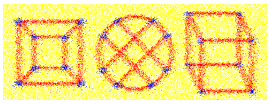
Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 25, no. 1, pp. 241–261 (2021)
DOI: 10.7155/jgaa.00558

Algebraic Algorithms for Betweenness and Percolation Centrality

Altansuren Tumurbaatar[1] *Matthew J. Sottile*[1,2]

[1]Department of Mathematics and Statistics, Washington State University, USA
[2]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, USA

Submitted: February 2021 Accepted: March 2021 Final: March 2021
Published: April 2021
Article type: Regular paper Communicated by: G. Liotta

Abstract. In this paper, we explored different ways to write the algebraic version of betweenness centrality algorithm. Particularly, we focused on Brandes' algorithm [8]. We aimed for algebraic betweenness centrality that can be parallelized easily. We proposed 3-tuple geodetic semiring as an extension to the usual geodetic semiring with 2-tuples [6]. Using the 3-tuple geodetic semiring, Dijkstra's and Brandes' algorithm, we wrote more concise and general algebraic betweenness centrality (ABC) algorithm which is valid for weighted and directed graphs. We also proposed an alternative version of ABC using the usual geodetic semiring with 2-tuple where we used a simple way to construct shortest path tree after computing shortest path distances in the usual geodetic semiring. This allows us to avoid computational complexity of ABC implementation using 3-tuple geodetic semiring. We used numba [18] to optimize and parallelize ABC. We evaluated the performance of ABC using 2-tuple geodetic semiring as compared to NetworkX [16], a common python package for graph algorithms. We did scalability experiments on parallel ABC and showed its total speedup. We also showed that with small modification, ABC can be adapted to algebraicly compute other centrality measures such as percolation centrality.

1 Introduction

Algorithms for computing centrality measures on graphs have received a great deal of attention in recent decades due to problems of interest across fields in which data has a graph structure. Centrality measures often help answer domain-specific questions about the importance or relevance of individual vertices in a graph. The use of parallel computing is of interest for calculating

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

E-mail addresses: altaa.tumurbaatar@wsu.edu (Altansuren Tumurbaatar) sottile2@llnl.gov (Matthew J. Sottile)

graph features when the problem size exceeds the capabilities of classical sequential or small-scale multicore CPUs. One mechanism for implementing such algorithms on parallel computers is to express them in the form of linear algebra operations that naturally map onto a data-parallel programming model. The GraphBLAS [1] and Graph500 [2] are recent notable efforts that adopt this model of graph algorithms via linear algebra for parallel computation. This is particularly relevant with the recent trend towards data parallel accelerators like graphics processing units.

In this paper we introduce and analyze algebraic algorithms for computing betweenness centrality measures using the tools of linear algebra. We demonstrate how this can be applied to other centrality measures that are based on betweenness centrality by defining an algebraic implementation of the percolation centrality algorithm [20].

1.1 Contribution

Our work contributes the following results to the existing field of graph algorithms in the language of linear algebra:

- Our algorithm for algebraic betweenness centrality (ABC) is valid for weighted and/or directed graphs,
- ABC is fully algebraic via a geodetic semiring, avoiding non-algebraic auxiliary functions,
- ABC forms the basis of other centrality measures expressed in terms of linear algebra, such as percolation centrality.
- We propose a 3-tuple geodetic semiring, extending a more commonly used geodetic semiring by adding parent information.

Betweenness centrality is a commonly studied graph algorithm and has been approached with the tools of linear algebra in the past. Early work focused on the unweighted case: [21] first formulated algebraic algorithm for betweenness centrality on unweighted graphs using linear algebra primitives and [9] implemented it as the distributed-memory betweenness centrality algorithm. As [9] noted their implementation can scale beyond thousands of processors, but only addresses the unweighted case. Shortly afterwards, [12] proposed a new parallelizable algorithm for the general case with low spatial complexity scaling to 64 processors on both weighted and unweighted graphs, but it wasn't algebraic.

The closest work to ours appears in [23], which describes the Maximal Frontier Betweenness Centrality (MFBC) method based on sparse matrix multiplication for both unweighted and weighted graphs using a parallel (distributed-memory) numerical library for multidimensional arrays called Cyclops [24]. MFBC is based on Bellman-Ford and Brandes' algorithms [8] and its overall logic is similar to the algorithm ABC-DB that we propose. Unlike MFBC, ABC-DB is based on Dijkstra's and Brandes' algorithms, and fully formulated based on linear algebra primitives. For example, MFBC uses a combination of different algebraic monoids and auxiliary functions to define matrix-vector and matrix-matrix multiplication. ABC-DB works with matrices and vectors from a single domain, referred to as the 3-tuple geodetic semiring. The semiring operations are used to implement the necessary logic of the algorithm which results a simple algebraic implementation of Brandes' algorithm for weighted and unwieghted graphs. MFBC uses matrix-vector multiplication with a monoid and an auxiliary function to find frontiers for back-propagation and centrality accumulation because it is based on Bellman-Ford. Tracking frontiers is not necessary with ABC-DB due to the order of traversal from Dijkstra's algorithm and parent information in the shortest path

tree from the geodetic semiring. This allows us to back-propagate centrality scores simply. Thus, ABC-DB only uses vector operations to back-propagate and accumulate centralities.

1.2 Graph notation and background

Let $G = (V, E)$, $E \subseteq V \times V$ be a graph with the set of nodes and the set of edges denoted as V and E respectively where each edge has a weight $w : E \to \overline{\mathbb{R}} = \mathbb{R} \cup (\infty)$. We denote $|V| = n$ and $|E| = m$. The following concepts on a graph such as a path, shortest path and shortest path distance are briefly defined as follows. These will be used later in this paper to understand the graph centrality algorithms.

Definition 1.1 (Path) *A path p from v_0 to v_k, denoted as $v_0 \overset{p}{\rightsquigarrow} v_k$ is a sequence of vertices $p = (v_0, v_1, \ldots, v_k)$ such that $(v_{i-1}, v_i) \in E$ and the weight of a path is defined as*

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i),$$

the sum of the weights of the edges which constitutes the path.

Definition 1.2 (Shortest path distance) *The shortest path distance from vertex s to t is*

$$d(s,t) = \begin{cases} \min\{w(p) \mid s \overset{p}{\rightsquigarrow} t\} & \text{if there exists a path from } s \text{ to } t \\ \infty & \text{otherwise} \end{cases} \tag{1}$$

Definition 1.3 (Shortest path) *A path, p from s to t, is a shortest path from s to t if $w(p) = d(s,t)$.*

1.3 Betweenness Centrality

In complex networks, we need some measure to tell which nodes play more important roles. One such measure is betweenness centrality. The betweenness centrality of a vertex represents what portion of all shortest paths in the graph go through the vertex. Let σ_{st} be the number of shortest paths between vertices s and t and $\sigma_{st}(v)$ be the number of shortest paths between s and t that includes the vertex v. Then betweenness centrality is defined as

$$C_B(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{2}$$

Intuitively, betweenness centrality of a vertex represents the change of the number of shortest paths in the network if we remove a vertex from the graph. The larger betweenness centrality of a vertex, the bigger change in total shortest paths. Another interpretation of betweenness centrality of a vertex is the probability that communication between any two vertices in a network goes through v, assuming all communications in the network happen through shortest paths. This assumption could become disadvantageous depending on a context. There are several terms related to betweenness centrality including Bellman criterion, pair dependency and dependency of a vertex which are defined as follows [8].

Lemma 1 (Bellman criterion) *A vertex $v \in V$ lies on a shortest path between vertices $s, t \in V$, if and only if $d(s,t) = d(s,v) + d(v,t)$.*

By the Bellman criterion the shortest path counts between $s, t \in V$ passing through $v \in V$ is

$$\sigma_{st}(v) = \begin{cases} 0 & \text{if } d(s,t) < d(s,v) + d(v,t) \\ \sigma_{sv} \cdot \sigma_{vt} & \text{otherwise} \end{cases} \tag{3}$$

which can be used to find the pair dependency of s, t on v, $\delta_{st}(v)$.

