Engineering Fused Lasso Solvers on Trees

Elias Kuthe
Computer Science XI, TU Dortmund University, Germany
elias.kuthe@tu-dortmund.de

Sven Rahmann
Genome Informatics, Institute of Human Genetics, University of Duisburg-Essen, Essen, Germany
http://www.rahmannlab.de/people/rahmann
Sven.Rahmann@uni-due.de

Abstract

The graph fused lasso optimization problem seeks, for a given input signal $y = (y_i)$ on nodes $i \in V$ of a graph $G = (V, E)$, a reconstructed signal $x = (x_i)$ that is both element-wise close to y in quadratic error and also has bounded total variation (sum of absolute differences across edges), thereby favoring regionally constant solutions. An important application is denoising of spatially correlated data, especially for medical images.

Currently, fused lasso solvers for general graph input reduce the problem to an iteration over a series of "one-dimensional" problems (on paths or line graphs), which can be solved in linear time. Recently, a direct fused lasso algorithm for tree graphs has been presented, but no implementation of it appears to be available.

We here present a simplified exact algorithm and additionally a fast approximation scheme for trees, together with engineered implementations for both. We empirically evaluate their performance on different kinds of trees with distinct degree distributions (simulated trees; spanning trees of road networks, grid graphs of images, social networks). The exact algorithm is very efficient on trees with low node degrees, which covers many naturally arising graphs, while the approximation scheme can perform better on trees with several higher-degree nodes when limiting the desired accuracy to values that are useful in practice.

2012 ACM Subject Classification Theory of computation → Mathematical optimization; Theory of computation → Dynamic programming; Mathematics of computing → Trees

Keywords and phrases fused lasso, regularization, tree traversal, cache effects

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.23

Supplementary Material Source code: `https://github.com/eqt/treelas`

Funding *Sven Rahmann*: DFG SFB 876/C1.

1 Introduction

The Fused Lasso Signal Approximator (FLSA), also known as Total Variation denoising [22], was introduced by Tibshirani and colleagues [27]. A general formulation is as follows.

▶ **Problem 1.** *Let $G = (V, E)$ be an undirected graph. Let $y = (y_i)_{i \in V} \in \mathbb{R}^V$ be a signal measured in each node. Let $\mu = (\mu_i) \geq 0$ be node weights, and let $\lambda = (\lambda_{ij})_{\{i,j\} \in E} \geq 0$ be edge weights. The problem is to find a minimizer x^* of the convex function*

$$f(x) := \frac{1}{2} \sum_{i \in V} \mu_i \left(x_i - y_i\right)^2 + \sum_{ij \in E} \lambda_{ij} \left|x_i - x_j\right| , \tag{1}$$

i. e., an x^* that is both element-wise close to the observation y (in squared error) and bounded in total variation across edges.

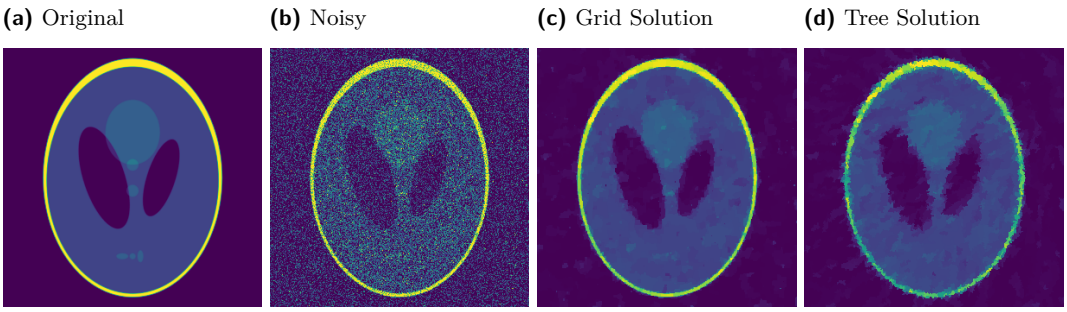

Figure 1 Image Denoising: The Shepp–Logan Phantom [23] (a standard test image), rendered at 300×300 (1a), with added Gaussian noise of standard deviation $\sigma = 0.25$ (1b). The fused lasso with parameter $\lambda = 0.2$ on a grid graph (1c) denoises while keeping most of the edges. The fused lasso on a random spanning tree (1d) is an approximation to the grid solution.

The node weights $\mu_i \geq 0$ represent the degree of confidence or precision of the observation y_i. We explicitly allow nodes i with weight $\mu_i = 0$, i. e. no observation y_i is available. Such nodes are called *latent* in contrast to *visible* nodes. The edge weights λ_{ij} represent our degree of belief that the signal does not change across edge $\{i, j\}$.

Important special cases of graphs for the FLSA are *line graphs*, where $V = \{1, \ldots, n\}$ and edges exist between i and $i + 1$ for $1 \leq i \leq n - 1$. Here one aims to reconstruct a signal x^* from observation y that is assumed to be piece-wise constant or of low total variation. For line graphs there exists a practically and theoretically fast algorithm by Johnson [15].

Other cases of practical importance are given by *grid graphs*, where $V = \{(i, j) : 1 \leq i \leq n,\ 1 \leq j \leq m\}$ with edges between (i, j) and (i', j') if and only if $|i - i'| + |j - j'| = 1$ (grid neighbors). With such graphs, one models pixels of images, such as magnetic resonance images (MRI) [12], and the FLSA is used for denoising these images without smoothing them, because sharp edges are preserved using FLSA, in contrast to the application of convolutions with kernels. Figure 1 shows an example using a common test image.

