Sandpile prediction on a tree in near linear time*

Akshay Ramachandran† Aaron Schild‡

November 2, 2016

Abstract
In the sandpile model, we are given an undirected graph $G$ and an initial list of chip counts on each vertex of $G$ and we may fire $\deg(v)$ chips from any vertex $v$ to its neighbors. Doing chip moves either results in a unique terminal configuration or recurs forever. On many families of graphs – including trees – the problem of computing the final configuration is P-complete [13] and simulation can take as long as $\Theta(n^3)$ time. We give a $O(n \log^2 n)$ time algorithm for trees that computes the terminal configuration or shows that chip firing will not terminate.

1 Introduction
The computational complexity of simulation has long been of interest in physics, mathematics, and computer science. This interest began with Turing completeness [17], which has had a large impact on our understanding of cellular automata like Conway’s Game of Life [1]. Since the 1990s, researchers have been interested in whether or not the output of a simulation can be computed using a parallel algorithm [13, 14]. In particular, they have been interested in which simulations are P-complete versus which ones are in NC.

In the past decade, with the introduction of larger datasets, researchers have become interested in more fine-grained notions of complexity. This has triggered interest in computing the output of simulations more efficiently using algorithms that are not necessarily parallelizable. This line of work started with shortcutting random walks in order to compute random spanning trees [11, 9].

We extend this line of work to the problem of sandpile prediction. In this problem, we are given an undirected graph with a nonnegative number of chips on each vertex. If a vertex $v$ has at least $\deg(v)$ chips, we may “fire” it, meaning that we may take $\deg(v)$ chips from it and distribute one to each of its neighbors. Firing vertices repeatedly either results in a terminal configuration with no vertex having $\deg(v)$ chips on it or continues forever and is recurrent. All valid orders of firings reach the same terminal configuration [3]. We want to find the terminal configuration if it exists or output that firing never terminates.

Sandpile prediction is of wide interest in physics, computer science, and mathematics, both for its beautiful algebraic structure [2, 8] and for its relevance to applications like load balancing [19] and derandomization of models like internal diffusion-limited aggregation [4, 5]. The sandpile model is related to many other models and physical phenomena, like the rotor-routing model [18], avalanche models [6], and self-organized criticality [15].

This interest in chip firing has extended to its computational complexity as well. Approaches to computing the terminal configuration fall into two categories:

- Bounding the number of chip firings required to reach the terminal configuration. This approach applies to general graphs. [3, 8, 7]
- Bypassing simulation to compute the terminal configuration more efficiently. This approach currently only works for paths. [13, 9]

The first approach began with a paper of Bjorner, Lovasz, and Shor [3] which showed that in a terminating sequence there can be at most $2|V|N/\lambda_2$ firings, where $N$ is the total number of chips and $\lambda_2$ is the smallest non-trivial eigenvalue of the graph Laplacian. A better bound based on a random walk argument showed that the number of chip moves is at most $2N|E|R_{\text{max}}$ [8], where $R_{\text{max}}$ is the maximum effective resistance between any two nodes. This bound is often as high as $\Omega(n^3)$ on sparse graphs. Sandpile prediction is P-complete on many classes of graphs, including trees [7] and grids with dimension greater than 3 [13]. The second approach only works on paths, achieving $O(n \log n)$ work algorithms with depths $O(\log^3 n)$ [13] and $O(\log^2 n)$ [9] respectively.

*Work done while both authors were students at University of California, Berkeley
†University of Waterloo, Supported by David R Cheriton Graduate Scholarship. aramachandrangraduate Waterloo.ca
‡University of California, Berkeley, Supported by NSF grants CCF-1528174 and CCF-1535989. aschild@berkeley.edu

Copyright © by SIAM
Unauthorized reproduction of this article is prohibited
In this paper, we extend the second approach to work for trees as well. Unlike the case of paths, computing the terminal configuration of a tree is P-complete, so we are unable to parallelize our algorithm. This paper is the first speedup over simulation obtained for sandpile prediction on a family of graphs for which the problem is P-complete.

1.1 Preliminaries In this paper, all graphs are undirected and unweighted. A graph $G$ has vertex set $V(G)$ and edge set $E(G)$. For any vertex $v \in V(G)$, $\deg(v)$ denotes the number of neighbors of $v$.

In the sandpile prediction problem, one is given an undirected graph $G$ and an initial configuration vector $\sigma^0 \in \mathbb{Z}_{\geq 0}^{V(G)}$. We call the ordered pair $(G, \sigma^0)$ a sandpile prediction instance. We can fire any vertex $v$ with $\sigma_v \geq \deg(v)$, which changes $\sigma$ in the following way:

$$
\sigma^{t+1}_u = \begin{cases} 
\sigma_u - \deg(v) & u = v \\
\sigma_u + 1 & u \text{ is a neighbor of } v \\
\sigma_u & \text{otherwise}
\end{cases}
$$

One of two outcomes occurs:

- No more chip firings are possible, that is there exists some time $t$ for which $\sigma^t_u < \deg(u)$ for all vertices $u \in V(G)$. We call such instances terminal and let $\sigma^t$ be a terminal configuration.
- There are always possible firings. These instances are called recurrent.

We say that an algorithm solves the sandpile prediction problem if it decides whether or not an instance is terminal and, if it is terminal, outputs the terminal configuration. An important background result is the following:

**Theorem 1.1.** [3] Any terminal instance of the sandpile prediction problem has a unique terminal configuration. In particular, this terminal configuration is independent of the order of firing.

In particular, any firing order will result in the same number of firings.

For a graph $G$, consider a vertex $r$. Do a depth-first search (DFS) from $r$. The DFS preorder is the order in which vertices are first visited by the DFS.

1.2 Summary of results Our main result is the following:

**Theorem 1.2.** There is a $O(n \log^5 n)$ time algorithm which solves the sandpile prediction problem when the input graph is a tree.

This is the first result that improves upon simulation for trees. Simulation on a path, for example, can take as long as $\Omega(n^3)$ time. This result is within a $\polylog(n)$ factor of the optimal runtime. Unlike in the case of paths, no $O(\polylog(n))$ depth parallel algorithm can be found for sandpile prediction on trees unless $P = NC$ [7].

To illustrate the ideas, we start by showing the following easier result in Section 2:

**Theorem 1.3.** There is an $O(n)$ time algorithm which, given a terminal configuration $\sigma$ on a tree $T$, solves the sandpile prediction instance $(T, \sigma^t)$, where $\sigma^t_v = \sigma_v + 1$ for some coordinate $v$ and $\sigma^t_u = \sigma_u$ for all $u \neq v$. In particular, this gives an $O(n^2)$ time algorithm for solving sandpile prediction on a tree.

In Section 3, we prove the following better amortized runtime bound:

**Lemma 1.1.** The algorithm used to prove Theorem 1.3 solves the sandpile prediction problem in $O(nD)$ time if chips are dropped in DFS preorder with respect to an arbitrary fixed vertex $r$, where $D$ is the diameter of the input tree $T$.

We use a stronger version of this result in Section 4 that characterizes the number of net firings that occur on any set $F \subseteq E(T)$. By limiting the set $F$ to the set of light edges of a heavy-light decomposition of $T$, we can reduce the number of operations required to $O(n \log n)$. We must design a data structure, though, that can skip all rounds that involve operations outside of $F$. We do this using an “exact data structure,” whose guarantees are given in Lemma 4.1, and an “approximate data structure,” whose guarantees are given in Lemma 4.2. The exact data structure, given the next operation in $F$, can update the configuration to reflect the result of that operation. The approximate data structure finds the next operation to occur in $F$.

