From Theory to Practice: Engineering Approximation Algorithms for Dynamic Orientation

Ernestine Grossmann ✉ 🆔
Heidelberg University, Germany

Henrik Reinstädtler ✉ 🆔
Heidelberg University, Germany

Eva Rotenberg ✉ 🆔
IT University of Copenhagen, Denmark

Christian Schulz ✉ 🆔
Heidelberg University, Germany

Ivor van der Hoog ✉ 🆔
IT University of Copenhagen, Denmark

Juliette Vlieghe ✉ 🆔
IT University of Copenhagen, Denmark

───── **Abstract** ─────

Dynamic graph algorithms have seen significant theoretical advancements, but practical evaluations often lag behind. This work bridges the gap between theory and practice by engineering and empirically evaluating recently developed approximation algorithms for dynamically maintaining graph orientations. We comprehensively describe the underlying data structures, including efficient bucketing techniques and round-robin updates. Our implementation has a natural parameter λ, which allows for a trade-off between algorithmic efficiency and the quality of the solution. In the extensive experimental evaluation, we demonstrate that our implementation offers a considerable speedup. Using different quality metrics, we show that our implementations are very competitive and can outperform previous methods. Overall, our approach solves more instances than other methods while being up to 112 times faster on instances that are solvable by all methods compared.

1   Introduction

A (fully) dynamic graph algorithm is a data structure designed to support edge insertions, edge deletions, and answer specific queries. While there has been extensive research on dynamic graph problems, many of these theoretically efficient algorithms have not been implemented or empirically evaluated. In this work, we dynamically maintain an orientation of a graph. Let $G = (V, E)$ be an undirected graph with n vertices and m edges. A *fractional orientation* $\overrightarrow{G}$ of G assigns to each edge $\{u, v\} \in E$ two non-negative weights $d(u \rightarrow v)$ and $d(v \rightarrow u)$ such that $d(u \rightarrow v) + d(v \rightarrow u) = 1$. The *fractional out-degree* $d^+(u)$ of a vertex u is the sum over all edges $\{u, v\} \in E$ of $d(u \rightarrow v)$. An *integral orientation* is a fractional orientation in which all weights are either 0 or 1. We will denote by $d^*(u)$ the out-degree of a vertex u if the orientation is integral. Graph orientations have a wide range of applications:

Fractional Orientations and Graph Mining.   Given a graph G, a *density decomposition* of G is a nested set of subgraphs of G of decreasing density. Finding dense subgraphs lies at the core of graph mining [25]. Density decompositions were first applied in finding clusters in web-data [23, 24, 30, 31] but quickly embedded themselves in pattern mining techniques of other fields such as social graph mining [2, 11, 18, 34, 41], bioinformatics [3, 21, 28, 39, 44], finance [19] and even drawing algorithms for large graphs [1]. Danisch, Chan, and Sozio [14] propose a state-of-the-art density composition of a graph, which they call the *local density*. Due to its applicability, there exist many *static* implementations to efficiently approximate (variants of) this decomposition [29, 35, 42, 43]. There exists a 1:1 correspondence between this density decomposition and fractional orientations. Indeed, consider a fractional orientation $\overrightarrow{G}$ that minimizes the sum-of-squares metric defined by $\mathtt{sum}^2(\overrightarrow{G}) := \sum_{v \in V} (d^+(v))^2$. The decomposition by Danish, Chan, and Sozio is the partition of vertices by their out-degree.

Integral Orientations and Dynamic Efficiency.   A crucial building block for efficient, fully dynamic algorithms is efficient, dynamic edge-orientations. Given an undirected graph G, the *edge orientation problem* asks for an orientation of G in which the maximum out-degree of the vertices is minimized. Many real-world graphs are relatively sparse and allow for edge orientations where the maximum out-degree is considerably smaller than the maximum degree of the graph. Low out-degree integral orientations allow for efficient and low-memory adjacency queries. This, in turn, has many applications, including dynamic map labeling, graph colouring, maximal matchings, and finding k-cliques [16]. Other example applications include Neiman and Solomon [36] who have shown how to maintain a maximal matching in $O(\log n / \log \log n)$ amortized time using a dynamic edge orientation algorithm, dynamic matrix vector multiplication [32], or Kowalik and Kurowski [33] who use dynamic edge orientations to answer shortest-path queries of length at most k in $O(k)$ time in planar graphs. The aforementioned algorithms use time proportional to the maximum out-degree in the orientation. Thus, it becomes important to minimize $\mathtt{max}(\overrightarrow{G}) := \max_{v \in V} d^+(v)$.

Minimax versus Least Square Regression.   In graph mining, we are interested in orientations $\overrightarrow{G}$ that minimize $\mathtt{sum}^2(\overrightarrow{G})$. Minimising this measure optimises for the "average" out-degree which identifies the density of the neighborhood of each vertex. On the other hand, many dynamic algorithmic application domains are interested in orientations $\overrightarrow{G}$ that minimize $\mathtt{max}(\overrightarrow{G})$. Danisch, Chan, and Sozio prove that, in this case, the least square regression is strictly more stringent than the minimax regression as any fractional orientation $\overrightarrow{G}$ minimizing $\mathtt{sum}^2(\overrightarrow{G})$ also minimizes $\mathtt{max}(\overrightarrow{G})$ [14]. The value of $\mathtt{max}(\overrightarrow{G})$ equals the pseudoarboricity of

the graph. The (pseudo-)arboricity is defined as the number of (pseudo-)forests that the graph can be partitioned into. The arboricity is either equal to the pseudoarboricity or plus one [37].

Prior Work in Theory.   Theoretical dynamic algorithms typically maintain a fractional orientation $\overrightarrow{G}$ over G. A naive rounding scheme then converts each fractional orientation $\overrightarrow{G}$ to an integral orientation $\overrightarrow{G}^*$ where for all $v \in V$, the out-degree $d^*(v)$ is at most $2d^+(v)$. A simple rounding scheme rounds all fractional edges with a value greater than $\frac{1}{2}$ and ones with equal value by a deterministic scheme, like the assigned vertex id. Christiansen and Rotenberg [12] propose a dynamic polylog-time rounding algorithm where for all $v \in V$, $d^*(v) \leq d^+(v) + 1$. The state-of-the-art dynamic theoretical algorithm to maintain a fractional orientation that minimizes $\mathtt{max}(\overrightarrow{G})$ is by Chekuri et al. [10]. They dynamically maintain a fractional orientation $\overrightarrow{G}$ where for all $\overrightarrow{G}'$, $\mathtt{max}(\overrightarrow{G}) \leq (1 + \varepsilon) \cdot \mathtt{max}(\overrightarrow{G}')$. Their algorithm uses linear space and has $O(\varepsilon^{-6} \log^4 n)$ update time. Christiansen, van der Hoog, and Rotenberg [13] show that the algorithm from [10], in actuality, maintains a fractional orientation $\overrightarrow{G}$ such that for all $\overrightarrow{G}'$, $\mathtt{sum}^2(\overrightarrow{G}) \leq (1 + \varepsilon) \cdot \mathtt{sum}^2(\overrightarrow{G}')$. Thus, the state-of-the-art theoretical algorithm maintains a fractional orientation that $(1+\varepsilon)$-approximates the optimal solution for both our metrics of interest in $O(\varepsilon^{-6} \log^4 n)$ time per update.

Prior Work in Practice.   In contrast to theoretical algorithms, most practical methods for computing edge-orientations on dynamic graphs maintain an integral orientation. We now discuss some experimentally evaluated approaches. Brodal and Fagerberg [9] were the first to consider the problem in the dynamic case. With a bound c on the arboricity of the graph as input, their approach supports adjacency queries in time $O(c)$, edge insertions in amortized time $O(1)$, and edge deletions in amortized time $O(c + \log n)$ time.

Berglin and Brodal [5] allow a *worst-case* user-specific trade-off between out-degree and running time of the operations. Their algorithm maintains an $O(\alpha + \log n)$ orientation in $O(\log n)$ worst-case time or an $O(\alpha \log^2 n)$ orientation in constant worst-case time. Borowitz et al. [6] experimentally evaluated different heuristics and approximation algorithms. This evaluation includes new approaches presented by the authors, which are based on a BFS strategy as well as the algorithm by Brodal and Fagerberg [9] and the K-Flip algorithm introduced by Berglin and Brodal [5]. In this experimental evaluation, the BFS approach performed best. Recently, Großmann et al. [27] presented a dynamic algorithm to solve the problem exactly. The worst-case complexity of this method for insertions is $O(m)$, while for deletion it is amortized $O(m)$. Their experiments show that their exact algorithm achieves up to $32\,\%$ lower running times than the inexact BFS approach by Borowitz et al. [6].