For general graphs (including grid graphs), so far only iterative approximations have been proposed, e. g. [3]. Often, an efficient line graph solver is used as a subroutine for these iterative methods [29], e. g. Consensus ADMM [25].

Trees (connected acyclic graphs) represent an interesting intermediate class of graphs. Recently, an exact $\mathcal{O}(n \log n)$ time dynamic programming algorithm was presented for trees [16]. To achieve this worst-case running time, it makes use of Fibonacci heaps [11], a complex data structure that has a reputation of being hard to implement, and being more efficient in theory than in practice. Perhaps this is also why we were unable to find any implementation of this method: There exists neither a link in the paper nor on the websites of V. Kolmogorov or T. Pock. We only found the code concerning their work on line graphs. Padilla and colleagues [20] suggested that this algorithm is slow in practice and proposed to use a heuristic instead:

> "Their algorithm [16] is theoretically very efficient, with $\mathcal{O}(n \log n)$ running time, but the implementation that achieves this running time (we have found) can be practically slow for large problem sizes, compared to dynamic programming on a chain graph. Alternative implementations are possible, and may well improve practical efficiency, but as far as we see it, they will all involve somewhat sophisticated data structures in the 'merge' steps in the forward pass of dynamic programming." [20]

So they propose to replace the tree by a line graph of the tree nodes in depth-first search (DFS) order and show that this yields a 2-approximation of the correct solution. We do not believe that such a sacrifice in accuracy should be made for the sake of efficiency.

Contributions. In this work we present the first public implementation of a fused lasso tree solver. We show that (except for adversarial generated instances) a carefully engineered tree solver is only 5 to 20 times slower than a line solver on the same number of nodes.

We describe two algorithms, an exact one that runs in $\mathcal{O}(nq \log q)$ time[1], and an approximation scheme that iteratively refines an approximate solution and closes the gap to the optimal x^*, with running time $\mathcal{O}(n \log(1/\delta))$ for a solution x with $\|x^* - x\|_\infty \leq \delta$. We compare under which conditions the exact tree algorithm and the approximation scheme yield better running time, up to double floating point precision, on different tree datasets with varying node degree distributions.

Our methods handle latent nodes and arbitrary node and edge weights, while other implementations use simplifying assumptions of $\mu_i = 1$ for all nodes $i \in V$ and $\lambda_{ij} = \lambda > 0$ for all edges $\{i, j\} \in E$.

The code, an optimized C++ implementation, is available at `github.com/eqt/treelas`. We provide bindings to Python using `pybind11` [14]. Correctness of solutions was verified via dual solutions on large graphs. The code was tested on Linux, Windows, and Mac OS to make it useful for others beyond a proof-of-concept implementation.

2 Definitions, Notation and Preliminaries

On Trees. We consider undirected simple graphs $G = (V, E)$. A tree is a connected acyclic undirected simple graph.

A tree can be rooted by selecting one node as the root r and directed by pointing all edges towards the root. The next node on the path from a node i to the root r is called the *parent* node $\pi(i)$ of i, and i is called a *child* of $\pi(i)$. The set of children of a node i is denoted as $C(i)$. A node without children is called a *leaf*. The subtree T_i rooted at node i (in the tree with root r) is the induced subgraph containing all nodes j such that i is on the path from j to r, including i itself. This set of nodes is denoted by $V(T_i)$.

A *post-order* on the nodes is any order that lists a parent after all of their children. A *pre-order* is any order that lists a parent before any of their children. As the root node r is treated in a special way, we exclude it from pre- and post-orders.

Merged Nodes. Let x^* be a solution of the fused lasso problem on a tree. We say that two nodes i and j are *merged* in the optimal solution x^* if they are in a region of equal values, i. e. $x^*_i = x^*_j = x^*_k$ for every k on the path from i to j.

The Clip Function and a Differentiability Lemma. The following *clip* function will be used frequently: For real numbers $a \leq b$, define the real-valued function

$$\mathrm{clip}_a^b(x) := \min\{b, \max\{a, x\}\}\,.$$

In other words, $\mathrm{clip}_a^b(x) = x$ is the identity for $x \in [a, b]$, and the value is "clipped" from below to a for $x < a$ and "clipped" from above to b for $x > b$.

[1] Here $q \leq 2n$ is an upper bound on the length of a certain priority queue used in the algorithm; it is a small two-digit constant in our experiments. This is further discussed below.

The following result shows that a certain function defined as a parameterized minimum is, perhaps surprisingly, differentiable in its parameter x. This is a fundamental result used in most fused lasso methods and sometimes called a "Min-Convolution".

▶ **Lemma 1** (Min-Convolution [9, 16]). *Let $g : \mathbb{R} \to \mathbb{R}$ be a convex differentiable function and $\lambda \geq 0$. Then the function*

$$h(x) := \min_z \left[g(z) + \lambda |z - x| \right]$$

is convex and differentiable in x with $h'(x) = \operatorname{clip}_{-\lambda}^{+\lambda}\left[g'(x)\right]$. Furthermore, for given x, we obtain the minimizing $z^* = \operatorname{clip}_a^b(x)$ with $a := \inf\{x \mid g'(x) \geq -\lambda\}$ and $b := \sup\{x \mid g'(x) \leq +\lambda\}$, where $\inf \emptyset = -\infty$ and $\sup \emptyset = +\infty$.