Definition 1.4 (pair dependency) *The pair dependency of $s, t \in V$ on an intermediary vertex, $v \in V$ is the portion of shortest paths between s and t that pass through v, defined as:*

$$\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{4}$$

If we sum all pair dependencies on v, we get the betweenness centrality of v:

$$C_B(v) = \sum_{s \neq v \neq t} \delta_{st}(v) \tag{5}$$

Definition 1.5 (dependency) *The dependency of a vertex $s \in V$ on a single vertex $v \in V$ is the portion of shortest paths starting from s and going through v, that is defined as follows.*

$$\delta_{s\bullet}(v) = \sum_{t \in V} \delta_{st}(v) \tag{6}$$

If we sum dependencies of all s on v, we get betweenness centrality of v as follows.

$$C_B(v) = \sum_{s \neq v} \delta_{s\bullet}(v) \tag{7}$$

Brandes proved that the dependency of s on v follows a recursive relation.

Theorem 1 (Recursive relation of dependency [8]) *The dependency of $s \in V$ on any $v \in V$ obeys*

$$\delta_{s\bullet}(v) = \sum_{w : v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_{s\bullet}(w)) \tag{8}$$

where $P_s(w)$ is the set of parents of a vertex w on shortest paths from s to w.

1.4 Semiring

A semiring $(S, \oplus, \otimes, 0, 1)$ is a set S with two binary operations $\oplus$ and $\otimes$ defined on S such that the following properties are satisfied

1. $\oplus$ is associative and commutative,
2. $\otimes$ is associative,
3. $\otimes$ distributes over $\oplus$,
4. 0 is an absorbing element (annihilator) under $\otimes$ such that $a \otimes 0 = 0 \otimes a = 0$.

A semiring $(S, \oplus, \otimes, 0, 1)$ is complete if it satisfies the following two conditions [4]:

- If $a_1, a_2, a_3, \ldots$ is a countable sequence of elements in S, then $a_1 \oplus a_2 \oplus a_3 \oplus \ldots$ exists and is unique.
- The commutativity, associativity and distributivity holds for countable cases as well.

A semiring $(S, \oplus, \otimes, 0, 1)$ is closed iff $a^\star = 1 + a \otimes a^\star = 1 + a^\star \otimes a$ where $\star$ is said to be the closure operation. A complete semiring is closed for the transitive closure defined as

$$a^\star = \sum_{k=0}^{\infty} a^k. \tag{9}$$

A matrix semiring over a complete semiring is also complete and closed for

$$A^* = \sum_{k=0}^{\infty} A^k. \tag{10}$$

Therefore, we can compute the transitive closure of the adjacency matrix A of a graph over a given complete semiring $(S, \oplus, \otimes, 0, 1)$ via Equation 10. If we say transitive closure of a graph, it means the transitive closure of the adjacency matrix of a graph. There are well-known works [4, 14, 17, 6, 10] on semirings and [4, 14] present theorems for path problems in graphs including the transitive closure of a graph.

1.5 Related works

Computing betweenness centrality has two parts, to compute All Pairs Shortest Paths (APSP) and to sum all pair-dependencies. The second part has $\Theta(n^3)$ time complexity and takes up a significant amount of computation time and storage space. In 2001, Brandes [8] introduced a faster algorithm which uses Single Source Shortest Paths (SSSP) and the recursive relation in dependency of a vertex to update centrality scores. By doing so, it eliminates computing APSP and calculates centrality scores without storing all shortest paths. As a result, Brandes' algorithm has time complexity of $\mathcal{O}(m + n \log n)$, $\mathcal{O}(nm + n^2 \log n)$ for unweighted and weighted graphs respectively. Since Brandes' algorithm is based on a priority queue data structure, its space complexity is $\mathcal{O}(m)$ and $\mathcal{O}(nm)$ for unweighted and weighted graphs respectively. The use of a priority queue limits parallelization and vectorization opportunities, especially those based on formulating the algorithm algebraically.

Many different semirings have been defined in the past for expressing graph algorithms in algebraic form, such as the tropical semiring for shortest path problems [17]. In 1974, Aho and Hopcroft [4] first proposed using semirings for path finding algorithms such as computation of costs between vertices, the transitive closure of a graph and shortest paths. These use different semirings or ways to define the adjacency matrix of the graph, but the algorithms themselves were not written in algebraic form. While [4] defined a closed semiring with a idempotent property, in 1980 [13] redefined a closed semiring without the idempotent property and proposed a general algorithm for computing costs between vertices. In 1994, Batagelj [6] introduced the geodetic semiring, a nonidempotent closed semiring and computed the transitive closure of a graph using Fletcher's algorithm [13]. Computing the transitive closure of a graph in the geodetic semiring produces the shortest distance and number of shortest paths between each pair of vertices. All these algorithms take $\mathcal{O}(n^3)$ time and they are not written in algebraic form, but they are implicitly algebraic. Given that a matrix formulation is possible for them, one can always write an algebraic version of the algorithms. These foundational algorithms are related to the work presented here because shortest path identification is core to betweenness centrality algorithms.

In the last two decades work on graph algorithms expressed in terms of linear algebra has been an active research area, especially with the rise of data parallel compute accelerators. In 2008, Robinson [21] formulated betweenness centrality based on matrix operations for unweighted graphs. Their work is based on the breadth first search (BFS) linear algebra primitive and included in Kepner's text that summarizes early work in the field [17]. Robinson's algorithm has time complexity of $\mathcal{O}(n^2 + nm)$ and space complexity of $\mathcal{O}(n)$. Robinson's algorithm implements Brandes' algorithm for unweighted graph and uses algebraic BFS (matrix-vector products) for SSSP and keep track of the level of the node in the tree with the source node as a root. To update centrality scores, the tree is traversed up from leaf nodes or from the bottom level and the parents of a node in the shortest path tree is determined from the adjacency matrix when back track. The CombBLAS [9] is an extensible distributed-memory parallel graph library that implements Robinson's algorithm. Most parallelizations of betweenness centralty is based on the BFS primitive [5, 19, 26, 25, 15]. Most recently, Solomonik [23] proposed an algebraic betweenness centrality algorithm called Maximal Frontier Betweenness Centrality (MFBC) for both unweighted and weighted graphs. MFBC is the closest to our work as discussed earlier.

Many algebraic graph algorithms are presented in [17] written in terms of matrix-vector products. Among them, a 3-tuple shortest path semiring is defined and used to express an algebraic Bellman-Ford algorithm. This 3-tuple consists of a path weight, path size and the penultimate vertex. The 3-tuple geodetic semiring that we defined for our work is similar to this semiring.

2 3-tuple Geodetic Semiring

Given $u, v \in V$, we want to represent the shortest path distance, number of shortest paths and a set of penultimate vertices (parents of end nodes of the shortest paths) as a tuple of a form (d, σ, π). Let $S = \overline{\mathbb{R}^+} \times \overline{\mathbb{N}} \times \mathcal{P}(V) \cup \{(\infty, 0, \emptyset), (0, 1, NIL)\}$ be a set where $\overline{\mathbb{R}^+} = \mathbb{R}^+ \cup \{\infty\}$, $\overline{\mathbb{N}} = \mathbb{N} \cup \{\infty\}$ and $(\infty, 0, \emptyset)$, $(0, 1, NIL)$ correspond to nonexistence of shortest paths (no path) and a path from a vertex to itself (self loop) respectively. We want to define addition $\oplus = \min$ and multiplication $\odot = +_{rhs}$ operations in S as follows.

$$\min\{(d_1, \sigma_1, \pi_1), (d_2, \sigma_2, \pi_2)\} = \begin{cases} (d_1, \sigma_1, \pi_1) & if\ d_1 < d_2 \\ (d_1, \sigma_1 + \sigma_2, \pi_1 \cup \pi_2) & if\ d_1 = d_2 \\ (d_2, \sigma_2, \pi_2) & otherwise \end{cases} \tag{11}$$

$$(d_1, \sigma_1, \pi_1) +_{rhs} (d_2, \sigma_2, \pi_2) = \begin{cases} (d_1 + d_2, \sigma_1 * \sigma_2, \pi_2) & if\ \pi_1 \neq \emptyset\ and\ \pi_2 \neq NIL \\ (d_1 + d_2, \sigma_1 * \sigma_2, \pi_1) & otherwise \end{cases} \tag{12}$$

Intiutively, $+_{rhs}$ concatenates paths represented by 3-tuples and min picks the shortest paths from given paths. If we concatenate no path with any path, the result is no path. If we concatenate a self loop with a valid path (not a self loop or no path), the result is the valid path itself. If we concatenate two valid paths, their weights will be combined and the parent information from the rightmost path will be that of the resulting path(s) (tuple). Since 3-tuple represents number of shortest paths from one vertex to another through σ, the resulting σ will be the multiplication of sigmas of the two paths concatenated.

