Minimum Chain Cover in Almost Linear Time

Manuel Cáceres
Department of Computer Science, University of Helsinki, Finland

Abstract

A minimum chain cover (MCC) of a k-width directed acyclic graph (DAG) $G = (V, E)$ is a set of k chains (paths in the transitive closure) of G such that every vertex appears in at least one chain in the cover. The state-of-the-art solutions for MCC run in time $\tilde{O}(k(|V| + |E|))$ [Mäkinen et at., TALG], $O(T_{MF}(|E|) + k|V|)$, $O(k^2|V| + |E|)$ [Cáceres et al., SODA 2022], $\tilde{O}(|V|^{3/2} + |E|)$ [Kogan and Parter, ICALP 2022] and $\tilde{O}(T_{MCF}(|E|) + \sqrt{k}|V|)$ [Kogan and Parter, SODA 2023], where $T_{MF}(|E|)$ and $T_{MCF}(|E|)$ are the running times for solving maximum flow (MF) and minimum-cost flow (MCF), respectively.

In this work we present an algorithm running in time $O(T_{MF}(|E|) + (|V| + |E|)\log k)$. By considering the recent result for solving MF [Chen et al., FOCS 2022] our algorithm is the first running in almost linear time. Moreover, our techniques are deterministic and derive a deterministic near-linear time algorithm for MCC if the same is provided for MF. At the core of our solution we use a modified version of the mergeable dictionaries [Farach and Thorup, Algorithmica], [Iacono and Özkan, ICALP 2010] data structure boosted with the SIZE-SPLIT operation and answering queries in amortized logarithmic time, which can be of independent interest.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Theory of computation → Network flows; Theory of computation → Sorting and searching

Keywords and phrases Minimum chain cover, directed acyclic graph, minimum flow, flow decomposition, mergeable dictionaries, amortized running time

Digital Object Identifier 10.4230/LIPIcs.ICALP.2023.31

Category Track A: Algorithms, Complexity and Games

Related Version *Previous Version*: `https://arxiv.org/abs/2305.02166`

Funding This work was funded by the Academy of Finland (grants No. 352821, 328877).

Acknowledgements I am very grateful to Alexandru I. Tomescu and Brendan Mumey for initial discussions on flow decomposition of a minimum flow representing an MPC. I am also very grateful to the anonymous reviewers for their useful suggestions.

1 Introduction

Computing a minimum-sized set of chains covering all vertices of a DAG $G = (V, E)$ is a well known poly-time solvable problem [15, 18], with many applications in widespread research fields such as bioinformatics [34, 7, 13, 4, 8, 32]. Here we call such an object a *minimum chain cover* (an MCC) $\mathcal{C}$ containing k chains $\mathcal{C} = \{C_1, \ldots, C_k\}$, which are paths in the transitive closure of G. The *size* k of an MCC is known as the *width* of G and equals the maximum number of pairwise unreachable vertices (antichain) of G, by Dilworth's theorem [15] on partially ordered sets (posets).

The history of MCC. It was Fulkerson [18] in the 1950s the first to show a poly-time algorithm for posets (transitive DAGs). His algorithm reduces the problem to finding a maximum matching in a bipartite graph with $2|V|$ vertices and $|E|$ edges, and thus can be solved in $O(|E|\sqrt{|V|})$ time by using the Hopcroft-Karp algorithm [21][1]. Improvements on these ideas were derived in the $O(|V|^2 + k\sqrt{k}|V|)$ and $O(\sqrt{|V|}|E| + k\sqrt{k}|V|)$ time algorithms of Chen and Chen [10, 11], and the $O(k|V|^2)$ time algorithm of Felsner et al. [17]. In the same article, Felsner et al. showed a combinatorial approach to compute a *maximum antichain* (MA) in near-linear time for the cases $k = 2, 3, 4$, which was latter generalized to $O(f(k)(|V| + |E|))$ [5], $f(k)$ being an exponential function. State-of-the-art approaches improve exponentially on its running time dependency on k. These approaches solve the strongly related problem of *minimum path cover* (MPC). An MPC $\mathcal{P}$ is a minimum-sized set of *paths* covering the vertices of G, and thus it is also a valid chain cover. Moreover, since every MCC can be transformed into an MPC (by connecting consecutive vertices in the chains), the *size* of an MPC also equals the width k.

The state-of-the-art for MCC. Mäkinen et al. [29] provided an algorithm for MPC, running in time $O(k(|V| + |E|)\log|V|) = \widetilde{O}(k(|V| + |E|))$ while Cáceres et al. [6] presented the first $O(k^2|V| + |E|)$ parameterized linear time algorithm. Both algorithms are based on a classical reduction to *minimum flow* [30], which we will revisit later. In the same work, Cáceres et al. [6] showed how to compute an MPC in time $O(T_{MF}(|E|) + ||\mathcal{P}||)$ and a MA in time $O(T_{MF}(|E|))$, where $T_{MF}(|E|)$ is the running time for solving *maximum flow* (MF) and $||\mathcal{P}||$ is the total length of the reported MPC. As such, by using the recent result for MF of Chen et al. [9] we can solve MPC and MA in (almost) optimal (input+output size) time. However, the same does not apply to MCC as the total length of an MCC can be exactly $|V|$ (e.g. by removing repeated vertices) while the total length of an MPC can be $\Omega(k|V|)$ in the worst case, as shown in Figure 1.