Proof. For the differentiability result, we refer to the literature [9, 16]. We derive the minimizing z^*: For z being minimal the subgradient has to contain 0, i. e.

$$0 \in g'(z^*) + \lambda \partial_z |z^* - x| \,. \tag{2}$$

Let us first assume $x < a$. As g is convex, the derivative g' is monotonically increasing, i. e. $g'(z) < -\lambda$ for all $z \leq x$. That is why the only option to make (2) true is $z^* = a$. Analogously for $x > b$ it follows $z^* = b$. For the case $x \in [a, b]$ we set $z^* = x$ and check $g'(z) = g'(x) \in [-\lambda, +\lambda]$ which shows that (2) is fulfilled. ◀

3 Algorithms

Before going into detail, we discuss the general idea of decomposing the problem on trees.

3.1 Dynamic Programming on Trees

We build solutions of subproblems (Problem 1 restricted to a subtree T_i with root i), while fixing (or assuming) a certain solution value $x_i = x$ in the subtree root.

Therefore, we define $f_i(x)$ as the minimal objective value when optimizing over the variables of the subtree T_i and fixing the subtree's root value to $x_i = x$. This way we obtain a series of functions $f_i(x)$ to be optimized by dynamic programming (bottom-up). Finally, we find the minimizer of f_r when r is the tree root.

For leaf nodes i we have $f_i(x) = \frac{1}{2}\mu_i(x - y_i)^2$ which is differentiable with $f_i'(x) = \mu_i(x - y_i)$. By induction over the tree we have the form

$$f_i(x) = \frac{1}{2}\mu_i(x - y_i)^2 + \sum_{j \in C(i)} \underbrace{\left(\min_{x_j} \lambda_j |x_j - x| + f_j(x_j) \right)}_{=: h_j(x)}, \tag{3}$$

where we abbreviated $\lambda_i := \lambda_{i,\pi(i)}$.

Minimizing along an edge term $h_j(x)$ is a form of "Min-Convolution". Applying Lemma 1 to Eq. (3), we obtain the main tool for the two algorithms described below.

▶ **Theorem 2.** *Let $f_i(x)$ be defined as in (3), as a function of the subtree root's value x. Then f_i is differentiable and*

$$f_i'(x) = \mu_i(x - y_i) + \sum_{j \in C(i)} \operatorname{clip}_{-\lambda_j}^{+\lambda_j}\left[f_j'(x)\right] \,. \tag{4}$$

The clip functions in Eq. (4) will be used as backtrace pointers in the dynamic programming algorithms, similarly to Johnson's algorithm on line graphs [15].

The *key observation* is the following one: While the derivatives $f_i' : \mathbb{R} \to \mathbb{R}$ are "infinite" objects, they have a finite representation, because they are *piecewise linear* (PWL) functions. (Because f_i is convex, f_i' is furthermore increasing.) This is seen by induction:

1. For a leaf node i, we minimize $f_i(x) = \frac{1}{2}\mu_i(x - y_i)^2$ which has the PWL derivative $f_i'(x) = \mu_i(x - y_i)$.
2. Assuming that all children's derivatives f_j' are PWL f_i' must also be PWL because the sum of PWLs is PWL, and clipping a PWL function again results in a PWL function.

Hence we can represent f_i' as an ordered sequence of knots where the slope s and intercept t of the PWL $x \mapsto sx + t$ changes. Both algorithms described below make use of this fact and the derivative recurrence in Eq. (4).

3.2 Exact Solver

The basic idea of the exact dynamic programming solver is to compute a suitable representation of the derivatives f_i'.

As all objective functions f_i, $i \in V$ are convex, the derivatives f_i' are monotonically increasing. This is why the "back-pointers" a_i and b_i with $f_i'(a_i) = -\lambda_i$ and $f_i'(b_i) = \lambda_i$ are of special interest: They define the interval $I_i := [a_i, b_i]$ of points x such that $f_i'(x)$ is not clipped, and, according to Lemma 1, the parent's value will be passed on (merged) later in the back-tracing step to determine the optimal value x_i^*. The *exact solver* first computes a complete representation of the derivative f_i' to determine the (exact) back-pointer $I_i = [a_i, b_i]$. As I_i suffices to compute the optimum x_i^*, the f_i' can be modified in place. In contrast, in the *approximation scheme*, we apply binary search to update an approximate back-pointer $[a_i^{(k)}, b_i^{(k)}]$ containing x_i^* at every iteration k.

Algorithm 1 shows the exact solver conceptually in pseudo-code; this is essentially Algorithm 2 from [16]. Each PWL derivative f_i' is represented by a priority queue containing information about its knots. There are two passes: first bottom-up (post-order) to compute back-pointers $[a_i, b_i]$ for each tree node i, then top-down (pre-order) to compute the optimal solution x^*.

Algorithm 1 TreeOpt. [16, Algorithm 2]

Input : Signal $y \in \mathbb{R}^n$, Weights $\mu \in \mathbb{R}^n$, $\lambda \in \mathbb{R}^n$, Tree T: Root r, Parent/child functions π, C
Output : Optimal solution $x^* \in \mathbb{R}^n$.