Contribution.   We engineer an implementation of the recently proposed approximation algorithms by Chekuri et al. [10]. In Section 2.1, we provide crucial details required for an implementation. A straightforward implementation of the algorithms would be quite slow and memory-intensive. This is because [10] defines a large (constant) number of values and pointers to guarantee that all algorithmic operations take theoretically $O(1)$ time. Real-world graphs contain many edges compared to the number of available bits in memory. Storing up to eight integers and pointers per edge is infeasible and a clear bottleneck.

Thus, in Section 3, we present engineering aspects to adapt the algorithm from [10] to reduce space use and considerably improve its efficiency. As a consequence, we obtain two implementations of [10] to maintain a fractional orientation $\overrightarrow{G}$ of a dynamic graph G that simultaneously approximately minimizes both $\mathtt{sum}^2(\overrightarrow{G})$ and $\mathtt{max}(\overrightarrow{G})$. We note that

these existing algorithms maintain an integral orientation $\overrightarrow{G}^*$ and are designed to minimize $\mathtt{max}(\overrightarrow{G}^*)$. To provide a fair comparison between our implementation and the current state-of-the-art, we convert our fractional orientation $\overrightarrow{G}$ to an integral orientation through the naive rounding scheme. Experiments in Section 4 show that our implementations offer a considerable speedup of up to 112.17 times whilst being very competitive on the measure $\mathtt{max}(\overrightarrow{G}^*)$. Our algorithm performs considerably better than previous implementations when using $\mathtt{sum}^2(\overrightarrow{G}^*)$ as the quality metric. Our implementation has a natural parameter λ, which allows for a trade-off between algorithmic efficiency and the quality of the solution.

2 Preliminaries

Let $G = (V, E)$ be a dynamic, unweighted, undirected graph subject to edge insertion and edge deletion. We say that $|V| = n$ at the time of an update. A *fractional* orientation $\overrightarrow{G}$ of G assigns for every edge $\{u, v\}$ two non-negative weights: $d(u \rightarrow v)$ and $d(v \rightarrow u)$ such that $d(u \rightarrow v) + d(v \rightarrow u) = 1$. The *out-degree* of a vertex u is defined as $d^+(u) = \sum_{v \in V} d(u \rightarrow v)$. An *integral orientation* is a fractional orientation where for all $\{u, v\} \in E$, $d(u \rightarrow v) \in \{0, 1\}$. To distinguish between integral and fractional orientations, we denote by $d^*(u)$ the out-degree of a vertex u if the orientation is integral. In integral orientations, we reference directed edges as $\overrightarrow{uv}$ and $\overrightarrow{vu}$. A naive rounding scheme converts each fractional orientation $\overrightarrow{G}$ into an integral orientation $\overrightarrow{G}^*$ where for all $v \in V$, the out-degree $d^*(v)$ is at most $2d^+(v)$.

Quality Measures. We denote by $\mathbb{O}(G)$ the infinite-size set of all fractional orientations of G. For any orientation $\overrightarrow{G} \in \mathbb{O}(G)$ we define two quality measures:

$$\mathtt{max}(\overrightarrow{G}) := \max_{v \in V} d^+(v), \qquad \text{and,} \qquad \mathtt{sum}^2(\overrightarrow{G}) := \sum_{v \in V} \left(d^+(v) \right)^2.$$

These measures induce respectively a linear and a convex optimization problem over $\mathbb{O}(G)$:

$$\Delta^{\min}(G) := \min_{\overrightarrow{G} \in \mathbb{O}(G)} \mathtt{max}(\overrightarrow{G}), \qquad \text{and,} \qquad \Delta^{local}(G) := \min_{\overrightarrow{G} \in \mathbb{O}(G)} \mathtt{sum}^2(\overrightarrow{G}).$$

Problem Statement. State-of-the-art edge orientation implementations use integral orientations. We therefore maintain for a dynamic unweighted graph G either:
- an integral orientation $\overrightarrow{G}^*$ that approximately minimizes $\mathtt{max}(\overrightarrow{G}^*)$, or,
- an integral orientation $\overrightarrow{G}^*$ that approximately minimizes $\mathtt{sum}^2(\overrightarrow{G}^*)$

To this end, we dynamically maintain a fractional orientation $\overrightarrow{G}$ and apply naive rounding.

Fairness. Sawlani and Wang [40] say that $\overrightarrow{G} \in \mathbb{O}(G)$ is *locally fair* whenever $d(u \rightarrow v) > 0$ implies that $d^+(u) \leq d^+(v)$. They prove that if $\overrightarrow{G}$ is locally fair, then $\mathtt{max}(\overrightarrow{G}) = \Delta^{\min}(G)$. Christiansen, van der Hoog, and Rotenberg [13] show an even stronger correlation between fair orientations and the sum-of-squares measure as $\overrightarrow{G}$ is locally fair if and only if $\mathtt{sum}^2(\overrightarrow{G}) = \Delta^{local}(G)$. A locally fair orientation cannot be maintained without requiring $\Omega(n)$ update time. Hence, several works relax the notion of fairness to approximate the local fairness and thereby $\Delta^{\min}(G)$ and $\Delta^{local}(G)$ [4, 7, 8, 13, 20, 40, 45]. For a fixed $\theta \in \{0, 1\}$ and $\lambda \in [0, 1)$, Chekuri et al. [10] define $\overrightarrow{G}$ as λ-fair if: $d(u \rightarrow v) \Rightarrow d^+(u) \leq (1 + \lambda) \cdot d^+(v) + \theta$.

If we subsequently smartly choose $\lambda \in \Theta\left(\frac{\varepsilon^2}{\log n}\right)$ and $\theta = 0$, then any λ-fair fractional orientation $\overrightarrow{G}$ guarantees that $\mathtt{max}(\overrightarrow{G})$ is a $(1 + \varepsilon)$-approximation of $\Delta^{\min}(G)$ [10] and that
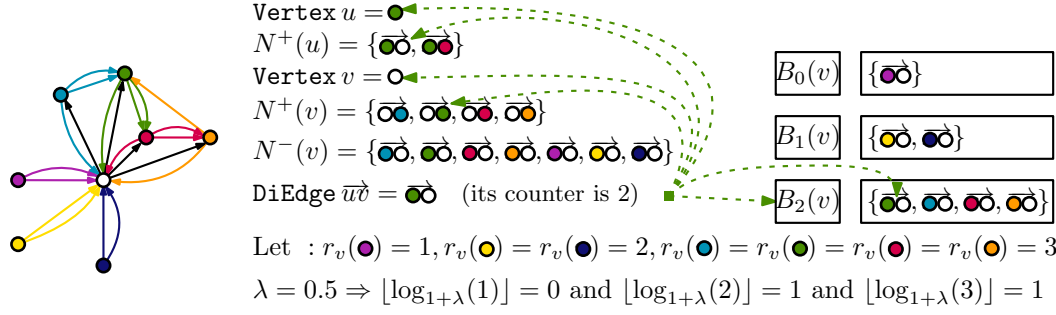
Figure 1 A graph G_b for $b = 2$ and an orientation G_b^*. We illustrate the data structure from [10]. Observe that the DiEdge $\overrightarrow{uv}$ stores six pointers in this data structure to allow constant-time updates.

$\mathtt{sum}^2(\overrightarrow{G})$ is a $(1 + \varepsilon)$-approximation of $\Delta^{local}(G)$ [13]. If $\theta = 1$, the λ-fair orientation may be integral instead, which is considerably easier to maintain. Setting $\theta = 1$ does come at a cost in quality, as the approximation becomes an additive error of $O(\log n)$.

2.1 The Algorithm for Maintaining a λ-fair Orientation by [10]

Let $\lambda \in [0, 1]$ be some quality parameter that the user fixed beforehand. Then, the algorithm by Chekuri et al. [10], on a high level, inserts (or deletes) directed edges in G_b^* (which is an integral orientation over the graph G where each edge is duplicated b times). They orient the edges in such way that they maintain an (integral) λ-fair orientation. After inserting a directed edge $\overrightarrow{uv}$, the out-degree of u increases. So, there may exist some edge $\overrightarrow{ux}$ for which $d^*(u) > (1 + \lambda)d^*(x)$. The algorithm then flips this edge, restoring the out-degree of x, and recurses on the edge-insertion $\overrightarrow{xu}$. Instead of spending $O(d^*(u))$ time to inspect all out-edges x of u, the authors show that it suffices to loop over a fraction of the out-edges of u in round-robin fashion. To make all supporting operations in this algorithm run in constant time, the algorithm uses elaborate data structures, data types, and information updates. We explain these first.