Lemma 2 (3-tuple Geodetic Semiring) *The set* $S = \overline{\mathbb{R}^+} \times \overline{\mathbb{N}} \times \mathcal{P}(V) \cup \{(\infty, 0, \emptyset), (0, 1, NIL)\}$ *under the operations* $\oplus = \min$ *and* $\odot = +_{rhs}$ *is a semiring with* $\mathbf{0} = (\infty, 0, \emptyset)$ *and* $\mathbf{1} = (0, 1, NIL)$ *such that*

1. min *is commutative and associative*
2. $+_{rhs}$ *is associative*
3. $+_{rhs}$ *is distributive over* min
4. $\mathbf{0}$ *is an absorbing element under* $+_{rhs}$ *such that every* $a \in S$

$$a +_{rhs} \mathbf{0} = \mathbf{0} +_{rhs} a = \mathbf{0}$$

5. *Absorbtion property: for every positive* $a = (d, \sigma, \pi)$ *s.t* $d > 0 \in S$,

$$\min\{\mathbf{1}, a\} = \mathbf{1}$$

Proof: The first part is a similar to the proof of Lemma 5.2 from [17]

1. min operation is clearly associative and commutative.
2. Consider
$$(d_1, \sigma_1, \pi_1) +_{rhs} (d_2, \sigma_2, \pi_2) +_{rhs} (d_3, \sigma_3, \pi_3)$$
and do the operation in any order. The result is
$$(d_1 + d_2 + d_3, \sigma_1 \cdot \sigma_2 \cdot \sigma_3, \pi').$$
Since $+_{rhs}$ defined Equation 12 is a path concatenation operation,
   - if any $\pi_i = \emptyset$, then $\pi' = \emptyset$.
   - if all $\pi_i \neq \emptyset$. Then $\pi' = \pi_i$, where i is the largest value that $\pi_i \neq NIL$.
3. Let $p = (d, \sigma, \pi)$, $p_1 = (d_1, \sigma_1, \pi_1)$ and $p_2 = (d_2, \sigma_2, \pi_2)$. Then we want to show that
$$p \odot (p_1 \oplus p_2) = (p \odot p_1) \oplus (p \odot p_2).$$
WLOG, by the commutativity of min, suppose $\min\{p_1, p_2\} = p_1$ where $d_1 < d_2$
   (a) Suppose $p = (\infty, 0, \emptyset) = \mathbf{0}$. Then
   $$\mathbf{0} \odot (p_1 \oplus p_2) = \mathbf{0} \odot p_1 = \mathbf{0} = (\mathbf{0} \odot p_1) \oplus (\mathbf{0} \odot p_2).$$
   (b) Suppose $p = (0, 1, NIL) = \mathbf{1}$. Then
   $$\mathbf{1} \odot (p_1 \oplus p_2) = \mathbf{1} \odot p_1 = p_1 = p_1 \oplus p_2 = (\mathbf{1} \odot p_1) \oplus (\mathbf{1} \odot p_2).$$
   (c) Suppose $p_2 = (\infty, 0, \emptyset) = \mathbf{0}$. Then
   $$p \odot (p_1 \oplus \mathbf{0}) = p \odot p_1 = (p \odot p_1) \oplus \mathbf{0} = (p \odot p_1) \oplus (p \odot \mathbf{0}).$$
   (d) Suppose $p_1 = (0, 1, NIL) = \mathbf{1}$ and $p, p_2 \in \mathbb{R}_0^+ \times \mathbb{N} \times \mathcal{P}(V)$. Since $\min\{p_1, p_2\} = p_1$, $d_2 > 0$. Hence,
   $$p \odot (\mathbf{1} \oplus p_2) = p \odot \mathbf{1} = (p \odot \mathbf{1}) \oplus (p \odot p_2)$$
   because $d + d_2 > d$.

(e) Suppose $p, p_1, p_2 \in \mathbb{R}_0^+ \times \mathbb{N} \times \mathcal{P}(V)$. Then

$$p \odot (p_1 \oplus p_2) = p \odot p_1 = (p \odot p_1) \oplus (p \odot p_2).$$

Similarly for right distributivity.

Now we need to check the case when $d_1 = d_2$ such that

$$\min\{p_1, p_2\} = (d_1, \sigma_1 + \sigma_2, \pi_1 \cup \pi_2) = p'$$

(i) Suppose $p = (\infty, 0, \emptyset) = \mathbf{0}$. Then

$$\mathbf{0} \odot (p_1 \oplus p_2) = \mathbf{0} \odot p' = \mathbf{0} = (\mathbf{0} \odot p_1) \oplus (\mathbf{0} \odot p_2).$$

(ii) Suppose $p = (0, 1, NIL) = \mathbf{1}$. Then

$$\mathbf{1} \odot (p_1 \oplus p_2) = \mathbf{1} \odot p' = p' = p_1 \oplus p_2 = (\mathbf{1} \odot p_1) \oplus (\mathbf{1} \odot p_2).$$

(iii) Suppose $p_1 = p_2 = (\infty, 0, \emptyset) = \mathbf{0}$. Then

$$p \odot (\mathbf{0} \oplus \mathbf{0}) = p \odot \mathbf{0} = (p \odot \mathbf{0}) \oplus (p \odot \mathbf{0}).$$

(iv) Suppose $p_1 = p_2 = (0, 1, NIL) = \mathbf{1}$. Then

$$p \odot (\mathbf{1} \oplus \mathbf{1}) = p \odot (0, 2, NIL) = (d, 2\sigma, \pi) = p \oplus p = (p \odot \mathbf{1}) \oplus (p \odot \mathbf{1}).$$

(v) Suppose $p, p_1, p_2 \in \mathbb{R}_0^+ \times \mathbb{N} \times \mathcal{P}(V)$. Then

$$\begin{aligned} p \odot (p_1 \oplus p_2) = p \odot p' &= (d + d_1, \sigma \cdot (\sigma_1 + \sigma_2), \pi_1 \cup \pi_2) \\ &= (d + d_1, \sigma \cdot \sigma_1, \pi_1) \oplus (d + d_2, \sigma \cdot \sigma_2, \pi_2) \\ &= [(d, \sigma, \pi) \odot (d_1, \sigma_1, \pi_1)] \oplus [(d, \sigma, \pi) \odot (d_2, \sigma_2, \pi_2)] \\ &= (p \odot p_1) \oplus (p \odot p_2). \end{aligned}$$

Similarly for right distributivity.