Initialize empty queues f_i' for each $i \in V$
Compute $\hat{\lambda}_i = \sum_{j \in C(i)} \lambda_i$ for each $i \in V$
for *each node i in post-order(T)* **do**
  $a_i \leftarrow \text{Clip}{\uparrow}\Big(f_i',\ \mu_i,\ -\mu_i y_i - \hat{\lambda}_i,\ -\lambda_i\Big)$
  $b_i \leftarrow \text{Clip}{\downarrow}\Big(f_i',\ \mu_i,\ -\mu_i y_i + \hat{\lambda}_i,\ +\lambda_i\Big)$
  $f_{\pi(i)}' \leftarrow f_{\pi(i)}' \cup f_i'$
$x_r^* \leftarrow \text{Clip}{\uparrow}\Big(f_r', \mu_r, -\mu_r y_r + \hat{\lambda}_r, 0\Big)$
for *each node i in pre-order(T)* **do**
  $x_i^* \leftarrow \text{clip}_{a_i}^{b_i}(x_{\pi(i)}^*)$

In terms of running time, most work is done in the Clip↑ function (Algorithm 2). In line 4 of Algorithm 1 we start with slope $s = \mu_i$ and intercept of $t = -\mu_i y_i - \hat{\lambda}_i$: This reflects the situations where the argument to the clip-function in Eq. (4) are when all summands $f'_j(x)$ evaluate to its minimum $-\lambda_j$. Then we step through the PWL f'_i, updating slope s and intercept t at every knot, until the position $\hat{x}$ is found where $f'_i(\hat{x}) = t_0 = s\hat{x} + t$.

Algorithm 2 Clip↑.

Input : Queue q, Slope s, Intercept t, Target t_0
Output : Position $\hat{x}$ (q is modified)
1 **while** q *not empty* **and** $s \times q.\min + t < t_0$ **do**
2 knot ← extract min from q
3 $s \leftarrow s +$ knot.s
4 $t \leftarrow t +$ knot.t
5 $\hat{x} \leftarrow (t_0 - t)/s$
6 Insert new knot with slope s, intercept t at position $\hat{x}$ into q

One has to be careful when processing latent nodes because in this case slope $s = 0$ can occur and Line 5 is undefined. In this case the solution of the current node is ambiguous and thus it is feasible to return $-\infty$ and not insert a new knot.

The algorithm for the counterpart Clip↓ works analogously: Iterate the loop in Line 1 while $s \times q.\max + t > t_0$ and subtract slope s and intercept t accordingly.

3.2.1 Double-Ended Mergable Queues

Algorithm 1 makes use of double-ended priority queues that are merged into the queues of their parent's queues (Line 6). For the theoretical worst-case running time the choice of the queue type defines the bottleneck.

▶ **Theorem 3.** *There is an $\mathcal{O}(n \log n)$ solver for fused lasso on trees.*

In their proof [16, Sec. 4.2] Kolmogorov et al. make use of two Fibonacci Heaps [11], a min-heap and a max-heap. The performance of Fibonacci Heaps in practical applications is controversially discussed [18, 5, 4]. Different types of double-ended priority queues as (Rank) Pairing Heaps [10, 24, 8, 13] or other sophisticated heap data structures are conceivable.

We believe that such complicated data structures are not necessary. Consider the algorithm processing node i: The decisive question for determining the running time is how many nodes are there in the queue in the forward pass when starting in Line 3? We found that the queues always contained not more than some hundred elements, independent of the number of nodes n.

▶ **Observation 4** (Queue Size). *The knot of a descendant $j \in V(T_i)$ is contained in the queue f'_i only if there exists an input $y_{\pi(i)}$ such that in the corresponding optimum $x^*(y)$ the nodes i and j are merged.*

Proof. According to Algorithm 1, nodes i and j can only be merged if none of the "back-pointers" $[a_k, b_k]$ on the way between them is clipped in Line 9. ◀

Note that a node j can be merged to i without having a knot in the priority queue f'_i, i. e. the observation gives only an upper bound for the number of elements in the queue.

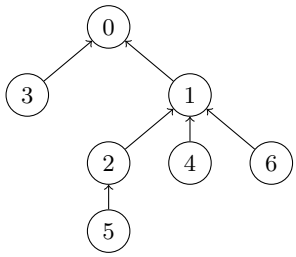

Figure 2 Queue layout: How to store the queues in memory such that every queue contains (the remaining) elements of its children's queues (separated by ":").

All in all we conjecture that in practice, the running time of the exact solver is more strongly affected by memory-related issues (e.g., cache effects) than by the actual queue data structure.

3.2.2 Implementation

We decided to replace the Fibonacci Heaps of [16], whose usage guarantees a $\mathcal{O}(n \log n)$ time algorithm, with simple plain sorted arrays. Sorting the queue in every step yields a (theoretically) suboptimal algorithm needing $\mathcal{O}(nq \log q)$ time, when q is an upper bound on the queue size. As the queue size could theoretically be $\Theta(n)$, we obtain an $\mathcal{O}\big(n^2 \log n\big)$ time algorithm in the worst case. However, our experiments show that in practice q behaves as a constant; giving an $\mathcal{O}(n)$ time algorithm in practice.

Observe that every node inserts exactly two knot elements, that define the new min and max, into the queue. This enclosing property makes it possible to pre-allocate the elements and store them in an order such that merging into the parent's queue (Line 6) is always possible and recently used elements are often in the cache (see Figure 2): For node i we allocate space for two elements at the position of its parenthesis in the parenthesis representation of the tree. To obtain the parenthesis form, walk the tree in depth-first search (DFS) [7]: Open a parenthesis when a node is discovered and close it when it is finished.

3.3 Approximation Scheme

The second algorithm we present is simpler but needs to be iterated to approximate the optimal solution. At every iteration k we refine for every node $i \in V$ the interval $[a_i^{(k)}, b_i^{(k)}]$, such that we can guarantee that the optimal solution x_i^* is contained.

