B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 247

Efficient Batched Distance and Centrality Computation in Unweighted and Weighted Graphs

Manuel Then[1], Stephan Günnemann[2], Alfons Kemper[3], Thomas Neumann[4]

Abstract: Distance and centrality computations are important building blocks for modern graph databases as well as for dedicated graph analytics systems. Two commonly used centrality metrics are the compute-intense closeness and betweenness centralities, which require numerous expensive shortest distance calculations. We propose batched algorithm execution to run multiple distance and centrality computations at the same time and let them share common graph and data accesses. Batched execution amortizes the high cost of random memory accesses and presents new vectorization potential on modern CPUs and compute accelerators. We show how batched algorithm execution can be leveraged to significantly improve the performance of distance, closeness, and betweenness centrality calculations on unweighted and weighted graphs. Our evaluation demonstrates that batched execution can improve the runtime of these common metrics by over an order of magnitude.

Keywords: Graph Databases, Graph Analytics, Closeness Centrality, Betweenness Centrality

1 Introduction

Recently, there has been growing interest in graph analytics as a means of analyzing large social networks, web graphs, road networks and gene interaction graphs. This lead to the creation of graph databases and dedicated graph analytics systems like Pregel [Ma10] or PGX [Ho15]. A common graph analytics use case is ranking vertices by importance to find the most important vertices. This can, for example, be used to find the most influential users in a social network, and, thus, to improve the effectiveness of targeted advertising campaigns. Algorithms that determine the importance of vertices are called *centrality algorithms*.

In practice, different centrality algorithms are used. They cover specific use cases and cannot be used interchangeably [NSJ11]. Simple degree-based centrality algorithms like degree centrality [Fr78] directly use the vertices' number of incident edges as a measure for their importance, and can be used as a first indication of relevant vertices. The PageRank [BP98] algorithm extends this idea by iteratively propagating vertices' influence through the graph. Both degree centrality and PageRank are well-researched and can be efficiently computed for large graphs. Other popular centrality algorithms are based on the notion of paths. The *closeness centrality* [Fr78] metric ranks vertices by their average geodesic distance to all other vertices, i.e., the length of the shortest paths to these other vertices. A vertex is

[1] TU Munich, Department of Informatics, Boltzmannstraße 3, 85748 Garching, then@in.tum.de
[2] TU Munich, Department of Informatics & Institute for Advanced Study, Boltzmannstraße 3, 85748 Garching, guennemann@in.tum.de
[3] TU Munich, Department of Informatics, Boltzmannstraße 3, 85748 Garching, kemper@in.tum.de
[4] TU Munich, Department of Informatics, Boltzmannstraße 3, 85748 Garching, neumann@in.tum.de

considered more central when it can reach other vertices in fewer steps; closeness centrality, hence, measures how fast information can propagate from a vertex through the network. In contrast, *betweenness centrality* [Fr78] counts the number of shortest paths between any two vertices that a vertex v is on. Thus, it is a measure for how much communication goes through v, and, as a result, for how much v can influence communication [NSJ11]. Closeness and betweenness centrality are computationally very expensive metrics with their complexities of $O(|V|^2 + |V| * |E|)$ and $O(|V| * |E|)$, respectively, on unweighted graphs, and $O(|V|^2 * |E|)$ and $O(|V| * |E| + |V|^2 * \log |V|)$, respectively, on weighted graphs [Br01]. The algorithms' high complexity on weighted graphs is caused by the required all pairs geodesic distance computations. This makes computing exact closeness and betweenness centralities prohibitively expensive on large-scale graphs that are used in practice. Hence, these metrics are often only approximated by means of sampling [EW01, Ba07].

To significantly reduce the runtimes of exact closeness and betweenness centrality computation, and to improve the accuracy of approximate centrality computation within a given time frame, we propose batched algorithm execution. In *batched algorithm execution* multiple executions of the same algorithm from different source vertices are run concurrently. They are synchronized to share common graph element accesses. Thus, batched algorithm execution amortizes the costs of random data accesses that are inherent to graph analytics. While this allows to greatly reduce the overall runtime of graph analytics algorithms, batched execution introduces new tradeoffs and makes necessary novel data structures.

In this paper we propose batched algorithms that efficiently calculate distances and centrality metrics in unweighted and weighted graphs. To that end, we revisit the existing multi-source breadth-first search algorithm MS-BFS [Th14] for batched closeness centrality in unweighted graphs and improve its efficiency by means of a constant-time batch counter. Moreover, we propose a batched betweenness centrality algorithm for unweighted graphs that extends MS-BFS to allow reverse traversal and low-overhead vertex predecessor detection. We then further the principles of batched multi-source execution and show how they can be applied to geodesic distance, closeness centrality and betweenness centrality computation in weighted graphs.

Our contributions are as follows.

- We introduce batched algorithm execution and explain how an algorithm can be run concurrently from multiple source vertices and share common data accesses.
- We present batched algorithms for closeness centrality and betweenness centrality for unweighted graphs.
- We propose batched geodesic distance, closeness centrality and betweenness centrality algorithms for weighted graphs.
- We evaluate our algorithms using multiple synthetic and real-world datasets.

The paper is structured as follows. After this introduction, in Section 2 we give a short overview of the terminology used in this paper. In Section 3 we present the concept of batched algorithm execution, which we apply to distance and centrality algorithms on unweighted

and weighted graphs in Sections 4 and 5, respectively. We evaluate our algorithms in Section 6. Section 7 elaborates on related work and Section 8 gives our conclusions.

2 Background

In this section we introduce the terminology and algorithms used throughout this paper. Furthermore, we explain the MS-BFS algorithm which we use as the basis of batched algorithm execution.

We define a graph G as the tuple (V, E) of vertices V and edges $E \subseteq V \times V$. Further, vertices and edges may have an arbitrary number of named properties attached to them.

2.1 Geodesic Distance

The *geodesic distance* between two vertices u and v is the length of the shortest path from u to v. In this paper we only consider the single-source shortest path case. For unweighted graphs, geodesic distance is measured in the number of traversed edges between u and v, which can efficiently be derived using a breadth-first search (BFS) from u. In weighted graphs, the distance from u to v is the sum of edge weights along the path between them. Depending on the type of graph and its topology, various algorithms exist to compute the geodesic distances. In this paper, we evaluate batched variants of Dijkstra's algorithm with a Fibonacci Heap [FT87] and the Bellman-Ford algorithm [Be58].