For $b \in \mathbb{N}$, denote by G_b the graph G where every edge is duplicated b times. Each edge insertion in G thereby inserts b edges into G_b. We maintain an integral orientation G_b^* over G_b where $d_b^*(v)$ denotes the out-degree. G_b^* induces a fractional orientation over G and we obtain an integral orientation G^* of G from G_b^* by naively rounding each edge.

We denote for all $u \in V$ by $N^+(u)$ all vertices v for which G_b^* has at least one edge $\overrightarrow{uv}$ (see Figure 1). The set $N^-(u)$ are all vertices v where $\overrightarrow{G_b}$ includes at least one edge $\overrightarrow{vu}$. Note that $d_b^*(u)$ is not necessarily equal to $|N^+(u)|$ because G_b is a multigraph.

Information Updates. The update algorithm to G_b^* implements a lazy information update scheme where adding an edge $\overrightarrow{uv}$ to G_b^* increments a counter recording the number of edges $\overrightarrow{uv}$ that G_b^* contains without updating all auxiliary information and data structures.

Formally we define an *information update* (we forward reference Algorithm 3). The algorithm may invoke an information along a directed edge.

▶ **Definition 1.** *Let $d_b^*(w)$ be the out-degree of w at the time of the information update. We define the* log-degree $r_u(w)$ *as the floor of the base $(1 + \lambda)$ logarithm of $d_b^*(w)$ at the time of an information update along $\overrightarrow{wu}$. Or formally: $r_u(w) = \lfloor \log_{1+\lambda} d_b^*(w) \rfloor$.*

This value is recorded such that each directed edge $\overrightarrow{wu}$ in G_b^* has access to the value $r_u(w)$.

Data Structure. Chekuri et al. [10] equip the oriented graph G_b^* with a data structure. They store for each vertex $u \in V$ the following (see also Figure 1):

(a) The *exact value* of $d_b^*(u)$.

(b) The set $N^+(u)$ in a linked list and a pointer some current "position" in the linked list.

(c) The set $N^-(u)$ in a doubly linked list of buckets $B_j(u)$ sorted by j from high to low. Each bucket $B_j(u)$ contains a doubly linked list of all $w \in N^-(u)$ where $r_u(w) = j$.

(d) A pointer to the bucket $B_j(u)$ where j is closest to $\log_{1+\lambda} d^+(u)$.

Data Types. To guarantee that all data structures have $O(1)$ update time, the proofs in [10] reference several pointers stored in their data types. We formalise these here:

> `Bucket` $B_i(v)$ stores a doubly linked list with all `DiEdges` $\overrightarrow{uv}$ with $i = r_v(u)$.
>
> `Vertex` u stores:
>
> - a linked list of all `DiEdges` $\overrightarrow{uv}$ (corresponding to vertices $v \in N^+(u)$).
> - a doubly linked list of all non-empty `Buckets` $B_i(u)$, ordered by i.
> - a pointer to the `DiEdge` *robin* which represents our "current position" in $N^+(u)$.
> - a pointer to the *primary bucket* $B_j(u)$ (here j is closest to $\log_{1+\lambda} d^+(u)$).
> - the value $d_b^*(u)$.
>
> `DiEdge` $\overrightarrow{uv}$ represents all directed edges $\overrightarrow{uv}$ in G_b^*, simultaneously. It stores:
>
> - two pointers to the `Vertices` u and v, respectively.
> - a *counter* counting how many edges $\overrightarrow{uv}$ there are in G_b^*,
> - the value $r_v(u)$,
> - a pointer to the location of $\overrightarrow{uv}$ in $N^+(u)$.
> - a pointer to the bucket $B_i(v)$ of v with $i = r_v(u)$,
> - a pointer to the location of $\overrightarrow{uv}$ in the linked list in the bucket $B_i(v)$.
> - a pointer to the `DiEdge` $\overrightarrow{vu}$.

2.1.1 Algorithm Definition

The algorithm has three parameters that are used throughout its algorithmic subroutines: $\theta \in \{0,1\}$, $\lambda \in [0,1]$ and $b \in \mathbb{N}$. We explain their interplay later in this subsection. Each update operation in G simply invokes b updates to the digraph G_b^* (see Algorithm 1 and 2).

■ **Algorithm 1** Insert(edge $\{u,v\}$ in G).
1 **for** $i \in [b]$ **do**
2   **if** $d_b^*(u) \leq d_b^*(v)$ **then**
3   Insert($\overrightarrow{uv}$)
4   **else**
5   Insert($\overrightarrow{vu}$)

■ **Algorithm 2** Delete(edge $\{u,v\}$ in G).
for $i \in [b]$ **do**
if $u \in N^-(v)$ **then**
Delete($\overrightarrow{vu}$)
else
Delete($\overrightarrow{uv}$)

Recall that the algorithm does not always immediately update the data structure. Rather, it only exactly maintains the value $d_b^*(u)$ for each vertex u and the set $N^+(u)$. The bucketing data structure is updated only when an *information update* (Algorithm 3) is triggered.

■ **Algorithm 3** InfoUpdate(`DiEdge` $\overrightarrow{xy}$, `Bucket` $B_j(y)$).
6 $r_y(x) = \log_{1+\lambda} d_b^*(x)$
7 Starting from $B_j(y)$ find the bucket $B_i(y)$ with $i = r_y(x)$ *Inspect at most $O(1)$ buckets*
8 Move $\overrightarrow{xy}$ into the bucket $B_i(y)$

Suppose that we add a directed edge $\overrightarrow{uv}$ to G_b^*. If there already exists an edge $\overrightarrow{uv}$ in $\overrightarrow{G_b}$ then we implicitly insert the extra copy by identifying the corresponding DiEdge $\overrightarrow{uv}$ and incrementing the counter. Otherwise, we need to update $N^+(u)$ and do an information update. The same principle holds when removing a directed edge, where we only do work if we remove the last copy of a directed edge in G_b^* (Algorithm 4 and 5).

▇ **Algorithm 4** Add(DiEdge $\overrightarrow{uv}$).	▇ **Algorithm 5** Remove(DiEdge $\overrightarrow{uv}$).
9 Increment the counter of $\overrightarrow{uv}$.	Decrement the counter of $\overrightarrow{uv}$
10 $d_b^*(u) = d_b^*(u) + 1$	$d_b^*(u) = d_b^*(u) - 1$
11 **if** $\overrightarrow{uv}$.counter $= 1$ **then**	**if** $\overrightarrow{uv}$.counter $= 0$ **then**
12 Add $\overrightarrow{uv}$ at the end of $N^+(u)$	Remove v from $N^+(u)$
13 $B_j(v) \leftarrow$ the *primary* Bucket of v	$B_i(v) \leftarrow$ the Bucket stored at $\overrightarrow{uv}$
14 InfoUpdate($\overrightarrow{uv}$, $B_j(v)$)	InfoUpdate($\overrightarrow{uv}$, $B_i(v)$)

The true complexity of this algorithm lies in the logic surrounding directed edge insertions and deletions. Intuitively, inserting an edge $\overrightarrow{uv}$ iterates over the out-edges $\overrightarrow{ux}$ of u in a round-robin fashion. If the out-degree of u is considerably larger than the out-degree of x, we flip an edge $\overrightarrow{ux}$ to become an edge $\overrightarrow{xu}$, we terminate the round-robin and recurse on the insertion of $\overrightarrow{xu}$. After this flip, the out-degree of u is unchanged and thus locally, the orientation is still λ-fair. This increased the out-degree of x by one, and we recurse on x (Algorithm 1). Deleting an edge $\overrightarrow{uv}$ only considers a single directed edge, which is an edge $\overrightarrow{xu}$ where the out-degree $d_b^*(x)$ is approximately maximal. If $d_b^*(u)$ is considerably larger than $d_b^*(u)$ then we flip an edge $\overrightarrow{xu}$. After this flip, the out-degree of u is unchanged. Instead, the out-degree of x is decreased by one, and we recurse on x (Algorithm 1).

Finally if, after either an edge insertion or deletion, a vertex u actually changes its out-degree then the algorithm will inform up to $\frac{2}{\lambda}$ out-neighbours of u of its new out-degree.