The $k|V|$ barrier was recently overcame by Kogan and Parter [26, 27] by reducing the total length of an MPC. They obtain this improvement by using *reachability shortcuts* and by devising a more involved reduction to *minimum cost flow* (MCF). Their algorithms run in time $\widetilde{O}(|E| + |V|^{3/2})$ [26] and $\widetilde{O}(\sqrt{k}|V| + |E|^{1+o(1)})$ [27] using the MCF algorithms of Bran et al. [35] and Chen et al. [9], respectively.

In this paper we present an algorithm for MCC improving its running time dependency on k exponentially w.r.t. the state-of-the-art.

▶ **Theorem 1.** *Given a k-width DAG $G = (V, E)$ we can compute a minimum chain cover in time $O(T_{MF}(|E|) + (|V| + |E|)\log k)$, where $T_{MF}(|E|)$ is the time for solving maximum flow.*

Thus by applying the flow algorithm of Chen et al. [9] we solve the problem in almost-linear time for the first time.

▶ **Corollary 2.** *Given a k-width DAG $G = (V, E)$ we can compute a minimum chain cover in time $O(|E|^{1+o(1)})$ w.h.p.*

Moreover, our solution in Theorem 1 uses a MF solver as a black box and it is deterministic otherwise. Therefore, we provide a deterministic MCC solution in *near-linear time* if one is found for MF. At the core of our solution we use mergeable dictionaries boosted with the SIZE-SPLIT operation to efficiently transform the flow outputted by the MF solver into an MCC. *Mergeable dictionaries* is a data structure maintaining a dynamic partition $\mathcal{K}$ of the natural numbers $\{1, \ldots, k\}$ (the reuse of k is intentional as this is how our approach will use the data structure), starting from the partition $\mathcal{K} = \{\{1, \ldots, k\}\}$ and supporting the following operations, for $K, K_1, K_2 \in \mathcal{K}$:

[1] Recent fast solutions for maximum matching do not speed up this approach as one needs to compute the transitive closure of the DAG first.

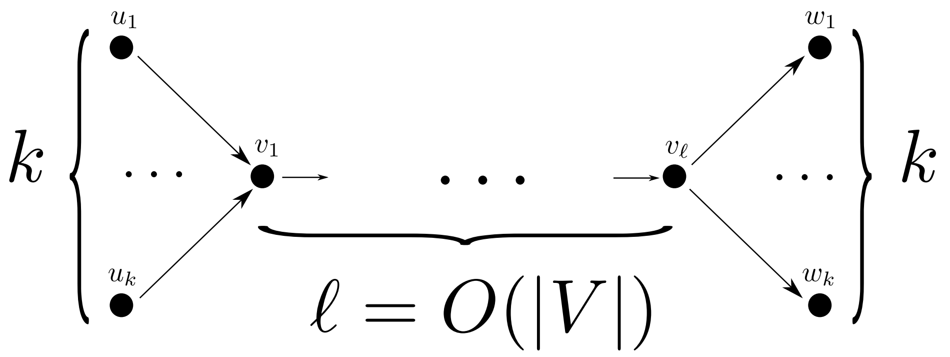

Figure 1 Example k-width DAG where every MPC $\mathcal{P}$ has total length $||\mathcal{P}|| = \Omega(k|V|)$. Indeed, every path in an MPC must start from some u_i and traverse the middle path $v_1, \ldots, v_\ell$ until reaching some w_j, since otherwise it is not possible to cover the rest of the graph minimally. Moreover, if $k = \ell = |V|/3$, then $||\mathcal{P}|| = \Omega(|V|^2)$. On the other hand, there is always an MCC $\mathcal{C}$ of total size $||\mathcal{C}|| = |V|$ (in this case only one of the chains cover the vertices of the middle path).

- SEARCH(K, j): returns $\max_{i \in K, i \leq j} i$ if any such element exists[2].
- MERGE(K_1, K_2): replaces K_1 and K_2 by $K_1 \cup K_2$ in $\mathcal{K}$.
- SPLIT(K, j): replaces K by $K' = \{i \in K \mid i \leq j\}$ (only if $K' \neq \emptyset$) and $K \setminus K'$ (only if $K \neq K'$) in $\mathcal{K}$.

Note that the MERGE operation does not assume that $\max_{i \in K_1} i < \min_{i \in K_2} i$ (sets are non-overlapping) as generated by the SPLIT operation. If the previous condition (or the analogous $\max_{i \in K_2} i < \min_{i \in K_1} i$) is assumed, the operation is known as JOIN. Mergeable dictionaries have applications in Lempel-Ziv decompression [16, 3, 31], mergeable trees [19] and generalizations of union-find-split [28]. In this paper we show how to modify them to obtain a fast MCC algorithm. Next, we review the different approaches used to implement mergeable dictionaries in the literature.

Mergeable dictionaries. The first efficient implementation of mergeable dictionaries can be derived[3] from the *segment merge* strategy proposed by Farach and Thorup [16] to efficiently MERGE two self-balanced binary search trees, assuming that operations SEARCH, SPLIT and JOIN are implemented in logarithmic time. The authors showed how to implement the MERGE operation by minimally SPLITing both sets into non-overlapping sets and pairwise JOINing the resulting parts. They proved that after $|V|+|E|$ operations (the abuse of notation is again intentional) their strategy works in $O(\log k \cdot \log(|V| + |E|))$ amortized time per operation. Later, Iacono and Özkan [22] presented a mergeable dictionaries implementation running in $O(\log k)$ amortized time per operation, based on biased skip lists [1]. The same amortized running time was later achieved by Karczmarz [24] with a very simple approach using tries [14] to represent the sets.