2.2 Closeness Centrality

Closeness centrality ranks the centralities of graph vertices by their average geodesic distance to all other vertices in a graph [Fr78]. Given two functions $reachable(v)$ that determines the set of vertices reachable from v, and $distance(v, u)$ which determines the geodesic distance from v to u, the normalized closeness centrality of a vertex v is defined as:

$$CC_v = \frac{|reachable(v)|^2}{(|V| - 1) * (\sum_{u \in reachable}(v) : distance(v, u))}$$

2.3 Betweenness Centrality

Betweenness centrality considers a vertex as central when it is on many shortest paths [Fr78]. In an undirected graph, the normalized betweenness centrality of v is defined as:

$$BC_v = \sum_{u,w \in V,\ u \neq v \neq w} : \frac{|\{\mathcal{P} \mid \mathcal{P} \in shortest_paths(u, w) \land v \in \mathcal{P}\}|}{|shortest_paths(u, w)| * (|reachable(v)|) * (|reachable(v)| - 1)}$$

Here, *shortest_paths*(u, w) denotes the set of all shortest paths $\mathcal{P}$ from u to w. The result is normalized by dividing the share of paths that v is on by the number of vertex pairs in the connected component that do not include v. Note that for undirected graphs the normalization factor has to be adapted to not include duplicate pairs. Naive implementations of betweenness centrality have a runtime in $O(|V|^3)$, even for unweighted graphs. In this paper, we use Brandes's algorithm to compute vertices' betweenness centrality, which reduces the runtime to $O(|V| * |E|)$ for unweighted graphs, and to $O(|V| * |E| + |V|^2 * \log |V|)$ for weighted graphs.

2.4 Multi-Source BFS

In [Th14] the batched multi-source breadth-first search MS-BFS was proposed as an efficient way of solving the geodesic distance problem and to compute vertices' closeness centrality in unweighted graphs. We use MS-BFS as the base of our efficient batched closeness and betweenness centrality algorithms on unweighted graphs, and extend the algorithm's ideas to general batched centrality computation on weighted graphs. In the following, we give an overview of MS-BFS.

MS-BFS runs BFSs from multiple source vertices at the same time and merges traversals that would happen redundantly in independent BFS runs. It is especially beneficial in small-world networks—graphs that have a small diameter and vertex degrees that follow a power-law distribution, e.g., social networks. In small-world networks highly-connected hub vertices are often discovered by multiple concurrent BFSs in the same BFS step, i.e., in the same distance from their respective source. MS-BFS leverages that these BFSs will be very similar for the remainder of their traversals.

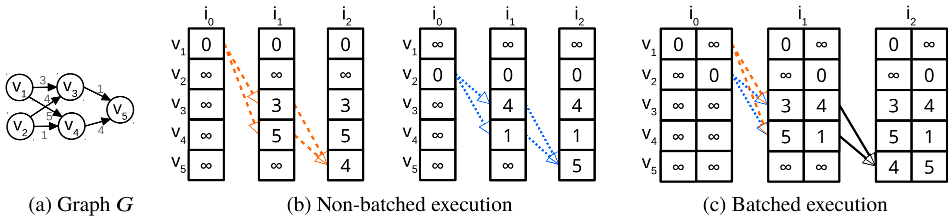
MS-BFS represents the vertices' traversal statuses in multiple concurrent BFSs as bitsets. On these bitsets, the BFS steps are processed concurrently for multiple traversals by means of bit operations, which also implicitly merge traversals. For example, when visiting a vertex v, MS-BFS determines all concurrent traversals that discover its neighbor n in the next BFS step by intersecting the bitset of traversals for which v is visited with the negation of the bitset of traversals that have already visited n. Thus, using SIMD instruction, MS-BFS can concurrently process BFS steps in hundreds of traversals.

3 Batched Algorithm Execution

MS-BFS shares data accesses and computations in multiple BFSs. While this allows for great speedups, MS-BFS is too restrictive. Redundant computations and data accesses do not only occur in multi-source BFSs, but also in many other analytical graph algorithms when they process a graph multiple times. Thus, we propose batched algorithm execution which generalizes the ideas of MS-BFS.

In *batched algorithm execution* an algorithm is redesigned so that it can be efficiently executed concurrently from multiple source vertices. The *concurrent executions* are synchronized to

share common computations and data accesses. Batched algorithms, thus, can amortize the cost of random data accesses over all concurrent executions, leading to greatly reduced overall runtimes. Furthermore, batched algorithm execution facilitates vectorization because the vertices' and edges' associated properties can be processed at the same time for multiple executions.



(a) Graph G (b) Non-batched execution (c) Batched execution

Fig. 1: Data access pattern of the Bellman-Ford algorithm using different executions.

Figure 1 shows how non-batched and batched execution run the Bellman-Ford algorithm. The Bellman-Ford algorithm is executed from the vertices v_1 and v_2 in the small weighted graph G, shown in Figure 1a, to determine their geodesic distances to all vertices. Figure 1b shows how the algorithm is run in traditional non-batched execution by depicting the algorithm's distances array in each iteration. The algorithm is first run from v_1, as shown on the left side: it is initialized in iteration i_0, in i_1 the distances to v_3 and v_4 are discovered, and in iteration i_2 the distance to v_5 is found. Afterward, the algorithm is run again, this time from v_2, as shown on the right side of Figure 1b. The two algorithm executions from v_1 and v_2 process the full graph separately. Consequently, all random accesses, e.g., to determine the current distances of a vertex's neighbors, as well as the resulting memory stalls happen separately for both algorithm executions.

In contrast, a batched variant of the Bellman-Ford algorithm, which we also propose in this paper, can compute the distances from multiple sources at the same time. Figure 1c depicts how our batched algorithm accesses the data when concurrently computing the distances from v_1 and v_2. The first algorithm iteration is still similar to the non-batched case, as no graph and data accesses can be shared among the execution from v_1 and v_2. The batched execution does, however, already show improved data locality. In the second iteration the two algorithm executions share their graph as well as their data accesses. Instead of separately reading each vertex's current distance and comparing it with its neighbor's distances, the two executions from v_1 and v_2 do so at the same time. This allows the batched algorithm to not only amortize its memory access latencies, but also to reduce its memory bandwidth requirements.

In this paper we refer to the *set of concurrent executions* as S, and to the actual concurrent executions as $s_1, \dots, s_\omega \in S$. We assume that there is a bijection between the concurrent executions and their source vertices; by convention we use the same subscript to associate them. In this section's Bellman-Ford example, there are $\omega = 2$ executions $S = \{s_1, s_2\}$ from v_1 and v_2, respectively.

