It Pays to Be Lazy: Reusing Force Approximations to Compute Better Graph Layouts Faster

Robert Gove
Two Six Labs, robert.gove@twosixlabs.com

***Abstract*—N-body simulations are common in applications ranging from physics simulations to computing graph layouts. The simulations are slow, but tree-based approximation algorithms like Barnes-Hut or the Fast Multipole Method dramatically improve performance. This paper proposes two new update schedules, uniform and dynamic, to make this type of approximation algorithm even faster by updating the approximation less often. An evaluation of these new schedules on computing graph layouts finds that the schedules typically decrease the running time by 9% to 18% for Barnes-Hut and 88% to 92% for the Fast Multipole Method. An experiment using 4 layout quality metrics on 50 graphs shows that the uniform schedule has similar or better graph layout quality compared to the standard Barnes-Hut or Fast Multipole Method algorithms.**

I. Introduction

Spring-electric algorithms are considered to be conceptually simple methods for computing graph layouts [1]–[5] and they have enjoyed widespread implementation. However, the brute force algorithm requires $O(|V|^2)$ time to compute repulsive forces, where $|V|$ is the number of vertices in a graph $G = (V, E)$. Tree-based approximation methods—e.g. Barnes-Hut (BH), the Fast Multipole Method (FMM), and the Well-Separated Pair Decomposition (WSPD)—reduce this running time to $O(|V| \log |V|)$. Many spring-electric algorithms employ these techniques to improve performance [4]–[16], so improving these techniques' speed can have a wide impact. Reducing the amount of computation can reduce energy consumption on battery powered devices, reduce interruptions to analysts' flow of thought and attention [17], and accelerate the visual analytics sensemaking process.

These approximation algorithms create tree-based data structures from the vertex positions, and the algorithms then use the trees to approximate repulsive forces. Because the spring-electric algorithm iteratively updates vertex positions, the approximation methods reconstruct the tree after each iteration using the new vertex positions. Tree construction runs in $O(|V| \log |V|)$ time, and therefore can be computationally costly, but it is unknown whether it is necessary to construct a new tree after every iteration. Many spring-electric algorithm implementations include a "cooling" parameter that reduces the change in vertex position over time, indicating that calculating new trees provides diminishing improvements to accuracy after each iteration of the algorithm.

This paper presents an evaluation[1] of three alternative schedules for updating tree-based approximations less frequently: Logarithmic, uniform, and dynamic. The evaluation compares these schedules to the standard schedule of reconstructing the tree after every iteration. This paper shows that using a logarithmic, uniform, or dynamic update schedule achieves significantly faster performance compared to the standard update schedule. In addition, the uniform schedule achieves the same or better graph layout quality as the standard schedule.

This paper makes the following contributions: (1) A new dynamic algorithm for deciding when to reconstruct trees used in tree-based approximations such as Barnes-Hut, the Fast Multipole Method, or the Well-Separated Pair Decomposition; (2) a new schedule for reconstructing trees at uniform frequency; (3) a reformulation of the angular resolution (dev) readability metric to make it yield a value in $[0, 1]$; and (4) an evaluation of the logarithmic, uniform, and dynamic update schedules compared to the standard update schedule showing that the new uniform schedule outperforms the other methods.

[1]The materials to reproduce the analysis are available at https://osf.io/re7nx/

II. Background

Spring-electric algorithms belong to the family of force-directed graph layout algorithms. Spring-electric algorithms cast the graph layout problem as an iterative physical simulation, where the algorithm models the graph's vertices similarly to charged particles that repel each other, and it models the graph's edges similarly to springs that define an ideal distance between vertices. This paper is concerned with improving the runtime of the repulsive force calculation.

The *Barnes-Hut* (BH) approximation builds a quadtree of vertex positions, and then considers distant groups of vertices as a single large vertex (see Barnes and Hut [18] and Quigley and Eades [19] for more details). This process of calculating the quadtree runs in $O(|V| \log |V|)$ time and reduces the force calculations to $O(|V| \log |V|)$.

The *Fast Multipole Method* (FMM), like Barnes-Hut, first builds a spatial tree based on the vertex positions in $O(|V| \log |V|)$ time, but then it aggregates nodes in the tree in order to calculate repulsive forces in $O(|V|)$ time [20]–[22].

Recently, Lipp et al [4], [5] proposed using the *Well-Separated Pair Decomposition* [23] (WSPD)—another tree-based approximation algorithm—to compute repulsive forces for graph layout algorithms. Both tree construction and repulsive force calculation run in $O(|V| \log |V|)$ time.

For all three of the above tree-based approximation methods, the graph layout algorithm must reconstruct the tree after each iteration because the vertex positions have changed.

However, is it necessary to calculate a new tree after every iteration? Lipp et al [4], [5] experimented with updating the WSPD whenever $\lfloor 5\log(i) \rfloor$ changes (where i is the current iteration number), instead of after every iteration. They found that this can decrease the number of edge crossings [5] compared to the standard update schedule (i.e. reconstructing the tree after every iteration), but they did not compare the running time of the $\lfloor 5\log(i) \rfloor$ method to the standard method, nor did they test other update schedules such as other multiples of $\log(i)$ or a uniform update schedule. They also suggested using a dynamic algorithm to determine when to update the tree [5], but they did not define an algorithm to accomplish this or evaluate this idea. Furthermore, they did not apply this to the Barnes-Hut approximation or Fast Multipole Method, so it is not clear if it will work with other tree-based approximations, or whether their $\lfloor 5\log(i) \rfloor$ update criteria is optimal.

III. Schedules for Updating Approximations

There are many different methods to determine when to update trees used in tree-based approximation methods such as the Barnes-Hut (BH) approximation, the Fast Multipole Method (FMM), or the Well-Separated Pair Decomposition (WSPD). The standard versions of these algorithms construct a new tree after every iteration, but the tree could be reconstructed less often. This could be determined by an algorithm that defines some sort of schedule of when to update the approximation by reconstructing the tree. In this paper, a *schedule* is a function that returns a boolean value indicating whether or not the tree should be reconstructed. This paper explores three alternative schedules to the standard schedule: logarithmic, uniform, and dynamic.