[2] This query is also known as *predecessor* query in the literature.
[3] The authors of [16] do not define mergeable dictionaries formally.

Our algorithm for MCC computes a minimum flow f^* encoding an MPC and then extracts an MCC from f^* by processing G in topological order and querying mergeable dictionaries boosted with the SIZE-SPLIT operation. Formally, we require the following operations, for $K, K_1, K_2 \in \mathcal{K}, s \in \{1, \dots, |K| - 1\}$:

- SOME(K): returns some element $i \in K$.
- MERGE(K_1, K_2): replaces K_1 and K_2 by $K_1 \cup K_2$ in $\mathcal{K}$.
- SIZE-SPLIT(K, s): replaces K by $K' \subseteq K, |K'| = s$ and $K \setminus K'$ in $\mathcal{K}$.

In Section 3 we show how to implement these operations in $O(\log k)$ amortized time each. Then, in Section 4 we show our MCC algorithm using the previously described data structure. Besides the theoretical improvement already explained, we highlight the simplicity of our solutions, both in our proposal for boosted mergeable dictionaries as well as in our algorithm for MCC.

2 Notation and preliminaries

Graphs. For a vertex $v \in V$ we denote $N^-(v)$ ($N^+(v)$) to be the set of *in(out)-neighbors* of v that is $N^-(v) = \{u \in V \mid (u, v) \in E\}$ ($N^+(v) = \{w \in V \mid (v, w) \in E\}$). A v_1v_ℓ-path is a sequence of vertices $P = v_1, \dots, v_\ell$ such that $(v_i, v_{i+1}) \in E$ for $i \in \{1, \dots, \ell - 1\}$, in this case we say that v_1 *reaches* v_ℓ. We say that P is *proper* if $\ell \geq 2$ and that P is a cycle if $v_1 = v_\ell$. A *directed acyclic graph* (DAG) is a graph without proper cycles. In a DAG we can compute, in linear time [23, 33], a *topological order* $v_1, \dots, v_{|V|}$ of its vertices such that for every $i < j$, $(v_j, v_i) \notin E$. In this paper we assume that G is a DAG and since our algorithms run in time $\Omega(|V| + |E|)$, we assume that an input *topological order* is given. A chain is a sequence of vertices $C = v_1, \dots, v_{\ell'}$ such that for each $i \in \{1, \dots, \ell' - 1\}$ v_i reaches v_{i+1}. We denote $|C| = \ell'$ to the length of the chain. A *chain cover* $\mathcal{C}$ is a set of chains such that every vertex appears in some chain of $\mathcal{C}$. We say that it is a *chain decomposition* if every vertex appears in exactly one chain of $\mathcal{C}$ and a *path cover* if every chain of $\mathcal{C}$ is a path, in this case we denote it $\mathcal{P}$ instead. We denote $||\mathcal{C}||$ to the total length of a chain cover that is $||\mathcal{C}|| = \sum_{C \in \mathcal{C}} |C|$. An antichain is a subset of vertices $A \subseteq V$ such that for $u, v \in A, u \neq v$, u does not reach v. The minimum size of a chain cover is known as the *width* of G and we denote it k.

Flows. Given a function of *demands* $d : E \to \mathbb{N}_0$ and $s, t \in V$, an *st-flow* is a function $f : E \to \mathbb{N}_0$ satisfying *flow conservation* that is $inFlow_v := \sum_{u \in N^-(v)} f(u, v) = \sum_{w \in N^+(v)} f(v, w) := outFlow_v$ for each $v \in V \setminus \{s, t\}$, and *satisfying the demands* that is $f(e) \geq d(e)$ for each $e \in E$. A *flow decomposition* of k (the reuse of notation is again intentional) paths of f is a collection $\mathcal{D} = P_1, \dots, P_k$ of st-paths such that for each edge $e \in E$, $f(e) = |\{P_i \in \mathcal{D} \mid e \in P_i\}|$.[4] The *size* $|f|$ of f is defined as the net flow entering t (equivalently exiting s by flow conservation) that is $|f| = inFlow_t - outFlow_t$. The problem of *minimum flow* looks for a feasible st-flow of minimum size. Finding an MPC can be reduced to decompose a specific minimum flow [30], we will revisit this reduction in Section 4. The same techniques applied for the problem of *maximum flow* can be used in the context of the minimum flow problem [12, 2]. In fact, for MPC one can directly apply a maximum flow algorithm with *capacities* at most $|V|$ [6, Theorem 2.2 (full version)].

[4] This is a simplified definition of flow decomposition which suffices for our purposes.

Data structures. A *self-balancing binary search tree* such as an AVL-tree or a red-black tree [20] is a binary search tree supporting operations SEARCH, SPLIT and JOIN in logarithmic time each (in the worst case). A (binary) *trie* [14] is a binary tree representing a set of integers by storing their *binary representation* as *root-to-leaf* paths of the trie. In our tries all root-to-leaf paths have the same length $\lfloor \log k \rfloor + 1$. For a data structure supporting a set of operations, and a potential function ϕ capturing the state of the data structure, we say that the *amortized* time of an operation equals to its (worst-case) running time plus the change $\Delta\phi$ in the potential triggered by the operation. If we apply a sequence of $O(|V|+|E|)$ operations whose amortized time is $O(\log k)$ the total (worst-case) running time is $O((|V|+|E|)\log k)$.