□

The semiring $S = \overline{\mathbb{R}^+} \times \overline{\mathbb{N}} \times \mathcal{P}(V) \cup \{(\infty, 0, \emptyset), (0, 1, NIL)\}$ is complete and closed with a closure

$$(d, \sigma, \pi)^* = \begin{cases} (0, \infty, \pi) & \text{if } d = 0, \sigma \neq 0 \\ (0, 1, NIL) & otherwise \end{cases} \tag{13}$$

We can verify this as follows.

$$\begin{aligned} (d, \sigma, \pi)^* &= (0, 1, NIL) \oplus (d, \sigma, \pi) \odot (d, \sigma, \pi)^* \\ &= (0, 1, NIL) \oplus (d + d_*, \sigma \cdot \sigma_*, \pi) \\ &= (0, 1, NIL) \oplus \begin{cases} (0, \infty, \pi) & \text{if } d + d_* = 0, \sigma \neq 0 \\ (d + d_*, \sigma \cdot \sigma_*, \pi) & \text{if } \ d + d_* > 0 \end{cases} \\ &= \begin{cases} (0, \infty, \pi) & \text{if } d = 0, \sigma \neq 0 \\ (0, 1, NIL) & otherwise \end{cases} \end{aligned}$$

We eliminated the case, $(d + d_* = 0, \sigma = 0)$ since the only element with $\sigma = 0$ is the non existing path, $(\infty, 0, \emptyset)$.

Now we present algebraic versions of shortest path algorithms including Dijkstra, Bellman-Ford in the 3-tuple geodetic semiring. Let us denote this semiring as GS3 in short. Let $G = (V, E)$ be a weighted, directed graph with edge weights $w : E \to \overline{\mathbb{R}}$. We define an $n \times n$ adjacency matrix A of a graph as

$$A(u,v) = \begin{cases} (\infty, 0, \emptyset) & \text{if } (u,v) \notin E \\ (w(u,v), 1, \{u\}) & \text{if } u \neq v, (u,v) \in E. \end{cases} \tag{14}$$

So that, elements of the adjacency matrix are from 3-tuple geodetic semiring. If the graph have self-loops, the self-loops are eliminated in the process of constructing the adjacency matrix.

Algorithm 1 Algebraic Dijkstra

Input: A ($n \times n$ adjacency matrix), s (source node)
Output: d (Shortest path distance)
1: $Q \leftarrow$ zero vector, $Q[s] = \infty$
2: $d = A[s, :]$
3: **while** $Q \neq \infty$ **do**
4: $\quad u = \text{argmin}\{d_d + Q\}$, $Q[u] = \infty$
5: $\quad d' = d[u]. + A[u, :]$
6: $\quad d = d. \min d'$
7: **end while**
8: $d[s] = (0, 1, NIL)$
9: **return** d

Algorithm 2 Algebraic Bellman-Ford

Input: A ($n \times n$ adjacency matrix), s (source node)
Output: d (Shortest path distance)
1: $d = A[s, :]$
2: $d_k = A[s, :]$
3: **for** $k = 1$ to $n - 1$ **do**
4: $\quad d_k = d_k \min . + A$
5: $\quad d = d. \min d_k$
6: **end for**
7: $d[s] = (0, 1, NIL)$
8: **return** d
9: // blank line for spacing

In Algorithm 1, the inputs are an $n \times n$ adjacency matrix A in 3-tuple geodetic semiring as defined in 14 and a source node s. Q is a n-dimensional vector used to indicate which vertices have been traversed so far. For example, if a vertex v is traversed, then $Q[v] = \infty$. Otherwise, $Q[v] = 0$. In Line 1, Q is initialized with 0 and $Q[s] = \infty$ which indicates that the source node is already traversed. In Line 2, d is the shortest path distance vector and we initialize it with s^{th} row of A since $A[s, :]$ contains shortest path distances from s to all other vertices after the level 1 traversal. The while loop between Line 3 - 7 iterates until all the vertices are traversed. In Line 4, d_d means the first (distance) component of 3-tuple shortest path distance vector and u is the next node closest to s in the shortest path tree where + operation is the usual plus operation. In Line 4, we also indicate that u is traversed. In Line 5, we get the new distance vector d' when we explore the resulting paths to the next level through the vertex u. In Line 6, we take element-wise min between d' and d and assign its result as the shortest path distance vector. + and min operations in Line 5 and 6 are 3-tuple geodetic semiring operations. At the end of the while loop, d contains the shortest path distance vector in the 3-tuple geodetic semiring, but $d[s]$ will be incorrect. So we need to assign $d[s] = (0, 1, NIL)$ in Line 8.

In Algortihm 2, the inputs are the same as the inputs to the Algebraic Dijkstra's algorithm 1. But as compared to Algebraic Dijkstra's algorithm 1, Algebraic Bellman-Ford algorithm 2 is much concise and straight-forward. In Line 1, the shortest path distance vector, d is initialized as s^{th} row of A since $A[s, :]$ contains shortest path distances from s to all other vertices after the level 1 traversal. In Line 2, we declared another temporary vector, d_k which is used to keep track of the paths with k-hops in the k^{th} level. In the for loop between Line 3 - 6, Line 4 further explores

k-hop paths and Line 5 takes the element-wise minimum between the shortest path distance vector d and k-hop path distance vector d_k. As a result, after $n-1$ iterations d contains the shortest path distances where $n-1$ iteration is enough because any shortest path in a graph has maximum $n-1$ length (hops).

3 Algebraic Algorithms for Betweenness Centrality

Algorithm 3 Brandes's Algorithm in unweighted graphs [8]

1: $C_B[v] = 0, v \in V$
2: **for** $s \in V$ **do**
3: $S \leftarrow$ empty stack
4: $P[w] \leftarrow$ empty list, $w \in V$
5: $\sigma[t] = 0, t \in V; \quad \sigma[s] = 1$
6: $d[t] = -1, t \in V; \quad d[s] = 0$
7: $Q \leftarrow$ empty queue
8: enqueue $s \rightarrow Q$
9: **while** Q *not empty* **do**
10: dequeue $u \leftarrow Q$
11: push $u \rightarrow S$
12: **for** each neighbor w of u **do**
13: **if** $d[w] < 0$ **then**
14: enqueue $w \rightarrow Q$
15: $d[w] = d[u] + 1$
16: **end if**
17: // shortest path to w via u
18: **if** $d[w] = d[u] + 1$ **then**
19: $\sigma[w] = \sigma[w] + \sigma[u]$
20: append $u \rightarrow P[w]$
21: **end if**
22: **end for**
23: **end while**
24: $\delta[v] = 0, v \in V$
25: // S returns vertices in order of non-increasing distance from s
26: **while** S *not empty* **do**
27: *pop* $w \leftarrow S$
28: **for** $v \in P[w]$ **do**
29: $\delta[v] = \delta[v] + \frac{\sigma[v]}{\sigma[w]}(1 + \delta[w])$
30: **end for**
31: **if** $w \neq s$ **then**
32: $C_B[w] = C_B[w] + \delta[w]$
33: **end if**
34: **end while**
35: **end for**

In Section 1, we briefly discussed different methods to compute betweenness centrality and there are mainly two ways. One is the traditional method to compute betweenness centrality based on APSP and the another is Brandes' algorithm based on SSSP. Since computing betweenness centrality based on APSP is very straight forward, the discussion in Section 1 is sufficient. Brandes' algorithm is more interesting because it enables computational efficiency. We will include Brandes' algorithm below to refer to when explaining its algebraic version.

Let $G = (V, E)$ be a weighted, directed graph with edge weights $w : E \rightarrow \overline{\mathbb{R}}$ and let A be the adjacency matrix of G in 3-tuple geodetic semiring defined the same as Equation 14. We want to write algebraic betweenness centrality algorithms based on some of the shortest path algorithms in 3-tuple geodetic semiring presented in Section 2. We are going to present two algebraic betweenness centrality algorithms named Algebraic BC - Dijkstra-Brandes (ABC-DB) and Algebraic BC - Bellman-Ford (ABC-BF). To compute betweenness centrality using Brandes algorithm, we need to choose a SSSP algorithm. Even though other semirings such as the tropical semiring for shortest paths can be used with Bellman-Ford algorithm to compute the shortest path distances, they don't count the number of shortest paths that we need in order to update BC scores. But shortest path algorithms (Dijkstra's 1 or Bellman-Ford 2) in 3-tuple geodetic

semiring can count shortest path distances, number of shortest paths and penultimate vertices. Since Dijksra's algorithm can give the order of traversal, we used Dijkstra's algorithm in Brandes's algorithm with 3-tuple geodetic semiring as follows.