A. Logarithmic Schedule

Many spring-electric algorithm implementations include a "cooling" parameter that reduces the change in vertex position over time, indicating that constructing new trees may provide diminishing improvements to accuracy after each iteration of the algorithm. In addition, most vertices tend to converge to a final position. This motivates a schedule that constructs a new tree with decreasing frequency at later iterations of the force-directed layout. Lipp et al. [4], [5] proposed a logarithmic schedule where a new tree is constructed if $\lfloor 5\log(i) \rfloor$ changes, where i is the current iteration number. However, they did not experiment with using scalars other than 5, and they did not specify the logarithmic base. Note that any two logarithmic functions with different bases differ by only a constant (i.e. $\log_b(x) = \log_a(x)/\log_a(b)$), so it suffices to use a base 10 logarithm and vary the scalar multiple. For this reason, this paper explores the family of schedules defined by $\lfloor k\log(i) \rfloor$ where k is an integer in $[1, 10]$. For 300 iterations of a graph layout algorithm, this corresponds to constructing a new tree 7, 13, 18, 22, 26, 31, 34, 38, 42, or 45 times for k ranging from 1 to 10.

B. Uniform Schedule

Even though vertex velocity tends to decrease over time, velocity does change at every iteration of a force-directed layout algorithm, and the exact amount of change can be difficult to predict. For this reason, it may be desirable to construct a new tree at uniform intervals. This paper proposes the following uniform update schedule: let u_k be the number of trees constructed using the logarithmic schedule for integer k. Then, for a layout that has n iterations, construct a new tree every n/u_k iterations. For a given value of k, this results in the same number of updates for both the logarithmic and the uniform schedules, and supports direct comparisons between them. The difference between the schedules is that, for a fixed value of k, the uniform schedule has the same number of iterations between each tree construction, whereas the logarithmic schedule constructs more trees at the beginning of the layout and fewer at the end.

C. Dynamic Schedule

During a force-directed layout, vertex velocity may temporarily decrease as the layout enters a local minimum before increasing as the layout escapes the minimum. This, combined with the "cooling" parameter described above, motivates a dynamic approach to deciding when to construct a new tree: if vertex positions are changing rapidly, then a new tree can improve accuracy, but if vertex positions are changing slowly, then constructing a new tree may be computationally costly with little improvement in accuracy. Therefore we would like to construct a new tree only if the old tree is out of date.

Algorithm 1 shows a dynamic algorithm that performs a check to decide whether to construct a new tree or use the old tree to calculate repulsive forces. This check keeps a running sum of the velocities (displacement) of all vertices since the last time a tree was constructed. If this running sum exceeds the previous sum, the new sum is assigned to the previous sum, the running sum is reset to 0, and a new tree is calculated before computing repulsive forces; otherwise, the algorithm decides the old tree is accurate enough and uses it to compute forces on the vertices. In practice, this algorithm constructs new a tree about 10–15 times out of 300 iterations.

Algorithm 1 Dynamic schedule. Initially, $currSum = prevSum = 0$. u_{v_x} and u_{v_y} are u's x, y velocities

```
for each vertex u do
    currSum ← currSum + |u_{v_x}| + |u_{v_y}|
if tree is null or currSum ≥ prevSum then
    prevSum ← currSum
    currSum ← 0
    tree ← newTree()
computeForces()
```

This dynamic algorithm has the benefit of generalizing to any approximation method or type of tree used. For example, an alternative dynamic algorithm might operate on the quadtrees in the Barnes-Hut approximation and check the number of vertices that are no longer in their original quadtree cells. However, such a dynamic algorithm would be restricted to approximation methods that depend on quadtrees, such as the Barnes-Hut approximation, and it would not generalize to

other tree-based approximation methods. Another downside to this alternative approach is that if only a few vertices moved out of their cells, but they moved a large distance, this may not exceed the threshold to construct a new quadtree. On the other hand, the dynamic method in Algorithm 1 can determine that a new tree should be constructed in this case where only a few vertices have moved but they moved a large distance.

Early experiments with this algorithm tested small coefficients for updating $currSum$ such as

$$currSum \leftarrow c * currSum + |u_{v_x}| + |u_{v_y}|$$

for $c = 1.01$ or $c = 0.99$. However, even such small coefficients resulted in constructing a new tree way more often than necessary (as in 1.01) or not enough to be useful (as in 0.99). Therefore the algorithm uses a coefficient of 1, which seems to yield good results.

IV. Exploratory Analysis

In order to develop hypotheses to test, this section presents an exploratory analysis of the update schedules. All data collection and analysis was conducted in NodeJS version 9.4.0 on a 2015-model MacBook Pro with a 3.1 GHz Intel Core i7 processor and 16 GB of RAM.

This exploratory analysis uses 50 graphs with 1000 vertices or fewer randomly selected from the KONECT [24] and SuiteSparse [25] graph collections.

This evaluation uses the D3.js framework [9] to compare the four update schedules (logarithmic, uniform, dynamic, and standard). The evaluation uses D3.js's default settings, and adds a central gravitational force with a strength of 0.001. The experiment uses D3.js's default stopping criteria, which is when the "cooling" parameter becomes sufficiently small; by default, this occurs after 300 iterations.

D3's default force-directed algorithm uses the Barnes-Hut (BH) approximation, which has been modified for this evaluation to support the new update schedules. This evaluation also uses a second algorithm, which is a modified version of D3's force-directed algorithm but uses a publicly available implementation of the Fast Multiple Method[2] (FMM). This second force-directed algorithm also supports all four update schedules. Although other tree-based approximation methods are available, they are implemented in other programming languages, and therefore cannot be directly compared to the JavaScript implementations. For this reason, this analysis only uses the aforementioned JavaScript versions of the BH and FMM approximations in order to minimize threats to validity.

This evaluation uses 10 versions each of the logarithmic and uniform update schedules parameterized with k from 1 to 10 as described in Section III-A and Section III-B.

D3.js initializes vertices in a disc-like phyllotaxis arrangement, where vertices at the beginning of the vertex array are at the center and vertices at the end of the vertex array are at the periphery. To minimize any possible effects of the initial positions on the experiment results, this experiment randomly shuffles the vertex array before calculating initial positions and running the spring-electric layout algorithm. This is done 20 times for each graph for each pair of update schedule and approximation method ($4 \times 2 = 8$ schedule-approximation pairs). The experiment then records the median runtime and median readability metrics for each graph and algorithm combination (this experiment uses the median instead of the arithmetic mean to avoid the results being skewed by outliers).