3 Mergeable dictionaries with SIZE-SPLIT

We show how to implement the *boosted* mergeable dictionaries supporting operations SOME, MERGE and SIZE-SPLIT in $O(\log k)$ amortized time each. To achieve this result we modify an existing solution of mergeable dictionaries implementing operations SEARCH, MERGE and SPLIT by adding the SELECT operation. Formally for $K \in \mathcal{K}, s \in 1, \ldots, |K|$,

- SELECT(K, s): returns the s-th smallest element in K.

With the SELECT operation we can use (normal) mergeable dictionaries to implement SOME and SIZE-SPLIT as follows:

- SOME$(K) \leftarrow$ SEARCH(K, k).
- SIZE-SPLIT$(K, s) \leftarrow$ SPLIT$(K$,SELECT$(K, s))$.

While for SOME it suffices to do a SEARCH with a known upper bound (recall that k is the maximum element in the universe considered), in the case of SIZE-SPLIT we can first SELECT the corresponding pivot and use this pivot to SPLIT the set by its value, obtaining the desired sizes for the split.

This reduction allows us to obtain the boosted mergeable dictionaries by simply implementing the SELECT operation with logarithmic amortized cost (SIZE-SPLIT can be seen as one call to SELECT followed by a separate call to SPLIT). Moreover, if the implementation of SELECT does not modify the data structure (and thus the potential ϕ), its amortized time equals its (worst-case) running time (as ϕ does not change). We show that this is indeed the case in both the segment merge strategy of Farach and Thorup [16], and the trie implementation of Karczmarz [24], the latter achieving the desired running time.

The mergeable dictionaries based on segment merge represent each set as a self-balancing binary search tree. As such, the SELECT operation can be implemented in $O(\log k)$ time by storing the sub-tree sizes at every node of the tree, which can be maintained (updated when the tree changes) in the same (worst-case) running time as the normal operations of the tree (see e.g. [25]).

Similarly, the implementation of Karczmarz [24] represents every set as a trie of its elements. As discussed in Section 2, a trie stores its elements as their binary representation encoded as (equal length) root-to-leaf paths of the trie. For example, if $k = 6$ the corresponding binary representation of 3 is 011, which is represented as the root-to-leaf path following the left child, then its right child and then its right child. Analogous to binary search trees, the nodes of a trie can be augmented to store the number of leaves in their respective sub-trees. If such augmentation of a trie is performed, then the operation SELECT can be implemented in $O(\log k)$ (worst-case) time similar to the implementation on binary search trees, since the leaves in the trie follow the same order as the elements they represent:

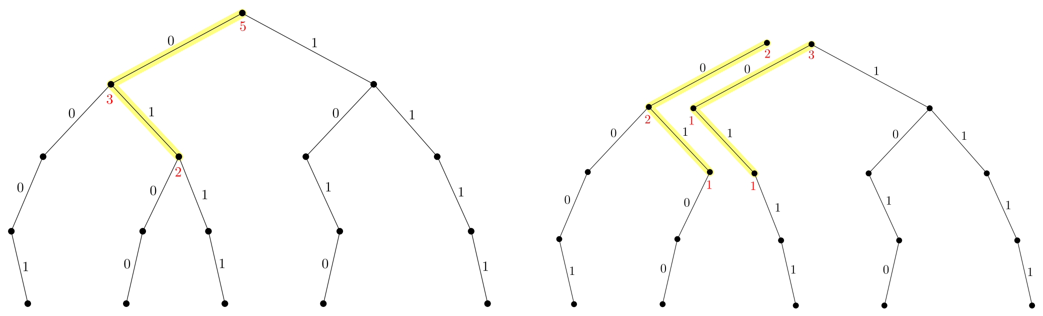

Figure 2 Result of calling SIZE-SPLIT($K = \{1, 4, 7, 10, 15\}, 2$) on a trie representation of boosted mergeable dictionaries. The binary representation of numbers in the set are spelled as root-to-leaf paths of the trie. Some number of leaves' counters are written in red under their respective nodes. Those counters are used to answer $4 \leftarrow$ SELECT($K, 2$). After this, SPLIT($K, 4$) is performed. The path P representing the longest prefix (01) between the binary representations of 4 (0100) and 7 (0111) is highlighted in yellow. The right figure shows the end result, note that only the counters in the nodes of P change.

for a query SELECT(s) on a trie node, it suffices to look at the number of leaves l (elements) under the left child, if $l \geq s$ we continue to the left child answering SELECT(s), otherwise we continue to the right child answering SELECT($s - l$).

Karczmarz [24] showed that SPLIT(K, j) can be performed by finding the root-to-node path P corresponding to the longest common prefix between the binary representations of SEARCH(K, j) and of the smallest value greater than j in K. It then splits the trie by removing the right children of the nodes of P (to form K') and then joining those nodes as right children of a copy of P (to create $K \setminus K'$). Note that this procedure can be easily augmented to maintain the number of leaves under each node: only the nodes in P (and in its copy) decrease their value by the number of leaves of their lost children. Figure 2 shows an example of the SPLIT operation. MERGE(K_1, K_2) is implemented as a simple recursive algorithm that at every step keeps one of the (common) nodes and then merges the corresponding left and right sub-trees (if one of those sub-trees is empty then it just keeps the other sub-tree, we refer to the original work [24] for details). In this case the maintenance of the number of leaves can be performed when returning from the recursive calls: simply recompute the number of leaves as the sum of the number of leaves of their two children. Each of these computations is a constant time operation, and thus they do not change the asymptotic running time of MERGE. We then obtain the following lemma.