3.1 Algebraic BC - Dijkstra-Brandes (ABC-DB)

Algorithm 4 ABC-DB GS3

Input: A ($n \times n$ adjacency matrix)
Output: C_B (Betweenness Centrality)

1: $C_B[v] = 0, v \in V$
2: **for** $s \in V$ **do**
3: $Q \leftarrow$ zero vector
4: $S \leftarrow$ zero vector
5: $Q[s] = \infty$
6: $d = A[s, :]$
7: $i = 1$
8: **while** $Q \neq \infty$ **do**
9: $u = \text{argmin}\{d_d + Q\}$
10: $S[i] = u$
11: $i = i + 1$
12: $Q[u] = \infty$
13: $d' = d[u]. + A[u, :]$
14: $d = d. \min d'$
15: **end while**
16: $d[s] = (0, 1, NIL)$
17: $\delta[v] = 0, v \in V$
18: **for** $i = n - 1$ to 1 **do**
19: $w = S[i]$
20: $\alpha = \frac{1}{d_\sigma[w]}(1 + \delta[w])$
21: $\delta = \delta + \alpha d_\pi[w]. * d_\sigma$
22: $C_B[w] = C_B[w] + \delta[w]$
23: **end for**
24: **end for**
25: // blank line for spacing

Algorithm 5 ABC-DB GS2

Input: A ($n \times n$ adjacency matrix)
Output: C_B (Betweenness Centrality)

1: $C_B[v] = 0, v \in V$
2: **for** $s \in V$ **do**
3: $Q \leftarrow$ zero vector
4: $S \leftarrow$ zero vector
5: $Q[s] = \infty$
6: $d = A[s, :]$
7: $i = 1$
8: **while** $Q \neq \infty$ **do**
9: $u = \text{argmin}\{d_d + Q\}$
10: $S[i] = u$
11: $i = i + 1$
12: $Q[u] = \infty$
13: $d' = d[u]. + A[u, :]$
14: $d = d. \min d'$
15: **end while**
16: $d[s] = (0, 1)$
17: $\delta[v] = 0, v \in V$
18: $\pi = d\,\text{argmin}\,. + A$
19: **for** $i = n - 1$ to 1 **do**
20: $w = S[i]$
21: $\alpha = \frac{1}{d_\sigma[w]}(1 + \delta[w])$
22: $\delta = \delta + \alpha\pi[w]. * d_\sigma$
23: $C_B[w] = C_B[w] + \delta[w]$
24: **end for**
25: **end for**

In Algorithm 4, Algebraic BC - Dijkstra-Brandes with GS3, in Line 1, we first initialize betweenness centrality for all vertices with 0. Then in the for loop between Line 2 and 23, we iterate over each vertices. Firstly, we find the shortest path tree rooted at the corresponding vertex s using Dijkstra's algorithm 1 where S is an array with length $n - 1$ for saving vertices in the order of traversal and i is a counter variable to indicate the order of traversal. Secondly, we updated betweenness centrality score using the shortest path tree rooted at s. Line 17, initializes the dependency of $s \in V$ on every other vertex as 0. Dependency of a vertex is defined in Definition 1.5. In the for loop at Line 18, we iterate S in reverse direction, from back to front. As a result, we will update betweenness centrality scores starting from the bottom of the shortest path tree rooted at s. In Line 20 and 21, the dependency of s on the parents of a vertex w on shortest paths from

s is accumulated using Theorem 1. Line 21 updates betweenness centrality of w by adding the dependency of s on w to betweenness centrality of w. These two parts are repeated for each all different source vertex s and the dependency of s for any $v \in V$ is accumulated for each iteration of loop between Line 2 and 24 and corresponding updates for betweenness centrality are done. At the end of algorithm, all betweenness centrality scores are correctly calculated. It is clear to compare this algorithm with Brandes' algorithm 3. So we exclude the first part for computing SSSP in both algorithms and explain the second part for accumulating betweenness centrality scores. The while loop between Line 26-34 in Brandes' algorithm and the for loop between Line 18-23 in ABC-DB are both for traversing the vertices in non-increasing order of their distance from s [8]. Line 19 of this algorithm corresponds to Line 27 in Brandes' algorithm. Line 20-21 corresponds to Line 28-30 in Brandes algorithm. In Line 20, α is a constant number for the dependency of $s \in V$ on any $v \in V$ in Theorem 1 and $d_\sigma[w]$ is the σ component of the shortest path distance vector from s to w (w^{th} entry of shortest path distance vector d). Line 21 accumulates dependency of s where $d_\pi[w]$ is a set of penultimate vertices, but we vectorized it into a n-dimensional vector whose entry is 1 for the penultimate vertices(parents) of a vertex w on shortest paths from s to w and 0 for all other vertices. d_σ is a n-dimensional vector containing the σ component of d. Then $d_\pi[w]$ and d_σ are multiplied element-wise. So that, for each parents of w on shortest paths, dependency of s is accumulated without for loop in Brandes' algorithm. Then the result is multiplied by α constant and added to δ. With these operations, dependency of s is accumulated as described in Theorem 1. Lastly, Line 22 corresponds with Line 31-33 of Brandes' algorithm. w in Line 19 is never going to be the same as s. Hence we don't need to check the condition in Line 31 in Brandes' algorithm.

Alternatively, we can use the usual geodetic semiring with 2-tuples [6] for writing Algebraic BC - Dijkstra-Brandes algorithm. Let us denote it as ABC-DB GS2. We can find the penultimate vertices set, π, the third component of 3-tuple geodetic semiring based on shortest path distances resulted by Dijkstra's algorithm using the usual geodetic semiring. Finding π is the same as computing the shortest path tree since shortest paths are usually represented by parent information in the shortest path tree. Algebraic BC algorithms works with graphs without self-loops or 0-weight cycles. In this case, computing the shortest path tree is simple. Let d be the shortest path distance vector from source node s and $u, v \in V$ such that $u \neq v$. By Bellman criterion, Lemma 1, if $d(u) + w(u, v) = d(v)$, then $u \in \pi$, that is, u is the penultimate vertex on a shortest path from s to v. Similarly, we can find all penultimate vertices for v. For all $u \in V$, $\pi(v) = \operatorname{argmin}_{u \neq v}\{d(u) + w(u, v)\}$. This can be expressed algebraicly as

$$\pi = d \operatorname{argmin} . + A \tag{15}$$

which means, add d elementwise to every column of A and take argmin for each column [17]. ABC-DB GS2 is given in Algorithm 5.

3.2 Algebraic BC - Bellman-Ford (ABC-BF)

We want to write algebraic betweenness centrality algorithm based on algebraic Bellman-Ford shortest path algorithm (Algorithm 2) in geodetic semiring. Then we need to back propagate centrality scores. But it is not possible to know the order of node traversal when using algebraic Bellman-Ford. Hence, we need to back propagate betweenness centrality scores differently than the method used in Brandes's algorithm. Methods such as keeping track of the levels of shortest path tree, similar to the betweenness centrality matrix formulation in [17] don't work on weighted graphs. We used the method from [23] that is based on partial centrality factors from [22]. Basically we are reproducing their work, but our algorithm details differ from theirs in terms of

algorithm details. For example, we used the usual geodetic semiring for Bellman-Ford instead of a monoid, employed mask to distinguish current frontiers. Partial centrality factors is defined as follows.

$$p_{s\bullet}(v) = \frac{\delta_{s\bullet}(v)}{\sigma_{sv}} = \sum_{w:v\in P_s(w)} \frac{1}{\sigma_{sw}} + p_{s\bullet}(w) \tag{16}$$

Then, we can rewrite betweenness centrality in terms of parial centrality factors as below.

$$B_C(v) = \sum_{s\neq v} p_{s\bullet}(v) \cdot \sigma_{sv} \tag{17}$$

Equation 16 is a restructure of the recursive relation of dependency of the source node, s. When back propagating betweenness centrality, we need to make sure that only finalized partial centrality factors can be back propagated and repeat the process till partial centrality factors are finalized for all vertices. Once partial centrality factors of the source node, s is computed, we accumulate betweenness centrality as in Equation 17. This process doesn't use the parent information from 3-tuple geodetic semiring. So we write the algorithm using usual geodetic semiring with 2-tuples [6]. Let (d, p, c) be a 3-tuple representing shortest path distance, partial centrality factor and number of children on the shortest path tree. [23] defined addition and multiplication operations on such 3-tuples and used it for back propagating centrality scores.

$$(d_1, p_1, c_1) \oplus (d_2, p_2, c_2) = \begin{cases} (d_1, p_1, c_1) & if\ d_1 > d_2 \\ (d_2, p_2, c_2) & if\ d_2 > d_1 \\ (d_2, p_1 + p_2, c_1 + c_2) & otherwise \end{cases} \tag{18}$$

$$(d_1, p_1, c_1) \otimes (d_2, p_2, c_2) = (d_2 - d_1, p_2, c_2) \tag{19}$$

where $\otimes$ is not associative. Hence, it is not a semiring.

