Discovering and Balancing Fundamental Cycles in Large Signed Graphs

Ghadeer Alabandi
Texas State University
San Marcos, TX, U.S.A
gaa54@txstate.edu

Jelena Tešić
Texas State University
San Marcos, TX, U.S.A
jtesic@txstate.edu

Lucas Rusnak
Texas State University
San Marcos, TX, U.S.A
lucas.rusnak@txstate.edu

Martin Burtscher
Texas State University
San Marcos, TX, U.S.A
burtscher@txstate.edu

ABSTRACT

Computing consensus states via global sign balancing is a key step in social network analysis. This paper presents graphB+, a fast algorithm for balancing signed graphs based on a new vertex and edge labeling technique, and a parallel implementation thereof for rapidly detecting and balancing all fundamental cycles. The main benefits of graphB+ are that the labels can be computed with linear time complexity, only require a linear amount of memory, and that the running time for balancing a cycle is linear in the length of the cycle times the vertex degrees but *independent* of the size of the graph. We parallelized graphB+ using OpenMP and CUDA. It takes 0.85 seconds on a Titan V GPU to balance the signs on the edges of an Amazon graph with 10 million vertices and 22 million edges, amounting to over 14 million fundamental cycles identified, traversed, and balanced per second.

CCS CONCEPTS

• **Computing methodologies** → **Massively parallel algorithms**.

KEYWORDS

Fundamental cycles, Signed-graph balancing, Parallelization, GPU computing

ACM Reference Format:
Ghadeer Alabandi, Jelena Tešić, Lucas Rusnak, and Martin Burtscher. 2021. Discovering and Balancing Fundamental Cycles in Large Signed Graphs. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3458817.3476153

1 INTRODUCTION

On-line social networks have become an important mode of human interaction. For instance, people receive news, participate in surveys, and express opinions via on-line social networks. Thus far, the field of Social Network Analysis has largely focused on community discovery [21], topic trending [32], quantifying the influence of a person [2], and recommender systems [24]. If a decision must be made in these tasks, majority voting is typically used [35]. However, this approach ignores any underlying network structure and is prone to bias due to super-influencers, people trying to game the system, etc. Statistical parity has been employed to mitigate bias on large data [15, 45], but it does not consider the more important and informative *network-wide consensus states*. Yet, consensus across social networks has already been well-researched in the field of psychology [1, 9, 17, 18], albeit at a small scale. This paper describes the parallelizable graphB+ algorithm that makes it possible to compute nearest consensus states of large real-world social networks based on this proven balance model from psychology.

Signed social networks are graphs where the edges store the attitude of a vertex (person) towards another. Two vertices (people) connected by an edge can be agreeable or antagonistic. A signed graph is *balanced*, i.e., in a global consensus state, if every cycle comprises an even number of antagonistic edges. This is the case because two antagonistic edges cancel each other out.

Social balance theory [20] and the mathematical foundation of attitudinal graphs [17] were the first to define and model consensus in social networks. Wasserman et al. introduced social network analysis in the form of algebraic graph representations and proposed a series of statistical tests [42]. Subsequent work mostly focused on predicting the existence of edges and sentiments in the graph, recommending products, or identifying unusual trends while relying on consensus-based models [15, 21]. However, these kinds of algorithms are rarely scrutinized for equity because consensus and majority voting are such well-established social constructs [25]. The few works that do scrutinize them have only examined how controversial the outcome is relative to the system-wide consensus [26] or have measured how subgroups mobilize against other groups [23]. To improve upon this, Rusnak and Tesic recently proposed the concept of frustration cloud analysis [33]. Their work considers all nearest consensus-driven balanced states. However, their approach does not scale to graphs with more than a few thousand vertices.

Our work focuses on making the needed graph operations efficient, in particular the discovery and balancing of the fundamental cycles (cf. Section 2). Our graphB+ implementation can handle real-world inputs [11, 19] with billions of edges on a single CPU or GPU. This paper makes the following contributions.

- A vertex and edge labeling technique for finding the fundamental cycles in large graphs that requires a linear amount of memory and can be computed in linear time.
- The graphB+ algorithm to traverse and balance each fundamental cycle in an amount of time that is linear in the product of the cycle length and the average vertex degree but independent of the size of the graph.

SC '21, November 14–19, 2021, St. Louis, MO, USA

ACM ISBN 978-1-4503-8442-1/21/11…$15.00
https://doi.org/10.1145/3458817.3476153

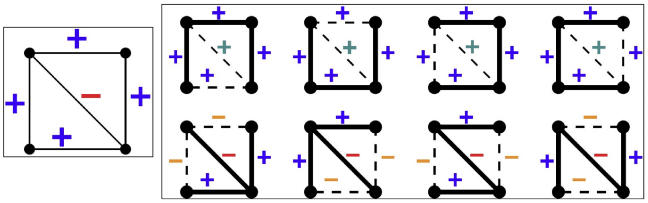

Figure 1: (a) signed input graph Σ; (b) all spanning trees and the corresponding nearest structurally balanced states of Σ

- A parallelization approach for each step of the graphB+ algorithm for OpenMP and CUDA.
- A performance evaluation that demonstrates the scalability and efficacy of our implementation.

The rest of the paper is organized as follows. Section 2 provides background information on balancing the signs of signed graphs. Section 3 explains the graphB+ algorithm and its parallelization. Section 4 summarizes related work. Section 5 describes the experimental methodology. Section 6 presents and analyzes the results. Section 7 concludes the paper with a summary.

2 BACKGROUND

This section explains how the balancing of signed graphs works and why it is important. It also describes performance issues of prior implementations and introduces related terminology.

As mentioned, two vertices in a signed graph that are connected by an edge can be agreeable (positive) or antagonistic (negative) [17]. The sign of a path in such a graph is the product of the signs of its edges, and a closed path is a cycle. A signed graph is *balanced* if every cycle is positive. Balance is equivalent to the following agreeability condition: A signed graph is balanced if, and only if, its vertices can be bipartitioned so that all negative edges occur between the *Harary bipartition* [9]. Fig. 1(a) shows a signed graph Σ with 4 vertices and 5 edges, and Fig. 1(b) summarizes all balanced states of Σ, where the negative-edge cutsets represent the *Harary cuts*. The Harary bipartition separates the vertices of the balanced graph into two sets such that the vertices in both sets internally agree with each other but disagree with the vertices from the other set. The ultimate goal is to identify the vertices and edges of the resulting balanced majority, i.e., the larger of these two sets.

The *frustration index* $Fr(\Sigma)$ of a signed graph Σ is the minimum number of edges whose negation results in balance [18, 20]. It measures the deviation from consensus. A balanced signed graph has $Fr(\Sigma) = 0$ and all paths between two vertices have the same sign. The frustration index can be expanded to a *frustration cloud* [33] by building a set that includes all *nearest* balanced states, i.e., all balanced signed graphs of Σ that require a minimal number of edge sign switches to produce a balanced graph. A balanced state is minimal (nearest) if no subset of the edge sign switches yields a balanced state. We are not interested in non-nearest balanced states since, based on social balance theory, we assume the group will not continue the discussion once it has reached a consensus. Fig. 2 shows the frustration cloud of the signed graph from Fig. 1(a). It contains 5 balanced states. *Note that balancing only affects the signs of the edges (it negates a few of them to reach a consensus state); it never adds or removes graph vertices or edges.*

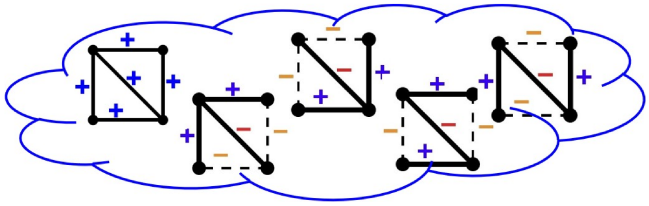

Figure 2: The frustration cloud of Σ contains a set of five unique nearest balanced states

2.1 Tree-based Signed Graph Balancing

Alg. 1 computes a nearest balanced state for the signed input graph Σ and the spanning tree T, as proven elsewhere [33].

Algorithm 1 Tree-based Signed Graph Balancing

Input: Signed graph $\Sigma = (G, \sigma)$
Input: Spanning tree T of Σ
  for all edges e, $e \in \Sigma \setminus T$ **do**
    if fundamental cycle $T \cup e$ is negative **then**
      switch edge sign: $e^- \rightarrow e^+; e^+ \rightarrow e^-$
    end if
  end for
Output: Balanced graph Σ_T

If T is a spanning tree of a given connected signed graph Σ and e is an edge of Σ that does not belong to T, then the *fundamental cycle* C_e, defined by e, is the cycle consisting of e together with the simple path in T that connects the endpoints of e. If m denotes the number of edges and n the number of vertices in Σ, there are exactly $m - (n - 1)$ fundamental cycles, one for each edge that does not belong to T. Each C_e is linearly independent from the remaining cycles because it includes an edge e that is not present in any other fundamental cycle. Different spanning trees can produce the same or a different nearest balanced state as outlined in Fig. 1, where the top balanced state is represented more often than the others as more spanning trees converge to this state.