The update is done by computing $\phi_i = f_i'(x_i)$ for a probing point $x_i \in [a_i^{(k)}, b_i^{(k)}]$: Recall that objective functions f_i are convex, hence f_i' are monotonically increasing. Because of Lemma 1 this means that if $\phi_i = f_i'(x_i) \geq -\lambda_i$ then $x_i^* \geq x$; analogously $\phi_i = f_i'(x_i) \leq +\lambda_i$ implies $x_i^* \leq x_i$. To compute ϕ_i for all $i \in V$ in linear time, we again apply the recursive Eq. (4). Algorithm 3 shows the pseudo-code.

It is best to set the next probing point as the middle point of the current interval, i. e. $x_i^{(k)} := (a_i^{(k)} + b_i^{(k)})/2$, such that

$$[a_i^{(k)}, b_i^{(k)}] = [x_i^{(k)} - \delta^{(k)}, x_i^{(k)} + \delta^{(k)}]$$

for some component-wise error $\delta^{(k)}$. So the next interval $[a_i^{(k+1)}, b_i^{(k+1)}]$ will be half the size $\delta^{(k+1)} = \delta^{(k)}/2$, independent of what happened in the loop in Lines 3–9.

Algorithm 3 TreeApx.

Input : Probe values $x_i \in I_i$, Bounds $I_i = [a_i, b_i]$ for $i \in V$; Tree T
Output : Improved bounds $[\hat{a}_i, \hat{b}_i] \subset [a_i, b_i]$
1 **for** *each node i in post-order(T)* **do**
2 $\quad \phi_i \leftarrow \mu_i(x_i - y_i) + \sum_{j \in C(i)} \operatorname{clip}_{-\lambda_j}^{+\lambda_j}[\phi_j]$
3 **for** *each node i in pre-order(T)* **do**
4 $\quad$ **if** $\phi_i \geq -\lambda_i$ **then**
5 $\quad\quad [\hat{a}_i, \hat{b}_i] \leftarrow [x_i, b_i]$
6 $\quad$ **else if** $\phi_i \leq +\lambda_i$ **then**
7 $\quad\quad [\hat{a}_i, \hat{b}_i] \leftarrow [a_i, x_i]$
8 $\quad$ **else**
9 $\quad\quad [\hat{a}_i, \hat{b}_i] \leftarrow [a_{\pi(i)}, b_{\pi(i)}]$

In the beginning, without apriori knowledge, we set δ_0 "big enough" such that it contains all possible solutions, e. g. $\delta_0 = \frac{1}{2}(\max_i y_i - \min_i y_i)$; further on we set the initial solutions all to the mean input signal $x_i^{(0)} = \overline{y}$. We obtain the following result.

▶ **Theorem 5.** *Algorithm 3 correctly updates the bounds $I_i^{(k)} = [a_i^{(k)}, b_i^{(k)}]$ such that after k iterations, one has $x_i^* \in I_i^{(k)}$ and $b_i^{(k)} - a_i^{(k)} = \delta_0\, 2^{-k}$ for every node $i \in V$.*

In other words, every interval I_i converges to the optimal solution x_i^* as $k \to \infty$. More precisely, to achieve component-wise accuracy of $\left\|x^* - x^{(k)}\right\|_\infty \leq \delta$ we need at least $k \geq \log_2 \delta_0 + \log_2 1/\delta$ iterations.

Engineering Memory Layout. Naively implemented, the approximation scheme (Algorithm 3) is not at all competitive to the exact solver (Algorithm 1): The reason is that walking a tree in pre- and post-order can cause a lot of memory cache misses as the next element is hard to predict for the hardware prefetcher.

To overcome this, we re-labeled the nodes such that memory layout is the pre-order (which is a reversed post-order). This way the next element will likely already be in the cache. The breadth first search (BFS) is a cache-friendly pre-order: Firstly the summands in Line 2 will all be adjacent in memory and secondly the parent's bounds in Line 9 will be accessed in a row for each child.

4 Experiments

We apply the exact algorithm TreeOpt and the $\mathcal{O}(n \log(1/\delta))$ approximation scheme TreeApx to differently structured tree datasets. The approximation is run for 20 iterations, for an an accuracy of $\delta = 2^{-20} < 10^{-6}$. In most practical applications probably fewer iterations are needed.

As there is no established library of test instances, we generate several ones. We use both randomly generated trees and random spanning trees of real-world datasets, as described in each section below. Table 1 gives an overview of the properties of different datasets.

For the sake of reproducibility the benchmarks are written as a Snakemake workflow [17] and included in the source code[2].

[2] `https://github.com/EQt/treelas/tree/master/data`

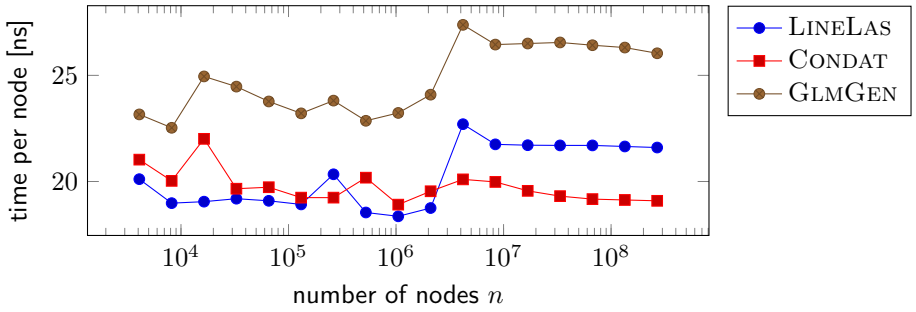

Figure 3 Running time per input node on line graphs of different size.