1.3 Techniques Think of sandpile prediction as a sequence of $O(|E(G)|)$ chip drops onto a graph (as an instance is recurrent if there are more than $2|E(G)| - |V(G)|$ chips [3]). After dropping a chip, we perform all chip firings required until the configuration becomes terminal before adding the next chip. On paths, Milterson [12] gave a simple procedure for computing the terminal configuration after one chip drop in $O(1)$ time. If the chip is dropped on a vertex $v$ with no chips, nothing happens. If there was already a chip on $v$, then the two nearest nodes with no chips gain one, and the node $v - z_1 - z_2$ becomes empty, where $z_1$ and $z_2$ are the positions of the two nearest gaps. This simple algorithm
is improved by carefully keeping track of an ordering of twos and zeroes, in which case the ending configuration of a path can be produced in $O(\log^3 n)$ depth, $O(n \log n)$ work [13].

We generalize the ideas of Milterson in the form of critical components to obtain a $O(nD)$ time algorithm for general trees in Section 2, where $D$ is the diameter of the tree. A critical component is a connected subtree of vertices $v$ with $\deg(v) - 1$ chips. Since a tree contains a unique path between any two vertices, the critical component is exactly the set of nodes that is connected to the dropped chip by a path of firing neighbors. The algorithm executes chip firing in a sequence of rounds. In each round, every fireable vertex is fired once. If a node and all its neighbors fire in a single round, its number of chips does not change. We exploit this phenomena to obtain a faster simulation, which we call round-based simulation. Say that an edge $e$ has a net firing if exactly one of its incident vertices fires during that round. In a tree, these net firings always move the boundary of the critical component inwards by one step. We refer to the leaves of a critical component as those nodes that fire across a net firing edge. The number of rounds can be as high as $\Theta(n)$ on a path, so it is not enough to simulate each round in $O(n)$ time.

Milterson’s approach skips all but two rounds by noticing that they consist of the critical component moving in one step on either side. On trees, shortcutting rounds is more complicated, even when chips are only dropped on one vertex. This is because when one executes many rounds, sections of the tree that remain in the critical component for longer times experience more rounds. We decompose a tree into paths by using the heavy-light decomposition [16]. This results in a decomposition of an arbitrary tree into a tree of paths that has $O(\log n)$ diameter. Furthermore, on rounds when there are no net-firings over light edges, the critical component acts as a set of paths. Unlike Milterson’s approach which does work only when chips are dropped on the graph, we must also do work when a chip crosses a light edge of the heavy-light decomposition. We show in Lemma 3.1 that there are only $O(n \log n)$ net firings over light edges.

Since only $O(n \log n)$ net firings occur over light edges, we can obtain a near-linear time algorithm by processing the result of each net firing over a light edge in polylog$(n)$ time. Ideally, we could design one data structure that does this. This data structure would need to do two tasks, each in polylog$(n)$ time:
• Find the next round in which net firings over light edges elapse.

• Process all changes to the tree that occur before the next round in which a net firing over a light edge elapses.

Unfortunately, it is difficult to design a single data structure that accomplishes both of these tasks. For example, think about the graph in Figure 1, which consists of a long path $p$ with side paths hanging off of each vertex of $p$. We need a data structure that keeps track of the numbers given in Figure 1, which represent the number of rounds until a new chip is added to each side path. The light edges of the decomposition (which are dashed) connect $p$ to each side path. Think of the data structure as storing a number for each path connected to $p$ indicating the number of rounds that need to elapse before the side path will accept a chip from $p$. There are two challenges:

• The number of rounds that elapses on each side path varies between two events. For example, only 1 is subtracted from the side path hanging off of $v_1$, since the critical component does not contain $v_1$ after one round. 2 is subtracted from all other side paths, since all other side paths are in the critical component for two rounds.

• We need to be able to find the minimum of the side-paths subject to the constraint that the side path is still in the critical component. This minimum tells us the first side path in which the boundary of the critical component hits $p$. For example, in Figure 2, the minimum number is that stored at $v_3$. This matches the fact that the boundary of the critical component in $v_3$’s side path will hit $p$ before any other side path.

Both of these criteria are hard to design into a single data structure like dynamic trees [16] because updating different entries by distinct values can change the sorted order of entries within a updated range. Instead, we use the structure of our problem to allow for a relatively small number of fake events. We deal with these issues using two data structures: an exact data structure and an approximate one. The exact data structure keeps track of all chip locations, but is not able to find out when the next event will occur. The approximate one outputs candidate events and checks against the exact data structure to make sure that they actually happen.

We charge each fake event on a side path $q$ to a constant factor reduction in the number of rounds that need to elapse before the next real event. This ensures that the total number of fake events is $O(\log n)$ times the number of real events.

2 An $O(n^2)$ algorithm for trees using net firings

Simulating chip firing can take as long as $\Omega(n^3)$ time, even on paths. Simulation takes a long time in large part because when a chip is dropped at a vertex $v$, many chips can be fired in the direction of $v$. Ideally, an algorithm for computing the terminal configuration would just have to send chips away from $v$.

In trees, we can speed up a simulation in this way if we focus on resolving the addition of one chip at a time. A priori, adding one chip and simulating the result could take as long as $\Omega(n^2)$ time. We will compute the results of the simulation in $O(n)$ time. We start with an important definition:

**Definition 1.** Consider a tree $T$. Call a vertex $v \in V(T)$ critical if the number of chips $\sigma_v$ on $v$ is equal to degree$(v) - 1$. Call $v$ supercritical and subcritical if $\sigma_v >$ degree$(v) - 1$ or $\sigma_v <$ degree$(v) - 1$ respectively.

Call a subtree $T' \subseteq T$ critical if all of the vertices in $T'$ are critical.

We will show the following:

**Theorem 2.1.** Algorithm 1 finds the terminal configuration after dropping a chip onto a critical subtree $T' \subseteq T$ in $O(|N(T') \cup V(T')|)$ time.

There are two key ideas behind the proof:

• Every vertex of $T'$ is critical at most once.

• No vertex outside of $T'$ is ever critical.

In particular, one should think of simulation as occurring in rounds. A round consists of firing all supercritical vertices exactly once. Notice that the chip counts on all vertices that are not on the boundary of $T'$ (adjacent to some vertex outside of $T'$) stay the same. Intuitively, one can think of the rounds as bringing the boundary inward one step at a time towards the vertex at which the chip was dropped. Once this vertex is on the boundary, firing stops. Without further ado, we prove the theorem.

**Proof.** First, inductively show that the critical component of $v$ on round $i$ consists of precisely the set of vertices $u$ with $d'_u \geq i$. For $i = 1$, vertices with $d'_u = 0$ do not fire on round 1 because they can only have one neighbor in the critical subtree $T'$, so they can only receive one chip in round 1. This is not enough to make $u$ fire, because $u$ is subcritical before round 1 takes place. Vertices with $d'_u \geq 1$ do fire because they are connected to the supercritical vertex $v$ by a path of critical vertices. This completes the proof that the round 1 critical component is precisely the set of vertices $u$ with $d'_u \geq 1$. 1118
Algorithm 1 \textbf{DropChip}(T, v, \sigma)

\textbf{Input:} tree $T$, vertex $v \in V(T)$, configuration $\sigma$
\textbf{Output:} terminal configuration $\tau$ after adding a chip to $v$

1: $T' \leftarrow \text{maximal critical subtree of } T \text{ containing } v$
2: $d \in \mathbb{R}^{V(T')} \leftarrow \text{distances of each vertex } u \in V(T')$
   to $V(T) \cup V(T')$ in the directed tree $T$ with edges directed away from $v$
3: \textbf{for all } $u \in V(T')$ \textbf{do}
4: \hspace{1em} $d'_u \leftarrow \min \text{ ancestors } w \text{ of } u \text{ including } u \text{ } d_w$
5: \textbf{end for}
6: \textbf{for all } $u \in V(T')$ \textbf{do}
7: \hspace{1em} $\tau_u \leftarrow (\text{number of neighbors } x \text{ with } d'_x \geq d'_u) + (\text{number of neighbors } x \text{ with } d'_x \geq d'_u + 1)$
8: \hspace{1em} $\tau_u = -1(u \neq v)$
9: \textbf{end for}
10: \textbf{return } $\tau$

Now, for $i > 1$ inductively assume that the vertices $u$ with $d'_u \geq i - 1$ are precisely the vertices that fire on round $i - 1$. Any vertex $u$ has an ancestor $w$, possibly equal to $u$, with $d'_u = d_w$. In particular, $w$ is adjacent to some vertex $x$ with $d'_x = d_w - 1$, so $d'_x < d'_u$. If $d'_u \leq i - 1$, then $d'_x \leq i - 2$ and $x$ does not fire on round $i - 1$ by the inductive hypothesis. $w$ either does not fire in round $i - 1$ or fires and loses a chip to $x$.