▶ **Lemma 3.** *There exists a data structure maintaining a dynamic partition $\mathcal{K}$ of $\{1, \ldots, k\}$ starting from $\mathcal{K} = \{\{1, \ldots, k\}\}$ and answering operations SOME, MERGE and SIZE-SPLIT such that for a sequence of $n = \Omega(k)$ operations*[5] *it answers in total $O(n \log k)$ time.*

Proof. We first note that operations SOME, MERGE and SIZE-SPLIT can be implemented in the same asymptotic (worst-case) running time as operations SEARCH, MERGE and SPLIT in the data structure of Karczmarz [24], respectively. Indeed, as previously discussed, SOME is implemented as one call to SEARCH, MERGE is implemented in the same way as in [24] but taking care of the number-of-leaves counters' updates which does not affect the asymptotic running time, and SIZE-SPLIT is implemented as one call to SELECT (in

[5] This requirement comes from the fact that mergeable dictionaries actually start from an empty collection of sets, however, one can create the singleton sets and merge then in total $O(k \log k)$ time.

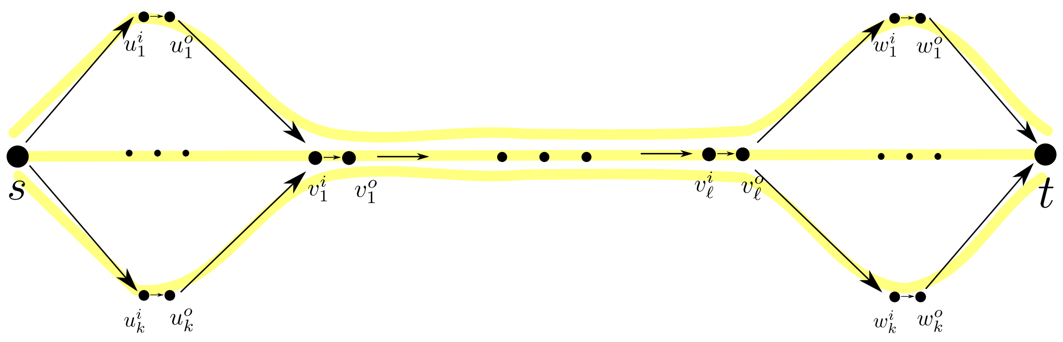

Figure 3 Flow reduction (demands are not shown) of the graph of Figure 1 and a minimum flow on it. Only edges with positive flow are shown. The flow is implicitly presented as a flow decomposition showing every path highlighted in yellow, this flow decomposition corresponds to an MPC of the original DAG.

$O(\log k)$ time) followed by one call to SPLIT (also in $O(\log k)$ time). For the amortized analysis we reuse the potential function used by Karczmarz [24], namely, the number of nodes on all tries. Operations SOME and MERGE follow the same potential change as in [24] and thus have an $O(\log k)$ amortized running time. Finally, since SELECT does not change the total number of nodes (nor any of the tries) the potential change of a SIZE-SPLIT is the same as the one of a SPLIT, that is $O(\log k)$ as in [24]. The lemma follows by using the amortized running times of the data structure's operations. ◀

4 An almost linear time algorithm for MCC

We show how to use Lemma 3 to obtain a fast MCC algorithm. Our algorithm computes a minimum flow f^* encoding an MPC of G and then uses the data structure from Lemma 3 to efficiently extract an MCC from f^*. Next, we describe the well known [30] reduction from MPC to MF by following the notation of [6, Section 2.3 (full version)].

Given the DAG G we build its *flow reduction* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ as the graph obtained by adding a global source s, a global sink t and splitting every vertex $v \in V$ into two copies connected by an edge. Additionally, the first copy, v^i, is connected from the in-neighbors of v and the second copy, v^o, is connected to the out-neighbors of v. Formally, $\mathcal{V} = \{s, t\} \cup \{v^i \mid v \in V\} \cup \{v^o \mid v \in V\}$, and $\mathcal{E} = \{(s, v^i) \mid v \in V\} \cup \{(v^o, t) \mid v \in V\} \cup \{(v^i, v^o) \mid v \in V\} \cup \{(u^o, v^i) \mid (u, v) \in E\}$. Note that $|\mathcal{E}| = O(|V| + |E|)$, and that $\mathcal{G}$ is also a DAG. We also define demands on the edges, $d : \mathcal{E} :\rightarrow \mathbb{N}_0$, as 1 if the edge is of the form (v^i, v^o), and 0 otherwise. Intuitively, the demands *require* that at least one unit of flow goes through every vertex (of G), which directly translates into to the path cover condition of covering each vertex with at least one path. In fact, every flow decomposition (recall Section 2) of a feasible st-flow f of $\mathcal{G}, d$ corresponds to a path cover of G of size $|f|$. Moreover, every decomposition of a minimum flow f^* of $\mathcal{G}, d$ corresponds to an MPC of G, and thus $k = |f^*|$ [6, Section 2.3 (full version)]. Figure 3 illustrates these ideas with an example.

Since every vertex cannot belong to more than $|V|$ paths in an MPC, the problem can be reduced to maximum flow [2, Theorem 3.9.1]. We summarize these results in the following lemma.