In batched algorithm execution it is important that accesses from multiple concurrent executions to the same graph property have spatial locality, i.e., data accesses in close temporal succession should operate on memory locations that are close together so that the CPU's caches and prefetchers can be leveraged. To achieve spatial locality we introduce a *batched execution-optimized memory layout* in which all executions' values for variables as well as for vertex and edge properties are co-located, as shown in Figure 1c. We refer to variables with co-located values as *batch variables* and denote a batch variable B of type $\mathcal{T}$ as *BatchVar*<$\mathcal{T}$> B. It is stored as a $\mathcal{T}$ array with as many elements as there are concurrent executions of the batched algorithm. Consider that the batched Bellman-Ford algorithm shown in Figure 1c stores the concurrent executions' distances for each vertex in a property *dists* of type *BatchVar*<*float*>. When the algorithm is executed concurrently from 16 sources, each vertex v's *dists* variable is a 16-element array in which the ith element contains the distance of v in execution i. Assuming that *float*s are 4 byte-wide, v's *dists* field fills a 64-byte cache line—a common size in many modern CPUs. Hence, all concurrent executions' *dists* variable values are in the cache at the same time and no cache space is wasted, e.g., for not required vertices' property values.

4 Distances and Centralities in Unweighted Graphs

In this section we show how batched algorithm execution can be applied to efficiently compute distances and centralities in unweighted graphs. Section 4.1 shortly discusses batched BFS-based distance computation, Section 4.2 describes a batched closeness centrality algorithm with highly efficient constant-time vertex counting, and Section 4.3 presents a novel batched betweenness centrality algorithm.

4.1 Distance

In unweighted graphs the geodesic distance from a source vertex to all other vertices in the same connected component can efficiently be determined using breadth-first search (BFS). In the context of centrality computations, we are interested in computing distances from multiple sources. Thus, we can apply the MS-BFS algorithm and directly write the distance of every discovered vertex into a vertex property that uses our batch-optimized memory layout. Modern CPUs and compute accelerators provide scatter instructions which allow us to write multiple distance assignments in a single instruction. As batched geodesic distance computation in unweighted graphs is already discussed in [Th14] we omit the full algorithm listing.

4.2 Closeness Centrality

A vertex's closeness centrality (CC) is its average geodesic distance to all other vertices. The CC values of a set of vertices S can, thus, be computed by averaging the results from the previous section's batched distance algorithm for every source $s \in S$. There is, however,

no need to actually store the distances of the discovered vertices. Instead, it is sufficient to run a BFS from every source s and directly sum the distances to all discovered vertices per concurrent execution. Listing 1 shows our batched CC algorithm that computes the metric for a set of *vertices* in the unweighted graph G and stores its results in the vertex property *cc*.

```
1   INPUT:   Graph G, Array<Vertex> vertices
2   OUTPUT:  VertexProperty<double> cc
3
4   BatchVar<int> iterationVertices=0, totalVertices=0, distanceSums=0
5
6   G.MS-BFS( sources: vertices,
7             onDiscovered: (vertex, discoveredExecutions) => {
8                 FOR EACH i in discoveredExecutions:
9                     iterationVertices[i] ++
10            },
11            onIterationEnd: (iteration) => {
12                distanceSums      += iterationVertices * iteration
13                totalVertices     += iterationVertices
14                iterationVertices  = 0
15            } )
16
17  FOR i = 1 .. vertices.length:                       ◁ Normalize CC values
18      cc[vertices[i]] = (totalVertices[i] * totalVertices[i])
19                        / (distanceSums[i] * (G.num_vertices -1))
```

List. 1: Batched closeness centrality algorithm for unweighted graphs

The algorithm leverages the inherent property of BFS traversal that all vertices discovered in the same iteration have the same distance to its source. So, rather than summing the actual distances directly, we count the vertices found in each iteration *iterationVertices*, and at the iteration's end add the number of vertices multiplied by the current distance to the actual sum of distances *distanceSums*. We elaborate on counting the number of discovered vertices in the subsequent section. In the algorithms last Lines 17 through 19 we compute the final, normalized CC values, as described in Section 2.2.

4.2.1 Efficient Batch Incrementer

Naively, incrementing *iterationVertices* for every concurrent execution that found a new vertex involves a loop over all executions, as is shown in Lines 8 and 9 of Listing 1. To determine if an execution found a new vertex, a branch whether or not *discoveredExecutions* is *true* for the respective execution is required. This branch is hard to predict and, thus, not efficient on modern CPUs with deep pipelines. In the following, we propose an *efficient batch incrementer* that allows incrementing *iterationVertices* in a branch-free manner, independent of the number of executions in which vertices were actually discovered.

The basic idea of our batch incrementer is to use a single-byte cache to sum up each execution's *true* entries in the *discoveredExecutions* fields over the course of multiple *onDiscovered* calls. Consider that there are 64 concurrent executions. We store all 64 executions' cache values as a contiguous chunk of memory which we can easily increment by *discoveredExecutions* for all iterations at the same time using bit operations. To do so,

we mask out all but each byte's least significant bit in the *discoveredExecutions* bitset and add these eight bytes (the masked bitset) to the first eight cache bytes. Afterward, we shift *discoveredExecutions* left by one bit, repeat the masking and add this masked bitset to the second eight bytes in the cache. This continues until the bitset was shifted seven times, and all cache bytes were touched. At this point, for each i with $discoveredExecutions[i] = true$, the cache byte $\lfloor i/8 \rfloor + (i \mod 8)$ was incremented.

As cache bytes may overflow after 255 increment operations, we flush it to the actual batch variable *discoveredExecutions* after 254 increments.The cache flush increments the batch variable's values by each execution's cached number. After the flush completes, the ith element of *discoveredExecutions* contains the number of times *iterationVertices* was *true* in execution i. All loops in our batch incrementer have a fixed iteration count independent of the actual number of increments; they gain further speedup by unrolling and vectorization. Using our efficient batch incrementer greatly improves the performance of batched closeness centrality computations in unweighted graphs.

4.3 Betweenness Centrality

The betweenness centrality (BC) value of a vertex is a measure for how many shortest paths it is on. As explained in Section 2.3, the state-of-the-art algorithm to compute BC is Brandes's algorithm [Br01]. In the following, we show an efficient batched variant of Brandes's BC algorithm that leverages a novel MS-BFS variant which allows reverse multi-source BFS traversal and can efficiently determine BFS predecessor relationships. In contrast to previous work [Br01, Ma09], it does not need to explicitly store vertex distance and predecessor information, which greatly improves its data locality.

4.3.1 Reverse MS-BFS

The MS-BFS algorithm is designed to discover vertices in increasing distance from the sources. To compute vertices' BC using Brandes's algorithm it is, however, necessary to first process all vertices in ascending distance to calculate their σ values, and then in descending distance order to calculate their δ values. Single-source BFS algorithms do this latter *reverse BFS traversal* by collecting all discovered vertices in a stack, which is traversed once the forward traversal is completed. This approach could be applied to MS-BFS as well by introducing a stack of vertex identifiers and *BatchVar<bool>*s. Instead of building a new datastructure during the MS-BFS traversal, it is more efficient to store and leverage the existing per-iteration bitsets of vertices that must be visited—the iterations' *frontiers*.