In particular, $w$ (if $w$ fires) or its parent (if $w$ does not fire) must be subcritical after round $i - 1$. Therefore, $u$ is not connected to $v$ through a path of critical vertices after round $i - 1$ and therefore does not fire on round $i$.

Now, suppose that $d'_u \geq i$. All neighbors $x$ of ancestors of $u$ have $d'_x \geq i - 1$. By the inductive hypothesis, they fire on round $i - 1$. This means that the number of chips on any vertex in the path from $u$ to $v$ does not change, so they all remain critical. Therefore, $u$ fires in round $i$. This completes the proof that the vertices with $d'_u \geq i$ are precisely the vertices that fire in round $i$.

We use this claim to analyze the algorithm. It takes $O(\nu(N(T')))!$ time since it just does two passes through $N(T') \cup T'$. It therefore suffices to show that it produces the correct configuration. For any neighbor $y$ of $u$, $|d'_y - d'_u| \leq 1$. On any round before round $d'_u$, $u$ and all of its neighbors lose, which means that $u$ has degree($u$) = $1(u \neq v)$ chips on it immediately before round $d'_u$. On round $d'_u$, $u$ loses chips to all of its neighbors $y$ with $d'_y = d'_u - 1$. On round $d'_u + 1$, it gains a chip from any neighbor $y$ with $d'_y = d'_u + 1$. $u$ does not gain or lose chips after this round. This proves that $\tau_u$ is correct.

3 Dropping chips takes $O(nD)$ net firings on a tree

We can get a better bound on the total number of chip firings over all chip drops by noticing that each round always moves chips away from the critical vertex that a chip was dropped at. This observation immediately leads to a proof that only $O(nD)$ net firings occur over the course of an arbitrary number of chip drops on a fixed vertex, where $D$ is the diameter of the tree. One can also show that there are at most $O(nD)$ net firings when the chip drops occur in a DFS order. More precisely, consider the following algorithm:

Algorithm 2 \textbf{DFSDrop}(T, \gamma)

\textbf{Input:} tree $T$, initial configuration $\gamma$
\textbf{Output:} terminal configuration $\sigma$

1: $r \leftarrow$ arbitrary vertex which is chosen to be the root of $T$
2: $v_1, v_2, \ldots, v_n \leftarrow$ DFS preorder of the vertices of $T$
   starting with $v_1 = r$
3: $\sigma \leftarrow$ all zeros vector on $V(T)$
4: \textbf{for all } $i = 1 \text{ through } n$ \textbf{do}
5: \hspace{1em} \textbf{for all } $j = 1 \text{ through } \gamma_{v_i}$ \textbf{do}
6: \hspace{2em} $\sigma \leftarrow \text{DropChip}(T, v_i, \sigma)$
7: \hspace{1em} \textbf{end for}
8: \textbf{end for}
9: \textbf{return } $\sigma$

Lemma 3.1. (Restatement of Lemma 1.1) Let $T$ be a tree and $F \subseteq E(T)$. Let $D$ be the diameter of the weighted tree $T$ with all edges in $F$ having length 1 and all edges outside of $F$ having length 0. Then the number of net firings across edges in $F$ in Algorithm 2 is at most $5nD$.

Proof. Direct all edges of $T$ away from $r$ and break up the set of net firings into two categories: away from $r$ and towards $r$. Let the numbers of such moves be $m_u$ and $m_d$ respectively. First, note that $m_u \leq m_d + nD$ since $D$ is an upper bound on the radius of $T$ with respect to $r$ and $F$. It therefore suffices to bound $m_d$.

For an edge $e \in F$, let $T_e$ denote the subtree of $T$ rooted at the leafward endpoint of $e$. Notice that net firings only enter $T_e$ when a chip is dropped outside $T_e$. Since the $v_i$s are a DFS preorder with respect to $r$, there are indices $i$ and $j$ for which $V(T \setminus T_e) = \{v_1, \ldots, v_i, v_j, \ldots, v_n\}$.

During the firings of these vertices, net firings can only enter $T_e$. Therefore, only $|V(T_e)|!$ net firings away from $r$ can occur across $e$ when firing $v_1, \ldots, v_i$. $T_e$ will send back a chip for every chip inserted into $T_e$, so there can be no new net firings when it is full. Applying the
same reasoning for \( v_j, \ldots, v_n \) shows that there are no more than \( 2|V(T_e)| \) downward net firings across \( e \) over the course of Algorithm 2.

Now, we sum up the lower bounds for all \( e \in F \). This bound is just

\[
\sum_{e \in F} 2|V(T_e)| = \sum_{e \in F} \sum_{v \in V(T_e)} 2 = \sum_{v \in V(T)} \sum_{e \in F \text{ that are rootward of } v} 2 \leq 2nD
\]

so \( m_d \leq 2nD \). This makes the total number of net firings at most \( 5nD \).

4 Accelerating the round-based algorithm using data structures

For a balanced binary tree, \( D = O(\log n) \) and \( F = T \), so Algorithm 2 takes \( O(n \log n) \) time. Unfortunately, though, trees do not have to be anywhere near balanced. Paths are the extreme case, in the sense that \( D = n \). Algorithm 1 takes linear time per chip drop on a path. On a path, though, one can simulate chip additions in \( O(1) \) time per chip drop. Ideally, we could combine this path-based speedup with the fact that low-diameter trees take little time.

In this section, we do this with fast data structures. The data structures speed up the simulation of rounds, with time corresponding to the round number. It helps decompose a tree into paths and to view each path of the path tree as a path with subtrees attached to vertices on the path. We need a data structure that can do the following:

- update elements in a subtree by “acceleration terms” depending on distance
- find the minimum element with key less than some value

The minimum element will correspond to the branch that is closest to being able to take in another chip. Call these times events. The acceleration terms update the rest of the tree with what happened between consecutive events.

Unfortunately, we are not able to implement both of these operations efficiently. Luckily, though, we only need to find approximate minima. We can charge the fake events (events that are not true minima) to a multiplicative reduction in the size of a branch. This ensures that the amortized runtime of finding the minimum is \( O(\text{polylog} n) \).

We will separate the data structures required into two separate data structures. The first will keep track of the real position of the leaf of the current critical subtree in each heavy path of the tree data structure. The second data structure will efficiently return approximate minima.

4.1 Decomposing a tree into paths We refer to the decomposition of a general rooted tree into a tree of heavy paths. The root of a heavy path, \( \text{Root}(p) \), is the endpoint of \( p \) that is closer to the root of the input tree. The parent of a path, denoted \( \text{Parent}(p) \), is the path containing \( \text{Root}(p) \). Recall that the heavy-light decomposition of a tree \( T \) [16], for every vertex \( u \), defines a heavy edge to a child \( x \) of \( u \) to be a edge for which \( |V(T_x)| > |V(T_u)|/2 \), where \( T_y \) is the subtree of \( T \) rooted at \( y \). If a vertex \( u \) has no heavy edges, pick one arbitrarily to be the heavy edge for \( u \). All other edges of \( T \) are called light edges. For two vertices \( a, b \) on a heavy path \( p \) with \( b \) leafward of \( a \), let \( b - a \) denote the distance between \( a \) and \( b \) on the path (and also in the tree).