Algorithm 1 Non-optimized pseudocode for our MCC algorithm. A naive implementation of this algorithm obtains an $O(||\mathcal{P}||)$ running time as explained in this manuscript.

```
Input: A directed acyclic graph $G = (V, E)$.
Output: A minimum chain decomposition $\mathcal{C} = C_1, \dots, C_k$ of $G$.
1  $(\mathcal{G}, d) \leftarrow$ Build the flow reduction of $G$ detailed in Section 4
2  $f^*, k = |f^*| \leftarrow$ Use Lemma 4 to obtain a minimum flow of $(\mathcal{G}, d)$
3  Initialize $C_i$ as an empty list for $i \in \{1, \dots, k\}$
4  $I_s \leftarrow \{1, \dots, k\}$
5  for $v \in V$ in topological order do
6      $I_v \leftarrow$ Take $f^*(s, v^i)$ elements from $I_s$
7      for $u^o \in N^-(v^i)$ do
8          $I_{uv} \leftarrow$ Take $f^*(u^o, v^i)$ elements from $I_u$
9          $I_v \leftarrow I_v \cup I_{uv}$
10     $i \leftarrow$ Choose an element from $I_v$
11     $C_i$.append($v$)
12 return $C_1, \dots, C_k$
```

▶ **Lemma 4** (Adaptation of [6, Theorem 2.2 (full version)]). *We can compute a flow f^* of $\mathcal{G}, d$ such that every flow decomposition of f^* corresponds to an MPC of G, in time $O(T_{MF}(|E|))$, where $T_{MF}(|E|)$ is the time for solving maximum flow.*

A decomposition algorithm is simple in this case: start from s and follow a path P of positive flow edges until arriving at t, and then update f^* decreasing the flow on the edges of P by one. Repeat this process until no flow remains. The st-paths obtained during the decomposition can be easily transformed into an MPC $\mathcal{P}$ of G (trim s and t and replace v^i, v^o by v on each path, see e.g. [27]).

If implemented carefully (see e.g. [26, Lemma 1.11 (full version)]), the previous algorithm runs in time $O(||\mathcal{P}||)$ and it outputs a valid MCC $\mathcal{P}$ (recall that every MPC is an MCC). However, as shown in Figure 1, $||\mathcal{P}||$ can be $\Omega(k|V|)$ in the worst case. Our algorithm overcomes this barrier by instead directly extracting (from f^*) a minimum chain decomposition (MCD, recall Section 2) and thus its total length is exactly $|V|$.

The main idea to extract an MCD $\mathcal{C} = C_1, \dots, C_k$ from f^* is to compute, for each vertex $v \in \mathcal{V}$, the set $I_v \subseteq \{1, \dots, k\}$ of indices such that v would belong to paths $\mathcal{P}_v = \{P_i \mid i \in I_v\}$ in a flow decomposition $\mathcal{D} = P_1, \dots, P_k$ of f^*. Note that $I_s = I_t = \{1, \dots, k\}$ by construction of $\mathcal{G}, d$. To efficiently compute these sets, we process the vertices in a topological order of $\mathcal{G}$, for example $s, v_1^i, v_1^o, \dots, v_{|V|}^i, v_{|V|}^o, t$ (recall that a topological order $v_1, \dots, v_{|V|}$ of G is assumed as input). When processing vertex v we compute I_v as follows: for every $u \in N^-(v)$ we *take* (exactly) $f^*(u, v)$ elements from I_u, let us denote I_{uv} to these elements. Then, we compute I_v as the *union* $\bigcup_{u \in N^-(v)} I_{uv}$. And finally, we *take an arbitrary element* $i \in I_v$ and append v to C_i. Algorithm 1 shows a pseudocode for this algorithm.

Note that vertices are added to their respective chains in the correct order since they are processed in topological order.

Moreover, since indices are moved from one vertex to the other only if there is an edge between them, consecutive vertices are always connected by a path in G, and thus the lists C_i correspond to proper chains. Finally, adding each vertex to only one such chain ensures that the end result is indeed an MCD (every vertex in exactly one chain).

An important aspect of the algorithm is that exactly $f^*(u^i, v^i)$ ($f^*(s, v^i)$) elements are *taken out* of I_u (I_s). This step is always possible thanks to flow conservation of f^*.

Algorithm 2 Our MCC algorithm running in time $O(T_{MF}(|E|) + (|V| + |E|) \log k)$, where $T_{MF}(|E|)$ is the time for solving maximum flow. The algorithm uses the boosted mergeable dictionaries from Section 3.

```
   Input: A directed acyclic graph $G = (V, E)$.
   Output: A minimum chain decomposition $\mathcal{C} = C_1, \dots, C_k$ of $G$.
 1 $(\mathcal{G}, d) \leftarrow$ Build the *flow reduction* of $G$ detailed in Section 4
 2 $f^*, k = |f^*| \leftarrow$ Use Lemma 4 to obtain a minimum flow of $(\mathcal{G}, d)$
 3 Initialize $C_i$ as an empty list for $i \in \{1, \dots, k\}$
 4 Initialize mergeable dictionaries maintaining a partition of $\{1, \dots, k\}$
 5 $I_s \leftarrow \{1, \dots, k\}$
 6 for $v \in V$ *in topological order* do
 7     $I_v, I_s \leftarrow$ SIZE-SPLIT$(I_s, f^*(s, v^i))$
 8     for $u^o \in N^-(v^i)$ do
 9         $I_{uv}, I_u \leftarrow$ SIZE-SPLIT$(I_u, f^*(u^o, v^i))$
10         $I_v \leftarrow$ MERGE$(I_v, I_{uv})$
11     $i \leftarrow$ SOME$(I_v)$
12     $C_i$.append$(v)$
13 return $C_1, \dots, C_k$
```

