Xeon Skylake CPU cores clocked at 2GHz, 120GB RAM, and SSD storage, running Ubuntu 20.04. We used the GCC 9.3.0 compiler. We ran each benchmark with 80 different parameters.

B. Analysis

Performance. We visualized the distribution of the execution times obtained during the experiments: load times are shown in Fig. 3a and computation times in Fig. 3b. For *load times*, other solutions consistently outperform our solution. AWFY provides particularly fast loads (which can be attributed to its usage of advanced CSV loading techniques presented in [21]). However, for *computation times*, our solution is competitive for Q2 (often outperforming other solutions). It provides good performance for Q3 and Q4, staying within an order of magnitude compared to the other highly-optimized solutions. Our solution exhibits a bimodal distribution for Q1, which can be attributed to the configuration parameter x: for $x > -1$, computing the induced subgraph is expensive. The execution times of our solution for Q1 are noticeably longer than the competition's due to precomputing the entire induced graph (instead of computing the relevant edges on-the-fly during traversal).
Conciseness. We characterized the conciseness of each solution using lines of C++ code: AWFY consisted of 9,800 lines, blxlrsmb used 6,500 lines, while our code used 3,500 lines.
Threats to validity. We remark that the solutions used in the experiments are 6+ years old and might be improved by further optimizations that were unavailable at the time of the contest. We have updated the solutions to GCC 9 but did not apply any further optimizations nor did we contact their original authors.



(a) Load times of the solutions.



(b) Computation times of the solutions.

Fig. 3: Execution times of queries.

VII. Conclusion and Future Work

We presented a concise GraphBLAS solution for the 2014 SIGMOD Programming Contest's queries. Even though our code has no direct calls to low-level CPU instructions (unlike the top contest solutions), its computation performance for queries 2, 3, and 4 on large graphs is within 1–2 orders of magnitude to the top solutions. A limitation of our current solution is an expensive load step, which often takes longer the computation itself. For this reason, our solution works best in cases when multiple queries are performed on the same graph.

As future work, we plan to apply direction-optimization [2], also known as push/pull [28] to the bidirectional and MSBFS algorithms, and improve the load performance by using a concurrent hashmap such as Folklore [18].

[2]https://github.com/ldbc/sigmod2014-pc-graphblas
[3]https://github.com/ftsrg/sigmod2014-pc-top-solutions
[4]https://github.com/ldbc/ldbc_snb_datagen/releases/tag/early2014

Acknowledgements

The research has received funding from the EU ECSEL JU under the H2020 Framework Programme, JU grant nr. 826452 (Arrowhead Tools project) and from the partners' national funding authorities. Davis would like to acknowledge NSF (1514406, 1835499), Intel, NVIDIA, IBM, Redis Labs, and MIT LL for their support. The authors would like to thank members of the GraphBLAS mailing list, Arnau Prat-Pérez, and Dean de Leo for discussions.

References

[1] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5):68:1–68:40, 2017.

[2] S. Beamer, K. Asanovic, and D. A. Patterson. Direction-optimizing breadth-first search. In *SC*. IEEE/ACM, 2012.

[3] S. Beamer, K. Asanovic, and D. A. Patterson. The GAP Benchmark Suite. *CoRR*, abs/1508.03619, 2015. http://arxiv.org/abs/1508.03619.

[4] M. Besta and T. Hoefler. Survey and taxonomy of lossless graph compression and space-efficient graph representations. *CoRR*, abs/1806.01799, 2018. http://arxiv.org/abs/1806.01799.

[5] M. Besta, E. Peter, R. Gerstenberger, M. Fischer, M. Podstawski, C. Barthels, G. Alonso, and T. Hoefler. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *CoRR*, abs/1910.09017, 2019. http://arxiv.org/abs/1910.09017.

[6] A. Buluç, T. Mattson, J. E. Moreira, S. McMillan, and C. Yang. The GraphBLAS C API specification. Version 1.3.0, 2019. https://people.eecs.berkeley.edu/~aydin/GraphBLAS_API_C_v13.pdf.

[7] P. Cailliau, T. Davis, V. Gadepally, J. Kepner, R. Lipman, J. Lovitz, and K. Ouaknine. RedisGraph GraphBLAS enabled graph database. In *GrAPL at IPDPS*, pages 285–286. IEEE, 2019.