Let \( F \) be the set of light edges of the heavy-light decomposition of \( T \). Notice that the diameter of the tree \( T \) with respect to \( F \) is at most \( 2 \log n \). By Lemma 3.1, the number of net firings across \( F \) is at most \( O(n \log n) \). Therefore, we just need to design a data structure that only needs to do \( \text{polylog}(n) \) work each time a net firing crosses an edge in \( F \).

4.2 The exact data structure We now give a data structure that will keep track of the exact positions of the leaves of the current critical subtree. For each heavy path \( q \), \( \text{AdvanceTime}(dt) \) modifies the leaves of \( q \)’s child paths using a hinge function, where the slope 1 part of the hinge function has width \( dt \). We now give the interface of the data structure and defer proofs to the appendix:

- SetupExact(T): Sets up the data structure on the tree \( T \) with no chips.
- MoveChip\((p_1, p_2)\): Inserts a chip from \( p_1 \) into a child heavy path \( p_2 \) and updates all timers appropriately. Assumes that the insertion is valid.
- AdvanceTime\((dt)\): Advances time by \( dt \) rounds. Assumes that no chips move across light edges during the rounds in between.
- Reroot\((s)\): Reroots the tree at a vertex \( s \). Chips are always added at the root of the current tree.
- LeafInPath\((p)\): Returns the vertex in the heavy path \( p \) that is the leaf of \( p \) in the current critical subtree.
Lemma 4.1. All functions in the exact data structure (Algorithm 4) produce the correct outputs under the assumption that MoveChip is called on the next correct event. Moreover, all operations besides Setup and DumpConfiguration take \( O(\log^2 n) \) worst case time, with those operations taking \( O(n \log n) \) time.

4.3 The approximate data structure We would really like to add a subroutine to the exact data structure that computes the heavy path with leaf closest to the its root. Unfortunately, adding accelerations to vertices and being able to find minima seem incompatible.

Luckily, though, there is a technique in the data structures literature that makes it possible to keep track of minima after adding velocities (diffs). We can approximate the effect of adding accelerations by a step function of approximations to the correct amounts. This data structure will always store multiplicative overapproximations to the real amount of time that has elapsed on a heavy path. This ensures that any event that actually occurs will not be missed by the approximate data structure. Using a step function ensures that we can implement each approximate data structure update using \( O(\log n/\delta) \) diff modifications.

Our approximate data structure has the following interface:

- **SetupApproximate**\( (T, \delta) \): sets up the approximate data structure on the tree \( T \) with multiplicative error \( (1 + \delta) \)

- **TryMove()**: Tries the next possible move and does it if it is valid. It picks the candidate move as follows. The approximate data structure maintains values \( n_p \) with the following \( n_p \) approximation property:

\[
x_p - \text{LeafInPath}(p) \leq x_p - n_p \leq (1 + \delta)(x_p - \text{LeafInPath}(p))
\]

where \( x_p \) is the position of the leaf of the critical component in \( p \) after the previous approximate data structure update to the path \( p \). **TryMove** then finds the solution to the following optimization problem:

\[
\min_{\text{paths } p} n_p \\
\text{s.t.} \\
\forall \text{ ancestor paths } q \text{ of } p : n_p + d_{pq} \leq \text{LeafInPath}(q)
\]

We now define \( d_{pq} \). Let \( u_{pq} \) denote the closest ancestor in \( T \) to \( \text{Root}(p) \) on \( q \). If one replaces \( n_p \) with \( \text{LeafInPath}(p) \) and \( d_{pq} \) with \( \text{dist}(u_{pq}, \text{Root}(q)) \), then this optimization finds the next net-firing light edge. The constraint ensures that \( \text{Root}(p) \) is adjacent to a vertex in the current critical component when a chip is added.

We do not actually define \( d_{pq} \) this way, as solving the resulting optimization problem becomes time consuming. Instead, we solve the problem with \( d_{pq} \) equal to some value with the following \( d_{pq} \) approximation property:

\[
x_q - \text{dist}(u_{pq}, \text{Root}(q)) \leq x_q - d_{pq} \leq (1 + \delta)(x_q - \text{dist}(u_{pq}, \text{Root}(q)))
\]

**TryMove** then checks the exact data structure to assess whether or not a move is valid. When **TryMove** processes a real move, it increments time by \( n_p \). Otherwise, it increments time by at least \( n_p/(1 + \delta) \).

**TryMove()** returns MOVE-EXISTS if and only if the critical component is not empty.

- **FastDropChip**\( (v) \): Drops a new chip at \( v \).

Lemma 4.2. Calls to **TryMove** and **FastDropChip** maintain the guarantees and each take \( O(\frac{1}{\delta} \log^3 n) \) time per call in the worst case.

---

Algorithm 3 FastDFS\( (T, \gamma) \)

**Input**: tree \( T \), initial configuration \( \gamma \)

**Output**: terminal configuration \( \sigma \)

1: \( r \leftarrow \) arbitrary vertex of \( v \) which is chosen to be the root of \( T \)
2: \( v_1, v_2, \ldots, v_n \leftarrow \) DFS preorder of the vertices of \( T \) starting with \( v_1 = r \)
3: \( \sigma \leftarrow \) all zeros vector on \( V(T) \)
4: \( A \leftarrow \) SetupApproximate\( (T, r, 1/2) \)
5: for all \( i = 1 \) through \( n \) do
6:  for all \( j = 1 \) through \( \gamma_{v_i} \) do
7:    FastDropChip\( (A, v_i) \)
8:  while **TryMove**\( (A) \) is MOVE-EXISTS do
9:    end while
10: end for
11: end for
12: return DumpConfiguration\( (A) \)

4.4 The full algorithm We split the analysis of this algorithm into two parts: correctness (Lemma 4.3) and
runtime (Lemma 4.4). Lemma 4.4 depends on Lemma 4.3 but not vice versa. We defer the proof of 4.3 to the appendix.

**Lemma 4.3.** FastDFSDrop computes the correct terminal configuration for a tree.

At a high level, the correctness proof shows that 1) all real events appear in the correct order and that 2) the algorithm never increments time past the occurrence of the next event. Both of these follow from the lower bounds of the approximation property for \( n \) and both bounds on the approximation property for \( d_{pq} \).

The desired runtime bound of \( O(n \log^3 n) \) follows immediately from the fact that each TryMove call takes \( O(n \log^3 n) \) time given Lemma 4.4.