4 Algebraic Algorithm for Percolation Centrality

[20] proposed a new centrality measure called *percolation centrality* which is more useful in network percolation scenarios such as disease transmission in a network of cities, spreading of infection or a virus over a social network of individuals. Percolation centrality is defined based on shortest paths counts, similar to betweenness centrality, but it accounts for the changing percolation states of vertices. Hence, our ABC-DB algorithm 4 can be easily extended to compute percolation centrality. Time slices of a static network with a varying node property, called percolation state is used in percolation centrality and as defined in [20], x_i^t is the precolation state of a node i at time t where $x_i^t \in [0, 1]$ and its value indicates different percolation states. If $x_i^t = 0$, it is a non-percolation state at time t. If $x_i^t = 1$, it is a fully percolated state at time t. If $0 < x_i^t < 1$, it is a partially percolated state. Then based on percolation status of nodes, a concept called a *percolated path* is introduced. A path starting from a percolated node and ending at a percolated or non-percolated, or partially percolated node is called a percolated path. The percolation centrality of a node v at a given time t is defined as the proportion of percolated paths passing through that vertex and it is formally defined as

$$C_P^t(v) = \frac{1}{n-2} \sum_{s\neq v\neq r} \frac{\sigma_{sr}(v)}{\sigma_{sr}} \frac{x_s^t}{[\sum x_i^t] - x_v^t} \tag{20}$$

Algorithm 6 Algebraic BC - Bellman-Ford (ABC-BF)

Input: A ($n \times n$ adjacency matrix)
Output: C_B (Betweenness Centrality)
$C_B[v] = 0, v \in V$
for $s \in V$ **do**
  $d = A[s,:]$, $d_k = A[s,:]$
  for $k = 1$ to $n-1$ **do**
    $d_k = d_k \min. + A$
    $d = d. \min d_k$
  end for
  $d[s] = (0, 1, NIL)$
  $A[i,j] = (w(i,j), 0, 0) \quad e_{ij} \in E$
  $A[i,j] = (\infty, 0, 0) \quad e_{ij} \notin E$
  $A[i,i] = (0, 0, 0)$
  $Z'[v] = (d[v], 0, -1) \quad v \in V$
  $Z[v] = (d[v], 0, 1) \quad v \in V$
  $Z = Z \oplus (A \cdot Z')$
  flag = false, mask[v] $= 1 \; \forall v \in V$
  $\forall v \in V$:
  if $Z[v].c = 0$ **then**
    $Z'[v] = (d_d[v], \frac{1}{d_\sigma[v]}, -1)$
    $mask[v] = 0$, flag = true
  else
    $Z'[v] = (-\infty, 0, 0)$
  end if
  while flag is true **do**
    $Z' = A \cdot Z'$
    $\forall v \in V$
    if $Z[v].c = 0$ **then**
      $Z[v].c = -1$
    else
      if $Z[v].c < 0$ **then**
        $Z[v].c = -Z[v].c$
      end if
    end if
    Z[mask] $= Z$[mask] $\oplus \; Z'$[mask]
    flag = false
    $\forall v \in V$:
    if $Z[v].c = 0$ **then**
      $Z'[v] = (d_d[v], Z[v].p + \frac{1}{d_\sigma[v]}, -1)$
      $mask[v] = 0$
      flag = true
    else
      $Z'[v] = (-\infty, 0, 0)$
    end if
  end while
  $C_B = C_B + Z_p \; .* \; d_\sigma$
end for

where σ_{sr} is the number of shortest paths between $s, r \in V$, $\sigma_{sr}(v)$ is the number of shortest paths between $s, r \in V$ that go through v and the fraction

$$c_{sv}^t = \frac{x_s^t}{[\sum x_i^t] - x_v^t} \tag{21}$$

is called the relative contribution of a percolated path starting from s to the percolation centrality (it is denoted as w_{sv}^t in [20]). The sum $[\sum x_i^t]$ is the total percolation in the network and the percolation state of v at time t, is subtracted from the sum for normalization purpose.

Now we want to modify dependency of a vertex, Definition 1.5 and define percolation dependency of a vertex.

Definition 4.1 (percolation dependency) *The percolation dependency of a vertex $s \in V$ on a single vertex $v \in V$ at time t is*

$$\delta_{s\bullet}^t(v) = \sum_{r \in V} \frac{\sigma_{sr}(v)}{\sigma_{sr}} \frac{x_s^t}{[\sum x_i^t] - x_v^t} = c_{sv}^t \sum_{r \in V} \frac{\sigma_{sr}(v)}{\sigma_{sr}} = c_{sv}^t \cdot \delta_{s\bullet}(v) \tag{22}$$

because $c_{sv}^t = \dfrac{x_s^t}{[\sum x_i^t] - x_v^t}$ is independent of r. i.e the portion of percolated shortest paths starting

from s and going through v such that

$$C_P^t(v) = \frac{1}{n-2}\sum_{s\neq v}\delta_{s\bullet}^t(v) = \frac{1}{n-2}\sum_{s\neq v} c_{sv}^t \cdot \delta_{s\bullet}(v) \tag{23}$$

Then by Theorem 1

$$\delta_{s\bullet}^t(v) = c_{sv}^t \sum_{w:v\in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1+\delta_{s\bullet}(w)) \tag{24}$$

where $P_s(w)$ is the set of parents of a vertex w on shortest paths from s to w. Hence, the percolation dependency of a vertex also follows a recursive relation and we can compute percolation centrality at a given time using Brandes' algorithm.

Algebraic Percolation Centrality (APC) algorithm is presented below and the blue lines are the extension parts that differs from or added to ABC-DB, Algorithm 4.

Algorithm 7 Algebraic Percolation Cenrality (APC)

Input: A ($n \times n$ adjacency matrix); $x^1, x^2, \ldots, x^\lambda$ (Percolation states of nodes at each time step)
Output: C_P (Percolation Centrality, an $n \times \lambda$ matrix)

1: $C_P = 0$
2: **for** $s \in V$ **do**
3: **for** $t = 1$ to λ **do**
4: $c_s^t = (x^t[s] * \mathbf{1}). \div (sum(x^t) * \mathbf{1} - x^t)$
5: $Q \leftarrow$ zero vector
6: $S \leftarrow$ empty vector
7: $Q[s] = \infty$
8: $d = A[s, :]$
9: $i = 1$
10: **while** $Q \neq \infty$ **do**
11: $u = \text{argmin}\{d_d + Q\}$
12: $S[i] = u$
13: $i = i + 1$
14: $Q[u] = \infty$
15: $d' = d[u]. + A[u, :]$
16: $d = d. \min d'$
17: **end while**
18: $d[s] = (0, 1, NIL)$
19: $\delta[v] = 0, v \in V$
20: **for** $i = n - 1$ to 1 **do**
21: $w = S[i]$
22: $\alpha = \frac{1}{d_\sigma[w]}(1 + \delta[w])$
23: $\delta = \delta + \alpha d_\pi[w]. * d_\sigma$
24: $C_P^t[w] = C_P^t[w] + c_s^t[w] * \delta[w]$
25: **end for**
26: **end for**
27: **end for**
28: $C_P = \frac{1}{n-2} C_P$