Storing all iterations' frontiers for the reverse traversal allows not only to reconstruct the BFS traversal order, but also to efficiently determine BFS predecessor relationships as we show in the next section. For small-world networks—the focus of this paper—the space overhead of storing all frontiers is negligible, as these graphs have a low diameter, which means that only few frontiers must be stored. Furthermore, because there are concurrent

BFS executions, the majority of the stored frontier entries are non-empty and would require a similar amount of memory, or even more, when the less versatile stack would be used.

4.3.2 Implicit Vertex Predecessors

Brandes's BC algorithm uses each vertex's predecessors to accumulate dependencies between vertices. The *predecessors* of a vertex v in a BFS traversal from a source s are all vertices u such that there is an edge (u, v) that is on the shortest path from s to v. Brandes's algorithm and its parallelized variant [Ma09] build an explicit list of predecessors for each vertex. One problem of explicitly stored predecessor lists is that they either require runtime memory allocations or significant over-allocation as their size cannot be determined in advance and is only bounded by the number of edges in the graph. This is especially problematic in the multi-source case, as vertices' predecessors are likely to differ between the concurrent executions. Thus, we propose reconstructing vertices' predecessors from the frontiers of previous MS-BFS iterations.

Lemma. *The bitset of executions in which a vertex p is a predecessor of v can be derived from the frontiers of the current iteration iter and the previous iteration iter*-1 *by means of a bitwise* and *operation:*

$$predecessorIn(p, v) = \begin{cases} frontiers[iter-1][p] \;\&\; frontiers[iter][v], & if\ (p, v) \in E \\ \varnothing, & otherwise \end{cases}$$

Proof. Assume there is an edge between p and v. If p was visited in the BFS iteration directly before v, then p must be in the last iteration's frontier and v in that of the current one, so p is v's predecessor. If p was visited, but v was not visited in the subsequent iteration, the bit operation results in p not being v's predecessor. Similarly, if v is marked in the frontier, but p not in the previous one, p is not v's predecessor. In case there is no edge between p and v, the former cannot be the latter vertex's predecessor, so the set of executions in which p is the predecessor is empty. □

4.3.3 Batched Betweenness Centrality

Building on our proposed reverse MS-BFS and the implicitly-defined vertex predecessors, Listing 2 shows our batched betweenness centrality algorithm for unweighted graphs. It closely follows the structure of Brandes's algorithm, but runs the algorithm from multiple vertices at the same time and batches its execution. Whenever a vertex v is visited in the same distance by multiple concurrent executions, all executions process v at the same time. This is especially beneficial for the complex numeric computations in Lines 17 and 18, as our batch-optimized data layout improves the algorithm's spatial locality and facilitates the use of wide vectorized instructions, allowing multiple executions' computation in the same instruction. Furthermore, batched execution allows to pre-aggregate the *delta* values in

deltaSum before they are added to the global *bc* property; this avoids congestion as multiple parallel threads may access this value.

```
1  INPUT:    Graph G
2  OUTPUT:   VertexProperty<double> bc
3
4  VertexProperty<BatchVar<double>> sigma = 0, delta = 0
5  FOR i = 1 .. G.num_vertices:
6      sigma[G.vertices[i]][i] = 1
7
8  G.MS-BFS( sources: G.vertices,
9            onDiscovered: (v, discoveredIn) => {
10               FOR EACH n in G.neighbors(v):
11                   FOR EACH i in (predecessorIn(v,n) & discoveredIn):
12                       sigma[n][i] += sigma[v][i]
13           },
14           onReverse: (v, discoveredIn) => {
15               FOR EACH n in G.neighbors(v):
16                   FOR EACH i in (predecessorIn(v,n) & discoveredIn):
17                       delta[v][i] += (sigma[v][i] / sigma[n][i])
18                                      * (delta[n][i] + 1)
19
20               double deltaSum = 0
21               FOR i = 1 .. G.vertices.num_vertices:
22                   IF v != G.vertices[i]:
23                       deltaSum += delta[v][i]
24               bc[v] += deltaSum
25           } )
26
27 FOR EACH v in G.vertices:                              ◁ Normalize BC values
28     bc[v] /= (G.num_vertices - 1) * (G.num_vertices - 2)
```

List. 2: Batched betweenness centrality algorithm for unweighted graphs

Note that in contrast to the previously shown closeness centrality algorithm that can selectively be run for a subset of the vertices to determine their centrality, BC must always process all vertices in the graph—or more specifically, in a given connected component—to compute the metric. All presented techniques can also be applied to approximate BC algorithms [Ba07].

For a lack of space we do not give details about the parallelization of our algorithm. Our approach is, however, similar to the parallelization presented in [Ma09].

5 Distances and Centralities in Weighted Graphs

In this section we focus on batched distance and centrality computation on weighted graphs. We first discuss batched weighted multi-source geodesic distance computation in Section 5.1, as it is an important building block for centrality computation. Afterward, in Sections 5.2 and 5.3, we show how the calculated distances can be used to efficiently derive the closeness and betweenness centrality metrics, respectively.

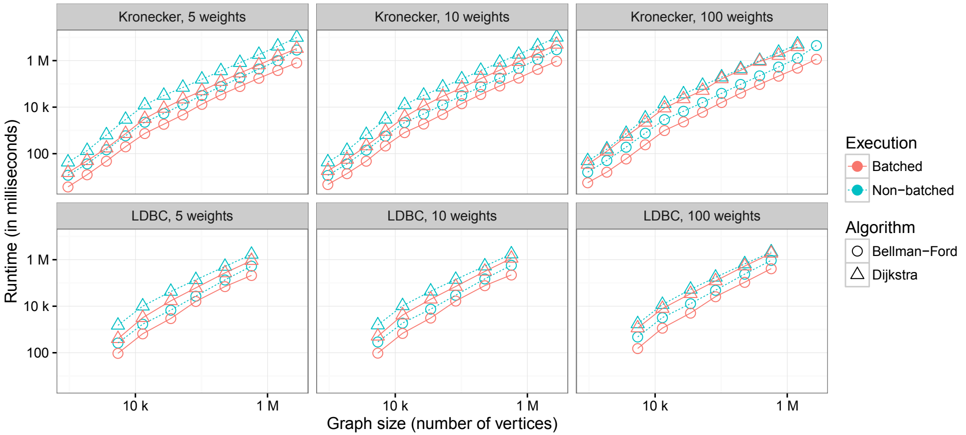


Fig. 2: Comparison of multi-source shortest geodesic distance runtimes for non-batched and batched variants of Dijkstra's algorithm and the Bellman-Ford algorithm, applied to Kronecker and LDBC graphs of various sizes and with different weight counts.