**Lemma 4.4.** FastDFSDrop makes \( O(n \log^2 n) \) calls to TryMove and FastDropChip.

**Proof.** To bound the number of TryMove calls, it suffices to bound the number of fake events that occur. We charge each fake firing to a real firing and show that each real firing has \( O(\log n) \) fake firings charged to it. We use the upper bound of the invariant on each \( n_p \) in Lemma 4.2. Consider a path \( p \) along with the edge \( e_p \) from its parent. Real events happen when the size of \( p \) is 0. We now show that between any two fake events on \( e_p \), LeafInPath\(^1(p) \) decreases by a factor of 3. Let superscripts of 0 and 1 on all variables denote the value of the variable immediately before and after \( p \) is updated due to a particular fake event over \( e_p \) respectively. By the \( n_p \) approximation property,

\[
x_p - n_p^0 \leq (1 + \delta)(x_p - \text{LeafInPath}^0(p))
\]

If time were incremented by \( n_p \), then \( n_p^1 = 0 \). Since the feasibility constraint in TryMove is relaxed, it is possible for \( n_p^1 \) to be greater than 0. By Lemma 4.2 we know that time is advanced by at least \( n_p^0/(1 + \delta) \), which means that

\[
\text{LeafInPath}^1(p) \leq \text{LeafInPath}^0(p) - \frac{n_p^0}{1 + \delta} \\
\leq \frac{1}{1 + \delta}(n_p^0 + \delta x_p) - \frac{n_p^0}{1 + \delta} \\
= \frac{\delta}{1 + \delta} x_p
\]

When a path \( p \) is visited in TryMove, it is updated to reflect its current state in the exact data structure. As a result, when a fake event occurs, \( x_p \) will be reinitialized to \( \text{LeafInPath}(p) \), which is at most \( \delta/(1 + \delta) \) times its previous value. Since \( \delta = \frac{1}{2} \), \( x_p \) decreases by a factor of 3. \( x_p \) only increases when real events happen. Therefore, between any two real events, only \( \log_3 n \) fake events can occur. Recall that there are \( O(n \log n) \) real events by Lemma 3.1. Therefore, there are at most \( O(n \log^2 n) \) fake events, which translates to \( O(n \log^2 n) \) TryMove calls.

**Acknowledgements** We thank Nikhil Srivastava for introducing this problem to us and for many helpful discussions. We also thank Hannah Cairns, Xiang Cheng, Alex Rusciano, and anonymous reviewers for helpful edits.

**References**


\[\text{Copyright © by SIAM}\]

Unauthorized reproduction of this article is prohibited

A Exact data structure implementation and correctness

We now give a data structure that proves Lemma 4.1. The key idea is that it suffices to keep track of how many rounds (how much time) have elapsed on each heavy path. Associate a time zone with each heavy path \( p \). The time zone of a heavy path is an object that stores the following five items:

- An integer \( r_p \) indicating the number of rounds in which the critical subtree has intersected the heavy path \( p \).
- An integer \( s_p \) indicating the previous round number in which a chip was added to this heavy path.
- An integer \( t_p \) indicating the location of the root-most subcritical vertex immediately after round number \( s_p \), or \( \infty \) if the entire path is critical.
- A stack of subcritical vertices \( S^p \) excluding \( \text{LeafInPath}(p) + 1 \) with the order of vertices from top to bottom being from root to leaf of the heavy path. Locations are specified by distances from \( \text{Root}(p) \).
- A data structure of accelerations \( A^p \) for the vertices on \( p \).

The data structure represents \( s_p, t_p, \) and \( S^p \) explicitly, while it represents \( r_p \) in terms of a sequence of diffs \( r_p = r_{\text{Parent}}(p) \), which we call velocities. For a path \( p \), the diff \( r_p = r_{\text{Parent}}(p) \) is stored implicitly in the data structure \( A_{\text{Parent}}(p) \). Between two firings across light edges, for a path \( q \), the diffs \( r_q' - r_q \) for children \( q' \) of \( q \) change by a piecewise linear function that consists of two constant parts with a slope -1 part in between. This slope -1 part arises from the fact that time elapses in the path \( q' \) only when \( \text{LeafInPath}(q) \) is leafward of the neighbor of \( q' \) in \( q \), and \( \text{LeafInPath}(q) \) moves towards the root between two updates. These updates can be made using accelerations of +1 and -1 at the starting and stopping points of \( \text{LeafInPath}(q) \) respectively. Figure 3 gives an example of this.

\( r_p \) can be computed by summing up the diffs in \( A^q \) for all ancestors \( q \). The current leaf of the critical subtree in \( p \) is \( t_p - (r_p - s_p) - 1 \). Therefore, if we can dynamically maintain \( r_p \), we can also maintain the leaves of the critical subtree after each round.

The following data structure for the accelerations can be implemented using diffs on the internal nodes of a binary tree. It is given in Appendix D.

- **Setup**(\( p, r \)): sets up an acceleration data structure for a set of vertices on a heavy path \( p \) with root \( r \)
- **ChangeAcceleration**(\( v, \Delta \)): changes the acceleration \( a_v \) of a vertex \( v \in p \) by \( \Delta \)
- **ChangeVelocity**(\( v, \Delta \)): changes the velocity \( v_o \) of a vertex \( v \) by \( \Delta \)
• **Velocity**($u$): outputs the current velocity of $u$ due to the accelerations in the current data structure. More precisely, the velocity of $u$ is $\sum_x \text{rootward of } u (a_x d(x, u) + v_x)$.

• **Reroot**($s$): reroots the data structure at a vertex $s \in V(p)$.

We now give an explicit construction of the Lemma 4.1 data structure given the above data structure:

**Algorithm 4** Exact data structure part 1

1. tree $T$ of heavy paths
2. a vertex $r$ in the tree
3. **function** `SetupExact(T', r')`
   **Input:** rooted tree $T'$ with root $r'$
4. $r \leftarrow r'$
5. $T \leftarrow \text{HeavyPaths}(T', r)$
6. for all heavy path node $p \in T$ in BFS order do
7.   $s_p \leftarrow 0$
8.   $t_p \leftarrow \text{Root}(p)$
9.   $S^p \leftarrow $ vertices of the path in sorted order with the rootmost endpoint on top, with degree($v$)−1 copies of each vertex $v$ in the stack and $\text{Root}(p)$ popped off
10. $A^p \leftarrow \text{Setup}(p)$
11. `ChangeAcceleration(A^p, \text{Root}(p), +1)`
12. end for
13. **end function**

14. **function** $R(p)$
   **Input:** heavy path $p$
   **Output:** current value of $r_p$
15. if $p$ is the root heavy path $\{r\}$ then
16.   **return** $\text{Velocity}(A^{\{r\}}, r)$
17. else
18.   $q \leftarrow \text{Parent}(p)$
19.   $l_q \leftarrow \text{LeafInPath}(q)$
   \> $l_q$ acts as a moving -1 acceleration term, whose effect is given by $z$
20.   $y \leftarrow $ vertex of $q$ adjacent to $\text{Root}(p)$
21.   $z \leftarrow \max(0, y - l_q)$
22.   **return** $R(q) + \text{Velocity}(A^q, y) - z$
   \> $\text{LeafInPath}$ and $R$ are computed with memoization
23. **end if**
24. **end function**

25. **function** $\text{LeafInPath}(p)$
   **Input:** heavy path $p$
   **Output:** current position of the leaf of the critical sub-
tree, with $\text{Root}(p) - 1 = -1$ denoting no intersection with the critical subtree
26. **return** $t_p - (R(p) - s_p) - 1$
27. **end function**

28. **function** $S(p)$
29. **return** $s_p$
30. **end function**

31. **function** $T(p)$
32. **return** $t_p$
33. **end function**

34. **function** $\text{AdvanceTime}(dt)$
35. `ChangeVelocity(A^{\{r\}}, r, dt)`
36. **end function**
Algorithm 5 Exact data structure part 2

function MOVECHIP($p_1, p_2$)

Input: heavy path $p_1$ and a child heavy path $p_2$

1. $q_1, q_2, \ldots, q_k = p_1 \leftarrow$ the path in $T$ from the root heavy path to $p_1$
   $\triangleright$ calculates amount of time between previous
   event and current event in $q_1$'s time zone
2. $dt \leftarrow$ LeafInPath($p_2$) + 1
3. $\triangleright$ other changes
4. AdvanceTime($dt$)
5. ChangeAcceleration($A^{p_2}, \text{Root}(p_2) - 1, -1$)
6. $s_{p_2} \leftarrow R(p_2)$
   $\triangleright$ sets $t_{p_2}$ to be $\infty$ if nothing is in the stack
7. $t_{p_2} \leftarrow$ Pop($S^{p_2}$)
8. ChangeAcceleration($A^{p_2}, t_{p_2} - 1, +1$)
9. if $p_2$ is not the trivial root path of $T$ then
10.   $l_{p_1} \leftarrow$ LeafInPath($p_1$)
11.   $s_{p_1} \leftarrow R(p_1)$
12.   $t_{p_1} \leftarrow$ neighbor of Root($p_2$) in $p_1$
13.   Push($S^{p_1}, l_{p_1}$)
14.   ChangeAcceleration($A^{p_1}, l_{p_1}, -1$)
15.   ChangeAcceleration($A^{p_1}, t_{p_1} - 1, +1$)
16. end if
17. end function

Lemma A.1. (Restatement of Lemma 4.1)
Algorithm 4 produces the correct outputs under the assumption that MoveChip is called on the next correct
event. Moreover, Setup and DumpConfiguration take $O(n \log n)$ time and all other operations take $O(\log^2 n)$
time.