▶ **Lemma 5.** *Given a k-width DAG $G = (V, E)$ as input, Algorithm 1 computes a minimum chain decomposition $\mathcal{C} = C_1, \dots, C_k$ of G.*

Proof. By Lemma 4, every flow decomposition of f^* corresponds to an MPC of G. We prove that each C_i is a chain of V, since every vertex is added to exactly one C_i we conclude that $\mathcal{C} = C_1, \dots, C_k$ is a minimum chain decomposition. Inductively, if v is added after v' in chain C_i, then v' reaches v in G. Indeed, $i \in I_v$ and in particular $i \in I_u$ for some $u \in N^-(v)$, and inductively v' reaches u in G and thus also reaches v. ◀

Moreover, since only splits and unions are performed, the sets I_v's and I_{uv}'s form a partition of $\{1, \dots, k\}$ at any point during the algorithm's execution.

A simple implementation of sets I_v's and I_{uv}'s as linked lists, allows us to perform the unions and element picks in constant time, but the splits in $O(f^*(u^o, v^i))$ (and $O(f^*(s, v^i))$) time each, and thus in $O\left(\sum_{v \in V} \left(f^*(s, v^i) + f^*(v^o, t)\right) + \sum_{(u,v) \in E} f^*(u^o, v^i)\right) = O(||\mathcal{P}||)$ time in total. However, we can implement the sets I_v's and I_{uv}'s as boosted mergeable dictionaries from Section 3 to speed up the total running time. Algorithm 2 shows the final result.

▶ **Theorem 1.** *Given a k-width DAG $G = (V, E)$ we can compute a minimum chain cover in time $O(T_{MF}(|E|) + (|V| + |E|) \log k)$, where $T_{MF}(|E|)$ is the time for solving maximum flow.*

Proof. The correctness of the algorithm follows from the previous discussion and Lemma 5 since Algorithm 2 is an implementation of Algorithm 1. Building the flow reduction takes $O(|V| + |E|)$ time and obtaining the minimum flow f^* takes $O(T_{MF}(|E|))$ time by Lemma 4. The rest of the running time is derived from the calls to the mergeable dictionaries' operations. SOME is called $O(|V|)$ times while SIZE-SPLIT and MERGE are called once per edge in the flow reduction that is $O(|\mathcal{E}|) = O(|V| + |E|)$ times. By applying Lemma 3 the total time of the $O(|V| + |E|)$ operation calls is $O((|V| + |E|) \log k)$. ◀

We finish our paper by encapsulating our result into a tool that can be used to efficiently extract a set of vertex-disjoint chains $\mathcal{C}$, encoding a set of paths $\mathcal{P}$, from a flow f that encodes $\mathcal{P}$ as a flow decomposition. If the problem can be modeled as a maximum flow/minimum cost flow problem, the result of Chen et al. [9] allows us to solve such problems in almost linear time. As a simple example, we could solve the ℓ-cover problem (find a set of ℓ vertex-disjoint chains covering the most vertices) in almost linear time.

▶ **Corollary 6.** *Let $G = (V, E)$ be a DAG, and $f : E \to \mathbb{N}_0$ a flow encoding a set of $|f|$ paths $\mathcal{P}$ of G as a flow decomposition into weight-1 paths. In $O((|V| + |E|) \log |f|)$ time, we can compute a set of $|f|$ vertex-disjoint chains $\mathcal{C}$ of G, which can (alternatively) be obtained by removing repeated vertices from $\mathcal{P}$.*

References

1 Amitabha Bagchi, Adam L Buchsbaum, and Michael T Goodrich. Biased skip lists. *Algorithmica*, 42:31–48, 2005.

2 Jørgen Bang-Jensen and Gregory Z Gutin. *Digraphs: theory, algorithms and applications*. Springer Science & Business Media, 2008.

3 Philip Bille, Mikko Berggren Ettienne, Travis Gagie, Inge Li Gørtz, and Nicola Prezza. Decompressing Lempel-Ziv compressed text. In *Proceedings of the 30th Data Compression Conference (DCC 2020)*, pages 143–152. IEEE, 2020.

4 Manuel Cáceres, Massimo Cairo, Andreas Grigorjew, Shahbaz Khan, Brendan Mumey, Romeo Rizzi, Alexandru I Tomescu, and Lucia Williams. Width helps and hinders splitting flows. In *Proceedings of the 30th Annual European Symposium on Algorithms (ESA 2022)*, pages 31:1–31:14, 2022.

5 Manuel Cáceres, Massimo Cairo, Brendan Mumey, Romeo Rizzi, and Alexandru I Tomescu. A linear-time parameterized algorithm for computing the width of a DAG. In *Proceedings of the 47th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2021)*, pages 257–269. Springer, 2021.

6 Manuel Cáceres, Massimo Cairo, Brendan Mumey, Romeo Rizzi, and Alexandru I Tomescu. Sparsifying, shrinking and splicing for minimum path cover in parameterized linear time. In *Proceedings of the 33rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2022)*, pages 359–376. SIAM, 2022.