5.1 Distance

In practice, two algorithms are commonly used to compute the distances from one or more vertices to all other vertices in a weighted graph: Dijkstra's algorithm with a Fibonacci heap [FT87], and the Bellman-Ford algorithm [Be58]. While the worst-case runtime of the Bellman-Ford algorithm is $O(|V| * |E|)$, previous work [Ye70] shows that it has an expected runtime of $O(E)$ in large dense graphs with low diameter. We, thus, analyzed the runtimes of the algorithms using LDBC and Kronecker small-world networks, which we also used in our evaluation. We generated edge weights in the range $[1, weightcount]$ for three weight counts: 5, 10 and 100; the former two are commonly used in datasets with user-generated ratings, and the latter indicates the algorithms' scalability with higher weight counts. Figure 2 shows our results for normal and batched variants (solid and dashed lines, respectively) of both Dijkstra's algorithm and the Bellman-Ford algorithm (triangles and circles, respectively). While both algorithms show similar scaling behavior with increasing graph size, the Bellman-Ford algorithm's absolute performance is 3-10× higher. In all measurements, the batched algorithm variants show better runtimes than the respective non-batched algorithm. Virtually unaffected by the used weight count, the Bellman-Ford algorithm's batched variant is 3-4× faster than the non-batched algorithm. In contrast, the benefit of our batched Dijkstra's algorithm becomes smaller for higher weight counts.

We only discuss the details of our batched multi-source Bellman-Ford algorithm as it—and even its original single-source variant—clearly outperforms Dijkstra's algorithm for the evaluated small-world networks with even hundreds of different weights. Listing 3 depicts our proposed algorithm. It expects a weighted graph G, and a set of *sources* from which the distances to all other vertices should be calculated. The results are stored in a batch vertex property *dists*, that contains the vertices' distance from each of the sources.

Listing 3 has the same structure as the original Bellman-Ford algorithm. It initializes the sources' distances with 0 and all other vertices' distances with infinity, and iteratively checks whether new shorter distances can be found based on the already known distances. Unlike the original algorithm, our batched multi-source Bellman-Ford algorithm discovers new shortest distances for *all* sources' executions at the same time. As a result, the executions can share the random data accesses to the neighbors' known distances, amortizing the memory access costs over all concurrent executions. Furthermore, our batched algorithm can leverage the wide vector instructions of modern compute accelerators like the Intel Xeon Phi, e.g., for the distance computation in Line 20.

We use an optimization proposed by Yen [Ye70] and only check vertices' neighbors if their distance was modified. For this we introduce a batch vertex property *modified* that keeps track of modified vertices and the executions in which their distance was updated. For simplicity, the depicted algorithm uses one *modified* entry per concurrent execution. Our actual implementation uses one entry per *i* executions, where *i* is the number of distance values that can be stored and processed simultaneously in the target CPU's largest SIMD register. Using one *modified* entry for multiple executions avoids branches in the algorithm's hot loop and has only negligible overhead for distance computations that are done unnecessarily as all operations are vectorized. When no distance was *changed* in an iteration of the algorithm, no shorter geodesic distances may be found and the algorithm finishes.

```
1   INPUT:   WeightedGraph G, Array<Vertex> sources
2   OUTPUT:  VertexProperty<BatchVar<double> > dists
3
4   VertexProperty<BatchVar<bool> > modified = false
5   dists = Infinite
6
7   FOR i = 1 .. sources.length:
8       Node v = sources[i]
9       dists[v][i]     = 0
10      modified[v][i] = true
11
12  bool changed = true
13  WHILE changed:
14      changed = false
15      FOR EACH v in G.vertices:
16          IF not modified[v].empty():
17              FOR EACH n in G.neighbors(v):
18                  double weight = edgeWeight(v,n)
19                  FOR EACH i in modified[v]:
20                      double newDist = min(dists[n][i], dists[v][i] + weight)
21                      IF newDist != dists[n][i]:
22                          dists[n][i]     = newDist
23                          modified[n][i] = true
24                          changed        = true
```

List. 3: Batched Bellman-Ford-based geodesic distance algorithm for weighted graphs

5.2 Closeness Centrality

Based on the batched multi-source geodesic distance computation described in the previous section, we propose the efficient batched closeness centrality (CC) algorithm shown in Listing 4. For a given weighted graph G, it computes the exact CC values of a set of *vertices* and stores them as the vertex property *cc*.

Our batched CC algorithm first computes the distances from the *vertices* of interest to all other vertices in their respective connected components using the batched distance computation presented in the previous section. It then counts the reachable vertices *totalVertices* and sums their distances *distanceSums* concurrently for all executions. Because of the data layout of the batched variables *totalVertices* and *distanceSums*, this computation can be automatically vectorized by modern compilers. At the end of the algorithm, the final CC values are normalized, as described in Section 4.2.

```
1   INPUT:  WeightedGraph G, Array<Vertex> vertices
2   OUTPUT: VertexProperty<double> cc
3
4   VertexProperty<BatchVar<double> > dists
5   dists = ms_geodesic_distances(G, vertices)          ◁ Listing 3
6   BatchVar<double> distanceSums = 0
7   BatchVar<int> totalVertices  = 0
8
9   FOR EACH v in G.vertices:
10      FOR i = 1 .. vertices.length:
11          IF dists[v][i] != Infinite:                ◁ Vertex is reachable
12              distanceSums[i]  += dists[v][i]
13              totalVertices[i] ++
14
15  normalize(G, vertices, cc)                          ◁ Normalize CC, see Listing 1
```

List. 4: Batched closeness centrality algorithm for weighted graphs

5.3 Betweenness Centrality

In the following, we present a novel batched betweenness centrality (BC) algorithm for weighted graphs that is again based on Brandes's algorithm.

5.3.1 Batched Distance Ordering

A batched variant of Brandes's BC algorithm must find a global order of traversing a weighted graph such each execution traverses the graph with ascending distance from its respective source. An efficient global order further ensures that the executions share as many computations and data accesses as possible. Our batched BC algorithm builds such a global order using a hash-based approach that determines for each vertex-distance pair (v, d) the set of executions $T \subseteq S$ which discover v in distance d. It then sorts the resulting

(d, v, T)-tuples by their distance d and merges tuples when this is possible without violating the ordering.