Proof. We focus on correctness, as the runtime follows
from the fact that the depth of the tree is $O(\log n)$ and
the fact that the low-level data structure has runtime
$O(\log n)$ per operation. Correctness relies on the fact
that between two events in a subtree, the leaf moves
towards the root at a rate of 1 step per iteration. It is
also important to think of the leaf as a temporary
acceleration of -1.

We prove the correctness by induction on the num-
ber of MoveChip calls. If there have been no move chip
calls, only SetupExact has been called. No chips are
present, so no time has elapsed and all initial values
are correct. LeafInPath will output -1 for each path,
which is correct because there are no critical vertices in
any path. $R$ will output 0 for every path, which is also
correct because no time has elapsed anywhere.

Suppose inductively that the data structure is cor-
rect at some state and do a MoveChip($p_1, p_2$) call. We
can assume that this is a valid MoveChip call. We
account for the global passage of time at $r$ by $dt$
rounds with the AdvanceTime call in MoveChip. After accounting
for this, the leaf changes location in $p_1$ and $p_2$ only.
The ChangeAcceleration calls on $p_2$ make the tempo-
ral acceleration of -1 on the old leaf (Root($p_2$) - 1)
permanent and counteract the new temporary acceleration
of -1 on the new leaf ($t_{p_2} - 1$). The third and
fourth ChangeAcceleration calls reflect the fact that
LeafInPath moved from $t_{p_1}$ to $t_{p_1}$. This completes the
proof that the velocity data structures, when combined
with a -1 acceleration from the leaf, compute the correct
differences between the $r_{p_1} - r_{p_2}$ for every parent-child
pair ($p_1, p_2$).

We now show that our updates to $S^{p_1}$ and $S^{p_2}$ are
correct. Since a chip is added to $p_2$, its first gap (at
Root($p_2$)) is filled, which makes the next value of $t_{p_2}$
the top of the stack. Similarly, $p_1$ gained a fixed gap at
$l_{p_1}$. This shows that stack updates are correct.

Now, we just need to show that $R(p)$ and
LeafInPath compute the correct values after this
MoveChip call. $R(p)$ is correct because the algorithm
is representing it as a sum of the velocities of the an-
ccestor paths along with -1 accelerations from the an-
cestral LeafInPaths. LeafInPath is correct given the
correctness of $R(p)$ because the leaf location after round
$s_{p}$ is $t_{p} - 1$. Therefore, the correctness of $R(p)$ and
LeafInPath($p$) follows from the correctness of the veloc-
ity data structure. This completes the inductive step.
Algorithm 6 Exact data structure part 3
1: function Reroot(s)
Input: new root s
\[\text{\triangleright Undo current root choice } r\]
2: if r broke up a heavy path in the initial heavy path decomposition into two paths \(p_1\) and \(p_2\), with \(p_1\) closer to s then
3: merge \(p_1\) and \(p_2\) back into a single path \(p\)
4: \(x \leftarrow \) closest endpoint of \(p_1\) to s
5: \(t'_p \leftarrow \text{LeafInPath}(p_1)\)
6: \(t'_p \leftarrow \text{LeafInPath}(p_2)\)
7: \(\text{Reroot}(A^{p_1}, x)\)
8: \(r_x \leftarrow R(p_1)\)
\[\text{\triangleright } r_x = \text{time elapsed so far at vertex } x\]
9: \(A^p \leftarrow A^{p_2}\) appended to \(A^{p_1}\). Do this by connecting the binary trees for \(A^{p_1}\) and \(A^{p_2}\) to a common root vertex with edge weights 0 and distance\((r, x)\) respectively
10: \(\text{Push}(S^{p_1}, t'_p)\)
11: \(\text{Push}(S^{p_2}, t'_p)\)
12: \(\text{Reverse}(S^{p_1})\)
13: \(S^q \leftarrow S^{p_1}\) pushed onto \(S^{p_2}\)
14: \(t_p \leftarrow \text{Pop}(S^{p})\)
15: \(s_p \leftarrow r_x\)
16: end if
\[\text{\triangleright Pick new root choice } s\]
17: \(q \leftarrow \text{the heavy path containing } s\)
18: \(q_1, q_2 \leftarrow \text{the paths obtained by removing an edge adjacent to } s \text{ from } q\)
19: reverse \(A^{q_1}\) and split the stack \(S^{q}\) into two stacks \(S^{q_1}\) and \(S^{q_2}\) with one reversed in \(O(1)\) time
20: assign \(t_{q_1}\) and \(t_{q_2}\) based on the top element of the new stacks
21: assign \(s_{q_1}\) and \(s_{q_2}\) to the current time
\[\text{\triangleright Fix all paths } a \text{ between } p \text{ and } q \text{ in } T\]
22: reverse \(A^a\)
23: push \(\text{LeafInPath}(a)\) onto \(S^a\), reverse the stack, and pop the top element to get \(t_a\)
24: assign \(s_a\) to the current time
25: \(r \leftarrow s\)
26: end function

function DumpConfiguration
27: for all vertex \(v \in T'\) do
28: \(\sigma_v \leftarrow \text{degree}(v) - 1 - \) the number of copies of \(v\) in \(S^{p} \cup \{\text{LeafInPath}(p) + 1\}\), where \(p\) is the heavy path containing \(v\)
29: end for
30: return \(\sigma\)
31: end function

and completes the proof that all methods continue outputting the correct value after each \text{MoveChip} call.

Now, we reason about each \text{Reroot} call. Notice that only \(O(\log n)\) paths are altered and that each stack reversal/split/low-level data structure modification can be implemented in \(O(\log n)\) time, for a total of \(O(\log^2 n)\). Correctness follows from the fact that no \(R(b)\) values change for any heavy path \(b\) that is not on the path between \(r\) and \(s\).

Finally, we show the correctness of \text{DumpConfiguration}. It suffices to notice that a chip is added to or removed from a vertex during an event when it is popped off or pushed onto its corresponding stack respectively. The only other gaps are the ones that form during nonevents, which are accounted for by \text{LeafInPath}(p) + 1 (the leafward neighbor of \text{LeafInPath}(p)).