▇ **Algorithm 6** Insert(DiEdge $\overrightarrow{uv}$).	▇ **Algorithm 7** Delete(DiEdge $\overrightarrow{uv}$).
15 Add($\overrightarrow{uv}$)	Remove($\overrightarrow{uv}$)
16 **for** $\lceil\frac{2}{\lambda}\rceil$ iterations do: **do**	$x \leftarrow$ First(Max(Bucket($N^-(u)$))))
17 u.robin $= (u$.robin $+ 1)$ mod $|N^+(u)|$	**if** $d_b^*(x) > \max\{(1+\lambda) \cdot d_b^*(u) + \theta, \frac{b}{4}\}$
18 DiEdge $\overrightarrow{ux} = {}^*u$.robin	**then**
19 **if** $d_b^*(u) > \max\{(1+\lambda) \cdot d_b^*(x) + \theta, \frac{b}{4}\}$ **then**	Add($\overrightarrow{ux}$)
20 Remove($\overrightarrow{ux}$)	*Delete($\overrightarrow{xu}$)*
21 *Insert($\overrightarrow{xu}$)*	**else**
22 BREAK	**for** the next $\lceil\frac{2}{\lambda}\rceil$ $\overrightarrow{uv}$ in $N^+(u)$ **do**
23 **for all** the next $\lceil\frac{2}{\lambda}\rceil$ $\overrightarrow{uv}$ in $N^+(u)$ **do**	InfoUpdate($\overrightarrow{uv}$, $\overrightarrow{uv}$.Bucket)
24 InfoUpdate($\overrightarrow{uv}$, $\overrightarrow{uv}$.Bucket)	

Algorithm Parameters and Asymptotic Complexity. The algorithm has three parameters:
- $\theta \in \{0, 1\}$ is a Boolean that is used in the logic of Algorithms 6 and 7. The Boolean specifies whether we want a slower $(1 + \varepsilon)$-approximation (setting $\theta \leftarrow 0$), or, whether we want a faster algorithm that achieves an additive $O(\log n)$-error (setting $\theta \leftarrow 1$).
- $\lambda \in [0, 1]$ is the approximation parameter.
  - If $\theta = 0$, then we may choose $\lambda \in \Theta(\frac{\varepsilon^2}{\log n})$ to obtain a $(1 + \varepsilon)$-approximation [10, 13].
  - If $\theta = 1$, then we may choose $\lambda \in \Theta(\frac{1}{\log n})$ to obtain an additive $O(\log n)$ error [10, 13].
- $b \in \mathbb{N}$ records how many edges the multigraph G_b has per edge in G. If $\theta = 1$ then $b = 1$. If $\theta = 0$ then $b > \frac{4}{\lambda}$ in order to make the algorithmic logic work. Indeed, consider an edge $\{u, v\}$ in G. Our algorithm maintains the invariant that if $\overrightarrow{uv}$ is in G_b^* then $d_b^*(u) \leq (1 + \lambda)d_b^*$. If both $d_b^*(u)$ and $d_b^*(v)$ are less than $\frac{1}{\lambda}$ then there is no way to orient $\{u, v\}$ in G_b^* such that our invariant is satisfied.

Each update in G triggers at most b updates in G_b. Each update to G_b calls Algorithm 6 or 7. These functions do at most $O(\frac{1}{\lambda})$ constant-time operations before recursing. These functions recurse at most $O(\log_{1+\lambda} n)$ times. It follows that the total runtime is at most:

$$O(b \cdot \frac{1}{\lambda} \cdot \log_{1+\lambda} n) \subset O(b\lambda^{-2} \log n) \Rightarrow \begin{cases} O(\lambda^{-3} \log n) & \text{if } \theta = 0 \\ O(\lambda^{-2} \log n) & \text{if } \theta = 1 \end{cases}$$

In theory, one chooses $\lambda \in \Theta(\varepsilon^2 / \log n)$ where ε is the approximation constant. In practice, we simply choose a smaller λ for a better approximation.

Space Usage.   In theory, the data type as we specified in this paper allows each of the operations in Algorithms 1–7 to take constant time. We give three examples: (i) we can flip a directed edge in G_b^* in constant time because the `DiEdge` $\overrightarrow{ux}$ stores a pointer to the `DiEdge` $\overrightarrow{xu}$, (ii) a `DiEdge` $\overrightarrow{uv}$ can remove itself from the linked list $N^+(u)$ in constant time because it stores a pointer to its location, and (iii) a `DiEdge` $\overrightarrow{uv}$ can remove itself from the doubly linked list $B_i(v)$ of $N^-(v)$ in constant time because it stores a pointer to its location.

While theoretically efficient, this solution faces a practical bottleneck due to high space usage. Indeed, it stores six pointers and two integers per edge in the original graph G.

Moreover, the excessive use of pointers makes executing the algorithm less predictable and slowing it down in practice. Additionally, each pointer is not efficient in addressing only a part of the memory, i.e. the out-going edges, since it requires constant (64-bit) memory.

3   Engineering Aspects

We now engineer an efficient implementation of [10]. Our key observation is that there are two data containers where the order of the elements in the container is irrelevant:
- `Vertex` u stores all `DiEdges` $\overrightarrow{uv}$ in arbitrary order, and
- `Bucket` $B_i(v)$ stores all `DiEdges` $\overrightarrow{uv}$ with $i = r_v(u)$ in arbitrary order.

By sorting these sets as arrays, we obtain more control over the size of these objects and the size of pointers to elements in these sets. This in turn allows us to use packing techniques to store the same information in fewer bits.

Arrays.   This allows us to use arrays instead of linked list as we give each:
- `Vertex` u an array A_u^+ with all `DiEdges` $\overrightarrow{uv}$ in arbitrary order.
- `Bucket` $B_i(v)$ an array B_i^v with all `DiEdges` $\overrightarrow{uv}$ with $i = r_v(u)$ in arbitrary order.

Adding a `DiEdge` $\overrightarrow{uv}$, given the `Vertex` u and the correct `Bucket` $B_i(v)$, takes amortized constant time. Indeed, we simply increase the sizes of A_u^+ and B_i^v and append $\overrightarrow{uv}$. Removing $\overrightarrow{uv}$ also takes amortized constant time: given its index a in A_u^+ (and b in B_i^v) we may swap $A_u^+[a]$ with the last element of A_u^+ (swap $B_i^v[b]$ with the last element) and decrement the array's size by one. These operations are supported by the `vector` data type in C++.

From Arrays to Packing.   Replacing doubly linked lists with vectors naturally reduces algorithmic overhead. It also has a secondary benefit: instead of maintaining, for a `DiEdge` $\overrightarrow{uv}$, pointers to its location in linked lists we store integers maintaining its index in vectors. Recall that we maintain an orientation G_b^* where the maximum out-degree $\text{Max}(G_b^*)$ is low. As a consequence, for any vertex v, the array A_u^+ is small. Thus, for a `DiEdge` $\overrightarrow{uv}$ its index in A_u^+ is small. Via the same logic, the integers $robin$, $d_u^*(u)$, $\lfloor \log_{1+\lambda}(r_u(v)) \rfloor$ and the index of the `DiEdge` $\overrightarrow{vu}$ are small. We pack these small values into 32-bit integers.

Indirection. Any DiEdge $\overrightarrow{uv}$ contained pointers to: the Vertex u, the Vertex v, and the DiEdge $\overrightarrow{vu}$. If each DiEdge only stores a pointer to its target then we can access the Vertex u from $\overrightarrow{uv}$ by first accessing the DiEdge $\overrightarrow{vu}$ and then reading its target.

Removing Bucket Pointers. Each Vertex u maintained a pointer to the primary bucket $B_j(u)$ and each DiEdge $\overrightarrow{uv}$ maintained a pointer to the bucket $B_i(v)$ that contains $\overrightarrow{uv}$. These pointers are only used in Algorithm 3. Chekuri et al. [10] prove that to find for a DiEdge $\overrightarrow{xy}$ is current bucket, one steps over at most $O(1)$ buckets from the given pointer. We choose to remove these pointers. Instead, we find the correct bucket for $\overrightarrow{xy}$ by looping over all buckets $B_i(y)$. Observe that there are at most $O(\log_{1+\lambda} n) \subseteq O(\lambda^{-1} \log n)$ buckets $B_i(y)$. Moreover, a call to Algorithm 6 or 7 triggers $O(\lambda^{-1})$ information updates *only if* it does not recurse. So, the asymptotic running time remains the same.