2.2 Tree-sampling-based Harary Bipartitioning

The frustration cloud encompasses all nearest balanced states of a signed graph Σ. To discover each of these states, Alg. 1 must be run with *all* spanning trees T of Σ. However, the total number of spanning trees for a graph grows exponentially with the number of vertices n. For example, the graph Σ in Fig. 1 with only 4 vertices and 5 edges already has 8 spanning trees. The highland tribes graph with 16 vertices and 58 edges [11] has 402,506,278,163 spanning trees. Since a small graph with just 16 vertices can have over 400 billion spanning trees, it is intractable to include all spanning trees in a graph-balancing computation for real-world social networks. As a remedy, a tree-sampling approach has been proposed [33]. Since spanning trees resulting from breadth-first searches (BFS) yield the maximum number of fundamental cycles of minimal length

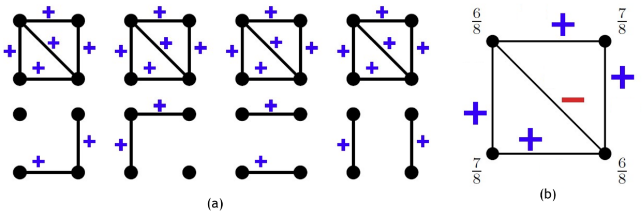

Figure 3: (a) Harary bipartitions of the balanced states produced by Alg. 1; (b) corresponding *status* of the vertices of Σ

[34], we adopt the BFS approach to maximize the resolution of the subsequent metric analysis [33].

Algorithm 2 Tree-sampling-based Harary Bipartitioning

Input: Signed graph $\Sigma = (G, \sigma)$
Input: Spanning-tree sampling method M (e.g., BFS, DFS, random)
  generate set $\mathcal{T}_k$ of k spanning trees of Σ using M
  for all spanning trees T, $T \in \mathcal{T}_k$ **do**
    find nearest balanced state Σ_T using Alg. 1 (or Alg. 3)
    form Harary bipartition of Σ_T
  end for
Output: k Harary bipartitions of Σ

Alg. 2 outlines how to compute a selection of Harary bipartitions using tree sampling. Once a nearest balanced state for a spanning tree is found, we identify the Harary cutset for that state and remove all negative edges. Then, we compute the induced bipartition. For our example graph Σ, the resulting Harary bipartitions are illustrated in Fig. 3(a). We use the bipartitions to derive vertex and edge attributes that are important for global consensus analysis.

2.3 Balancing-based Graph Attributes

The *status* of a vertex (person) denotes the probability it will be in the majority, i.e., the likelihood that a vertex v in $\Sigma = (G, \sigma)$ belongs to the larger bipartition over all nearest balanced states of Σ. In Fig. 3(a), the top left vertex belongs to the larger Harary bipartition 6 out of 8 times, so its status is $6/8 = 0.75$, as shown in Fig. 3(b). For the tree-sampling approach, the status is defined as the normalized sum over the sampled trees T in which v ends up in the larger bipartition: $status(v) = \frac{1}{|\mathcal{T}_G|} \sum_{T \in \mathcal{T}_G} \delta_T(v)$, where $\delta_T(v) = 1$ if v is in the larger bipartition, 0.5 if the bipartitions have the same size, and 0 otherwise. Thus, the status measures how likely a vertex is to contribute to the consensus decision. Other attributes for global consensus analysis, such as *agreement*, *influence*, and *authority*, can similarly be computed from the Harary bipartitions [33].

2.4 Benefits of Graph Balancing Attributes

This section illustrates some benefits of using graph-balancing-based attributes on the example of the Wikipedia Requests for Administratorship (wiki-Elec) dataset, which contains 7,115 vertices (either casting a vote or being voted on) and 103,689 edges (positive or negative votes). Importantly, the dataset also includes the actual results: 1,200 promotions and 1,500 refusals [25].

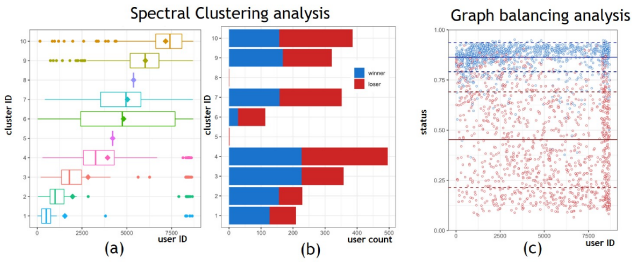

Figure 4: wiki-Elec data outcome (blue: won, red: lost) analysis using spectral clustering and graph balancing: (a) box-plot of spectral clusters over user IDs; (b) cluster makeup by election outcome; (c) outcome analysis where x-axis denotes user ID and y-axis denotes *status* computed using a sample of 1000 nearest balanced states.

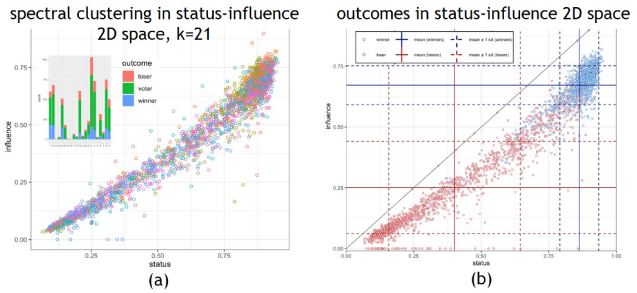

Figure 5: wiki-Elec data analysis in the status-influence graph-balancing space (x-axis denotes status, y-axis denotes influence): (a) spectral clustering results for 21 color-coded clusters; (b) actual outcome results.

Fig. 4 shows the wiki-Elec analysis results for spectral clustering and graph balancing. Spectral clustering for $k = 10$ clusters predominantly forms clusters based on users' adjacency rather than on their sentiments towards each other, as illustrated in the boxplots in Fig. 4(a). Note that the user IDs are assigned in temporal order. Fig. 4(b) shows that spectral clustering does not cluster by election outcome as the relative number of winners and losers is similar in each cluster. In contrast, the status derived from graph balancing discriminates user influence based on the entire signed graph, as depicted in Fig. 4(c), which exhibits a high correlation between the status of a person and winning the election, irrespective of user ID.

Fig. 5 presents additional advantages of graph-balancing-based attribute analysis. The results of spectral clustering and majority voting as well as the actual outcome are plotted in the graph-balancing status-influence space. Spectral clustering (Fig. 5(a)) yields little relevant information. The outcome plot (Fig. 5(b)) identifies people who are not in line with other, similar people (low-status, low-influence should lead to rejection whereas high-status, high-influence should lead to acceptance), and the votes for them should be examined for possible bias. Note that majority voting by itself cannot expose these examples of potential outcome bias.

2.5 Complexity Analysis of Prior Work

Tesic and Rusnak's original graph-balancing code [39], which is based on Python, stores the spanning trees in h5 files and uses dictionaries to record edge information. Unfortunately, this makes the

code too slow and memory intensive to process the large graphs we are interested in. Its work complexity is $O(n \times m)$ per tree, where n is the number of vertices and m the number of graph edges. The space complexity is $O(n \times n)$ since the code is based on an adjacency matrix. Computing 1000 nearest balanced states on the small wiki-Elec graph takes about 1.5 hours on a 16-node HPC cluster with two 14-core 2.4 GHz Xeon processors per node. Processing the somewhat larger but still relatively small social networks Slashdot and Epinions with about 500k edges [25] on the same system exceeded the 60 GB maximum memory utilization per node when using BFS spanning trees. Switching to random spanning trees with Spark parallelization made it work but required 40 minutes to compute just 20 balanced states. Hence, this implementation cannot be used on social networks with millions of users and edges [28].

3 THE GRAPHB+ ALGORITHM

We designed a new graph-balancing algorithm from scratch, which we named "graphB+". It incorporates a novel approach for efficiently identifying, traversing, and balancing all fundamental cycles of a graph. Alg. 3 outlines how graphB+ works on the signed graph $\Sigma = (G, \sigma)$ using a provided spanning tree T of Σ. The tree can be generated with any spanning tree algorithm. graphB+ requires one word of storage per vertex to record the new ID as well as two words of storage per (tree) edge to record the beginning and end of the reachable vertex range, i.e., a linear amount of memory.

Algorithm 3 graphB+ Algorithm

Input: $\Sigma = (G, \sigma)$
Input: Spanning tree $T \in \Sigma$
  for all vertices $v, v \in T$ **do** ▷ Vertex relabeling
    relabel ID of vertex v based on a pre-order traversal of T
  end for
  for all edges $e, e \in T$ **do** ▷ Edge labeling
    record $[ID_{min}, ID_{max}]$-range of vertices that are reachable in T when traversing e based on a pre-order and a post-order traversal of T
  end for
  for all edges $e, e \in \Sigma \setminus T, e = (v_{src}, v_{dst})$ **do** ▷ Cycle traversal
    set count $c = 0$
    set vertex $v = v_{src}$
    while $v \neq v_{dst}$ **do**
      find edge $g = (v, v_{nxt})$ whose range includes v_{dst}
      increment c if g is negative
      traverse edge g: set $v = v_{nxt}$
    end while
    if c is odd **then**
      switch edge sign: $e^- \rightarrow e^+; e^+ \rightarrow e^-$
    end if
  end for
Output: Balanced graph Σ_T

We illustrate the operation of the graphB+ algorithm, including the preceding and following steps to provide context, on the signed graph shown in Fig. 6(a). The red pluses and minuses indicate the signs of the edges and are part of the input. Assume vertex R is selected to be the root of the spanning tree as illustrated in Fig. 6(b). The resulting BFS tree is outlined in Fig. 6(c), where the tree edges are arrows pointing from the parent to the child and the non-tree edges are dashed. This constitutes the input of the graphB+ algorithm, which performs the following three computation steps.