[8] G. A. Chernishev, V. Sevostyanov, K. Smirnov, and I. Shkuratov. On several social network analysis problems. In *All-Russian Scientific Conference "Digital libraries: Advanced Methods and Technologies, Digital Collections"*, pages 234–242, 2014.

[9] T. A. Davis. Graph algorithms via SuiteSparse:GraphBLAS: Triangle counting and K-truss. In *HPEC*. IEEE, 2018.

[10] T. A. Davis. Algorithm 1000: SuiteSparse:GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Trans. Math. Softw.*, 2019.

[11] T. A. Davis. User guide for SuiteSparse:GraphBLAS, version 3.3.3, 2020. https://people.engr.tamu.edu/davis/GraphBLAS.html.

[12] M. Elekes and G. Szárnyas. An incremental GraphBLAS solution for the 2018 TTC Social Media case study. 2020. GrAPL at IPDPS.

[13] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In *SIGMOD*, pages 1433–1445. ACM, 2018.

[14] F. T. Jamour, I. Abdelaziz, Y. Chen, and P. Kalnis. Matrix algebra framework for portable, scalable and efficient query engines for RDF graphs. In *EuroSys*, pages 27:1–27:15. ACM, 2019.

[15] M. Kaufmann, M. Then, A. Kemper, and T. Neumann. Parallel array-based single- and multi-source breadth first searches on large dense graphs. In *EDBT*, 2017.

[16] J. Kepner et al. Mathematical foundations of the GraphBLAS. In *HPEC*. IEEE, 2016.

[17] M. Kumar, J. E. Moreira, and P. Pattnaik. GraphBLAS: Handling performance concerns in large graph analytics. In *CF*, pages 260–267. ACM, 2018.

[18] T. Maier, P. Sanders, and R. Dementiev. Concurrent hash tables: Fast and general(?)! *ACM Trans. Parallel Comput.*, 5(4):16:1–16:32, 2019.

[19] T. Mattson, T. A. Davis, M. Kumar, A. Buluç, S. McMillan, J. E. Moreira, and C. Yang. LAGraph: A community effort to collect graph algorithms built on top of the GraphBLAS. In *GrAPL at IPDPS*, 2019.

[20] T. G. Mattson, C. Yang, S. McMillan, A. Buluç, and J. E. Moreira. Graphblas C API: Ideas for future versions of the specification. In *HPEC*. IEEE, 2017.

[21] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann. Instant loading for main memory databases. *VLDB*, 6(14):1702–1713, 2013.

[22] M. Pham, P. A. Boncz, and O. Erling. S3G2: A scalable structure-correlated social graph generator. In *TPCTC*, pages 156–172. Springer, 2012.

[23] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The ubiquity of large graphs and surprising challenges of graph processing: Extended survey. *VLDB J.*, 29(2):595–618, 2020.

[24] S. Sakr, S. Elnikety, and Y. He. G-SPARQL: A hybrid engine for querying large attributed graphs. In *CIKM*, pages 335–344, 2012.

[25] B. Shao, Y. Li, H. Wang, and H. Xia. Trinity graph engine and its applications. *IEEE Data Eng. Bull.*, 40(3):18–29, 2017.

[26] M. Then, M. Kaufmann, F. Chirigati, T. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo. The more the merrier: Efficient multi-source graph traversal. *PVLDB*, 8(4):449–460, 2014.

[27] M. Then, M. Kaufmann, A. Kemper, and T. Neumann. Evaluation of parallel graph loading techniques. In *GRADES at SIGMOD*. ACM, 2016.

[28] C. Yang, A. Buluç, and J. D. Owens. Implementing push-pull efficiently in GraphBLAS. In *ICPP*, pages 89:1–89:11. ACM, 2018.

[29] C. Yang, A. Buluç, and J. D. Owens. GraphBLAST: A high-performance linear algebra-based graph framework on the GPU. *CoRR*, abs/1908.01407, 2019. http://arxiv.org/abs/1908.01407.

[30] Y. Zhang, A. Azad, and Z. Hu. FastSV: A distributed-memory connected component algorithm with fast convergence. In *PP*. SIAM, 2020.