Compact Types. By our above analysis, we obtain more compact data types (see Figure 2):

Bucket $B_i(u)$ stores a vector B_i^u with DiEdges $\overrightarrow{wu}$ with $i = r_u(w)$.

Vertex u stores:

- a vector A_u^+ storing all DiEdges $\overrightarrow{uv}$ (corresponding to vertices $v \in N^+(u)$).
- a linked list of all non-empty Buckets $B_i(u)$, ordered by i.
- a 32-bit integer storing our "current position" *robin* in A_u^+ and the value $d_b^*(u)$.

DiEdge $\overrightarrow{uv}$ represents all directed edges $\overrightarrow{uv}$ in G_b^*, simultaneously. It stores:

- A pointer to the Vertex v.
- A 32-bit integer storing the *counter*, the value $r_v(u)$, and the index of $\overrightarrow{vu}$ in A_u^+.
- An 32-bit integer storing the index of $\overrightarrow{uv}$ in B_i^v for $i = r_v(u)$.

3.1 A Space-Time Trade-off

Finally, we propose one more modification to the algorithm. The algorithm in [10] and our above implementation store for a vertex v all non-empty buckets $B_i(v)$ in a linked list. The algorithm in [10] accesses these buckets by storing a pointer for each edge in G. In our implementation, we find the bucket $B_i(v)$ where $i = r_v(u)$ in $O(\log_{1+\lambda} n)$ time by iterating over all buckets $B_j(v)$. We showed that that this does not increase the asymptotic running time. However, it does increase the *actual* running time of the program.

Both methods have an inherent downside: [10] uses a lot of memory per edge, and our above implementation is relatively slow. Moreover, doubly linked list inherently bring a lot of overhead and cache misses. We therefore propose a third solution to store the buckets:

We define for a Vertex v the *Bucket array* $\mathbb{B}_v$ such that for all i, $\mathbb{B}_v[i] = B_i(v)$. Any DiEdge $\overrightarrow{uv}$ can access the bucket $B_i(v)$ where $i = r_v(u)$ by simply indexing $\mathbb{B}_v[r_v(u)]$ in constant time. By storing the array $\mathbb{B}_v$ as a vector, we require only a relatively small amount
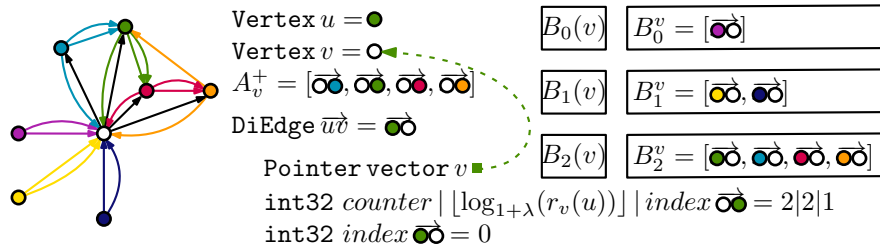
Figure 2 We illustrate our more compact data structure compared to the one from Figure 1.

of memory for every empty bucket. There are $O(\log_{1+\lambda}(n))$ buckets for each vertex v, so this additively increases our space usage by $O(\frac{n}{\lambda} \log n)$. Since the number of edges m is considerably larger than the number of vertices n, this is space-efficient when compared to [10]. This solution does use more space compared to our above implementation, with the benefit of having a significantly faster implementation of Algorithm 3. Thus, we offer two implementations – one using singly linked lists, and one using this array data structure – that together provide a trade-off between time and space used.
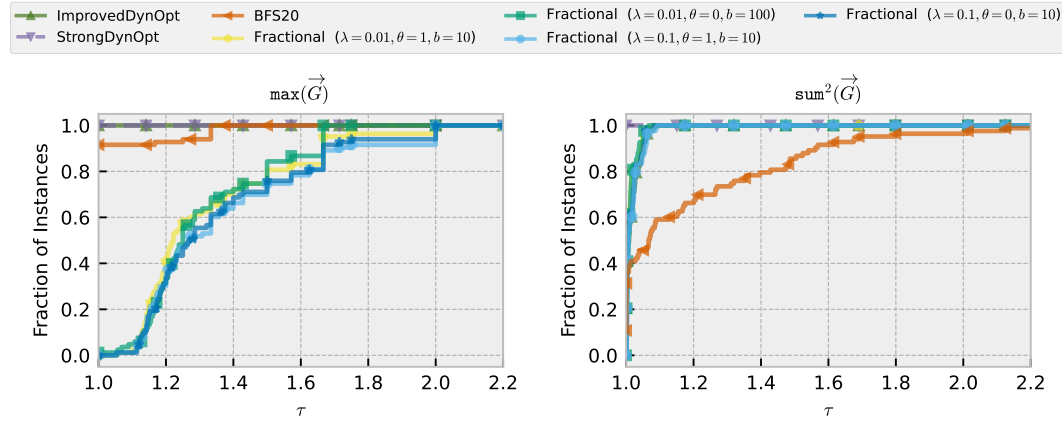
4   Experimental Evaluation

In this section we evaluate our algorithm and compare its performance with previous work by Großmann et al. [27] and Borowitz et al. [6]. We compare them on the sparse data set introduced in [6] and present a new dense data set featuring also deletions. The analysis is done in separate sections (sparse, dense) since the performance of the various algorithms highly depend on the density of used inputs.

Methodology.   We carried out the implementation of the algorithms in `C++` and compiled the programs with `g++-11.4`. We tested each algorithm on a machine equipped with 768 GB of RAM and an AMD EPYC 9754 128-Core Processor running at 2.25 GHz having a cache of 256 MB. The time loading the graphs is not measured, but instead we measure every operation and sum up the timings. We repeated every experiment 3 times, computations taking longer than a day in total were cancelled and are marked as out of time (OOT) in the result tables. For comparing the results we use performance profiles as introduced by Dolan and Moré [17]. For a factor of $\tau > 1$ we plot the fraction of instances that could be solved within that factor of the best result per instance. Algorithms in the top left corner of the plot (e.g. Figure 3) are considered the best algorithm.

Instances.   We use two sets of graphs for the experiments. First, the *sparse set* containing 83 instances collected by Borowitz et al. [6]. Only four of the instances feature deletions. The size of the graphs is small to medium and they have low-density and out-degree. Second, the *dense set*, that contains three, large, denser graphs sourced from the Suite Sparse Matrix collection [15] and four ultra-dense graphs generated with KaGen [22]. See also our supplemental details. The ultra-dense graphs were generated in the Erdos-Renyi model having 8192 vertices and between $\frac{1}{2}$ and $\frac{7}{8}$ of all possible edges. The graphs and their features are listed in the full version. For these instance we generated three different update sequences. The first, building up the graph in lexicographical ordering (LEX). The other two also include deletion of edges afterwards. The edges to be deleted are selected based on an optimal orientation of the final graph, computed with HeiOrient [38]. In the LEX+TOP50 configuration all edges outgoing from the upper half of vertices ordered by out-degree are deleted.

The LEX+OUT50 configuration is built by maintaining a heap of the vertices by out-degree. Outgoing edges are deleted and the out-degree is updated in the heap until the maximum out-degree is half of the optimal orientation. We select these sequences in order to see how different deletion strategies impact the quality, memory and runtime metrics.

Comparisons and Implementations.   We refer to our basic implementation of the algorithm by Chekuri et al. [10] as Fractional and with our engineering improvements as PackedFractional. We provide two implementations of PackedFractional. One with PackedFractional list organizes

the in-buckets in a singly linked list, and the one without suffix works with directly addressable, possibly empty, buckets. We compare with BFS20 by Borowitz et al. [6] and StrongDynOpt and ImprovedDynOpt by Großmann et al. [27]. These algorithms are the best performing in the extensive studies in [6] and [27] with regards to the max out-degree metric $\max(\overrightarrow{G})$. The $\mathrm{sum}^2(\overrightarrow{G})$ metric has not been studied in practice. The BFS20 algorithm performs a limited breadth-first-search of length up to 20 to find improving paths upon insertion and removes only the edge on deletion, not optimizing the orientation. The StrongDynOpt runs a breadth-first-search to find improving paths on every insertion and deletion, returning the optimal orientation concerning out-degree and squared-sum of out-degree. The ImprovedDynOpt only searches for improving paths if it is necessary to maintain the optimal out-degree.