This evaluation uses the edge crossing, edge crossing angle, angular resolution (min), and angular resolution (dev) graph layout readability metrics implemented in greadability.js[3], which are defined below. Other readability metrics exist, such as stress or standard deviation of edge length, but this evaluation avoids these readability metrics because they have known issues [10], [26], [27]. Namely, non-uniform edge lengths are often necessary to achieve good layouts for real-world graphs [27], [28]; preserving shortest-path distances, as measured by stress, may not be ideal for producing good layouts [27]; stress is not defined on graphs with more than one component, which often occur in real-world data; and two layouts that convey a graph's structure equally can have different stress values [26]. In addition, stress and standard deviation of edge length do not have normalized versions that support accurate comparisons between different graphs.

Edge crossings, denoted $\aleph_c$, measures the number of edges that cross, or intersect, in the layout. The metric scales the number of edge crossings, c, by an approximate upper bound so that $\aleph_c \in [0, 1]$.

$$\aleph_c = 1 - c \Big/ \left(\frac{|E|(|E|-1)}{2} - \frac{1}{2} \sum_{u \in V} \deg(u)(\deg(u)-1) \right)$$

If the denominator is 0, then $\aleph_c = 1$.

Edge crossing angle, denoted $\aleph_{ca}$, measures the average deviation of each edge's crossing angle from the ideal angle ϑ of 70 degrees.

$$\aleph_{ca} = 1 - \frac{\sum_{e \in E} \sum_{e' \in c(e)} |\vartheta - \theta_{e,e'}|}{c\vartheta}$$

where $c(e)$ is the set of edges that intersect e, and $\theta_{e,e'}$ is the acute angle of the two intersecting edges. If $c\vartheta = 0$, then $\aleph_c = 1$.

Angular resolution (min) is the average deviation of incident edge angles from the ideal minimum angle for each vertex u.

$$\aleph_{rm} = 1 - \frac{1}{|V|} \sum_{u \in V} \frac{|\vartheta_u - \theta_{u_{min}}|}{\vartheta_u}$$

Here, $\vartheta_u = 360/d(u)$, the degrees between each incident edge if the angles were uniform, and $\theta_{u_{min}}$ is the smallest measured angle between edges incident on u.

Angular resolution (dev) is the average deviation of angles between incident edges on each vertex u. This paper presents a new formulation of the angular resolution (dev) metric described by Dunne *et al* [29]. Dunne *et al*'s metric can

[2]https://github.com/davidson16807/fast-multipole-method

[3]https://github.com/rpgove/greadability

produce negative numbers, but the version presented here produces a value in $[0, 1]$.

$$\aleph_{rd} = 1 - \frac{1}{|V|} \sum_{u \in V,\ d(u)>1} \left(\frac{1}{2d(u) - 2} \sum_{i}^{d(u)} \frac{|\vartheta_u - \theta_{i,(i+1)}|}{\vartheta_u} \right)$$

Here, $\theta_{i,(i+1)}$ is the acute angle between adjacent edges i and $i+1$ that are incident on vertex u, modulo $d(u)$, the degree of u. To see that $\aleph_{rd} \in [0, 1]$, consider that the maximum deviation on a vertex u will occur when $\theta_{i,(i+1)} \approx 0$ for all incident edges except one where $\theta_{i,(i+1)} \approx 360$. For the central vertex in a star graph with $|V|$ vertices, the sum of the deviations of its incident edges would be

$$\sum_{i}^{d(u)} \frac{|\vartheta_u - \theta_{i,(i+1)}|}{\vartheta_u} = \sum_{i}^{d(u)} \frac{|360/d(u) - \theta_{i,(i+1)}|}{360/d(u)}$$

Since $\theta_{i,(i+1)} \approx 0$ for all but one pair of incident edges, we then have

$$\begin{aligned} \sum_{i}^{d(u)} \frac{|360/d(u) - \theta_{i,(i+1)}|}{360/d(u)} &\approx (d(u) - 1) + \frac{|360/d(u) - 360|}{360/d(u)} \\ &= (d(u) - 1) + (d(u) - 1) \\ &= 2d(u) - 2 \end{aligned}$$

This is the upper bound for the deviation of a vertex u, so we must divide the deviation of each vertex by this quantity. In contrast, the smallest value would occur in a graph where every pair of incident edges had the ideal angle, which would make the deviation 0. Therefore, $\aleph_{rd}$ is in the range $[0, 1]$.

See Dunne *et al* [29] for a more detailed discussion of these and other readability metrics.

A. Hypothesis Generation

Figure 1 shows the median runtime and readability metrics across all 50 graphs in the exploratory dataset. The runtime and readability metrics were calculated for the Barnes-Hut (BH) and Fast Multipole Method (FMM) approximation algorithms using each update schedule (standard, dynamic, logarithmic for k from 1 to 10, and uniform for k from 1 to 10). This gives us some insight into how each approximation algorithm performs for the different update schedules, allowing us to develop hypotheses to formally test.

In the exploratory data, the standard BH and FMM methods are slower than all of the dynamic, log, and uniform update schedules. The dynamic algorithm had about the same number of updates as $k = 2$ (median is 11 and 14 for BH and FMM respectively), and they also have about the same runtime. This leads to **Hypothesis 1**: Using a dynamic or fixed update schedule (e.g. logarithmic or uniform) is faster than the standard update schedule for both the BH and FMM algorithms.

In order to test Hypothesis 1 experimentally, we must choose a value of k to use for the logarithmic and uniform update schedules. Ideally, k will be small in order to minimize the runtime, but k should also be large enough to produce good quality layouts. For these reasons, let us choose $k = 4$ (i.e. constructing a new tree 22 times out of 300 iterations). Although the runtime is not as short as $k = 1$, the layout quality appears substantially better for the majority of update schedules and approximation algorithms. Higher values of k do not appear to provide much improvement in layout quality, and in fact large values of k sometimes produce lower quality layouts (e.g. for number of crossings when $k = 9$ or 10).

For both the BH and FMM algorithms, the dynamic schedule appears to have faster runtime than the logarithmic and uniform schedules for $k > 2$. Therefore, **Hypothesis 2** is that the dynamic schedule will decrease the BH and FMM runtime more than the logarithmic or uniform schedules if $k = 4$.