B Approximate data structure implementation and correctness

Let \(L = O(\log n)\) be the depth of the heavy path decomposition. Our data structure keeps track of the exact data structure \(E\), a diff-minimization data structure \(F\), and the following information for each heavy path \(p\):

- \(x_p\): the position of the leaf on the previous approximate data structure update to \(p\)
- \(y_p\): the time of the previous update according to \(p\)'s time zone
- \(n_p\): a number associated with \(p\) that has the following property after any update to \(p\) or any descendant of \(p\):

\[x_p - \text{LeafInPath}(p) \leq x_p - n_p \leq (1+\delta)(x_p - \text{LeafInPath}(p))\]

- \(m_p\): the minimum of all descendant \(n_p\) values with the amount of time that has elapsed subtracted off. \(n_p\) and \(m_p\) are stored implicitly in the data structure \(F^p\)
- \(p_p\): the heavy path \(q\) for which \(m_q = n_q\)
- \(F^p\): a data structure that allows for diffs and minimization. This data structure implicitly stores the \(m_p\) values for all children

Each \(F^p\) has the following interface, which can be implemented using a typical diff tree data structure:

- \text{Setup}(p, r): initializes a data structure based on \(p\) with root \(r\)
- ChangeVelocity\((u, v, \Delta)\): changes the value of all vertices in the path \(p\) between \(u\) and \(v\) by \(\Delta\)
- ChangeValue\((v, D)\): sets the value of vertex \(v\) to \(D\)
- GetMinInRange\((u, v)\): returns the minimum between vertices \(u\) and \(v\)
- Reroot\((s)\): reroots the entire data structure at a vertex \(s\)

We now implement the desired data structure:

**Algorithm 7** Approximate data structure part 1

1: tree \(T\) of heavy paths
2: accuracy parameter \(\delta\)
3: exact data structure \(E\)
4: diff data structure \(F\)
5: for each path heavy path \(p\)
6: exact location of the previous leaf \(x_p\) when \(p\) was previously updated
7: previous update times \(y_p\)
8: function SetupApproximate\((T', r', \delta)\)
   9: \(E \leftarrow \text{SetupExact}(T', r')\)
10: \(T \leftarrow \text{same heavy paths decomposition as in } E\)
11: set everything else to 0
12: setup all \(F_p\)s with root at the rootmost vertex of \(p\)
13: end function
14: function ApxAccelerate\((p, u, v\)\)
   15: \(\Delta + 1 + \delta\)-overapproximate the effect of putting -1 acceleration on \(s\) and +1 acceleration of \(t\) using a \(O(\log n/\delta)\)-sparse step function for the velocities
   16: \(k \leftarrow \lceil \log_{1+\delta} \text{dist}(u, v) \rceil\)
   17: \(v_0 \leftarrow u\)
   18: \(v_k \leftarrow u\)
   19: for all \(i \in 1, \ldots, k\) do
      20: \(v_i \leftarrow \text{vertex on } p\) with distance \(\lfloor (1 + \delta)^i \rfloor\) from \(v\) if \(i \neq k\)
      21: \(\text{ChangeVelocity}(F_p, v_i, v_{i-1}, \lfloor (1 + \delta)^{i+1} \rfloor)\)
   22: end for
   23: \(\text{ChangeVelocity}(F_p, \text{Root}(p), u, d(u, v))\)
24: end function
25: function GetMinToPropagate\((p)\)
   26: \(\text{Returns minimum of the branches of } p\) subject to a relaxation of the constraint \((\text{branch } + \text{ distance}) \leq \text{LeafInPath}(p)\)
   27: \(v_0 \leftarrow \text{LeafInPath}(p)\)
   28: \(m \leftarrow v_0\)
   29: \(p_{\text{min}} \leftarrow p\)
   30: for all \(i = 1, 2, \ldots, k = \lceil \log_{1+\delta} d(v_0, \text{Root}(p)) \rceil\) do
      31: \(v_i \leftarrow \text{rootward vertex on } p\) with distance \(\lfloor (1 + \delta)^i \rfloor\) from \(v_0\) if \(i \neq k\)
      32: \((q, m_i) \leftarrow \text{GetMinInRange}(F_p, v_i, v_{i-1})\)
         \(\text{exact version of the following constraint would replace } v_i \text{ with the neighbor of } \text{Root}(q)\)
      33: if \(m_i + \text{dist}(v_i, \text{Root}(p)) \leq \text{dist}(v_0, \text{Root}(p))\) then
         34: \(m \leftarrow \text{min}(m, m_i)\)
         35: update \(p_{\text{min}}\) if \(m\) changed to \(q\)
      36: end if
   37: end for
   38: return \((m, p_{\text{min}})\)
39: end function
We now prove the correctness and bound the runtime of this data structure:

**Lemma B.1. (Restatement of Lemma 4.2)**

*TryMove* and *FastDropChip* are correct and each take \(O(\frac{1}{\delta} \log^3 n)\) worst-case time.

**Proof.** The runtimes of each method follow from the fact that operations in the exact data structure are called \(O(\log n)\) times per *TryMove* and *FastDropChip* call and that data structures are called \(O(\log^2 n)\) times. It is also very important that the acceleration updates are done using \(O(\frac{1}{\delta} \log n)\) velocity changes. For the remainder of the proof, we focus on correctness.

First, we discuss *TryMove*. The return value is correct because events happen until the root is subcritical. We need to show that all variables are correctly maintained after each *TryMove* call. We start by showing that when a path \(p\) is visited, \(n_p\) satisfies the desired invariant whenever \(n_p\) is "seen" by the algorithm; that is whenever \(p = p_x\) for some visited path \(x\). Recall that each \(F^p\) represents the \(n_q\)'s for all child paths \(q\) of \(p\). First, we show the left hand side inequality; that is

\[
x_p - \text{LeafInPath}(p) \leq x_p - n_p
\]

Suppose that \(n_p\) is seen as \(m_q\), for some ancestor \(q_1\) of \(p\). Let \(q_0 = \text{Parent}(q_1)\). Whenever \(p\) is visited, \(x_p\) and \(n_p\) are both reset to \(\text{LeafInParent}(p)\) if \(n_p\) is the minimum, because \(x_p\) is explicitly set to that in *TryMove* and \(n_p\) is set to it implicitly (see \(m \leftarrow v_0\) in *GetMinToPropagate*). When \(q_0\) is visited, the value \(m_{q_1}\) is decremented in *ApxAccelerate* by an overapproximation to the true amount, since the true amount is bounded above by \((1 + \delta)^{i+1}\). This completes the proof of the desired inequality.

To show that

\[
x_p - n_p \leq (1 + \delta)(x_p - \text{LeafInPath}(p))
\]

it suffices to notice that the \(x\) change in *ApxAccelerate* is at most \((1 + \delta)\)-overapproximation to the true change. The correctness of all other variables follow from their resetting throughout the code.

The correctness proof for *FastDropChip* is similar to the correctness proof for *Reroot* in the exact data structure, because dropping a chip is equivalent to rerooting and moving a chip from the root to the child path.

**C Proof of Lemma 4.3**

**Proof.** Let \(F\) be the set of light edges in the heavy-light decomposition. First, we show that *FastDFSDrop* computes the correct final configuration. To do this, it
Algorithm 9 Approximate data structure part 3

function FastDropChip(s)
    update all paths between r and s using ApxAccelerate and GetMinToPropagate
    reverse all data structures $F^p$ for those paths
    Reroot($E, s$)
    MoveChip(s, child path of s)
end function

function DumpConfiguration()
    return DumpConfiguration(E)
end function

suffices to notice that TryMove executes only the chip moves that happen across F.

Notice that no invalid chip move occurs because TryMove checks the validity of any candidate move using the exact data structure. If $dt = \text{LeafInPath}(p)$ and $\text{Root}(p)$ is in the critical subtree after $dt - 1$ rounds, then the move from $\text{Parent}(p)$ to $p$ is valid for the following reasons. First, $\text{Root}(p)$ will no longer be in the critical subtree after $dt$ rounds. Second, $\text{Root}(p)$ is adjacent to a vertex in the critical subtree after $dt$ rounds thanks to the second condition. This means that a chip will move across the light edge connecting $\text{Root}(p)$ to its parent in $T'$. Therefore, these conditions suffice to ensure that no fake event triggers the first if statement in TryMove, which is the only place in which the exact data structure is changed. Each fake event violates one of these conditions, so the if statement is triggered if and only if an event is real.