Parameters. The algorithms Fractional and PackedFractional derived in Section 2.1 and Section 3 have the three parameters: λ, θ, b. First, $\lambda > 0$ and $\theta \in \{0, 1\}$ control the quality of the approximation by defining its λ-fairness. If $\theta = 0$, b must be larger than $\frac{1}{\lambda}$ so that the fractional orientation inequality can be satisfied. Otherwise ($\theta = 1$), b can be set to 1 making the orientation integral. This comes at costs of quality in the form of an additive error of $O(\log n)$ in the approximation.

4.1 Experiments on Sparse Instances

The following experiments are conducted on the sparse instance set collected by Borowitz et al. [6] and analyse the running times and quality of the orientations. Since some of graphs in this set were too small to consistently measure the maximum resident memory by repeatedly querying the linux internal process stats, we examine the memory consumption only for the dense instances. The maximum peak memory is 23 GB for the Fractional and PackedFractional with $\lambda = 0.01$ and $\theta = 0$ implementations on the LJOURNAL-2008 instance. The PackedFractional list algorithm with $\lambda = 0.1$, $\theta = 1$ only requires 4.4 GB on this instance - only 2.29 times the amount of StrongDynOpt.
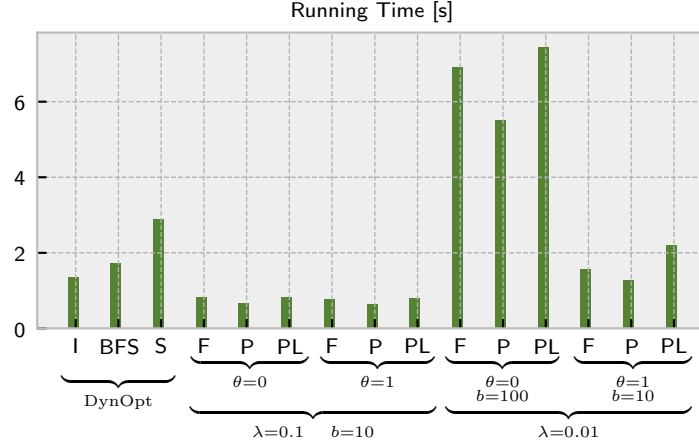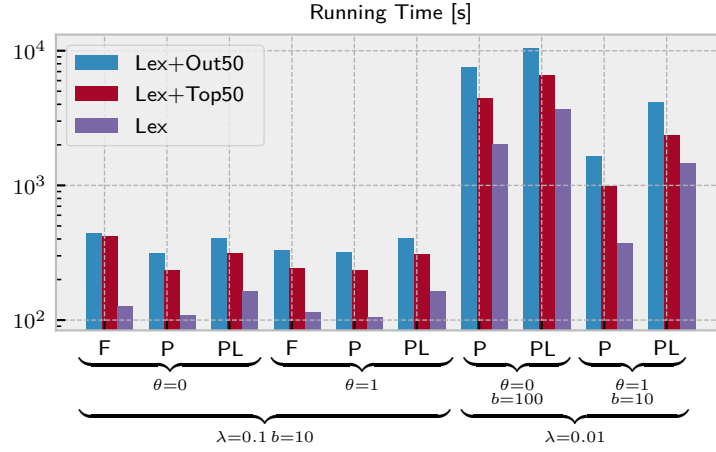
Figure 4 Running times on the sparse set comparing Fractional (F), PackedFractional (P), and PackedFractional, list (PL) with BFS20 (BFS), ImprovedDynOpt (I), and StrongDynOpt (S). Parameters: λ overall fairness of the orientation maintained. θ is a binary variable deciding an additional $O(\log n)$ error, b is the number of fractional edges.

Quality Metrics. In Figure 3 we compare different configurations of Fractional with the state-of-the-art for integral orientations. We evaluate the performance using the two quality measures discussed in Section 2. For better readability, the results for the packed implementations are omitted. They achieve similar to those results of the Fractional configurations with a maximum difference in quality of less than 0.3%. We can observe that smaller values of λ lead to improved solution quality for both metrics.

Maximum Out-Degree. When comparing the algorithms using the metric $\max(\overrightarrow{G})$, we can observe that both exact approaches ImprovedDynOpt and StrongDynOpt always find the optimum solution. Compared to these, Fractional performs worse overall, but maintains a factor of less than two for all parameter choices.

Squared Sum of Out-Degrees. When evaluating with $\mathrm{sum}^2(\overrightarrow{G})$ of the naively rounded fractional orientations, Fractional performs significantly better than the BFS20 algorithm, matching the results of ImprovedDynOpt. The StrongDynOpt algorithm returns the best values on every instance. This is the expected behaviour, because improving all possible paths minimizes both objective functions. Our newly engineered algorithms perform very well in the $\mathrm{sum}^2(\overrightarrow{G})$ metric as they were designed for it. The variation in $\lambda \in \{0.01, 0.1\}$ is not visible in this metric. In exchange, they allow themselves a higher maximum out-degree on some of the vertices which is not accounted for in the definition of λ-fair.

Running Time. Figure 4 shows the geometric mean running time for the approaches on the sparse data set. The normalized running times can be found in the full version. Our fastest algorithm PackedFractional is twice as fast as ImprovedDynOpt, which is the fastest state-of-the-art method on this data set. The running time of our approach is highly dependent on the parameter choices. Setting $\theta = 1$ is faster than $\theta = 0$. Decreasing λ results in longer running times. The smallest value $\lambda = 0.01$ we tested is 8.58 times slower than the fastest configuration with $\lambda = 0.1$. The packed list implementation is up to 35 % slower. The measured difference between ImprovedDynOpt and BFS20 matches the result in [27].

Figure 5 Geometric mean running times on the dense set comparing `Fractional` (F), `PackedFractional` (P), and `PackedFractional, list` (PL). We evaluate them for the different deletion sequences.

4.2 Experiments on Dense Instances

We conduct the following experiments on the new dense instance set, which includes three real-world graphs and four ultra-dense graphs. As in previous sections, we analyze running time and solution quality, and additionally provide an in-depth examination of memory usage.

Quality Metrics. Only three out of the 21 instances were solved optimally by `StrongDynOpt` on the set of dense instances in the given timeout of 24 hours. Therefore, we show performance profiles for our parameter choices and `PackedFractional` in the full version on all instances and limited the following discussion about quality on those three instances.

Maximum Out-Degree. Evaluating the performance using $\max(\overrightarrow{G})$ the solutions of `Fractional` are between 24 % ($\lambda = 0.01$) up to 30 % ($\lambda = 0.1$) worse than the optimal orientations. The method `BFS20` also performs worse on these instances with solutions being 26 % worse. Note that this approach does not handle deletions explicitly by running a improving path search, which can explain the observed suboptimal results on our instances featuring deletions. These results match the observations of previous work.

Squared Sum of Out-Degrees. Using $\mathrm{sum}^2(\overrightarrow{G})$, we observe a solution quality for `PackedFractional` which is less than 0.8 % higher for $\lambda = 0.1$ and only 0.3 % higher when setting $\lambda = 0.01$ than the optimum computed by `StrongDynOpt`. With this, we achieve better results on these instances compared to `ImprovedDynOpt`, which in the geometric mean has a solution quality that is 15 % worse. The `ImprovedDynOpt` is specialised in maintaining the minimum maximum out-degree while requiring fewer operations than `StrongDynOpt`. Therefore, its $\mathrm{sum}^2(\overrightarrow{G})$ metric is worse than our results. The impact of the λ factor, in the range we tested, on the $\mathrm{sum}^2(\overrightarrow{G})$ metric is small for our new algorithms.

Running Time. The running times on the dense set for our implementations are displayed in Figure 5. We detail these results in the full version. Detailed results per instance are shown in the full version. The choice of lambda greatly impacts the running time. Reducing the value of λ by a factor of 10 from 0.1 to 0.01 (keeping $\theta = 0$) can result in running time that is 20 times slower in the geometric mean. Setting $\theta = 1$ speeds up the algorithm by up to a factor of 5 for small $\lambda = 0.01$ compared to $\theta = 0$. The methods from Borowitz et al. [6]

and Großmann et al. [27] find solutions for considerably fewer number of instances in this data set while requiring more running time. More precisely, BFS20, ImprovedDynOpt, and StrongDynOpt find solutions on 15, 5, and 3 instances while having a geometric mean running time which is 112.17, 29.15, and 25.98 times slower than our fastest algorithm (PackedFractional) on the respective instances.