For BH, the dynamic schedule appears to generally have worse readability metric performance than the standard schedule. **Hypothesis 3** is that the dynamic schedule will perform worse on all readability metrics than the standard schedule, except for edge crossings where the dynamic schedule will perform about the same.

On the other hand, for BH with the logarithmic and uniform schedules where $k = 4$, these schedules tend to have better readability metrics. **Hypothesis 4** is that, for BH, the logarithmic and uniform schedules will have better readability metrics compared to the standard schedule, except for crossing angle where logarithmic and uniform will perform worse.

For FMM, we see similar, but somewhat different, relationships in the readability metrics. By looking at the data, we believe that all other schedules perform better than the standard schedule on the edge crossings and angular resolution (min) metrics (**Hypothesis 5**), all schedules perform worse than the standard schedule on the crossing angle metric (**Hypothesis 6**), and that the dynamic schedule will perform worse and the logarithmic and uniform schedules will perform better than the standard schedule on the angular resolution (dev) metric (**Hypothesis 7**).

V. Experimental Comparison

This analysis tests the hypotheses developed in the exploratory analysis (Section IV) on a new set of 50 graphs. These graphs are different from the graphs used in Section IV, but they were collected using the same process.

Following existing best practices [30], [31], this evaluation's experimental design is as follows. It uses a within-subjects design on the three schedules (dynamic, logarithmic with $k = 4$, and uniform with $k = 4$), two approximation algorithms (BH and FMM), and four layout readability metrics discussed in Section IV. The first quartile, median, and third quartile for the dynamic schedule's number of updates on the Barnes-Hut algorithm was 11, 12, and 13, and for the Fast Multipole Method algorithm it was 12, 14, and 15. Because this is comparing three alternative schedules to the standard schedule across five response variables (runtime and the four readability metrics), the evaluation uses a Bonferroni corrected significance level $\alpha = 0.05/15 = 0.00\overline{3}$. The corresponding confidence interval is $[0.001\overline{6}, 0.998\overline{3}]$.

Because the readability metrics are bounded in $[0, 1]$, and many are not normally distributed, this evaluation uses bootstrapped confidence intervals with 10,000 samples. Effect sizes

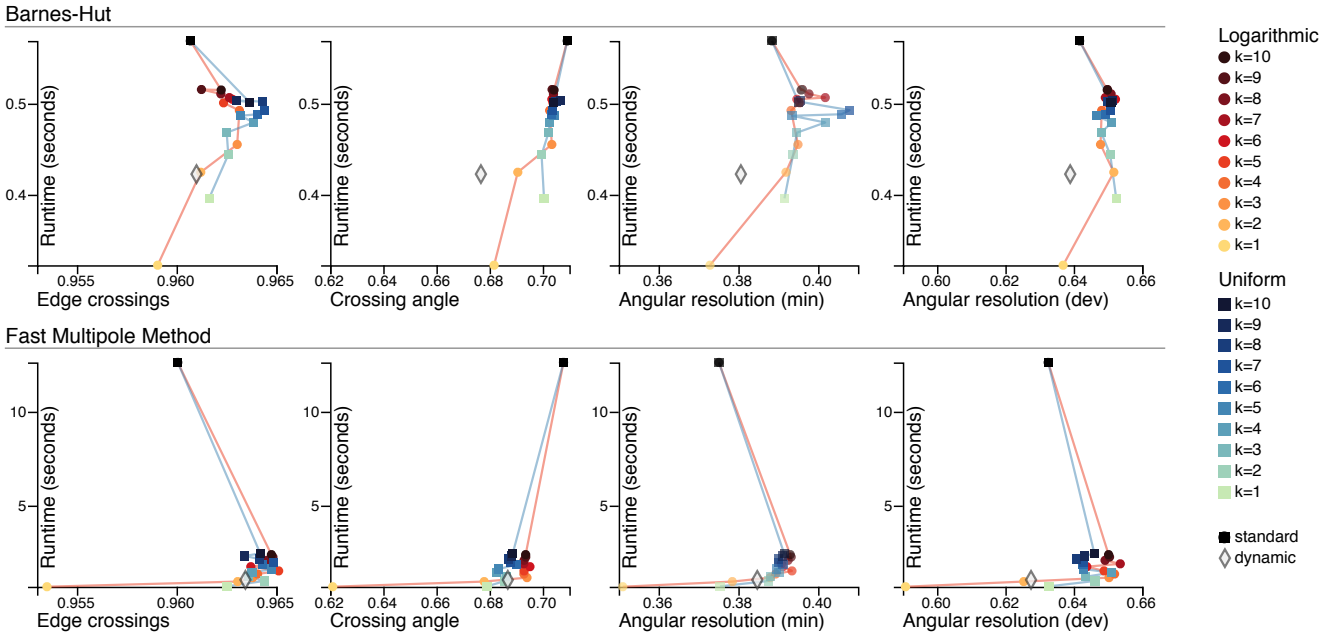

Fig. 1. Median runtime and readability metrics across all 50 exploratory graphs calculated for each update schedule (logarithmic for k from 1 to 10, uniform for k from 1 to 10, the standard schedule, and the dynamic schedule) and approximation algorithm (Barnes-Hut and Fast Multipole Method).

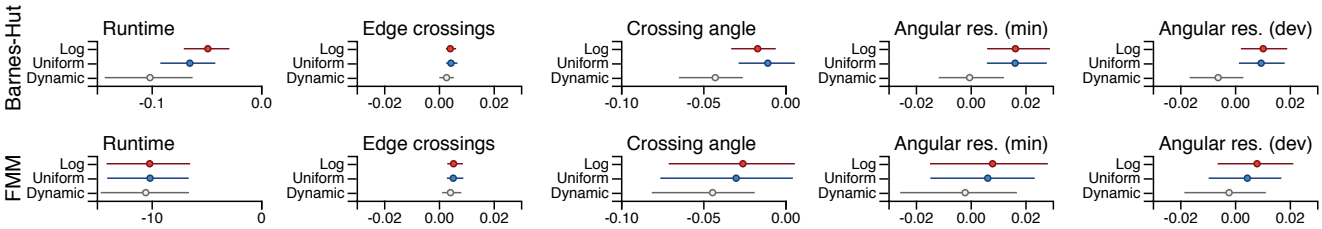

Fig. 2. Bootstrap 99.$\overline{6}$% confidence interval of the effect sizes of the runtime and graph readability metrics. Confidence intervals are percentile bootstrap with 10,000 samples. Effect size is sample mean difference (log, uniform, or dynamic minus standard). Note the differences in x-axis scales.

are sample mean difference ($s_{alt} - s_{standard}$ where s_{alt} is the value for one of the alternate schedules, i.e. logarithmic, uniform, or dynamic). Therefore negative effects indicate the standard schedule has a higher value, whereas positive effects indicate the alternate schedule under test has a higher value. Figure 2 shows the estimation plots for the runtime and readability metrics for the two approximation algorithms.