Table 1 Characteristics of the tree graphs used for experiments: Relevant for the experiments are the percentage of "leaf" nodes and the degree distribution $\deg_0$ of the non-leaf nodes; mean ± standard deviation are reported.

Name	n	leaf	mean $\deg_0$
Binary	100 000 000	50.0 %	3 ± 0
Europe	50 912 018	11.3 %	2.128 ± 0.367
HighDeg/LowDeg	50 000 000	37.2 %	2.592 ± 71.323
com-Orkut	3 072 441	48.6 %	2.944 ± 3.308
Phantom	1 000 000	33.6 %	2.506 ± 0.640

The benchmark system has an Intel® Core™ i7-5500U processor with cache sizes of 32 KB+32 KB (L1), 256 KB (L2) and 4096 KB (L3), and sufficient RAM (16 GB) to store each dataset plus auxiliary data structures in memory at all times. The code was compiled with GCC (`g++`) version 9.2.1, optimization level `-O3` with `-mtune=native`. All time measurements are averaged over 10 runs.

Input data $y = (y_i)$ is randomly drawn from a standard normal distribution $\mathcal{N}(0, 1)$ independently in each node. Node weights are set to $\mu_i = 1$ and edge weights are constant $\lambda_{ij} = \lambda$. To examine the effect of weak, intermediate and strong regularization, we run each algorithm with three different values $\lambda \in \{0.01, 0.1, 1\}$.

4.1 Line Graphs

For line graphs, efficient exact implementations exist. We here compare against the frequently used implementation GlmGen [1], which is algorithmically equivalent to our exact tree algorithm restricted to line graphs, and against an algorithm by Laurent Condat[3], an improvement of his earlier work [6], which is based on a dual formulation of the problem.

We measure running time per input node across a wide range of graph sizes between 5000 and 100 000 000 nodes. Overall, the time spent per node is in a narrow range between 18 and 28 nanoseconds, essentially independently of input size n. The small differences indicate that our implementation (LineLas) provides more optimization possibilities to the compiler than GlmGen, which uses the same algorithm. For large instances, Condat's algorithm is slightly faster. In 2016, Condat's implementation was considered to be the fastest one for line graphs [16]; for $n \leq 10^6$, our implementation is as fast.

[3] `https://lcondat.github.io/download/Condat_TV_1D_v2.c`; version v2, accessed January 2020