Memory. As a smaller λ value results in more buckets, we expect the memory consumption to increase with decreasing λ. This behaviour can be observed in Figure 6. Here, we compare the memory consumption of our implementations, which is evaluated separately for the different deletion sequences. Already for $\lambda = 0.1$, we can observe that the PackedFractional is improving over Fractional, where we achieve a geometric mean reduction in memory consumption by a factor of 1.94. In the case of $\lambda = 0.01$, we omit Fractional due to its already high memory consumption for $\lambda = 0.1$ and focus on comparing the implementations PackedFractional and PackedFractional, list. With this configuration, the List implementation can save up to a factor of 2.72 memory ($\lambda = 0.01, \theta = 0, b = 100$). Overall, we can see that the choice for λ has the highest impact regarding memory consumption. However, the parameters θ and b also influence the memory needed. The memory consumption for the competitor algorithms is generally smaller. Compared to our best approach, the competitor methods perform up to a factor of 3.32 better. However, this result is expected since they do not maintain a fractional orientation but an integral one. They store each edge's target on both sides in combination with the position of the reverse edge in the other array. Lastly, we observe that the deletion of edges (Top50, Out50) requires more memory than just the Lex ordering, with the Out50 requiring the most. This is caused by the spreading of indices in the in-bucket, as seen by the smaller difference in the list implementations.

> **Practitioner Advice:** When optimizing the sum of squared out-degrees we recommend using the packed PackedFractional implementation since it is the fastest implementation overall and the difference in quality to the exact method (StrongDynOpt) is minimal. The list implementation should only be used if memory consumption is of utmost importance. The choice of parameters is paramount. Choosing higher values for λ speeds up the computation and reduces the memory needed. Setting $\theta = 1$ can be used to relax the running time further, but is penalized by a quality trade-off. The b parameter should be chosen small to speed up computation, but has to stay above $\frac{1}{\lambda}$ if $\theta = 0$.

5   Conclusion

We have bridged the gap between theoretical results and practical experiments for adaptive edge orientations. We engineered a space efficient implementation of a λ-fair fractional orientation algorithm by Chekuri et al. [10] and presented several effective methods to reduce the memory footprint of the algorithm. In extensive experiments we show the feasibility of our approach in comparison with several competitors from previous work. Our results for the squared sum degree metric are close to optimal, while the running time is considerably lower than competing algorithms. Our algorithm PackedFractional solves more instances than the competitors within one day of compute time per instance while it is up to 112 times faster on the instances that are solvable by all methods compared. Our list-based bucket implementations can decrease the memory consumption by a factor of 2.72. Future avenues of work include reducing the memory costs further and looking at batch updates. These aspects could be combined with efforts to parallelize the algorithms.
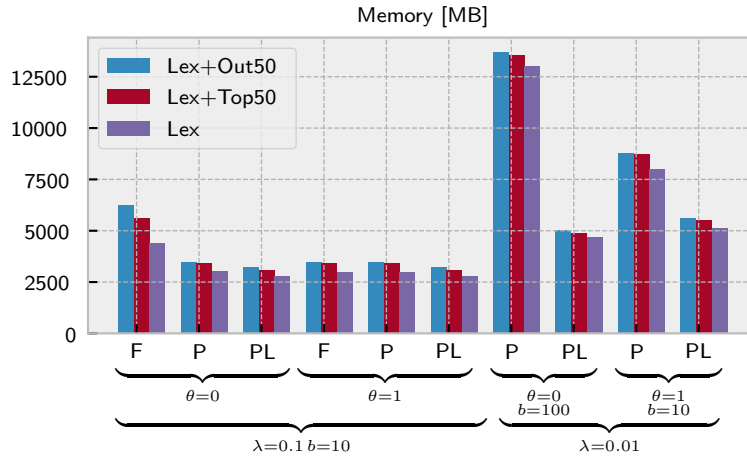
Figure 6 Memory consumption of Fractional (F), PackedFractional (P), and PackedFractional, list (PL) for different configurations evaluated separately for the different deletions sequences.

References

1   J. Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. Large scale networks fingerprinting and visualization using the k-core decomposition. In *Advances in Neural Information Processing Systems 18 [Neural Information Processing Systems, NIPS 2005, December 5-8, 2005, Vancouver, British Columbia, Canada]*, volume 18, pages 41–50, 2005. URL: `https://proceedings.neurips.cc/paper/2005/hash/b19aa25ff58940d974234b48391b9549-Abstract.html`.

2   Albert Angel, Nick Koudas, Nikos Sarkas, Divesh Srivastava, Michael Svendsen, and Srikanta Tirthapura. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. *The VLDB journal*, 23:175–199, 2014. `doi:10.1007/s00778-013-0340-z`.

3   Gary D. Bader and Christopher W. V. Hogue. An automated method for finding molecular complexes in large protein interaction networks. *BMC Bioinform.*, 4:2, 2003. `doi:10.1186/1471-2105-4-2`.

4   Suman K. Bera, Sayan Bhattacharya, Jayesh Choudhari, and Prantar Ghosh. A new dynamic algorithm for densest subhypergraphs. In Frédérique Laforest, Raphaël Troncy, Elena Simperl, Deepak Agarwal, Aristides Gionis, Ivan Herman, and Lionel Médini, editors, *WWW '22: The ACM Web Conference 2022, Virtual Event, Lyon, France, April 25 - 29, 2022*, pages 1093–1103. ACM, 2022. `doi:10.1145/3485447.3512158`.

5   Edvin Berglin and Gerth Stølting Brodal. A Simple Greedy Algorithm for Dynamic Graph Orientation. *Algorithmica*, 82(2):245–259, 2020. `doi:10.1007/s00453-018-0528-0`.

6   Jannick Borowitz, Ernestine Großmann, and Christian Schulz. Engineering Fully Dynamic Δ-orientation Algorithms. In *ACDA*, pages 25–37, 2023. `doi:10.1137/1.9781611977714.3`.

7   Glencora Borradaile, Jennifer Iglesias, Theresa Migler, Antonio Ochoa, Gordon Wilfong, and Lisa Zhang. Egalitarian graph orientations. *Journal of Graph Algorithms and Applications*, 2017. `doi:10.7155/jgaa.00435`.

8   Glencora Borradaile, Theresa Migler, and Gordon Wilfong. Density decompositions of networks. In *Conference on Complex Networks (CompleNet)*. Springer, 2018. `doi:10.7155/jgaa.00505`.

9   Gerth Stølting Brodal and Rolf Fagerberg. Dynamic Representation of Sparse Graphs. In Frank K. H. A. Dehne, Arvind Gupta, Jörg-Rüdiger Sack, and Roberto Tamassia, editors, *Algorithms and Data Structures, 6th International Workshop, WADS '99, Vancouver, British Columbia, Canada, August 11-14, 1999, Proceedings*, volume 1663 of *Lecture Notes in Computer Science*, pages 342–351, 1999. `doi:10.1007/3-540-48447-7_34`.

10   Chandra Chekuri, Aleksander Bjørn Christiansen, Jacob Holm, Ivor van der Hoog, Kent Quanrud, Eva Rotenberg, and Chris Schwiegelshohn. Adaptive out-orientations with applications. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3062–3088. SIAM, 2024. `doi:10.1137/1.9781611977912.110`.

11   Jie Chen and Yousef Saad. Dense subgraph extraction with application to community detection. *IEEE Trans. Knowl. Data Eng.*, 24(7):1216–1230, 2012. `doi:10.1109/TKDE.2010.271`.

12   Aleksander B. G. Christiansen and Eva Rotenberg. Fully-Dynamic $\alpha + 2$ Arboricity Decompositions and Implicit Colouring. In Mikołaj Bojańczyk, Emanuela Merelli, and David P. Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming (ICALP 2022)*, volume 229 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 42:1–42:20, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ICALP.2022.42`.

13   Aleksander Bjørn Christiansen, Ivor van der Hoog, and Eva Rotenberg. Local Density and Its Distributed Approximation. In Olaf Beyersdorff, Michał Pilipczuk, Elaine Pimentel, and Nguyễn Kim Thang, editors, *Theoretical Aspects of Computer Science (STACS)*, volume 327 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:20, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.STACS.2025.25`.

14   Maximilien Danisch, T.-H. Hubert Chan, and Mauro Sozio. Large scale density-friendly graph decomposition via convex programming. In Rick Barrett, Rick Cummings, Eugene Agichtein, and Evgeniy Gabrilovich, editors, *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*, WWW '17, pages 233–242, Republic and Canton of Geneva, CHE, 2017. ACM. `doi:10.1145/3038912.3052619`.