1) graphB+ relabels the vertices as outlined in the *vertex relabeling* step of Alg. 3. It does this by performing a pre-order traversal of the spanning tree. During this traversal, each reached vertex is assigned a new ID that is equal to the number of previously visited vertices. The result of this relabeling is depicted in Fig. 6(d).

2) graphB+ records a range (a pair of values) on each tree edge as outlined in the *edge labeling* step of Alg. 3. This range denotes which vertices, identified by their new IDs, are reachable when traversing the edge in the parent-to-child direction. The result is illustrated in Fig. 6(e). Traversing an edge in the opposite (child-to-parent) direction leads to the inverse of the recorded range, i.e., all vertex IDs excluding those in the range. The beginning of each range is determined using a pre-order traversal of the tree and the end using a post-order traversal. Note that the ranges can always be expressed by just two values because of the prior vertex relabelling step, which guarantees each range to be a contiguous set of vertex IDs. *This feature of graphB+ is essential to keep the memory consumption low and to make the following cycle traversals fast.*

3) graphB+ identifies and traverses all cycles that are created when inserting one non-tree edge at a time as outlined in the *cycle traversal* step of Alg. 3. We illustrate how this works on the example of edge $6 \rightarrow 7$, which requires us to start with vertex 7 and search for the path in the tree that leads back to vertex 6 to complete the cycle. With the help of the recorded ranges, this path can be found efficiently. First, we search the edges in vertex 7's adjacency list to find the one that eventually leads to vertex 6. Specifically, we search the range of each outgoing edge as well as the inverse of the range of the incoming parent edge. In this case, we find that edge $0 \rightarrow 7$ traversed in the opposite direction lies on the path to vertex 6 as it leads to all vertex IDs other than 7 through 9. We select this edge and traverse it to reach vertex 0. Second, we search vertex 0's edge ranges and find that 6 is in the range of edge $0 \rightarrow 3$. Hence, we move on to vertex 3. Third, we search vertex 3's tree-edge ranges and find that 6 is in the range of edge $3 \rightarrow 6$. Travsersing this edge to vertex 6 completes the cycle as illustrated in Fig. 6(f). Note that we never visited a vertex that is not on the cycle. As we process the cycle, we count the number of traversed edges with a negative sign (one in the example) and set the sign of the non-tree edge $6 \rightarrow 7$ such that the cycle has an even number of negative signs (negative in the example). The remaining non-tree edges undergo the same procedure, ultimately yielding the balanced graph Σ_T with the same vertices and edges as Σ but possibly different signs on the non-tree edges. The resulting balanced graph is presented in Fig. 6(g), which includes two changed signs, one on edge $B \rightarrow F$ and the other on edge $D \rightarrow H$. This concludes the graphB+ computation.

Balancing only the fundamental cycles (e.g., cycles $B \rightarrow R \rightarrow E \rightarrow F \rightarrow B$ and $R \rightarrow A \rightarrow D \rightarrow E \rightarrow R$ in Fig. 6(g)) guarantees that all other cycles are also balanced (e.g., cycle $B \rightarrow R \rightarrow A \rightarrow D \rightarrow E \rightarrow F \rightarrow B$). This is the case because, based on algebraic graph theory, all cycles of a graph can be constructed via binary sums of the 0,1-choice vectors of the fundamental cycles [13, 16]. Once each fundamental cycle is balanced, all paths between two vertices in a cycle have the same sign [17]. Any new cycles formed by

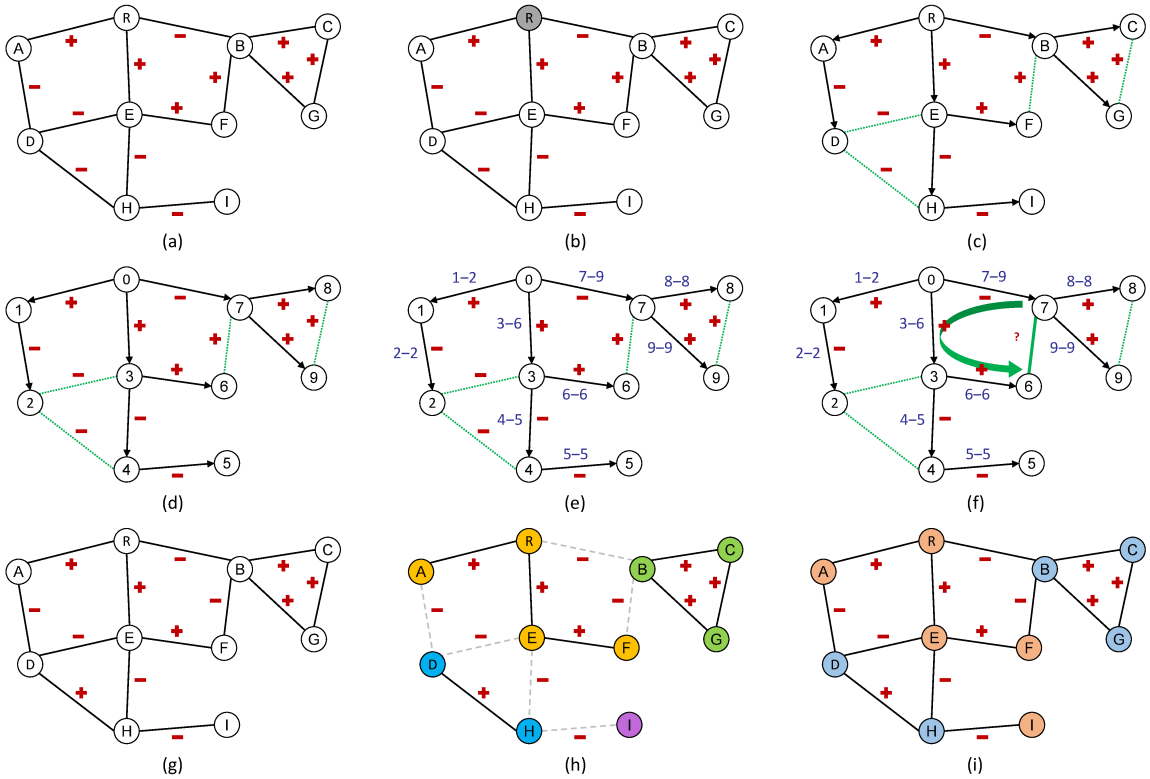

Figure 6: Example illustrating the steps before (a-c), during (d-g), and after (h-i) the graphB+ computation: (a) signed sample graph, (b) root selection, (c) spanning BFS tree, (d) graphB+ vertex relabeling, (e) graphB+ tree-edge labeling, (f) graphB+ cycle balancing, (g) resulting balanced graph, (h) Harary cuts, and (i) Harary bipartitions

overlapping fundamental cycles inherit this property by following another path in the second cycle, thus forcing global balance.

As outlined in Alg. 2, the output of graphB+ can, for example, be used to determine the Harary bipartition. This may be done as follows. First, all negative edges from Σ_T are ignored and the resulting connected components (CCs) are computed, which are color-coded in Fig. 6(h). Second, the bipartition is formed by combining all CCs with an even number of negative edges between them. This can be done by collapsing each CC into a single vertex, performing a BFS on the resulting graph, and assigning all vertices in the even levels of the BFS tree to one bipartition and the remaining vertices to the other. Fig. 6(i) illustrates the result, where the brown vertices make up one bipartition and the blue vertices the other.

3.1 Complexity Analysis

Assume Σ to be a connected graph with n vertices and m edges. Since Σ is connected, $m \geq n-1$ and any spanning tree T of Σ has $n-1$ edges. Step 1 of the graphB+ algorithm performs a pre-order traversal of T, which requires $O(n)$ work as it processes each vertex and each tree edge. Step 2 also requires $O(n)$ work since it performs both a pre- and a post-order traversal of T. Step 3 requires $O(m \times k \times t)$ work, where t is the average tree-degree of the vertices on a cycle, k is the average cycle length, and there are $m-(n-1) = O(m)$ fundamental cycles (i.e., non-tree edges). Since there exists a path in T from any vertex a to any other vertex b that goes through the root of the tree, and assuming that $d(x)$ denotes the tree depth of vertex x, $d(a)+d(b)+1$ is an upper bound on the length of the cycle formed by the edge $a \rightarrow b$. Hence, the average cycle length is linear (or sub-linear) in the average tree depth. For a random BFS tree of a random graph, the expected average tree depth is $O(log(n))$. Thus, we expect Step 3 to require $O(m \times log(n) \times t)$ work. Since $m \geq n-1$, Step 3 dominates and determines the work complexity of the graphB+ algorithm. As mentioned, the space complexity is $O(n+m)$ because graphB+ stores a constant amount of information in each vertex and on each edge. In contrast, the preexisting signed-graph-balancing code (cf. Section 2.5) has a work complexity of $O(n \times m)$ per tree and a space complexity of $O(n \times n)$. Hence, graphB+ represents a substantial improvement.

3.2 Implementation

This section discusses key aspects of our graphB+ implementation that help make it fast. The C++/OpenMP and CUDA codes, which parallelize the entire graphB+ algorithm, are freely available under the 3-clause BSD license [3].

3.2.1 Data Structure. We employ the widely-used, compact, and quick-to-access compressed sparse row (CSR) format [8, 37] to represent the graph in memory. To keep the memory consumption low, only a single copy of the graph exists, which we use to represent Σ, T, and Σ_T. All required dynamic memory is allocated once at the beginning of the program and reused throughout the processing.