In Algorithm 7, we compute percolation centrality of each node at each time slice. Since the network is static but percolation states vary over time, a single adjacency matrix A and percolation states if each nodes at different time steps (can be inputed as a $n \times \lambda$ matrix) are given where λ is the number of time slices. In Line 1, C_P is a $n \times \lambda$ matrix for storing percolation states and its j^{th} column corresponds to percolation centralities of all n nodes at time j^{th} time. For each source vertex s and for each time step, we compute the shortest path tree rooted at s at time t and we start from leaf nodes of the tree and back track vertices in the reverse order of traversal. During this process, we accumulate percolation dependency of s (modified with the relative contribution) on all relevant nodes and update the corresponding percolation centrality of the current node at the corresponding time. Most part of the algorithm is the same as ABC-DB 4, we focus on explaining the extension parts in blue. Line 4 computes the relative contribution of a percolated path starting at s as described in Equation 21 where x^t is the percolation states of

nodes at time t and c_s^t is a n-dimensional vector whose v^{th} entry is c_{sv}^t, the relative contribution of a percolated path starting at s and going through v, to the percolation centrality of v at time t. Line 24 accumulates percolation centrality according to Equation 23.

5 Implementation

The algebraic algorithms introduced in this paper all have time complexity bounded by that of matrix products. In many cases there exist algorithms that perform better on serial computers that are not expressed algebraically. These algorithms often exploit the sparsity of the graph adjacency matrix and avoid the cubic complexity of direct operations expressed in terms of products of the dense adjacency matrix. On the other hand, these serial algorithms often are difficult to parallelize due to data structures or algorithmic choices that induce operational dependencies that reduce opportunities for concurrent execution. The use of priority queues are a common example of this. Algebraic graph algorithms are based on matrix-matrix or matrix-vector multiply and can leverage known methods for implementing these algebraic primitives in parallel (and potentially using sparse data structures where appropriate). In this paper we implemented the algorithms that we define to study their correctness, performance, and scalability on small-scale shared memory parallel systems.

The starting point of the implementation is the fundamental algebraic structure, the 3-tuple semiring. Early experiments in implementation used the relatively new Julia language [7] because it allows us to define a user-defined type for the semiring and overload the usual $+, *$ operations to use our custom semiring operations $\oplus, \otimes$ respectively. This has the appealing property that the implementation of the algorithms will be concise and will closely resemble the pseudo-code used to define the algorithm in the paper. Unfortunately, we encountered a number of issues in achieving high performance in Julia that often required the addition of explict types, manually specialized functions, and other change that ultimately lowered the abstraction level of the code. We found that implementing the algorithms in Python with Numpy for arrays and Numba for just-in-time compilation to parallel code was preferable.

One example of an efficiency issue that we encountered was mapping the 3-tuple geodetic semiring to a specific data structure. In particular the third component, π of 3-tuple geodetic semiring is a set structure which requires set union operation. We need to have highly optimized, set-like operation to get good performance. The overhead of a tree-based set data structure proved to be a major performance bottleneck. We implemented Algebraic BC - Dijkstra-Brandes (ABC-DB GS3), Algorithm 4 using dense arrays. For example, adjacency matrix of a graph in the 3-tuple geodetic semiring is split into three separate dense arrays where A_d is the usual adjacency matrix of edge weights, A_σ is a boolean matrix representing the number of shortest paths between edges, A_π is also a boolean matrix such that nonzero entries of i-th row vector represents the penultimate vertices. This representation trades space for efficiency in performing the set operations for the π component.

For a shortest path distance vertor d in 3-tuple geodetic semiring, it is represented by 2 separate vectors and a boolean matrix where d is the shortest distance vector from s to other vertices, σ is a vector of the number of shortest paths from s to other vertices and π is a boolean matrix in which nonzero entries of i-th row vector represents the penultimate vertices of the shortest paths from s to the vertex i. We defined functions corresponding to operations in 3-tuple geodetic semiring such as elementwise min ($\oplus = \min$) and elementwise add ($\odot = +_{rhs}$) required in ABC-DB.

Similarly, we also implemented ABC-DB GS2, Algorithm 5 using dense arrays. It differs from ABC-DB GS3 by working with 2-tuple geodetic semiring and finding penultimate vertices after

finding shortest path distances in geodetic semiring.

6 Experiment

We choose Algebraic BC - Dijkstra-Brandes algorithm for our experiment since it is relatively simpler than Algebraic BC - Bellman-Ford that we don't need to do extra computation for finding the frontiers when back propagating centraliy scores. Particularly, we choose ABC-DB GS2, Algorithm 5 for our experiment. This eliminates the need to implement set operations required by GS3, thus simplifying the storage and time requirements of the algorithm. Our experiments showed that the GS3-based algorithms performed very poorly compared to those using GS2, and profiling results pointed at the set data structures and corresponding operations as a cause.

Table 1: ABC-DB performance in seconds.

ABC-DB \ n	100	200	300	400	500
GS 3	0.094	0.867	3.551	9.986	24.713
GS 2	0.042	0.271	0.819	1.843	3.461

We implemented ABC-DB in Python and optimized it with Numba, an open source JIT compiler that translates a subset of Python and NumPy code into fast code via the LLVM compiler [18]. Numba can produce optimized serial code, multicore parallel code, as well as kernels for execution on GPU accelerators. We run serially ABC-DB with Numba's nopython option and we also parallelized it with Numba's parallel option with 20 cores. We used the Washington State University Kamiak HPC cluster provided by Center for Institutional Research Computing (CIRC) [3]. We experimented on a single node which has 256 GB memory and two 10-core Intel Xeon CPUs (Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz) running the CentOS Linux 7 operating system. We compare the performace of ABC-DB GS2 with NetworkX [16], a commonly used python package for network algorithms. We used Erdös-Rényi graph randomly generated by NetworkX with 0.5 probablity of edge creation. Both serial and parallel ABC-DB GS2 is faster than NetworkX for Erdös-Rényi graphs refer to Figure 1. However, when we experiment with GRENOBLE set of sparse unsymmetric graphs from SuiteSparse Matrix Collection [11], serial ABC-DB GS2 is slower than NetworkX and NetworkX performs almost as good as parallel ABC-DB GS2 shown in Figure 2. ABC-DB GS2 scales well in Figure 3 and its scability is stable as graph size increases in Figure 4. The performance of NetworkX versus our Numba accelerated algorithms shows what we expect:

- For very sparse graphs, the Python code with better asymptotic complexity outperforms the linear algebraic algorithms even when they are compiled. This is due to the use of dense matrices in the NumPy/Numba implementation.
- For less sparse graphs, the optimized linear algebra based methods scale better as the graph size grows. This likely is due to the overhead of the interpreted Python code in NetworkX and less wasted work in the dense matrix methods used by our code.
- The benefits of parallel execution of the algebraic methods is a function of the graph size. For small graphs the benefit of parallelism is exhausted with small core counts, but as the graphs grow we see speedups that benefit from ever larger core counts. As showin in Figure 4, for

small graphs the scaling stops at 4 cores, but continues up to and beyond the maximum core count of our testbed for larger graph sizes.

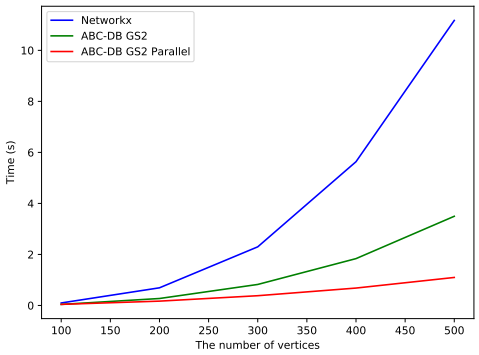

Figure 1: ABC-DB GS2 vs NetworkX for Erdös-Rényi graphs.