We now show that all real events occur in the right order. It suffices to show this inductively. Assume that real event $i$ just happened. By Lemma 4.2, all $n_p$ values are underapproximations to their true value. By the first condition of the if statement of TryMove, real events will satisfy equality, that is $n_p = dt = \text{LeafInPath}(p)$. Therefore, no real event can occur before real event $i + 1$. It now suffices to show that each real event occurs. To do this, we need to show that each real event satisfies the relaxed feasibility condition in the comment of GetMinToPropagate. For a feasible event on a path $q$, notice that

$$\text{LeafInPath}(q) + \text{dist}((\text{Root}(\text{Parent}(q)), \text{Root}(q) - 1) \leq \text{LeafInPath}(\text{Parent}(q))$$

because the leaf of $q$ needs to be a part of the critical subtree when the event for $q$ triggers. Since $d(\text{Root}(\text{Parent}(q)), \text{Root}(q) - 1) \geq d(\text{Root}(\text{Parent}(q)), v_i)$, the condition given in GetMinToPropagate is a relaxation of the true condition and any real event will be captured by the minimization in GetMinToPropagate.

Finally, we need to show that when a (fake or real) event for $q$ triggers, we need to ensure that the leaf on any ancestor path does not pass rootward of it. This is why $dt \leftarrow dt/(1 + \delta)$ in the else if of TryMove. The relaxed feasibility constraint has the property that for any event triggered on a path $q$, and any ancestor $q'$:

$$\text{dist}(\text{Root}(q), \text{LeafInPath}(q)) \leq \text{dist}(v_i, \text{LeafInPath}(q')) \\ \leq (1 + \delta)\text{dist}(\text{Root}(\text{Parent}(q)), \text{LeafInPath}(q'))$$

when combined with the assumption that $q$ is indeed a minimizer over $\text{Parent}(q)$. Therefore, decrementing time by $\text{dist}(x, \text{LeafInPath}(q))/(1 + \delta)$ ensures that $q$ will always be in the critical subtree when an event happens. This completes the proof of correctness.

D Low-level data structures

D.1 Diff data structure with minimization This data structure is essentially the same as the one given in Chapter 17 of [10], but we describe it here for completeness. It keeps a balanced binary tree with each leaf node of the tree representing a vertex on the path $p$ supplied to Setup($p, r$). Each node stores the following information:

- $\Delta_v$: $x_v - x_{\text{parent}(v)}$, where $x_v$ is the value of vertex $v$, which is also GetValue$(v)$
- $\Delta\text{min}_v$: $\min_v - x_v$, where $\min_v$ is the minimum value of any vertex in the subtree rooted at $v$.
- $v.(\text{start, end})$: specifies the interval of vertices in the subtree rooted at this node
- $v.(\text{left, right})$: the two children of $v$

The minimum of a subtree can be defined recursively using $\min_v = \min(x_v, \min_{v.l}x, \min_{v.r}x)$. Subtracting $x_v$ from both sides shows that $\Delta\text{min}_v = \min(0, \Delta\text{min}_{v.l} + \Delta\text{min}_{v.r})$ which allows us to compute $\Delta\text{min}_v$ recursively solely in terms of the $\Delta$s. This allows us to do each operation in $O(\log n)$ time.

D.2 Heavy light decomposition Here, we implement the HeavyPaths function. This function takes as input a tree $T$ and a root $r$ and outputs a tree $T'$, where the vertices of $T'$ are paths in $T$ and edges in $T'$ are edges in $T$. $r$ is part of a one vertex path.
Algorithm 10 Acceleration data structure

1: function Setup(p, s)
2: mid ← (length of p)/2
3: if mid is 0 then
4: return makeTree( (dist,(acceleration,velocity)) is (p[0],(0,0)) )
5: end if
6: t ← makeTree( (key, (acc, vel)) is (p[0], (0,0)) )
7: if mid > 0 then
8: t.left ← Setup(p[:mid-1],s)
9: t.right ← Setup(p[mid:],r)
10: end if
11: return t
12: end function

13: function ChangeAcceleration(v, ∆)
14: Node c ← t.find(v)
15: while c.parent.key is v → in the bottom level of the tree
16: c ← c.parent
17: end while
18: c.changeAcceleration(∆)
19: while c is not t.root do
20: while c.key is not c.parent.key → go up and left as far as possible
21: c ← c.parent
22: end while
23: while c.key is c.parent.key → go up and right one at a time
24: c ← c.parent
25: c.right.changeVelocity(∆) · (c.right.key-v)
26: end while
27: end while
28: end while
29: end function

30: function ChangeVelocity(v, ∆)
31: Node c ← t.find(v)
32: c.changeVelocity(∆)
33: end function

34: function Velocity(v)
35: return t.find(v).getVelocity()
36: end function

37: function Reroot(s)
38: Move root to the opposite side
39: Flip the sign of all accelerations
40: end function

Algorithm 11 Diffs with minimization

1: function Setup(p,r)
2: Input: r always represents one side of p
3: mid ← (length of p)/2
4: t ← makeTree( ((start,end),(∆v,∆minv)) = ((p.first,p.last),(0,0)) )
5: if mid > 0 then
6: t.left ← Setup(p[:mid])
7: t.right ← Setup(p[mid+1:])
8: end if
9: return t
10: end function

11: function GetValue(u)
12: return ∑ ancestors a of (u) ∆a
13: end function

14: function ChangeValue(u,x)
15: ChangeVelocity(u,u,x - GetValue(u))
16: end function

17: function Decompose(u,v)
18: if u is v then return t.findNode((u,u))
19: (u,w) ← root of largest subtree starting at u where w ≤ v
20: return (u,w), Decompose((w,v))
21: end if
22: end function

23: function ChangeVelocity(u,v,d)
24: dec ← Decompose(u,v)
25: for all root nodes r in dec do
26: ∆r ← ∆r + d
27: for all ancestors v of r do
28: ∆minv ← ∆minv - ∆v.left + ∆minv.left + ∆v.right + ∆minv.right
29: end for
30: end for
31: end function

32: function GetMinInRange(u,v)
33: dec ← Decompose(u,v)
34: return min nodes r in dec (GetValue(r) + ∆minr)
35: end function

36: function Reroot(s) → doesn’t need to do anything, as diffs are always applied to a proper subpath
37: end function

Copyright © by SIAM
Unauthorized reproduction of this article is prohibited
Algorithm 12 Heavy light decomposition of an $r$-rooted tree $T$

1: function HeavyPathsImpl($T, r, \text{counts}$)
2: if $V(T) = \{r\}$ then
3: return $T$
4: end if
5: $x \leftarrow$ the child of $r$ with the most descendants
6: $T' \leftarrow (\emptyset, \emptyset)$
7: for all child $y$ of $r$ do
8: $T_y \leftarrow$ subtree of $T$ rooted at $y$
9: $T'_y \leftarrow$ HeavyPathsImpl($T_y, y$)
10: if $x = y$ then
11: $p \leftarrow$ root (heavy path) of $T'_x$
12: $q \leftarrow (r, x)$ concatenated to $p$
13: $V(T') \leftarrow V(T') \cup V(T'_x) \setminus \{p\} \cup \{q\}$
14: $E(T') \leftarrow E(T') \cup E(T'_x)$
15: else
16: $V(T') \leftarrow V(T') \cup V(T'_y)$
17: $E(T') \leftarrow E(T') \cup E(T'_y) \cup \{(r, y)\}$
18: end if
19: end for
20: return $T'$
21: end function

22: function HeavyPaths($T, r$)
23: $\text{counts} \leftarrow$ array of sizes of subtrees of vertices of $T$
24: $T' \leftarrow$ HeavyPathsImpl($T, r$)
25: $p \leftarrow$ path in $T'$ rooted at $r$
26: $e_r \leftarrow$ edge in $p$ incident with $r$
27: $q \leftarrow p \setminus \{r\}$
28: return $(V(T') \setminus \{p\} \cup \{q, \{r\}\}, E(T') \cup e_r)$
29: end function