To speed up the cycle traversal, which is the most performance critical code section, the edge data is encoded in two words as follows. The beginning of the range is stored together with a 1-bit value that indicates whether the range is inverted. The end of the range is stored together with a 1-bit value that indicates the sign of the edge. This encoding minimizes the memory footprint, which boosts in-cache presence and, therefore, performance.

3.2.2 Adjacency Lists. To accelerate the graph traversals and cycle processing, our implementation partitions the adjacency list of each vertex such that the tree edges precede the non-tree edges and moves the parent edge (if present) to the front of the list. Moreover, it uses a 1-bit flag to mark whether an edge is in the tree or not. The partitioning is fast and only takes linear time. Its execution time is easily amortized as it enables the following optimizations.

- All loops that process non-tree edges traverse the adjacency list from the back to the front and terminate as soon as they encounter the first tree edge.
- All loops that process tree edges traverse the adjacency list from the front to the back and stop when they encounter the first non-tree edge (or reach the end of the list).
- All loops that process tree edges process the parent edge first, which boosts performance since this is the most likely edge to be needed when traversing the cycles because, on average, it leads to the largest number of vertices.

3.3 Parallelization

Parallelizing Alg. 2 is straightforward for distributed-memory systems. Each compute node gets a copy of the graph and a subset of the tree roots. The compute nodes then independently generate a spanning tree, run graphB+, and count how often each vertex ends up in the larger Harary bipartition. A single MPI_Reduce call at the end suffices to obtain the status of each vertex. However, most compute nodes (as well as workstations and laptops) contain multiple CPUs and many include GPUs. Hence, the graphB+ algorithm should also be parallelized within a compute node for maximum performance. This is the focus of the rest of the paper.

Within a compute node, the simplest way to parallelize Alg. 2 is also to process a different given tree on each thread because the k trees can be balanced independently. However, this approach does not scale. On high-end CPUs, which need dozens of threads to keep their cores busy, it would result in dozens of trees being stored in memory and accessed at the same time, which limits the maximum graph size and hurts performance due to poor locality of reference. On high-end GPUs, which require on the order of 100,000 threads to unleash their full performance, this parallelization technique would only work for tiny graphs and would result in far more trees being processed than needed, thus eliminating any speedup.

We use a different strategy where the threads collaborate to make the processing of a single tree faster. This requires more complex techniques but still accelerates the computation while making it possible to handle large graphs with hundreds of millions of edges or more on a single CPU or GPU. Whereas both our OpenMP and CUDA codes fully parallelize the entire graphB+ algorithm, the following paragraphs describe only the key parallelization aspects of our implementation.

3.3.1 Vertex and Edge Labeling. Re-labeling the vertex IDs (Step 1) is based on a pre-order traversal of the tree and labeling the edges with the ranges (Step 2) is based on a pre- and a post-order traversal. These traversals are difficult to parallelize. However, the same result can be obtained with a bottom-up followed by a top-down pass over the tree levels as outlined in Alg. 4. With this approach, all vertices at a given level (i.e., tree depth) can be processed in parallel, meaning the three for-all loops in Alg. 4 are parallel.

First, the code computes the size of the subtree rooted in each tree node. It starts with a count in each vertex that is initialized to one. The bottom-up pass atomically adds the count of each vertex to the count of its parent. The level-by-level order guarantees that we only use a vertex's final count (subtree size) to update its parent.

Second, the top-down pass assigns the edge ranges and the new vertex IDs based on these counts. For each parent p, the code traverses the children and assigns them their new ID. The ID of child c is the ID of p plus 1 plus the counts of all earlier children of p. This value also serves as the beginning of the range stored in the edge $p \rightarrow c$. The end of the range is the beginning value plus the count of c minus 1, i.e., one less than the new ID of the next child.

Algorithm 4 Parallel Vertex and Tree-edge Labeling

Input: $\Sigma = (G, \sigma)$
Input: Spanning tree $T \in \Sigma$
  for all vertices v, $v \in \Sigma$ **do** ▷ Initialization
    $v_{cnt} = 1$
  end for
  for level l of T from bottom to top **do** ▷ Bottom-up pass
    for all vertices v, $v \in level[l]$ **do**
      $p = v_{parent}$
      atomic add: $p_{cnt} = p_{cnt} + v_{cnt}$
    end for
  end for
  $root_{id} = 0$
  for level l of T from top to bottom **do** ▷ Top-down pass
    for all vertices v, $v \in level[l]$ **do**
      $n = v_{id} + 1$
      for vertices c, $c \in child[v]$ **do**
        $e = edge\ v \rightarrow c$
        $e_{min} = n$
        $e_{max} = n + c_{cnt} - 1$
        $c_{id} = n$
        $n = n + c_{cnt}$
      end for
    end for
  end for
Output: Re-labeled vertex IDs (v_{id})
Output: Ranges on edges of T (e_{min}, e_{max})

Since we target signed social networks, which tend to be scale-free graphs, we expect the graph diameter to be low. Consequently, there should only be relatively few tree levels. The results in Section 6.7 confirm this assumption. Hence, level-by-level parallelization should be efficient and provides ample of parallelism in most levels.

3.3.2 Cycle processing. Processing the cycles (Step 3) is the core of graphB+. Due to the vertex relabeling and the range information

stored in each tree edge, this step requires just 16 statements in our OpenMP implementation. To maximize the performance, the code only processes the non-tree edges in one direction. Based on the range information, it follows the appropriate edge from vertex to vertex until the cycle is complete. Along the way, it counts the number of negative edges and, ultimately, sets the sign of the non-tree edge such that the total number is even.

The cycle processing is parallelized over the vertices. The non-tree edges of each vertex are processed consecutively in the OpenMP code. As the number of non-tree edges per vertex and the cycle lengths vary, we use a dynamic schedule for load balancing.

GPUs require much higher degrees of parallelism than CPUs, so our CUDA implementation is parallelized not only across the vertices but also across the edges. In particular, we employ a hierarchical parallelization scheme where the many warps process the vertices in parallel and the 32 threads within each warp process the non-tree edges of the given vertex in parallel.

No synchronization is needed when processing cycles in parallel since the tree vertices and edges (i.e., the shared data) are only read. The non-tree edges are read and written but only by one thread each (i.e., they constitute thread-private data). Processing multiple non-tree edges of the same vertex in parallel, as is done by the warp threads in our GPU code, is also safe for the same reason.

4 RELATED WORK

No prior publications exist on parallelizing the balancing of fundamental cycles. Hence, this section discusses other related work not already covered in Sections 1 and 2.

The frustration index is one of several measures of balance that have been proposed to analyze real-world signed graphs containing conflicting observations. Computing it is a key operation in many fields of research, including physics [4, 7], economics [44], negative feedback loops in Boolean networks [38], and statistical mechanics [36], making graphB+ valuable beyond social network analysis.

Computing the frustration index of a signed graph is equivalent to finding a nearest ground (balanced) state in disordered systems. This is an NP-hard problem in general, but there exist scenarios that are solvable in polynomial time and for which exact large-scale solutions are possible. A survey [4] illustrates and reviews their applications to physics problems, especially Ising models and two-dimensional spin glasses. The frustration cloud approach [33] obviates the need for determining a balanced state with a minimum number of sentiment changes. Instead, it determines a set of nearest states with possibly varying numbers of sentiment changes. This redirects the focus from a single balanced state to a family of ground states. The computational complexity of these algorithms is bounded by a polynomial function of the size of the underlying graph. However, the upper bound is still prohibitive for a system of the size of the social networks we are targeting.

Wu and Chen proposed a branch-and-bound algorithm to balance signed graphs by editing edges and deleting vertices and demonstrated its efficiency over trivial and heuristic algorithms on inputs with up to $n = 40$ vertices [43]. In control multi-agent systems, Altafini analyzed the convergence to a balanced state in the decision-making process and presented an effective way to compute the average consensus for a network with up to 100 vertices [5]. Aref et al. developed three binary linear programming models to compute the frustration index quickly and exactly as the solution to a global optimization problem. They demonstrated the efficiency of their techniques for inputs with up to 15,000 edges [6]. Our work has the same goal, but our solution scales to more than three orders of magnitude larger signed graphs.

Our graphB+ algorithm requires a spanning tree as input. Computing spanning trees efficiently, especially in parallel, is an active research area. Early work targeted the theoretical PRAM model [10]. More recent work describes parallel multi-core CPU [29, 30, 41], distributed-memory CPU [22], and GPU [12, 31, 40] algorithms.

5 EXPERIMENTAL METHODOLOGY

The performance comparisons in this paper are based on the graphB+ runtime, excluding earlier and later computations such as the tree building and the Harary bipartitioning. We ran each experiment 5 times and report the best measured runtime. We only ran the Python code [39] once for each input because it is slow. On the graphs that can be processed with the Python code, we compared the results with our results to ensure that they agree.

The system we used is based on a 16-core 3.5 GHz AMD Ryzen Threadripper 2950X CPU. Hyperthreading is enabled, i.e., the 16 cores can simultaneously run 32 threads. Each core has a 32 kB L1 data cache, a 512 kB L2 cache, and all cores share an 8 MB L3 cache. The 48 GB main memory has a peak bandwidth of 87.4 GB/s. The operating system is Fedora 30. The system contains an NVIDIA Titan V GPU with 5120 processing elements distributed over 80 multiprocessors (SMs). Each SM has 96 kB of L1 data cache. The 80 SMs share a 4.5 MB L2 cache as well as 12 GB of global memory with a peak bandwidth of 652 GB/s.