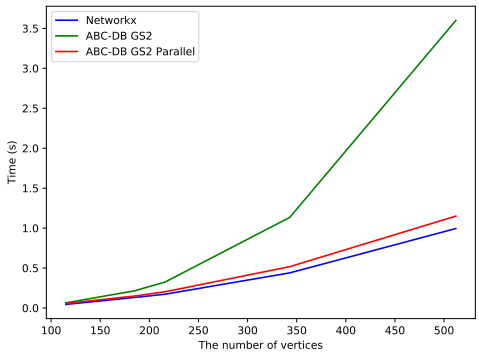

Figure 2: ABC-DB GS2 vs NetworkX for GRENOBLE set of sparse graphs.

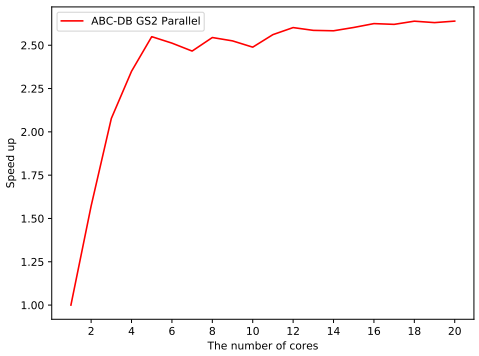

Figure 3: Speedup of parallelized ABC-DB GS2 for a Erdös-Rényi graph, 500 vertices.

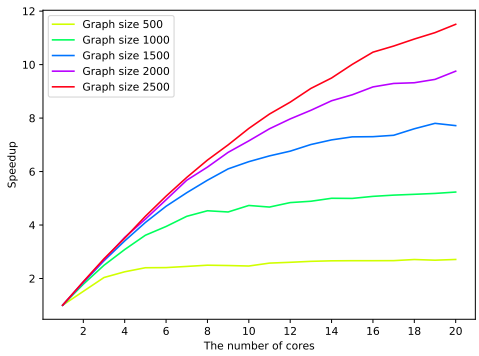

Figure 4: Speedup of parallelized ABC-DB GS2 for Erdös-Rényi graphs.

These basic experiments show that our methods implementing algebraic BC show the same benefits as other graph algorithms expressed via linear algebra: they map easily onto parallel architectures using known parallelization methods, and thus can exploit parallel architectures for processing large graphs that are beyond the capability of single processor systems. We expect that implementing them using existing parallel sparse linear algebra libraries will allow our BC algorithms to scale further for large sparse graphs.

7 Conclusion

We presented a 3-tuple geodetic semiring (GS3) and used it to write algebraic betweenness algorithms (ABC). Among different variants, we find ABC based on Dijkstra's and Brandes' algorithm (ABC-DB) is better choice than the other options such as ABC based on Bellman-Ford and back tracking frontiers. We revealed the problem of implementing ABC based on GS3 due to the set

nature of parent information in GS3. Hence, we proposed an alternative version of ABC-DB algorithm using GS2. We implemented it using dense arrays to use numba's nopython option and auto parallelization. Our experiment result shows that ABC-GS2 is suitable for medium size denser graphs and it scales well as we increase the number of cores. The future work is to implement ABC based on sparse matrices. Furthermore, designing a specific data structure to efficiently handle complex structures such as GS3 is an open problem. Since we can use GS2 and compute parent information at the end in ABC-DB, GS3 may not be the first choice for implementing ABC, but there could be different graph algorithms which utilizes GS3 better using parent information carried by GS3 in every iterations or one could propose a new graph algorithm based on GS3.

References

[1] http://graphblas.org.

[2] http://graph500.org.

[3] https://hpc.wsu.edu.

[4] A. V. Aho and J. E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1974.

[5] D. A. Bader and K. Madduri. Parallel Algorithms for Evaluating Centrality Indices in Real-world Networks. In *2006 International Conference on Parallel Processing (ICPP'06)*, pages 539–550, Aug 2006.

[6] V. Batagelj. Semirings for social networks analysis. *The Journal of Mathematical Sociology*, 19(1):53–68, 1994. doi:10.1080/0022250X.1994.9990135.

[7] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: A Fast Dynamic Language for Technical Computing. *CoRR*, abs/1209.5145, 2012. arXiv:1209.5145.

[8] U. Brandes. A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology*, 25(2):163–177, 2001. doi:10.1080/0022250X.2001.9990249.

[9] A. Buluç and J. R. Gilbert. The Combinatorial BLAS: Design, Implementation, and Applications. *Int. J. High Perform. Comput. Appl.*, 25(4):496–509, Nov. 2011. doi:10.1177/1094342011403516.

[10] M. Cerinšek and V. Batagelj. *Semirings and Matrix Analysis of Networks*, pages 1681–1687. Springer New York, New York, NY, 2014. doi:10.1007/978-1-4614-6170-8_152.

[11] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. 38(1), Dec. 2011. doi:10.1145/2049662.2049663.

[12] N. Edmonds, T. Hoefler, and A. Lumsdaine. A space-efficient parallel algorithm for computing betweenness centrality in distributed memory. In *2010 International Conference on High Performance Computing*, pages 1–10, Dec 2010. doi:10.1109/HIPC.2010.5713180.

[13] J. G. Fletcher. A More General Algorithm for Computing Closed Semiring Costs between Vertices of a Directed Graph. *Commun. ACM*, 23(6):350–351, June 1980. doi:10.1145/358876.358884.

[14] M. Gondran and M. Minoux. *Graphs, Dioids and Semirings: New Models and Algorithms*, volume 41 of *Operations Research/Computer Science Interfaces*. Springer-Verlag, New York, NY, 1. aufl. edition, 2008. doi:10.1007/978-0-387-75450-5.

[15] O. Green and D. A. Bader. Faster Betweenness Centrality Based on Data Structure Experimentation. *Procedia Computer Science*, 18:399 – 408, 2013. 2013 International Conference on Computational Science. doi:10.1016/j.procs.2013.05.203.

[16] A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring Network Structure, Dynamics, and Function using NetworkX. In G. Varoquaux, T. Vaught, and J. Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[17] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, 2011. doi:10.1137/1.9780898719918.

[18] S. K. Lam, A. Pitrou, and S. Seibert. Numba: A LLVM-Based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2833157.2833162.

[19] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. Chavarria-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–8, 2009. doi:10.1109/IPDPS.2009.5161100.

[20] M. Piraveenan, M. Prokopenko, and L. Hossain. Percolation Centrality: Quantifying Graph-Theoretic Impact of Nodes during Percolation in Networks. *PLOS ONE*, 8(1):1–14, 01 2013. doi:10.1371/journal.pone.0053095.

[21] E. Robinson and J. Kepner. Array based betweenness centrality. In *SIAM Conference on Parallel Processing for Scientific Computing*, 2008.

[22] A. E. Sariyüce, K. Kaya, E. Saule, and U. V. Çatalyürek. Betweenness Centrality on GPUs and Heterogeneous Architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, page 76–85, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2458523.2458531.

[23] E. Solomonik, M. Besta, F. Vella, and T. Hoefler. Scaling Betweenness Centrality Using Communication-Efficient Sparse Matrix Multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3126908.3126971.

[24] E. Solomonik and T. Hoefler. Sparse Tensor Algebra as a Parallel Programming Model. *CoRR*, abs/1512.00066, 2015. arXiv:1512.00066.

[25] G. Tan, V. Sreedhar, and G. Gao. Analysis and performance results of computing betweenness centrality on IBM Cyclops64. *The Journal of Supercomputing*, 56:1–24, 04 2011. doi:10.1007/s11227-009-0339-9.

[26] G. Tan, D. Tu, and N. Sun. A Parallel Algorithm for Computing Betweenness Centrality. In *2009 International Conference on Parallel Processing*, pages 340–347, 2009. doi:10.1109/ICPP.2009.53.