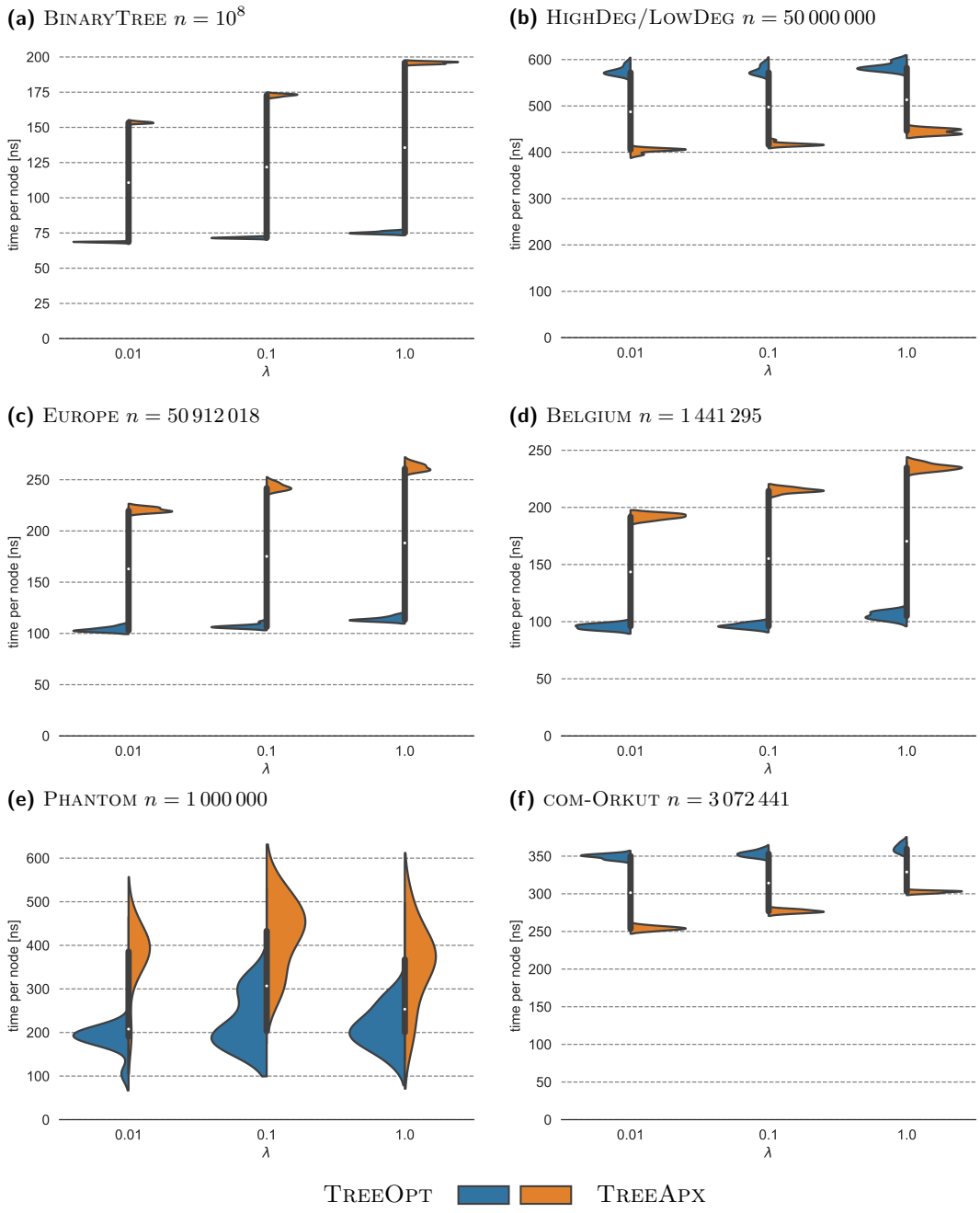

Figure 4 Violin plots of running times per node (nanoseconds) for the exact tree solver (blue) and the approximation scheme (orange) with $\delta = 2^{-20}$, averaged over 10 runs and 10 different random spanning trees when the data is a network.

4.2 Random Binary Trees

We started with a simple, regular tree: In a (full) binary tree every node has exactly two children except for the leaf nodes which constitute half of the nodes (see Table 1).

Neither algorithm is challenged by this instance (see Figure 4a): Even for $n = 10^8$ nodes the running time is roughly 4 times as slow as for a line graph.

4.3 Random High/Low Degree Trees

To challenge the exact solver, we included an instance HighDeg/LowDeg having an abundance of high-degree nodes because we expect a drop in performance, as the queue size obviously depends on the node degree. High degree nodes inevitably lead to more leaf nodes, i. e. low degree nodes. In the generated tree with $n = 5 \times 10^7$ nodes this is reflected in a relatively high standard deviation of the node degrees of 71 .

We generated the graph by means of a Prüfer sequence [21]: Thereto we find a sequence $S \in \{1, \ldots, n\}^{n-2}$ of length $n - 2$, whereby $S_i = j$ indicates that some node has parent j. From the Prüfer sequence we can reconstruct the tree in linear time [28]. We generated S as the sum of a normal distribution (high degree nodes), mixed with uniform distribution (some low degree nodes to make it more realistic):

$$S \sim |\mathcal{N}(0, 0.01)| + \mathcal{U}(1, n)$$

rounded to the closest integer in $\{1, \ldots, n\}$.

The results in Figure 4b show that indeed the approximation scheme performs better, as was expected.

4.4 Spanning Trees of Road Networks

There are a lot of applications of spatial data, like some noisy data on a map [26]. We include two graphs, Belgium (1.4 M nodes) and Europe (50 M nodes), generated out of OpenStreetMap data as provided by the "10th DIMACS Implementation Challenge: Street Networks" [2]. To obtain trees from the street graphs we computed 10 random spanning trees.

The experiments illustrate how efficiently the exact solver can process these low degree trees (see Figure 4c–4d).

4.5 Spanning Trees of Image Grid Graph

As a grid graph example we chose the Shepp–Logan Phantom [23] standard test image as shown in the introductory example (Figure 1). To be realistic as a model of a real photograph, we rendered the graph to $n = 1000 \times 1000$ nodes and added Gaussian noise of standard deviation $\sigma = 0.25$.

Interestingly the running time varies more than in the other cases (see Figure 4e). One explanation could be that this is the only instance that has a real input signal y.

4.6 Social Media Network

Cluster analysis becomes more important nowadays. Here we include the graph com-Orkut of a the free online social-media platform `www.orkut.com` which is part of the Stanford Network Analysis Project (SNAP) [19].

Although the tree with $n = 3\,072\,441$ nodes[4] is small, compared to the other graphs, the running time per node is larger as for e. g. EUROPE (see Figure 4f). With $\mathcal{O}(nq \log q)$ we would expect longer running times for larger trees. Again, the variety of node degrees (on average 2.94 ± 3.3) suits the approximation scheme better.

4.7 Summary of Observations

We see that either algorithm can be faster, depending on degree properties. As expected, the existence of nodes with high degree, i. e. high variation in node degrees, slows down the exact algorithm more than the approximation scheme. The number of nodes n is secondary for predicting the running time.

We further observe that the running times of both algorithms increase slightly with the weakening of regularization (i. e. increasing edge weight λ).

Compared to the times in Figure 3, we see that solving fused lasso on trees takes 5 times (PHANTOM) to 30 times (HIGHDEG/LOWDEG) longer, but the latter may be an unrealistic example in practice. On typical datasets, the factor is less than 20 (COM-ORKUT).

We found that about 40 % of the time of TREEAPX is used for initialization, i. e. to compute a child index and to permute the input memory. Without the memory re-ordering (re-labeling of the nodes), the running time of TREEAPX is about 10 to 20 times slower (excluding initialization).

In general both algorithms perform slightly slower with increasing tuning parameter λ: For TREEOPT this reflects Observation 4 that the sizes of the queues depends on the (potential) size of segments size. For TREEAPX the increase in running time stems from the fact that if i and $\pi(i)$ are separated in the solution, i. e. $x^*_i \neq x^*_{\pi(i)}$, the computation can be done independently.

5 Conclusion and Discussion

We presented the first implementations of fused lasso solvers on trees. We engineered both, an exact algorithm that runs in $\mathcal{O}(nq \log q)$ time and an approximation scheme that runs in $\mathcal{O}(n \log(1/\delta))$ time for a given target accuracy δ, such as $\delta = 2^{-20}$. Depending on the node degree distribution, either algorithm can be faster.

Instead of theoretically optimal data structures like Fibonacci heaps for the central priority queue in the algorithm, in practice we find that their length q is short on most natural datasets, and considerations about cache efficiency and memory layout are more important for the running time than asymptotic considerations.