The serial C++ CPU code was compiled with g++ 9.3.1 using the "-O3 -march=native" optimization flags and the OpenMP code additionally with the "-fopenmp" flag. We compiled the GPU code with nvcc 11.0 using the "-O3 -arch=sm_70" flags.

We used the 20 graphs from Table 1 as inputs. They are listed by size within each category. They were obtained from the Stanford Network Analysis Platform (SNAP) [25] and from Amazon [19, 27]. The table shows the name of each input, the number of vertices, edges, and fundamental cycles in the largest connected component, and the maximum and average degree as well as the original size in terms of ratings or reviews included across all vertices. Note that we only process the largest connected component with graphB+, which encompasses nearly the entire input as the small difference between the number of ratings and the number of edges shows. All tables and figures in the following sections use A* to refer to the Amazon datasets and S* to refer to the SNAP datasets.

6 RESULTS

In this section, we first compare the performance of our graphB+ implementation to that of the original Python code. Next, we study the throughput of our serial, OpenMP, and CUDA codes. Then, we analyze the dynamic memory usage. We also evaluate the OpenMP scaling and the CUDA runtime of individual algorithmic steps. Finally, we investigate some relevant graph properties.

	Largest Connected Component			Entire Input graph		
Amazon Ratings	**# vertices**	**# edges**	**# cycles**	**max degree**	**avg degree**	**# ratings**
Books	9,973,735	22,268,630	12,294,896	43,201	2.23	22,507,155
Electronics	4,523,296	7,734,582	3,211,287	18,244	1.71	7,824,482
Clothing, Shoes, and Jewlery	3,796,967	5,484,633	1,687,667	3,047	1.44	5,748,920
Movies and TV	2,236,744	4,573,784	2,337,041	11,906	2.04	4,607,047
CDs and Vinyl	1,959,693	3,684,143	1,724,451	5,755	1.88	3,749,004
Sports and Outdoors	2,147,848	3,075,419	927,572	6,016	1.43	3,268,695
Android App	1,373,018	2,631,009	1,257,992	25,368	1.92	2,638,172
Toys and Games	1,489,764	2,142,593	652,830	10,281	1.44	2,252,771
Automotive	950,831	1,239,450	288,620	2,738	1.30	1,373,768
Patio, Lawn, and Garden	735,815	939,679	203,865	3,180	1.28	993,490
Baby	559,040	892,231	333,192	3,648	1.60	915,446
Digital Music	525,522	702,584	177,063	1,953	1.34	836,006
Instant Video	433,702	572,834	139,133	12,633	1.32	583,993
Musical Instruments	355,507	457,140	101,634	3,523	1.29	500,176
Amazon Reviews	**# vertices**	**# edges**	**# cycles**	**max degree**	**avg degree**	**# reviews**
Digital Music core5	9,109	64,706	55,598	578	7.10	64,706
Instant Video core5	6,815	37,126	30,312	455	5.45	37,126
Musical Instruments core 5	2,329	10,261	7,933	163	4.41	10,621
SNAP Signed Networks	**# vertices**	**# edges**	**# cycles**	**max degree**	**avg degree**	**# signed edges**
soc-sign-epinions	119,130	704,267	585,138	3,558	5.91	841,372
soc-sign-Slashdot090221	82,140	500,481	418,342	2,548	6.09	549,202
wiki-Elec	7,539	112,058	104,520	1,079	14.86	114,040

Table 1: Pertinent information about the signed input graphs. The number of vertices, edges, and cycles reflect the number in the largest connected component of each dataset.

Graph	Serial	OpenMP	CUDA	Python
A*_Instruments_core5	0.73	0.47	0.18	114.2
A*_Music_core5	6.97	1.40	0.47	1039.0
A*_Video_core5	3.31	1.23	0.62	593.7
S*_wiki	12.30	2.19	1.13	1088.5
GEOMEAN	3.79	1.16	0.49	526.2

Table 2: Graph balancing runtime in seconds of our serial, OpenMP, and CUDA codes as well as the original Python code on four small input graphs for 1000 breadth-first-search (BFS) trees

6.1 Comparison to Original Python Code

This subsection compares the performance of our serial, OpenMP, and CUDA implementations of graphB+ to that of the original graph-balancing code written in Python [39]. Table 2 lists the sum of the runtimes over 1000 trees. Due to the long runtimes of the Python code, we only present results for a few small graphs. Fig. 7 shows the corresponding throughputs in millions of fundamental cycles balanced per second to visualize the performance. Note that the y-axis is logarithmic and that higher throughputs are better.

The results show that graphB+ is orders of magnitude faster, even when run serially. This highlights the effectiveness of the new algorithm we developed for discovering, traversing, and balancing fundamental cycles. The results also make it clear why the original code cannot be used on large graphs. Based on the geometric mean over these four inputs, our serial code is 140 times faster than the Python code and our GPU code is over 1000 times faster. The Python code takes almost 9 minutes to compute 1000 nearest balanced states compared to under 0.5 seconds for our CUDA code.

Our parallel code does not scale well on these small inputs. The geometric-mean speedup of the OpenMP version is under 3.3 and the speedup of the CUDA version is under 8 relative to serial graphB+. The reason for these low speedups is that the inputs, and therefore the runtimes, are so small that parallelization overheads dominate. Recall that we parallelize within and not across the spanning trees, meaning that the runtime of the parallelized operations is just 1/1000 of the times listed in Table 2. Moreover, on

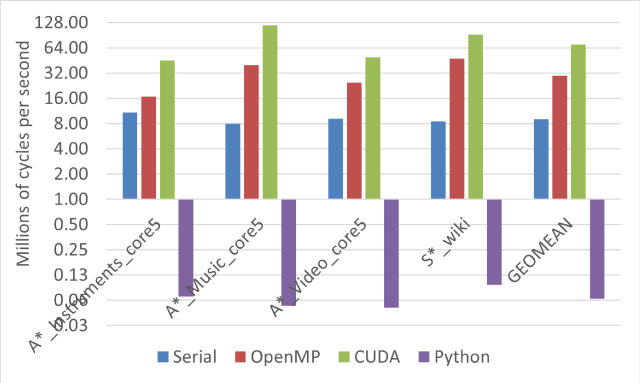

Figure 7: Throughput in millions of fundamental cycles balanced per second of our serial, OpenMP, and CUDA codes and the original Python code

Graph	Serial	OpenMP	CUDA
A*_Android	2812.7	256.1	281.3
A*_Automotive	406.0	54.7	16.0
A*_Baby	310.7	38.2	15.3
A*_Book	38775.0	3193.8	851.2
A*_Electronics	8327.4	768.2	255.0
A*_Games	983.8	111.1	55.1
A*_Garden	256.9	36.7	11.4
A*_Instruments	97.0	16.1	8.3
A*_Jewelry	2990.7	352.3	56.6
A*_Music	163.3	25.7	7.8
A*_Outdoors	1469.8	195.0	42.0
A*_TV	3447.9	342.6	87.4
A*_Video	309.2	53.8	117.9
A*_Vinyl	2302.3	238.6	49.0
S*_eopinion	220.5	22.7	11.9
S*_slashdot	122.7	11.0	6.8
GEOMEAN	881.9	103.2	40.8

Table 3: Graph balancing runtime in seconds of our serial, OpenMP, and CUDA codes on the larger input graphs for 1000 BFS trees

the GPU, there are many threads that end up with no work because there are not enough vertices and edges in these small graphs. Yet, the CUDA code delivers a geometric-mean throughput of 70 million fundamental cycles balanced per second. In contrast, the Python code is only able to balance 65,336 fundamental cycles per second.

6.2 Performance on Larger Graphs

This subsection presents the performance of our serial, OpenMP, and CUDA graphB+ implementations on the 16 larger inputs. Table 3 lists the sum of the runtimes over 1000 trees and Fig. 8 shows the corresponding throughputs in millions of fundamental cycles balanced per second. Again, the y-axis is logarithmic.

These results show that, even on the largest input with 10M vertices and 22M edges, it takes the GPU code less than 15 minutes to compute 1000 nearest balanced states, i.e., under 1 second per sample. Since the runtime is roughly proportional to the input size, our graphB+ implementation should be able to balance inputs that

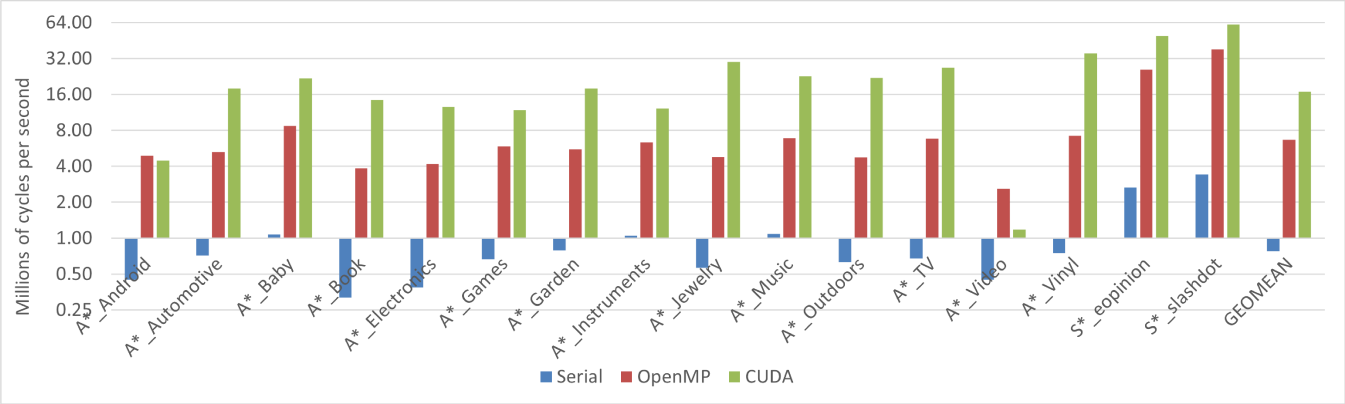

Figure 8: Throughput in millions of fundamental cycles balanced per second of our serial, OpenMP, and CUDA code on the larger graphs

are 10 times larger in a few seconds per sample, making it tractable to analyze graphs with 100s of millions of vertices and edges.