None of the confidence intervals in the runtime plots cross 0. This indicates that all of the tested update schedules perform faster than the standard update schedule for both the Barnes-Hut (BH) and Fast Multipole Method (FMM) approximation algorithms, which supports **Hypothesis 1**.

Although this experiment does not explicitly test **Hypothesis 2** using a null hypothesis significance test, the dynamic schedule has a much larger effect size on runtime than the logarithmic or uniform schedules for BH. The raw effect size is -0.101 seconds, compared to -0.065 seconds for uniform, and -0.049 seconds for logarithmic. This supports, but does not prove, Hypothesis 2.

We have good evidence to support parts of **Hypothesis 3**. Compared to the standard schedule on BH, the dynamic schedule is better on edge crossings than the standard schedule (the effect is positive and the confidence interval does not cross 0), but the dynamic schedule is worse on crossing angle. For both angular resolution metrics the confidence interval crosses zero, and therefore there is no evidence to say that the dynamic schedule performs better or worse than the standard schedule.

Similarly, we have good evidence to support parts of **Hypothesis 4**. For BH, both the logarithmic and uniform schedules perform better than the standard schedule on edge crossings and the angular resolution metrics. On the crossing angle metric, the logarithmic schedule performs worse than the standard schedule, but the confidence interval for the uniform schedule crosses 0, so we are unable to say whether its performance is better or worse than the standard schedule.

We partly accept **Hypothesis 6**. Although none of the FMM confidence intervals for edge crossings cross 0, they all cross 0 for the angular resolution (min) metric. Therefore we conclude that all of the schedules perform better on edge crossings than the standard schedule, but we cannot say whether there

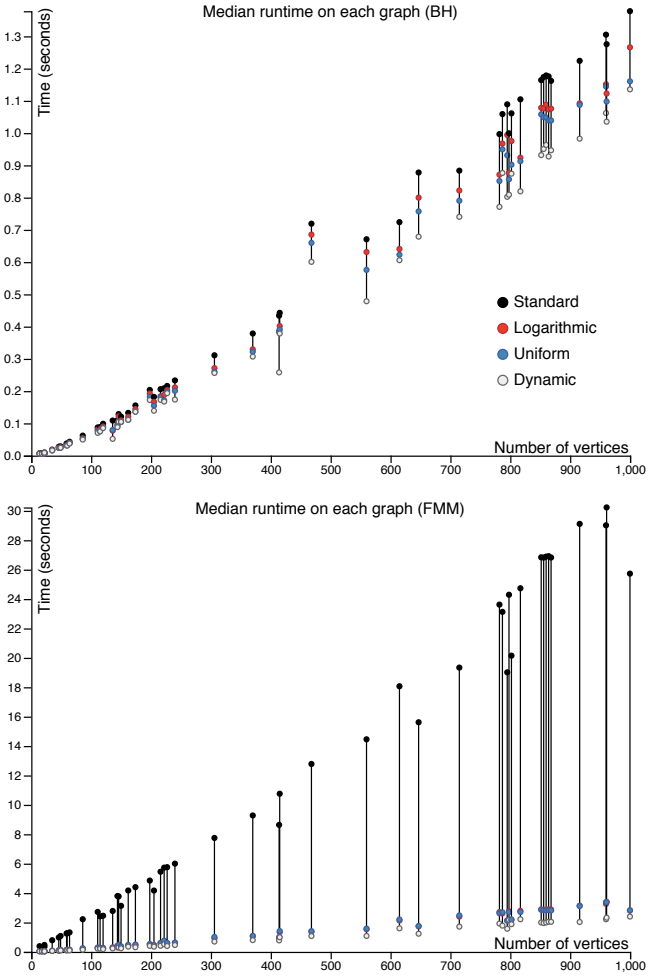

Fig. 3. Median runtime on each of the 50 graphs in the experimental data set. Dots represent the standard schedule (black), the logarithmic schedule (red), the uniform schedule (blue), and the dynamic schedule (gray). The top graph is for the Barnes-Hut algorithm, and the bottom is for Fast Multipole Method. Horizontal jitter resolves occlusion for graphs with identical numbers of vertices.

is any difference from the standard schedule for the angular resolution (min) metric.

Although **Hypothesis 6** is that all of the schedules perform worse than the standard schedule on crossing angle, the evidence only shows that the dynamic schedule performs worse. We do not have enough evidence to say that the logarithmic or uniform schedules perform worse.

Finally, we do not find evidence to support **Hypothesis 7**; all of the confidence intervals for the angular resolution (dev) cross 0, so the logarithmic, uniform, and dynamic schedules do not appear to have significantly different performance from the standard schedule.

VI. Effect of Graph Size on Runtime

The analysis in Section V showed that the logarithmic, uniform, and dynamic schedules have a decrease in runtime over the standard schedule. This section explores this further

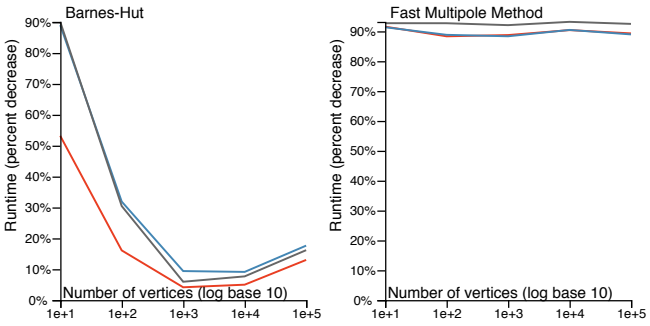

Fig. 4. Percent decrease in runtime from the standard schedule for the logarithmic (red), uniform (blue), and dynamic (gray) update schedules. The percent decrease is calculated for each of the sparse graphs, which have 10, 100, 1,000, 10,000, and 100,000 vertices. Note the log scale on the x-axis.

to better understand the runtime decrease seen in practice and to better understand the influence of graph size.