15   Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1–25, December 2011. `doi:10.1145/2049662.2049663`.

16   Laxman Dhulipala, Quanquan C Liu, Julian Shun, and Shangdi Yu. Parallel batch-dynamic k-clique counting. In *Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 129–143. SIAM, 2021. `doi:10.1137/1.9781611976489.10`.

17   Elizabeth D. Dolan and Jorge J. Moré. Benchmarking Optimization Software With Performance Profiles. *Math. Program.*, 91(2):201–213, 2002. `doi:10.1007/s101070100263`.

18   Yon Dourisboure, Filippo Geraci, and Marco Pellegrini. Extraction and classification of dense communities in the web. In *Proceedings of the 16th international conference on World Wide Web*, pages 461–470, 2007. `doi:10.1145/1242572.1242635`.

19   Xiaoxi Du, Ruoming Jin, Liang Ding, Victor E Lee, and John H Thornton Jr. Migration motif: a spatial-temporal pattern mining approach for financial markets. In *Proceedings of the 15th ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144, 2009. `doi:10.1145/1557019.1557142`.

20   András Frank and Kazuo Murota. Fair integral network flows. *Mathematics of Operations Research*, 2023. `doi:10.1287/moor.2022.1303`.

21   Eugene Fratkin, Brian T. Naughton, Douglas L. Brutlag, and Serafim Batzoglou. Motifcut: regulatory motifs finding with maximum density subgraphs. In *Proceedings 14th International Conference on Intelligent Systems for Molecular Biology 2006, Fortaleza, Brazil, August 6-10, 2006*, volume 22, pages 156–157. Oxford University Press, 2006. `doi:10.1093/bioinformatics/btl243`.

22   Daniel Funke, Sebastian Lamm, Ulrich Meyer, Manuel Penschuck, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. Communication-free massively distributed graph generation. *J. Parallel Distributed Comput.*, 131:200–217, 2019. `doi:10.1016/J.JPDC.2019.03.011`.

23   David Gibson, Jon Kleinberg, and Prabhakar Raghavan. Inferring web communities from link topology. In *Proceedings of the ninth ACM conference on Hypertext and hypermedia: links, objects, time and space-structure in hypermedia systems: links, objects, time and space-structure in hypermedia systems*, pages 225–234, 1998. `doi:10.1145/276627.276652`.

24 David Gibson, Ravi Kumar, and Andrew Tomkins. Discovering large dense subgraphs in massive graphs. In *Proceedings of the 31st international conference on Very large data bases*, pages 721–732, 2005. URL: `http://www.vldb.org/archives/website/2005/program/paper/thu/p721-gibson.pdf`.

25 Aristides Gionis and Charalampos E Tsourakakis. Dense subgraph discovery: Kdd 2015 tutorial. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2313–2314, 2015. `doi:10.1145/2783258.2789987`.

26 Ernestine Grossmann, Henrik Reinstädtler, Eva Rotenberg, Christian Schulz, Ivor van der Hoog, and Juliette Vlieghe. Implementation of our dynamic algorithm for minimum orientation. Software, Villum Fonden VIL37507, DFG SCHU 2567/8-1, Marie Skłodowska-Curie 899987, sw-hId: `swh:1:dir:0ea52feafca9f6d8ce3ee893686fde2510513e9e` (visited on 2025-09-04). URL: `https://github.com/DynGraphLab/DynDeltaApprox`, `doi:10.4230/artifacts.24670`.

27 Ernestine Großmann, Henrik Reinstädtler, Christian Schulz, and Fabian Walliser. Engineering fully dynamic exact-orientation algorithms. In *2025 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 15–28. SIAM, 2025. `doi:10.1137/1.9781611978339.2`.

28 Patric Hagmann, Leila Cammoun, Xavier Gigandet, Reto Meuli, Christopher J Honey, Van J Wedeen, and Olaf Sporns. Mapping the structural core of human cerebral cortex. *PLoS biology*, 6(7):e159, 2008.

29 Elfarouk Harb, Kent Quanrud, and Chandra Chekuri. Faster and scalable algorithms for densest subgraph and decomposition. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, volume 35, pages 26966–26979, 2022. URL: `http://papers.nips.cc/paper_files/paper/2022/hash/ac8fbba029dadca99d6b8c3f913d3ed6-Abstract-Conference.html`.

30 Jon M Kleinberg et al. Authoritative sources in a hyperlinked environment. In Howard J. Karloff, editor, *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, 25-27 January 1998, San Francisco, California, USA*, volume 98, pages 668–677. Citeseer, ACM/SIAM, 1998. URL: `http://dl.acm.org/citation.cfm?id=314613.315045`.

31 Jon M Kleinberg, Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew S Tomkins. The web as a graph: Measurements, models, and methods. In *International Computing and Combinatorics Conference*, pages 1–17. Springer, 1999. `doi:10.1007/3-540-48686-0_1`.

32 Tsvi Kopelowitz, Robert Krauthgamer, Ely Porat, and Shay Solomon. Orienting Fully Dynamic Graphs with Worst-case Time Bounds. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*, volume 8573 of *Lecture Notes in Computer Science*, pages 532–543, 2014. `doi:10.1007/978-3-662-43951-7_45`.

33 Lukasz Kowalik and Maciej Kurowski. Oracles for Bounded-length Shortest Paths in Planar Graphs. *ACM Trans. Algorithms*, 2(3):335–363, 2006. `doi:10.1145/1159892.1159895`.

34 Ravi Kumar, Jasmine Novak, and Andrew Tomkins. Structure and evolution of online social networks. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 611–617, 2006. `doi:10.1145/1150402.1150476`.

35 Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks V. S. Lakshmanan, Wenjie Zhang, and Xuemin Lin. Efficient algorithms for densest subgraph discovery on large directed graphs. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 1051–1066. ACM, 2020. `doi:10.1145/3318464.3389697`.

36 Ofer Neiman and Shay Solomon. Simple Deterministic Algorithms for Fully Dynamic Maximal Matching. *ACM Trans. Algorithms*, 12(1):7:1–7:15, 2016. `doi:10.1145/2700206`.

37 Jean-Claude Picard and Maurice Queyranne. A network flow solution to some nonlinear 0-1 programming problems, with applications to graph theory. *Networks*, 12(2):141–159, 1982. `doi:10.1002/NET.3230120206`.

38 Henrik Reinstädtler, Christian Schulz, and Bora Uçar. HeiOrient. Software, DFG-SCHU 2567/3-1, swhId: `swh:1:dir:be7317d125554a54dfd1c9d17521c500c4e1ebc3` (visited on 2024-11-28). `doi:10.4230/artifacts.22501`.

39 Barna Saha, Allison Hoch, Samir Khuller, Louiqa Raschid, and Xiao-Ning Zhang. Dense subgraphs with restrictions and applications to gene annotation graphs. In *Annual International Conference on Research in Computational Molecular Biology*, pages 456–472. Springer, 2010. `doi:10.1007/978-3-642-12683-3_30`.

40 Saurabh Sawlani and Junxing Wang. Near-optimal fully dynamic densest subgraph. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 181–193, 2020. `doi:10.1145/3357713.3384327`.

41 Mauro Sozio and Aristides Gionis. The community-search problem and how to plan a successful cocktail party. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 939–948, 2010. `doi:10.1145/1835804.1835923`.

42 Bintao Sun, Maximilien Danisch, T.-H. Hubert Chan, and Mauro Sozio. Kclist++: A simple algorithm for finding k-clique densest subgraphs in large graphs. *Proc. VLDB Endow.*, 13(10):1628–1640, 2020. `doi:10.14778/3401960.3401962`.

43 Yichen Xu, Chenhao Ma, Yixiang Fang, and Zhifeng Bao. Efficient and effective algorithms for generalized densest subgraph discovery. *Proceedings of the ACM on Management of Data*, 1(2):1–27, 2023. `doi:10.1145/3589314`.

44 Bin Zhang and Steve Horvath. A general framework for weighted gene co-expression network analysis. *Statistical applications in genetics and molecular biology*, 4(1), 2005.

45 Yalong Zhang, Ronghua Li, Qi Zhang, Hongchao Qin, and Guoren Wang. Efficient algorithms for density decomposition on large static and dynamic graphs. *Proc. VLDB Endow.*, 17(11):2933–2945, 2024. `doi:10.14778/3681954.3681974`.