7 Manuel Cáceres, Brendan Mumey, Edin Husić, Romeo Rizzi, Massimo Cairo, Kristoffer Sahlin, and Alexandru I Tomescu. Safety in multi-assembly via paths appearing in all path covers of a DAG. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 19(6):3673–3684, 2021.

8 Ghanshyam Chandra and Chirag Jain. Sequence to graph alignment using gap-sensitive co-linear chaining. In *Proceedings of the 27th Annual International Conference on Research in Computational Molecular Biology (RECOMB 2023)*, pages 58–73. Springer, 2023.

9 Li Chen, Rasmus Kyng, Yang P Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *Proceedings of the 63rd IEEE Annual Symposium on Foundations of Computer Science (FOCS 2022)*, pages 612–623. IEEE, 2022.

10 Yangjun Chen and Yibin Chen. An efficient algorithm for answering graph reachability queries. In *Proceedings of the 24th International Conference on Data Engineering (ICDE 2008)*, pages 893–902. IEEE, 2008.

11 Yangjun Chen and Yibin Chen. On the graph decomposition. In *Proceedings of the 4th IEEE Fourth International Conference on Big Data and Cloud Computing (BDCloud 2014)*, pages 777–784. IEEE, 2014.

12 Eleonor Ciurea and Laura Ciupala. Sequential and parallel algorithms for minimum flows. *Journal of Applied Mathematics and Computing*, 15(1):53–75, 2004.

13 Nicola Cotumaccio and Nicola Prezza. On indexing and compressing finite automata. In *Proceedings of the 32nd ACM-SIAM Symposium on Discrete Algorithms (SODA 2021)*, pages 2585–2599. SIAM, 2021.

14 Rene De La Briandais. File searching using variable length keys. In *Papers presented at the the March 3-5, 1959, western joint computer conference*, pages 295–298, 1959.

15 Robert P Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51(1):161–166, 1950.

16 Martin Farach and Mikkel Thorup. String matching in Lempel—Ziv compressed strings. *Algorithmica*, 20(4):388–404, 1998.

17 Stefan Felsner, Vijay Raghavan, and Jeremy Spinrad. Recognition algorithms for orders of small width and graphs of small Dilworth number. *Order*, 20(4):351–364, 2003.

18 Delbert R Fulkerson. Note on Dilworth's decomposition theorem for partially ordered sets. *Proceedings of the American Mathematical Society*, 7(4):701–702, 1956.

19 Loukas Georgiadis, Haim Kaplan, Nira Shafrir, Robert E Tarjan, and Renato F Werneck. Data structures for mergeable trees. *ACM Transactions on Algorithms*, 7(2):1–30, 2011.

20 Leo J Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (FOCS 1978)*, pages 8–21. IEEE, 1978.

21 John E Hopcroft and Richard M Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

22 John Iacono and Özgür Özkan. Mergeable dictionaries. In *Proceedings of the 37th International Colloquium on Automata, Languages, and Programming (ICALP 2010)*, pages 164–175. Springer, 2010.

23 Arthur B Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

24 Adam Karczmarz. A simple mergeable dictionary. In *Proceedings of the 15th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2016)*, pages 7:1–7:13, 2016.

25 Donald E Knuth. *The art of computer programming: Volume 3: Sorting and Searching*. Addison-Wesley Professional, 1998.

26 Shimon Kogan and Merav Parter. Beating matrix multiplication for $n^{1/3}$-directed shortcuts. In *Proceedings of the 49th International Colloquium on Automata, Languages, and Programming (ICALP 2022)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. Full version available at `https://www.weizmann.ac.il/math/parter/sites/math.parter/files/uploads/main-lipics-full-version_3.pdf`.

27 Shimon Kogan and Merav Parter. Faster and unified algorithms for diameter reducing shortcuts and minimum chain covers. In *Proceedings of the 34th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2023)*, pages 212–239. SIAM, 2023.

28 Katherine Jane Lai. Complexity of union-split-find problems. Master's thesis, Massachusetts Institute of Technology, 2008.

29 Veli Mäkinen, Alexandru I Tomescu, Anna Kuosmanen, Topi Paavilainen, Travis Gagie, and Rayan Chikhi. Sparse Dynamic Programming on DAGs with Small Width. *ACM Transactions on Algorithms*, 15(2):1–21, 2019.

30 Simeon C Ntafos and S Louis Hakimi. On path cover problems in digraphs and applications to program testing. *IEEE Transactions on Software Engineering*, 5(5):520–529, 1979.

31 Simon J Puglisi and Massimiliano Rossi. On Lempel-Ziv decompression in small space. In *Proceedings of the 29th Data Compression Conference (DCC 2019)*, pages 221–230. IEEE, 2019.

32 Nicola Rizzo, Manuel Caceres, and Veli Mäkinen. Chaining of maximal exact matches in graphs. *arXiv preprint*, 2023. `arXiv:2302.01748`.

33 Robert E Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–185, 1976.

34 Cole Trapnell, Brian A Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J Van Baren, Steven L Salzberg, Barbara J Wold, and Lior Pachter. Transcript assembly and quantification by RNA-Seq reveals unannotated transcripts and isoform switching during cell differentiation. *Nature Biotechnology*, 28(5):511, 2010.

35 Jan van den Brand, Yin Tat Lee, Yang P Liu, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Minimum cost flows, MDPs, and ℓ_1-regression in nearly linear time for dense instances. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing (STOC 2021)*, pages 859–869, 2021.