Figure 3 shows the runtime of each schedule with each approximation algorithm on each of the 50 graphs used in the experimental analysis in Section V.

For Barnes-Hut (BH), the dynamic algorithm has the largest decrease in runtime compared to the standard schedule (the smallest percent decrease is 10%, the largest is 52%, and the median is 18%). (Percent decrease is the difference between the schedule's runtime and the standard schedule's runtime, divided by the standard schedule's runtime.) The uniform schedule did better than the logarithmic schedule in the worst and typical cases (the smallest and median percent decrease was 8% and 14% for the uniform schedule and 5% and 9% for the logarithmic schedule), but in the best case the logarithmic schedule slightly outperformed the uniform schedule (29% compared to 27%).

For the Fast Multipole Method (FMM), the percent decrease in runtime is substantially larger, but very similar for each update schedule (bottom of Figure 3). The logarithmic schedule ranged from 88% to 90% (median 89%), the uniform schedule ranged from 87% to 90% (median 89%), and the dynamic schedule ranged from 89% to 94% (median 92%).

In order to examine the performance of each schedule as $|V|$ increases, Figure 4 shows the percent decrease of each schedule compared to the standard schedule on five sparse graphs of varying size. The five sparse graphs have 10, 100, 1,000, 10,000, and 100,000 vertices, and the graphs were generated with $|E| = 2|V|$ random edges (without replacement). Keeping the proportion of edges fixed allows us to more easily understand the change in runtime performance for repulsive force calculation as graphs get larger.

For BH, the uniform and dynamic schedules have the largest percent decrease for small graphs, but the overall runtime is small so this translates into a savings of about 26 and 37 milliseconds for graphs with 10 and 100 vertices respectively when using the uniform update schedule. For larger graphs, the uniform schedule appears to offer the largest percent decrease; for the largest graph, the percent decrease is about 18%, which saves about 86 seconds (reduced from about 490 seconds to

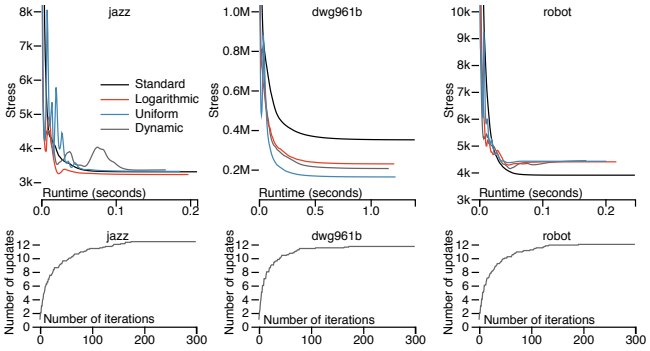

Fig. 5. (TOP) Convergence rate of the stress metric for the four update schedules on the Barnes-Hut approximation. (BOTTOM) The average number of updates of the dynamic schedule at each iteration.

about 404 seconds). For comparison, on the largest graph the dynamic schedule had a percent decrease of about 16%, or about 79 seconds.

For FMM, the percent decrease is large and consistent regardless of graph size and update schedule, ranging from about 88% to about 93%. In contrast to BH, the dynamic schedule has a larger percent decrease, whereas the uniform and logarithmic schedules are about the same.

For BH, the dynamic schedule updated 11 times on the smallest graph and either 8 or 9 times on all other graphs, while for FMM the dynamic schedule updated 18 times on the smallest graph and 12 times on all other graphs. This indicates that the size of the graph does not seem to affect the number of updates.

VII. Convergence

Although using stress as a graph layout quality metric can have problems (see the discussion in Section IV), it is widely used and regarded as an important metric. We are also interested in understanding whether the logarithmic, uniform, and dynamic schedules affect the convergence rate of the layout algorithm. Therefore, this section presents an analysis of stress for each update schedule over 300 iterations on three graphs. These graphs were chosen from the 50 experimental graphs. The stress and layout time are averaged after each iteration across 10 runs of each update schedule. The results are shown in Figure 5. (Due to space constraints, only the Barnes-Hut convergence rates are shown.) Sometimes all of the schedules converge to similar values (the jazz graph), sometimes the alternative schedules converge faster and to lower values than the standard schedule (the dwg961b graph), and sometimes the standard schedule converges to lower values (the robot graph). In most cases, the alternative schedules converge less smoothly than the standard schedule, which is probably because the tree oscillates between being up to date and out of date. However, the logarithmic and uniform schedules have converged by the time the standard schedule has.

The bottom of Figure 5 shows the average number of updates of the dynamic schedule at each iteration. We see that most updates occur during the early iterations. This appears to mirror the convergence charts, and this indicates that the dynamic schedule constructs new trees less often as the layout converges. By the time the layout has converged, the dynamic algorithm typically has stopped constructing new trees.

VIII. Graph Layout Examples

Figure 6 shows the layouts for three graphs generated by the Barnes-Hut (BH) and Fast Multipole Method (FMM) algorithms using the standard, dynamic, logarithmic, and uniform update schedules. These are the same graphs used to test convergence in Section VII. For the most part, the layouts produced by the standard schedule are extremely similar to the layouts produced by the other schedules. Notably, the dynamic FMM layout for the jazz and robot graphs appears to put some vertices too close together while putting other vertices too far apart. The layouts for dwg961b all appear very similar and seem to show the same structure and shape. The dynamic BH and some of the FMM layouts also appear to hide the curvature of the robot layout (due to the central gravitational force in the layout algorithm). These results corroborate the findings in Section V that the logarithmic and uniform schedules produce layouts with similar quality as the standard schedule. It also indicates that the final stress values shown in Figure 5 may not indicate worse layout quality.

IX. Discussion

The analysis in this paper shows that the time required to calculate new trees is a nontrivial part of the overall runtime of spring-electric algorithms that use the Barnes-Hut (BH) approximation and the Fast Multipole Method (FMM). By reducing the number of times the algorithm computes a new tree, we can reduce the time required to compute layouts. Note that these experiments report the reduction in the total running time of graph layout algorithms rather than only the reduction in repulsive force calculation running time. The graph layout algorithms also include calculations for spring forces and a central gravitational force, so the reduction in repulsive (electric) force calculation running time is likely much larger than the results reported here.