Our next goal is to examine whether an iterative fused lasso solver for general graphs can be developed efficiently on the basis of one of the tree solvers presented here. Currently, general graph solvers partition the graph into overlapping paths (line graphs), solve many line graph problems and iteratively integrate the obtained solutions until they converge towards a global solution. It is reasonable to expect that by using spanning trees instead of random paths as subproblems, much fewer iterations may be necessary, which may lead to general graph solvers that converge much faster, even though each iteration on trees takes longer than an iteration on lines.

[4] Before processing the graph we had to exclude 185 nodes which are not connected to the largest connected component.

References

1 Taylor Arnold, Veeranjaneyulu Sadhanala, and Ryan J. Tibshirani. GLMGEN: Fast generalized lasso solver, 2019. URL: https://github.com/glmgen/glmgen.

2 David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner, editors. *Graph Partitioning and Graph Clustering–10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13–14, 2012*, volume 588. American Mathematical Society, 2013.

3 Amir Beck and Marc Teboulle. Fast gradient-based algorithms for constrained total variation image denoising and deblurring problems. *Image Processing, IEEE Transactions on*, 18(11):2419–2434, November 2009. doi:10.1109/TIP.2009.2028250.

4 Gerth S. Brodal. Worst-case efficient priority queues. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '96, pages 52–58, Philadelphia, PA, USA, 1996. Society for Industrial and Applied Mathematics.

5 Gerth S. Brodal. A survey on priority queues. In Andrej Brodnik, Alejandro López-Ortiz, Venkatesh Raman, and Alfredo Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms*, volume 8066 of *Lecture Notes in Computer Science*, pages 150–163. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-40273-9_11.

6 Laurent Condat. A direct algorithm for 1-d total variation denoising. *Signal Processing Letters, IEEE*, 20(11):1054–1057, November 2013. doi:10.1109/LSP.2013.2278339.

7 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

8 Amr Elmasry. Pairing heaps with costless meld. In Mark de Berg and Ulrich Meyer, editors, *Algorithms – ESA 2010*, volume 6347 of *Lecture Notes in Computer Science*, pages 183–193. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-15781-3_16.

9 Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Distance transforms of sampled functions. *Theory of computing*, 8(1):415–428, 2012.

10 Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1-4):111–129, 1986. doi:10.1007/BF01840439.

11 Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

12 Alexandre Gramfort, Bertrand Thirion, and Gaël Varoquaux. Identifying predictive regions from fmri with tv-l1 prior. In *Pattern Recognition in Neuroimaging (PRNI), 2013 International Workshop on*, pages 17–20. IEEE, 2013.

13 Bernhard Haeupler, Siddhartha Sen, and Robert E. Tarjan. Rank-pairing heaps. *SIAM Journal on Computing*, 40(6):1463–1485, 2011. doi:10.1137/100785351.

14 Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11 – Seamless operability between C++11 and Python, 2017. URL: https://github.com/pybind/pybind11.

15 Nicholas Johnson. A dynamic programming algorithm for the fused lasso and L_0-segmentation. *Journal of Computational and Graphical Statistics*, 22(2):246–260, 2013. doi:10.1080/10618600.2012.681238.

16 Vladimir Kolmogorov, Thomas Pock, and Michal Rolinek. Total variation on a tree. *SIAM J. Imaging Sciences*, 9(2):605–636, 2016.

17 Johannes Köster and Sven Rahmann. Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics*, 28(19):2520–2522, 2012.

18 Daniel H. Larkin, Siddhartha Sen, and Robert E. Tarjan. A back-to-basics empirical study of priority queues. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 61–72. SIAM, 2014.

19 Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection, 2014. URL: http://snap.stanford.edu/data.

20 Oscar H. M. Padilla, James Sharpnack, and James G Scott. The DFS fused lasso: Linear-time denoising over general graphs. *The Journal of Machine Learning Research*, 18(1):6410–6445, 2017.

21 Heinz Prüfer. Neuer Beweis eines Satzes über Permutationen. *Arch. Math. Phys.*, 27:742–744, 1918.

22 Leonid I. Rudin. *Images, numerical analysis of singularities and shock filters*. Dissertation (Ph. D.), California Institute of Technology, 1987.

23 Lawrence A. Shepp and Benjamin F. Logan. The Fourier reconstruction of a head section. *IEEE Transactions on nuclear science*, 21(3):21–43, 1974.

24 John T. Stasko and Jeffrey S. Vitter. Pairing heaps: Experiments and analysis. *Commun. ACM*, 30(3):234–249, March 1987. doi:10.1145/214748.214759.

25 Wesley Tansey and Jeffrey G Scott. A fast and flexible algorithm for the graph-fused lasso, May 2015. arXiv:1505.06475.

26 Wesley Tansey, Jesse Thomason, and James G. Scott. Maximum-variance total variation denoising for interpretable spatial smoothing. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018*, 2018.

27 Robert Tibshirani, Michael Saunders, Saharon Rosset, Ji Zhu, and Keith Knight. Sparsity and smoothness via the fused lasso. *Journal of the Royal Statistics Society: Series B*, 67(1):91–108, 2005. doi:10.1111/j.1467-9868.2005.00490.x.

28 Xiaodong Wang, Lei Wang, and Yingjie Wu. An optimal algorithm for Prufer codes. *Journal of Software Engineering and Applications*, 2(02):111, 2009.

29 Álvaro Barbero and Suvrit Sra. Modular proximal optimization for multidimensional total-variation regularization, 2014. arXiv:1411.0589.