Our approach does not yet consider possible tuple reorderings that do violate the strict distance order without impacting the algorithm's result. Assume there are three tuples o_1, o_2, o_3 with $o_i = (d_i, v_i, T_i)$ and $d_1 < d_2 < d_3$. When $v_1 = v_3$ and $T_1 \cap T_2 \cap T_3 = \emptyset$, o_1 and o_3 can safely be merged to allow improved execution sharing. There is, however, a tradeoff between preprocessing time to determine an optimal order and the actual algorithm execution time. Exploring this tradeoff and finding heuristics for efficient reordering are interesting directions for future work.

5.3.2 Batched Betweenness Centrality

We propose the batched algorithm shown in Listing 5 to efficiently compute the BC values for all vertices in a weighted graph G and store them in a vertex property *bc*. It computes the geodesic distances for all vertices using the batched Bellman-Ford-based algorithm proposed in Section 5.1 and builds a global traversal order in which Brandes's algorithm is executed for multiple executions concurrently, sharing common data accesses and computations.

```
1  INPUT:   WeightedGraph G
2  OUTPUT:  VertexProperty<double> bc
3
4  VertexProperty<BatchVar<double> > dists
5  dists = ms_geodesic_distances(G, G.vertices)       ◁ Listing 3
6  List<Tuple<double, Node, ExecutionSet» > traversalOrder
7  traversalOrder = findOrdering(G, dists)            ◁ Section 5.3.1
8  initialize(sigma, delta)                           ◁ See Lines 4-6 in Listing 2
9
10 FOR EACH (d,v,T) in traversalOrder:
11     FOR EACH n in G.neighbors(v):
12         FOR EACH s in T:                           ◁ Executions with v in distance d
13             IF dists[n][s] == d + edgeWeight(v,n):
14                 sigma[v][s] += sigma[n][s]         ◁ Vertex is a predecessor
15
16 FOR EACH (d,v,T) in traversalOrder.reverse():
17     FOR EACH n in G.reverseNeighbors(v):
18         FOR EACH s in T:
19             IF dists[n][s] + edgeWeight(n,v) == d:
20                 delta[n][s] += sigma[n][s]*(1+delta[v][s])/sigma[v][s]
21         addBC(bc[v], delta[v], s)                  ◁ See Lines 20-24 in Listing 2
22
23 normalize(G, bc)                                   ◁ See Lines 27-28 in Listing 2
```

List. 5: Batched betweenness centrality algorithm for weighted graphs

While the structure of the algorithm is then similar to our batched unweighted BC algorithm, it differs in two important ways: One, instead of using the BFS-defined traversal order, Listing 5 uses the optimized batch distance ordering *traversalOrder* for both the forward and reverse traversal. Two, as explained for the batched unweighted BC, it is unfeasible to explicitly store the predecessors of all vertices in all concurrent executions. Our weighted

BC algorithm does, however, not reconstruct each vertex v's predecessors from the traversal history like the former algorithm does, but determines them directly by checking the neighbors' distances to v: when the neighbor n's distance plus the incident edge's weight equals v's distance, then n must be on the shortest path to v, and, hence, its predecessor.

Note that for simplicity, the depicted algorithm computes the *sigma* and *delta* values for all vertices in the graph at the same time. In an implementation that is suited to scale to large real-world graphs, the algorithm's executions for Lines 4 through 21 must be run using smaller batches of executions, such that only a subset of the source vertices is considered at a time [Th14].

6 Evaluation

In this section we evaluate the runtime and scalability of the algorithms we propose in this paper. We first give a short description of our experiment setup. Next, we evaluate how the batch size—the number of concurrently processed executions—influences the efficiency of batched algorithm execution. Afterward, we discuss how our centrality algorithms scale with increasing graph size; we omit the distance algorithms' scalability, as it was already discussed in Section 5.1.

6.1 Experiment Setup

To evaluate the efficiency of the batched algorithms proposed in this paper, we implemented them as standalone C++14 programs and compiled them using GCC 5.2.1. We derived our MS-BFS implementation from the original authors' provided sources [Th14]. As competitors we used non-batched variants of the algorithms and optimized them according to the state-of-the-art. For unweighted and weighted betweenness centrality we additionally ported Brandes's implementation, kindly provided by the author [Br01], to the graph structures used in all other implementations.

We evaluated all algorithms using synthetic and real-world datasets of various sizes. We obtained the Citeseer (384k vertices), DBLP (1.3M vertices), Hudong (3M vertices) and Wikipedia (1.9M vertices) real-world graphs datasets from the KONECT repository [Ku13]. The synthetic LDBC and Kronecker graphs we used comprised up to 4.2M vertices and 300M edges. Linked Database Counsil (LDBC) graphs are designed to resemble real-world social networks [Io16]. We generated them using the LDBC SNB generator version 0.2.6[5]. Kronecker graphs are used in the common graph benchmark Graph500 and were built using the benchmark's data generator[6] with edge factor set to 32.

All experiments were executed on a dual-socket system equipped with Intel Xeon E5-2660 v2 CPUs (20 logical threads at 2.2GHz) and 256GB of main memory. The used operating system was Ubuntu Linux 15.10 with kernel 4.2. To reduce the experiments' runtimes, we ran all algorithms for 15,360 deterministically random selected vertices in the graphs.

[5] https://github.com/ldbc/ldbc_snb_datagen
[6] https://github.com/graph500/graph500

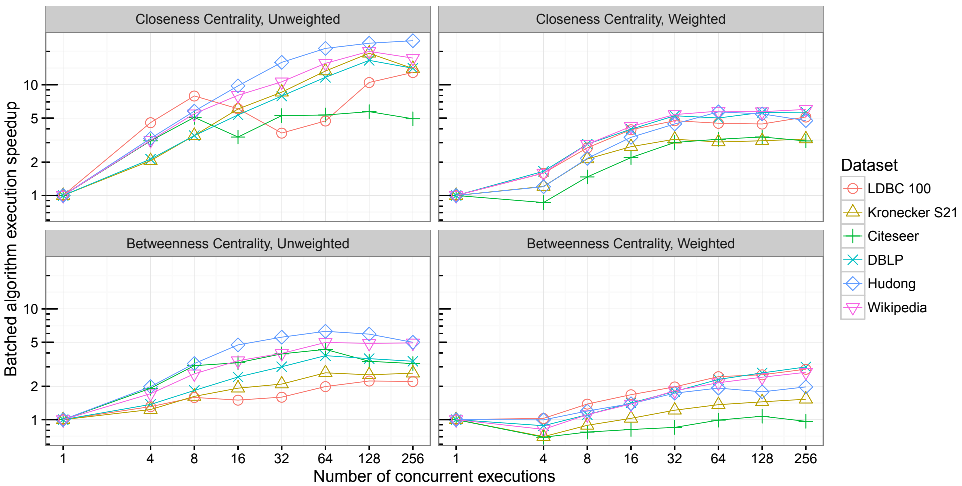


Fig. 3: Speedup through batched algorithm execution with increasing number of concurrent executions.