The geometric-mean throughput of the CUDA code on these larger graphs is 16.8 million fundamental cycles balanced per second. This is about four times lower than the throughput on the small inputs shown in Fig. 7. The primary reason for this drop is the large difference in average degree (cf. Table 1). The small graphs have between 3.4 and 13.9 fundamental cycles per tree edge whereas most of the large graphs have fewer than than one.

Fig. 9 shows the speedup of our OpenMP and CUDA codes over the serial code. Higher speedups are better. The GPU code is 2.6 to 53 times faster than the serial code (the geometric mean is 21.6) and up to 6.2 times faster than the OpenMP code (the geometric mean is 2.5). On two inputs, the CPU outperforms the GPU. The OpenMP code is 5.7 to 12.1 times faster on 16 cores than the serial code (the geometric mean is 8.5). These results highlight the benefit of parallelization and, in particular, of using an accelerator.

We correlated the runtime of our GPU code with the graph properties from Table 1 and found a strong linear correlation ($r > 0.9$) with the number of vertices, edges, and cycles and a particularly strong correlation ($r = 0.96$) with the maximum degree, meaning that the CUDA code takes longer to run for graphs that have a high maximum degree. Section 6.6 explains why.

6.3 OpenMP Scalability

This subsection investigates the strong scaling of our OpenMP code. Fig. 10 shows the speedups relative to the serial code (not relative to the OpenMP code running with one thread) for all 20 inputs. Unlike the previous charts, this chart lists the inputs sorted by the number of fundamental cycles to better visualize the trends. The results are once again obtained when processing 1000 trees.

For all investigated thread counts, the speedups tend to be higher for larger inputs. On our 16-core platform, the maximum speedups range from about 2 to 8 on the smaller inputs and from about 8 to 12 on the larger inputs. They are not higher due to the short runtimes of under one second per tree for all but the A*_Book input. Depending on the number of BFS-tree levels, the code executes between about 20 and 60 parallelized OpenMP regions during this fraction of a second. The runtimes per parallel code region are so short, especially for the smaller inputs, that the parallelization overhead (e.g., the forking and joining of the threads) substantially impacts performance. Hence, we expect the speedup to be significantly higher for larger inputs with 100s of millions or more vertices.

Fig. 10 illustrates that hyperthreading (using 32 threads on our 16-core machine) helps only little and hurts on some inputs, particularly on the smallest inputs. This is expected as hyperthreading tends to not be effective on data-parallel codes (which is why supercomputers like Frontera [14] have it disabled). Moreover, hyperthreading is often especially ineffective on irregular memory-bound codes like graphB+ because the extra thread contexts only provide additional computation resources but no additional memory bandwidth or cache capacity.

6.4 Dynamic Memory Usage

This subsection studies the amount of dynamically allocated memory in our CPU and GPU codes for storing the data needed by graphB+ as well as for the Harary bipartitioning and the vertex status computation. The memory usage is fixed and independent of the number of trees processed. Table 4 lists the megabytes allocated in the OpenMP and CUDA codes. For CUDA, we separately list the host (CPU) and device (GPU) amounts.

The results confirm that the memory usage is linear in the graph size, i.e., $O(n+m)$. Note that the CUDA host memory usage is about two thirds of that of the OpenMP code as some data only resides on the GPU. The CUDA device memory usage is 22% higher than the OpenMP memory usage due to two worklists that are not present in the OpenMP code. We use the two worklists to store alternating tree levels when computing the Harary bipartitions.

Based on these results, our CUDA code running on a GPU with 12 GB of memory should be able to process graphs with up to 150 million edges. Our OpenMP code running on a system with 128 GB of memory should support graphs with up to 2 billion edges.

6.5 Kernel Breakdown

This subsection investigates the fraction of the runtime spent in the various kernels of our CUDA code. For reference, we include results for computing the spanning trees and the Harary bipartitions, which are not part of graphB+. Fig. 11 shows the breakdown.

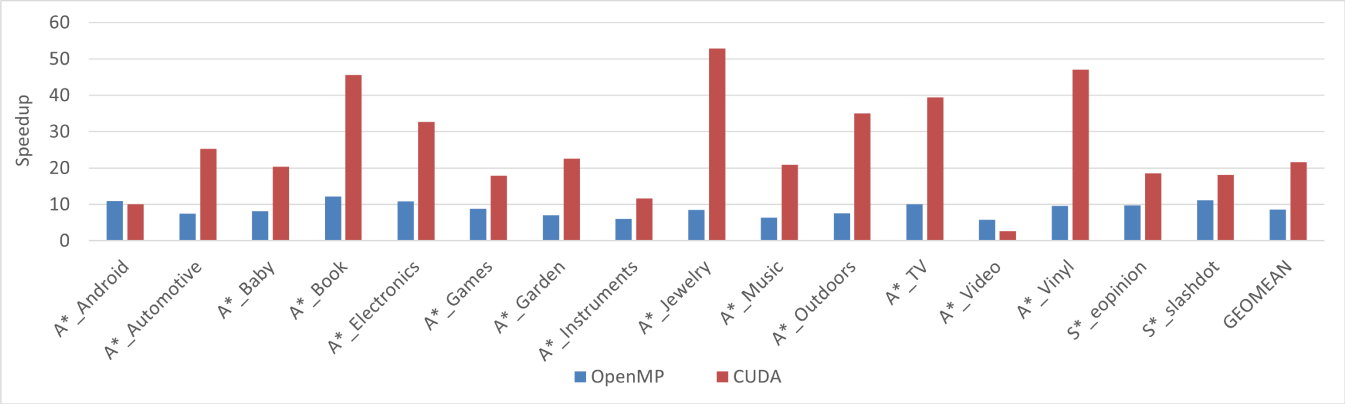

Figure 9: Speedup of the OpenMP and CUDA codes over the serial code on the larger graphs

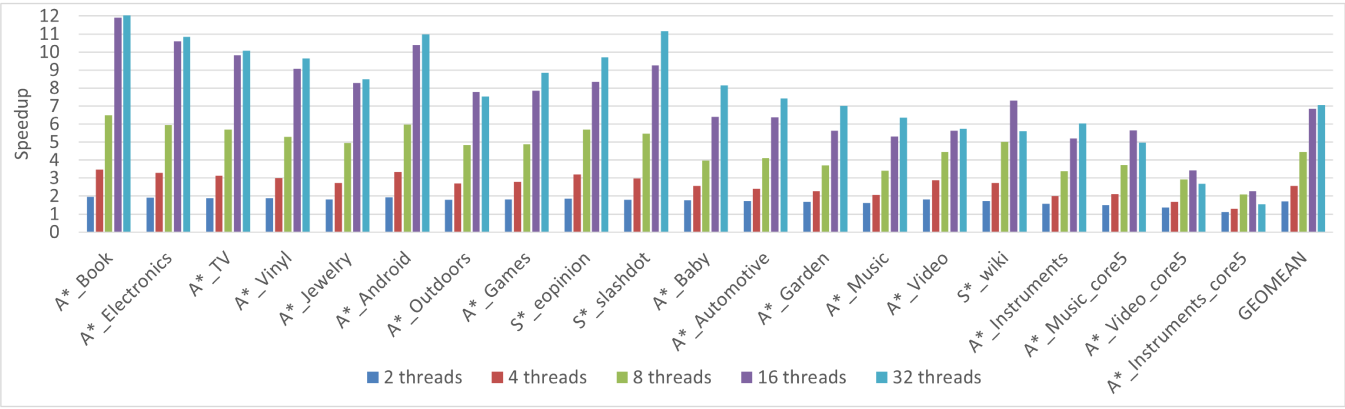

Figure 10: Speedup of the OpenMP code over the serial code for different thread counts (inputs sorted by cycle count)

The rightmost bar reflects the average over the 20 inputs. The bars are stacked, from bottom to top, according to the average runtime.

The general trends are quite similar across the different inputs, so we focus our discussion on the averages. In all cases, the fundamental cycle processing takes more time than any other operation. On average, it accounts for 64% of the overall runtime. Next is the labeling of the tree vertices and edges, which accounts for nearly 20% of the runtime. The Harary bipartitioning takes less than 10% and the spanning tree generation 6%. Hence, graphB+, which entails the tree labeling and the cycle processing, takes 5.5 times as long as computing the spanning tree and the bipartitions. In other words, even our optimized implementation takes several times as long as the rest of the operations needed to compute a metric such as the status of each vertex. This highlights the need for fast signed graph balancing, which is the premise of our work.