Overall, the dynamic schedule appears to be the fastest, although in some cases the runtime improvement may not be much more than the logarithmic and uniform schedules. The dynamic schedule's improvement in runtime appears to come at the cost of lower quality graph layouts. The runtime of the logarithmic and uniform schedules appear similar, although perhaps uniform is slightly faster. Furthermore, the uniform schedule does not appear to have the diminished graph layout quality that we sometimes see in the logarithmic schedule, and in many cases it performs better on readability metrics than the standard schedule. Therefore, practitioners and system implementers would be best choosing either the uniform schedule with $k = 4$ (i.e. construct a new tree approximately once every 13 or 14 iterations) or the dynamic schedule, depending on their preference for trading off speed and quality. These schedules provide modest runtime improvements for BH, but quite substantial runtime improvements for FMM.

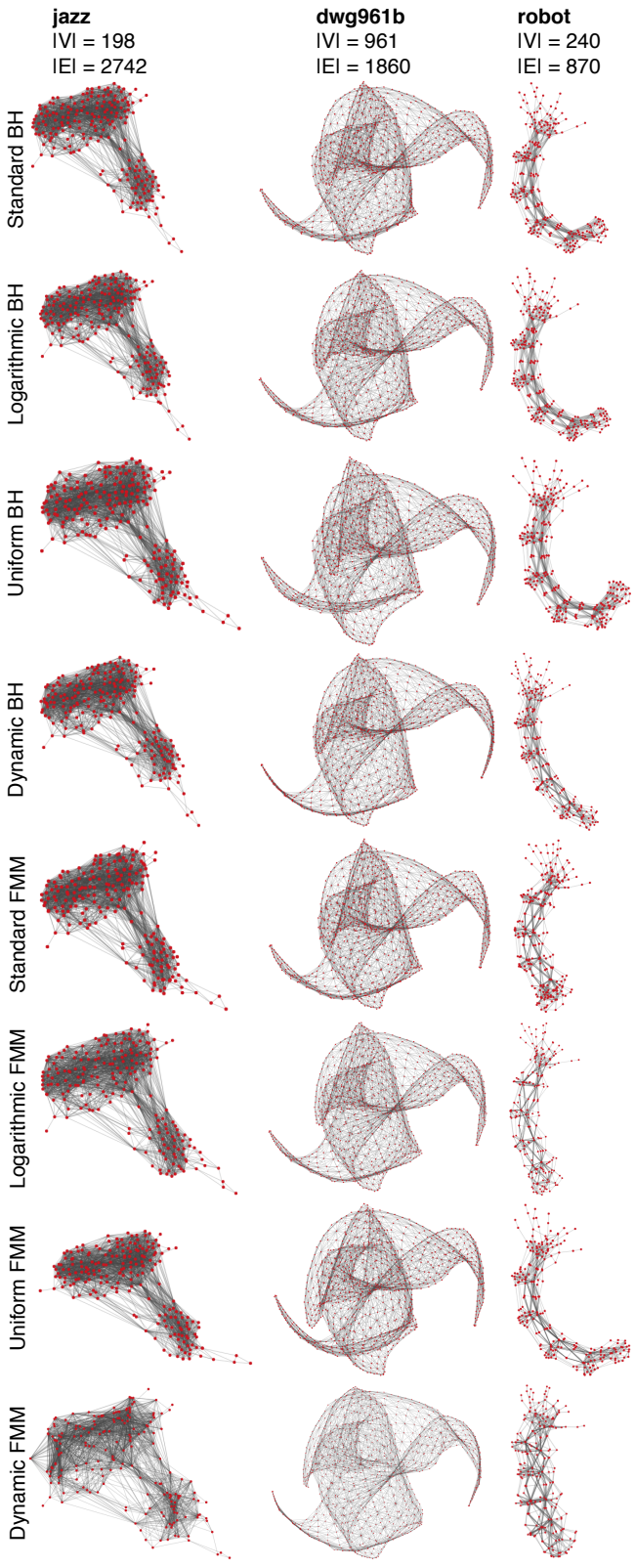

Fig. 6. Layouts of three graphs produced by the Barnes-Hut (BH) and Fast Multipole Method (FMM) algorithms using the four update schedules.

The fact that the uniform schedule is less precise than the standard schedule but produces better quality layouts seems counter intuitive. However, this is consistent with the preliminary results presented by Lipp et al. [5].

The reason why the logarithmic, uniform, and dynamic schedules improve the FMM runtime so much is likely because building the tree for the FMM is $O(|V| \log |V|)$, but computing forces is only $O(|V|)$. Therefore, constructing a new tree is the largest computational cost, and reducing the number of trees constructed has a major impact on the runtime.

It is not clear why the FMM implementation used in this evaluation is so much slower than the BH implementation. The implementations were created by different people, so it is likely that the performance difference is due to differing levels of effort to optimize the implementations. Nonetheless, we still get a clear idea of how much runtime can be reduced by using the alternative update schedules.

It is worth noting that the observed layout quality effect sizes in the experimental analysis may be considered small in practice. For example, an effect size of -0.011 on crossing angle corresponds to a difference of 0.77 degrees in mean deviation from the ideal crossing angle. And an effect size of 0.004 on edge crossings means that if a graph has 1000 edges that can cross, then there will be 4 fewer edge crossings. It is unclear if humans would notice such small differences in practice, or if such small differences would have a detectable effect on speed or errors in analysis tasks.

X. Conclusion

This paper presented two new update schedules (uniform and dynamic) for determining when to construct a new tree in tree-based approximation algorithms such as Barnes-Hut, the Fast Multipole Method, or the Well-Separated Pair Decomposition. The evaluations show that constructing a new tree at a uniform frequency of once every 13–14 iterations achieves significantly faster performance compared to the standard update schedule. In addition, the uniform schedule achieves better edge crossing and angular resolution graph readability metrics, and it does not appear to come at the cost of a degradation in edge crossing angle that occurs with the dynamic and logarithmic update schedules. The uniform update schedule also appears to improve the angular resolution metrics for the Barnes-Hut approximation, but not with the Fast Multipole Method. These new update schedules are simple modifications to the existing approximation algorithms, and therefore they present an easy but effective way to improve the runtime.