6.2 Scalability with Number of Concurrent Executions

This paper proposes batched algorithm execution for geodesic distance and centrality computation. In the following we show how batching influences the runtime of the presented algorithms. For each algorithm we measured the non-batched runtime as well as the absolute runtimes for increasing numbers of concurrent executions. For weighted graphs we assume a weight count of five. Figure 3 shows the speedups gained through batched algorithm execution over non-batched execution.

It can be seen that batching leads to significant speedups, even for few concurrent executions. The more concurrent executions are used, the higher the speedup, because more executions can share graph and data accesses. For unweighted graphs, batched closeness centrality shows 5-11× speedup over non-batched execution, depending on the analyzed graph. The significantly more complex batched betweenness centrality algorithm exhibits 2-7× speedup on unweighted graphs.

Both algorithms' speedups are a result of the amortized memory access costs achieved by batched execution. For weighted graphs our batched weighted closeness centrality algorithm shows between 3 and 6× better performance than the respective state-of-the-art non-batched algorithm. Batched betweenness centrality on weighted graphs shows up to 3× speedup. Our measurements show, however, that batching can also have a negative impact on execution performance for low numbers of concurrent executions. This is the case when the additional computation added by batched processing cannot be amortized by the savings from its improved memory access pattern. When enough concurrent executions exist, the speedups of our weighted algorithms are again caused by the amortized memory access costs, but also by the increased numeric throughput achieved by using SIMD instructions during the geodesic distance computation.

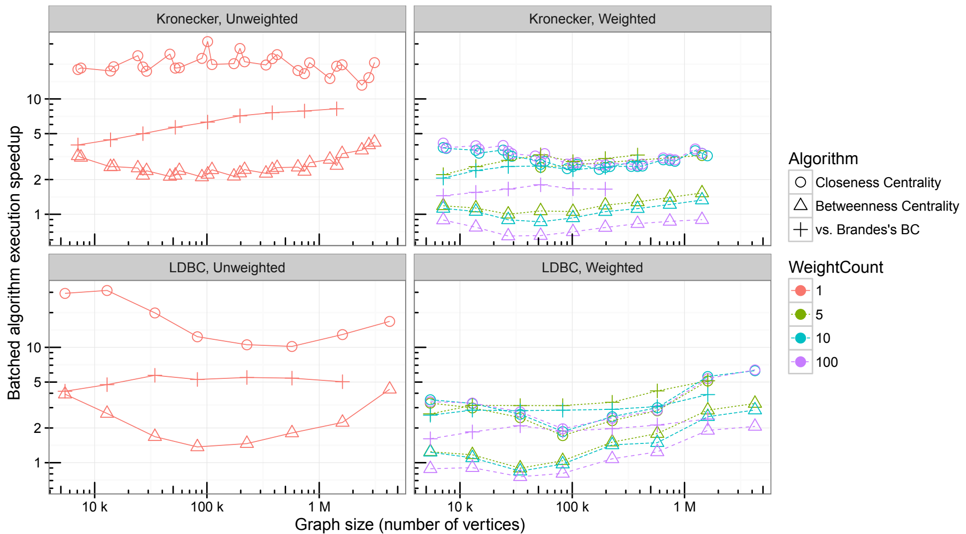


Fig. 4: Speedup of batched algorithms for Kronecker and LDBC graphs of various sizes.

6.3 Graph Size Scalability

Batched algorithm execution is designed to amortize the cost of random data accesses. While random accesses do not significantly impact small graphs with less than a million edges—as such graphs easily fit into modern CPU's caches—for large real-world graphs they are likely to cause cache misses, which lead to high-latency main memory accesses and CPU stalls. We evaluate the speedup of our proposed batched algorithms over non-batched implementations for Kronecker and LDBC graphs of various sizes in Figure 4. The figure shows the speedups of our batched closeness centrality and betweenness centrality algorithms over non-batched execution as circles and triangles, respectively, and depicts our speedup compared to Brandes's implementation using crosses. All measurements were done for graphs with various counts of edge weights, which we represent as different colors and line types.

Our measurements exhibit the expected effects: When small graphs are analyzed, both the graph and the algorithm's working set fit into the CPU cache. In this situation, batched execution benefits mostly from vectorization and avoided redundant computation in the concurrent executions. For medium-size graphs, the speedup of our batched algorithms reduces slightly. The reason for this is that because batched algorithms run multiple executions concurrently, they have a larger working set than non-batched algorithms and outgrow the CPU cache faster. For large graphs, the speedup of batched execution increases again. At this point, both the batched and non-batched algorithms' graph data and working sets do not fit into the CPU cache anymore, but only batched algorithms efficiently amortize the cost of main memory accesses. While both the Kronecker and the LDBC graphs exhibit the expected behavior, Figure 4 shows that it is more pronounced for the LDBC graph.

The achieved speedups are similar for Kronecker and LDBC graphs of similar sizes as we already discussed in the previous section. Compared to Brandes's implementation, our batched betweenness centrality algorithms show significant speedups of around 5× for both datasets in the unweighted case or with few different weights.

6.4 Edge Weight Count Scalability

We further evaluated how the number of different edge weights influences the batched algorithms' speedup. Figure 4 shows that our weighted closeness centrality algorithm is nearly independent of the number of different weights. Its runtime is dominated by the batched Bellman-Ford shortest distance computations, which in turn is mostly influenced by the graph's diameter, as discussed in Section 5.1.

In contrast, our weighted betweenness centrality algorithm is noticeably influenced by the weight count. While it also uses the batched Bellman-Ford distance algorithm, its runtime is dominated by the forward and reverse traversal in the graph, which allows less sharing when more different weights are used. A consequence of this reduced sharing is that for very high weight counts our speedup over Brandes's algorithm become less pronounced.

7 Related Work

For decades there has been research on the geodesic distance problem, which is the basis for closeness and betweenness centrality calculations. We focus on distances in small-world networks—low-diameter graphs with a degree distribution that follows the power law. To calculate geodesic distances in unweighted and non-indexed small-world networks we build on the MS-BFS algorithm [Th14]. Kaufmann et al. [Ka17] recently proposed the highly-optimized parallelized MS-PBFS which is orthogonal to our algorithms and can be used to improve the performance of our MS-BFS-based unweighted centrality algorithms.