6.6 Fundamental Cycle Properties

This subsection studies the fundamental cycles in more detail. Table 5 lists the average length of the fundamental cycles as well as the average degree of the vertices on each cycle. The averages are based on 1000 BFS spanning trees.

The average cycle length is between 5.0 and 10.6, which is surprisingly short. In contrast, the average vertex degree encountered on a cycle is 147.7, which is surprisingly high for graphs with an average degree of just 3.3. Note, however, that these are power-law social network graphs with several high-degree vertices (cf. Table 1). Evidently, most fundamental cycles include at least one of these high-degree vertices. This makes sense in the context of BFS trees. Whenever such a vertex is reached during tree building, its high fan-out leads to a very shallow but wide tree. (The results of the next subsection corroborate this fact.) Hence, the fundamental cycle processing in graphB+ is primarily bottlenecked by determining which edge to follow.

6.7 Spanning Tree Properties

This subsection investigates the depth of the BFS spanning trees. Table 6 lists, for each input, the minimum and maximum depth of any of the 1000 trees and the average depth over all 1000 trees.

Every single random BFS spanning tree of these social networks is shallow as the largest depth over all graphs and trees is just 21 and the average depth is under 18 in all cases. As mentioned, this is a consequence of the inputs containing some high-degree vertices, as is common for social networks. Consequently, there are only few

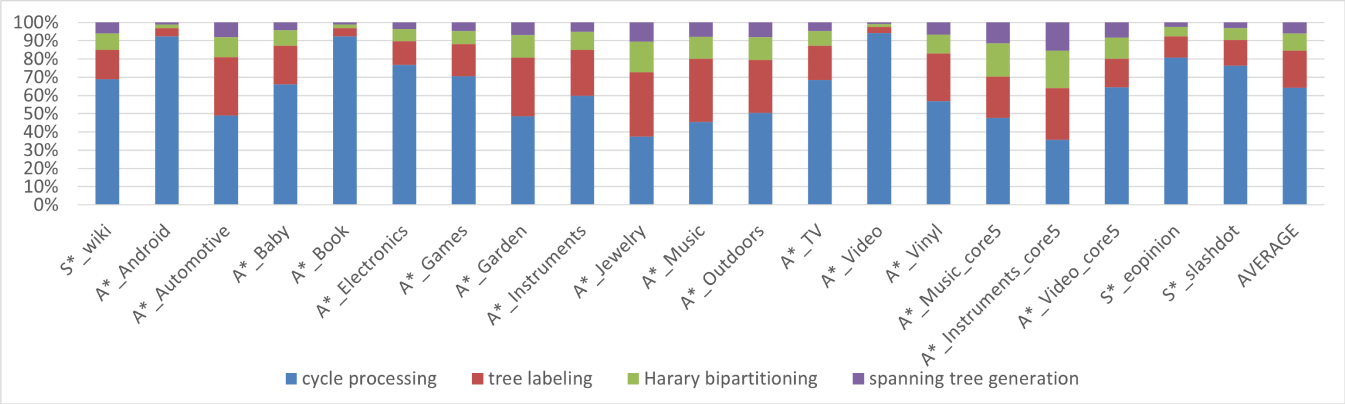

Figure 11: Relative GPU kernel runtime breakdown in percent (bipartitioning and tree generation are not part of graphB+)

Graph	OpenMP host(MB)	CUDA device(MB)	CUDA host(MB)
A*_Android	162.1	197.0	106.5
A*_Automotive	84.5	99.8	56
A*_Baby	57.5	69.0	37.9
A*_Book	1,328.2	1629.9	869.8
A*_Electronics	489.6	590.4	322.3
A*_Games	141.9	169.0	93.8
A*_Garden	64.5	76.0	42.7
A*_Instruments	362.9	36.9	20.7
A*_Instruments_core5	0.6	0.7	0.4
A*_Jewelry	362.9	432.1	239.8
A*_Music	47.5	56.3	31.5
A*_Music_core5	3.3	4.3	2.1
A*_Outdoors	204.0	242.7	134.8
A*_TV	277.8	339.1	182.2
A*_Video	38.9	46.0	25.8
A*_Video_core5	2.0	2.5	1.3
A*_Vinyl	228.0	276.7	149.8
S*_eopinion	36.1	47.1	23.8
S*_slashdot	26.1	33.4	16.8
S*_wiki	5.5	7.2	3.6

Table 4: Dynamic memory usage of our OpenMP and CUDA codes

Graph	Avg cycle length	Avg degree on cycles
A*_Android	7.15	432.01
A*_Automotive	10.63	76.37
A*_Baby	8.54	95.67
A*_Book	8.21	492.34
A*_Electronics	8.37	364.59
A*_Games	9.91	104.99
A*_Garden	10.19	79.25
A*_Instruments	10.15	66.03
A*_Instruments_core5	7.84	5.84
A*_Jewelry	10.60	96.32
A*_Music	8.90	64.34
A*_Music_core5	7.05	16.08
A*_Outdoors	9.85	108.77
A*_TV	7.09	238.59
A*_Video	8.40	351.73
A*_Video_core5	7.62	10.68
A*_Vinyl	8.11	151.57
S*_eopinion	5.21	103.25
S*_slashdot	5.55	66.33
S*_wiki	5.03	29.01
AVERAGE	8.22	147.69

Table 5: Average length of the fundamental cycles and average degree of the vertices on the cycles based on 1000 spanning trees

BFS levels and most levels contain many vertices and edges. This makes our level-by-level parallelization strategy effective.

The results in this and the previous subsection provide empirical evidence that the average cycle length is linear in the average tree depth and that the expected average tree depth is $O(log(n))$ as assumed in Section 3.1.

7 SUMMARY AND FUTURE WORK

This paper describes an efficient algorithm called *graphB+* for balancing the signs on the edges of signed graphs. It runs in expected $O(m \times log(n) \times t)$ time and requires $O(n + m)$ storage, where m is the number of edges, n the number of vertices, and t the average spanning-tree degree of the vertices on each cycle (cf. Section 3.1). graphB+ is based on a new vertex and edge labeling technique for rapidly determining and balancing all fundamental cycles of a graph. This is a key step in social network analysis algorithms.

We parallelized graphB+ using both OpenMP and CUDA. The source code is freely available under the 3-clause BSD licence [3]. The GPU code detects, traverses, and balances tens of millions of fundamental cycles per second and easily scales to graphs with tens of millions of vertices and edges on a single device. Even when run serially, our graphB+ implementation is orders of magnitude faster than the preexisting Python code, which demonstrates the benefit of our algorithm. The parallelization boosts the performance by another factor of 20 on average. Good speedup only manifests itself on larger graphs because inputs with fewer than about 10,000 vertices are too small to fully load the GPU.

Graph	Min depth	Max depth	Avg depth
A*_Android	10	13	12.2
A*_Automotive	15	19	17.3
A*_Baby	11	15	12.9
A*_Book	15	19	17.1
A*_Electronics	11	12	11.7
A*_Games	15	18	16.8
A*_Garden	12	15	13.6
A*_Instruments	14	21	17.2
A*_Instruments_core5	5	6	5.7
A*_Jewelry	14	16	15.7
A*_Music	14	18	15.8
A*_Music_core5	5	7	6.0
A*_Outdoors	14	17	15.2
A*_TV	12	15	13.9
A*_Video	11	15	12.9
A*_Video_core5	5	7	5.8
A*_Vinyl	13	15	13.7
S*_eopinion	8	11	9.5
S*_slashdot	7	9	7.9
S*_wiki	4	5	4.9
AVERAGE	10.8	13.7	12.3

Table 6: Minimum, maximum, and average tree depth of 1000 trees

Despite the high performance of graphB+, graph balancing still accounts for 84% of the overall runtime in a social network analysis application that computes the status of all vertices. This illustrates the need for fast signed graph balancing algorithms.

We found the fundamental cycles to be surprisingly short, under 11 vertices on average, and the average vertex degree on the cycles to be surprisingly large, about 150 on average. This information may prove useful to further enhance the performance of graphB+.

As expected for power-law graphs like signed social networks, the employed level-by-level parallelization strategy is effective because the BFS spanning trees used by graphB+ tend to only have a few levels, under 18 on average for our twenty inputs.

In future work, we are planning to apply graphB+ in the field for actual studies of attitudinal social networks. We also intend to analyze the choice of spanning trees and the sampling frequency. Finally, we want to quantify how various graph characteristics, such as sparsity and the percentage of negative signs, affect the algorithm's performance.

ACKNOWLEDGMENTS

This work has been supported in part by the National Science Foundation under Award Number 1955367, by the Department of Energy, National Nuclear Security Administration under Award Number DE-NA0003969, and by a hardware donation from NVIDIA Corporation.

REFERENCES

[1] Robert P. Abelson and Milton J. Rosenberg. 1958. Symbolic psycho-logic: A model of attitudinal cognition. *Behavioral Science* 3, 1 (1958), 1–13.

[2] S. Al-Yazidi, J. Berri, M. Al-Qurishi, and M. Al-Alrubaian. 2020. Measuring Reputation and Influence in Online Social Networks: A Systematic Literature Review. *IEEE Access* 8 (2020), 105824–105851. https://doi.org/10.1109/ACCESS.2020.2999033

[3] Ghadeer Alabandi and Martin Burtscher. 2021. graphB+ code. https://cs.txstate.edu/~burtscher/research/graphBplus/.