Because spring-electric algorithms have many uses, such as in multi-level graph layout algorithms [14] and flow diagrams [32], the logarithmic, uniform, and dynamic schedules can be used to improve runtime in many applications. This includes domains such as n-body simulations or the t-SNE algorithm [33]. Future work should evaluate these other uses.

Acknowledgement

Thanks to Nathan Danneman, Ben Gelman, Jessica Moore, Tony Wong, and the anonymous reviewers for providing helpful feedback on this work.

REFERENCES

[1] A. Arleo, W. Didimo, G. Liotta, and F. Montecchiani, "A Million Edge Drawing for a Fistful of Dollars," in *International Symposium on Graph Drawing and Network Visualization*, 2015, pp. 44–51.

[2] T. M. Fruchterman and E. M. Reingold, "Graph drawing by forcedirected placement," *Software: Practice and Experience*, vol. 21, no. 11, pp. 1129–1164, 1991.

[3] S. G. Kobourov, "Force-Directed Drawing Algorithms," in *Handbook of Graph Drawing and Visualization*, 1st ed., R. Tamassia, Ed. Boca Raton: Chapman and Hall/CRC, 2016, ch. 12, pp. 383–408.

[4] F. Lipp, A. Wolff, and J. Zink, "Faster force-directed graph drawing with the well-separated pair decomposition," in *International Symposium on Graph Drawing and Network Visualization*, 2015, pp. 52–59.

[5] ——, "Faster Force-Directed Graph Drawing with the Well-Separated Pair Decomposition," *Algorithms*, vol. 9, no. 3, p. 53, 2016.

[6] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: An Open Source Software for Exploring and Manipulating Networks," in *International AAAI Conference on Weblogs and Social Media*, 2009, pp. 361–362.

[7] N. G. Belmonte, "JavaScript InfoVis Toolkit." [Online]. Available: http://philogb.github.io/jit/

[8] M. Bostock and J. Heer, "Protovis: A graphical toolkit for visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1121–1128, 2009.

[9] M. Bostock, V. Ogievetsky, and J. Heer, "Data-Driven Documents," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 2301–2309, 2011.

[10] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *Software Practice and Experience*, vol. 30, no. 11, pp. 1203–1233, 1999.

[11] S. Hachul and M. Jünger, "Large-Graph Layout Algorithms at Work: An Experimental Study," *Journal of Graph Algorithms and Applications JGAA*, vol. 11, no. 2, pp. 345–369, 2007.

[12] J. Heer, S. K. Card, and J. A. Landay, "Prefuse: a toolkit for interactive information visualization," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2005, pp. 421–430.

[13] J. Heer, "Flare: Data Visualization for the Web," 2008.

[14] Y. Hu, "Efficient, High-Quality Force-Directed Graph Drawing," *Mathematica Journal*, vol. 10, no. 1, pp. 37–71, 2005.

[15] M. Jacomy, T. Venturini, S. Heymann, and M. Bastian, "ForceAtlas2, a continuous graph layout algorithm for handy network visualization designed for the Gephi software," *PloS one*, vol. 9, no. 6, 2014.

[16] M. Khoury, Y. Hu, S. Krishnan, and C. Scheidegger, "Drawing Large Graphs by Low-Rank Stress Majorization," *Computer Graphics Forum*, vol. 31, no. 3pt1, pp. 975–984, 2012.

[17] J. Nielsen, *Usability Engineering*, 1st ed. Academic Press, 1993.

[18] J. Barnes and P. Hut, "A hierarchical O(N logN) force calculation algorithm," *Nature*, vol. 324, pp. 446–449, 1986.

[19] A. Quigley and P. Eades, "FADE: Graph drawing, clustering, and visual abstraction," in *International Symposium on Graph Drawing*, 2000, pp. 197–210.

[20] S. Aluru, J. Gustafson, G. Prabhu, and F. E. Sevilgen, "Distribution-independent hierarchical algorithms for the n-body problem," *The Journal of Supercomuting*, vol. 12, no. 4, pp. 303–323, 1998.

[21] L. F. Greengard, "The rapid evaluation of potential fields in particle systems," Ph.D. dissertation, Yale University, New Haven, CT, USA, 1987.

[22] S. Hachul and M. Jünger, "Drawing large graphs with a potential-field-based multilevel algorithm," in *International Symposium on Graph Drawing*, 2004, pp. 285–295.

[23] P. B. Callahan and S. R. Kosaraju, "A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields," *Journal of the ACM*, vol. 42, no. 1, pp. 67–90, 1995.

[24] J. Kunegis, "KONECT - The Koblenz Network Collection," in *Web Observatory Workshop*, 2013, pp. 1343–1350.

[25] T. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1:1–1:25, 2011.

[26] P. Eades, S.-H. Hong, A. Nguyen, and K. Klein, "Shape-Based Quality Metrics for Large Graph Visualization," *Journal of Graph Algorithms and Applications*, vol. 21, no. 1, pp. 29–53, 2017. [Online]. Available: http://jgaa.info/getPaper?id=405

[27] J. F. Kruiger, P. E. Rauber, R. M. Martins, A. Kerren, S. Kobourov, and A. C. Telea, "Graph Layouts by t-SNE," *Computer Graphics Forum*, vol. 36, no. 3, pp. 283–294, 2017.

[28] E. R. Gansner, Y. Koren, and S. C. North, "Graph drawing by stress majorization," in *International Symposium on Graph Drawing*, 2004, pp. 239–250.

[29] C. Dunne, S. I. Ross, B. Shneiderman, and M. Martino, "Readability metric feedback for aiding node-link visualization designers," *IBM Journal of Research and Development*, vol. 59, no. 2/3, pp. 14:1–14:16, 2015.

[30] P. Dragicevic, "Fair statistical communication in hci," in *Modern Statistical Methods for HCI*, J. Robertson and M. Kaptein, Eds. Springer, 2016, ch. 13, pp. 291–330.

[31] C. Rainey, "Faster Force-Directed Graph Drawing with the Well-Separated Pair Decomposition," *Journal of Political Science*, vol. 58, no. 4, pp. 1083–1091, 2014.

[32] K. Wongsuphasawat and D. Gotz, "Exploring flow, factors, and outcomes of temporal event sequences with the outflow visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 12, pp. 2659–2668, 2012.

[33] L. van der Maaten, "Accelerating t-sne using tree-based algorithms," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3221–3245, 2014.