For weighted non-indexed small-world networks we confirmed [Ye70] that the Bellman-Ford algorithm is very efficient and furthered this work with our batched Bellman-Ford algorithm. This is the first work that proposes batched execution for geodesic distance calculation in dense weighted graphs. In contrast to techniques that are designed for general graphs like Thorup et al. [Th04], our algorithms are optimized for the specifics of dense graphs and the characteristics of modern systems with long memory access latencies. However, while our algorithms optimize for the properties of dense small-world networks, they work on general graphs and do not require special properties like planarity [Kl05]. Furthermore, our work does not use prior graph indexing, as is done in [AIY13], but may be suited to speed up existing indexing techniques.

Once all geodesic distances from a vertex are known, this vertex's closeness centrality value can be computed. Because exact closeness calculation is very expensive in large real-world graphs, approximate algorithms [EW01] and heuristics to find the top-k vertices with the highest closeness centrality values [OLH14] were proposed. Our batched unweighted

closeness centrality algorithm is orthogonal to these approximations and heuristics, and can be used to improve their performance. Building on Brandes's algorithm [Br01], parallelized exact betweenness centrality algorithms [Ma09] as well as approximations [Ba07] have been proposed. The concepts of our batched betweenness centrality algorithm can be applied to further improve their performance. We found existing work on centralities to only explicitly cover the case of unweighted graphs. Weighted graphs are seen as a trivial extension and only briefly mentioned. This is the first work to explicitly evaluate the tradeoffs and optimizations of weighted betweenness centrality and how batching can be applied to it.

8 Conclusion

Batched algorithm execution significantly improves the runtime of distance, closeness centrality, and betweenness centrality computation on unweighted and weighted graphs. In future work we want to find further algorithms that are suited for batched execution.

9 Acknowledgments

This research was supported by the German Research Foundation (DFG), Emmy Noether grant GU 1409/2-1, and by the Technical University of Munich - Institute for Advanced Study, funded by the German Excellence Initiative and the European Union Seventh Framework Programme under grant agreement no 291763, co-funded by the European Union. Manuel Then is a recipient of the Oracle External Research Fellowship. Part of this work was conducted during an internship at Oracle Labs.

References

[AIY13] Akiba, Takuya; Iwata, Yoichi; Yoshida, Yuichi: Fast Exact Shortest-Path Distance Queries on Large Networks by Pruned Landmark Labeling. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. ACM, pp. 349–360, 2013.

[Ba07] Bader, David A; Kintali, Shiva; Madduri, Kamesh; Mihail, Milena: Approximating Betweenness Centrality. In: International Workshop on Algorithms and Models for the Web-Graph. Springer, pp. 124–137, 2007.

[Be58] Bellman, Richard: On a Routing Problem. Quarterly of applied mathematics, pp. 87–90, 1958.

[BP98] Brin, Sergey; Page, Larry: The Anatomy of a Large-Scale Hypertextual Web Search Engine. In: WWW. pp. 3825–3833, 1998.

[Br01] Brandes, Ulrik: A Faster Algorithm for Betweenness Centrality. Journal of Mathematical Sociology, 25:163–177, 2001.

[EW01] Eppstein, David; Wang, Joseph: Fast Approximation of Centrality. In: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics, pp. 228–229, 2001.

[Fr78] Freeman, Linton C: Centrality in Social Networks Conceptual Clarification. Social networks, 1(3):215–239, 1978.

[FT87] Fredman, Michael L.; Tarjan, Robert Endre: Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. J. ACM, 34(3):596–615, July 1987.

[Ho15] Hong, Sungpack; Depner, Siegfried; Manhardt, Thomas; Van Der Lugt, Jan; Verstraaten, Merijn; Chafi, Hassan: PGX. D: a fast distributed graph processing engine. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, p. 58, 2015.

[Io16] Iosup, Alexandru; Hegeman, Tim; Ngai, Wing Lung; Heldens, Stijn; Prat, Arnau; Manhardt, Thomas; Chafi, Hassan; Capota, Mihai; Sundaram, Narayanan; Anderson, Michael et al.: LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. Proceedings of the VLDB Endowment, 9(12), 2016.

[Ka17] Kaufmann, Moritz; Then, Manuel; Kemper, Alfons; Neumann, Thomas: Parallel Array-Based Single- and Multi-Source Breadth First Searches on Large Dense Graphs. In: EDBT. 2017.

[Kl05] Klein, Philip N: Multiple-Source Shortest Paths in Planar Graphs. In: SODA. volume 5, pp. 146–155, 2005.

[Ku13] Kunegis, Jérôme: Konect: The Koblenz Network Collection. In: Proceedings of the 22nd international conference on World Wide Web companion. International World Wide Web Conferences Steering Committee, pp. 1343–1350, 2013.

[Ma09] Madduri, Kamesh; Ediger, David; Jiang, Karl; Bader, David A; Chavarria-Miranda, Daniel: A Faster Parallel Algorithm and Efficient Multithreaded Implementations for Evaluating Betweenness Centrality on Massive Datasets. In: Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on. IEEE, pp. 1–8, 2009.

[Ma10] Malewicz, Grzegorz; Austern, Matthew H; Bik, Aart JC; Dehnert, James C; Horn, Ilan; Leiser, Naty; Czajkowski, Grzegorz: Pregel: A System for Large-Scale Graph Processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, pp. 135–146, 2010.

[NSJ11] Ni, Chaoqun; Sugimoto, Cassidy; Jiang, Jiepu: DegreE, Closeness, and Betweenness: Application of Group Centrality Measurements to Explore Macro-Disciplinary Evolution Diachronically. In: Proceedings of ISSI. pp. 1–13, 2011.

[OLH14] Olsen, Paul W.; Labouseur, Alan G.; Hwang, Jeong-Hyon: Efficient Top-k Closeness Centrality Search. In: 2014 IEEE 30th International Conference on Data Engineering. pp. 196–207, March 2014.

[Th04] Thorup, Mikkel: Integer Priority Queues with Decrease Key in Constant Time and the Single Source Shortest Paths Problem. J. Comput. Syst. Sci., 69(3):330–353, November 2004.

[Th14] Then, Manuel; Kaufmann, Moritz; Chirigati, Fernando; Hoang-Vu, Tuan-Anh; Pham, Kien; Kemper, Alfons; Neumann, Thomas; Vo, Huy T.: The More the Merrier: Efficient Multi-source Graph Traversal. Proceedings of the VLDB Endowment, 8(4):449–460, December 2014.

[Ye70] Yen, Jin Y: An Algorithm for Finding Shortest Routes from All Source Nodes to a Given Destination in General Networks. Quarterly of Applied Mathematics, pp. 526–530, 1970.