[4] M.J. Alava, P.M. Duxbury, C.F. Moukarzel, and H. Rieger. 2001. Exact combinatorial algorithms: Ground states of disordered systems. In *In: C. Domb and J.L. Lebowitz, eds., Phase Transitions and Critical Phenomena, Vol. 18.* Academic Press, San Diego.

[5] C. Altafini. 2019. A dynamical approach to privacy preserving average consensus. In *2019 IEEE 58th Conference on Decision and Control (CDC)*. 4501–4506. https://doi.org/10.1109/CDC40024.2019.9029712

[6] Samin Aref, Andrew J. Mason, and Mark C. Wilson. 2016. An exact method for computing the frustration index in signed networks using binary programming. *CoRR* abs/1611.09030 (2016). arXiv:1611.09030 http://arxiv.org/abs/1611.09030

[7] F. Barahona. 1982. On the computational complexity of Ising spin glass models. *J. Phys. A: Math. Gen.* 15 (1982), 3241–3253.

[8] Aydın Buluç, John Gilbert, and Viral B Shah. 2011. Implementing sparse matrices for graph algorithms. In *Graph Algorithms in the Language of Linear Algebra*. SIAM, 287–313.

[9] D. Cartwright and F. Harary. 1956. Structural balance: a generalization of Heider's theory. *Psychological Rev.* 63 (1956), 277–293.

[10] Ka Wong Chong, Yijie Han, Yoshihide Igarashi, and Tak Wah Lam. 2003. Improving the efficiency of parallel minimum spanning tree algorithms. *Discrete Applied Mathematics* 126, 1 (2003), 33–54. https://doi.org/10.1016/S0166-218X(02)00560-7 5th Annual International Computing and combinatorics Conference.

[11] Tim Davis, Yifan Hu, and Scott Kolodziej. 2020. SuiteSparse Matrix Collection. Website: https://sparse.tamu.edu/.

[12] Jucele França de Alencar Vasconcellos, Edson Norberto Cáceres, Henrique Mongelli, and Siang Wun Song. 2017. A Parallel Algorithm for Minimum Spanning Tree on GPU. In *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. 67–72. https://doi.org/10.1109/SBAC-PADW.2017.20

[13] Reinhard Diestel. 2010. *Graph theory*. Graduate Texts in Mathematics, Vol. 173. Springer, Heidelberg.

[14] Frontera. 2021. https://frontera-portal.tacc.utexas.edu/user-guide/system/.

[15] Kiran Garimella, Gianmarco De Francisci Morales, Aristides Gionis, and Michael Mathioudakis. 2017. Reducing Controversy by Connecting Opposing Views. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining* (Cambridge, United Kingdom) *(WSDM '17)*. 81–90.

[16] Chris Godsil and Gordon Royle. 2001. *Algebraic graph theory*. Graduate Texts in Mathematics, Vol. 207. Springer-Verlag, New York. xx+439 pages.

[17] F. Harary. 1953. On the notion of balance of a signed graph. *Michigan Math. J.* 2(2) (1953), 143–146.

[18] F. Harary. 1959. On the measurement of structural balance. *Behavioral Sci.* 4 (1959), 316–323.

[19] Ruining He and Julian McAuley. 2016. Ups and Downs: Modeling the Visual Evolution of Fashion Trends with One-Class Collaborative Filtering. In *Proceedings of the 25th International Conference on WWW*. 507–517.

[20] F. Heider. 1946. Attitudes and cognitive organization. *J. Psychology* 21 (1946), 107–112.

[21] M.A. Javed, M.S. Younis, S. Latif, J. Qadir, and A. Baig. 2018. Community detection in networks: A multidisciplinary review. *Journal of Network and Computer Applications* 108 (2018).

[22] Yuede Ji, Hang Liu, and H. Howie Huang. 2018. Parallel Identification of Strongly Connected Components with Spanning Trees. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas) *(SC '18)*. IEEE Press, Article 58, 12 pages. https://doi.org/10.1109/SC.2018.00061

[23] Srijan Kumar, William L. Hamilton, Jure Leskovec, and Dan Jurafsky. 2018. Community Interaction and Conflict on the Web. In *Proceedings of the WWW* (Lyon, France). 933–943.

[24] Jure Leskovec. 2015. New Directions in Recommender Systems. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining* (Shanghai, China) *(WSDM '15)*. ACM, 3–4.

[25] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[26] K W Li, D M Kilgour, and K W Hipel. 2005. Status quo analysis in the graph model for conflict resolution. *Journal of the Operational Research Society* 56, 6 (2005), 699–707. https://doi.org/10.1057/palgrave.jors.2601870 arXiv:https://doi.org/10.1057/palgrave.jors.2601870

[27] Julian McAuley. 2015. Amazon product data. https://jmcauley.ucsd.edu/data/amazon/.

[28] Julian McAuley, Christopher Targett, Qinfeng Shi, and Anton van den Hengel. 2015. Image-Based Recommendations on Styles and Substitutes. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Santiago, Chile) *(SIGIR '15)*. Association for Computing Machinery, New York, NY, USA, 43–52. https://doi.org/10.1145/2766462.2767755

[29] Badri Munier, Muhammad Aleem, Muhammad Arshad Islam, Muhammad Azhar Iqbal, and Waqar Mehmood. 2017. A Fast Implementation of Minimum Spanning Tree Method and Applying it to Kruskal's and Prim's Algorithms. *Sukkur IBA Journal of Computing and Mathematical Sciences* 1, 1 (2017), 58–66. https://doi.org/10.30537/sjcms.v1i1.8

[30] Suryanarayana Murthy Durbhakula. 2020. Parallel Minimum Spanning Tree Algorithms and Evaluation. *arXiv e-prints*, Article arXiv:2005.06913 (May 2020), arXiv:2005.06913 pages. arXiv:2005.06913 [cs.DC]

[31] Wen-Bao Qiao and Jean-Charles Créput. 2019. GPU implementation of Borůvka's algorithm to Euclidean minimum spanning tree based on Elias method. *Applied Soft Computing* 76 (2019), 105–120. https://doi.org/10.1016/j.asoc.2018.10.046

[32] Michael Röder, Andreas Both, and Alexander Hinneburg. 2015. Exploring the space of topic coherence measures. In *Proceedings of the eighth ACM international conference on Web search and data mining*. 399–408.

[33] Lucas Rusnak and Jelena Tešić. 2020. Characterizing Attitudinal Network Graphs through Frustration Cloud. https://arxiv.org/abs/2009.07776. arXiv:2009.07776 [cs.SI]

[34] Stuart Russell and Peter Norvig. 2009. *Artificial Intelligence: A Modern Approach* (3rd ed.). Prentice Hall Press, USA.

[35] Farah Saab, Imad H. Elhajj, Ayman Kayssi, and Ali Chehab. 2019. Modelling Cognitive Bias in Crowdsourcing Systems. *Cognitive Systems Research* 58 (2019), 1 – 18. https://doi.org/10.1016/j.cogsys.2019.04.004

[36] James P. Sethna. 2006. *Statistical Mechanics: Entropy, Order Parameters, and Complexity*. Master Ser. in Physics, Vol. 14. Oxford Univ. Press, Oxford.

[37] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2018. Graph processing on GPUs: A survey. *ACM Computing Surveys (CSUR)* 50, 6 (2018), 1–35.

[38] Eduardo Sontag, Alan Veliz-Cuba, Reinhard Laubenbacher, and Abdul Salam Jarrah. 2008. The Effect of Negative Feedback Loops on the Dynamics of Boolean Networks. *Biophysical Journal* 95, 2 (2008), 518 – 526. https://doi.org/10.1529/biophysj.107.125021

[39] Jelena Tešić, Joshua Mitchell, Eric Hull, and Lucas Rusnak. 2020. graphB: Python software package for graph analysis. https://github.com/DataLab12/graphB.

[40] Vibhav Vineet, Pawan Harish, Suryakant Patidar, and P. J. Narayanan. 2009. Fast Minimum Spanning Tree for Large Graphs on the GPU. In *Proceedings of the Conference on High Performance Graphics 2009* (New Orleans, Louisiana) *(HPG '09)*. Association for Computing Machinery, New York, NY, USA, 167–171. https://doi.org/10.1145/1572769.1572796

[41] Yiqiu Wang, Shangdi Yu, Yan Gu, and Julian Shun. 2021. Fast Parallel Algorithms for Euclidean Minimum Spanning Tree and Hierarchical Spatial Clustering. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD/PODS '21)*. Association for Computing Machinery, New York, NY, USA, 1982–1995. https://doi.org/10.1145/3448016.3457296

[42] Stanley Wasserman and Katherine Faust. 1994. *Social network analysis: Methods and applications*. Vol. 8. Cambridge university press.

[43] Bang Ye Wu and Jia-Fen Chen. 2013. Balancing a Complete Signed Graph by Editing Edges and Deleting Nodes. In *Advances in Intelligent Systems and Applications - Volume 1*, Ruay-Shiung Chang, Lakhmi C. Jain, and Sheng-Lung Peng (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 79–88.

[44] Takeo Yoshikawa, Takashi Iino, and Hiroshi Iyetomi. 2011. Market Structure as a Network with Positively and Negatively Weighted Links. In *Intelligent Decision Technologies*, Junzo Watada, Gloria Phillips-Wren, Lakhmi C. Jain, and Robert J. Howlett (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 511–518.

[45] J. Zhou, L. Li, A. Zeng, Y. Fan, and Z. Di. 2018. Random walk on signed networks. *Physica A: Statistical Mechanics and its Applications* 508 (2018), 558–566.